Algoritmi e Strutture Dati

appunti del corso

9 febbraio 2005

Indice

1 Algoritmi di ordinamento **5**
- 1.1 Insertion Sort . . . 6
  - 1.1.1 Correttezza . . . 7
  - 1.1.2 Complessità . . . 7
  - 1.1.3 Domande . . . 8
- 1.2 Selection Sort . . . 8
- 1.3 Bubble Sort . . . 10
- 1.4 Merge Sort . . . 10
- 1.5 Heap . . . 14
  - 1.5.1 Rappresentazione di una Heap . . . 14
  - 1.5.2 Procedure su Heap: Heapify . . . 16
  - 1.5.3 Procedure su Heap: Build Heap . . . 18
  - 1.5.4 Heap Sort . . . 21
- 1.6 Quick Sort . . . 22
  - 1.6.1 Complessità . . . 23
- 1.7 Lower Bound al problema dell'ordinamento. . . . 27

2 Algoritmi lineari **29**
- 2.1 Counting Sort . . . 29
- 2.2 Radix Sort . . . 30
  - 2.2.1 Correttezza . . . 31
- 2.3 Bucket Sort . . . 32
  - 2.3.1 Correttezza . . . 34
- 2.4 Selection Problem . . . 34
  - 2.4.1 Complessità . . . 35
- 2.5 Order Statistics . . . 36

3 Esercizi (Ordinamento) **39**

4 Strutture Dati **51**
- 4.1 Stacks (pile) . . . 52
- 4.2 Queues (code) . . . 52
- 4.3 Linked List . . . 53

4.4 Hash Table . . . 56
4.4.1 Chaining . . . 56
4.4.2 Open addressing . . . 58
4.5 Binary Search Trees . . . 61
4.5.1 In order tree walk . . . 62
4.5.2 Operazioni . . . 63
4.6 Red Black Trees . . . 67
4.6.1 Proprietà . . . 67
4.6.2 Operazioni . . . 70
4.6.3 Applicazioni: Join . . . 77
4.7 B-Tree . . . 77
4.7.1 Operazioni . . . 79
4.8 Esercizi . . . 86

5 Union Find **93**
5.1 Operazioni su insiemi disgiunti . . . 93
5.1.1 Una applicazione . . . 93
5.1.2 Rappresentazione di insiemi disgiunti . . . 94
5.2 Esercizi . . . 99

6 Algoritmi su grafi **101**
6.1 Breadth-first search . . . 102
6.1.1 Breadth-first tree . . . 106
6.2 Depth-first search . . . 107
6.2.1 Complessità . . . 107
6.2.2 Classificazione degli archi . . . 110
6.2.3 Topological sort . . . 110
6.2.4 Componenti fortemente connesse . . . 111
6.3 Minimum spanning tree . . . 114
6.3.1 Algoritmo generico per MST . . . 114
6.3.2 Algoritmo di Kruskal . . . 116
6.3.3 Algoritmo di Prim . . . 117
6.4 Single source shortest path problem . . . 118
6.4.1 Dijkstra . . . 119
6.4.2 Bellman-Ford . . . 124
6.5 All pairs shortest path problem . . . 125
6.5.1 Notazioni . . . 125
6.5.2 Soluzione generica . . . 126
6.5.3 Algoritmo di Floyd Warshall . . . 129
6.5.4 Algoritmo di Johnson . . . 130
6.6 Esercizi . . . 134

7 Raccolta Esercizi D'Esame **143**

Capitolo 1

Algoritmi di ordinamento

Un generico algoritmo di ordinamento è caratterizzato da:

Input: Una sequenza di n numeri $a_1, a_2, \ldots, a_n$;

Output: Una permutazione π degli n numeri tale che:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq, \ldots, \leq a_{\pi(n)}$$

Possiamo distinguere tra gli algoritmi che ordinano *in place* e quelli no. Un algoritmo ordina in place solo se in ogni istante c'è al più un numero costante di elementi memorizzati al di fuori dello spazio assegnato per l'input.

Scriveremo gli algoritmi in *pseudo codice*. Lo scopo di questultimo è solo quello di permettere una corretta e più semplice analisi della complessità, principale obiettivo del corso. Tale rappresentazione ci permette infatti di tralasciare molti dettagli, migliorando quindi la leggibilità dell'algoritmo in esame. Esistono tuttavia particolari linguaggi per la rapida prototipazione che sono molto aderenti al formalismo che noi adottiamo.

Ogni algoritmo che analizzeremo sarà valutato in termini di *correttezza* e *complessità* considerando anche le strutture dati che vengono impiegate. La correttezza indicherà se l'algoritmo riesce ad assolvere al compito per il quale è stato pensato, in particolare valuteremo la terminazione dell'algoritmo; la complessità può essere riferita a diverse risorse, riusabili e non, ad esempio l'utilizzo della memoria, il tempo di CPU,... noi analizzeremo principalmente la complessità in termini di tempo di esecuzione.

Tratteremo la complessità in termini asintotici, assegnando costo unitario alle istruzioni elementari e ignorando i problemi relativi all'architettura sottostante.

Tratteremo sempre l'ordinamento di n numeri interi qualsiasi. L'ipotesi non è restrittiva, infatti ordinare strutture più complesse (es: record) fa variare la complessità solo di un fattore costante. Indichiamo con codice *tight* (stretto) quello che contiene tutte e sole operazioni necessarie (es: insertion sort) e con codice *non tight* il rimanente. Con la dicitura *in place*

indichiamo il codice che usa solo la spazio assegnatogli per l'input, più un valore costante.

1.1 Insertion Sort

Il principio con il quale funziona è lo stesso con cui si ordina una mano di carte.

Algoritmo 1.1 Insertion Sort(A)

Insertion Sort(A)

```
1: for j ← 2 to length(A) do
2:     key ← A[j]
3:     i ← j − 1
4:     while (i > 0) and (key < A[i]) do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

Al fine di determinarne la complessità associamo all'i-esima riga di programma una costante c_i che indica il costo di quella istruzione. Il costo effettivo di esecuzione di insertion sort è quindi uguale alla somma delle costanti di riga per il numero di volte che quella riga di codice viene eseguita.

Mentre il numero di iterazioni del ciclo for è ben determinato fin dall'inizio quelle del ciclo while variano in relazione al valore $A[j]$. Per risolvere il problema indichiamo questo numero con $t_j \in [0, j]$ ottenendo:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \tag{1.1}$$

Guardando la formula proviamo un certo fastidio dovuto alla presenza di numerose costanti. Quello che facciamo è di uniformarle alla massima coinvolta ottenendo così un'analisi indipendente dalle stesse. Formalmente questo è giustificato dall'analisi asintotica della complessità. La complessità ottenuta risulta dipendere dall'input e in base a come questo si presenta potremmo avere un caso *migliore*, uno *medio* e uno *pessimo*.

Il caso migliore risulta poco significativo: difficilmente si è fortunati, di conseguenza il non determinismo introdotto da questa ipotesi risulta poco appetibile per una trattazione matematica approfondita. Sicuramente più interessante è il caso peggiore in quanto pone una limitazione superiore a qualsiasi esecuzione.

L'ottimo sarebbe lo studio del caso medio ma gli strumenti matematici necessari risultano particolarmente difficili da usare.

Dato n numero di elementi in input, si ottiene che Insertion Sort ha la seguente complessità:

- $O(n)$: lineare nel caso migliore in quanto il ciclo while non viene mai eseguito;
- $O\big(n^2\big)$: quadratica nel caso medio.
- $O\big(n^2\big)$: quadratica nel caso peggiore. Eseguiamo infatti n iterazioni del ciclo for e n iterazioni del ciclo while

1.1.1 Correttezza

Dobbiamo dimostrare che dato in input ad Insertion Sort un array contenente n interi qualsiasi, $A[1, \ldots, n]$, l'algoritmo modifica le posizioni degli interi in modo da ordinarli all'interno di A.

Sfrutteremo l'induzione sul numero di elementi dell'array ricordando che nel caso in cui l'algoritmo sia puramente iterativo ignorando l'iterazione n-esima dei cicli otteniamo la versione dell'algoritmo alla quale possiamo applicare l'ipotesi induttiva. Sia n la coardinalità dell'array.

$n = 1$ L'algoritmo non fa nulla ed è corretto. Per definizione un array di un solo elemento è ordinato.

$n > 1$ Dall'ipotesi induttiva abbiamo che dopo le prime $n - 1$ iterazioni del ciclo più esterno, $A[1, \ldots, n-1]$ contiene una lista ordinata. Durante l'n-esima iterazione il ciclo while termina solo quando la condizione non è più soddisfatta, cioè quando il valore assegnato a *key* è inferiore al valore contenuto in $A[j]$ per $j > i + 1$. Quindi $A[n] = key$ viene inserito nella posizione corretta.

1.1.2 Complessità

Solitamente la complessità è indicata con la funzione $T(n)$, essa rappresenta il tempo impiegato da un qualsiasi algoritmo su input generico di dimensione n per portare a termine il suo compito.

Da quanto detto in precedenza noi indicheremo con $T(n)$ il tempo di esecuzione di un'algoritmo sull'input peggiore.

Cerchiamo ora di determinare la funzione che indica la complessità di Insertion Sort, rifacendoci all'espressione 1.1 considerando però il caso peggiore, ovvero sostituendo a t_j l'indice j stesso:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

Sappiamo che:

$$\begin{aligned}\sum_{j=0}^{n} j &= \frac{n(n+1)}{2} \\ &= \frac{n+n^2}{2}\end{aligned}$$

Chiameremo questo termine il *termine dominante* per via dell'esponente. Otteniamo quindi che per opportuni coefficienti a, b, c, a':

$$\begin{aligned}T(n) &\leq an^2 + bn + c \\ &\leq a'n^2\end{aligned}$$

Quindi l'equazione di complessità risulta essere dominata da una funzione di complessità quadratica.

1.1.3 Domande

1. Nel caso migliore possibile, Insertion Sort è ancora quadratico?

   La risposta è no. Si dimostra che nel caso migliore la complessità è: $T'(n) \leq \overline{b}n$

2. Si può fare di meglio?

   La risposta è si, come lo vedremo più avanti.

1.2 Selection Sort

La complessità è $O(n^2)$ causa i due cicli innestati.

ESERCIZI:

1. si può evitare l'utilizzo di B;
2. dimostrare la correttezza di Selection Sort;
3. Calcolare la complessità nel caso migliore e nel caso peggiore. Verificare che entrambe le complessità sono quadratiche.

Osservazioni:

Nel calcolo dell'equazione di complessità per quanto riguarda il costrutto *if-then-else* considereremo il peggiore dei due rami, a meno che vi siano motivi fondati di ritenere che l'esecuzione privilegierà una delle due alternative.

◇

Algoritmo 1.2 SelectionSort(A)

SelectionSort(A)

```
1: curlength ← length(A)
2: for j ← 1 to lenght(A) do
3:    curmin ← A[1]
4:    for i ← 2 to curlength do
5:       if curmin < A[i] then
6:          A[i − 1] ← A[i]
7:       else
8:          A[i − 1] ← curmin
9:          curmin ← A[i]
10:      end if
11:   end for
12:   B[j] ← curmin
13:   curlength ← curlength - 1
14: end for
```

Algoritmo 1.3 Bubble Sort(A)

Bubble Sort(A)

```
1: for j ← length(A) downto 2 do
2:    for i ← 2 to j do
3:       if A[i − 1] < A[i] then
4:          swap(A[i − 1], A[i])
5:       end if
6:    end for
7: end for
```

1.3 Bubble Sort

ESERCIZI:

1. Calcolare la complessità ($O(n^2)$ in tutti i casi);
2. Dimostrare la correttezza.

1.4 Merge Sort

Algoritmo 1.4 MergeSort(A, p, r)

MergeSort(A, p, r**)**

```
1: if p < r then
2:     q ← ⌊(p+r)/2⌋
3:     MergeSort(A, p, q)
4:     MergeSort(A, q + 1, r)
5:     Merge(A, p, q, r)
6: end if
```

Leggendo attentamente il codice sorgente 1.4 si può dire che i parametri p ed r servono a limitare l'azione della chiamata ricorsiva. Rimane da chiarire come operi Merge(A, p, q, r).

Non è in-place. Determinare un algoritmo che lo sia risulta al quanto complesso.

Questo algoritmo usa un approccio tipico degli algoritmi ricorsivi definito *divide-and-conquer*.

Complessità

Cerchiamo di esplicitare la complessità in termini semplici, disegnabili. Facciamo riferimento all'albero delle chiamate ricorsive.

Tutti i rami hanno la stessa profondità?

Osservazioni:

1. Se n è una potenza di 2, l'albero di ricorsione è *completo*. Assumeremo sempre di lavorare con potenze di due (la complessità generale differirà soltanto per una costante);
2. Cosa succede alle chiamate che corrispondono alle foglie?
3. Il numero di foglie è più o meno la metà dei nodi, infatti 'schiacciando' l'albero a livello delle foglie si ottiene una sequenza del tipo: foglia, nodo interno, foglia, nodo interno, ...

Algoritmo 1.5 Merge(U, x, y, z) procedura di Merge Sort

Merge(U, x, y, z)

```
1: k ← i ← x
2: j ← y + 1
3: while (i ≤ y) ∨ (j ≤ z) do
4:    if i > y then
5:       V[k] ← U[j]
6:       j ← j + 1
7:    else
8:       if j > z then
9:          V[k] ← U[i]
10:         i ← i + 1
11:      else
12:         if U[i] < U[j] then
13:            V[k] ← U[i]
14:            i ← i + 1
15:         else
16:            V[k] ← U[j]
17:            j ← j + 1
18:         end if
19:      end if
20:   end if
21:   k ← k + 1
22: end while
23: U[x, ..., z] ← V[x, ..., z]
```

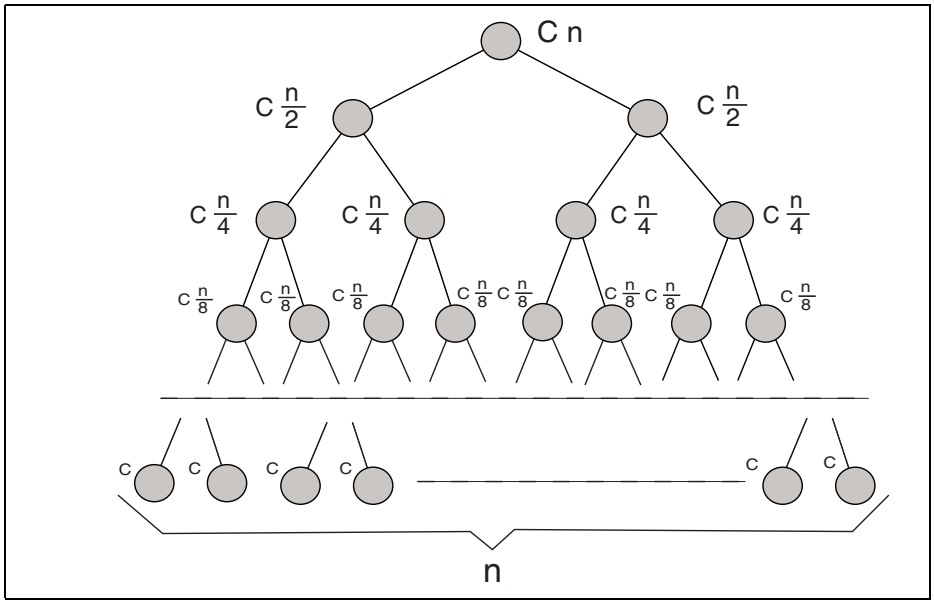

Figura 1.1: Albero delle chiamate ricorsive.

◇

Quando definiamo un'equazione di complessità ricorsiva, dobbiamo specificare anche il caso base:

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + cn & \text{se } n > 1 \\ c & \text{se } n = 1 \end{cases}$$

L'albero delle chiamate ricorsive ha n foglie quindi l'altezza dell'albero è $\log(n)$. Assunto $n = 2^k$, abbiamo che l'altezza dell'albero delle chiamate ricorsive associata a Merge Sort su un input di dimensione n è $k = \log(n)$.

Ad ogni livello eseguiamo cn operazioni, quindi globalmente eseguiamo $cn\log(n)$ operazioni. Riassumendo otteniamo $T(n) = cn\log(n)$.

Vediamo come risolvere il problema in modo più formale. Assumiamo per ipotesi che $n = 2^k$ o equivalentemente che $\log(n) = k$ e cerchiamo di risolvere l'equazione precedente. Se avessimo un'equazione del tipo:

$$T(n) = \begin{cases} T(n-1) + c & \text{se } n > 1 \\ c & \text{se } n = 1 \end{cases}$$

sarebbe più semplice e avremmo che:

$$T(n) = \underbrace{c + c + \ldots + c}_{n} = nc$$

Cerchiamo di manipolare l'equazione di partenza per fare in modo che

ammetta *telescoping*[1].

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \text{ sostituendo } n = 2^k \\ T\left(2^k\right) &= 2T\left(\frac{2^k}{2}\right) + c2^k \\ &= 2T\left(2^{k-1}\right) + c2^k \end{aligned}$$

Poniamo ora $G(k) = T\left(2^k\right)$, sostituendo otteniamo:

$$\begin{aligned} G(k) &= 2G(k-1) + c2^k \\ \frac{G(k)}{2^k} &= \frac{2G(k-1)}{2^k} + c\frac{2^k}{2^k} \\ &= \frac{G(k-1)}{2^{k-1}} + c \end{aligned}$$

Poniamo ora $H(k) = G(k)/2^k$, otteniamo:

$$\begin{aligned} H(k) &= H(k-1) + c \quad \text{ammette telescoping} \\ &= ck \end{aligned}$$

Riordinando ed effettuando le opportune sostituzioni abbiamo

$$T(n) = cn\ \log(n)$$

quindi asintoticamente la complessità è

$$O(n \log(n))$$

ESERCIZI:

1. Pensare ad un algoritmo che abbia come equazione di complessità:
$$T(n) = T(n-1) + c$$
2. Risolvere:
   - $T(n) = 3T\left(\frac{n}{2}\right) + cn$;
   - $T(n) = 2T\left(\frac{n}{2}\right) + \log(n)$;

[1] Una somma si dice telescopica se è del tipo:

$$\sum_{i=1}^{n}(a_i - a_{i+1}) = a_1 - a_{n+1}$$

Figura 1.2: Esempio di albero binario completo (a sinistra) e quasi completo (a destra)

1.5 Heap

Quello che analizziamo ora è un algoritmo di ordinamento veloce, asintoticamente quanto Merge Sort (vedi sezione 1.4), che sfrutta una particolare struttura dati, adatta anche all'implementazione di operazioni tipiche delle *code di priorità*, detta appunto *Heap*.

Definizione 1.1 (Heap) *Una heap è un albero binario quasi completo contenente n chiavi ognuna delle quali deve soddisfare la seguente proprietà:*

$$\text{KEY}(x) \geq \text{KEY}(left(x)), \text{KEY}(right(x))$$

Specifichiamo alcuni importanti concetti:

- albero *binario*: ogni nodo ha 2 figli;
- albero *completo*: tutti i livelli sono pieni. Detto in altro modo, le foglie sono tutte allo stesso livello;
- albero *quasi completo*: mancano solo i nodi all'ultimo livello e quelli che mancano, mancano a partire da destra.

In una Heap il massimo si trova alla radice. Tolto il massimo non è banale ripristinare la Heap.

1.5.1 Rappresentazione di una Heap

Una heap può esssere rappresentata in vari modi, noi sceglieremo la rappresentazione tramite array che offre grossi vantaggi per quanto riguarda la pulizia del codice e la facilità d'uso.

Sfruttando l'array disponiamo i nodi della heap scrivendoli in sequenza, come evidenziato in figura 1.5.1. A questo punto possiamo definire delle procedure che permettano facilmente di accedere al figlio destro, sinistro e al genitore del nodo corrente.

Disponendo delle procedure *left*, *right* e *parent* possiamo ridefinire la Heap-property:

Algoritmo 1.6 Parent(i) procedura di Heap sort

Parent(i)

1: return $\lfloor \frac{i}{2} \rfloor$

Algoritmo 1.7 Left(i) procedura di Heap Sort

Left(i)

1: return $2i$

Algoritmo 1.8 Right(i) procedura di Heap Sort

Right(i)

1: return $2i + 1$

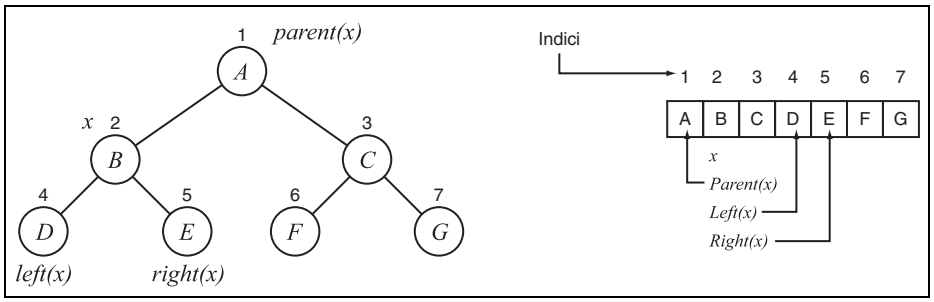

Figura 1.3: Implementazione di heap tramite array ed esempi di funzioni.

Definizione 1.2 (Heap) *Sia A l'array che implementa la heap. Per ogni nodo i della heap diverso dalla radice deve valere la seguente disuguaglianza:*

$$A[parent(i)] \geq A[i]$$

In base al verso delle disuguaglianza parliamo di *max heap*, se è come sopra, oppure di *min heap*, nel senso inverso.

Va ricordato che la rappresentazione qui adottata è completamente statica, una struttura completamente dinamica può essere ottenuta mediante record e puntatori.

Osservazioni:

1. Sarà nostro obiettivo mantenere gli alberi più bassi possibile, a parità di numero di nodi. Infatti più l'albero è basso più è pieno. Le heap sono gli alberi più bassi che si possono ottenere contenenti n elementi, con n non necessariamente pari a 2^{n-1}.
2. Qualunque operazione interessi le foglie diventa automaticamente molto costosa, infatti queste sono circa la metà dei nodi della heap.

$\diamond$

ESERCIZI:

1. Dimostrare che un albero binario completo contenente n nodi ha $\lceil \frac{n}{2} \rceil$ foglie. (suggerimento: per induzione sull'altezza).
2. Dimostrare che una heap contenente n nodi ha $\lceil \frac{n}{2} \rceil$ foglie.

1.5.2 Procedure su Heap: Heapify

Heapify(A, i) è una procedura che prende in input una posizione i in A tale che il sottoalbero radicato nel nodo in posizione i soddisfa la proprietà delle heap ovunque, meno eventualmente che in $A[i]$.

Dati A ed i in input, Heapify restituisce un albero radicato in i tale che è una heap. Heapify non si deve preoccupare della struttura della heap di partenza perché comunque lavora su sottoalberi che sono Heap.

Vediamo ora come agisce Heapify, supponendo $x = A[i]$ ed una chiamata del tipo Heapify(A, i).

- In prima istanza si verifica se la proprietà della Heap non è soddisfatta in x;
- qualora la proprietà tale proprietà risultasse violata in x, viene scambiato x con il più grande dei suoi due figli

- dato che la heap-property è una proprietà locale, fatto lo scambio viene chiamata ricorsivamente Heapify sul figlio sul quale è stato fatto lo scambio.

Il caso peggiore si ha quando il minimo è sulla radice, allora Heapify eseguirà una serie di scambi fino ad arrivare alle foglie, percorrendo tutto l'albero. Il codice è riportato nell'algoritmo 1.9.

Algoritmo 1.9 Heapify(A, i) procedura di Heap Sort

Heapify(A, i)
1: $l \leftarrow left(i)$
2: $r \leftarrow right(i)$
3: **if** $(l \leq heapsize(A)) \wedge (A[l] > A[i])$ **then**
4: $\quad largest \leftarrow l$
5: **else**
6: $\quad largest \leftarrow i$
7: **end if**
8: **if** $(r \leq heapsize(A)) \wedge (A[r] \geq A[largest])$ **then**
9: $\quad largest \leftarrow r$
10: **end if**
11: **if** $largest \neq i$ **then**
12: $\quad swap(A[i], A[largest])$
13: $\quad Heapify(A, largest)$
14: **end if**

Correttezza

Ci preoccuperemo ora di stabilire se Heapify ritorna o meno una Heap. Procediamo per induzione sull'altezza h:

$h = 0$ Un unico nodo è una heap.

$h > 0$ Se la proprietà della heap è verificata in x abbiamo finito e l'algoritmo correttamente non fa alcuna operazione; altrimenti scambiamo x con il più grande dei suoi figli che, senza perdita di generalità, assumiamo essere $left(x)$.

Abbiamo che $left(x) \geq right(x)$ e $left(x) \geq x$, quindi la proprietà è vera alla radice. La proprietà è vera nel sottoalbero radicato in $right(x)$ per ipotesi (dal momento che non abbiamo modificato nulla). Infine è vera nel sottoalbero radicato nel figlio sinistro della radice per ipotesi induttiva.

Complessità

Per determinare la complessità di Heapify è necessario risolvere la seguente equazione di complessità:

$$T(n) = \begin{cases} T(n') + O(1) & n > 1 \\ O(1) & n = 1 \end{cases}$$

Dove $O(1)$ rappresenta nella prima equazione il costo del lavoro da svolgere diverso dalla chiamata ricorsiva, mentre nella seconda indica la base della ricorsione e n' indica la dimensione del sottoalbero su cui Heapify viene chiamata dopo lo swap, sempre nel caso peggiore.

Il caso peggiore che si possa presentare si ha quando la chiamata ricorsiva viene eseguita su ad esempio $left(x)$ e $T(left(x))$ ha la configurazione indicata nella figura 1.4 b. In questo caso n' è uguale a $\frac{2}{3}n$, ottenendo così che:

$$T(n) = T\left(\frac{2}{3}n\right) + O(1)$$

Considerando $n = \left(\frac{2}{3}\right)^k$, con alcune semplici sostituzioni otteniamo:

$$T(n) = O\Big(\log_{\frac{3}{2}} n\Big) = O(\log n)$$

I due logaritmi infatti differiscono solo per una costante, non influente in termini asintotici.

Osservazioni:

L'ipotesi sotto la quale Heapify lavora è che l'unico punto in cui la proprietà della Heap risulta non essere soddisfatta è la radice.

◇

La procedura Heapify serve in una varietà di casi, il più importante è il caso in cui una volta eliminato il massimo si voglia ricostruire la heap.

1.5.3 Procedure su Heap: Build Heap

Il primo problema che risolviamo è la costruzione di una Heap. Il nostro obiettivo è di ordinare n elementi limitando la complessità a $O(n \log n)$. BuildHeap(A) sarà il nostro algoritmo e avrà complessità $O(n)$.

Vediamo prima lo schema di un altro algoritmo di costruzione di una heap ricorsivo:

- inseriamo gli elementi uno alla volta, partendo dalla heap vuota.
- inseriti i elementi, inseriamo l'$(i+1)$-esimo elemento come figlio destro;

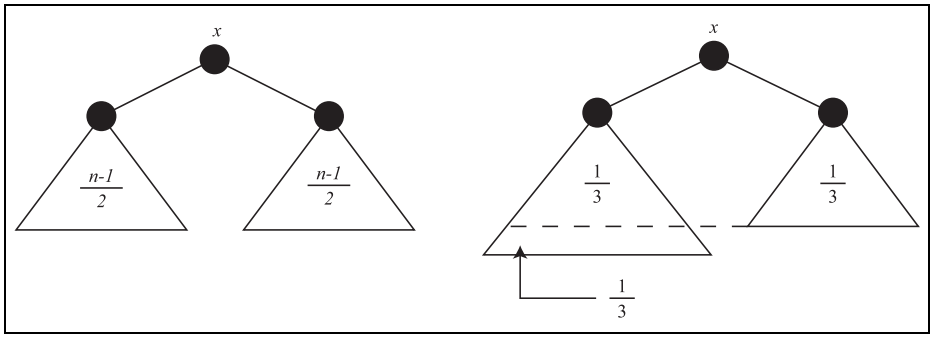

Figura 1.4: Rappresentazione delle chiamate di Heapify: a) un caso generico; b) caso peggiore.

- se la proprietà della Heap non è soddisfatta tra l'ultimo elemento aggiunto e il padre allora li scambiamo e ripetiamo il procedimento finché necessario.

  ESERCIZI:

  1. Scrivere il codice;
  2. Valutare la complessità;
  3. Valutare la correttezza;

Passiamo ora a Build Heap.
L'idea sviluppata nell'algoritmo 1.10 è la seguente:

- partiamo dalle foglie e osserviamo che ognuna costituisce una heap;
- per ogni altro elemento, a partire dall'ultimo a destra, che non sia una foglia eseguiamo una chiamata a Heapify.

Algoritmo 1.10 BuildHeap(A) procedura di Heap Sort

```
BuildHeap(A)
1: heapsize(A) ← length(A)
2: for i ← ⌊length(A)/2⌋ downto 1 do
3:    Heapify(A, i)
4: end for
```

Complessità

Da un'analisi grossolana possiamo notare che il ciclo for viene eseguito $n/2$ volte, con un costo di ogni singola iterazione paria a $O(\log n)$ da quanto

dimostrato in precedenza, dandoci un costo totale di $O(n \log n)$. Cerchiamo ora di raffinare l'analisi.

Se definiamo $f(h)$ come il numero di nodi di altezza h con chiaramente il vincolo $1 \leq h \leq \log n$ la complessità è data dalla formula:

$$\sum_{h=1}^{\log n} f(h)\, O(h) \tag{1.2}$$

Procediamo nell'analisi della complessità stimando qual'è il numero di nodi di altezza h in funzione del numero totale di nodi.

Proposizione 1.5.1 Se T è un albero binario *completo* con n nodi, allora T ha $\lceil n/2^{h+1} \rceil$ nodi di altezza h.

Dimostrazione. Per $h = 0$ abbiamo $\lceil n/2 \rceil$ nodi di altezza 0 (foglie), per $h = \log_2(n+1) - 1$ abbiamo $\lceil n/n+1 \rceil = 1$ nodi di altezza pari alla radice.

Il numero di nodi in un albero binario completo in funzione dell'altezza è $2^{h+1} - 1$, la proprietà quindi deve mostrare il fatto che abbiamo 1 nodo di altezza massima (la radice) e $\lceil n/2 \rceil$ nodi di altezza 0.

Procediamo per induzione sull'altezza h:

$h = 0$: la proprietà è chiaramente verificata, infatti un albero binario completo T ha $\lceil n/2 \rceil$ foglie.

$h > 0$: dall'ipotesi induttiva ci sono $\lceil n/2^{h+1-1} \rceil$ nodi di altezza $h-1$. Quindi abbiamo $\lceil n/2^h \rceil \cdot 1/2$ nodi di altezza h (un nodo di altezza h ogni due nodi di altezza $h-1$).

Otteniamo $\lceil n/2^{h+1} \rceil$ nodi di altezza h.

■

Osservazioni:

1. Se T fosse un albero *quasi completo*, come una heap, allora la proposizione precedente andrebbe riscritta con *... al più* $\lceil \frac{n}{2^{h+1}} \rceil$ *nodi di altezza* h.
2. Se un algoritmo è composto di due o più parti il costo di tutto l'algoritmo sarà quello della parte che costa di più.

◇

Considerando la proposizione precedente l'equazione di complessità definita dall'espressione 1.2 può essere così aggiornata:

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

dimostreremo che la complessità è pari a $O(n)$. Ricordiamo prima che:

$$\sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1 - x}$$

e nel caso in cui $x < 1$ e $n \to \infty$ vale che:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

e

$$\sum_{k=0}^{\infty} kx^k = x \sum_{k=0}^{\infty} kx^{k-1} = \frac{x}{(1 - x)^2}$$

Sfruttando queste uguaglianze con $x = \frac{1}{2}$ otteniamo:

$$\sum_{h=0}^{\log n} \frac{nh}{2^{h+1}} = \frac{n}{2} \sum_{h=0}^{\log n} \frac{h}{2^h}$$

Se estendiamo la sommatoria a infinito, dato che il termine decresce all'aumentare dell'indice h possiamo maggiorarla con n sfuttando i seguenti passaggi:

$$\begin{aligned} \frac{n}{2} \sum_{h=0}^{\log n} \frac{h}{2^h} &\leq \frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &= \frac{n}{2} \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} \\ &= n \end{aligned}$$

1.5.4 Heap Sort

Sfruttiamo ora tutte le procedure viste sulle Heap per realizzare un algoritmo di ordinamento in-place. Analizzando il codice sorgente 1.11 notiamo che il ciclo for viene eseguito n volte. Ogni operazione ha costo costante eccetto Heapify che costa $O(\log n)$. Otteniamo quindi che Heap Sort ha complessità

$$O(n \log n)$$

ESERCIZI:

1. Scrivere HeapSort senza usare gli array;
2. Dimostrare la correttezza di Heap Sort.
3. Immaginiamo di avere una Heap e un puntatore ad un nodo $A[i]$. Scrivere un algoritmo che dato un intero compreso fra 1 e heapsize(A), modifichi il valore di A[i] e gli assegni valore k', restituendo una heap. Verificarne poi correttezza e complessità (deve risultare logaritmica).

Algoritmo 1.11 HeapSort(A)

HeapSort(A)

```
1: BuildHeap(A)
2: for i ← length(A) downto 2 do
3:     swap(A[1], A[i])
4:     heapsize(A) ← heapsize(A) − 1
5:     Heapify(A, 1)
6: end for
```

1.6 Quick Sort

Quick Sort è un altro algoritmo di ordinamento di tipo ricorsivo, molto usato nella pratica in quanto pur avendo una complessità asintotica pari a $O(n^2)$ nel caso peggiore questa si abbassa a $O(n \log n)$ nel caso migliore e nel caso medio.

Altre caratterizzazioni sono l'approccio usato, denominato *divide and conquer*, la semplicità nelle strutture dati usate e il fatto di essere in-place.

Da quanto si evince dall'analisi del codice di 1.12 ci dobbiamo preoccupare che Partition ritorni sempre due sottoarray entrambi non vuoti altrimenti si rischia un loop infinito. Otteniamo che la suddivisione è critica per la complessità. Al fine di verificare la precedente consideriamo i seguenti tre casi:

1. x è minimo in $A[p, \ldots, r]$
2. x è massimo in $A[p, \ldots, r]$
3. non è vero 1) e non è vero 2)

Se siamo nel caso 1. allora i rimane fermo e j scorre fino a i. Nel caso 2. invece i si sposta fino a j. In entrambi i casi una chiamata viene fatta su array di almeno un elemento.

Algoritmo 1.12 QuickSort((A, p, r)

QuickSort((A, p, r)

```
1: if p < r then
2:     q ← Partition(A, p, r)
3:     QuickSort(A, p, q)
4:     QuickSort(A, q + 1, r)
5: end if
```

Algoritmo 1.13 Partition(A, p, r) procedura di Quick Sort

Partition(A, p, r)

```
1: x ← A[p]
2: i ← p − 1
3: j ← r + 1
4: while TRUE do
5:    repeat
6:       j ← j − 1
7:    until A[j] ≤ x
8:    repeat
9:       i ← i + 1
10:   until A[i] ≥ x
11:   if i < j then
12:      swap(A[i], A[j])
13:   else
14:      return j
15:   end if
16: end while
```

1.6.1 Complessità

Caso peggiore

Ogni elemento in $A[p, \ldots, r]$ ha indice tale che risulti uguale ad i oppure a j al più una volta (i cresce ad ogni iterazione e j decresce ad ogni iterazione).

Il lavoro eseguito quando l'indice i o j è quello relativo a un dato elemento è $O(1)$, quindi globalmente eseguiamo $O(n)$ istruzioni di base. Calcoliamo ora l'equazione di complessità.

Sia $q \in \{1, \ldots, n-1\}$:

$$T(n) \leq T(q) + T(n-q) + O(n)$$

La complessità dipende da q e nel caso peggiore sarà:

$$T(n) = \max_{q \in [1,\ldots,n-1]} \{T(q) + T(n-q) + O(n)\}$$

congetturiamo che $T(n) \leq cn^2$ e per ipotesi induttiva avremo:

$$T(n) \leq \max_{q \in [1,\ldots,n-1]} \left\{c\, q^2 + c\,(n-q)^2 + O(n)\right\}$$

consideriamo ora $O(n)$ come kn con k costante fissata e sia $c \geq k$ allora:

$$T(n) \leq c \max_{q \in [1,\ldots,n-1]} \left\{q^2 + (n-q)^2 + n\right\}$$

sviluppando:

$$T(n) \leq c \max_{q \in [1,\ldots,n-1]} \left\{ 2q^2 + n^2 - 2nq + n \right\}$$

che se considerata in q, è l'equazione di una parabola con concavità verso l'alto, il massimo quindi si trova nei due estremi del dominio cioè in $q = 1$ e $q = n - 1$.

$$\begin{aligned} T(n) &\leq c\left(2 + n^2 - 2n + n\right) \\ &= c\left(n^2 - n + 2\right) \\ &\leq cn^2 \end{aligned}$$

La complessità nel caso peggiore è $T(n) \leq O\left(n^2\right)$. Viene spontaneo chiedersi quando si presenti il caso peggiore: esso si ha quando la suddivisione dell'array è fortemente sbilanciata, ovvero quando q è sempre pari a 1. In tal caso avremo infatti:

$$\begin{aligned} T(n) &= T(n-1) + T(1) + O(n) \\ &= O\left(n^2\right) \end{aligned}$$

Caso migliore

Notiamo che il caso migliore si presenta quando q è sempre uguale a $n/2$, infatti in tal caso il comportamento di Quick Sort è simile a quello di MergeSort. In questo caso l'equazione diventa:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\ &= O(n \log n) \end{aligned}$$

Dimostriamo ora che se anche fossimo in grado di dividere l'input prima delle chiamate ricorsive in due parti di dimensione fissa (es: 1/10 e 9/10) otterremo comunque complessità logaritmica.

Verifichiamo questo fatto osservando l'albero di ricorsione in figura 1.6.1. Si nota chiaramente che nel caso peggiore eseguiamo $\log_{\frac{10}{9}} n$ volte un lavoro pari a $O(n)$. Globalmente eseguiamo meno di $O\left(n \log_{\frac{10}{9}} n\right)$ che è uguale a $O(n \log n)$ perché differisce al massimo per una costante. Discorsi analoghi valgono per una qualunque divisione fissa dell'input.

Caso medio

Il caso medio di Quick Sort ha tempi di esecuzione molto vicini a quelli del caso migliore. Prima di tutto conviene chiedersi che distribuzione di probabilità assumiamo per l'input, in questo caso considereremo le n-ple che hanno le stesse relazioni tra gli elementi.

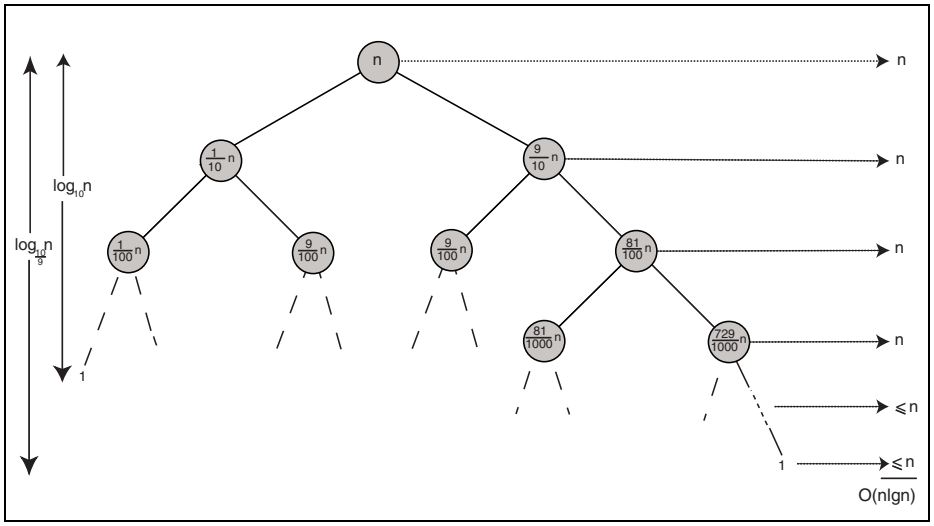

Figura 1.5: Esempio di albero di ricorsione di QuickSort

Definizione 1.3 (Rango) *Si dice* Rango *di x in S il numero di elementi in S che sono minori o uguali di x.*

$$R(x) = |\{a \mid a \in S,\ a \leq x\}|$$

In generale non vi sono garanzie che il pivot scelto da QuickSort (da Partition) sia completamente casuale. Se così fosse potremmo utilizzare una funzione che dato n restituisca un elemento casuale i compreso fra 1 e n e potremmo definire RandomizeQuickSort come QuickSort modificato in modo da scambiare l'i-esimo elemento con il primo, prima della chiamata a Partition. Il valore numerico del pivot è di scarso interesse, per la complessità di QuickSort importa solo il rango. Possiamo assumere che il pivot sia casuale, dobbiamo fare attenzione ai linguaggi che mettono a disposizione funzioni pseudo-casuali.

La probabilità che un dato pivot sia quello utilizzato da Partition (cioè che il suo rango sia $i \in \{1, \ldots, n\}$) è $1/n$; assumiamo cioè una distribuzione uniforme.

La complessità di Quick Sort con un pivot di rango i sarà:

$$T(i) + T(n-i) + O(n)$$

L'equazione di complessità nel caso medio:

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n)$$

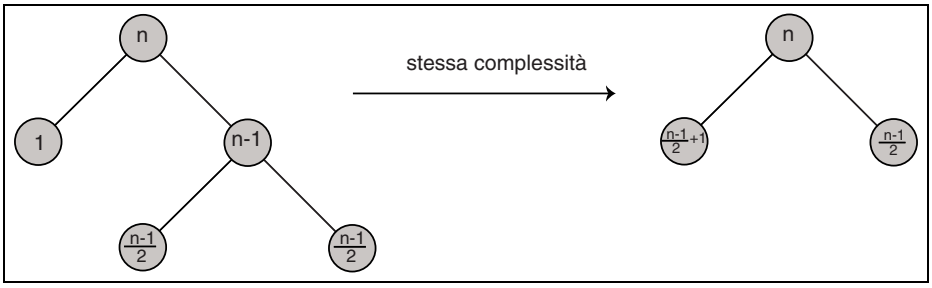

Figura 1.6: Altro esempio di albero di ricorsione.

la possiamo riassumere come

$$T(n) \leq \Theta(n) + \frac{2}{n}\sum_{k=1}^{n-1} T(k)$$

Dimostriamo induttivamente che $T(n)$ è $\Theta(n \log n)$ ovvero $T(n) \leq an \log n + b$. Assumiamo questo vero per $k < n$ e lo dimostriamo per induzione su n:

- caso base: ovvio.
- passo induttivo:

$$\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{k=1}^{n-1}(ak\log k + b) + \Theta(n) \\
&= \frac{2a}{n}\underbrace{\sum_{k=1}^{n-1} k\log k}_{\leq \frac{1}{2}n^2\log n - \frac{1}{8}n^2} + \frac{2b}{n}(n-1) + \Theta(n) \quad \text{verifichiamo poi il passaggio} \\
&\leq an\log n - \frac{a}{4}n + 2b - \frac{2b}{n} + \Theta(n) \\
&= an\log n + b + \left[-\frac{a}{4}n + b - \frac{2b}{n} + \Theta(n)\right] \\
&\leq an\log n + b
\end{aligned}$$

Verifichiamo ora che la maggiorazione usata nella dimostrazione effettivamente vale.

$$\begin{aligned}
\sum_{k=1}^{n-1} k\log k &= \sum_{k=1}^{\lceil \frac{n}{2}\rceil - 1} k\log k + \sum_{k=\lceil \frac{n}{2}\rceil}^{n-1} k\log k \\
&\leq \log\frac{n}{2}\sum_{k=1}^{\lceil \frac{n}{2}\rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2}\rceil}^{n-1} k
\end{aligned}$$

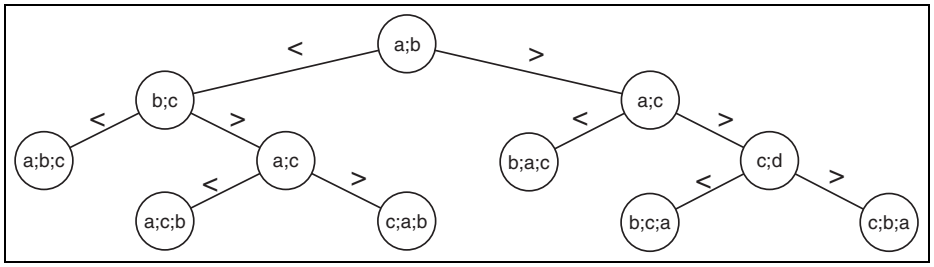

Figura 1.7: Esempio di Decision Tree

$$
\begin{aligned}
&= \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \\
&\leq \log \frac{n(n-1)}{2} - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{1}{2} \\
&\leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2
\end{aligned}
$$

1.7 Lower Bound al problema dell'ordinamento.

Per quanto riguarda gli algoritmi di ordinamento basati sul confronto dimostreremo ora un importate risultato e cioè che la complessità di tali algoritmi è almeno pari a:

$$\Omega(n \log n)$$

Per realizzare tale dimostrazione utilizzeremo i *decision tree* Un decision tree evidenzia tutti i confronti necessari per stabilire in che ordine si trovano n elementi. Più semplicemente troveremo nelle foglie tutte le possibili permutazioni degli elementi in ingresso, in dettaglio:

- i nodi interni sono etichettati con coppie di elementi;
- gli archi sono etichettati con un '<' o un '>' (a seconda del risultato del confronto fra i due elementi presenti nel nodo interno che li generano);
- tutte le foglie sono etichettate con le permutazioni degli n elementi;
- i confronti dalla radice ad una qualsiasi foglia devono produrre le permutazioni associate alle foglie.

Possiamo quindi vedere un decision tree come una rappresentazione di tutte le possibili esecuzioni di un qualsiasi algoritmo di ordinamento basato sul confronto. Tale algoritmo deve fare almeno i confronti presenti sul cammino dalla radice alla permutazione ordinata.

Il *caso peggiore* in termini delle sole operazioni di confronto necessarie corrisponde al cammino più lungo che riusciamo ad individuare. Preoccuppandoci di ordinare n elementi otteniamo che il relativo decision tree ha $n!$ foglie, pari al numero di permutazioni possibili di n elementi.

Come si correlano il numero di foglie e l'altezza? La minima altezza possibile per un albero binario con $n!$ foglie (realizzato quando l'albero binario è completo) è $\log n!$, ne segue che l'altezza h di un decision tree è maggiore o uguale a $\log n!$. Daremo ora una stima di $\log n!$ ricordando che il nostro obiettivo è di dimostrare che è uguale a $\Omega(n \log n)$.

Iniziamo osservando che:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2} + 1\right) \cdot \left(\frac{n}{2} + 2\right) \cdot \ldots \cdot n$$

Se consideriamo i fattori a partire dal $\left(\frac{n}{2} + 1\right)$-esimo si può notare che ognuno degli $\frac{n}{2}$ fattori considerati è maggiore di $\frac{n}{2}$. Quindi posso affermare che:

$$n! \geq 1 \cdot \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}}_{\frac{n}{2}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Riassumendo:

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$$

ovvero l'altezza h di un decision tree per n elementi è $\Omega(n \log n)$.

Abbiamo così dimostrato che ogni algoritmo di ordinamento basato sul confronto, non facendo altro che percorrere uno dei rami del decision tree collegato al suo input, ha costo almeno di $n \log n$.

Se scarichiamo ora l'ipotesi che l'unico mezzo a nostra disposizione per ordinare dei numeri sia il confronto, vedremo che sarà possibile ottenere algoritmi con costo lineare.

Capitolo 2

Algoritmi lineari

In questo capitolo sono raccolti alcuni algoritmi di ordinamento che sfruttano operazioni diverse dal confronto diretto tra elementi per produrre una permutazione ordinata dell'input.

2.1 Counting Sort

Se invece di n interi qualsiasi consideriamo n interi compresi tra 0 e k (con k costante fissata a priori) possiamo migliorare la complessità dell'ordinamento, ad esempio usando counting sort.

L'idea alla base di questo algoritmo è di determinare per ogni elemento in input x il numero di elementi minori dello stesso. Con una scansione dell'input possiamo contare il numero di volte che ciascun numero compreso tra 1 e k compare in input. Quindi con una ulteriore scansione di lunghezza k identifichiamo il numero di elementi presenti in input minori o uguali a i, dove $i \in [1, k]$. Infine un'ultima scansione dell'input ci permetterà di identificare la corretta posizione di ogni elemento. Intuitivamente otterremo complessità pari a $O(k + n)$=$O(n)$ in quanto k è una costante. In particolare utilizzeremo tre array:

- A di dimensione n contenente l'input
- B di dimensione n contenente l'output
- C di dimensione k contenente il contatore

Come si può facilmente notare non è un algoritmo in-place, prevede infatti l'uso di un array C di dimensione k e nel caso in cui k cresca l'algoritmo può diventare molto dispendioso in termini di memoria.

Una delle caratteristiche principali di questo algoritmo è quella di essere *stabile*, come evidenziato in figura 2.1.

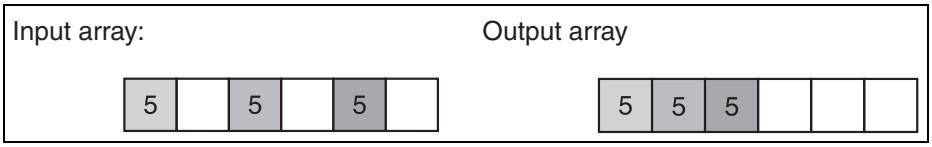

Figura 2.1: Esempio di algoritmo di ordinamento stabile

Definizione 2.1 (Stabile) *Si dice che un algoritmo di ordinamento è stabile se e solo se elementi di pari valore nell'input conservano la loro posizione relativa nell'output.*

Tale proprietà risulta molto utile, ad esempio nell'ordinamento di stringhe.
Vediamo ora il sorgente:

Algoritmo 2.1 Countign Sort(A, B, k)

Countign Sort(A, B, k)

```
1: for i ← 0 to k do
2:     C[i] ← 0
3: end for
4: for j ← 1 to length(A) do
5:     C[ A[j] ] ← C[ A[j] ] + 1
6: end for
7: for i ← 2 to k do
8:     C[i] ← C[i − 1] + C[i]
9: end for
10: for j ← length(A) downto 1 do
11:     B[ C[ A[j] ] ] ← A[j]
12:     C[ A[j] ] ← C[ A[j] ] − 1
13: end for
```

Si può perfezionare l'algoritmo eliminando B e riutilizzando A per riscrivere il vettore ordinato.

2.2 Radix Sort

Vediamo ora un altro algoritmo di ordinamento, radix sort che utilizza counting sort (algoritmo 2.1) come subroutine.

Le principali caratteristiche sono:

- non è basato sul confronto;
- si utilizza per ordinare interi di k digit. I numeri da ordinare quindi sono compresi fra 0 e $10^k - 1$, se si usa la base 10, altrimenti tra 0 e $b^k - 1$ con una generica base b. Le ipotesi ci permettono di utilizzare Counting Sort, anche se può risultare costoso in termini di spazio.

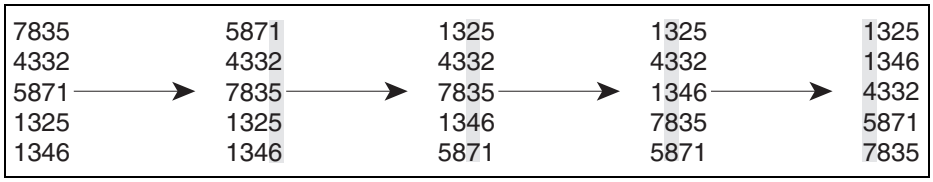

Figura 2.2: Esempio di funzionamento di Radix Sort

- utilizza Counting Sort come subroutine. Un qualunque altro algoritmo di ordinamento andrebbe bene, l'importante è che sia stabile.

L'idea base è quella di ordinare i numeri in ingresso confrontando le singole cifre considerando prima quelle più significative e poi via via quelle meno significative. Applicando lo schema quindi iniziamo con l'ordinare rispetto alla cifra più significativa, così facendo otteniamo dei gruppi di numeri che hanno la cifra più significativa uguale. A questo punto possiamo riapplicare l'idea ai singoli gruppi di numeri ottenuti al passo precedente, considerando però la seconda cifra più significativa.

Questo algoritmo può essere implementato sia in maniera ricorsiva sia in maniera iterativa, comunque si ottiene un algoritmo tutt'altro che semplice. La difficoltà infatti è detrminata dal mantenere distinti i vari gruppi che si vengono via via a formare.

Quello che si fa realmente è di usare la stessa idea partendo però dalla cifra meno significativa. Si ottiene un algoritmo sufficientemente semplice e non ricorsivo. In figura 2.2 è riportato un esempio di Radix Sort su un input di 5 interi di 4 cifre ognuno.

Algoritmo 2.2 RadixSort(A, k)

RadixSort(A, k)

1: **for** $i \leftarrow 1$ to k **do**
2: SortStable(A, i) {Ordina A rispetto all'i-esima cifra}
3: **end for**

Come già detto essendo gli interi compresi tra 0 e $d^k - 1$, dove d è la base della rappresentazione, la complessità sarebbe di $O\big(n + d^k\big)$ ma dovremmo utilizzare un array di supporto di dimensione d^k.

Con Radix Sort iteriamo k passi per ciascuno dei quali eseguiamo Counting Sort su n elementi compresi tra 0 e $d - 1$. Questo implica che otteniamo una complessità pari a $O(d(n + k))$, ma k e d costanti portano a una complessità di $O(dn)$.

2.2.1 Correttezza

Ci poniamo ora il problema di dimostrare la correttezza dell'algoritmo.

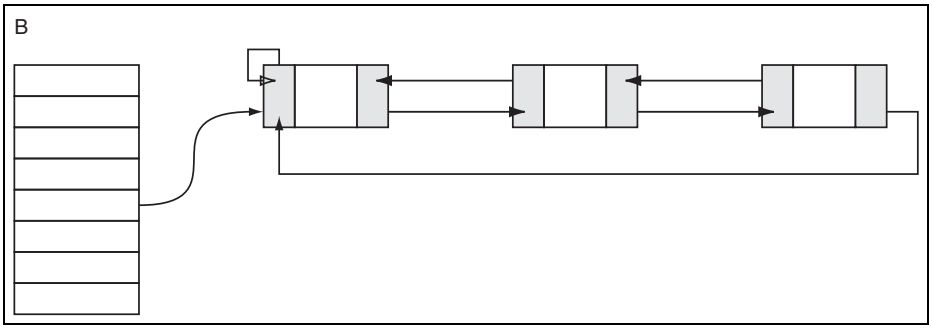

Figura 2.3: Schema di realizzazione di Bucket Sort

Teorema 2.2.1 *Dati n interi rappresentati con stringhe di k digits in base d, ciascuno costituente un array A k-dimensionale, Radix Sort(A, k) li ordina correttamente.*

Dimostrazione. Vediamone uno schema.

Prima di tutto è fondamentale fare un'assunzione sulla correttezza di SortStable(), se infatti questo algoritmo non risultasse corretto, tutto il nostro lavoro sarebbe inutile. Faremo la dimostrazione per induzione su k.

$k = 1$ la correttezza discende da Sort Stable.

$k > 0$ le prime $d-1$ iterazioni del ciclo for costituiscono una chiamata a RadixSort(A', d') con $d' = d-1$ e A' array contenente gli n numeri ottenuti cancellando il digit più significativo dagli elementi di A.

Per ipotesi induttiva, dopo le prime $d-1$ iterazioni gli elementi di A sono ordinati rispetto ai $d-1$ digit meno significativi. L'ultima iterazione ordina rispetto al digit più significativo e due elementi aventi uguale digit risultano ordinati come conseguenza della stabilità di SortStable() e della ipotesi induttiva.

■

2.3 Bucket Sort

Bucket Sort riceve come input un array A di n elementi il cui valore è compreso fra 0 e 1. Tale ipotesi risulta non essere riduttiva: qualunque reale può essere infatti ricondotto ad un reale nell'intervallo suddetto a mezzo di opportune divisioni.

L'idea è di dividere l'intervallo $[0, 1)$ in sottointervalli, detti *bucket*, di uguali dimensioni nei quali vengono distribuiti gli n numeri in input.

L'ipotesi migliore sotto la quale Bucket Sort funziona è che l'array in input contenga n reali uniformemente distribuiti nell'intervallo $[0, 1)$, in quanto ci assicura che i singoli bucket non verranno sovraccaricati.

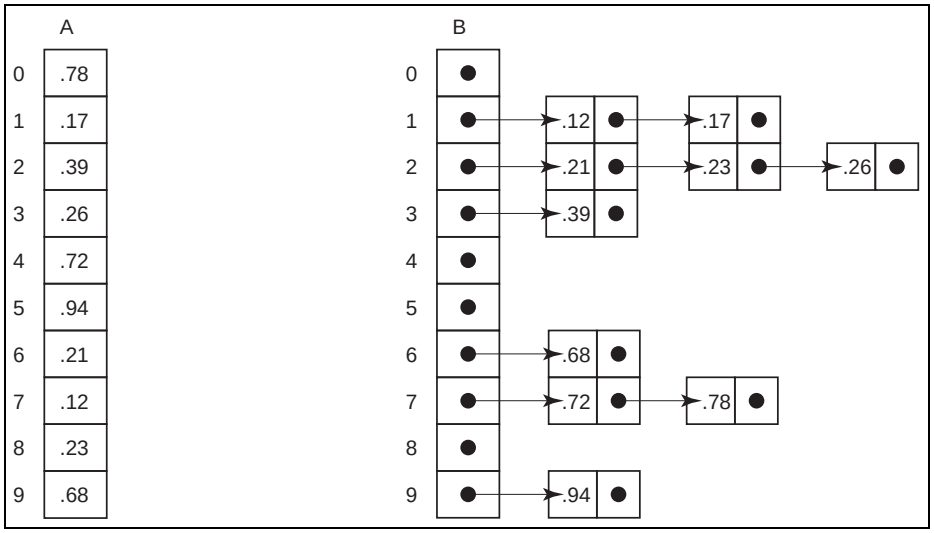

Figura 2.4: Esempio di funzionamento di Bucket Sort

Per implementare Bucket Sort sfrutteremo un array B di linked list, conseguentemente l'algoritmo risulta non in-place. Ogni bucket viene rappresentato da un un puntatore ad una linked list, l'insieme dei bucket da un array di puntatori come in figura 2.3. Per produrre l'output si ordinano gli elementi presenti in ogni bucket e poi si scorrono i bucket in ordine.

Algoritmo 2.3 Bucket Sort(A)

Bucket Sort(A)

1: $n \leftarrow length(A)$
2: **for** $i \leftarrow 0$ to $n-1$ **do**
3: InsertList($A[i]$, $B[\lfloor nA[i] \rfloor]$)
4: **end for**
5: **for** $i \leftarrow 0$ to $n-1$ **do**
6: sort($B[i]$)
7: **end for**
8: concatena le liste $B[0], B[1], \ldots, B[n-1]$

Per ottenere una buona complessità è necessario garantire che il numero di elementi in ogni bucket $B[i]$ sia piccolo. Il caso peggiore si ha quando tutti gli elementi vanno a finire in uno stesso bucket, nel qual caso la complessità coincide con quella di sort(). Si può dimostrare che nell'ipotesi di distribuzione uniforme dell'input la complessità media di bucket sort è lineare anche usando una routine $O(n^2)$ come sort().

Se riusciamo a garantire che le liste siano sufficientemente corte non ha importanza quale algoritmo usiamo per ordinare i singoli bucket infatti

anche se fosse di complessità cubica, avrebbe un numero costante di elementi, dandoci una complessità asintotica lineare.

Vediamo comunque quali sono le complessità di questo algoritmo nei vari casi:

Peggiore: $O(n \log n)$ in questo caso tutti gli elementi di A vanno a finire nello stesso bucket;

Migliore: $O(n)$ in questo caso ogni elemento cade in un bucket distinto;

Medio: $O(n)$ con ipotesi di distribuzione uniforme dell'input.

2.3.1 Correttezza

Siano $A[i]$ e $A[j]$ due elementi in input generici, dobbiamo considerare due casi:

1. I due elementi cadono nello stesso bucket, il loro ordinamento è quindi assicurato dalla correttezza di sort()

2. I due elementi cadono in due bucket distinti.

   Siano $B[i']$ e $B[j']$ i due bucket che accolgono rispettivamente $A[i]$ e $A[j]$ e supponiamo $i' < j'$, quando i bucket vengono concatenati gli elementi di $B[i']$ precedono gli elementi di $B[j']$ e quindi anche $A[i]$ precederà $A[j]$ nella sequenza di output. Bisogna quindi dimostrare che $A[i] < A[j]$, proviamo a dimostrarlo per assurdo. Se $A[i] > A[j]$ allora si avrebbe che $i' = \lfloor nA[i] \rfloor > \lfloor nA[j] \rfloor = j'$ e quindi deve essere che $A[i] < A[j]$.

2.4 Selection Problem

Cerchiamo ora di dare una soluzione al seguente problema:

Input: una lista di n interi e un indice $i \in \{1, \ldots, n\}$;

Output: l'i-esimo elemento della ricerca ordinata dell'input.

Possiamo quindi dare la seguente definizione:

Definizione 2.2 (i-esimo elemento) *x è l'i-esimo elemento di A se e solo se in A ci sono almeno $i - 1$ elementi più piccoli dell'i-esimo.*

In particolare identificheremo il *minimo* come il primo elemento, e il *massimo* come l'ultimo elemento. Possiamo anche identificare l'elemento di mezzo.

Ricaviamo una soluzione banale per stabilire un upper bound di partenza: ordiniamo il vettore e accediamo all'i-esimo elemento con un costo di $O(n \log n)$.

Stiamo svolgendo del lavoro inutile, infatti se ordiniamo tutto il vettore non solo potremmo identificare l'elemento cercato ma anche tutti gli altri, risolvendo tutta una classe di problemi. Un'altra considerazione che ci induce a pensare che si possa fare meglio è l'analisi dei due casi particolari: massimo e minimo. Entrambi infatti posso essere identificati in tempo lineare $O(n)$.

Vediamo una soluzione lineare, sarà legata alla procedura partition (vedi algoritmo 1.13). Identificato un elemento come pivot partition mette da una parte gli elementi più piccoli e da un'altra quelli più grandi dell'elemento prescelto. Possiamo contare gli elementi del primo gruppo e se questo numero è maggiore di i, continuare la ricerca solo in questo gruppo ignorando tutto il secondo. Viceversa se i è maggiore del numero di elementi del primo gruppo ignoriamo questo e continuiamo la ricerca nel secondo gruppo usando la stessa tecnica.

Il problema è dividere efficientemente l'array. Dobbiamo usare come pivot l'elemento medio, in questo modo dimezziamo ad ogni passo il problema, ma cercare l'elemento medio significa saper ricavare correttamente l'i-esimo elemento, siamo quindi al problema di partenza.

Ci troviamo in una situazione in cui per risolvere il problema abbiamo bisogno di una soluzione dello stesso da usare in una procedura ricorsiva.

Vorremmo usare ricorsivamente partition.

Algoritmo 2.4 RandomizedSelect(A, p, r, i)

RandomizedSelect(A, p, r, i**)**

```
1: if p = r then
2:    return A[p]
3: end if
4: q ← RandomizedPartition(A, p, r)
5: k ← q − p + 1
6: if i ≤ k then
7:    return RandomizedSelect(A, p, q, i)
8: else
9:    return RandomizedSelect(A, q + 1, r, i − k)
10: end if
```

2.4.1 Complessità

$$T(n) = O(n) + \begin{cases} T(1) & \text{caso migliore } O(n) \\ T(n-1) & \text{caso peggiore } O\big(n^2\big) \end{cases}$$

Il caso peggiore lo otteniamo quando continuiamo a dividere usando come pivot l'elemento massimo o minimo producendo due sottoarray sbilanciati,

in questo caso l'equazione diviene:

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= \sum_{i=1}^{n} O(i) \\ &= O\big(n^2\big) \end{aligned}$$

Nel caso medio invece l'equazione di complessità è data da:

$$\begin{aligned} T(n) &= T\Big(\frac{n}{2}\Big) + \Theta(n) \text{ sia } n = 2^k \\ T\Big(2^k\Big) &= T\Big(2^{k-1}\Big) + \Theta\Big(2^k\Big) \\ G(k) &= G(k-1) + \Theta\Big(2^k\Big) \\ \sum_{i=0}^{\log n} O\big(2^i\big) &= O\left(\sum_{i=0}^{\log n} 2^i\right) = \Theta\big(2^i\big) \\ T\Big(2^k\Big) = \Theta\big(2^i\big) &\Leftrightarrow T(n) = \Theta(n) \end{aligned}$$

Il problema può essere risolto in maniera che l'algoritmo lavori in tempo $O(n)$ nel caso peggiore. L'idea è quella di garantire che il pivot si trovi in posizione tale da produrre una chiamata ricorsiva su un input che ha dimensione pari ad una frazione fissa dell'input di partenza.

2.5 Order Statistics

$O(n)$ dividiamo gli elementi in input in gruppi di 5 elementi;

$O(n)$ ordiniamo ognuno dei gruppi ottenuti e determiniamo l'elemento medio di ogni gruppo;

$T\big(\frac{n}{5}\big)$ eseguiamo una chiamata ricorsiva sulla lista degli elementi medi ottenuta al passo precedente e otteniamo la mediana di quegli elementi;

$O(n)$ applichiamo partition su tutto l'array in input utilizzando l'elemento ottenuto al passo precedente come pivot;

$T\big(\frac{3}{4}n\big)$ procediamo ricorsivamente su uno dei sottoarray determinato come nel caso di Randomize Select.

Per analizzare il tempo di esecuzione dell'algoritmo dobbiamo prima determinare quanti elementi sono più grandi del pivot. Chiamiamo p il pivot così trovato. Almeno metà delle mediane trovate al passo 2 sono maggiori della mediana delle mediane e quindi almeno metà degli $\lfloor \frac{n}{5} \rfloor$ gruppi contribuiscono con 3 elementi che sono più grandi di x tranne il gruppo di x

stesso e quello che ha meno elementi se 5 non divide esattamente n e quindi avremo almeno:

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right)\geq\frac{3}{10}n-6$$

elementi più grandi di x. Un ragionamento speculare può essere fatto per il numero di elementi minori di x che quindi sono $\geq\frac{3}{10}n-6$.

Nel caso peggiore quindi la chiamata verrà effettuata su di un array di dimensione $\frac{7}{10}n+6$ elementi. Possiamo quindi usare l'equazione:

$$T(n)=\begin{cases}\Theta(1)\ \text{per n}\leq\text{costante}\\ T\left(\left\lceil\frac{n}{5}\right\rceil\right)+T\left(\frac{7}{10}n+6\right)+\Theta(n)\end{cases}$$

Dimostriamo che per qualche costante c $T(n)\leq cn$

$$\begin{aligned}T(n)&\leq c\left\lceil\frac{n}{5}\right\rceil+c\left\lceil\frac{7}{10}n+6\right\rceil+O(n)\\&\leq c\frac{n}{5}+c+\frac{7}{10}cn+6c+O(n)\\&\leq\frac{9}{10}cn+7c+O(n)\\&\leq cn\end{aligned}$$

Quindi abbiamo:

$$c\frac{9}{10}n+7c+kn\leq cn\Leftrightarrow kn\leq c\left(\frac{1}{10}n-7\right)\Leftrightarrow k\leq c\left(\frac{1}{10}-\frac{7}{n}\right)\Leftrightarrow c\geq\frac{k}{\left(\frac{1}{10}-\frac{7}{n}\right)}$$

Se prendiamo n ≥ 80 $c=\frac{k}{\frac{1}{10}-\frac{7}{80}}$ allora c è sicuramente più grande.

Capitolo 3

Esercizi (Ordinamento)

Esercizio 3.1 *(tratto dall'esame di ASD del 07-07-03)*

Si consideri il problema di ordinare un array $A[1 \ldots n]$ contenente al più k interi distinti, dove k è una costante.

- Si proponga un algoritmo efficiente per il problema proposto e se ne valuti la complessità.
- Si proponga un algoritmo efficiente ed *in place* per il problema proposto e se ne valuti la complessità.
- **Facoltativo** Si proponga un algoritmo *in place* e *stabile* per il problema proposto e se ne valuti la complessità.

Esercizio 3.2 *(tratto dall'esame di ASD del 02-12-02)*

Si supponga di rappresentare mediante una coppia di interi $[l, u]$ con $l \leq u$, l'intervallo di interi compresi tra l ed u (estremi inclusi). Sia dato l'insieme di n intervalli $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$.

a Si descriva un algoritmo efficiente per determinare se esistono (almeno) due intervalli disgiunti in S.

b Si delinei lo pseudocodice dell'algoritmo descritto al punto precedente e se ne studi la complessità.

c Si riconsiderino i precedenti due punti nel caso il problema sia quello di determinare se esistono due intervalli non disgiunti.

Esercizio 3.3 *(tratto dall'esame di ASD del 26-09-02)*

Sia A un vettore di n interi e si consideri il problema di determinare se esistono due interi che occorrono in A lo stesso numero di volte.

a Si descriva un algoritmo efficiente per il problema proposto nel caso in cui in A occorrono c valori distinti, dove c è una costante intera positiva.

b Si descriva un algoritmo efficiente per il problema proposto.

c Si determini la complessità degli algoritmi proposti.

Esercizio 3.4 *(tratto dall'esame di ASD del 02-09-02)*

Sia A un vettore di n interi compresi tra 1 e una costante intera positiva c.

a Si descriva un algoritmo efficiente che ordini A.

b Si descriva un algoritmo efficiente *in place* che ordini A.

c Si determini la complessità degli algoritmi proposti in funzione sia di n che di c e si valuti la complessità di tali algoritmi nei casi in cui $c = O(\lg n)$ e $c = O(n)$.

Esercizio 3.5 *(tratto dall'esame di ASD del 18-07-02)*

Sia A un vettore di $n = 2^k$ interi e si supponga di disporre di un algoritmo `Mediano`(A, i, j) che restituisce l'elemento mediano della sequenza $A[i..j]$, ovvero l'elemento di posizione $\lfloor (i + j)/2 \rfloor$ nella sequenza ordinata degli elementi di $A[i..j]$.

- Scrivere lo pseudo-codice di un algoritmo efficiente `Select`(A, i) che determini l' i-esimo piu' piccolo elemento di A.
- Si dimostri la correttezza e si valuti la complessita' dell'algoritmo proposto, supponendo che l' algoritmo `Mediano`(A, i, j) abbia complessita' $O(j - i)$.

Esercizio 3.6 *(tratto dall'esame di ASD del 18-06-02)*

Sia A un array contentente $n = 2m$ interi:

- Si proponga un algoritmo che, preso in input A, produca come output un array in cui gli elementi in posizione pari costituiscano una successione crescente e quelli in posizione dispari costituiscano una successione decrescente.

  Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

- Si riconsideri il punto precedente e si proponga un algoritmo che svolga la stessa funzione ma che, in piú, operi *in-place*.

Esercizio 3.7 *(tratto dall'esame di ASD del 24-09-01)*

Si consideri un alfabeto Σ costituito da quattro caratteri: $\Sigma = \{G, A, T, C\}$ e sia α una stringa finita su Σ (i.e. $\alpha \in \Sigma^\star$). Per ognuno dei quattro caratteri $X \in \Sigma$, si supponga di avere in input l'insieme S_X delle lunghezze dei prefissi di α che terminano con X:

$$S_X \quad = \quad \{a \mid a = |\beta| \wedge \beta \sqsubseteq \alpha \wedge \beta(a) = X\}.$$

Si proponga un algoritmo che determini α a partire da quattro array contenenti ognuno uno dei possibili S_X. Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 3.8 *(tratto dall'esame di ASD del 13-07-01)*

Si descrivano due generici input che rappresentino, rispettivamente, il *best case* ed il *worst case* per l'algoritmo `quick_sort`.

Esercizio 3.9 *(tratto dall'esame di ASD del 28-06-01)*

Data una lista A di 3^k ($k \geq 0$) interi distinti, si consideri il seguente algoritmo $\mathcal{M}(A)$:

step 1 se $k = 0$ $\mathcal{M}(A)$ ritorna in output l'unico elemento di A, altrimenti divide A in 3^{k-1} gruppi di 3 elmementi ciascuno;

step 2 per ognuno dei gruppi determinati al passo precedente determina l'elemento medio;

step 3 $\mathcal{M}(A)$ esegue una chiamata ricorsiva sulla lista costituita dai 3^{k-1} elementi determinati al passo precedente.

Si determini la complessità di $\mathcal{M}(A)$; si provi o si refuti la seguente affermazione: *l'elemento ritornato in output da $\mathcal{M}(A)$ è l'elemento medio di A.*

Esercizio 3.10 *(tratto dal compitino di ASD del 30-03-01)*

Dato un vettore A contenente n interi $a_1, \ldots, a_n$ distinti tra di loro, definiamo *spiazzamento* dell'elemento a_i l'intero k tale che $i + k$ sia la posizione di a_i nella versione ordinata del vettore A. Definiamo inoltre un vettore *quasi-ordinato* se tutti i suoi elementi hanno uno spiazzamento che in valore assolutao è minore o uguale ad uno.

a Si valuti la complessità degli algoritmi di ordinamente insertion-sort, heap-sort e quick-sort, quando il vettore di ingresso sia quasi-ordinato.

b Si proponga un'algoritmo di complessità lineare per l'ordinamento di vettori quasi-ordinati.

c Si fornisca una versione modificata dell'algoritmo di quick-sort ottimizzata al caso di dover ordinare vettori quasi-ordinati.

Esercizio 3.11 *(tratto dall'esame di ASD del 14-12-00)*

Sia x un array $(x_1, \ldots, x_n)$ di interi e sia p una permutazione degli indici $\{1, \ldots, n\}$. Si consideri il seguente algoritmo:

Algoritmo 3.1

```
1: for j := 1 to n do
2:    k := p(j);
3:    while k > j do
4:       k := p(k);
5:    end while
6:    if k = j then
7:       y := x[j]; l := p(k);
8:       while l ≠ j do
9:          x[k] := x[l]; k := l; l := p(k);
10:      end while
11:      x[k] := y;
12:   end if
13: end for
```

Si commenti **Algoritmo 1** e si provi che ricevuto in input x produce in output l'array $(x_{p(1)}, \ldots, x_{p(n)})$. Si valuti la complessità di **Algoritmo 1**.

Si produca un input relativamente al quale la complessità di **Algoritmo 1** sia pessima.

Esercizio 3.12 *(tratto dall'esame di ASD del 29-11-00)*

Un vettore S di n numeri interi si defisce *k-quasi ordinato* se esiste un vettore ordinato S' che differisce da S per al più k elementi, con k costante.

- Si proponga un algoritmo lineare per l'ordinamento di vettori k-quasi ordinati. Si discuta la correttezza dell'algoritmo proposto.
- Si proponga un algoritmo lineare di ordinamento per il caso in cui il vettore S differisca da un vettore ordinato per al più $\sqrt{n}$ elementi.

Esercizio 3.13 *(tratto dall'esame di ASD del 31-08-00)*

Si consideri un array A contenente n interi (a due a due distinti) quale input per l'algoritmo di ordinamento `Heapsort`.

Qual è la complessità di `Heapsort` quando gli interi in A costituiscono già una sequenza ordinata? Si giustifichi la risposta. Si supponga che A sia sempre costituito da un segmento finale di lunghezza almeno $\lfloor n/2 \rfloor$ già

ordinato, e si ottimizzi `Heapsort` in modo da migliorarne la complessità su A.

Esercizio 3.14 *(tratto dal compitino di ASD del 22-2-00)*

Diciamo *propriamente bitonica* una sequenza S di numeri interi $a_1, \ldots, a_n$ per cui esista $k \in \{1, \ldots, n\}$ tale che $a_1, \ldots, a_k$ sia monotona crescente (decrescente) e $a_k, \ldots, a_n$ sia monotona decrescente (crescente). Diremo *bitonica* una sequenza di interi che possa essere resa propriamente bitonica mediante una permutazione ciclica degli indici.

Esempi: $\{3, 5, 7, 11, 12, 8, 4, -2, -11\}$ sequenza (propriamente) bitonica; $\{7, 11, 12, 8, 4, -2, -11, 3, 5\}$ sequenza bitonica.

Una sequenza bitonica è una sequenza con al più un *picco* (massimo locale interno) e al più una *valle* (minimo locale interno)

(a) Si proponga un algoritmo che, data in input una sequenza bitonica S, produca in output due sequenze bitoniche S_1 ed S_2 di dimensione $n/2$ tali che

$$S_1 \cap S_2 = \emptyset, \quad S_1 \cup S_2 = S, \quad \forall s_1 \in S_1, \forall s_2 \in S_2 (s_1 \leq s_2)$$

(b) Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 3.15 *(tratto dal compitino di ASD del 22-2-00)*

Definiamo *heap ternario* la naturale modifica dello heap (binario); ossia un albero ternario, (ogni nodo ha tre figli, eventalmente vuoti) in cui la chiave associata al padre è maggiore delle chiavi associate ai tre figli.

- Si proponga un implementazione dello heap ternario traminte un vettore. In particolare si scrivano le procedure: `Parent(i), Left(i), Heapify(A,i) e Build-Heap(A)`.
- Si discuta il punto precedente nel caso di un *heap esponenziale* ossia di un heap dove ogni nodo di altezza h, ha 2^h figli. Qual è la complessità di un algoritmo di heap-sort che utilizzi un heap esponenziale?

Esercizio 3.16 *(tratto dall'esame di ASD del 24-1-00)*

Dato un insieme S contenente n numeri interi, si considerino i seguenti due problemi:

- determinare la coppia di elementi s_1, s_2 in S tale che il valore $|s_1 - s_2|$ risulti minimo;
- determinare la tripla di elementi s_1, s_2, s_3 in S tale che la somma $|s_1 - s_2| + |s_2 - s_3| + |s_1 - s_3|$ risulti minima.

Si scriva lo pseudo-codice di due algortimi che risolvano i i due problemi precedenti. Si determini e si giusitfichi la complessità degli algoritmi proposti.

Esercizio 3.17 *(tratto dall'esame di ASD del 2-9-99)*

Dato un numero naturale n, si consideri un vettore A contenente n valori compresi tra 1 ed 2^n, e tale che in ogni intervallo $[2^m, 2^{(m+1)}]$ ci siano al più k valori di A, con k costante.

(a) Si scriva lo pseudo codice di un algorimo che ordini il vettore A.

(b) Si discuta la correttezza dell'algoritmo proposto e se ne valuti la complessità, nell'ipotesi ogni operazione sugli interi abbia costo costante e nell'ipotesi i numeri siano rappresentati in base 2 e le operazioni fondamentali sugli interi abbiano un costo proporzionale al loro numero di cifre.

(c) Si confronti la complessità dell'algoritmo proposto con quella di mergesort.

(d) Si riconsiderino i punti (a) e (b) nel caso in cui ogni intervallo $[2^m, 2^{(m+1)}]$ contenga al più $\ln n$ valori di A.

Si riconsiderino i punti (a) e (b) nel caso in cui ogni intervallo $[2^m, 2^{(m+1)}]$ contenga al più $\sqrt{n}$ valori di A.

Esercizio 3.18 *(tratto dall'esame di ASD del 8-7-99)*

Dati un numero naturale m e una matrice $A = (a_{i,j})$ quadrata di dimesione n, avente come valori numeri interi distinti e soddisfacente la sequente proprieta':

$$\forall i, i', j, j' \, (\, i < i' \Rightarrow a_{i,j} < a_{i',j'}),$$

Si scriva lo pseudocodice di un algoritmo che, considerando l'ordine sugli interi, determina l'm-esimo elemento della matrice A. Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 3.19 *(tratto dall'esame di ASD del 15-6-99)*

Sia A una matrice $n \times n$ di numeri interi.

(a) Si dia lo psedocodice di un algoritmo che determina l'elemento di A che appartiene al maggior numero di righe, nell'ipotesi che ogni riga contenga elementi tutti distinti.

(b) Si risolva il punto precedente per una generica matrice.

(c) Si discuta la correttezza e si valuti la complessità degli algoritmi proposti.

Esercizio 3.20 *(tratto dal compitino di ASD del 23-2-99)*

Si considerino gli algoritmi *build_heap*, *extract_min* e *insert* definiti per operare su una *min_heap* H di profondità h e contenente n nodi.

Si proponga una variante della struttura dati *min_heap* atta a supportare le suddette procedure e anche una procedura *extract_max* di estrazione del massimo. Si discuta la correttezza e si valutino le complessità degli algoritmi proposti in funzione dei parametri h ed n.

Esercizio 3.21 *(tratto dal compitino di ASD del 23-2-99)*

Sia dato un vettore $A[1..n]$ di interi positivi.

Si scriva lo pseudo-codice di un algoritmo efficiente che trovi un indice k tale che:

$$| \sum_{i=1}^{k-1} A[i] - \sum_{j=k+1}^{n} A[j] \ | \leq \ A[k].$$

Si riconsiderino il problema nell'ipotesi di dover cercare un elemento A_k tale che:

$$| \sum_{A[i]<A[k]} A[i] - \sum_{A[k]<A[j]} A[j] \ | \leq \ A[k].$$

Esercizio 3.22 *(tratto dall'esame di ASD del 10-2-99)*

Sia dato un vettore A di interi, diversi da 0, di dimensione n.

(a) Si proponga un algoritmo efficiente ed *in–place* che modifichi il vettore A in modo tale che al termine il (sotto)insieme dei numeri positivi sia ordinato in modo crescente e quello dei numeri negativi in modo decrescente.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

(c) Si riconsiderino i quesiti (a) e (b) nell'ipotesi che venga richiesto anche che al termine dell'algoritmo nel vettore A ogni elemento positivo sia seguito da un elemento negativo, fino all'esaurimento dei numeri positivi o dei numeri negativi, e quindi compaiano tutti i numeri, positivi o negativi, rimanenti.

Esercizio 3.23 *(tratto dall'esame di ASD del 26-1-98)*

Sia H una heap contenente gli elementi $\{a_1, ..., a_n\}$ e dato A sottoinsieme di $\{a_1, ..., a_n\}$, sia $H(A)$ la più piccola sotto-heap di H contenente gli elmenti di A.

Si proponga un algoritmo efficiente che, dati $A_1, ..., A_k$ sottoinsiemi di $\{a_1, ..., a_n\}$, li ordini in modo tale che $A_i < A_j \Rightarrow H(A_i) \subseteq H(A_j)$.

Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 3.24 *(tratto dall'esame di ASD del 26-1-98)*

Si considerino n numeri interi $a_1, ..., a_n$ la cui somma sia n. Si disegni un algoritmo efficiente per ordinare $a_1, ..., a_n$. Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 3.25 *(tratto dall'esame di ASD del 28-9-98)*

Sia dato un generatore di sequenze (localmente) ordinate di numeri naturali, di valore compreso tra 0 e c (con c costante prefissata), il cui comportamento può essere descritto come una successione infinita di passi che si articolano nelle seguenti due fasi:

(fase 1) generazione di sequenze ordinate di lunghezza esponenzialmente crescente 2^i, con $i = 0, 1, \ldots, k$;

(fase 2) generazione di sequenze ordinate di lunghezza esponenzialmente decrescente 2^i, con $i = k, k-1, \ldots, 0$.

Proporre lo pseudo-codice, adeguatamente commentato, di un algoritmo che riceva in input le $2 \cdot (k+1)$ sequenze ordinate generate in un singolo passo e restituisca in output un'unica sequenza ordinata. Provare la correttezza e determinare la complessità dell'algoritmo proposto.

Esercizio 3.26 *(tratto dall'esame di ASD del 9-9-98)*

Si considerino le due seguenti varianti dell'algoritmo *heap-sort*. A partire da una *min-heap*,

- dopo aver restituito in output il minimo corrente x, (anziché rimuoverlo sostituendolo con la foglia più a destra dell'ultimo livello) determina l'elemento minimo fra le foglie dell'ultimo livello, scambialo con la foglia più a destra e sostituisci ad x tale elemento;
- scambia il nodo x che si trova in posizione di radice (minimo corrente) con il minimo fra i suoi figli e ripeti l'operazionione finché x non raggiunge la posizione di una foglia; a quel punto restituisci x in output e rimuovilo.

Si fornisca lo pseudo-codice commentato delle due varianti di heap-sort sopra descritte. Si provi la correttezza e si determini la complessità degli algoritmi proposti.

Esercizio 3.27 *(tratto dall'esame di ASD del 12-2-98)*

Sia A un vettore contenente n interi $a_1, \ldots, a_n$ a due a due distinti. Definiamo *spiazzamento* dell'elemento a_i l'intero k tale che $i + k$ sia la posizione di a_i nella versione ordinata di A.

Si fornisca lo pseudo-codice di commentato dettagliatamente di un algoritmo che, preso in input il vettore A, produca come output il vettore $B = [b_1, \ldots, b_n]$ degli spiazzamenti di $a_1, \ldots, a_n$. Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 3.28 *(tratto dall'esame di ASD del 28-1-98)*

Si assuma che la coppia di interi $[a_i, b_i]$, con $a_i \leq b_i$, rappresenti l'insieme degli interi compresi tra a_i e b_i (estremi inclusi). Si scriva un algoritmo efficiente con le caratteristiche che seguono:

- *Input:* un insieme di n coppie $[a_i, b_i]$, con $1 \leq i \leq n$, tali che $\forall i, j$ $(1 \leq i, j \leq n)$ $b_i - a_i = b_j - a_j$;
- *Output:* un insieme di m coppie $[c_i, d_i]$, con $1 \leq i \leq m$, comprendenti tutti e soli gli interi che appartengono ad uno ed uno solo degli insiemi $[a_i, b_i]$.

Si valutino correttezza e complessità dell'algoritmo proposto. Si discutano, infine, gli effetti della rimozione della condizione $\forall i, j$ $(1 \leq i, j \leq n)$ $b_i - a_i = b_j - a_j$ sulle soluzioni proposte per i quesiti (a) e (b).

Esercizio 3.29 *(tratto dall'esame di ASD del 1-10-97)*

Si scriva un algoritmo efficiente che, ricevuto in ingresso un insieme di n intervalli $[a_i, b_i]$, con a_i, b_i numeri interi e $1 \leq i \leq n$, stabilisca se la loro unione è un intervallo (nel qual caso, restituisca tale intervallo) o meno. Si fornisca lo pseudo-codice commentato dell'algoritmo e se ne dimostrino correttezza e complessità.

Esercizio 3.30 *(tratto dall'esame di ASD del 23-6-97)*

Si descriva un algoritmo efficiente che, ricevuto in input un vettore A di n interi positivi, restituisca in output l'insieme degli elementi che occupano una delle posizioni comprese tra la k-esima posizione e la $(k + m)$-esima posizione (estremi inclusi) nella permutazione che ordina A, con $1 \leq k \leq k + m \leq n$. Si fornisca lo pseudo-codice commentato dell'algoritmo e se ne valutino correttezza e complessità.

Esercizio 3.31 *(tratto dall'esame di ASD del 3-6-97)*

Si consideri la tecnica di implementazione di una min-heap mediante un array. Si consideri l'operazione, che chiameremo $insert\_cut(x)$, che consiste nell'inserire l'elemento x nella heap e, contemporaneamente, cancellare dalla heap tutti gli elementi maggiori di x.

- Si dimostri che l'array (rappresentante la heap) ottenuto a partire dall'array vuoto dopo m operazioni di tipo $insert\_cut(x)$ è ordinato.
- Si proponga una implementazione atta a supportare esclusivamente operazioni di tipo $insert\_cut(x)$.
- Si discuta una modifica della tecnica usata per l'implementazione che permetta di trattare anche operazioni di tipo *delete_min*, in modo che n operazioni *delete_min* ed m operazioni di tipo $insert\_cut(x)$ abbiano costo globale $\theta(m+n)$.

Esercizio 3.32 *(tratto dall'esame di ASD del 3-6-97)*

Sia A un array di interi positivi o negativi, di dimensione n e tale che

$$A[1] < A[2] < \ldots < A[n].$$

Scrivere lo pseudocodice commentato di un algoritmo che, preso in input A, produca come output un indice i tale che $A[i] = i$, se tale i esiste, o NIL altrimenti. Dimostrare la correttezza e determinare la complessità dell'algoritmo proposto.

Esercizio 3.33 *(tratto dall'esame di ASD del 27-1-97)*

Sia H una min-heap di altezza h contenente $2^{h+1} - 1$ elementi. Sia $H(p)$ l'insieme dei nodi di H aventi profondità p:

$$H(p) \quad = \quad \{a \in H \mid \text{ depth}(a) = p\}.$$

Si assuma che H sia rappresentata mediante un vettore e che, per ogni p, il sotto-vettore contenente la lista dei nodi in $H(p)$ sia ordinata.

- Si dimostri che il vettore contenente H non è necessariamente ordinato.
- Si proponga lo pseudo-codice commentato di un algoritmo efficiente che produca la lista ordinata dei nodi in H e se ne dimostri correttezza e complessità. (Suggerimento: Si consideri un algoritmo ricorsivo)

Esercizio 3.34 *(tratto dall'esame di ASD del 27-1-97)*

Sia A un vettore contenente $2n$ numeri interi, di cui n positivi ed n negativi.

Si scriva lo pseudo-codice commentato di un algoritmo efficiente che dati x, y con $x < y$ determina se esiste una coppia (i, j) tale che $A[i] < 0, A[j] > 0$ e $x < A[i] + A[j] < y$. Si dimostri la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 3.35 *(tratto dall'esame di ASD del 2-9-96)*

Sia k una costante intera maggiore di 1 e sia A un vettore di n interi tali che $0 \leq A[i] < kn$.

Scrivere lo pseudo-codice commentato di un algoritmo che ordina il vettore A in tempo lineare e che usa non più di $3n + 3$ locazioni ausiliarie di memoria. Dimostrare la correttezza dell'algoritmo proposto.

Esercizio 3.36 *(tratto dall'esame di ASD del 22-7-96)*

Sia A un vettore di n interi e sia m il numero di posizioni i tali che $A[i] > A[i+1]$.

- Scrivere lo pseudo-codice commentato, dimostrare la correttezza e determinare la complessità di un algoritmo efficiente che ordina il vettore A nel caso in cui m è una costante.
- Scrivere lo pseudo-codice commentato, dimostrare la correttezza e determinare la complessità di un algoritmo efficiente che ordina il vettore A nel caso in cui $m = O(\log n)$.
- Gli algoritmi proposti sono stabili? Motivare la risposta e, in caso negativo, indicare come rendere stabili i suddetti algoritmi.

Esercizio 3.37 *(tratto dall'esame di ASD del 24-6-96)*

Scrivere un algoritmo di complessità $O(n \cdot \log m)$ per fondere m vettori ordinati in un vettore A, dove n è il numero totale di elementi contenuti negli m vettori. È possibile risolvere il problema con un algoritmo avente complessità $O(n)$?

Esercizio 3.38 *(tratto dall'esame di ASD del 22-7-96)*

Si consideri un vettore di $n = 2^{2^k}$ elementi. Utilizzando la soluzione all'esercizio 3.37, scrivere lo pseudo-codice e l'equazione della complessità asintotica della seguente variante di MERGESORT:

- Suddividere il vettore in $\sqrt{n}$ sottovettori;
- ordinare (ricorsivamente) i sottovettori;
- fondere i sottovettori ordinati.

Valutare la complessità dell' algoritmo delineato.

Capitolo 4

Strutture Dati

Abbiamo visto quanto gli insiemi giochino un ruolo importante nell'informatica e nella matematica. In particolare in informatica sono spesso soggetti ad aumenti e diminuzioni nel numero di elementi, tale caratteristica ha portato a definirli come *insiemi dinamici*.

Possiamo immaginare che ogni insieme sia rappresentabile come una collezione di oggetti, ognuno dei quali identificato da un campo chiave, ad esempio KEY(), al quale si possono associare dei valori satellite che caratterizzano meglio l'oggetto. Possiamo a questo punto definire con più precisione cosa intendiamo con *strutture dati*:

Definizione 4.1 (Struttura Dati) *Indichiamo con struttura dati un insieme di metodi e tecniche per organizzare insiemi dinamici di dati.*

Supponiamo ora di avere un insieme dinamico S, una chiave k e un elemento x, ci preoccuperemo di garantire tutte o parte delle seguenti operazioni:

Search(S, k)	Input:	S insieme dinamico ed una chiave k;
	Output:	puntatore ad x tale che KEY$(x) = k$ oppure NIL
Insert(S, x)	Input:	S insieme dinamico ed una chiave k;
	Output:	$S \cup \{x\}$
Delete(S, x)	Input:	S insieme dinamico ed una chiave k;
	Output:	$S \setminus \{x\}$
Min(S)	Input:	S insieme dinamico ordinabile;
	Output:	puntatore al minimo;
Max(S)	Input:	S insieme dinamico ordinabile;
	Output:	puntatore al massimo;
Predecessor(S, x)	Input:	S insieme dinamico ordinabile;
	Output:	Il più grande tra gli elementi più piccoli di x
Successor(S, x)	Input:	S insieme dinamico ordinabile;
	Output:	Il più piccolo tra gli elementi più grandi di x

Vale la pena notare che le operazioni appena descritte si possono classificare in due grandi categorie: operazioni di *interrogazione* dell'insieme quali ricerca, minimo, massimo, successore, predecessore; operazioni di *modifica* dell'insieme quali inserimento e cancellazione. La complessità per ognuna di queste operazioni si esprimerà di solito in termini di $n = |S|$.

Un buon esempio di struttura dati sono è la Heap, già trattata nel paragrafo 1.5.

Vedremo da qui in avanti le seguenti strutture dati: stacks, code, linked lists, rooted trees.

4.1 Stacks (pile)

Permettono di implementare la politica LIFO (*Last In First Out*) e mettono a disposizione la procedura di $push(S, x)$ per inserimento di un nuovo elemento e la procedura $pop(S)$ per la cancellazione dell'ultimo elemento inserito.

La rappresentazione di una pila avviene per mezzo di un array $S[1, \ldots, n]$ ed un intero $top(S)$ tale che $top(S) \in \{0, \ldots, n\}$ e $top(S)$ punta sempre al primo elemento della pila.

Algoritmo 4.1 StackEmpty(S) procedura di Stack

StackEmpty(S)

```
1: if top(S) = 0 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

Algoritmo 4.2 Push(S, x) procedura di Stack

Push(S, x)

```
1: top(S) ← top(S)+1
2: S[top(S)] ← x
```

Ogni operazione ha costo $O(1)$. Vale la pena evidenziare che le procedure riportate non considerano i casi di *overflow* , ovvero il tentativo di inserire un elemento in una pila piena, e i casi di *underflow*, cioè il tentativo di estrazione da una pila vuota.

4.2 Queues (code)

Permettono di implementare la politica FIFO (*First In First Out*). Le operazioni di inserimento vengono effettuate per mezzo della procedura *enqueue*,

Algoritmo 4.3 Pop(S) procedura di Stack

Pop(S)
1: **if** StackEmpty(S) **then**
2: return (Underflow)
3: **else**
4: $top(S) \leftarrow top(S)-1$
5: return $S[top(S)+1]$
6: **end if**

mentre la cancellazione avviene attraverso la procedura *dequeue*. Solitamente una coda viene implementata con un array $Q[1, \ldots, n]$ e vengono mantenuti due puntatori $head(Q)$ e $tail(Q)$ che puntano rispettivamente alla testa e alla coda di Q.

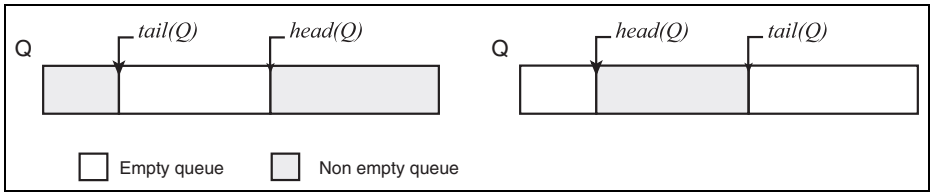

Figura 4.1: Esempio di uso di head e tail in una coda

Algoritmo 4.4 Dequeue(Q) procedura di Queue

Dequeue(Q)
1: $x \leftarrow Q[head(Q)]$
2: **if** $head(Q) = length(Q)$ **then**
3: $head(Q) \leftarrow 1$
4: **else**
5: $head(Q) \leftarrow head(Q) + 1$
6: **end if**
7: return x

4.3 Linked List

Sono strutture dinamiche che supportano tutte le operazioni su insiemi dinamici, anche se in modo non particolarmente efficiente. Nell'implementazione ogni elemento x della lista L è costituito da una chiave KEY(x) e da due puntatori: $next(x)$, $prev(x)$ che puntano rispettivamente all'elemento della lista successivo e precedente ad x; vengono mantenuti inoltre due puntatori, $head(L)$ e $tail(L)$, che indicano rispettivamente la testa e la coda della lista.

Algoritmo 4.5 Enqueue(Q, x) procedura di Queue

Enqueue(Q, x)

1: $Q[tail(Q)] \leftarrow x$
2: **if** $tail(Q) = length(Q)$ **then**
3: $\quad tail(Q) \leftarrow 1$
4: **else**
5: $\quad tail(Q) \leftarrow tail(Q) + 1$
6: **end if**

Una lista può avere diverse caratterizzazioni in particolare può essere *singly linked*, se il puntatore all'elemento precedente viene omesso oppure *doubly linked* se vengono considerati entrambi i puntatori, inoltre se $prev(head(L))$ punta a $tail(L)$ e $next(tail(L))$ punta a $head(L)$ la lista viene definita *circolare*.

Noi assumiamo di lavorare sempre con liste doubly linked circolari i cui elementi non siano già ordinati.

Per trattare in modo un po più elegante i casi relativi alle liste vuote, si introduce un particolare elemento con chiave NIL definito *sentinella* o elemento dummy, che premette di considerare la lista vuota come una normale lista formata dalla sola sentinella.

I costi delle operazioni sono rispettivamente:

- ListSearch: $\Theta(n)$
- ListInsert: $O(1)$
- ListDelete: $\Theta(n)$, in quanto l'elemento da eliminare deve prima essere individuato con una chiamata a ListSearch.

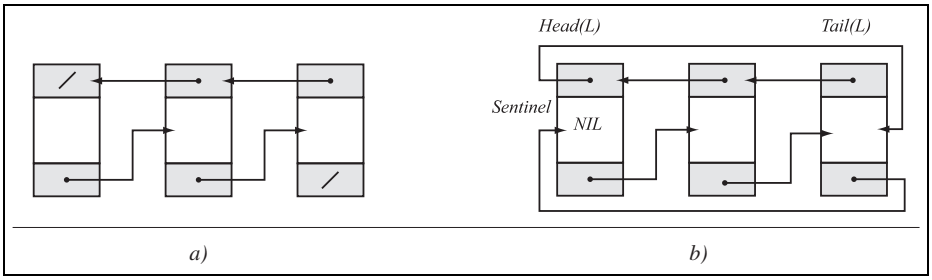

Figura 4.2: Esempi di: a) linked list; b) doubly linked list con sentinella

Algoritmo 4.6 ListSearch(L, k) procedura di Linked List

ListSearch(L, k)

```
1: x ← head(L)
2: while (x ≠ NIL ) ∧ (KEY(x) ≠ k) do
3:     x ← next(x)
4: end while
5: return x
```

Algoritmo 4.7 ListInsert(L, x) procedura di Linked List

ListInsert(L, x)

```
1: next(x) ← head(L)
2: if head(L) ≠ NIL then
3:     prev(head(L)) ← x
4: end if
5: head(L) ← x
6: prev(x) ← NIL
```

Algoritmo 4.8 ListDelete(L, x) procedura di Linked List

ListDelete(L, x)

```
1: if prev(x) ≠ NIL then
2:     next(prev(x)) ← next(x)
3: else
4:     head(L) ← next(x)
5: end if
6: if next(x) ≠ NIL then
7:     prev(next(x)) ← prev(x)
8: end if
```

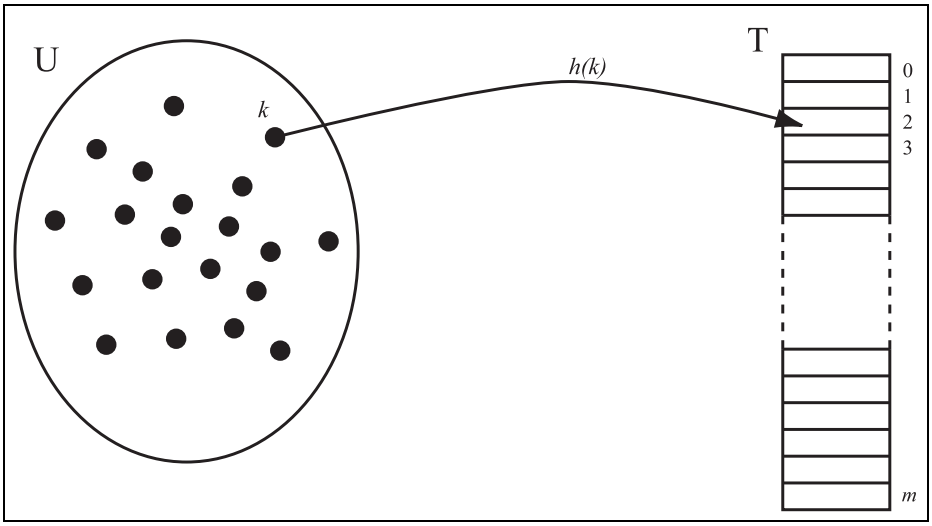

Figura 4.3: Universo di chiavi e tabella hash

4.4 Hash Table

Sono strutture dati che servono per implementare i dizionari, infatti supportano in maniera particolarmente efficiente le operazioni di inserimento, cancellazione e ricerca.

Dato un universo U di chiavi, il problema che le hash table risolvono è quello di mantenerne in memoria un sottoinsieme. Una tabella hash è una generalizzazione del concetto di array, con la differenza che l'indice della tabella viene calcolato a partire dalla chiave k tramite una funzione h, detta funzione di hash.

Sembra che il tempo necessario ad eseguire insert, delete e search sia $O(1)$, sfortunatamente abbiamo due problemi. Il primo è il tempo necessario per calcolare $h(k)$ ma verrà ignorato, mentre il secondo è la presenza di *collisioni*, cioè la presenza di due o più chiavi $k_1, k_2 \in U$ tali che $k_1 \neq k_2$ cui viene assegnato dalla funzione h lo stesso slot, cioè tali che $h(k_1) = h(k_2)$.

Quest'ultimo si presenta sempre per il *pigeon hole principle*:

$$h : U \to \{0, \ldots, m-1\}$$

non può essere iniettiva[1] quando $|U| > m$. Le collisioni quindi non possono essere evitate, soltanto ridotte. Due tecniche particolarmente efficaci sono *chaining* e *open addressing*.

4.4.1 Chaining

Ricorda il bucket sort (vedi paragrafo 2.3), infatti la tabella hash viene rappresentata tramite un array $T[0, \ldots, m-1]$ dove ad ogni posizione, o *slot*,

[1] Sia $\varphi : A \to B$, si dice iniettiva se $\forall\, a, a' \in A \;\; \varphi(a) = \varphi(a') \;\Rightarrow\; a = a'$

corrisponde a una chiave k. In caso di collisioni gli elementi che consumano lo stesso slot vengono gestiti tramite una linked list.

Lo spazio di memorizzazione che la tabella hash richiede è enorme, con conseguente aumento dei costi per le operazioni.

Ad esempio l'inserimento può costare $O(1)$ mentre la cancellazione può richiedere $O(n)$.

Analisi di hashing con chaining

Cerchiamo ora di valutare quanto costa la ricerca per una data chiave in una tabella hash.

Definizione 4.2 (Load factor) *Data una tabella hash T con m slot di cui n occupati indichiamo con α il fattore di carico.*

$$\alpha = \frac{n}{m} \tag{4.1}$$

Esso rappresenta il numero medio di elementi registrati in uno slot.

Nel peggiore dei casi con la tecnica del chaining tutte le n chiavi presenti nella tabella vengono allocate dalla funzione di hash all'interno dello stesso slot, creando di fatto una linked list di dimensione n.

L'immediata conseguenza è che la ricerca di un elemento venga a costare $\Theta(n)$ più il tempo $T(h(k))$ per computare la funzione di hash sulla chiave k.

Andiamo ora ad analizzare il caso medio che come è facile immaginare dipende strettamente dalla qualità della funzione di hash h.

Risulta particolarmente utile vincolare h con un'ipotesi detta *simple uniform hashing*:

Definizione 4.3 (Simple unifor hashing) *Ogni elemento ha la stessa probabilità di occupare un qualunque slot della tabella indipendentemente da dove gli altri elementi sono stati registrati.*

Facciamo poi un'ulteriore assunzione, ovvero che $h(k)$ con k generico sia calcolata in $O(1)$. In questo modo possiamo affermare che il tempo per la ricerca di k è linearmente dipendente da $T(h(k))$.

Proposizione 4.4.1 Il costo per la ricerca di una chiave k con l'ipotesi di simple uniform hashing è

$$\Theta(1 + \alpha)$$

Dimostrazione. Suddividiamo la dimostrazione in due casi: ricerca con insuccesso, e con successo.

- Ricerca con insuccesso. Possiamo dire che il tempo medio di ricerca con insuccesso è pari a quello di scorrere tutta la lista associata ad uno slot. Considerando l'ipotesi di simple uniform hashing il numero medio di elementi registrati in uno slot è pari ad α, se consideriamo anche il tempo necessario

per calcolare $h(k)$ otteniamo che il costo per la ricerca di una chiave k nel caso medio è

$$\Theta(1+\alpha)$$

- Ricerca con successo. La chiave cercata può essere una qualunque delle n chiavi registrate. Assumiamo inoltre che hash insert inserisca gli elementi in coda anzichè in testa. Ora possiamo dire che il numero atteso di controlli effettuati sarà 1 in meno della lunghezza della lista. Per trovare il numero atteso di elementi da esaminare in media tra gli n elementi della tabella consideriamo l'i-esima lista, ricordandoci che contiene all'ultimo posto l'elemento appena inserito. Otteniamo che la lunghezza attesa della lista è: $(i-1)/m$ e così il numero atteso di elementi esaminati è:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\end{aligned}$$

  Il tempo richiesto per una ricerca con successo considerando anche il tempo per la computazione della funzione risulta essere:

$$\Theta\left(2+\frac{\alpha}{2}+\frac{1}{2m}\right)=\Theta(1+\alpha)$$

■

4.4.2 Open addressing

Come si può facilmente notare la tecnica di chaining risolve le collisioni, ma comporta uno spreco di memoria; oltre ad allocare la tabella hash, dobbiamo considerare lo spazio per le liste associate ai vari slot.

Open addressing prevede invece di registrare le chiavi all'interno della tabella T, evitando sprechi ma aprendo la strada alla possibilità che la tabella si riempia. Notiamo quindi che con open addressing il fattore di carico α deve sempre rispettare la relazione

$$\alpha \leq 1$$

Dato che le collisioni non sono eliminabili rimane il problema di come gestirle. L'idea è quella di fornire insieme ad h una strategia per la ricerca di una posizione libera in caso di collisione. Un esempio potrebbe essere quello di provare nello slot successivo fino a che non ne troviamo uno libero, in realtà le posizioni successive alla prima vengono calcolate.

Per permettere tale calcolo dobbiamo ridefinire la funzione di hash h secondo il seguente schema:

$$h : U \times \{0,\ldots,m-1\} \rightarrow \{0,\ldots,m-1\}$$

dove $m = |T|$ rappresenta anche il numero di prove. Con open addressing richiediamo inoltre che la sequenza di prove $\langle h(k,0),\ h(k,1), \ldots,\ h(k,m-1)\rangle$ sia una permutazione di $\langle 0, \ldots, m-1\rangle$.

Algoritmo 4.9 Hash Insert(T,k) procedura di Hash Table

Hash Insert(T,k)

```
1:  i ← 0
2:  repeat
3:     j ← h(k, i)
4:     if  T[j] = NIL   then
5:        T[j] ← k
6:        return j
7:     else
8:        i ← i + 1
9:     end if
10: until i = m
11: error has table full
```

Algoritmo 4.10 Hash Search(T,k) procedura di Hash Table

Hash Search(T,k)

```
1:  i ← 0
2:  repeat
3:     j ← h(k, i)
4:     if  T[j] = k  then
5:        return j
6:     end if
7:     i ← i + 1
8:  until (T[j] = NIL ) ∨ (i = m)
```

L'eliminazione di una chiave dalla tabella risulta più complicata. Non possiamo semplicemente riportare a NIL il contenuto dello slot perché potremmo impedire che la ricerca funzioni correttamente rendendo vera la guardia del repeat until prima del tempo. Una possibile soluzione è di introdurre dei marcatori come DELETED e di modificare di conseguenza l'inserimento e la ricerca.

Prima di affrontare l'analisi di open addressing è necessario formulare un'ipotesi detta *uniform hashing*, estensione della precedente.

Definizione 4.4 (Uniform hasing) *Ogni chiave k considerata ha la stessa possibilità di avere come sequenza di prove una qualunque delle $m!$ possibili permutazioni di $\{0, 1, \ldots, m-1\}$.*

Ci sono principalmente 3 tecniche che permettono di calcolare la corretta

sequenza di prove per open addressing: *linear probing*, *quadratic probing* e *double hashing*; tutte cercano di rispettare l'ipotesi di uniform hashing.

Ricordiamo inoltre che la caratteristica principale di una funzione di hash $h(k, i)$ è che al variare di i per un dato k, $h(k, i)$ copra tutte le possibilità, ovvero:

$$\{h(k,0),\, h(k,1), \ldots,\, h(k,m-1)\} = \{0, \ldots, m-1\} \tag{4.2}$$

Seguono le strategie di risoluzione delle collisioni

Linear probing

Definiamo la funzione di hash $\forall i \in \{0, \ldots, m-1\}$

$$h(k,i) = (h'(k) + i) \bmod m$$

Dato k, il primo slot testato è $T[h'(k)]$ seguono poi in ordine: $T[h'(k) + 1], \ldots, T[m-1]$, $T[0], \ldots, T[h'(k)-1]$ per un totale di m prove.

Si possono avere dei problemi di *primary clustering*: si generano delle catene (clusters) di elementi che occupano porzioni contigue di tabella.

Il problema deriva dal fatto che se un elemento viene mandato da h in un qualunque punto del cluster, questo elemento trova una posizione libera solo dopo averlo percorso tutto, aumentando sensibilmente i tempi medi di ricerca e inserimento.

Quadratic probing

Definiamo la funzione di hash $\forall i \in \{0, \ldots, m-1\}$

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Con questa nuova definizione si nota che il primo slot provato è $T[h'(k)]$ mentre i successivi si trovano ad un offset che varia in maniera quadratica rispetto ad i. Potremmo incorrere in problemi di *secondary clustering*: se partono dallo stesso seme incorriamo nella stessa sequenza di prove in quanto $\forall i \in \{0, \ldots, m-1\}$

$$h(k_1, 0) = h(k_2, 0) \;\Rightarrow\; h(k_1, i) = h(k_2, i)$$

Comunque prima verifichiamo se vale sempre l'uguaglianza 4.2. In effetti non è sempre vero, se poniamo $c_1 = c_2 = 0$ l'identità non viene soddisfatta, comunque è sempre possibile determinare $c_1,\ c_2$ in modo tale che l'uguaglianza sia verificata.

Double Hashing

Costruiamo una funzione di hash come la seguente:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

con h_1, h_2 funzioni di hash ausiliarie.

Risulta essere il migliore, risolve i problemi di primary e secondary clustering in quanto la sequenza di prove dipende in due modi distinti da k e da i.

Ricordiamo che è possibile dimostrare che il tempo medio di ricerca con open addressing e con l'ipotesi di uniform hashing è pari a

$$O\left(\frac{1}{1-\alpha}\right)$$

4.5 Binary Search Trees

Struttura dati che supporta tutte le operazioni su insiemi dinamici. Come è facile immaginare le operazioni base sui BST (*binary search trees*) costano in termini di tempo proporzionalmente all'altezza dell'albero.

Una possibile implementazione di un BST prevede l'uso dei seguenti elementi per un generico nodo x :

- KEY(x): valore chiave del nodo
- $left(x)$: puntatore al figlio sinistro se presente, NIL altrimenti
- $right(x)$: puntatore al figlio destro se presente, NIL altrimenti
- $p(x)$: puntatore al padre di x, NIL se x è la radice

Tutte le chiavi in un BST sono organizzate in modo da rispettare la seguente proprietà:

Definizione 4.5 (BST property) *Dato x nodo in T, con T BST generico, se y appartiene al sottoalbero radicato nel figlio di sinistro di x abbiamo che:*

$$\text{KEY}(y) \leq \text{KEY}(x)$$

Analogamente se y appartiene al sottoalbero radicato nel figlio destro di x abbiamo che:

$$\text{KEY}(y) \geq \text{KEY}(x)$$

I nodi di un BST possono essere visitati secondo tre schemi:

- *in-order* tree walk
- *pre-order* tree walk
- *post-order* tree walk

4.5.1 In order tree walk

Sia x un puntatore alla radice del BST, l'algoritmo 4.11 ne effettua la visita in-order.

Algoritmo 4.11 InOrder(x)

InOrder(x)

```
1: if x ≠ NIL then
2:    InOrder(left(x))
3:    print(KEY(x))
4:    InOrder(right(x))
5: end if
```

Complessità

Sia n il numero di nodi del nostro BST T, possiamo definire per ogni nodo x il numero di nodi del sottoalbero radicato nel figlio sinistro di x con n_l e n_r analogamente per il sottoalbero radicato nel figlio destro.

Dato che la procedura è ricorsiva abbiamo un'equazione di complessità di questo tipo:

$$T(n) = T(n_l) + T(n_r) + O(1)$$

sapendo che $n_l + n_r = n - 1$ scriviamo:

$$\begin{aligned} T(n) &\leq T(n-1) + T(n-1) + O(1) \\ &\leq 2T(n-1) + O(1) \end{aligned}$$

veritiero ma non sufficientemente preciso.

Congetturiamo che $T(n)$ sia lineare e lo dimostriamo induttivamente. Dobbiamo provare che esiste una costante c tale che: $\forall n \geq 1,\ T(n) \leq cn$. Procediamo per induzione.

$n = 1\ \ T(1) = 2 \leq c$

$n \geq 1\ \ T(n) \leq cn$. Questo implica l'assumere che $T(n_l) \leq cn_l$ e $T(n_r) \leq cn_r$ quindi, dall'ipotesi induttiva:

$$\begin{aligned} T(n) &\leq cn_l + cn_r + c \\ &= c\,(n_l + n_r + 1) \\ &= cn \end{aligned}$$

Non è possibile, dati n interi, costruire un BST T in tempo inferiore a $O(n \log n)$. Se ciò fosse possibile una volta costruito T l'output di InOrder(T) sarebbe la lista ordinata degli n interi di partenza ottenendo un assurdo per la dimostrazione del lower bound al problema dell'ordinamento (paragrafo 1.7).

4.5.2 Operazioni

Ricerca

Algoritmo 4.12 TreeSearch(x, k)

TreeSearch(x, k)

```
1: if (x = NIL ) ∨ (k = KEY(x)) then
2:    return x
3: end if
4: if k ≤ KEY(x) then
5:    return TreeSearch(left(x), k)
6: else
7:    return TreeSearch(right(x), k)
8: end if
```

L'approccio usato nell'algortimo 4.12 è chiamato *tail recursive*, è caratterizzato dalla facilità di implementazione in modo iterativo, così facendo si risparmia lo spazio per i record di attivazione relativi alle chiamate ricorsive, pur rimanendo sempre e comunque in-place.

Algoritmo 4.13 ItTreeSearch(x, k)

ItTreeSearch(x, k)

```
1: while (x ≠ NIL ) ∧ (KEY(x) ≠ k) do
2:    if k ≤ KEY(x) then
3:       x ← left(x)
4:    else
5:       x ← right(x)
6:    end if
7: end while
8: return x
```

Sia h l'altezza dell'albero, la complessità di TreeSearch risulta essere:

$$O(h)$$

Risulta fondamentale cercare di mantenere l'albero il più basso possibile, nel caso in cui si sbilanci fortemente ci ricondurrebbe una lista, rendendo inutile l'uso dell'albero stesso.

Minino e Massimo

Vediamo ora le procedure per l'inserimento ed estrazione del massimo e del minimo.

La complessità risulta

$$O(h)$$

Algoritmo 4.14 TreeMin(x)

TreeMin(x)

```
1: while left(x) ≠ NIL do
2:     x ← left(x)
3: end while
4: return x
```

Algoritmo 4.15 TreeMax(x)

TreeMax(x)

```
1: while right(x) ≠ NIL do
2:     x ← right(x)
3: end while
4: return x
```

Successore

Prima di procedere è necessaria la seguente definizione.

Definizione 4.6 (Successore) *Diciamo successore di x un nodo y tale che:*

$$\text{KEY}(y) = \min\{\text{KEY}(z) : \text{KEY}(z) \geq \text{KEY}(x)\}$$

e lo indicheremo con $succ(x)$.

Per semplicità assumeremo che in T non vi siano 2 nodi distinti aventi la stessa chiave.

Sia T un BST e $x \in T$, indichiamo con $T(x)$ il sottoalbero di T radicato in x.

Proposizione 4.5.1 Dato $x \in T$ e $y = succ(x)$[2] possiamo dire che le seguenti proposizioni sono equivalenti:

1. $x \in T(y)$ oppure $y \in T(x)$;
2. se $x \in T(y)$ allora $x = max(T(left(y)))$ e $right(x) = \text{NIL}$;
3. se $y \in T(x)$ allora $y = min(T(right(x))$ e $left(y) = \text{NIL}$;

Dimostrazione.

1. Per assurdo sia $x \notin T(y)$ e $y \notin T(x)$ e z antenato comune ad entrambi di profondità massima. In questo caso $x \in T(left(z))$ e $y \in T(right(z))$ allora $\text{KEY}(x) < \text{KEY}(z) < \text{KEY}(y)$ e quindi $y \neq succ(x)$ quindi assurdo per l'ipotesi fatta.
2. $x \in T(y)$. Poiché $y \geq x$, $x \in T(left(y))$, a questo punto deve necessariamente essere $right(x) = \text{NIL}$ altrimenti x non sarebbe il massimo in $T(left(x))$.

[2] $x \neq max(T)$

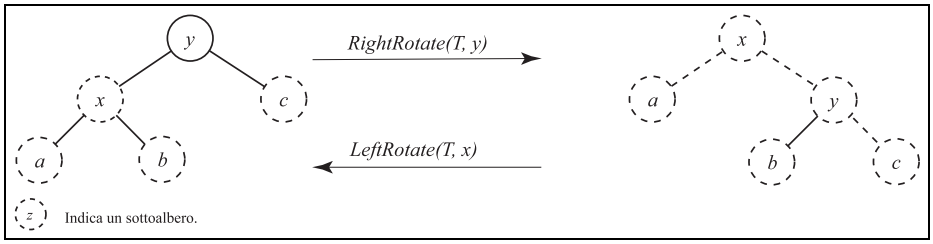

Figura 4.4: Schema di rotazione destra e sinistra.

3. $y \in T(x)$. y è il più piccolo fra gli elementi più grandi di x e quindi $y = min(right(x))$, necessariamente $left(y) =$ NIL altrimenti y non sarebbe il minimo in $right(x)$.

■

Dalla precedente proposizione si evince che se un nodo x ha figli di destra allora il successore di x si ritrova come minimo del sottoalbero di destra di x, altrimenti si continua a risalire da x verso i suoi avi fino a quando i nodi incontrati sono il figlio destro del rispettivo padre, non appena il nodo diviene figlio sinistro considereremo il genitore come successore di x.

L'idetificazione del successore ha complessità $O(h)$ con h altezza del BST.

Algoritmo 4.16 TreeSuccessor(x)

TreeSuccessor(x)

```
1: if right(x) ≠ NIL then
2:     return TreeMin(right(x))
3: end if
4: while (y ≠ NIL ) ∧ (x = right(y)) do
5:     x ← y
6:     y ← p(y)
7: end while
8: return y
```

Rotazioni

Le rotazioni sono operazioni di tipo locale definite sui BST di costo $O(1)$ e si propongono come mezzo per ribilanciare l'albero, pur mantenendo la BST property.

Esistono due diversi tipi di rotazioni chimate *left rotate* e *right rotate*. Uno schema di come agiscono tali rotazioni è riportato in figura 4.4, mentre il codice relativo a left rotate è presente nell'algoritmo 4.17.

Algoritmo 4.17 LeftRotate(T, x)

LeftRotate(T, x)

```
1: y ← right(x)
2: right(x) ← left(y)
3: if left(y) ≠ NIL then
4:    p(left(y)) ← x
5:    p(y) ← p(x)
6:    if p(x) = NIL then
7:       root(x) ← y
8:    else
9:       if x = left(p(x)) then
10:         left(p(x)) ← y
11:      else
12:         right(p(x)) ← y
13:      end if
14:   end if
15: end if
16: left(y) ← x
17: p(x) ← y
```

Proposizione 4.5.2 Dato T BST e $x \in T$, T' ottenuto dopo RightRotate(T, x) (oppure LeftRotate(T, x)) è ancora un BST.

Inserimento e cancellazione

Risulta chiaro che le operazioni di inserimento e cancellazione sono radicalmente diverse da quelle esaminate fino ad ora, esse infatti modificano la struttura del BST sul quale operano. Il vero problema è effettuare entrambe queste operazioni assicurandosi che la BST-property continui a valere.

La complessità dell'inserimento, algoritmo 4.18, risulta ancora una volta $O(h)$, con h altezza del BST, come nel caso della cancellazione, algoritmo 4.19, dove è determinata dalla chiamata a TreeSuccessor.

La cancellazione viene gestita nel codice in tre fasi distinte a seconda del numero di figli del nodo z che intendiamo eliminare:

- z ha 0 figli, il caso è banale perchè possiamo semplicemente eliminare il nodo in questione, la BST property continua a valere;
- z ha 1 figlio, semplicemente sostituiamo a z suo figlio, anche così la BST continua a valere;
- z ha 2 figli, sostituiamo z con il suo successore o predecessore a seconda.

  Dato che $succ(z)$ è il più grande di tutti i nodi di $T(left(z))$ e il $pred(z)$ è il più piccolo di tutti gli elementi di $T(right(z))$ e per la proprietà

precedente $succ(z)$ (o $pred(z)$) non può avere due figli, posso quindi spostare tale elemento al posto di z. La BST property continua a valere.

Algoritmo 4.18 TreeInsert(T, z)

TreeInsert(T, z)

```
 1: y ← NIL
 2: x ← root[T]
 3: while x ≠ NIL do
 4:    y ← x
 5:    if KEY(z) < KEY(x) then
 6:       x ← left(x)
 7:    else
 8:       x ← right(x)
 9:    end if
10: end while
11: p(z) ← y
12: if y = NIL then
13:    root(T) ← z
14: else
15:    if KEY(z) < KEY(y) then
16:       left(y) ← z
17:    else
18:       right(y) ← z
19:    end if
20: end if
```

4.6 Red Black Trees

Abbiamo appena dimostrato che in un albero binario di ricerca di altezza h le operazioni base proprie degli insiemi dinamici come inserimento, estrazione, successore, predecessore,... possano essere effettuate in tempo $O(h)$.

Rimane problematica l'altezza, infatti qualora l'albero risultasse particolarmente sbilanciato, otterremmo prestazioni analoghe alle liste.

Gli alberi binari di ricerca che introduciamo ora, i red-black trees (*rb-tree*), hanno la gradevole caratteristica di essere *bilanciati*, permettendo così di portare a termine le operazioni base in tempo $O(\log n)$ nel caso peggiore.

4.6.1 Proprietà

Un *rb-tree* è un albero binario di ricerca i cui nodi possiedono un ulteriore campo che ne specifica il colore il quale può essere *rosso* oppure *nero*.

Algoritmo 4.19 TreeDelete(T, z)

TreeDelete(T, z)

```
1: if (left(z) = NIL ) ∨ (right(z) = NIL ) then
2:    y ← z
3: else
4:    y ← TreeSuccessor(z)
5: end if
6: if left(y) ≠ NIL then
7:    x ← left(y)
8: else
9:    x ← right(y)
10: end if
11: if x ≠ NIL then
12:    p(x) ← p(y)
13: end if
14: if p[y] = NIL then
15:    root(T) ← x
16: else
17:    if y = left(p(y)) then
18:       left(p(y)) ← x
19:    else
20:       right(p(y)) ← x
21:    end if
22: end if
23: if y ≠ z then
24:    KEY(z) ← KEY(y)
25: end if
26: return y
```

Riassumendo, ogni nodo contiene ora i campi: *color, key, left, right* e *p*. Se un figlio o il padre sono assenti il corrispondente puntatore assume valore NIL . Tutti i nodi NIL che si vengono a formare li chiamiamo *nodi esterni* o foglie.

Regolando il modo in cui i nodi vengono colorati possiamo ottenere che un cammino dalla radice alle foglie sia al massimo lungo il doppio di ogni altro, quindi l'albero risulta quasi bilanciato. Definiamo ora quali sono le proprietà che deve rispettare un BST per essere definito red black tree.

Definizione 4.7 (Red-black property) *Un albero binario di ricerca è un* rb-tree *se soddisfa le seguenti red-black properties:*

1. *ogni nodo è rosso o nero;*
2. *ogni foglia è nera;*
3. *se un nodo è rosso, allora entrambi i suoi figli sono neri;*
4. *ogni cammino da un nodo a una foglia contiene lo stesso numero di nodi neri.*

Definizione 4.8 (Black-height) *Definiamo come altezza nera, $bh(x)$, di un nodo x il numero di nodi neri che si incontrano in un cammino di profondità massima dal nodo x escluso ad una foglia.*

Definiamo analogamente la black-height di un albero T, $bh(T)$, come la black-height della sua radice.

Enunciamo e dimostriamo un lemma che spiega perchè nei red black tree l'operazione di ricerca risulta particolarmente efficiente.

Lemma 4.6.1 *Sia h l'altezza di un rb-tree con n nodi interni, allora vale la seguente relazione*

$$h \leq 2\log(n+1)$$

Dimostrazione. Prima del lemma dobbiamo dimostrare che il sottoalbero radicato in un qualunque nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni. Lo proviamo per induzione sull'altezza h di x:

$h = 0$ x è un figlio, quindi un nodo esterno con $bh(x) = 0$. Questo implica che $T(x)$ ha $2^{bh(0)} - 1 = 0$ nodi interni.

$h = n$ Chiaramente supponiamo $n > 0$ e senza perdita di generalità assumiamo che x sia nodo interno con due figli, ognuno di quali risulta avere in base alle red black properties una black height di $bh(x)$ o $bh(x) - 1$.

Dato che l'altezza di un figlio di x è sicuramente inferiore dell'altezza di x possiamo usare l'ipotesi induttiva e affermare che ogni figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni. Ora possiamo affermare che il sottoalbero centrato in x contiene almeno $\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$ nodi.

A questo punto per completare la dimostrazione dobbiamo solo dire che se h è l'altezza del nostro red black tree T, almeno metà dei nodi di un qualunque cammino dalla radice esclusa alle foglie risultano essere neri. Di conseguenza $bh(T) \geq (h/2$, applicando quanto appena dimostrato otteniamo $n \geq 2^{\frac{h}{2}} - 1$ e riorganizzando:

$$h \leq 2\log(n+1)$$

■

4.6.2 Operazioni

Inserimento

La prima parte dell'inserimento viene eseguita come in un normale BST, quindi su foglie. Abbiamo due opzioni relative al colore del nuovo nodo, rosso oppure nero.

Conviene colorare il nodo nuovo di rosso. Se l'inserimento del nodo rosso è avvenuto sotto una foglia nera, allora abbiamo finito, altrimenti abbiamo due nodi rossi in fila violando la rb-property 3.

Strategia generale per l'inserimento:

- Inseriamo come in binary search tree e coloriamo il nodo di rosso;
- se non si generano due nodi rossi uno figlio dell'altro allora abbiamo finito;
- altrimenti facciamo una analisi per casi con l'obiettivo di spostare il problema dei due nodi rossi in fila verso la radice;
- nel caso peggiore, dopo una quantità logaritmica di rotazioni e ricoloramenti, il problema è alla radice dove viene risolto colorando la radice di nero.

Dopo aver inserito e colorato di rosso il nuovo nodo, possiamo trovarci in una di queste due situazioni:

zio rosso Il caso è riportato in figura 4.5 a). In questo caso coloriamo di nero A e D e di rosso C. L'effetto sarà quello di, eventualmente, aver spostato il problema dei due nodi rossi in fila a C e suo padre. I due casi in cui x è figlio sinistro di A e quello in cui A è figlio destro di C sono perfettamente analoghi.

zio nero Il caso è riportato in figura 4.5 b) e c). Nel primo caso eseguiamo una rotazione a destra e abbiamo finito, mentre nel secondo una a sinistra e come si vede ci riconduciamo al caso precedente.

Algoritmo 4.20 RBTreeInsert(T, x)

RBTreeInsert(T, x)

```
 1: TreeInsert(T, x)
 2: color[x] ← RED
 3: while (x ≠ root(T)) ∧ (color[p(x)] ≠ BLACK) do
 4:    if p(x) = left(p(p(x))) then
 5:       y ← right(p(p(x)))
 6:       if color[y] = RED then
 7:          color[p(x)] ← BLACK
 8:          color[y] ← BLACK
 9:          color[p(p(x))] ← RED
10:          x ← p(p(x))
11:       else
12:          if x = right(p(x)) then
13:             x ← p(x)
14:             LeftRotate(T, x)
15:          end if
16:          color[p(x)] ← BLACK
17:          color[p(p(x))] ← RED
18:          RightRotate(T, p(p(x)))
19:       end if
20:    else
21:       trattiamo i casi simmetrici...
22:    end if
23: end while
24: color[root(T)] ← BLACK
```

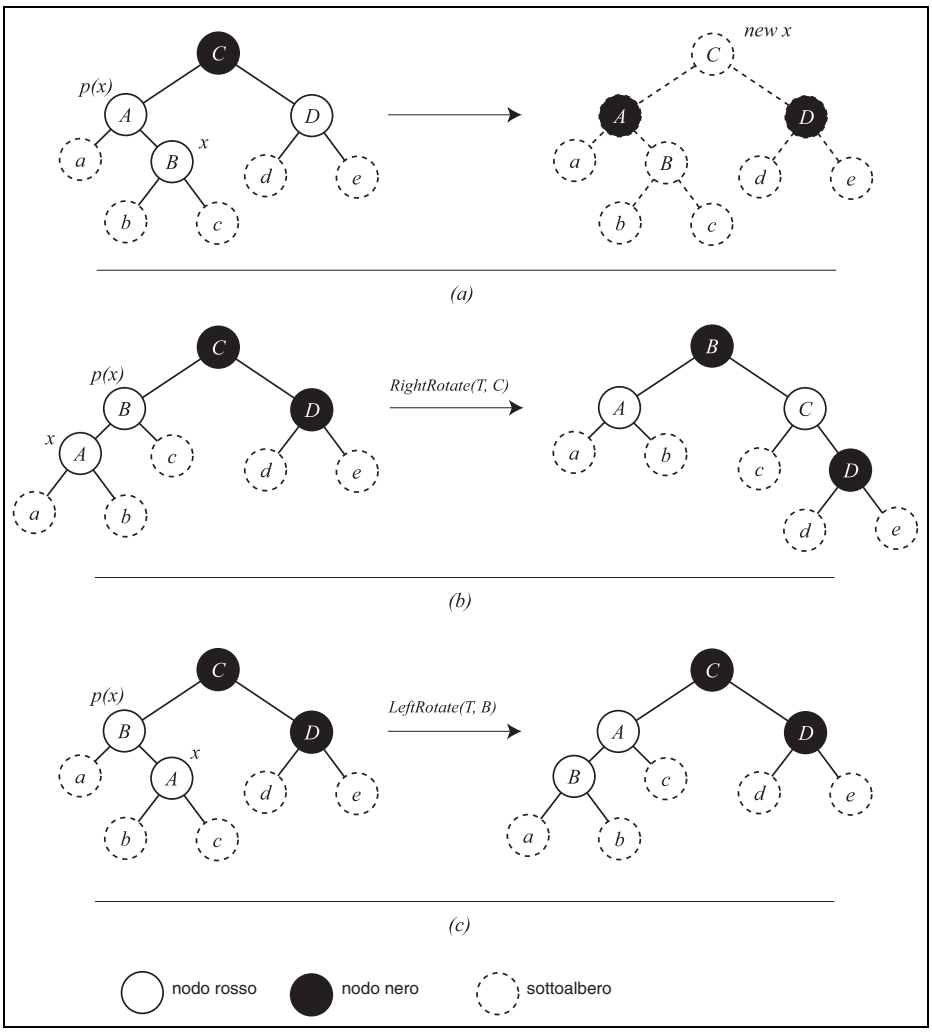

Figura 4.5: Situazioni e azioni da intraprendere dopo un inserimento in un red black tree.

Prima di passare alla cancellazione di un elemento vale la pena analizzare il codice di RBTreeInsert (algoritmo 4.20) per capirlo meglio, vedendo cosa succede istruzione per istruzione, facendo riferimento al numero di linea:

1-2 inseriamo come in un normale BST e coloriamo il nuovo nodo di rosso;

3 entriamo nel ciclo while fino a quando non siamo arrivati alla radice e fino a quando il genitore di x è diverso da nero;

4 controlliamo che il padre di x sia figlio sinistro del nonno di x;

5 facciamo puntare y allo zio di x;

6-10 Controlliamo il colore dello zio. Se risulta essere rosso siamo nel caso riportato in figura 4.5 a): coloriamo il padre di x e x di nero, mentre lo zio di rosso. Facciamo puntare x a suo nonno;

11-19 Se il colore dello zio è nero allora controlliamo se x è figlio destro, se è così facciamo puntare x a suo padre ed eseguiamo una rotazione sinistra sul nuovo x come si vede in figura 4.5 c). Coloriamo poi il padre di x di nero il nonno di rosso ed eseguiamo una rotazione destra sul nonno di x;

20-22 trattiamo i casi simmetrici;

24 coloriamo la radice di nero.

Dopo al massimo $O(\log n)$ operazioni (rotazioni e ricoloramenti) potremmo aver finito, se siamo caduti nel caso zio nero c), oppure abbiamo spinto il problema alla radice, che può essere colorata di nero senza problemi.

Cancellazione

La cancellazione di un elemento nei *rb-tree* avviene inizialmente come nei binary search tree rimane poi il problema di ristabilire le red black property, qualora una o più risultassero violate.

Il codice per cancellare un elemento un red black tree T è l'algoritmo 4.21. Come si nota è uguale alla procedura per eliminare un elemento da un BST (algoritmo 4.19) eccetto per la chiamata a RBDeleteFixUp nel caso in cui il nodo eliminato sia nero. Questa procedura prende in ingresso un RBT T e un suo nodo x in cui la proprietà 4 dei RBT fallisce: a partire da x tutti i cammini verso le foglie hanno un nodo nero in meno; restituisce T modificato in modo che tutte le proprietà siano nuovamente soddisfatte.

Per risolvere il problema di ristabilire la proprietà 4 si attribuisce a x un extracredito, cioè lo considereremo di peso doppio, questo implica che se x è rosso possiamo risolvere colorandolo di nero, mentre se è già nero nel calcolo della black height lo considereremo due volte.

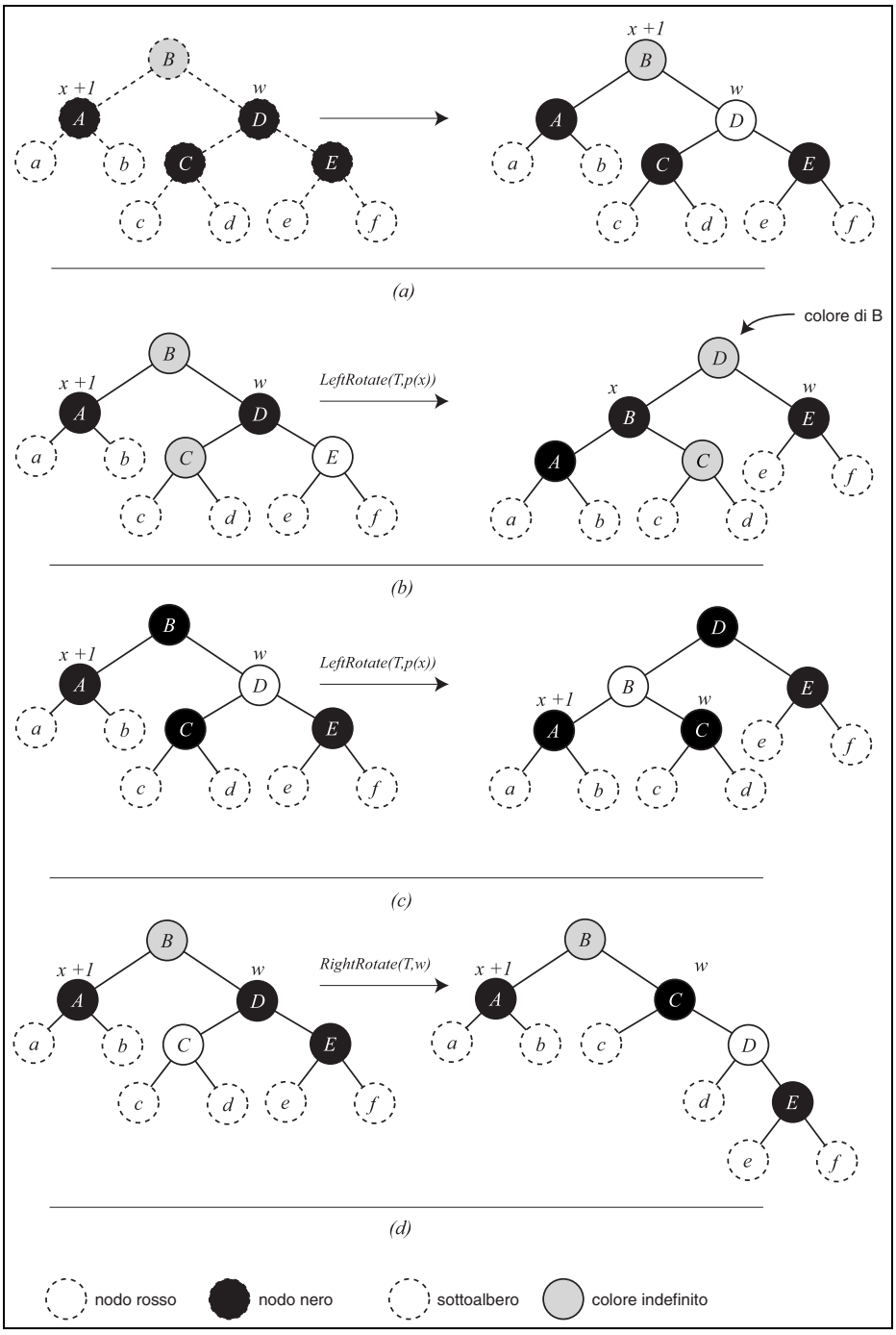

Figura 4.6: Situazioni di RBDeleteFixUp.

RBDeleteFixUp, riportato nell'algoritmo 4.22, cercherà mediante rotazioni e ricoloramenti di spingere l'extra credito verso la radice, dove verrà ignorato. Se nel cammino verso la radice viene ad interessare un nodo rosso esso sarà semplicemente colorato di nero.

Algoritmo 4.21 RBTreeDelete(T, z)

RBTreeDelete(T, z)

```
1: if (left(z) = NIL ) ∨ (right(z) = NIL ) then
2:    y ← z
3: else
4:    y ← TreeSuccessor(z)
5: end if
6: if left(y) ≠ NIL  then
7:    x ← left(y)
8: else
9:    x ← right(y)
10: end if
11: if x ≠ NIL  then
12:    p[x] ← p[y]
13: end if
14: if p(y) = NIL  then
15:    root(T) ← x
16: else
17:    if y = left(p(y)) then
18:       left(p(y)) ← x
19:    else
20:       right(p(y)) ← x
21:    end if
22: end if
23: if y ≠ z then
24:    KEY(z) ← KEY(y)
25: end if
26: if color[x] = BLACK then
27:    RBDeleteFixUp(T, x)
28: end if
29: return y
```

Cerchiamo ora di analizzare con maggiore dettaglio la procedura RBDeleteFixUp, con l'aiuto di quanto riportato in figura 4.6. Ricordiamo comunque che i casi sono riportati in ordine differente dal libro di testo, in particolare a) corrisponde a 2), b) a 4), c) a 1) e d) a 3).

Algoritmo 4.22 RBDeleteFixUp(T, x)

RBDeleteFixUp(T, x)

```
1: while (x ≠ root(T) ∧ (color[x] = BLACK) do
2:    if (x = left(p(x))) then
3:       w ← right(p(x))
4:       if color[w] = RED then
5:          color[w] ← BLACK
6:          color[p(x)] ← RED
7:          LeftRotate(T, p(x))
8:          w ← right(p(x))
9:       end if
10:      if (color[left(w)] = BLACK) ∧ (color[right(w)] = BLACK)  then
11:         color[w] ← RED
12:         x ← p(x)
13:      else
14:         if color[right(w)] = BLACK then
15:            color[left(w)] ← BLACK
16:            color[w] ← RED
17:            RightRotate(T, w)
18:            w ← right(p(x))
19:         end if
20:         color[w] ← color[p(x)]
21:         color[p(x)] ← BLACK
22:         color[right(w)] ← BLACK
23:         LeftRotate(T, p(x))
24:         x ← p(x)
25:      end if
26:   else
27:      trattiamo i casi simmetrici...
28:   end if
29: end while
30: color[x] ← BLACK
```

4.6.3 Applicazioni: Join

L'operazione di join ci permette, dati in input S_1 e S_2 insiemi e un elemento x tale che: $\forall\, x_1 \in S_1,\ \forall\ x_2 \in S_2,\ x_1 \leq x \leq x_2$, di avere come output un nuovo insieme formato da $S_1 \cup \{x\} \cup S_2$.

Per riuscire ad ottenere una complessità logaritmica per l'operazione di join, rappresentiamo i due insiemi mediante due rb-tree che indichiamo con S_1 e S_2. Supponiamo S_1 più alto di S_2, allora esiste sicuramente in S_1 un elemento che è alla stessa altezza di S_2. Se potessimo inserire S_2 a tale livello avremmo finito, sarebbe sufficiente colorare x di rosso e nel caso in cui $p(x)$ fosse rosso opereremmo come nel caso di RBInsert.

Vediamo ora i passi per eseguire l'operazione di join in tempo logaritmico.

- manteniamo un campo $bh(x)$ per ogni nodo x modificando le operazioni di inserimento e cancellazione in modo da avere il valore corretto;
- assumendo $bh(S_1) \geq bh(S_2)$ determiniamo il nodo y nei rami di destra di S_1 tale che $bh(y) = bh(S_2)$;
- inseriamo x al posto di y e rendiamo il sottoaltero di S_1 radicato in y figlio sinistro di x e S_2 figlio destro
- coloriamo x di rosso e operiamo come in RB-Insert.

Il costo risulta:

$$O(h(S_1) - h(S_2)) = O(\log n)$$

4.7 B-Tree

I B-Tree[3] sono alberi di ricerca bilanciati che sembrano essere particolarmente adatti per lavorare con ingenti quantità di dati, memoria di massa, essi infatti minimizzano il tempo di calcolo e il numero di accessi alla memoria secondaria.

Definizione 4.9 (B-Albero) *Un B-Albero di grado (minimo) $t \geq 2$ sarà un albero T tale che:*

- *Ogni nodo x in T ha i seguenti campi:*
  - ✓ *$n[x]$ numero di chiavi memorizzate in x;*
  - ✓ *$n[x]$ campi contenenti le chiavi ordinatamente:*

$$\text{KEY}_1(x) \leq \text{KEY}_2(x) \leq \ldots \leq \text{KEY}_{n[x]}(x)$$

  - ✓ *$leaf(x)$ ritorna un valore booleano pari a* TRUE *se x è una foglia,* FALSE *altrimenti.*

[3] La B indica *balanced* non binary...

- *Se x è un nodo interno allora contiene anche $n[x]+1$ puntatori ad altri nodi che vengono denotati con:*

$$c_1[x], \ldots, c_{n[x]+1}[x]$$

- *Se k_i è una chiave contenuta nel nodo cui punta $c_i[x]$ avremo che:*

$$k_i \leq k_i[x] \leq k_{i+1}$$

- *Ogni foglia ha la stessa profondità*

- *Ogni nodo diverso dalla radice ha almeno $t-1$ chiavi ed al più $2t-1$ chiavi, la radice ha al più $2t-1$ chiavi.*

Si deve avere $t \geq 2$ per evitare che si formino alberi sbilanciati che avrebbero tempo lineare invece di logaritmico.

Teorema 4.7.1 *Dato un B-Albero di grado t, altezza h e contenente n chiavi si ha:*

$$h \leq \log_t \frac{n+1}{2}$$

ovvero:

$$h = O(\log n)$$

Dimostrazione. Se un B-Tree ha altezza h il numero dei suoi nodi è minimizzato quando la radice ha una chiave e gli altri nodi hanno tutti $t-1$ chiavi. In questo caso si hanno 2 nodi ad altezza 1, $2t$ nodi di altezza 2,..., $2t^{h-1}$ nodi di altezza h. Quindi il numero di chiavi soddisfa le equazioni:

$$\begin{aligned}
h = 1 &\Rightarrow n \geq 2(t-1)+1 \\
h = 2 &\Rightarrow n \geq (2t+2)(t-1)+1 \\
&\vdots \\
&\quad n \geq (2t^{t-1} + \ldots + 2t + 2)(t-1) + 1
\end{aligned}$$

Abbiamo quindi:

$$\begin{aligned}
n &\geq (2t^{t-1} + \ldots + 2t + 2)(t-1) + 1 \\
&= 2(t^{h-1} + \ldots + t + 1)(t-1) + 1 \\
&= 2\frac{t^h - 1}{t-1}(t-1) + 1 \\
&= 2t^h - 1 \\
&\Leftrightarrow n \geq 2t^h - 1 \\
&\Leftrightarrow \frac{n+1}{2} \geq t^h \\
&\Leftrightarrow \log_t \frac{n+1}{2} \geq h
\end{aligned}$$

■

Si può dimostrare per induzione sull'altezza h che a profondità $i \in \{1 \dots h\}$ ci sono $2t^{i-1}$ nodi:

$i = 1$ ci sono 2 nodi

$i > 1$ per ipotesi induttiva ci sono $2t^{i-2}$ nodi di profondità $i-1$, poiché per ogni nodo di profondità $i-1$ devono esserci t figli avremo $2^{i-2}t = 2t^{i-1}$ nodi di profondità i.

Quindi globalmente otteniamo $\sum_{i=1}^{h} 2t^{i-1} = 2\sum_{i=1}^{h} t^{i-1} = 2\frac{t^h-1}{t-1}$ nodi poiché ogni nodo contiene almeno $t-1$ chiavi e una per la radice avremo $n \geq 2\frac{t^h-1}{t-1}(t-1) - 1$ e come prima: $h \leq \log_t \frac{n+1}{2}$.

4.7.1 Operazioni

Ricerca

L'operazione di ricerca prevede in input un puntatore x alla radice di un b-albero e una chiave da ricercare k. Come output ci attendiamo un puntatore y ad un nodo ed un indice i tale che: $1 \leq i \leq n[y]$ e $\text{KEY}_i(y) = k$ oppure NIL .

Algoritmo 4.23 BTreeSearch(x, k)

BTreeSearch(x, k)

```
1: i ← 1
2: while (i ≤ n[x]) ∧ (KEY_i(x) < k) do
3:    i ← i + 1
4: end while
5: if (i ≤ n[x]) ∧ (k = KEY_i(x)) then
6:    return(x, i)
7: end if
8: if leaf(x) then
9:    return NIL
10: else
11:    DiskRead(c_i[x])
12:    return BTreeSearch(c_i[k], k)
13: end if
```

La procedura ricorsiva BTreeSearch, algoritmo 4.23, è una semplice generalizzazione della procedura Tree Search definita per gli alberi binari di ricerca.

Nelle linee 1-4 troviamo il più piccolo i tale che $k \leq \text{KEY}_i(x)$. Alla linea 5 controlliamo se abbiamo trovato la chiave cercata nel qual caso termini-

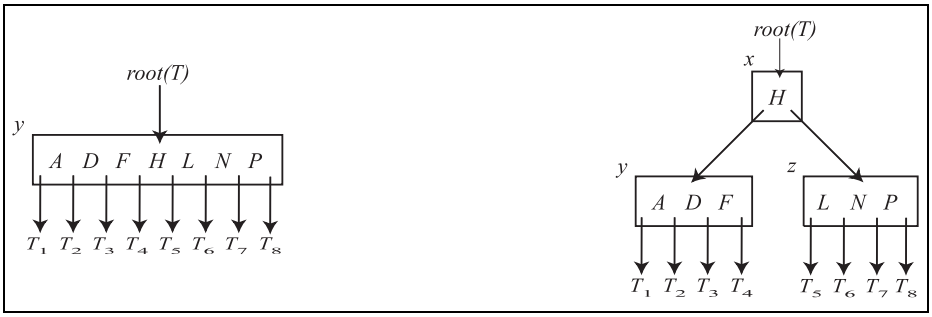

Figura 4.7: Schema di BTreeSplit.

amo, altrimenti richiamiamo la procedura sul sottoalbero identificato dal puntatore $c_i[x]$.

Il numero di pagine a cui accede BTreeSearch è $O(h) = O(\log_t n)$, considerando che $n[x] \leq 2t$ il costo del while-loop è $O(t)$ per ogni nodo, per un totale di

$$O(th) = O(t \log_t n)$$

Creazione

Per costruire un BTree T utilizziamo prima la procedura BTreeCreate per costruire la radice e poi sfruttiamo BTreeInsert per aggiungere nuove chiavi. Entrambe le procedure citate utilizzano una sottoprocedura chiamata AllocateNode allo scopo di allocare in memoria lo spazio necessario per un nodo del BTree, assumendo che AllocateNode non richieda accessi alla memoria di massa possiamo sostenere che BTreeCreate costa $O(1)$ in termini di tempo e $O(1)$ in termini di spazio.

Algoritmo 4.24 BTreeCreate(T)

BTreeCreate(T)

1: $x \leftarrow$ AllocateNode()
2: $leaf(x) \leftarrow$ TRUE
3: $n[x] \leftarrow 0$
4: DiskWrite(x)
5: root$[T] \leftarrow x$

Split di nodi pieni

L'inserimento di una chiave in un b-albero è più complesso che in un normale albero binario di ricerca. Una delle operazioni fondamentali usate durante l'inserimento è lo *split* di un nodo pieno contenente $2t-1$ chiavi, in due nodi aventi $t-1$ chiavi ciascuno. La divisione viene effettuata in base all'elemento medio del nodo, $\text{KEY}_t(y)$, che fungerà da padre per i due nuovi nodi.

In BTreeSplit(x, i, y) (algoritmo 4.25) supponiamo che y sia l'i-esimo figlio di x. Per ipotesi y è un nodo pieno, cioè possiede $2t - 1$ figli. Il nodo z adotta $t - 1$ figli di y e diviene poi nuovo figlio di x posizionato appena dopo y. La chiave mediana di y sale a x per dividere y e z. La procedura BTreeSplit ha complessità $\Theta(t)$. Vale la pena sottolineare che se y non ha genitore allora stiamo facendo lo split della radice, facendo aumentare l'altezza dell'albero di 1.

La procedura BTreeSplit prende in input un nodo interno non pieno x, un indice i e un nodo pieno y tale che $y = c_i[x]$. Lo schema di come agisce BTreeSplit è evidenziato in figura 4.7. Il costo di ogni split è $\Theta(t)$.

Algoritmo 4.25 BTreeSplit(x, i, y)

BTreeSplit(x, i, y)

```
 1: z ← AllocateNode()
 2: leaf(z) ← leaf(y)
 3: n[z] ← t − 1
 4: for j ← 1 to t − 1 do
 5:    KEY_j(z) ← KEY_{j+t}(y)
 6: end for
 7: if ¬(leaf(y)) then
 8:    for j ← 1 to t do
 9:       c_j[z] ← c_{j+t}[y]
10:    end for
11: end if
12: n[y] ← t − 1
13: for j ← n[x] + 1 downto i + 1 do
14:    c_{j+1}[x] ← c_j[x]
15: end for
16: c_{i+1}[x] ← z
17: for j ← n[x] downto i do
18:    KEY_{j+1}(x) ← KEY_j(x)
19: end for
20: KEY_i(x) ← KEY_t(y)
21: n[x] ← n[x] + 1
22: DiskWrite(y); DiskWrite(x); DiskWrite(z)
```

Inserimento

Al fine di garantire un corretto inserimento dovremmo fare tutti gli split dei nodi pieni che troviamo lungo il cammino che ci porta al nodo in cui dobbiamo inserire la nuova chiave.

Probabilmente faremo degli split in eccesso, comunque in numero costante non variando la complessità asintotica e favorendo una miglior distribuzione

di chiavi nell'albero. Lo split aumenta l'altezza dell'albero ma dimagrisce i nodi, portandoli ad avere t chiavi ciascuno. Arrivati al termine della ricerca saremo su una foglia non piena e quindi sarà possibile inserire.

L'inserimento richiede quindi di percorrere una sola volta l'albero per tutta la sua altezza con un numero di $O(h)$ accessi al disco ed un costo in termini di tempo di CPU pari a

$$O(th) = O(t \log_t n)$$

Algoritmo 4.26 BTreeInsert(T, k)

BTreeInsert(T, k)

```
1:  r ← root(T)
2:  if n[r] = 2t − 1 then
3:      s ← AllocateNode()
4:      root(T) ← s
5:      n[s] ← 0
6:      c_1[s] ← r
7:      BTreeSplit(s, 1, r)
8:      BTreeInsertNonFull(s, k)
9:  else
10:     BTreeInsertNonFull(r, k)
11: end if
```

Nella procedura BTreeInsert controlliamo prima di tutto che la radice r non sia un nodo pieno, nel qual caso r viene diviso in due nodi ed s diviene la nuova radice. Procediamo poi usando la procedura BTreeInsertNonFull, algoritmo 4.27, che traccia in modo ricorsivo il cammino verso la foglia nella quale andremo ad inserire, ed effettuando lo split dei nodi pieni che via via incontra.

Cancellazione

In base al nodo che dobbiamo cancellare distinguiamo due casi a seconda che sia una foglia o un nodo interno. Se è una foglia la cancellazione è banale, se è un nodo interno allora operiamo come nei BST, eseguiamo la cancellazione mediante sostituzione della chiave da cacellare con il predecessore/successore. Chiaramente il predecessore/successore andrà a sua volta cancellato e procederemo ricorsivamente.

Risulta fondamentale garantire che la chiamata all'operazione di cancellazione avvenga sempre su di un nodo con almeno t chiavi. Il codice di BTreeDelete(x, k) gestisce quindi la cancellazione per casi:

1. (banale) x punta ad una foglia con almeno t chiavi, allora cancelliamo semplicemente k;

Algoritmo 4.27 BTreeInsertNonFull(x, k)

BTreeInsertNonFull(x, k)

```
1: i ← n[x]
2: if leaf(x) then
3:    while (i ≥ 1) ∧ (k < KEY_i(x)) do
4:       KEY_{i+1}(x) ← KEY_i(x)
5:       i ← i − 1
6:    end while
7:    KEY_{i+1}(x) ← k
8:    n[x] ← n[x] + 1
9:    DiskWrite(x)
10: else
11:   while (i ≥ 1) ∧ (k < KEY_i(x)) do
12:      i ← i − 1
13:   end while
14:   i ← i + 1
15:   DiskRead(c_i[x])
16:   if n[c_i[x]] = 2t − 1 then
17:      BTreeSplit(x, i, c_i[x])
18:      if k > KEY_i(x) then
19:         i ← i + 1
20:      end if
21:   end if
22:   BTreeInsertNonFull(c_i[x], k)
23: end if
```

2. $k \in x$ ed x ha almeno t chiavi, ci sono tre sottocasi:
   (a) il nodo figlio y che precede k ha almeno t chiavi, allora troviamo il predecessore k' di k, cancelliamo k' con una chiamata a BTreeDelete(y, k') e poniamo k' di k in x;
   (b) il nodo z che segue k ha almeno t chiavi, allora procedo come nel caso precedente scambiando il predessore con il successore;
   (c) se abbiamo che sia y che z hanno meno di t chiavi costruiamo un nodo pieno fondendo y, k e z, possiamo quindi eliminare k da x eseguendo una chiamata ricorsiva a BTreeDelete;
3. $k \notin x$, dobbiamo garantire che quando eseguiamo le chiamate ricorsive lo facciamo su di un nodo con almeno t chiavi. Se in $c_i[x]$ ci sono almeno t chiavi allora eseguo la chiamata ricorsiva. Abbiamo tre sottocasi:
   (a) il fratello di destra di $c_i[x]$ ha almeno t chiavi
   (b) il fratello di sinistra di $c_i[x]$ ha al più $t-1$ chiavi allora è analogo al precedente;
   (c) ne il fratello di sinistra ne quello di destra hanno almeno t chiavi.

La complessità è $O(h)$ e si dimostra osservando che:

- in ognuno dei casi meno il caso 2.a) eseguiamo una quantità costante di operazioni;
- BTreeDelete viene chiamata al più una volta per livello;
- Nel caso 2.a) (che prende tempo $O(h)$) possiamo finire al più una volta.

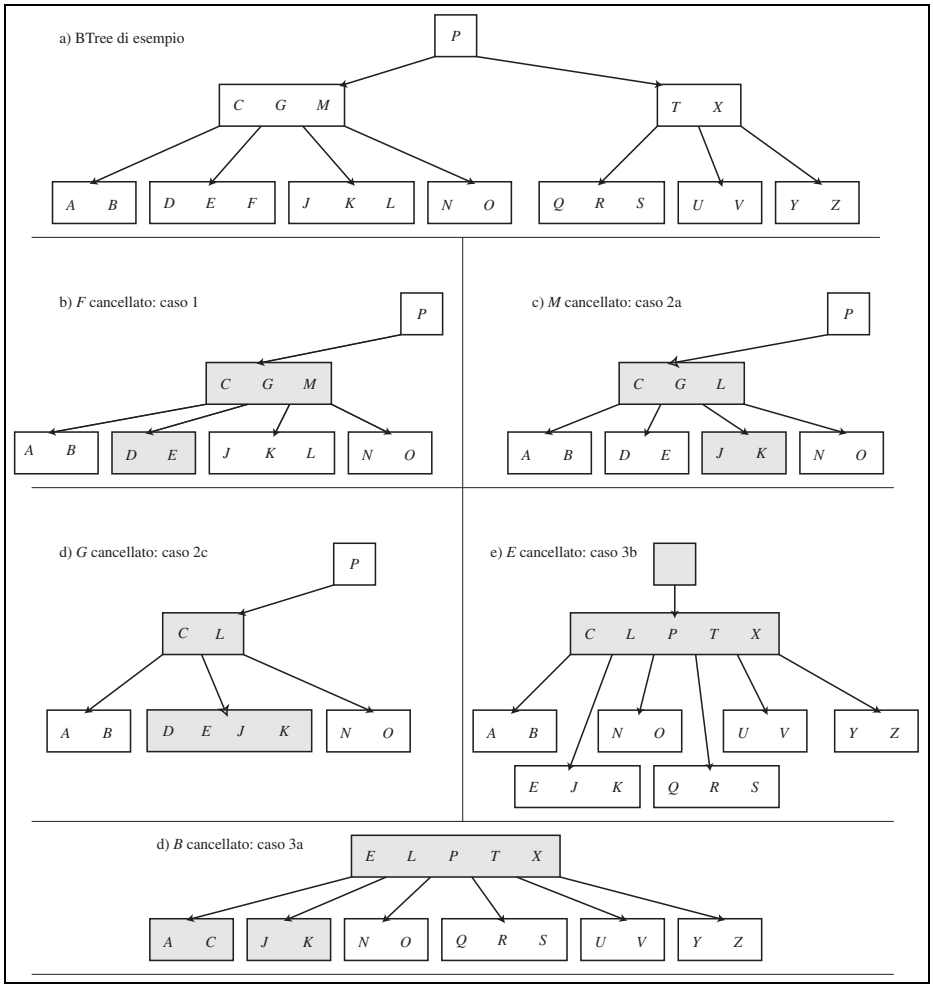

Figura 4.8: Casi di BTreeDelete.

4.8 Esercizi

Esercizio 4.1 *(tratto dall'esame di ASD del 09-12-03)*

Si consideri il problema di definire delle funzioni di hash che mappano gli elementi dell'universo $\mathtt{U} = \{1 \dots N\}$ nelle celle di una tabella di hash $\mathtt{T}[0 \dots m]$. Dopo aver elencato le caratteristiche di una buona funzione di hash, per ognuna delle seguenti funzioni h_i, si dica se h_i può essere considerata una ragionevole funzione di hash, giustificando la risposta.

- $h_1(i) = (i \mathrm{mod} m) + 3$
- $h_2(i) = (2 * i) \mathrm{mod} m$
- $h_3(i) = (max(N - m, i)) \mathrm{mod} m$
- $h_4(i) = ((i \mathrm{mod} m) + 127) \mathrm{mod} m$

Esercizio 4.2 *(tratto dall'esame di ASD del 02-09-03)*

Dato un array ordinato contenente n interi, $A[1 \dots n]$, si consideri il problema di costruire un *Red Black Tree* contenente $A[1], \dots, A[n]$ come chiavi. Si proponga un algoritmo efficiente per il problema proposto e se ne dimostri la correttezza. Si valuti infine la complessitá della procedura proposta.

Esercizio 4.3 *(tratto dall'esame di ASD del 26-09-02)*

Siano B_1 e B_2 due B-alberi di grado t, tali che per ogni chiave k_1 in B_1 e k_2 in B_2 si abbia

$$k_1 \leq k_2.$$

a Si descriva un algoritmo efficiente che, prendendo in input B_1 e B_2, esegua la *fusione* dei due B-alberi e ritorni in output un B-albero B contenente tutte le chiavi precedentemente in B_1 e B_2 (che vengono eliminati).

b Si determini la complessità computazionale dell'algoritmo proposto valutando, in particolare, il numero massimo di accessi alla memoria secondaria nell'ipotesi solo un numero costante di nodi di un B-albero di grado t possa risiedere contemporaneamente in memoria centrale.

Esercizio 4.4 *(tratto dall'esame di ASD del 10-09-01)*

Si consideri un Binary Search Tree T contenente $n/2$ elementi.

Si descriva un algoritmo che, dati altri $n/2$ elementi, li inserisca in T e, successivamente, produca la lista ordinata degli elementi di T. Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 4.5 *(tratto dall'esame di ASD del 13-07-01)*

Dato un B-albero T di grado 2 e altezza h.

a Si proponga un algoritmo che costruisca un RB-albero T' contenente le stesse chiavi di T e con altezza nera $h + 1$.

b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

c Si dimostri o si refuti l'affermazione: *Ogni RB-albero T' può essere ottenuto quale output del programma proposto a partire da un opportuno B-albero T.*

Esercizio 4.6 *(tratto dal compitino di ASD del 01-06-01)*

Siano S' ed S'' due generici insiemi dinamici rappresentati mediante due B-alberi T' e T''. Sia $x \notin S' \cup S''$ tale che se $x' \in S'$ e $x'' \in S''$, allora $key[x'] < key[x] < key[x'']$. Si consideri l'operazione *join* che, dati in input T', T'' e x, restituisce il B-albero T rappresentante l'insieme $S = S' \cup \{x\} \cup S''$.

Si proponga un algoritmo che implementi *join* e si argomenti la correttezza e la complessità della procedura proposta.

Esercizio 4.7 *(tratto dal compitino di ASD del 30-03-01)*

Si scriva lo pseudo-codice di un algoritmo che ricevuto in ingresso un albero binario B, un colore c ($\in \{Red, Black\}$) ed un numero naturale h, determina se B può essere decorato in maniera tale da diventare un RB-tree con radice di colore c ed altezza nera h. Si valuti la complessità dell'algoritmo proposto.

Suggerimento Un algoritmo efficiente può essere ottenuto valutando ricorsivamente per ogni sottoalbero A la lista di tutte le possibili coppie (c, h) tali che A possa diventare un RB-tree con radice di colore c e altezza nera h.

Esercizio 4.8 *(tratto dall'esame di ASD del 13-07-00)*

Sia T un albero binario di ricerca privo di nodi aventi la stessa chiave ($key[\cdot]$),

a Si proponga un algoritmo efficiente che dati due nodi a e b in T ritorni la lista dei nodi x tali che $key[a] \leq key[x] \leq key[b]$.

b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

c Si modifichi l'algoritmo di cui sopra in modo che ritorni il nodo avente chiave mediana fra le chiavi relative ai nodi della lista ritornata in output.

Esercizio 4.9 *(tratto dall'esame di ASD del 22-06-00)*

Si consideri una struttura dati S che gestisca un insieme di numeri naturali e implementi le tre seguenti operazioni:

- `Insert(n)`, inserisce il numero naturale `n` all'interno della struttura dati S,
- `Extract_min`, estrae l'elemmento minimo da S
- `Extract_min_even`, estrae da S il più piccolo numero pari.

a Si proponga un'implementazione per la struttura dati S.

b Si analizzi la complessità degli algoritmi proposti e se ne motivi la correttezza.

c Si riconsiderino i due punti precedenti nel caso in cui S implementi anche l'operazione `Add(i)` che somma il numero naturale `i` ad ogni elemento contenuto in S. Si cerchi di proporre un implementazione in cui `Add(i)` abbia costo costante.

Esercizio 4.10 *(tratto dal compitino di ASD del 01-06-00)*

Si consideri un *binary search tree* T.

Si proponga un algoritmo che determini se è possibile "colorare" i nodi di T in modo da renderlo un *red-black tree*. Si proponga un algoritmo che dato in input T lo trasformi in un red-black tree. Si provino correttezza e complessità delle procedure proposte.

Esercizio 4.11 *(tratto dall'esame di ASD del 14-2-00)*

In una sequenze di n elementi si definisce *elemento medio* l'elemento nella posizione $\lfloor n/2 \rfloor$.

(a) Proporre una struttura dati che permetta di realizziare, in maniera efficiente, le seguenti operazioni:

- `Insert(a)`: Inserisce un nuovo elemento all'interno della struttura
- `Extract_Min`: Estrae l'elemento con chiave minima.
- `Find_Med`: Trova (ma non estrae) l'elmento medio rispetto all'ordine sulle chiavi.

(b) Si discuta la correttezza degli algoritmi proposti e se ne valuti la complessità.

(Suggerimento: Una possibile soluzione consiste nel far uso di alberi bilanciati).

Esercizio 4.12 *(tratto dall'esame di ASD del 24-1-00)*

Si consideri un B-albero T con di grado t con n chiavi.

Si proponga lo pseudo-codice di un algoritmo per l'inserimento di una chiave k in T che *minimizzi il numero di nuovi nodi generati* (**suggerimento**: si consideri una variante in due passi dell'algoritmo di inserimento visto a lezione).

Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto in termini dei parametri t e k.

Esercizio 4.13 *(tratto dall'esame di ASD del 30-9-99)*

Sia S una stuttura dati, formata da un insieme di numeri naturali, su cui sia possibile eseguire le seguenti tre operazioni.

- Insert(n), inserisce il numero naturale n all'interno della struttura dati,
- Extract(k), estrae dalla struttura dati il più piccolo multiplo di k in essa contenuto.
- Delete(k), rimuove dalla struttura dati tutti i multipli di k contenuti in essa.

Si supponga inoltre che il valori di k sia limitato da una costante c.

a Si proponga una implementatizione per la struttura dati S.

b Si analizzi la complessita' degli algoritmi proposti e se ne discuta la correttezza.

Esercizio 4.14 *(tratto dall'esame di ASD del 8-7-99)*

Dato un RB-albero, si consideri il problema di mantenere un campo *black-height[x]* che, per ogni nodo x appartenente all'albero, contenga l'altezza nera di x.

(a) Si proponga lo pseudocodice commentato e dettagliato di opportune versioni delle procedure RB-TREE-INSERT e RB-TREE-DELETE atte a garantire il mantenimento del campo *black-height*;

(b) si discuta la correttezza e si valutino le complessità delle procedure proposte;

(c) si discuta brevemente il problema di mantenere un campo *height* che contenga l'altezza.

Esercizio 4.15 *(tratto dal compitino di ASD del 15-6-99)*

Dato un B-albero di grado t si consideri il problema di mantenere un campo *height[x]* che, per ogni nodo x appartenente all'albero, contenga l'altezza di x.

(a) Si proponga lo pseudocodice commentato e dettagliato di opportune versioni delle procedure B-TREE-INSERT e B-TREE-DELETE atte a garantire il mantenimento del campo *height*;

(b) si discuta la correttezza e si valutino le complessità delle procedure proposte.

Esercizio 4.16 *(tratto dal compitino di ASD del 13-5-99)*

Definiamo LRB-tree (*Loose* Red-Black tree) come un albero binario di ricerca in cui:

i) tutti i nodi sono etichettati come neri o rossi,

ii) ogni foglia (*nil*) è nera,

iii) se un nodo x è rosso ed il padre di x è rosso allora entrambi i figli di x sono neri,

iv) ogni cammino semplice dalla radice ad una foglia contiene lo stesso numero di nodi neri.

(a) Si mostri un LRB-tree che non sia un RB-tree.

(b) Qual e' l'altezza massima di un LRB-tree con n nodi?

(c) Se si applica l'algoritmo RB-Insert ad un LRB-Tree l'albero risultate e' un LRB-tree?

Esercizio 4.17 *(tratto dall'esame di ASD del 30-6-98)*

Dato un albero k-ario τ (generalizzazione della nozione di albero binario in cui ogni nodo interno ha k, anzichè 2, figli) perfettaente bilanciato e un nodo interno $x \in \tau$, siano $\downarrow (x, 0), ..., \downarrow (x, k-1)$ rispettivamente il primo, il secondo, ... il k-esimo figlio di x.

Sull'insieme dei nodi di τ sia definita una relazione di ordinamento totale $<$ nel seguente modo: $<$ è la chiusura transitiva della relazione $\prec$, tale che:

i per ogni nodo interno x, $\downarrow (x, 0) \prec x$ e $x \downarrow (x, j)$, con $j = 1, ..., k-1$;

ii per ogni nodo interno x, $\downarrow (x, j) \prec \downarrow (x, j+1)$, per $j = 1, ..., k-2$;

iii dati $x, x' \in \tau$, se $x \prec x'$ e x' non appartiene al sottoalbero radicato in x, allora $\downarrow (x, k-1) \prec x'$;

iv dati $x, x' \in \tau$, se $x \prec x'$ e x' non appartiene al sottoalbero radicato in x', allora $x \prec \downarrow (x', 0)$.

Si dimostri la verità o la falsità delle seguenti proposizioni (in caso di falsità si produca un controesempio):

a dati $x, x' \in \tau$, se profondità(x) = profondità(x') e $x < x'$, allora per ogni nodo y appartenente al sottoalbero radicato in x, e per ogni nodo y' appartenente al sottoalbero radicato in x', $y < y'$;

b per ogni nodo interno x è vero che, per ogni nodo appartenente al sottoalbero radicato in $\downarrow (x, 0)$, $y < x$ e che, per ogni nodo y' appartenente al sottoalbero radicato in $\downarrow (x, 1)$, $x < y'$;

c (generalizzazione di 1a) dati $x, x' \in \tau$, se $x < x'$ allora per ogni nodo y appartenente al sottoalbero radicato in x e per ogni nodo y' apartenente al sottoalbero radicato in x', $y < y'$.

Esercizio 4.18 *(tratto dal compitino di ASD del 27-2-98)*

Si considerino un insieme S di numeri naturali e le seguenti operazioni:

1. $extract\_min(S)$ che estrae (cancellandolo) l'elemento minimo da S;
2. $insert(x, S)$ che inserisce l'elemento x in S;
3. $print\_log\_sequence(S)$ che stampa una sequenza ordinata di $\log(|S|)$ elementi di S tale che il primo elemento della sequenza sia il minimo in S e l'ultimo sia il massimo in S.

(a) Si proponga una struttura dati atta a memorizzare S ed a supportare efficientemente le operazioni sopra elencate.

(b) Si provi la correttezza e si determini la complessità degli algoritmi proposti per supportare le operazioni considerate.

(c) Si discuta il problema di supportare anche l'operazione di estrazione del massimo.

Esercizio 4.19 *(tratto dall'esame di ASD del 18-9-97)*

Si consideri un cammino dalla radice ad una delle foglie in un albero binario di ricerca. Siano A l'insieme delle chiavi associate ai nodi alla sinistra del cammino, B l'insieme delle chiavi associate ai nodi appartenenti al cammino, C l'insieme delle chiavi associate ai nodi alla destra del cammino. Sia inoltre x la chiave associata alla radice.

Si dimostri la verità o falsità delle seguenti proposizioni (in caso di falsità, si produca un controesempio):

1. $\forall y, z((y \in B \wedge z \in C) \rightarrow y \leq z)$;
2. $\forall y(y \in A \rightarrow y \leq x) \vee \forall y(y \in C \rightarrow y \geq x)$;
3. $\forall y \in A \forall z \in C(y \leq z)$.

Esercizio 4.20 *(tratto dall'esame di ASD del 2-9-96)*

Si dica albero binario un albero in cui ogni nodo ha $0, 1$ o 2 figli, si dica inoltre albero binario *perfetto* un albero in cui ogni nodo ha 0 o 2 figli.

- Si dimostri che il numero di foglie f di un albero binario perfetto T è uguale ad $i + 1$, con i numero di nodi interni in T.
- Si proponga lo pseudo-codice commentato di un algoritmo che, dato un albero binario di ricerca, lo trasformi in un albero binario perfetto di ricerca, se questo è possibile, oppure ritorni NIL.
- Si provi la correttezza e si valuti la complessità dell'algoritmo proposto al punto precedente.

Esercizio 4.21 *(tratto dall'esame di ASD del 24-6-96)*

Sia T_{rb} un red-black tree di altezza h.

- Si disegni e si valuti la complessità di un algoritmo che trasformi T_{rb} in un B-albero T_B di grado 2 e di altezza $\Theta(h)$.
- Si dimostri che T_B è un B-albero.

Esercizio 4.22 *(tratto dall'esame di ASD del 24-6-96)*

Si consideri un albero binario di ricerca T e un nodo x in T con chiave associata $key[x]$.

- Scrivere lo pseudo-codice e dimostrare la correttezza di una procedura MODIFY-KEY(T, x, key) che modifica x assegnando a $key[x]$ valore key, e ritorna T se questo è ancora un albero binario di ricerca, altrimenti ritorna NIL. Si valuti la complessità dell'algoritmo proposto.
- Si riconsideri il precedente punto nel caso l'albero T sia un red-black tree.

Capitolo 5

Union Find

Proporremo ora un inseme di strutture dati e di operazioni sulle stesse per mantenere una famiglia di insiemi disgiunti con le operazioni di unione e ricerca.

5.1 Operazioni su insiemi disgiunti

Rappresenteremo $S_1, \ldots, S_k$ insiemi disgiunti, ovvero tali che $\forall\, i, j\ S_i \cap S_j = \emptyset$ se $i \neq j$ e proporremo delle soluzioni per le seguenti operazioni:

make(x) crea un nuovo insieme formato dal solo elemento x.

union(x, y) unisce gli insiemi dinamici i cui rappresentanti sono rispettivamente x e y, più propriamente potremmo dire che se $x \in S_1$ e $y \in S_2$ allora UNION(x, y) crea un nuovo insieme S tale che $S = S_1 \cup S_2$.

find(x) ritorna un puntatore a un insieme dinamico S_i tale che $x \in S_i$.

Ogni soluzione che proporremo per questo tipo di problema assumerà di rappresentare un insieme S mediante un suo elemento $x \in S$ che chiameremo *rappresentante*.

Una particolare soluzione proposta verrà valutata in termini di n, numero di chiamate a MAKE() ed m numero totale di chiamate a MAKE(), UNION() e FIND(). Meno formalmente n rappresenta il numero di elementi sui quali viene costruito l'insieme che intendiamo rappresentare, sicuramente $m \geq n$ ed eseguiremo al più $n - 1$ operazioni di tipo UNION().

5.1.1 Una applicazione

Potremmo utilizzare quanto detto finora per calcolare le *componenti connesse* di un grafo non orientato $G = (V, E)$.

L'algoritmo 5.2 permette di riconoscere se due elementi appartengono o meno allo stesso insieme.

Algoritmo 5.1 ConnectedComponents(G)

ConnectedComponents(G)

```
1: for each v ∈ V[G] do
2:     MAKE(v)
3: end for
4: for each e = (u, v) ∈ E[G] do
5:     if FIND(u) ≠ FIND(v) then
6:         UNION(u, v)
7:     end if
8: end for
```

Algoritmo 5.2 SameComponent(u, v)

SameComponent(u, v)

```
1: if FIND(u) ≠ FIND(v) then
2:     return FALSE
3: else
4:     return TRUE
5: end if
```

5.1.2 Rappresentazione di insiemi disgiunti

Vediamo ora come rappresentare in modo efficace insiemi dinamici che supportino operazioni di tipo MAKE(), UNION(), e FIND().

Linked list

Uno delle strutture dati più semplici che si possono utilizzare per realizzare le operazioni citate sono sicuramente le liste.

Per convenzione eleggiamo a rappresentate il primo elemento della lista e tutti gli elementi della lista ridefiniamo il puntatore al predecessore verso il rappresentante.

Grazie a queste semplici modifiche le operazioni di MAKE() e FIND()

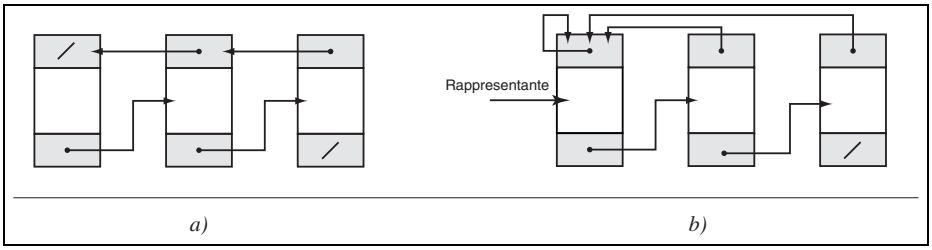

Figura 5.1: a) linked list e b)linked list modificata per soddisfare le esigenze di MAKE() e FIND().

risultano particolarmente semplici anche in termini asintotici richiedendo un costo di $O(1)$, tuttavia tale soluzione penalizza particolarmente l'operazione di UNION().

Supponiamo di eseguire un'operazione di tipo UNION(x, y), dobbiamo aggiornare tutti i puntatori all'elemento precedente di una lista al rappresentate dell'altra quindi se $x \in S_1$ e $y \in S_2$ e decidiamo di 'accodare' y a x il costo per l'operazione è $O(|S_2|)$.

Non è difficile considerare una sequenza di m operazioni che richiedano tempo $O(m^2)$. Sia $n = \lceil m/2 \rceil + 1$ e $q = m - n = \lfloor m/2 \rfloor - 1$ e supponiamo di avere come elementi $x_1, \ldots, x_n$. Eseguiamo ora la sequenza di $m = n + q$ operazioni riportata nella tabella seguente.

Operazione		Oggetti aggiornati
MAKE(x_1)	$\{x_1\}$	1
MAKE(x_2)	$\{x_2\}$	1
$\vdots$	$\vdots$	$\vdots$
MAKE(x_n)	$\{x_n\}$	1
UNION(x_1, x_2)	$\{x_1, x_2\}, \{x_3\}, \ldots, \{x_n\}$	1
UNION(x_3, x_1)	$\{x_1, x_2, x_3\}, \{x_4\}, \ldots, \{x_n\}$	2
$\vdots$	$\vdots$	$\vdots$
UNION(x_{q-1}, x_q)	$\{x_1, \ldots, x_q\}$	$q - 1$

Il costo per realizzare queste operazioni è $\Theta(n)$ per realizzare gli n MAKE(), mentre l'i-esima unione aggiorna i oggetti, il numero totale di oggetti aggiornati di tutte le chiamate a UNION() è:

$$\sum_{i=1}^{q-1} i = \Theta(q^2)$$

il tempo totale richiesto è quindi $\Theta(n + q^2)$ che è $\Theta(m^2)$ dato che $n = \Theta(m)$ e $q = \Theta(m)$.

L'euristica weighted-union Possiamo migliorare la situazione introducendo un'euristica, più precisamente un'ottimizzazione locale, che ci possa garantire che le operazioni di tipo UNION(S_i, S_j) siano eseguite in modo tale che: $|S_j| \leq |S_i|$. Tale euristica, chiamata *weighted union*, presenta due problemi riguardanti l'implementazione e il verificare di aver migliorato la situazione nel caso peggiore.

Per quanto riguarda il primo problema è sufficiente aggiungere un campo che rappresenti il numero di elementi dell'insieme e modificare UNION() e MAKE() in modo che aggiornino tale valore. Per quanto riguarda il secondo, notiamo che il costo di una singola operazione di tipo UNION() non è diminuito nel caso peggiore e quindi se l'euristica produce qualche effetto questo è visibile su una sequenza di unioni.

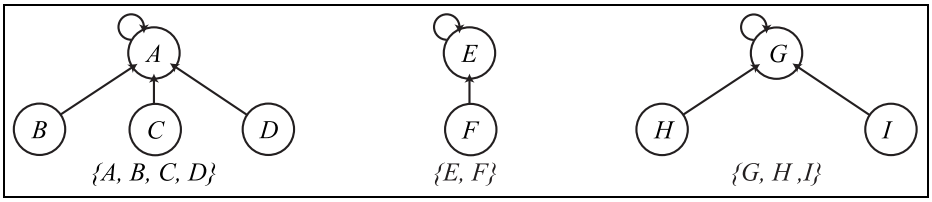

Figura 5.2: Esempio di rappresentazione di insiemi disgiunti tramite alberi di arietà variabile.

Dovremo analizzare il costo di $n-1$ operazioni di UNION() in senso ammortizzato.

Teorema 5.1.1 *Una sequenza di m operazioni di tipo* MAKE(), FIND(), WEIGHTED UNION(), *n delle quali di tipo* MAKE(), *ha costo:*

$$O(m + n \log n)$$

Dimostrazione. Prima di tutto è necessario notare che un elemento del quale ridefiniamo il puntatore all'indietro passa sempre da un insieme ad uno che ha cardinalità almeno doppia.

Il costo di una singola operazione di WEIGHTED UNION() corrisponde al numero di puntatori all'indietro che dobbiamo ridefinire. Un singolo elemento può vedere ridefinito il proprio puntatore all'indietro al più $\log n$ volte. Quindi il costo di tutte le operazioni WEIGHTED UNION() corrisponde al più a $\log n$ ridefinizioni del puntatore all'indietro per ognuno degli n elementi e quindi $O(n \log n)$.

Dobbiamo solo ricordare che il tempo richiesto per una singola operazione di tipo MAKE() e FIND() è pari a $O(1)$, ne facciamo m quindi la tesi:

$$O(m + n \log n)$$

■

Altre rappresentazioni

Alternativamente alle linked list possiamo utilizzare degli alberi di arietà variabile i cui nodi rappresentano gli elementi dell'insieme e la cui radice è il rappresentante dell'insieme. Un esempio di tale rappresentazione è visibile in figura 5.2.

Secondo questa nuova rappresentazione possiamo dire che l'operazione di MAKE() genera un albero formato dalla sola radice e ha costo $O(1)$, mentre FIND() richiede di percorre l'albero verso la radice, con costo di $O(h)$ dove h è l'altezza dell'albero. L'operazione UNION() viene eseguita semplicemente rendendo uno dei due rappresentati figlio dell'altro con costo $O(1)$.

Anche se abbiamo migliorato le prestazioni delle operazioni, dobbiamo fare attenzione all'altezza dell'albero. Se viene lasciato senza vincoli può

crescere in modo sbilanciato fino ad avere altezza $O(n)$, la ricerca conseguentemente viene a costare $O(n)$.

Introduciamo quindi una nuova euristica, *union by rank*. Questa tecnica prevede di associare ad ogni albero un valore h che rappresenta un limite superiore all'altezza. Quando dobbiamo eseguire un'operazione di tipo UNION() renderemo la radice dell'albero più basso figlia della radice dell'albero più alto.

Teorema 5.1.2 *Utilizzando l'euristica* UNION BY RANK *l'altezza h di ogni albero è tale che:*

$$h \leq \log n$$

Dimostrazione. Proviamo che ogni albero di altezza h contiene almeno 2^h nodi. Usiamo l'induzione su h:

$h = 0$ abbiamo la sola radice: $2^h = 2^0 = 1$.

$h > 0$ l'albero è stato ottenuto come unione di due alberi di altezza h_1 e h_2. Se $h_1 > h_2$ allora $h_1 = h$ e quindi anche l'albero di altezza h_1 è ottenuto come unione di due alberi. Quindi ci possiamo ricondurre al caso in cui un albero di altezza h è unione di due alberi di altezza $h-1$. Dall'ipotesi induttiva ogni albero contiene almeno 2^{h-1} nodi e quindi l'albero unione contiene almeno $2^{h-1} + 2^{h-1} = 2^h$ nodi.

■

Abbiamo quindi ottenuto che il costo di m operazioni di tipo MAKE() - UNION() - FIND(), n delle quali di tipo MAKE() è ora di $O(m \log n)$, cioè lo stesso ottenuto usando linked list e weighted union.

Il problema è che le operazioni di tipo FIND(), molto frequenti, ora costano di più. Per diminuirne il costo possiamo utilizzare l'euristica *path compression*. Tale euristica prevede di rendere tutti i nodi che troviamo in un cammino che percorriamo durante un'operazione di tipo FIND() figli della radice.

Il costo delle operazioni di tipo FIND() asintoticamente non cambia, il costo delle successive operazioni di tipo FIND() su uno dei cammini compressi è $O(1)$, se non facciamo altre operazioni di tipo UNION().

L'euristica path compression viene implementata in modo ricorsivo. L'analisi della complessità ottenuta con union by rank e path compression è difficile. Il risultato è $O(mA(m,n))$ dove $A(m,n)$ è l'inverso della funzione di Ackermann di complessità circa $O(m)$.

Algoritmo 5.3 Make(x)

Make(x)

1: $p(x) \leftarrow x$
2: $rank(x) \leftarrow 0$

Algoritmo 5.4 Find(x)

Find(x)

```
1: if x ≠ p(x) then
2:     p(x) ← FIND(p(x))
3: end if
4: return p(x)
```

Algoritmo 5.5 Union(x, y)

Union(x, y)

```
1: Link(FIND(x), FIND(y))
```

Algoritmo 5.6 Link(x, y) procedura di Union

Link(x, y)

```
1: if rank(x) > rank(y) then
2:     p(y) ← x
3: else
4:     p(x) ← y
5:     if rank(x) = rank(y) then
6:         rank(y) ← rank(y) + 1
7:     end if
8: end if
```

5.2 Esercizi

Esercizio 5.1 *(tratto dall'esame di ASD del 05-06-01)*

Si consideri la seguente estensione della struttura dati "disjoint-sets": accanto alle operazioni standard *Make-Set(x), Union(x,y)*, vengono introdotte le operazioni:

- *Find-Max(x)* Restituisce l'elemento massimo dell'insieme contente l'elemento x (supponiamo che gli elementi di un insieme possano essere ordinati),
- *List (x)* Restituisce la lista di tutti gli elementi contenuti nell'insieme contenente il valore x.

Si propongano delle strutture dati atte a supportare le operazioni indicate. Si argomenti la correttezza degli algoritmi proposti e si valuti la complessità dell'esecuzione di n operazioni di tipo *Make/Union/Find-Max/List*, m delle quali siano di tipo *Make* e p delle quali siano di tipo *List*.

Esercizio 5.2 *(tratto dall'esame di ASD del 29-11-00)*

Si consideri il problema *Union-Find* nell'ipotesi gli elementi siano numeri interi e gli insiemi siano, quindi, insiemi disgiunti di numeri interi.

Si considerino le usuali operazioni **Make**, **Union** e **Find** e si consideri inoltre l'operazione

Split che prende in input un elemento x ed un insieme A tale che $x \in A$, e torna in output:

- i due insiemi $A' = \{y \in A \mid y \leq x\}$ e $A'' = \{y \in A \mid y > x\}$, se entrambi A' e A'' sono non vuoti,
- A altrimenti.

Si propongano delle strutture dati atte a supportare le suddette operazioni. Si discuta la correttezza e si valutino le complessità degli algoritmi proposti.

Esercizio 5.3 *(tratto dall'esame di ASD del 22-7-96)*

Si supponga di dover eseguire una sequenza di operazioni di tipo MAKE-SET, UNION e FIND. Si assuma che tutte le operazioni di tipo MAKE-SET precedano quelle di tipo UNION e che queste, a loro volta, precedano quelle di tipo FIND.

- Si propongano strutture dati ed algoritmi che permettano di implementare efficientemente le operazioni considerate.
- Si analizzino le complessità degli algoritmi proposti ed il costo asintotico di una sequenza generica di operazioni che soddisfi l'assunzione relativa all'ordine delle operazioni.

Capitolo 6

Algoritmi su grafi

Ricordiamo che un grafo G è formato da due insiemi: $G = (V, E)$ dove V è l'insieme dei vertici o nodi del grafo e $E \subseteq V \times V$ è l'insieme degli archi; se $|E| \ll |V|^2$ allora diciamo che G è un *grafo sparso*, mentre se $|E| \approx |V|^2$ viene detto *grafo denso*.

Un grafo può essere rappresentato in due modi:

- mediante liste di adiacenza, *adjacency list*, dove ad ogni nodo $d \in G$ viene associata una lista $adj[d]$ nella quale sono inseriti tutti i nodi raggiungibili direttamente da d;

- mediante una matrice di adiacenza, *adjacency matrix*, dove i nodi vengono riportati come etichette di riga e colonna di una matrice e vari elementi a_{ij} della matrice possono assumere i seguenti valori, distinguendo tra grafi pesati e non:

$$a_{ij} = \begin{cases} 0 & \text{assenza di arco} \\ 1 & \text{presenza di arco} \end{cases} \qquad a_{ij} = \begin{cases} \infty & \text{assenza di arco} \\ w(i,j) & \text{peso dell'arco} \end{cases}$$

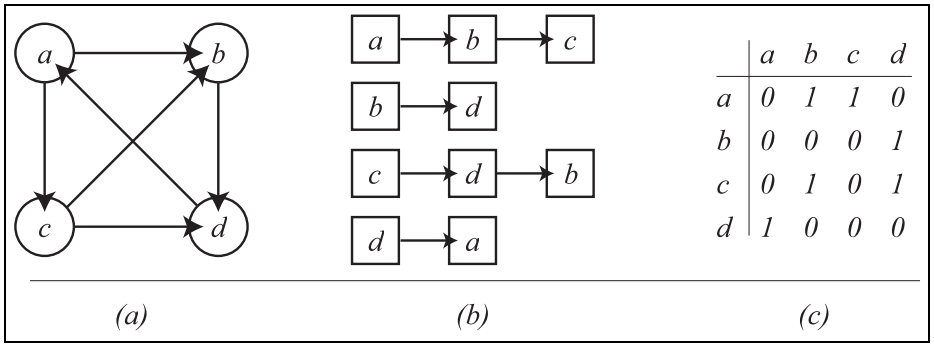

Figura 6.1: (a) grafo orientato; (b) rappresentazione mediante lista di adiacenza; (c) rappresentazione mediante matrice di adiacenza.

Definizione 6.1 (Grafo pesato) *Un grafo $G = (V, E)$ viene detto grafo pesato se esiste una funzione peso w così definita:*

$$w : E \to \mathbb{R}$$

Se A è la matrice di adiacenza di un grafo, allora indichiamo con A^T la sua *trasposta* ottenuta invertendo righe e colonne. A^T rappresenta un grafo con gli archi invertiti rispetto a quello rappresentato da A.

Nel caso di grafi indiretti abbiamo che $A^T = A$ in quanto la matrice è simmetrica.

6.1 Breadth-first search

Breadth-first search è un metodo di visita in ampiezza di grafi ottimo per calcolare la distanza tra nodi, mentre risulta carente nell'evidenziare la presenza di cicli in presenza dei quali conviene usare una visita in profondità.

Dato un grafo $G = (V, E)$ e un nodo sorgente $s \in V$ vorremmo visitare tutti i nodi raggiungibili da s nel minor tempo possibile.

Adottiamo il seguente schema di colorazione dei nodi:

- *bianco* indica che il nodo non è stato ancora scoperto;
- *grigio* indica che il nodo è stato individuato, ma l'algoritmo non ha ancora finito di trattarlo;
- *nero* indica che il nodo è stato scoperto e che l'algoritmo ha finito di analizzarlo.

Durante la visita del grafo determiniamo per ogni nodo $u \in V$ due valori distinti:

- $d(u)$ distanza dalla sorgente nel cammino minimo;
- $\pi(u)$ genitore di u in un albero dei cammini minimi o BF-albero di radice s.

BFS visita tutti i nodi raggiungibili da s determinandone la distanza da s stesso sia G orientato o meno.

Per quanto riguarda la complessità di BFS, algoritmo 6.1, la parte di inizializzazione (linee da 1 a 9) prende tempo $O(|V|)$, analogamente il ciclo while (linee da 10 a 22) prende tempo $O(|V|)$ poiché un nodo può essere accodato solo quando è bianco ed esce dalla coda quando viene colorato di nero.

L'unico limite superiore imponibile a $|adj[u]|$ è $O(|V|)$ il costo del ciclo for è $O(|V|)$, quindi da un'analisi grossolana potremmo concludere che BFS ha complessità $O\big(|V|^2\big)$. Cerchiamo ora di raffinare l'analisi, dimostreremo

Algoritmo 6.1 BFS(G, s)

BFS(G, s)

```
1: for each v ∈ V \ {s} do
2:    color[v] ← WHITE
3:    d(v) ← ∞
4:    π(v) ← NIL
5: end for
6: color[s] ← GRAY
7: d(s) ← 0
8: π(s) ← NIL
9: Q ← {s}
10: while Q ≠ ∅ do
11:    u ← head(q)
12:    for each v ∈ adj[u] do
13:       if color[v] = WHITE then
14:          color[v] ← GRAY
15:          d(v) ← d(u) + 1
16:          π(v) ← u
17:          enqueue(Q, v)
18:       end if
19:    end for
20:    dequeue(Q)
21:    color[u] ← BLACK
22: end while
```

infatti che la complessità è pari a $O(|E| + |V|)$, come dire il massimo tra $|V|$ e $|E|$.

Osserviamo che all'interno del ciclo for le operazioni hanno costo $O(1)$ e sono eseguite $|adj[u]|$ volte. Il ciclo while esegue una quantità di istruzioni pari a $\sum_{u \in V}(|adj[u]|)O(1) = O(|E|)$. Se consideriamo quindi l'inizializzazione otteniamo una complessità pari a:

$$O(|E| + |V|)$$

Dimostriamo ora che BFS è corretto rispetto al calcolo della distanza d dalla sorgente s, ovvero che dopo l'esecuzione di BFS(G, s) abbiamo che $\forall\, v \in V,\ d(v)$ contiene la distanza di v da s e $\pi(v)$ è il genitore di v in un albero dei cammini minimi di radice s.

Siano $s, v \in V$, definiamo la distanza di v da s nel modo seguente:

$$\delta(u, v) = \begin{cases} \infty & \text{se non esiste un cammino } p : s \overset{p}{\rightsquigarrow} v \\ n & \text{peso degli archi sul cammino minimo da } s \text{ a } v \end{cases}$$

Chiaramente deve valere la disuguaglianza $0 \leq \delta(s, v) \leq |V| - 1$.

Lemma 6.1.1 *Sia $G = (V, E)$ grafo orientato o non orientato e sia $s \in V$ un nodo arbitrario, allora per ogni arco $(u, v) \in E$ si ha che:*

$$\delta(s, v) \leq \delta(s, u) + 1$$

Lemma 6.1.2 *Sia $G = (V, E)$ un grafo orientato o non orientato, sia $s \in V$ nodo arbitrario allora dopo BFS(G, s) abbiamo che $\forall\, u \in V$:*

$$d(u) \geq \delta(s, u)$$

Dimostrazione. La dimostrazione si fa per induzione sul numero di nodi inseriti nella coda Q. In altri termini stiamo provando la validità di un invariante del ciclo while.

$e = 1$ l'unico nodo inserito è s, allora $d(s) = 0$ e $\forall\, v \in V\ d(v) = \infty \geq \delta(s, v)$;

$e > 1$ dopo $e - 1$ inserimenti dall'ipotesi induttiva $\forall\, u \in V,\ d(u) \geq \delta(s, u)$. Eseguiamo ora l'inserimento del nodo v e supponiamo di eseguire l'assegnazione $d(v) \leftarrow d(u) + 1$. Dall'ipotesi induttiva segue $d(v) = d(u) + 1 \geq \delta(s, u) + 1$ Poiché $(u, v) \in E$ applicando il lemma 6.1.1 abbiamo che:

$$\begin{aligned} d(v) &= d(u) + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \end{aligned}$$

e quindi la tesi.

■

Lemma 6.1.3 *Supponiamo che durante BFS(G, s) sia $Q = \langle v_1, \ldots, v_r \rangle$ dove head$(Q) = v_1$ allora:*

1. $d(v_r) \leq d(v_1) + 1$
2. $\forall\, i \in \{1, \ldots, r-1\},\ d(v_i) \leq d(v_{i+1})$

Dimostrazione. Per induzione sul numero di operazioni sulla coda Q.

$e = 1$ La sola operazione eseguita è l'inserimento di s, quindi le altre distanze sono tutte pari a ∞:

1. $r = 1 \Rightarrow v_r = v_1 \Rightarrow d(v_r) = d(v_1) < d(v_1) + 1$
2. l'insieme è vuoto, l'affermazione è banalmente verificata.

$e > 1$ dopo $e - 1$ inserimenti/cancellazioni 1) e 2) valgono.

Se la e-esima operazione è un cancellamento la tesi è ovvia, se erano ordinati i primi r allora anche gli ultimi $r - 1$ erano ordinati.

Se la e-esima operazione è un inserimento ricordiamo che il nodo v inserito è tale che:

$$d(v) = d(v_1) + 1 \quad (*)$$

dopo l'inserimento di $v = v_{r+1}$ la coda sarà $\langle v_1, \ldots, v_r, v_{r+1} \rangle$ quindi 1) segue dall'osservazione $(*)$. Dall'ipotesi induttiva abbiamo che $d(v_r) \leq d(v_1) + 1 = d(v_{r+1})$ e quindi 2).

■

Teorema 6.1.1 (Correttezza di BFS) *Sia $G = (V, E)$ un grafo orientato o non orientato, siano $s, v \in V$ nodi arbitrari. Dopo l'esecuzione di BFS(G, s) abbiamo che:*

1. *$d(v) = \delta(s, v)$ anche se v non è raggiungibile da s;*
2. *Se v è raggiungibile da s allora uno dei cammini per raggiungere v passa per $\pi(v)$.*

Dimostrazione. Se v non è raggiungibile da s, allora $\delta(s, v) = \infty$ e per il lemma 6.1.2: $\infty = \delta(s, v) \leq d(v) = \infty$.

Sia v è raggiungibile da s, definiamo:

$$V_k \equiv \{v \in V :\ \delta(s, v) = k\}$$

dimostriamo 1) e 2) per ogni nodo in V_k al variare di k, quindi per induzione su k. In particolare dimostriamo che dato $v \in V_k$, esiste una singola iterazione del ciclo for in cui vengono eseguite le seguenti operazioni:

1. v viene colorato di grigio;
2. $d(v)$ passa da ∞ a k;
3. $s \neq v$ allora $\pi(v)$ passa da NIL a un nodo appartenente a V_{k-1};

4. v viene inserito nella coda;

Vediamo l'induzione su k:

$k = 0$ abbiamo che $V_k = V_0 = \{s\}$:

1. s viene colorato di grigio nella fase di inizializzazione;
2. $d(s)$ passa da ∞ a 0 nella fasi di inizializzazione;
3. dato che $s = v$ l'affermazione è vera;
4. s viene inserito nella coda nella fase di inizializzazione;

$k \geq 1$ per ipotesi induttiva $\forall\, u \in V_{k-1}$ esiste un unica iterazione in cui vengono eseguite 1)-4) su u. Quindi ogni $u \in V_{k-1}$ è tale che $(u, v) \in E$ viene inserito in Q. Inoltre per la parte 2) del lemma 6.1.3, ogni $u \in V_{k-1}$ viene inserito in Q prima di v. Durante un iterazione in cui $head(Q) \in V_{k-1}$ e $(head(Q), v) \in E$, v viene inserito in Q. In quell'iterazione:

1. v viene colorato di grigio;
2. $d(v)$ diviene $d(head(Q)) = k - 1 + 1 = k$
3. $\pi(v) = head(Q) \in V_{k-1}$
4. v viene inserito nella coda.

Per concludere basta osservare che 2) implica la prima affermazione del teorema, mentre 3) la seconda.

■

6.1.1 Breadth-first tree

La procedura BFS genera un'albero, chiamato *bread-first tree*, ottenuto dai valori di $\pi(v)$ di ogni nodo v. Formalmente, vale la seguente definizione:

Definizione 6.2 (Predecessor subgraph) *Dato un grafo $G = (V, E)$, un nodo sorgente $s \in G$ ed eseguito BFS(G, s) diciamo predecessor subgraph di G il seguente grafo:*

$$G_\pi = (V_\pi, E_\pi) \tag{6.1}$$

dove:

$$\begin{aligned} V_\pi &= \{v \in V : \pi(v) \neq \text{NIL}\ \} \cup \{s\} \\ E_\pi &= \{(\pi(v), v) \in E :\ v \in V_\pi \setminus \{s\}\} \end{aligned}$$

Notiamo che G_π è sicuramente un albero, definito *bread-first tree*, in quanto connesso poiché ogni elemento di V_π è raggiungibile da s, ed inoltre $|E_\pi| = |V_\pi| - 1$ e quindi per il lemma sui free-tree G_π è un albero.

Dato $v \in V_\pi$, il cammino da s a v in G_π è l'unico cammino minimo in G_π per raggiungere v da s ed ha lunghezza pari alla lunghezza del cammino minimo in G.

6.2 Depth-first search

Depth-first search è un metodo di visita in profondità di grafi, esso utilizza lo stesso schema di colorazione definito per breadth-first search, ma definisce per ogni nodo $v \in V$ due nuovi valori:

- $d(v)$ registra l'istante in cui il nodo v viene scoperto;
- $f(v)$ registra l'istante in cui l'algoritmo finisce di trattare il nodo v.

Entrambi i contatori sono mantenuti tramite un clock globale. Analogamente a quanto detto per la visita in ampiezza i valori $\pi()$ generano un *predecessor subgraph* G_π definito però in modo leggermente differente:

$$G_\pi = (V, E_\pi) \tag{6.2}$$

dove:

$$E_\pi = \{(\pi(v), v) : v \in V \text{ e } \pi(v) \neq \text{NIL } \}$$

Algoritmo 6.2 DFS(G)

DFS(G)

```
1: for each v ∈ V do
2:    color[v] ← WHITE
3:    π(v) ← NIL
4: end for
5: time ← 0
6: for each u ∈ V do
7:    if color[u] = WHITE then
8:       DFSVisit(u)
9:    end if
10: end for
```

6.2.1 Complessità

Prima di tutto cerchiamo di stimare il tempo totale delle chiamate a DFSVisit. Esso è proporzionale al tempo necessario ad eseguire tutte le istruzioni diverse dalle chiamate ricorsive più il tempo necessario ad eseguire DFSVisit(v) per ogni $v \in adj[u]$.

Il tempo necessario ad eseguire tutte le istruzioni diverse da chiamate ricorsive è $O(|adj[u]| + 1)$, quindi il tempo necessario ad eseguire tutte le chiamate a DFSVisit è:

$$\begin{aligned} \sum_{u \in V} O(|adj[u]| + 1) &= \sum_{u \in V} O(|adj[u]|) + \sum_{u \in V} O(1) \\ &= O(|E|) + O(|V|) \\ &= O(|E| + |V|) \end{aligned}$$

Algoritmo 6.3 DFSVisit(u)

DFSVisit(u)

```
1: color[u] ← GRAY
2: d(u) ← time ← time + 1
3: for each v ∈ adj[u] do
4:    if color[v] = WHITE then
5:       π[v] ← u
6:       DFSVisit(v)
7:    end if
8: end for
9: color[u] ← BLACK
10: f(u) ← time ← time + 1
```

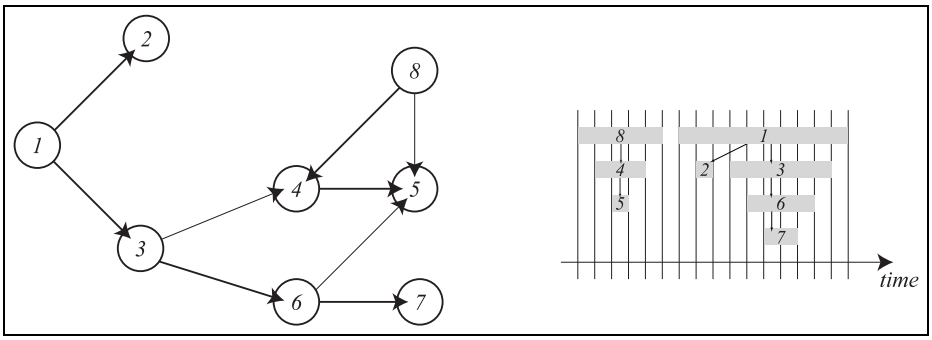

Figura 6.2: Esempio di grafo e di come agisce DFS.

Analizziamo il sottografo G_π ottenuto da G considerando solo gli archi $(\pi(u), u)$, dimostriamo che G_π è una foresta, *depth-first forest*, formata da df-alberi, *depth-first tree*.

L'esecuzione di DFS ci permette di associare ad ogni nodo un intervallo temporale, rappresentato da una coppia di parentesi in modo tale che due intervalli risultino disgiunti oppure incapsulati, seguendo l'esempio in figura 6.2 avermo:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
(8	(4	(5	5)	4)	8)	(1	(2	2)	(3	(6	(7	7)	6)	3)	1)

Teorema 6.2.1 (Delle parentsi) *Sia $G = (V, E)$ un grafo orientato/non orientato. Dopo l'esecuzione di DFS(G) abbiamo che $\forall\, u, v \in V$ vale esattamente una delle seguenti tre condizioni:*

1. $[\, d(u), f(u)\,] \cap [\, d(v), f(v)\,] = \emptyset$
2. $[\, d(u), f(u)\,] \subseteq [\, d(v), f(v)\,]$
3. $[\, d(u), f(u)\,] \supseteq [\, d(v), f(v)\,]$

Dimostrazione. Sia $d(u) < d(v)$ allora abbiamo due casi possibili:

1. $d(u) < f(v)$. La procedura DFSVisit(v) viene chiamata prima che termini DFSVisit(u), quindi è chimata da DFSVisit(u) o da una delle procedure ricorsive chiamate al suo interno, quindi terminerà prima di DFSVisit(u), questo implica $f(v) < f(u)$ quindi il caso 3) del teorema.

2. $d(u) > f(v)$. Questo implica $[\,d(u), f(u)\,] \cap [\,d(v), f(v)\,] = \emptyset$ ovvero il caso 1) del teorema.

Se invece assumiamo che $d(u) > d(v)$ otteniamo due casi analoghi ai precedenti, che implicano rispettivamete il caso 2) e il caso 1) del teorema. ■

Dal teorema è possibile ricavare il seguente:

Corollario 6.2.1 v è un discendente di u nella depth-first forest (df-forest) se e solo se:

$$d(u) < d(v) < f(v) < f(u)$$

Teorema 6.2.2 (del cammino bianco) *Sia $G = (V, E)$ un grafo orientato o non orientato e DFS(G) eseguito. Siano $u, v \in V$ allora v è discendente di u in G_π se e solo se all'istante $d(u)$ esiste un cammino da u a v costituito da soli nodi bianchi.*

Dimostrazione. Analizziamo i due versi del se e solo se:

$\Rightarrow$ Assumiamo v discendente di u. Sia w un qualunque nodo lungo il cammino da u a v in un df-tree, allora w è discendente di u. Utilizzando il corollario 6.2.1 abbiamo che $d(w) > d(u)$ e quindi color[w] è WHITE all'istante $d(u)$;

$\Leftarrow$ Per assurdo, sia v il primo nodo per cui esiste un cammino bianco da u a v e v non risulti discendente di u in G_π. Sia w il predecessore[1] di v nel cammino bianco da u a v. Per il corollario $f(w) \leq f(u)$. Poiché v è bianco ed è raggiungibile da w allora $d(v) < f(w)$ e quindi:

$$d(u) < d(v) < f(w) \leq f(u)$$

quindi per il teorema 6.2.1, delle parentesi:

$$[\,d(u), f(u)\,] \supseteq [\,d(v), f(v)\,]$$

quindi per il corollario 6.2.1 v è discendente di u e quindi abbiamo un assurdo.

■

[1] inteso come nodo che viene scoperto appena prima

6.2.2 Classificazione degli archi

Cerchiamo ora di classificare gli archi presenti in una depth first forest G_π, prodotta da un depth first search su G:

- *tree-edges*: archi della df-forest G_π. L'arco (u, v) è un tree-edge se v è stato scoperto esplorando l'arco (u, v).
- *back-edges*: archi da un discendente ad un avo in un df-tree in G_π. Self-loops sono considerati back-edges.
- *forward-edges*: archi da un avo ad un discendente in G_π che non sono interni ad un df-albero.
- *cross-edges*: tutti gli altri. Possono collegare nodi nello stesso df-albero oppure appartenenti a df-alberi diversi.

Proposizione 6.2.1 Se $T_1, \ldots, T_k$ sono i df-alberi generati da DFS(G) ogni cross-edge:

1. va da un nodo in T_i ad un nodo in T_i oppure
2. va da un nodo in T_i ad un nodo in T_j se $i > j$

Teorema 6.2.3 *Sia $G = (V, E)$ un grafo non orientato, allora $\forall\, (u, v) \in E$:*

1. *(u, v) è un tree-edge oppure*
2. *(u, v) viene esplorato da v ad u con v discendente di u.*

Seguono alcune importanti applicazioni di DFS.

6.2.3 Topological sort

Una delle applicazioni di DFS è la realizzazione dell'ordinamento topologico di grafi orientati e aciclici, più brevemente DAG (*directed acyclic graph*).

Definizione 6.3 (Topological sort) *Sia $G = (V, E)$ un grafo diretto aciclico, topological sort di G è un ordinamento dei nodi di G tale che:*

$$(u, v) \in E \Rightarrow u < v$$

Una delle principali applicazioni di topological sort è di determinare la precedenza tra eventi.

Lemma 6.2.1 *Un grafo G è un DAG se e solo se DFS(G) non produce back-edges.*

Dimostrazione. Analizziamo i due versi dell'implicazione:

$\Rightarrow$ Per assurdo. Se G è un DAG e possiede un back-edge allora G ha un ciclo e quindi si contraddice la definizione di DAG.

$\Leftarrow$ Per contrapposizione. Dato un ciclo in G sia v il primo nodo in tale ciclo ad essere scoperto. Sia (u, v) l'arco che precede v nel ciclo.

Poiché v è il primo nodo del ciclo ad essere scoperto allora tutti i nodi del ciclo sono bianchi nell'ipotesi in cui si scopra v.

Per il teorema 6.2.2, del cammino bianco, e l'osservazione precedente segue che tutti i nodi del ciclo sono discendenti di v nella df-forest, quindi u sarà un discendente di v e quindi (u, v) un back-edge.

■

Utilizzando il precedente lemma possiamo produrre un ordine topologico di un qualunque grafo aciclico utilizzando un algoritmo ricorsivo basato su DFS(), la cui complessità sarà $O(V + E)$ determinata appunto dalla chiamata a DFS.

- chiamiamo DFS(G) per calcolare $f(v)$ $\forall v \in V$;
- ogni volta che terminiamo con un vertice lo inseriamo in testa ad una linked list;
- ritorniamo la linked list dei nodi ottenuta

Teorema 6.2.4 *Un algoritmo che rispetta lo schema precedente produce un ordinamento topologico di un DAG G.*

Dimostrazione. Supponiamo di eseguire DFS sul grafo di input per determinare il *finishing time* per i suoi vertici; dimostriamo ora che per ogni coppia di vertici distinti $u, v \in V$ se esiste un arco da u a v allora $f(v) \leq f(u)$.

Consideriamo un arco (u, v) esplorato da DFS(G), v non può essere grigio infatti v sarebbe avo di u e (u, v) sarebbe un back-edge contraddicendo il lemma 6.2.1. Dobbiamo verificare cosa succede se v è bianco oppure nero; semplicemente:

- se v è bianco allora è discendente di u quindi $f(v) < f(u)$;
- se v è nero allora $f(v) < f(u)$

quindi per ogni arco (u, v) del DAG di input abbiamo che $f(v) < f(u)$. ■

6.2.4 Componenti fortemente connesse

Vediamo ora un'altra applicazione tipica di DFS, ovvero la scomposizione di un grafo orientato nelle sue componenti fortemente connesse o SCC, *strongly connected components*.

Definizione 6.4 (Mutua raggiungibilità) *u e v sono in relazione di mutua raggiungibilità in G se e solo se esiste un cammino da u a v ed un cammino da v a u.*

Definizione 6.5 (Strongly Connected Components) *Le componenti fortemente connesse di G sono le classi di equivalenza rispetto alla relazione di mutua raggiungibilità.*

Notiamo quindi che la relazione di mutua raggiungibilità è una relazione di equivalenza.

Definizione 6.6 (Grafo trasposto) *Dato un grafo G diciamo grafo trasposto G^T il grafo ottenuto invertendo il verso di ognuno degli archi di G.*

Equivalentemente il grafo trasposto di G è il grafo la cui matrice di adiacenza è la trasposta delle matrice di adiacenza di G.

Algoritmo 6.4 ComponenteTrasposta(G) procedura di SCC

ComponenteTrasposta(G)

```
1: for each v ∈ V do
2:    for each u ∈ adj[v] do
3:       adj_1[u] ← v
4:    end for
5: end for
6: for each v ∈ V do
7:    adj[v] ← adj_1[v]
8: end for
```

Vediamo quale è la complessita della procedura componente trasposta riportata nell'algoritmo 6.4. Il primo ciclo for ha la seguente complessità:

$$O\left(\sum_{v \in V}(adj[v]+1)\right) = O(|E|+|V|)$$

mentre il secondo ciclo for costa:

$$O(|V|)$$

per un totale di:

$$O(|V|+|E|)$$

Analogamente SCC(G) ha complessità: $O(|E|+|V|)$, ottenuta con la seguente analisi:

Algoritmo 6.5 SCC(G)

SCC(G)

```
1: DFS(G)
2: G ← ComponenteTrasposta(G)
3: for each u ∈ V do
4:    color[u] ← WHITE
5:    π(u) ← NIL
6:    F[f(u)] ← u {vettore che accoglie i vertici nell'ordine in cui si è finito
      di analizzarli}
7: end for
8: time ← 0
9: for i = size(F) downto 1 do
10:   if (F[i] ≠ NIL ) ∧ (color[F[i]] = WHITE) then
11:      DFSVisit(F[i])
12:   end if
13: end for
```

Id linea	Costo
1	$O(\|E\| + \|V\|)$
2	$O(\|E\| + \|V\|)$
3-7	$O(\|V\|)$
8	$O(1)$
9-13	$O(\|V\| + \|E\|)$

L'ultima chiamata a DFSVisit produce df-alberi che sono le componenti fortemente connesse cercate. Valgono inoltre le seguenti considerazioni.

- Poiché i cross-edge vanno solo da alberi con indice maggiore ad alberi con indice minore segue che una SCC è sempre contenuta in T_i poiché esiste un cammino solo in una direzione e non in quella opposta.

- Le chiamate a DFSVisit sul grafo trasposto avvengono nell'ordine inverso dei valori di $f(v)$ assegnati dalla prima chiamata a DFS.

- Il primo nodo x sul quale viene chiamata DFSVisit sul grafo trasposto è la radice dell'ultimo albero prodotto dalla prima chiamata a DFS.

- I nodi nella SCC contenente x sono i nodi di T_k raggiungibili da x in G^T, poiché erano raggiungibili da x anche in G

- Dopo aver prodotto la SCC contenente x (come df-albero) ci ritroviamo in una situazione del tutto analoga a quella precedente la chiamata a DFS

6.3 Minimum spanning tree

Il problema richiede in input un grafo $G = (V, E)$ indiretto connesso e pesato. Sia $w(u, v) \in \mathbb{R}^+$ la funzione peso per G. L'output che vorremmo è un insieme di archi T tale che $T \subseteq E$ e (V, T) è connesso ed inoltre $w(T)$ è *minimo*, dove:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

Dato che T è aciclico e connette tutti i suoi vertici deve formare un albero, che chiameremo *spanning tree*; in conseguenza al fatto che $w(t)$ deve essere minimo lo chiameremo *minimum spanning tree*.

Analizzeremo gli algoritmi di Kruskal e Prim per la determinazione dei minimum spanning tree. Entrambi introducono un'euristica chiamata *greedy strategy* che in caso di scelte multiple intraprende la strada che sembra migliore al momento.

6.3.1 Algoritmo generico per MST

Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso $w : E \to \mathbb{R}$, vorremmo trovare un minimum spanning tree per G.

L'algoritmo che proponiamo considera un insieme A il quale sarà sempre sottoinsieme di qualche MST, ad ogni passo determineremo un arco (u, v) che possa essere aggiunto ad A senza violare l'invariante, ovvero anche $A \cup \{(u, v)\}$ risulterà sottoinsieme di qualche MST.

Prima di analizzare l'algoritmo generico per la determinazione di un MST diamo la seguente definizione:

Definizione 6.7 (Safe-edge) *Sia $A \subseteq T$ con T qualche MST, un arco (u, v) è un safe-edge per A se e solo se $A \cup \{(u, v)\}$ risulta ancora sottoinsieme di qualche MST.*

Algoritmo 6.6 MST(G)

MST(G)

```
1: A ← ∅
2: while A non forma un MST do
3:    find(u, v) safe-edge per A
4:    A ← A ∪ {(u, v)}
5: end while
6: return A
```

Vediamo ora di stabilire come identificare i safe-edge.

Definizione 6.8 *Seguono alcune definizioni inerenti gli MST:*

- *indichiamo con* taglio *la coppia* $(S, V - S)$ *dove* $S \subseteq V$*;*
- (u, v) attraversa il taglio $(S, V - S)$ *se e solo se* $u \in S$ *e* $v \in V - S$
- $A \subseteq E$ rispetta un taglio $(S, V - S)$ *se e solo se nessuno dei suoi elementi attraversa il taglio*
- *Dato una qualunque proprietà* P *di archi, diciamo che* (u, v) *è* leggero per P *se e solo se* (u, v) *soddisfa* P *ed è di peso minimo tra gli archi che soddisfano* P*. In particolare un arco è un arco leggero rispetto a un taglio se è di peso minimo tra tutti gli archi che attraversano il taglio stesso.*

Teorema 6.3.1 *Sia* $G = (V, E)$ *grafo non orientato e connesso,* $w : E \to \mathbb{R}^+$ *funzione di peso,* $A \subseteq T$ *con* T *MST per* G*,* $(S, V - S)$ *taglio di* G *e* A *rispetta* $(S, V - S)$*.*

Se (u, v) *è l'arco leggero che attraversa* $(S, V - S)$ *allora* (u, v) *è un safe-edge per* A*.*

Dimostrazione. Sia T un minimum spanning tree che include A, dobbiamo dimostrare che $A \cup \{(u, v)\}$ è sottoinsieme di un qualche MST T' che andremo a costruire.

Se $(u, v) \in T$ abbiamo finito, altrimenti dobbiamo determinare T' MST tale che $A \cup \{(u, v)\} \subseteq T'$. Se $(u, v) \notin T$ allora supponiamo esistano $x, y \in V$ tali che (x, y) attraversa $(S, V - S)$ e $(x, y) \in T$; supponiamo anche che $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$ è un MST per G.

L'arco (u, v) forma un ciclo con l'unico cammino in T che congiunge (u, v), poiché u e v sono dalle parti opposte del talgio ci deve essere almeno un arco del MST T che attraverssa il taglio.

Sia (x, y) tale arco, esso non è in A perché A rispetta il taglio, poiché (x, y) è nell'unico cammino da u a v in T rimuovendolo rompiamo T in due componenti, aggiungendo (u, v) le riconnettiamo in un nuovo albero:

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$

Poiché si ha che (u, v) è di peso minimo fra gli archi che attraversano $(S, V - S)$ si ha $w(u, v) \leq w(x, y) \Rightarrow w(T') \leq w(T) \Rightarrow w(T') = w(T)$. ■

Utilizzando il precedente teorema ed il fatto che l'algoritmo generico è corretto possiamo ottenerne uno per la determinazione di un MST se sappiamo determinare un taglio ed un arco leggero che lo attraversa ad ogni iterazione. Per ottenere il taglio conviene pensare ad A come definente una relazione di equivalenza fra nodi le cui classi di equivalenza sono alberi.

Corollario 6.3.1 Sia $G = (V, E)$ grafo connesso non orientato, $w : E \to \mathbb{R}^+$, $A \subseteq T$ MST e C componente connessa (Albero) della relazione di equivalenza definita da A. Sia (u, v) arco leggero che connette C ad un'altra componente connessa allora (u, v) è un safe-edge per A.

Dimostrazione. $(C, V \setminus C)$ è un taglio rispettato da A e quindi possiamo concludere per il teorema. ■

Dobbiamo usare delle strutture dati che:

- ci permettano di mantenere le componenti connesse C
- ci permettano di individuare velocemente un arco leggero che connette una C al resto del grafo

6.3.2 Algoritmo di Kruskal

Cercheremo di ricavare un minimum spanning tree per un grafo G utilizzando un algoritmo che procede in modo *greedy* sfruttando le strutture dati introdotte per union-find.

Algoritmo 6.7 MST-Kruskal(G, w)

MST-Kruskal(G, w**)**

```
1: A ← ∅
2: for each v ∈ V do
3:    MAKE(v)
4: end for
5: sort(E, w) {ordiniamo in maniera crescente rispetto al peso w degli
   archi}
6: for each (u, v) ∈ E do
7:    if FIND(u) ≠ FIND(v) then
8:       A ← A ∪ {(u, v)}
9:       UNION(u, v)
10:   end if
11: end for
12: return A
```

Inizialmente ogni nodo appartenente al grafo forma un albero, ovvero componenti connesse diverse, A quindi è una foresta. I safe-edge da aggiungere ad A vengono individuati trovando tutti gli archi di peso minimo che connettono due componenti connesse diverse (alberi) C_1 e C_2.

La correttezza si prova per induzione sul numero di iterazioni eseguite dal secondo ciclo for: ad ogni iterazione A è sottoinsieme di un MST (per dimostrarlo usiamo il corollario con C classe di equivalenza di u).

Sfruttando le euristiche come union-by-rank e path-compression, si ottiene una complessità per il secondo ciclo for pari a:

$$O(|E|\alpha(|E|, |V|)) \simeq O(|E| \log |E|)$$

dove α è l'inversa della funzione di Ackermann.

6.3.3 Algoritmo di Prim

Ecco riportato lo schema:

- A sarà un albero dall'inizio alla fine, avremo sempre una sola classe di equivalenza non unitaria;
- Ad ogni passo determiniamo in modo greedy un nuovo arco da aggiungere ad A;
- Dimostreremo la correttezza usando il corollario (il taglio che usiamo è $(dom(A), V \setminus dom(A))$)
- Useremo una coda di priorità per eseguire ottimamente la scelta dell'arco da aggiungere ad A. La chiave per gli elementi in Q sarà il minimo fra i pesi di uno degli archi che connette un nodo in A ad uno dei nodi non in A.

Algoritmo 6.8 MST-Prim(G, w, r)

MST-Prim(G, w, r)

```
1: Q ← V
2: for each u ∈ V do
3:    KEY(u) ← ∞
4: end for
5: π(r) ← NIL
6: KEY(r) ← 0
7: BuildHeap(Q)
8: while Q ≠ ∅ do
9:    u ← ExtractMin(Q)
10:   for each v ∈ adj[u] do
11:      if (v ∈ Q) ∧ (w(u, v) < KEY(v)) then
12:         π(v) ← u
13:         KEY(v) ← w(u, v)
14:         DecreaseKey(v, Q)
15:      end if
16:   end for
17: end while
```

La complessità di tutte le iterazioni del ciclo while è pari a:

$$O\left(\sum_{v \in V} (|adj[v]| + 1) \log |V|\right)$$

il costo di DecreaseKey(v, Q) è quindi $O(|E| + |V| \log(|V|))$.

La correttezza si prova per induzione sul numero di operazioni eseguite dal ciclo while, cioè ad ogni iterazione l'albero A è un sottoinsieme di T MST, infatti l'arco (u, v) che inseriamo è l'arco leggero che attraversa il taglio $(dom(A), V \setminus dom(A))$ e quindi concludiamo per il corollario 6.7.

6.4 Single source shortest path problem

Come si evince dal titolo il problema che intendiamo risolvere è la determinazione, dato un grafo pesato G ed un vertice *sorgente* $s \in G$, i cammini minimi da s ad un qualunque altro vertice in G.

Utilizzeremo i seguenti elementi: un grafo diretto e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbb{R}$; indicheremo con $p = \langle v_0, \ldots, v_k \rangle$ un *cammino* in G e denoteremo con $w(p)$ il peso del cammino nel seguente modo:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Diciamo poi che un *cammino minimo* p da u a v in G è tale che:

$$w(p) = \delta(u, v) = \begin{cases} \min\{w(p') : u \overset{p'}{\rightsquigarrow}_G v\} \\ \infty \text{ se tale cammino non esiste} \end{cases}$$

Oltre al problema che analizziamo vale la pena citare alcune varianti:

- *single destination shortest path* problem;
- *single pair shortest path* problem: dati $u, v \in V$ determinare il cammino minimo da u a v. Non ci solo soluzioni migliori note che risolvano *sssp* a partire da u
- *all pairs shortest path* problem: si può risolvere applicando n volte un algoritmo per il *sssp* ma si può fare meglio

Vedremo un algoritmo che lavora nell'ipotesi di assenza di archi di peso negativo. Algoritmi che trattano anche il caso in cui gli archi possano avere peso negativo devono comunque assumere l'assenza di cicli di peso negativo raggiungibili dalla sorgente s, infatti in presenza di tali cicli il cammino minimo risulta non ben definito.

Rappresenteremo la soluzione mediante il *predecessor sub-graph* $G_\pi = (V_\pi, E_\pi)$ che dimostreremo essere un albero di radice s che chiameremo *shortest path tree* tale che:

- $V_\pi = \{v \in V : \pi(v) \neq \text{NIL } \} \cup \{s\}$, sarà l'insieme dei nodi raggiungibili da s;
- $E_\pi = \{(\pi(v), v) \in E : v \in V_\pi - \{s\}\}$;

- G_π avrà radice s;
- $\forall\, v \in V$ il cammino da s a v in G_π sarà il cammino minimo da s a v in G.

Le soluzioni che analizzeremo sono due: *Dijkstra*, che funziona solo se non ci sono archi negativi e *Bellman-Ford* che funziona sempre.

6.4.1 Dijkstra

L'algoritmo risolve *sssp* su un grafo pesato e orientato $G = (V, E)$, assumendo che $w(u, v) \geq 0$ per ogni arco in G.

L'algoritmo mantiene un insieme di vertici S il cui cammino minimo dalla sorgente s è già stato determinato, ovvero per tutti i vertici $v \in S$, abbiamo che $d(v) = \delta(s, v)$.

Lemma 6.4.1 *Sia $p = \langle v_1, \ldots, v_k \rangle$ lo shortest path da v_1 a v_k, allora $\forall\, i, j$: $1 \leq i \leq j \leq k$ abbiamo che $\langle v_i, \ldots, v_j \rangle$ è uno shortest path da v_i a v_j.*

Corollario 6.4.1 Sia $s \overset{p}{\rightsquigarrow} v$ shortest path da s a v e sia decomponibile in $s \overset{p'}{\rightsquigarrow} u \to v$ allora:

$$\delta(s, v) = \delta(s, u) + w(u, v)$$

Lemma 6.4.2 $\forall\, (u, v) \in E$ *si ha* $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Dimostrazione. Lo shortest path ha per definizione il peso minimo fra tutti i cammini da s a v e quindi ogni altro cammino ha peso maggiore o uguale. ■

Analizziamo ora una tecnica di rilassamento degli archi (algoritmo 6.9). $\forall\, v \in V$ manteniamo $d(v)$ che rappresenta un'approssimazione a $\delta(s, v)$ tale che $d(v) \geq \delta(s, v)$. Per ogni arco $(u, v) \in E$ calcoliamo $d(u) + w(u, v)$ ed eventualmente aggiorniamo $d(v)$.

Algoritmo 6.9 Relax(u, v, w)

Relax(u, v, w)

```
1: if d(v) > d(u) + w(u, v) then
2:     d(v) ← d(u) + w(u, v)
3:     π(v) ← u
4: end if
```

Lemma 6.4.3 *Dopo Relax(u, v, w) abbiamo che $d(v) \leq d(u) + w(u, v)$*

Lemma 6.4.4 *Dopo la chiamata a ISS abbiamo che:*

Algoritmo 6.10 InitializeSingleSource(G, s)

InitializeSingleSource(G, s)

```
1: for each v ∈ V do
2:     d(v) ← ∞
3:     π(v) ← NIL
4: end for
5: d(s) ← 0
```

1. $\forall\, v \in V\ d(v) \geq \delta(s, v)$
2. *la 1) è mantenuta dopo una qualunque seguenza di passi di rilassamento.*

Dimostrazione. Per induzione sul numero di chiamate a Relax.

$n = 0$ Segue direttamente dalla definizione di ISS

$n > 0$ Per assurdo. Supponiamo (u, v) essere il primo arco tale che durante il suo rilassamento abbiamo: $d(v) < \delta(s, v)$. Dopo la chiamata a Relax otteniamo:

$$d(u) + w(u, v) = d(v) < \delta(s, v) \leq^{(*)} \delta(s, u) + w(u, v)$$

$(*)$ per il lemma 6.4.2

quindi:

$$d(u) + w(u, v) < \delta(s, u) + w(u, v) \Rightarrow d(u) < \delta(s, u)$$

Assurdo, poiché avevamo supposto v come primo nodo sul quale falliva e $d(u)$ non è stato modificato durante l'operazione di rilassamento di (u, v).

■

Osservazioni:

$d(v)$ può solamente diminuire il suo valore e quindi, per il lemma 6.4.4 se viene a raggiungere il valore $\delta(s, v)$ rimane fisso.

◇

Lemma 6.4.5 *Sia $s \rightsquigarrow u \rightarrow v$ uno shortest path da s a v, supponiamo siano già stati eseguiti ISS e una sequenza di passi di rilassamento tra cui relax(u, v, w).*

Se $d(u) = \delta(s, u)$ prima di relax(u, v, w) allora $d(v) = \delta(s, v)$ dopo tale chiamata.

Dimostrazione. $d(v) \leq d(u) + w(u, v) =^{(*)} \delta(s, u) + w(u, v) =^{(**)} \delta(s, v)$

$(*)$ per ipotesi
$(**)$ per l'ipotesi e il corollario al lemma 6.4.1

quindi $d(v) \leq \delta(s,v)$, per il lemma 6.4.4 $d(v) \geq \delta(s,v)$ e quindi $d(v) = \delta(s,v)$ ■

Lemma 6.4.6 *Se G non contiene cicli negativi raggiungibili da s dopo ISS ed un arbitrario numero di passi di rilassamento allora G_π è un albero di radice s.*

Dimostrazione. Per induzione sul numero di passi di rilassamento dimostriamo che G_π è aciclico. Ragionando per assurdo sia:

$$c = \langle v_0, \ldots, v_k \rangle$$

un ciclo in G_π cioè $v_k = v_0$. Possiamo assumere, senza perdita di generalità, che (v_{k-1}, v_k) sia l'arco che rilassato abbia generato il ciclo. Poiché $\pi(v_i) = v_{i-1}\ \forall\, i \in \{1, \ldots, k-1\}$ l'ultimo update fatto a $v_i\, \forall\, i$ deve essere:

$$d(v_i) \leftarrow d(v_{i-1}) + w(v_{i-1}, v_i)$$

quindi appena prima di rilassare l'arco (v_{k-1}, v_k) avremo:

$$\begin{array}{lcll} d(v_i) & \geq & d(v_{i-1}) + w(v_{i-1}, v_i) & \forall\, i \in \{1, \ldots, k-1\} \\ d(v_k) & > & d(v_{k-1}) + w(v_{k-1}, v_k) & i = k \end{array}$$

Segue che:

$$\begin{aligned} \sum_{i=1}^{k} d(v_i) &> \sum_{i=1}^{k} (d(v_{i-1}) + w(v_{i-1}, v_i)) \\ &> \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

e quindi:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

assurdo perché avremo un ciclo negativo.

Concludiamo provando che G_π è un albero di radice s per induzione sul numero di passi di rilassamento.

$n = 0$ ovvio

$n > 0$ dopo l'ultima chiamata a relax(u, v, w) modifichiamo solo $d(v)$ e $\pi(v)$ e tali valori vengono modificati solo se $d(v) > d(u) + w(u, v) \Rightarrow \pi(u) \neq \text{NIL}$ per cui dall'ipotesi induttiva u è raggiungibile da s in G_π e quindi v è raggiungibile da s in G_π

■

Lemma 6.4.7 *Se G non contiene cicli negativi raggiungibili da s, dopo ISS ed una qualunque sequenza di passi di rilassamento tale che $\forall\, v \in V\ d(v) = \delta(s,v)$ abbiamo che G_π è lo shortest path tree.*

Dimostrazione. Proviamo che:

- V_π è l'insieme dei nodi raggiungibili da s.

  Dim. v raggiungibile da s se e solo se $d(v) = \delta(s,v) \Leftrightarrow \pi(v) \neq \text{NIL} \Leftrightarrow v \in V_\pi$

- G_π è un albero di radice s.

  Dim. Segue dal lemma 6.4.6

- $\forall\, v \in V_\pi$ il cammino da s a v in G_π è uno shortest path.

  Dim. Sia $s \overset{p}{\rightsquigarrow} v$ in G_π con $p = \langle v_0, \ldots, v_k \rangle$ dove $v_0 = s$ e $v_k = v$, come per il lemma 6.4.6:

$$\begin{aligned}
d(v_i) &\geq d(v_{i-1}) + w(v_{i-1}, v_i) \Leftrightarrow \\
\delta(s, v_i) &\geq \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \Leftrightarrow \\
w(v_{i-1}, v_i) &\leq \delta(s, v_i) - \delta(s, v_{i-1}) \Leftrightarrow \\
\sum_{i=1}^{k} w(v_{i-1}, v_i) &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \Leftrightarrow \\
\sum_{i=1}^{k} w(v_{i-1}, v_i) &\leq \delta(s, v_k) - \delta(s, v_0) = \delta(s, v)
\end{aligned}$$

■

Algoritmo 6.11 Dijkstra(G, w, s)

Dijkstra(G, w, s)

```
1: ISS(G, s)
2: Q ← V
3: S ← ∅
4: while Q ≠ ∅ do
5:    u ← EstractMin(Q)
6:    S ← S ∪ {u}
7:    for each v ∈ adj[u] do
8:       Relax(u, v, w)
9:    end for
10: end while
```

Quanto detto fino ad ora ci permette di enunciare e successivamente dimostrare il seguente teorema.

Teorema 6.4.1 *Dijkstra*(G, w, s) *se non ci sono archi negativi è tale che:*

1. *termina;*
2. $\forall\, v \in V\ d(v) = \delta(s, v)$ *alla fine.*

Dimostrazione. Dimostriamo i due casi separatamente:

1. il ciclo while viene eseguito al più $|Q|$ volte perché l'unica operazione eseguita sulla coda è l'estrazione del minimo. Inoltre il ciclo for viene eseguito al più $|adj[u]|$ volte, per qualche u, ad ogni iterazione.
2. Per assurdo. Sia u il primo vertice inserto in S tale che $d(u) \neq \delta(s,u)$ notiamo che la sorgente $s \in S$ e $s \neq u$. Poiché $d(u) \neq \delta(s,u)$ u deve essere raggiungibile da s (perché $d(u)$ approssima $\delta(s,u)$ dall'alto e quindi se non è raggiungibile $d(u) = \infty$ e $\delta(s,u) = \infty$ e quindi non sono diversi.

   Sia p uno shortest path da s a u, sia y primo nodo dello shortest path tale che $y \in S \setminus V$ e x predecessore di y con x nello shortest path tale che $x \in S$. A questo punto sicuramente $d(x) = \delta(s,x)$ poiché u è il primo nodo su cui è fallita la proprietà. Inoltre $s \overset{p}{\rightsquigarrow} u$ è uno shortest path e quindi per il lemma 6.4.1 $s \rightsquigarrow y$ è uno shortest path e poiché quando abbiamo inserito x in S abbiamo rilassato tutti gli archi uscenti da x abbiamo rilassato anche l'arco (x,y) (ultimo arco in uno shortest path) e quindi $d(y) = \delta(s,y)$ per il lemma 6.4.5.

   Riassumendo:

$$d(y) = \delta(s,y) \overset{lemma\ 6.4.2}{\leq} \delta(s,u) \overset{lemma\ 6.4.4}{\leq} d(u)$$

   poiché all'istruzione successiva inseriamo $u \in S$ deve essere anche $d(u) \leq d(y)$, perché viene scelto l'elemento minimo e quindi: $d(y) = \delta(s,y) = \delta(s,u) = d(u)$, assurdo.

■

Per determinare la complessità dell'algoritmo è necessario specificare prima quale tecnica useremo per implementare la coda di priorità Q. Ci sono almeno due modi che analizzeremo separatamente:

Array che memorizza i valori $d()$.

Con questa tecnica l'inizializzazione (linee 1-3) viene a costare $O(|V|)$; il costo di tutte le iterazioni del for è pari a $O(|E|)$ mentre il ciclo while esegue un'istruzione (linea 5) di costo $O(|V|)$ un numero di volte pari a $O(|V|)$ per una complessità totale del ciclo di $O\big(|V|^2\big)$. Se consideriamo ora entrambi i cicli otteniamo complessità di:

$$O\big(|V|^2 + |E|\big) = O\big(|V|^2\big)$$

che possiamo considerare ottimale in presenza di grafi densi.

Min-Heap che memorizza i valori $d()$.

Analogamente a prima l'inizializzazione viene a costare $O(|V|)$, tutte le iterazioni del ciclo for ora vengono a costare $O(|E|\log|V|)$; le istruzioni, diverse dal for, presenti nel ciclo while hanno costo di $O(|V|\log|V|)$ per un totale di:

$$O(|E|\log|V|) + O(|V|\log|V|) = O((|V| + |E|)\log|V|)$$

quindi più adatto per grafi sparsi.

6.4.2 Bellman-Ford

Questo algoritmo risolve il problema della determinazione del cammino di costo minimo da una data sorgente anche in presenza di archi di peso negativo.

Algoritmo 6.12 Bellman-Ford(G, w, s)

Bellman-Ford(G, w, s)

```
1: ISS(G, s)
2: for i ← 1 to |V| − 1 do
3:    for each (u, v) ∈ E do
4:       Relax(u, v, w)
5:    end for
6: end for
7: for each (u, v) ∈ E do
8:    if d(v) > d(u) + w(u, v) then
9:       return FALSE
10:   end if
11: end for
12: return TRUE
```

La complessità risulta essere $O(|V||E|)$.

Lemma 6.4.8 *Se G non contiene cicli negativi raggiungibili da s allora dopo Bellman-Ford(G, w, s) risulta che $\forall\, v \in V\ d(v) = \delta(s, v)$.*

Dimostrazione. Ogni shortest path in G contiene al più $|V| - 1$ archi. Sia $\langle v_0, \ldots, v_k \rangle$ con $v_0 = s$ e $v_k = v$ uno shortest path da s a v.

Proviamo per induzione su $i \in \{0, \ldots, k\}$ che dopo i iterazioni del primo ciclo for vale che $d(v_i) = \delta(s, v_i)$.

$i = 0$ $d(v_0) = \delta(s, s)$, ok.

$i > 0$ $d(v_{i-1}) = \delta(s, v_{i-1})$ per ipotesi induttiva dopo $i - 1$ iterazioni. Alla i-esima iterazione del ciclo for rilassiamo tutti gli archi uscenti da v_{i-1}, in particolare l'arco (v_{i-1}, v_i), quindi $d(v_i) \leq \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$ ottnendo $d(v_i) = \delta(s, v_i)$.

Da questo segue la tesi poiché il primo for viene eseguito $|V|-1$ volte e uno shortest path ha al più $|V| - 1$ archi. ■

Teorema 6.4.2 *Bellman-Ford ritorna* TRUE *e $\forall\, v\ \delta(s, v) = d(v)$ e G_π è lo shortest path tree se e solo se G non contiene cicli negativi raggiungibili da s.*

Dimostrazione. Analizziamo separatamente i due versi dell'implicazione:

$\Leftarrow$ G non contiene cicli negativi raggiungibili da s allora $\forall\, v \in V\ \delta(s,v) = d(v)$ e quindi $\forall\,(u,v) \in E\ d(v) = \delta(s,v) \leq \delta(s,u) + w(u,v) = d(u) + w(u,v)$ quindi Bellman-Ford ritorna TRUE .

$\Rightarrow$ Per contrapposizione. Supponiamo che G contenga un ciclo $c = \langle v_0, \ldots, v_k \rangle$ negativo raggiungibile da s:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \quad (*)$$

Per assurdo assumiamo che Bellman-Ford(G, w, s) ritorni TRUE , allora $\forall\,(u,v) \in E\ d(v) \leq d(u) + w(u,v)$ quindi $d(v_i) \leq d(v_{i-1}) + w(v_{i-1}, v_i)$ per $i \in \{1, \ldots, k\}$.

$$\sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} (d(v_{i-1}) + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

e quindi si ha:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

Assurdo perché in contrasto con l'ipotesi $(*)$.

■

6.5 All pairs shortest path problem

Consideriamo ora il problema di trovare il cammino minimo tra tutte le coppie di vertici in un grafo. Una prima soluzione, se consideriamo grafi in cui i pesi degli archi non possono essere negativi, può essere di eseguire Dijkstra per ogni nodo ottenendo una complessità che varia da $|V| \cdot O\big(|V|^2 + |E|\big) = O\big(V^3\big)$ nel caso di una implementazione che sfrutta gli array, a $|V| \cdot O(|E| \cdot \log(|V|)) = O(|V| \cdot |E| \log(|V|))$ se implementato con una heap binaria.

Se permettiamo l'uso di archi di peso negativo, siamo costretti ad eseguire Bellman-Ford per ogni nodo, ottenendo una complessità di $O\big(V^2 E\big)$ che nel caso di un grafo denso diventa $O\big(V^4\big)$.

Cercheremo qui di migliorare le performance di algoritmi per la soluzione di questo tipo di problemi. Prima di iniziare specifichiamo le notazioni che saranno via via usate.

6.5.1 Notazioni

L'input e l'output degli algoritmi che analizzeremo si assume siano in forma matriciale. Gli archi con peso negativo sono ammessi, ma assumiamo l'assenza di cicli di peso negativo.

Sia $n = |V|$, la matrice di input $n \times n$ viene indicata con W e i suoi elementi w_{ij} sono così definiti:

$$w_{ij} = \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } i \neq j \text{ e } (i,j) \in E \\ \infty & \text{se } i \neq j \text{ e } (i,j) \notin E \end{cases}$$

L'output viene fornito con la matrice D anch'essa $n \times n$ i cui elementi, indicati con d_{ij}, sono così definiti:

$$d_{ij} = \delta(i,j)$$

assumendo che $\delta(i,j)$ indichi il cammino minimo dal nodo i al nodo j.

Sfortunatamente abbiamo bisogno di un'ulteriore matrice, indicata con $\prod = (\pi_{ij})$ e chiamata matrice dei predecessori, dove:

$$\pi_{ij} = \begin{cases} \text{NIL} & \text{se } i = j \text{ oppure } i \not\rightsquigarrow j \\ v & \text{un qualche predecessore di } j,\ v \in i \overset{min}{\rightsquigarrow} j \end{cases}$$

Come G_π è un albero dei cammini minimi così l'i-esima riga della matrice $\prod$ è a sua volta un albero dei cammini dei cammini con radice il nodo i.

Per ogni vertice $i \in V$ definiamo poi il sottografo dei predecessori, *predecessor subgraph*, come $G_{\pi,i} = (V_{\pi\, i}\, ,\, E_{\pi\, i})$ dove:

$$\begin{aligned} V_{\pi\, i} &= \{j \in V : \pi_{ij} \neq \text{NIL}\ \} \cup \{i\} \\ E_{\pi\, i} &= \{(\pi_{ij}\, ,\, j) :\ j \in V_{\pi\, i} \text{ e } \pi_{ij} \neq \text{NIL}\ \} \end{aligned}$$

6.5.2 Soluzione generica

Una soluzione generale, che funziona anche in presenza di archi di peso negativo basata sulla tecnica del *dynamic programming* è strutturata nel seguente modo:

1. Caratterizzare la struttura della soluzione
2. Definizione ricorsiva della soluzione
3. Computazione della soluzione in modo bottom-up

Schematicamente:

$$W \rightsquigarrow d_{m-1} \rightarrow d_m \rightsquigarrow D$$

Possiamo vedere W come la matrice dei costi dei cammini di lunghezza 1, quindi a partire da W possiamo determinare la matrice dei costi dei cammini di lunghezza 2. Seguendo questo schema possiamo arrivare alla soluzione poiché D è la matrice dei costi dei cammini di lunghezza $|V| - 1$.

Cerchiamo di ridefinire alcuni elementi in modo da rendere più chiaro lo schema. Sia $D^1 \equiv W$ e $D^n \equiv D$ possiamo scrivere:

$$D^1 \rightsquigarrow D^m \rightsquigarrow D^n$$

useremo anche la matrice $D^0 = (d^0_{ij})$ definita nel modo seguente:

$$d^0_{ij} = \begin{cases} 0 & i = j \\ \infty & i \neq j \end{cases}$$

Vediamo ora come definire ricorsivamente la soluzione come richiesto al punto 2) dello schema generale proposto. Calcoliamo dunque (d^m_{ij}) partendo da (d^{m-1}_{ij}):

$$d^m_{ij} = \min\left(d^{m-1}_{ij}, \min_{1\leq k\leq n}\left(d^{m-1}_{ik} + w_{kj}\right)\right)$$

Dato che $w_{jj} = 0$ possiamo semplificare l'espressione precedente e scrivere semplicemente:

$$d^m_{ij} = \min_{1\leq k\leq n}\left(d^{m-1}_{ik} + w_{kj}\right) \tag{6.3}$$

quindi d^m_{ij} è il minimo costo di un cammino da i a j contenente al più m archi.

Soluzione iterativa

L'input è la matrice $W = (w_{ij})$, calcoliamo una serie di matrici $D^1, D^2, \ldots, D^{n-1}$ dove per $m = 1, \ldots, n-1$ abbiamo:

$$D^m = \left(d^m_{ij}\right)$$

la matrice finale D^{n-1} contiene i cammini minimi, noto $d^1_{ij} = w_{ij}\ \forall\, i, j \in V$ abbiamo che $D^1 = W$ e quindi il cuore della nostra soluzione sarà una procedura che computi correttamente D^m a partire da D^{m-1}. Questa estende i cammini minimi con al più $m-1$ archi nei cammini minimi con al più m archi (algoritmo 6.13).

Il tempo di esecuzione è chiaramente $\Theta\left(n^3\right)$ dovuto ai tre cicli for annidati. Come si può notare l'algoritmo ha la stessa struttura di un algoritmo per la moltiplicazione di matrici $n \times n$.

Supponiamo di voler calcolare il prodotto di $C = A \cdot B$, allora $i, j = 1, \ldots, n$ calcoliamo:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{6.4}$$

se effettuiamo le seguenti sostituzioni:

Algoritmo 6.13 ExtendShortestPath(D, W)

ExtendShortestPath(D, W)

```
1: n ← row(D)
2: D' = (d'_ij)
3: for i ← 1 to n do
4:    for j ← 1 to n do
5:       d'_ij ← ∞
6:       for k ← 1 to n do
7:          d'_ij ← min(d'_ij , d_ik + w_kj)
8:       end for
9:    end for
10: end for
11: return D'
```

d^{m-1}	$\rightarrow$	a
w	$\rightarrow$	b
d^m	$\rightarrow$	c
min	$\rightarrow$	$+$
$+$	$\rightarrow$	$\cdot$

Dall'equazione 6.3 otteniamo l'equazione 6.4. Giunti a questo punto è sufficiente sostituire ∞, l'identità di min, con 0, l'identità della somma e otteniamo un algoritmo con complessità $\Theta(n^3)$ per la moltiplicazione di matrici.

Algoritmo 6.14 MatrixMultiply(A, B)

MatrixMultiply(A, B)

```
1: n ← rows(A)
2: C = (c_ij) {matrice n × n}
3: for i ← 1 to n do
4:    for j ← 1 to n do
5:       c_ij ← 0
6:       for k ← 1 to n do
7:          c_ij ← c_ij + a_ik · b_kj
8:       end for
9:    end for
10: end for
11: return C
```

Ritorniamo ora al nostro problema principale, fino ad ora abbiamo calcolato il peso dei cammini minimi estendendoli arco dopo arco. Sia $A \cdot B$ la matrice prodotto calcolata dall'algoritmo 6.14 con A, B come parametri, allora calcoliamo la sequenza di $n-1$ matrici:

$$\begin{array}{lclcl} D^0 & = & D^0 \cdot W & = & W \\ D^1 & = & D^1 \cdot W & = & W^2 \\ D^2 & = & D^2 \cdot W & = & W^3 \\ & \vdots & & \vdots & \\ D^{n-1} & = & D^{n-2} \cdot W & = & W^{n-1} \end{array}$$

Come detto in precedenza la matrice $D^{n-1} = W^{n-1}$ contiene il peso dei cammini minimi. La seguente procedura calcola la sequenza in $\Theta(n^4)$

Algoritmo 6.15 SlowAPSP(W)

SlowAPSP(W)

```
1: n ← row(D)
2: D' ← W
3: for m ← 2 to n − 1 do
4:     D' ← ExtendShortestPath(D', W)
5: end for
6: return D'
```

Miglioriamo i tempi

Come si nota facciamo del lavoro in più, infatti determiniamo tutte le matici D^m, mentre ci interessa solamente D^{n-1}. Sappiamo che in caso di assenza di cicli negativi $D^m = d^{n-1}\ \forall m \geq n-1$. Possiamo quindi calcolare D^{n-1} con solamente $\lceil \log(n-1) \rceil$ prodotti di matrice, calcolando la sequenza:

$$\begin{array}{lclcl} D^1 & = & W & & \\ D^2 & = & W^2 & = & W \cdot W \\ D^4 & = & W^4 & = & W^2 \cdot W^2 \\ D^8 & = & W^8 & = & W^4 \cdot W^4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ D^{2^{\lceil \log(n-1) \rceil}} & = & W^{2^{\lceil \log(n-1) \rceil}} & = & W^{2^{\lceil \log(n-1) \rceil}-1} \cdot W^{2^{\lceil \log(n-1) \rceil}-1} \end{array}$$

Dato che $2^{\lceil \log(n-1) \rceil} \geq n-1$, il prodotto finale $D^{2^{\lceil \log(n-1) \rceil}}$ è eguale a D^{n-1}.

L'algoritmo 6.16 computa la suddetta sequenza di matrici sfruttando le considerazioni appena citate, ovvero la tecnica del *repeated squaring*. Il tempo di esecuzione è $\Theta(n^3 \log n)$ in quanto ognuno dei $\lceil \log n \rceil$ prodotti consuma tempo $O(n^3)$.

6.5.3 Algoritmo di Floyd Warshall

Sfruttiamo ora una diversa formalizzazione del problema. Sia $G = (V, E)$ il grafo in input e sia $V = \{1, \ldots, n\}$ l'insieme dei vertici. Indicheremo con d_{ij}^k il costo del cammino minimo da i a j passante per nodi aventi indice in

Algoritmo 6.16 FasterAPSP(W)

FasterAPSP(W)

1: $n \leftarrow \text{row}(D)$
2: $D' \leftarrow W$
3: $m \leftarrow 1$
4: **while** $n - 1 > m$ **do**
5: $\quad D^{2m} \leftarrow \text{ExtendShortestPath}(D^m, D^m)$
6: **end while**
7: return D^m

$\{1, \ldots, k\}$. L'algoritmo permette la presenza di archi negativi nel grafo di input ma non di cicli di peso negativo.

Cerchiamo ora di determinare la matrice (d_{ij}^k) a partire da (d_{ij}^{k-1}). Abbiamo due possibilità:

- k non è un vertice appartenente al cammino da i a j di indice fino a k, quindi il cammino minimo appartiene a d_{ij}^{k-1};
- k è nel cammino da i a j di indice fino a k, allora possiamo spezzare il cammino in due cammini $i \rightsquigarrow k$ e $k \rightsquigarrow j$ dal lemma 6.4.1 segue che entrambi sono due cammini minimi e conseguentemente la loro somma è un cammino minimo e il costo dei due cammini è presente in d_{ik}^{k-1} e d_{kj}^{k-1}

Algoritmo 6.17 Floyd-Warshall(W)

Floyd-Warshall(W)

1: $n \leftarrow \text{row}(D)$
2: $D \leftarrow W$
3: **for** $k \leftarrow 1$ to n **do**
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad$ **for** $j \leftarrow 1$ to n **do**
6: $\quad\quad\quad d_{ij}^k \leftarrow \min\left(d_{ij}^{k-1},\, d_{ik}^{k-1} + d_{kj}^{k-1}\right)$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

Il costo di esecuzione di Floyd-Warshall è $O\left(n^3\right)$ determinato dalla presenza dei tre cicli for annidati.

6.5.4 Algoritmo di Johnson

Cerchiamo una soluzione all'APSPP che migliori Floyd-Warshall tanto quanto l'implementazione di Dijkstra con la heap migliora l'implementazione di

Dijkstra con l'array (nel caso di grafi sparsi). Definiamo ora un algoritmo basato su n applicazioni di Dijkstra (implementato usando una heap) che funzioni anche in presenza di archi negativi.

Dobbiamo ora introdurre la tecnica del ripesamento, *reweighting*. Una prima idea potrebbe portarci a ridefinire i pesi degli archi aggiungendo a tutti una quantità costante, sfortunatamente non funziona.

Quello che vorremmo definire è un ripesamento che non modifichi i cammini minimi, formalmente data una funzione peso $w : E \to \mathbb{R}$ cerchiamo $\hat{w}$ tale che:

1. $\forall\, u, v \in V$ un cammino minimo da u a v calcolato secondo w lo sia anche secondo $\hat{w}$.

2. $\forall\, u, v \in V\ \hat{w}(u, v) \geq 0$

Si può determinare una famiglia di $\hat{w}$ che soddisfino 1) e 2) a partire da una opportuna funzione $h : V \to \mathbb{R}$ seguendo il seguente schema:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \tag{6.5}$$

Indicheremo da ora con δ il peso di un cammino minimo calcolato secondo w, mentre con $\hat{\delta}$ il peso del cammino minimo calcolato secondo $\hat{w}$.

Lemma 6.5.1 *$\forall\, u, v \in V$ e $\forall p : u \overset{p}{\rightsquigarrow} v$ si ha $w(p) = \delta(u, v) \Leftrightarrow \hat{w}(p) = \hat{\delta}(u, v)$.*

Se G ha un ciclo negativo secondo w allora G ha un ciclo negativo secondo $\hat{w}$.

Dimostrazione. Prima di tutto dimostriamo che la definizione di $\hat{w}$ generalizza ai cammini, cioè $\hat{w}(p) = w(p) + h(u) - h(v)$.

Sia $p = \langle v_0, \ldots, v_k \rangle$ dove $v_0 = u$ e $v_k = v$ allora, base alla definizione 6.5 vale:

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1},\, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1},\, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \left(\sum_{i=1}^{k} w(v_{i-1},\, v_i) \right) + h(v_0) - h(v_k) \text{ perchè telescopica} \\
&= w(p) + h(u) - h(v)
\end{aligned}$$

Se $w(p) = \delta(u, v)$ dobbiamo provare che $\hat{w}(u, v) = \hat{\delta}(u, v)$. Per assurdo sia $u \overset{p'}{\rightsquigarrow} v$ tale che $\hat{w}(p') < \hat{w}(p)$ allora $\hat{w}(p_1) + h(u) - h(v) < w(p) + h(u) - h(v) = \hat{w}(p)$ da cui si ricava che $w(p') < w(p)$, assurdo perchè p era un cammino minimo secondo w.

Se $\hat{w}(p) = \hat{\delta}(u,v)$ e dobbiamo provare che $w(p) = \delta(u,v)$. Per assurdo $\hat{w}(p') = w(p') + h(u) - h(v) < w(p) + h(u) - h(v) = \hat{w}(p)$ assurdo poichè p cammino di peso minimo secondo $\hat{w}$.

Infine supponiamo che $c = \langle v_0, \ldots, v_k \rangle$ sia un ciclo negativo in G secondo $\hat{w}$ ottenendo $\hat{w}(c) < 0 \Leftrightarrow w(c) + h(v_0) - h(v_k) < 0 \Leftrightarrow < 0$ e quindi avrei un ciclo negativo in G secondo w. ■

Quindi la proprietà 1) di $\hat{w}$ è garantita vediamo ora come verificare la 2).

Vorremmo ottenere $\hat{w}(u,v) \geq 0$ per tutti gli archi $(u,v) \in E$. Consideriamo il grafo di partenza G orientato e pesato con funzione peso w e inserendo un *nuovo* nodo s, costruiamo un nuovo grafo $G' = (V', E')$ dove:

$$\begin{aligned} V' &\equiv V \cup \{s\} \\ E' &\equiv E \cup \{(s,v) : \ v \in V\} \end{aligned}$$

Procediamo ora estendendo la funzione peso w nel seguente modo: $w(s,v) = 0$ per ogni $v \in V$.

Dato che s non ha archi entranti, non esistono cammini minimi contenenti s diversi da quelli che partono da s, inoltre G' non possiede cicli negativi se e solo se non erano presenti nemmeno in G.

Supponiamo ora che G e G' siano entrambi privi di cicli negativi, definiamo $h(v) = \delta(s,v)$ per ogni nodo $v \in V'$. Da quanto dichiarato al lemma 6.4.2 $h(v) \leq h(u) + w(u,v)$ per ogni arco $(u,v) \in E'$, quindi se definiamo una nuova funzione peso $\hat{w}(u,v)$ secondo l'equazione 6.5 otteniamo:

$$w(u,v) = w(u,v) + h(u) - h(v) \geq 0$$

Il costo dell'algoritmo 6.18 è $O\big(V^2 \log V + VE\big)$ se Dijkstra è implementato con heap di Fibonacci, mentre nel caso di implementazione con semplici heap binarie è $O(VE \log V)$ che comunque rimane asintoticamente migliore di Floyd-Warshall in presenza di grafi sparsi.

Algoritmo 6.18 Johnson(G)

Johnson(G)

1: $V' \leftarrow V \cup \{s\}$
2: $E' \leftarrow E \cup \{(s,u) : u \in V\}$
3: **if** BellmanFord(G', w, s) = FALSE **then**
4: return FALSE
5: **else**
6: **for** each $vinV'$ **do**
7: $h(v) \leftarrow \delta(s,v)$ {calcolato da B.F.}
8: **end for**
9: **for** each $(u,v) \in E'$ **do**
10: $\hat{w}(u,v) \leftarrow w(u,v) + h(u) - h(v)$
11: **end for**
12: **for** each $u \in V$ **do**
13: Dijkstra(G, $\hat{w}$, u) {calcoliamo $\hat{\delta}(u,v)$}
14: **for** each $v \in V$ **do**
15: $d_{uv} \leftarrow \hat{\delta}(u,v) + h(v) - h(u)$
16: **end for**
17: **end for**
18: **end if**
19: return D

6.6 Esercizi

Esercizio 6.1 *(tratto dall'esame di ASD del 09-12-03)*

Sia $G = \langle V, E \rangle$ un grafo non orientato, connesso e pesato. Si supponga di aver determinato un *minimum spanning tree* di G, T, avente radice v. Si assuma che il peso dell'arco $(u, v) \in E$ venga decrementato di 1 e si consideri il problema di determinare, dato T, un minimum spanning tree del grafo modificato.

- Si definisca un algoritmo efficiente per il problema proposto e se ne valutino correttezza e complessità.

(Suggerimento: si pensi ad un taglio che separi il nodo u dal nodo v ...)

Esercizio 6.2 *(tratto dall'esame di ASD del 02-09-03)*

Sia dato un grafo $G = \langle V, E \rangle$ non orientato e si considerino i seguenti problemi:

a) stabilire se G é un *albero*;
(Suggerimento: si osservi che, ovviamente, se esiste $r \in V$ tale che G é un albero di radice r, allora per ogni $r' \in V$, G é un albero di radice r')

b) dato $k \in \mathbb{N}$, stabilire se esiste un nodo $r \in V$ tale che G é un albero di altezza k e radice r.

Si definisca un algoritmo efficiente per ciascuno dei problemi proposti. Si dimostri la correttezza delle procedure delineate e se ne valuti la complessitá.

Esercizio 6.3 *(tratto dall'esame di ASD del 07-07-03)*

Si consideri un grafo, $G = \langle V, E \rangle$, orientato ed aciclico. Dato un vertice $v \in V$, sia *d-foglie*(v) il minimo numero di archi in un cammino da v ad una foglia del grafo. Si denoti invece mediante *D-foglie*(v), il massimo numero di archi in un cammino da v ad una foglia del grafo.

- Si proponga un algoritmo efficiente che, dato un grafo orientato aciclico G ed un nodo v, determini *d-foglie*(v). Si valuti la complessità della procedura proposta.

- Si proponga un algoritmo efficiente che, dato un grafo orientato aciclico G ed un nodo v, determini *D-foglie*(v). Si valuti la complessità della procedura proposta.

- **Facoltativo** Si proponga un algoritmo efficiente che associ ad *ogni* vertice di un grafo orientato ed aciclico G, i valori *D-foglie* e *d-foglie*.

Esercizio 6.4 *(tratto dall'esame di ASD del 02-12-02)*

Si consideri un grafo non orientato e pesato $G = \langle V, E \rangle$ e si considerino gli algoritmi di Prim e Kruskal visti a lezione. Si supponga G non connesso.

Qual è la l'output prodotto da ognuno dei due algoritmi menzionati su G?

Qual è la complessità dei due algoritmi menzionati su G?

Esercizio 6.5 *(tratto dall'esame di ASD del 02-09-02)*

Sia $G = (V, E)$ un grafo non orientato e pesato, con funzione di peso $w : E \to \Re^+$.

a Si descriva brevemente un algoritmo efficiente (per esempio tra quelli visti a lezione) che determini un *Minimum Spanning Tree* per G e se ne valuti la complessità.

b Si provi o si refuti la seguente proprietà: dato un *Minimum Spanning Tree* per G questo contiene almeno un arco tra quelli aventi peso minimo in G.

c Si provi o si refuti la seguente proprietà: dato un *Minimum Spanning Tree* per G questo contiene un cammino minimo per ogni coppia di nodi in G.

Esercizio 6.6 *(tratto dall'esame di ASD del 18-07-02)*

Sia $G = (V, E)$ un grafo orientato.

- Si proponga un algoritmo efficiente che determini se G è un albero, valutandone la complessità e dimostrandone la correttezza.
- Si proponga un algortimo efficiente che determini se G è trasformabile in un albero mediante l'eliminazione di un opportuno numero di archi. Si valuti la complessità e si dimostri la correttezza dell'algoritmo proposto.

Esercizio 6.7 *(tratto dall'esame di ASD del 18-06-02)*

Sia $G = (V, E)$ un grafo non orientato, connesso e con un numero di archi uguale al numero di nodi.

a Si proponga un algoritmo efficiente che determini un Minimum Spanning Tree di G, valutandone la complessità e dimostrandone la correttezza.

b Diciamo *Secondo Migliore* Minimum Spanning Tree di G, uno Spanning Tree T' di G tale che $T < T' \leq T''$ per ogni Minimum Spanning Tree T di G e per ogni Spanning Tree T'' di G.

Si proponga un algoritmo efficiente che determini, se esiste, un Secondo Migliore Minimum Spanning Tree di G valutandone la complessità e dimostrandone la correttezza.

Esercizio 6.8 *(tratto dall'esame di ASD del 24-09-01)*

Si proponga un algoritmo che, dato in ingresso un grafo non orientato e pesato $G = (V, E)$ ed un suo arco $(u, v) \in E$, determini l'esistenza di un MST per G *non* contenente l'arco (u, v). Si motivi la correttezza e si calcoli la complessità dell'algoritmo proposto. Si riconsiderino infine il problema, nel caso in cui si voglia determinare l'esistenza di un MST per il grafo G contenente l'arco (u, v)

Esercizio 6.9 *(tratto dall'esame di ASD del 10-09-01)*

Definiamo *grafo colorato* un grafo non orientato $G = (V, E)$, in cui agli elementi di V è associato un campo *colore* che può assumere uno tra due possibili valori (diciamo *bianco* e *nero*). Definiamo inoltre un *cammino alternato* un cammino in cui ogni coppia di nodi consecutivi hanno colore diverso, in altre parole un cammino alternato contiene unicamente archi che collegano nodi con colore diverso. Infine definiamo un *cammino quasi alternato* un cammino che contiene al più una coppia di nodi consecutive dello stesso colore, mentre tutte le altre coppie di nodi consecutivi hanno colore diverso.

a Si descriva un algoritmo che dati un grafo colorato $G = (V, E)$ ed una coppia di nodi $v, u \in V$, determini un cammino alternato tra v e u con un numero di nodi minimo.

b Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

c Si risolvano i punti precedenti, derminando però in questo caso un cammino quasi alternato.

Esercizio 6.10 *(tratto dall'esame di ASD del 28-06-01)*

Dato un grafo orientato $G = (V, E)$, con la funzione peso $w : E \to \Re^+$ a valori positivi, e due nodi $v, t \in V$.

a Si proponga un algoritmo che determini, oltre ad un cammino minimo p_1 da v a t che supponiamo essere $p_1 = \langle v, s_1, \ldots s_n, t\rangle$, un secondo cammino p_2 che sia il cammino di lunghezza minima tra quelli che congiungono v con t e non contengono l'arco (s_n, t).

b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

c Si proponga un algoritmo che determini oltre un cammino minimo p_1, da v a t, un secondo cammino p_2 che sia il cammino di lunghezza minima tra quelli che congiungono v con t e sono distinti da p_1.

d Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.11 *(tratto dall'esame di ASD del 05-06-01)*

Dato un grafo orientato $G = (V, E)$, si assuma che agli elementi di V sia associato un campo *colore* che può assumere uno tra due possibili valori (diciamo *bianco* e *nero*).

Si proponga un algoritmo che determini l'esistenza di un ciclo contenente un nodo nero. Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.12 *(tratto dal compitino di ASD del 01-06-01)*

Dato un grafo $G = (V, E)$, diremo *grado di connessione* di un nodo v in G il numero di nodi raggiungibili da v in G.

Si proponga un algoritmo che determini il nodo con grado di connessione massimo in un grafo G orientato e aciclico. Si riconsideri il problema nel caso di un grafo non orientato. Si argomenti la correttezza e si valuti la complessità degli algoritmi proposti.

Esercizio 6.13 *(tratto dall'esame di ASD del 14-12-00)*

Si proponga un algoritmo che, dato un grafo G, pesato, orientato e aciclico, e un suo nodo v, determini, tra i cammini uscenti da v, quello di lunghezza massima. Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto. (Suggerimento: modificando il peso degli archi, si riduca il problema ad un problema di ricerca di cammino minimo)

Esercizio 6.14 *(tratto dall'esame di ASD del 14-09-00)*

Sia $G = (V, E)$ un grafo orientato e pesato e sia $s \in V$ un nodo sorgente.

Si proponga un algoritmo che risolva il *single source shortest path problem* a partire da s nell'ipotesi in cui i pesi degli archi possano essere solo 1 o 2. Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto. Si riconsideri infine il problema, nel caso i pesi possano essere nell'insieme $\{1, \ldots, k\}$, con k costante fissa.

Esercizio 6.15 *(tratto dall'esame di ASD del 31-08-00)*

Sia $G = (V, E)$ un grafo non orientato.

a Si proponga un algoritmo che determini se ogni coppia di vertici in G sia collegabile con un cammino contenente al più quattro archi.

b Si proponga un algoritmo che determini l'esistenza di uno spanning tree T per il grafo G tale ogni coppia di vertici sia collegabile con un cammino, in T, contenente al più quattro archi. (Suggerimento: in un albero ogni coppia di vertici sono collegabili con un cammino contenente al più quattro archi se e solo se esiste un nodo raggiungibile da ogni altro nodo percorendo al più due archi.)

c Si discuta la correttezza e si calcoli la complessità degli algoritmi proposti.

Esercizio 6.16 *(tratto dall'esame di ASD del 13-07-00)*

Sia $G = (V, E)$ un grafo pesato ed orientato, in cui la funzione peso $w : E \rightarrow \mathcal{R}$ assume solo valori positivi e sia $v \in V$ un nodo del grafo.

a Si proponga un algoritmo che determini se esistono cicli a cui v appartine e nel caso trovi quello di peso minimo.

b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

c Si proponga un algoritmo che determini il ciclo di peso minimo in G.

Esercizio 6.17 *(tratto dall'esame di ASD del 22-6-00)*

Sia $G = (V, E)$ un grafo orientato,

a Si proponga un algoritmo efficiente che determini se G è un albero.

b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

c Si proponga un algoritmo che, nell'ipotesi G non sia un albero, stabilisca qual è il numero minimo di archi che vanno aggiunti/eliminati per trasformare G in un albero. Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

Esercizio 6.18 *(tratto dal compitino di ASD del 1-6-00)*

Sia $G = (V, E)$ un grafo pesato ed orientato, in cui la funzione peso $w : E \rightarrow \mathcal{R}$ assume un valore negativo su di un unico arco $(u, v) \in E$.

a Si proponga un algoritmo efficiente che determini se nel grafo esiste un ciclo negativo.

b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

c Si considerino i punti precedenti nel caso in cui la funzione peso w assuma valore negativo su due archi $(u_1, v_1), (u_2, v_2)$ del grafo G.

d Si suggerisca l'idea per un algoritmo efficiente che determini l'esistenza di cicli negativi in grafi in cui il numero di archi negativi sia molto limitato, per esempio sia minore di $\ln n$ dove n e' il numero dei nodi del grafo.

Esercizio 6.19 *(tratto dall'esame di ASD del 14-2-00)*

Sia $G = (V, E)$ un grafo orientato rappresentato mediante liste di adiacenza. Ricordiamo che un *ordinamento topologico* dei nodi di G è un ordinamento $\prec$ degli elementi di V tale che, per ogni $(u, v) \in E$, si abbia che $u \prec v$.

(a) Si proponga un algoritmo che preso in input G, produca un ordinamento topologico dei nodi di G.

(b) Si modifichi il precedente algoritmo in modo tale che, se esistono più ordinamenti topologici di G, due successive chiamate dell'algoritmo producano differenti ordinamenti.

(c) Si discuta la correttezza degli algoritmi proposti e se ne valuti la complessità.

Esercizio 6.20 *(tratto dall'esame di ASD del 30-9-99)*

Sia $G = (V, E)$ un grafo non orientato pesato, tale che ogni arco abbia peso negativo.

a Si proponga un algoritmo che risolva il problema della determinazione dei cammini minimi da una data sorgente $s \in V$.

b Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

c Si riconsideri il precedente problema nel caso G sia orientato

Esercizio 6.21 *(tratto dall'esame di ASD del 2-9-99)*

Sia $G = (V, E)$ un grafo non orientato pesato (pesi positivi e negativi sono ammessi), tale che ogni ciclo non contiene archi negativi.

Si proponga un algoritmo che risolva il problema della determinazione dei cammini minimi da una data sorgente $s \in V$. Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.22 *(tratto dal compitino di ASD del 13-5-99)*

Si consideri un grafo pesato e orientato $G = (V, E)$ con funzione di peso $w : \Re \to E$.

Si proponga un'algoritmo che determini tutti i nodi x che si trovano su un ciclo negativo e, per ognuno di tali x, produca in output un ciclo negativo che contenga x. Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.23 *(tratto dall'esame di ASD del 10-2-99)*

Sia $G = (V, E)$ un grafo pesato non orientato in cui non ci sono due archi di peso uguale e sia T un *minimum spanning tree* di G.

(a) Dato un arco $e = (i, j) \in E$ si dimostri che e appartiene a T se e solo se ogni cammino di lunghezza maggiore o uguale a 2 da i a j contiene un arco di peso maggiore del peso di e. Inoltre si provi o si refuti l'unicitá di T.

(b) Si descriva un algoritmo che dati G e T nelle ipotesi di cui sopra e dato un arco pesato $\bar{e} \notin E$ di peso diverso dal peso di ogni arco in E, ritorni un minimum spanning tree T' per $G' = (V, E \cup \{\bar{e}\})$?

Esercizio 6.24 *(tratto dall'esame di ASD del 28-9-98)*

Sia $G = (V, E)$ un grafo non-orientato e connesso. Diciamo che un vertice $v \in V$ è un *cut-vertice* se la rimozione da G di v (e di tutti gli archi che lo contengono) sconnette G.

Proporre lo pseudo-codice, adeguatamente commentato, di un algoritmo che ricevuto in input G restituisca in output tutti i cut-vertici di G. Provare la correttezza e determinare la complessità dell'algoritmo proposto (Suggerimento: usare la visita in profondità e la classificazione degli archi di G che si ottiene eseguendo tale visita).

Esercizio 6.25 *(tratto dall'esame di ASD del 9-9-98)*

Sia $G = (V, E)$ un grafo diretto, $w : E \to \Re^+$ una funzione peso, $s \in V$ ed $(u, v) \in E$ un arco.

(a) Si proponga un algoritmo per determinare se esiste un cammino minimo da s a v che contiene l'arco (u, v);

(b) si proponga un algoritmo per determinare se esiste un cammino minimo da s a v che non contiene l'arco (u, v);

(c) si provi la correttezza e determini la complessità degli algoritmi proposti.

Esercizio 6.26 *(tratto dall'esame di ASD del 30-6-98)*

Sia dato un grafo $G = (V, E)$ diretto aciclico. Si proponga un algoritmo efficiente che associ ad ogni nodo v la lunghezza del cammino più lungo da v ad una foglia (cioè un nodo privo di archi uscenti). Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.27 *(tratto dal compitino di ASD del 10-6-98)*

Si consideri un grafo orientato e pesato $G = (V, E)$ e si supponga che G contenga una *radice* (cioè un nodo dal quale ogni altro nodo in V risulti raggiungibile).

(a) Si proponga un algoritmo efficiente per determinare se G contiene un ciclo negativo;

(b) si modifichi l'algoritmo proposto al passo precedente in modo da produrre in output i nodi di un ciclo negativo (quando ne esista uno);

(c) si provi la correttezza e si valuti la complessità degli algoritmi proposti.

Esercizio 6.28 *(tratto dall'esame di ASD del 12-2-98)*

Dato un grafo non orientato $G = (V, E)$, diciamo che G è *bipartito* se esistono V_1 e V_2 tali che

- $V_1 \cap V_2 = \emptyset$;
- $V_1 \cup V_2 = V$;
- per ogni $e = (u, v) \in E$, $u \in V_1 \Leftrightarrow v \notin V_2$.

Si fornisca lo pseudo-codice di commentato dettagliatamente di un algoritmo che determini se un grafo G è bipartito. Si provi la correttezza e si determini infine la complessità dell'algoritmo proposto.

Esercizio 6.29 *(tratto dall'esame di ASD del 28-1-98)*

Sia dato un grafo non orientato (non pesato) $G = (V, E)$ ed un insieme di nodi $F \subseteq V$.

Si proponga lo pseudocodice di un algoritmo efficiente che determini, se esiste, un albero di supporto per G in cui ogni nodo di F risulti essere una foglia. Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 6.30 *(tratto dall'esame di ASD del 1-10-97)*

Si dia una risposta motivata ad ognuna delle seguenti domande:

1. Sia T un *minimum spanning tree* di un grafo indiretto G con pesi positivi. Se aggiungiamo a G esclusivamente un vertice v ed archi (di peso positivo) uscenti da v, il peso di un *minimum spanning tree* per il grafo ottenuto è sicuramente superiore al peso di T?

2. Tutti gli *spanning trees* (sia minimi che non) di un grafo G hanno lo stesso numero di archi?

3. Se aggiungiamo archi ad un grafo diretto aciclico, il grafo ottenuto è ancora un grafo diretto aciclico?

4. Se cancelliamo archi da un grafo diretto aciclico, il grafo ottenuto è ancora un grafo diretto aciclico?

5. Il grafo indiretto ottenuto eliminando la direzione degli archi da un grafo diretto aciclico, è aciclico?

6. Supponiamo che durante la visita in profondità (DFS) di un grafo diretto G vengano assegnati ad ogni vertice v due numeri: $pre(v)$ e $post(v)$. Tali valori corrispondano, rispettivamente, alle posizioni di v nelle liste in pre-ordine ed in post-ordine dei nodi nel DFS-albero/i ottenuto dalla visita. Sono vere le seguenti affermazioni?

   - se $pre(v) < pre(w)$ e $post(v) > post(w)$ allora c'è un cammino da v a w in G;
   - se $pre(v) < pre(w)$ e $post(v) < post(w)$ allora non c'è un cammino da v a w G;

Esercizio 6.31 *(tratto dall'esame di ASD del 18-9-97)*

Sia $G = (V, E)$ un grafo diretto e aciclico, e siano s e t due vertici distinti in V. Si consideri il problema di determinare un insieme *massimale* di cammini disgiunti (privi di nodi in comune) da s a t.

Si producano G, s, e t tali che esiste un insieme massimale ma non di cardinalità massima, di cammini disgiunti da s a t in G. Si proponga lo pseudo codice commentato di un algoritmo che risolva il problema proposto e se ne valuti la complessità.

Esercizio 6.32 *(tratto dall'esame di ASD del 30-7-97)*

Sia $G = (V, E)$ un grafo diretto e aciclico. Si proponga un algoritmo che determinini un ordinamento $<$ dei nodi di V tale che per $u, v \in V$, se esiste un cammino da u a v in G allora $u < v$. Si dimostri la correttezza e si valuti la complessitá dell'algoritmo proposto.

Esercizio 6.33 *(tratto dall'esame di ASD del 23-6-97)*

Sia $G = (V, E)$ un grafo indiretto e siano s, t ed u tre vertici distinti in V. Si proponga un algoritmo che determini se ogni cammino da s a t passa per u. Si definisca inoltre una procedura un algoritmo che determini se esiste un cammino da s a t passante per u. Si valutino, infine, correttezza e complessità degli algoritmi proposti.

Esercizio 6.34 *(tratto dall'esame di ASD del 14-10-96)*

Sia $G = \langle V, E \rangle$ un grafo indiretto, connesso e aciclico con n nodi.

- Si dimostri che esiste sempre un cammino che percorre tutti gli archi del grafo G in entrambe le direzioni.
- Si proponga lo pseudo-codice commentato di un algoritmo che produca un cammino con le caratteristiche di cui al punto precedente.
- Si valuti la complessità dell'algoritmo proposto.

Capitolo 7

Raccolta Esercizi D'Esame

§ **Esame di ASD del 13-12-04**

Esercizio 1: Sia $A[n, n]$ una matrice quadrata di interi distinti. Si supponga che le righe di A siano ordinate in modo crescente da sinistra verso destra. Si assuma, inoltre, che le colonne di A siano ordinate in modo crescente dal basso verso l'alto. Si consideri il problema di determinare se, dato un intero k, k compare in A.

(1.a) Si definisca un algoritmo di complessitá strettamente inferiore ad $\mathcal{O}(n^2)$ per il problema proposto;

(1.b) si valuti la correttezza e la complessità dell'algoritmo definito al passo precedente.

Esercizio 2: Sia $G = \langle V, E \rangle$ un grafo non orientato sui cui archi é definita la funzione di peso $w : E \mapsto \mathbb{N}$. Si consideri una versione modificata dell'algoritmo di Kruskal, che *non* esegue alcun ordinamento preliminare degli archi di G.

(2.a) che cosa restituisce in output l'algoritmo delineato?

(2.b) si produca lo pseudocodice della procedura proposta e se ne valuti la complessità, assumendo che la disjoint set forest, utilizzata nell'algoritmo, venga implementata mediante alberi con le euristiche di union by rank e path compression.

§ **Esame di ASD del 06-09-04**

Esercizio 1: Dato un array $A[1 \ldots n]$, contenente n interi, si consideri il problema di determinare se A contiene più di $\lceil \frac{n}{2} \rceil$ ripetizioni di un suo elemento.

(1.a) Si definisca un algoritmo efficiente per il problema proposto;

(1.b) si valuti la correttezza e la complessità dell'algoritmo proposto al passo precedente.

Esercizio 2: Sia $G = \langle V, E \rangle$ un grafo orientato e si assuma che i nodi in G siano colorati mediante il colore rosso oppure verde. Si consideri il problema di determinare se G possiede un ciclo semplice contenente *solo* nodi verdi.

(2.a) Si proponga un algoritmo efficiente per il problema proposto;

(2.b) si valuti la correttezza e la complessità della procedura delineata;

(2.c) **Facoltativo**: si riconsiderino i punti precedenti nell'ipotesi di determinare se G possiede un ciclo semplice contenente *almeno* un nodo verde.

§ **Esame di ASD del 20-07-04**

Esercizio 1: Sia $A[1 \dots n]$ un vettore contenente $n = 3m$ interi. Si consideri il problema di determinare gli elementi di A maggiori o uguali ad almeno $\frac{n}{3}$ interi in A e minori o uguali ad almeno $\frac{n}{3}$ interi in A.

- Si proponga un algoritmo lineare per risolvere il problema proposto;
- si discuta la correttezza e la complessità dell'algoritmo definito.

Esercizio 2: Si consideri un grafo diretto e pesato $G = \langle V, E \rangle$. Si assuma che la funzione di peso, $w : E \mapsto \mathbb{N}$, assegni ad ogni arco $e \in E$ un peso $1 \leq w(e) \leq k$, dove k é una costante.

- Si proponga un algoritmo efficiente per determinare il peso di un cammino minimo da un nodo $s \in V$, ad un nodo $v \in V$;
- si analizzi la correttezza e la complessità dell'algoritmo proposto.
- **Facoltativo** Si proponga una versione modificata dell'algoritmo di DIJKSTRA che operi in modo lineare nelle ipotesi delineate sopra.

§ **Esame di ASD del 01-07-04**

Esercizio 1: Si consideri l'algoritmo di ordinamento COUNTING-SORT visto a lezione.

1-a Si proponga una versione di COUNTING-SORT che utilizzi soltanto un vettore ausiliario (il vettore C, nella terminologia del testo adottato a lezione), in luogo dei due vettori utilizzati nella versione classica (i vettori B e C).

1-b Rispetto a quale parametro l'algoritmo delineato peggiora le performance della versione di COUNTING-SORT vista a lezione?

Esercizio 2: Si consideri la struttura dati *Binary Search Tree* (BST) ed il problema di determinare la chiave media mantenuta in un BST.

2-a Si proponga un algoritmo efficiente che, dato un BST T contenente n chiavi distinte, determini la chiave media in T (i.e. la chiave k tale che T contiene $\lfloor \frac{n}{2} \rfloor$ chiavi minori di k)

2-b Si proponga una struttura dati che permetta di mantenere un insieme di interi distinti e di gestire, in modo efficiente, le seguenti operazioni:

- inserimento di un intero k nella struttura dati;
- cancellazione di un intero k dalla struttura dati;
- determinazione dell'elemento medio mantenuto nella struttura dati.

Si descriva brevemente e si valuti la complessità di ognuna delle operazioni delineate sulla struttura dati proposta.

§ **Esame di ASD del 09-12-03**

Esercizio 1: Si consideri il problema di definire delle funzioni di hash che mappano gli elementi dell'universo $\mathtt{U} = \{1 \ldots N\}$ nelle celle di una tabella di hash $\mathtt{T}[0 \ldots m]$. Dopo aver elencato le caratteristiche di una buona funzione di hash, per ognuna delle seguenti funzioni h_i, si dica se h_i può essere considerata una ragionevole funzione di hash, giustificando la risposta.

- $h_1(i) = (i \bmod m) + 3$
- $h_2(i) = (2 * i) \bmod m$
- $h_3(i) = (max(N - m, i)) \bmod m$
- $h_4(i) = ((i \bmod m) + 127) \bmod m$

Esercizio 2: Sia $G = \langle V, E \rangle$ un grafo non orientato, connesso e pesato. Si supponga di aver determinato un *minimum spanning tree* di G, T, avente radice v. Si assuma che il peso dell'arco $(u, v) \in E$ venga decrementato di 1 e si consideri il problema di determinare, dato T, un minimum spanning tree del grafo modificato.

- Si definisca un algoritmo efficiente per il problema proposto e se ne valutino correttezza e complessità.

(**Suggerimento**: si pensi ad un taglio che separi il nodo u dal nodo $v \ldots$)

§ **Esame di ASD del 02-09-03**

Esercizio 1: Dato un array ordinato contenente n interi, $A[1 \dots n]$, si consideri il problema di costruire un *Red Black Tree* contenente $A[1], \dots, A[n]$ come chiavi. Si proponga un algoritmo efficiente per il problema proposto e se ne dimostri la correttezza. Si valuti infine la complessitá della procedura proposta.

Esercizio 2: Sia dato un grafo $G = \langle V, E \rangle$ non orientato e si considerino i seguenti problemi:

a) stabilire se G é un *albero*;
(**Suggerimento**: si osservi che, ovviamente, se esiste $r \in V$ tale che G é un albero di radice r, allora per ogni $r' \in V$, G é un albero di radice r')

b) dato $k \in \mathbb{N}$, stabilire se esiste un nodo $r \in V$ tale che G é un albero di altezza k e radice r.

Si definisca un algoritmo efficiente per ciascuno dei problemi proposti. Si dimostri la correttezza delle procedure delineate e se ne valuti la complessitá.

§ **Esame di ASD del 07-07-03**

Esercizio 1: Si consideri il problema di ordinare un array $A[1 \dots n]$ contenente al più k interi distinti, dove k è una costante.

- Si proponga un algoritmo efficiente per il problema proposto e se ne valuti la complessità.
- Si proponga un algoritmo efficiente ed *in place* per il problema proposto e se ne valuti la complessità.
- **Facoltativo** Si proponga un algoritmo *in place* e *stabile* per il problema proposto e se ne valuti la complessità.

Esercizio 2: Si consideri un grafo, $G = \langle V, E \rangle$, orientato ed aciclico. Dato un vertice $v \in V$, sia *d-foglie*(v) il minimo numero di archi in un cammino da v ad una foglia del grafo. Si denoti invece mediante *D-foglie*(v), il massimo numero di archi in un cammino da v ad una foglia del grafo.

- Si proponga un algoritmo efficiente che, dato un grafo orientato aciclico G ed un nodo v, determini *d-foglie*(v). Si valuti la complessità della procedura proposta.

- Si proponga un algoritmo efficiente che, dato un grafo orientato aciclico G ed un nodo v, determini *D-foglie*(v). Si valuti la complessità della procedura proposta.
- **Facoltativo** Si proponga un algoritmo efficiente che associ ad *ogni* vertice di un grafo orientato ed aciclico G, i valori *D-foglie* e *d-foglie*.

§ **Esame di ASD del 02-12-02**

Esercizio 1: Si consideri un grafo non orientato e pesato $G = \langle V, E \rangle$ e si considerino gli algoritmi di Prim e Kruskal visti a lezione. Si supponga G non connesso.

1 Qual è la l'output prodotto da ognuno dei due algoritmi menzionati su G?

1-b Qual è la complessità dei due algoritmi menzionati su G?

Si giustifichino le risposte. **Esercizio 2:** Si supponga di rappresentare mediante una coppia di interi $[l, u]$ con $l \leq u$, l'intervallo di interi compresi tra l ed u (estremi inclusi). Sia dato l'insieme di n intervalli $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$.

2-a Si descriva un algoritmo efficiente per determinare se esistono (almeno) due intervalli disgiunti in S.

2-b Si delinei lo pseudocodice dell'algoritmo descritto al punto precedente e se ne studi la complessità.

2-c (Facoltativo.) Si riconsiderino i precedenti due punti nel caso il problema sia quello di determinare se esistono due intervalli non disgiunti.

§ **Esame di ASD del 26-09-02**

Esercizio 1: Sia A un vettore di n interi e si consideri il problema di determinare se esistono due interi che occorrono in A lo stesso numero di volte.

1-a Si descriva un algoritmo efficiente per il problema proposto nel caso in cui in A occorrono c valori distinti, dove c è una costante intera positiva.

1-b Si descriva un algoritmo efficiente per il problema proposto.

1-c Si determini la complessità degli algoritmi proposti.

Esercizio 2: Siano B_1 e B_2 due B-alberi di grado t, tali che per ogni chiave k_1 in B_1 e k_2 in B_2 si abbia

$$k_1 \leq k_2.$$

2-a Si descriva un algoritmo efficiente che, prendendo in input B_1 e B_2, esegua la *fusione* dei due B-alberi e ritorni in output un B-albero B contenente tutte le chiavi precedentemente in B_1 e B_2 (che vengono eliminati).

2-b Si determini la complessità computazionale dell'algoritmo proposto valutando, in particolare, il numero massimo di accessi alla memoria secondaria nell'ipotesi solo un numero costante di nodi di un B-albero di grado t possa risiedere contemporaneamente in memoria centrale.

§ Esame di ASD del 02-09-02

Esercizio 1: Sia A un vettore di n interi compresi tra 1 e una costante intera positiva c.

1-a Si descriva un algoritmo efficiente che ordini A.

1-b Si descriva un algoritmo efficiente *in place* che ordini A.

1-c Si determini la complessità degli algoritmi proposti in funzione sia di n che di c e si valuti la complessità di tali algoritmi nei casi in cui $c = O(\lg n)$ e $c = O(n)$.

Esercizio 2: Sia $G = (V, E)$ un grafo non orientato e pesato, con funzione di peso $w : E \to \Re^+$.

2-a Si descriva brevemente un algoritmo efficiente (per esempio tra quelli visti a lezione) che determini un *Minimum Spanning Tree* per G e se ne valuti la complessità.

2-b Si provi o si refuti la seguente proprietà: dato un *Minimum Spanning Tree* per G questo contiene almeno un arco tra quelli aventi peso minimo in G.

2-c Si provi o si refuti la seguente proprietà: dato un *Minimum Spanning Tree* per G questo contiene un cammino minimo per ogni coppia di nodi in G.

§ Esame di ASD del 18-07-02

Esercizio 1: Sia A un vettore di $n = 2^k$ interi e si supponga di disporre di un algoritmo `Mediano`(A, i, j) che restituisce l'elemento mediano

della sequenza $A[i..j]$, ovvero l'elemento di posizione $\lfloor (i+j)/2 \rfloor$ nella sequenza ordinata degli elementi di $A[i..j]$.

1-a Scrivere lo pseudo-codice di un algoritmo efficiente `Select`(A, i) che determini l' i-esimo piu' piccolo elemento di A.

2-a Si dimostri la correttezza e si valuti la complessita' dell'algoritmo proposto, supponendo che l' algoritmo `Mediano`(A, i, j) abbia complessita' $O(j-i)$.

Esercizio 2: Sia $G = (V, E)$ un grafo orientato.

2-a Si proponga un algoritmo efficiente che determini se G è un albero, valutandone la complessità e dimostrandone la correttezza.

2-b (Facoltativo) Si proponga un algortimo efficiente che determini se G è trasformabile in un albero mediante l'eliminazione di un opportuno numero di archi. Si valuti la complessità e si dimostri la correttezza dell'algoritmo proposto.

§ Esame di ASD del 18-06-02

Esercizio 1: Sia A un array contentente $n = 2m$ interi:

1-a Si proponga un algoritmo che, preso in input A, produca come output un array in cui gli elementi in posizione pari costituiscano una successione crescente e quelli in posizione dispari costituiscano una successione decrescente.

Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

1-b Si riconsideri il punto precedente e si proponga un algoritmo che svolga la stessa funzione ma che, in piú, operi *in-place*.

Esercizio 2: Sia $G = (V, E)$ un grafo non orientato, connesso e con un numero di archi uguale al numero di nodi.

2-a Si proponga un algoritmo efficiente che determini un Minimum Spanning Tree di G, valutandone la complessità e dimostrandone la correttezza.

2-b Diciamo *Secondo Migliore* Minimum Spanning Tree di G, uno Spanning Tree T' di G tale che $T < T' \leq T''$ per ogni Minimum Spanning Tree T di G e per ogni Spanning Tree T'' di G.

Si proponga un algoritmo efficiente che determini, se esiste, un Secondo Migliore Minimum Spanning Tree di G valutandone la complessità e dimostrandone la correttezza.

§ **Esame di ASD del 24-09-01**

Esercizio 1: Si consideri un alfabeto Σ costituito da quattro caratteri: $\Sigma = \{G, A, T, C\}$ e sia α una stringa finita su Σ (i.e. $\alpha \in \Sigma^\star$). Per ognuno dei quattro caratteri $X \in \Sigma$, si supponga di avere in input l'insieme S_X delle lunghezze dei prefissi di α che terminano con X:

$$S_X \quad = \quad \{a \mid a = |\beta| \wedge \beta \sqsubseteq \alpha \wedge \beta(a) = X\}.$$

1-a Si proponga un algoritmo che determini α a partire da quattro array contenenti ognuno uno dei possibili S_X.

1-b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2:

2-a Si proponga un algoritmo che, dato in ingresso un grafo non orientato e pesato $G = (V, E)$ ed un suo arco $(u, v) \in E$, determini l'esistenza di un MST per G *non* contenente l'arco (u, v).

2-b Si motivi la correttezza e si calcoli la complessità dell'algoritmo proposto.

2-c Si riconsiderino i punti precedenti nel caso in cui si voglia determinare l'esistenza di un MST per il grafo G contenente l'arco (u, v)

§ **Esame di ASD del 10-09-01**

Esercizio 1: Si consideri un Binary Search Tree T contenente $n/2$ elementi.

1.a Si descriva un algoritmo che, dati altri $n/2$ elementi, li inserisca in T e, successivamente, produca la lista ordinata degli elementi di T.

1.b Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2: Definiamo *grafo colorato* un grafo non orientato $G = (V, E)$, in cui agli elementi di V è associato un campo *colore* che può assumere uno tra due possibili valori (diciamo *bianco* e *nero*). Definiamo inoltre un *cammino alternato* un cammino in cui ogni coppia di nodi consecutivi hanno colore diverso, in altre parole un cammino alternato contiene unicamente archi che collegano nodi con colore diverso. Infine definiamo un *cammino quasi alternato* un cammino che contiene al più una coppia di nodi consecutive dello stesso colore, mentre tutte le altre coppie di nodi consecutivi hanno colore diverso.

2.a Si descriva un algoritmo che dati un grafo colorato $G = (V, E)$ ed una coppia di nodi $v, u \in V$, determini un cammino alternato tra v e u con un numero di nodi minimo.

2.b Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

2.c Si risolvano i punti precedenti, derminando però in questo caso un cammino quasi alternato.

§ **Esame di ASD del 13-07-01**

Esercizio 1: Dato un B-albero T di grado 2 e altezza h.

1-a Si proponga un algoritmo che costruisca un RB-albero T' contenente le stesse chiavi di T e con altezza nera $h + 1$.

1-b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

1-c Si dimostri o si refuti l'affermazione: *Ogni RB-albero T' può essere ottenuto quale output del programma proposto a partire da un opportuno B-albero T.*

Esercizio 2:

2-a Si descriva un generico input che rappresenti il *best case* per l'algoritmo `quick_sort`.

2-b Si descriva un generico input che rappresenti il *worst case* per l'algoritmo `quick_sort`.

2-c Si giustifichino le risposte ai due precedenti punti.

§ **Esame di ASD del 28-06-01**

Esercizio 1: Dato un grafo orientato $G = (V, E)$, con la funzione peso $w : E \to \Re^+$ a valori positivi, e due nodi $v, t \in V$.

1-a Si proponga un algoritmo che determini, oltre ad un cammino minimo p_1 da v a t che supponiamo essere $p_1 = \langle v, s_1, \ldots s_n, t\rangle$, un secondo cammino p_2 che sia il cammino di lunghezza minima tra quelli che congiungono v con t e non contengono l'arco (s_n, t).

1-b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

1-c Si proponga un algoritmo che determini oltre un cammino minimo p_1, da v a t, un secondo cammino p_2 che sia il cammino di lunghezza minima tra quelli che congiungono v con t e sono distinti da p_1.

1-d Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2: Data una lista A di 3^k ($k \geq 0$) interi distinti, si consideri il seguente algoritmo $\mathcal{M}(A)$:

step 1 se $k = 0$ $\mathcal{M}(A)$ ritorna in output l'unico elemento di A, altrimenti divide A in 3^{k-1} gruppi di 3 elmementi ciascuno;

step 2 per ognuno dei gruppi determinati al passo precedente determina l'elemento medio;

step 3 $\mathcal{M}(A)$ esegue una chiamata ricorsiva sulla lista costituita dai 3^{k-1} elementi determinati al passo precedente.

2-a Si determini la complessità di $\mathcal{M}(A)$;

2-b Si provi o si refuti la seguente affermazione: *l'elemento ritornato in output da $\mathcal{M}(A)$ è l'elemento medio di A.*

§ **Esame di ASD del 05-06-01**

Esercizio 1: Dato un grafo orientato $G = (V, E)$, si assuma che agli elementi di V sia associato un campo *colore* che può assumere uno tra due possibili valori (diciamo *bianco* e *nero*).

1-a Si proponga un algoritmo che determini l'esistenza di un ciclo contenente un nodo nero.

1-b Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2: Si consideri la seguente estensione della struttura dati "disjoint-sets": accanto alle operazioni standard *Make-Set(x), Union(x,y)*, vengono introdotte le operazioni:

- *Find-Max(x)* Restituisce l'elemento massimo dell'insieme contente l'elemento x (supponiamo che gli elementi di un insieme possano essere ordinati),
- *List (x)* Restituisce la lista di tutti gli elementi contenuti nell'insieme contenente il valore x.

2-a Si propongano delle strutture dati atte a supportare le operazioni indicate.

2-b Si argomenti la correttezza degli algoritmi proposti e si valuti la complessità dell'esecuzione di n operazioni di tipo *Make/Union/Find-Max/List*, m delle quali siano di tipo *Make* e p delle quali siano di tipo *List*.

§ **Compitino di ASD del 01-06-01**

Esercizio 1: Dato un grafo $G = (V, E)$, diremo *grado di connessione* di un nodo v in G il numero di nodi raggiungibili da v in G.

1-a Si proponga un algoritmo che determini il nodo con grado di connessione massimo in un grafo G orientato e aciclico.

1-b Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

1-c Si discutano i punti precedenti nel caso di un grafo non orientato.

Esercizio 2: Siano S' ed S'' due generici insiemi dinamici rappresentati mediante due B-alberi T' e T''. Sia $x \notin S' \cup S''$ tale che se $x' \in S'$ e $x'' \in S''$, allora $key[x'] < key[x] < key[x'']$. Si consideri l'operazione *join* che, dati in input T', T'' e x, restituisce il B-albero T rappresentante l'insieme $S = S' \cup \{x\} \cup S''$.

2-a Si proponga un algoritmo che implementi *join*.

2-b Si argomenti la correttezza e si valuti la complessità dell'algoritmo proposto.

§ **Compitino di ASD del 30-03-01**

Esercizio 1: Dato un vettore A contenente n interi $a_1, \ldots, a_n$ distinti tra di loro, definiamo *spiazzamento* dell'elemento a_i l'intero k tale che $i + k$ sia la posizione di a_i nella versione ordinata del vettore A. Definiamo inoltre un vettore *quasi-ordinato* se tutti i suoi elementi hanno uno spiazzamento che in valore assolutao è minore o uguale ad uno.

1-a Si valuti la complessità degli algoritmi di ordinamente insertion-sort, heap-sort e quick-sort, quando il vettore di ingresso sia quasi-ordinato.

1-b Si proponga un'algoritmo di complessità lineare per l'ordinamento di vettori quasi-ordinati.

1-c Si fornisca una versione modificata dell'algoritmo di quick-sort ottimizzata al caso di dover ordinare vettori quasi-ordinati.

Esercizio 2:

2-a Si scriva lo pseudo-codice di un algoritmo che ricevuto in ingresso un albero binario B, un colore c ($\in \{Red, Black\}$) ed un numero naturale h, determina se B può essere decorato in maniera tale da diventare un RB-tree con radice di colore c ed altezza nera h.

2-b Si valuti la complessità dell'algoritmo proposto.

Nota. E' accettabile come soluzione anche un algoritmo non efficiente.

Suggerimento. Un algoritmo efficiente può essere ottenuto valutando ricorsivamente per ogni sottoalbero A la lista di tutte le possibili coppie (c, h) tali che A possa diventare un RB-tree con radice di colore c e altezza nera h.

§ **Esame di ASD del 14-12-00**

Esercizio 1: Sia x un array $(x_1, \ldots, x_n)$ di interi e sia p una permutazione degli indici $\{1, \ldots, n\}$. Si consideri il seguente algoritmo:

Algoritmo 7.1

```
1: for j := 1 to n do
2:    k := p(j);
3:    while k > j do
4:       k := p(k);
5:    end while
6:    if k = j then
7:       y := x[j]; l := p(k);
8:       while l ≠ j do
9:          x[k] := x[l]; k := l; l := p(k);
10:      end while
11:      x[k] := y;
12:   end if
13: end for
```

1-a Si commenti **Algoritmo 1** e si provi che ricevuto in input x produce in output l'array $(x_{p(1)}, \ldots, x_{p(n)})$.

1-b Si valuti la complessità di **Algoritmo 1**.

1-c (Facoltativo) Si produca un input relativamente al quale la complessità di **Algoritmo 1** sia pessima.

Esercizio 2:

2-a Si proponga un algoritmo che, dato un grafo G, pesato, orientato e aciclico, e un suo nodo v, determini, tra i cammini uscenti da v, quello di lunghezza massima.

2-b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

(**Suggerimento**: modificando il peso degli archi, si riduca il proplema ad un problema di ricerca di cammino minimo)

§ **Esame di ASD del 29-11-00**

Esercizio 1: Un vettore S di n numeri interi si defisce *k-quasi ordinato* se esiste un vettore ordinato S' che differisce da S per al più k elementi, con k costante.

1-a Si proponga un algoritmo lineare per l'ordinamento di vettori k-quasi ordinati.

1-b Si discuta la correttezza dell'algoritmo proposto.

1-c (facoltativo) Si proponga un algoritmo lineare di ordinamento per il caso in cui il vettore S differisca da un vettore ordinato per al più $\sqrt{n}$ elementi.

Esercizio 2: Si consideri il problema *Union-Find* nell'ipotesi gli elementi siano numeri interi e gli insiemi siano, quindi, insiemi disgiunti di numeri interi.

Si considerino le usuali operazioni **Make**, **Union** e **Find** e si consideri inoltre l'operazione

Split che prende in input un elemento x ed un insieme A tale che $x \in A$, e torna in output:

- i due insiemi $A' = \{y \in A \mid y \leq x\}$ e $A'' = \{y \in A \mid y > x\}$, se entrambi A' e A'' sono non vuoti,
- A altrimenti.

2-a Si propongano delle strutture dati atte a supportare le suddette operazioni.

2-b Si discuta la correttezza e si valutino le complessità degli algoritmi proposti.

§ **Esame di ASD del 14-09-00**

Esercizio 1:

1-a si proponga un algoritmo che dati due insiemi di numeri interi A, B, determini la coppia di elementi $a \in A$ e $b \in B$ per cui l'espressione $\mid a - b \mid$ assuma il valore minimo.

1-b Si discuta la correttezza dell'algoritmo proposto e se ne valuti la complessità.

1-c Risolvere i punti precedenti nel caso l'algoritmo abbia come dato di ingresso tre insiemi distintinti A, B, C e debba trovare gli elementi $a \in A, b \in B$ e $c \in C$ per cui l'espressione $| a-b | + | b-c |$ assuma valore minimo.

Esercizio 2: Sia $G = (V, E)$ un grafo orientato e pesato e sia $s \in V$ un nodo sorgente.

2-a Si proponga un algoritmo che risolva il *single source shortest path problem* a partire da s nell'ipotesi in cui i pesi degli archi possano essere solo 1 o 2.

2-c Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

2-c (Facoltativo). Si riconsideri il problema nel caso i pesi possano essere nell'insieme $\{1, \ldots, k\}$, con k costante fissa.

§ **Esame di ASD del 31-08-00**

Esercizio 1: Si consideri un array A contenente n interi (a due a due distinti) quale input per l'algoritmo di ordinamento `Heapsort`.

1-a Qual è la complessità di `Heapsort` quando gli interi in A costituiscono già una sequenza ordinata? Si giustifichi la risposta.

1-b Si supponga che A sia sempre costituito da un segmento finale di lunghezza almeno $\lfloor n/2 \rfloor$ già ordinato, e si ottimizzi `Heapsort` in modo da migliorarne la complessità su A.

Esercizio 2: Sia $G = (V, E)$ un grafo non orientato.

2-a Si proponga un algoritmo che determini se ogni coppia di vertici in G sia collegabile con un cammino contenente al più quattro archi.

2-b Si proponga un algoritmo che determini l'esistenza di uno spanning tree T per il grafo G tale ogni coppia di vertici sia collegabile con un cammino, in T, contenente al più quattro archi. (Suggerimento: in un albero ogni coppia di vertici sono collegabili con un cammino contenente al più quattro archi se e solo se esiste un nodo raggiungibile da ogni altro nodo percorendo al più due archi.)

2-c Si discuta la correttezza e si calcoli la complessità degli algoritmi proposti.

§ Esame di ASD del 13-07-00

Esercizio 1: Sia $G = (V, E)$ un grafo pesato ed orientato, in cui la funzione peso $w : E \to \mathcal{R}$ assume solo valori positivi e sia $v \in V$ un nodo del grafo.

1-a Si proponga un algoritmo che determini se esistono cicli a cui v appartine e nel caso trovi quello di peso minimo.

1-b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

1-c (Facoltativo) Si proponga un algoritmo che determini il ciclo di peso minimo in G.

Esercizio 2: Sia T un albero binario di ricerca privo di nodi aventi la stessa chiave ($key[\cdot]$),

2-a Si proponga un algoritmo efficiente che dati due nodi a e b in T ritorni la lista dei nodi x tali che $key[a] \leq key[x] \leq key[b]$.

2-b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

2-c (Facoltativo) Si modifichi l'algoritmo di cui sopra in modo che ritorni il nodo avente chiave mediana fra le chiavi relative ai nodi della lista ritornata in output.

§ Esame di ASD del 22-06-00

Esercizio 1: Si consideri una struttura dati S che gestisca un insieme di numeri naturali e implementi le tre seguenti operazioni:

- `Insert(n)`, inserisce il numero naturale `n` all'interno della struttura dati S,
- `Extract_min`, estrae l'elemmento minimo da S
- `Extract_min_even`, estrae da S il più piccolo numero pari.

1-a Si proponga un'implementazione per la struttura dati S.

1-b Si analizzi la complessità degli algoritmi proposti e se ne motivi la correttezza.

1-c Si riconsiderino i due punti precedenti nel caso in cui S implementi anche l'operazione `Add(i)` che somma il numero naturale `i` ad ogni elemento contenuto in S. Si cerchi di proporre un implementazione in cui `Add(i)` abbia costo costante.

Esercizio 2: Sia $G = (V, E)$ un grafo orientato,

2-a Si proponga un algoritmo efficiente che determini se G è un albero.

2-b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

2-c (Facoltativo) Si proponga un algoritmo che, nell'ipotesi G non sia un albero, stabilisca qual è il numero minimo di archi che vanno aggiunti/eliminati per trasformare G in un albero. Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

§ **Compitino di ASD del 01-06-00**

Esercizio 1: Sia $G = (V, E)$ un grafo pesato ed orientato, in cui la funzione peso $w : E \to \mathcal{R}$ assume un valore negativo su di un unico arco $(u, v) \in E$.

1-a Si proponga un algoritmo efficiente che determini se nel grafo esiste un ciclo negativo.

1-b Si discuta la correttezza e si calcoli la complessità dell'algoritmo proposto.

1-c Si considerino i punti precedenti nel caso in cui la funzione peso w assuma valore negativo su due archi $(u_1, v_1), (u_2, v_2)$ del grafo G.

1-d (Facoltativo) Si suggerisca l'idea per un algoritmo effeciente che determini l'esistenza di cicli negativi in grafi in cui il numero di archi negativi sia molto limitato, per esempio sia minore di $\ln n$ dove n e' il numbero dei nodi del grafo.

Esercizio 2: Si consideri un *binary search tree* T.

2-a Si proponga un algoritmo che determini se è possibile "colorare" i nodi di T in modo da renderlo un *red-black tree*.

2-b Si proponga un algoritmo che dato in input T lo trasformi in un red-black tree.

2-c Si provi la correttezza e si valutino le complessità degli algoritmi proposti.

§ **Compitino di ASD del 22-2-00**

Esercizio 1: Definiamo *heap ternario* la naturale modifica dello heap (binario); ossia un albero ternario, (ogni nodo ha tre figli, eventalmente vuoti) in cui la chiave associata al padre è maggiore delle chiavi associate ai tre figli.

(a) Si proponga un implementazione dello heap ternario traminte un vettore. In particolare si scrivano le procedure: `Parent(i)`, `Left(i)`, `Heapify(A,i)` e `Build-Heap(A)`.

(b) (Facoltativo) Si discuta il punto (a) nel caso di un *heap esponenziale* ossia di un heap dove ogni nodo di altezza h, ha 2^h figli. Qual è la complessità di un algoritmo di heap-sort che utilizzi un heap esponenziale? Si motivino le risposte.

Esercizio 2: Diciamo *propriamente bitonica* una sequenza S di numeri interi $a_1, \ldots, a_n$ per cui esista $k \in \{1, \ldots, n\}$ tale che $a_1, \ldots, a_k$ sia monotona crescente (decrescente) e $a_k, \ldots, a_n$ sia monotona decrescente (crescente). Diremo *bitonica* una sequenza di interi che possa essere resa propriamente bitonica mediante una permutazione ciclica degli indici.

Esempi: $\{3, 5, 7, 11, 12, 8, 4, -2, -11\}$ sequenza (propriamente) bitonica; $\{7, 11, 12, 8, 4, -2, -11, 3, 5\}$ sequenza bitonica.

Una sequenza bitonica è una sequenza con al più un *picco* (massimo locale interno) e al più una *valle* (minimo locale interno)

(a) Si proponga un algoritmo che, data in input una sequenza bitonica S, produca in output due sequenze bitoniche S_1 ed S_2 di dimensione $n/2$ tali che

$$S_1 \cap S_2 = \emptyset, \quad S_1 \cup S_2 = S, \quad \forall s_1 \in S_1, \forall s_2 \in S_2 (s_1 \leq s_2)$$

(b) Si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

§ Esame di ASD del 14-2-00

Esercizio 1: In una sequenze di n elementi si definisce *elemento medio* l'elemento nella posizione $\lfloor n/2 \rfloor$.

(a) Proporre una struttura dati che permetta di realizziare, in maniera efficiente, le seguenti operazioni:

- `Insert(a)`: Inserisce un nuovo elemento all'interno della struttura
- `Extract_Min`: Estrae l'elemento con chiave minima.
- `Find_Med`: Trova (ma non estrae) l'elmento medio rispetto all'ordine sulle chiavi.

(b) Si discuta la correttezza degli algoritmi proposti e se ne valuti la complessità.

Suggerimento: Una possibile soluzione consiste nel far uso di alberi bilanciati.

Esercizio 2: Sia $G = (V, E)$ un grafo orientato rappresentato mediante liste di adiacenza. Ricordiamo che un *ordinamento topologico* dei nodi di G è un ordinamento $\prec$ degli elementi di V tale che, per ogni $(u, v) \in E$, si abbia che $u \prec v$.

(a) Si proponga un algoritmo che preso in input G, produca un ordinamento topologico dei nodi di G.

(b) Si modifichi il precedente algoritmo in modo tale che, se esistono più ordinamenti topologici di G, due successive chiamate dell'algoritmo producano differenti ordinamenti.

(c) Si discuta la correttezza degli algoritmi proposti e se ne valuti la complessità.

§ **Esame di ASD del 24-1-00**

Esercizio 1: Dato un insieme S contenente n numeri interi, si considerino i seguenti due problemi:

- determinare la coppia di elementi s_1, s_2 in S tale che il valore $|s_1 - s_2|$ risulti minimo;
- determinare la tripla di elementi s_1, s_2, s_3 in S tale che la somma $|s_1 - s_2| + |s_2 - s_3| + |s_1 - s_3|$ risulti minima.

(a) Si scriva lo pseudo-codice di due algortimi che risolvano i i due problemi precedenti;

(b) si determini e si giusitfichi la complessità degli algoritmi proposti.

Esercizio 2: Si consideri un B-albero T con di grado t con n chiavi:

(a) si proponga lo pseudo-codice di un algoritmo per l'inserimento di una chiave k in T che *minimizzi il numero di nuovi nodi generati* (**suggerimento**: si consideri una variante in due passi dell'algoritmo di inserimento visto a lezione);

(b) si discuta la correttezza e si valuti la complessità dell'algoritmo proposto in termini dei parametri t e k.

§ **Esame di ASD del 30-9-99**

Esercizio 1: Sia S una stuttura dati, formata da un insieme di numeri naturali, su cui sia possibile eseguire le seguenti tre operazioni.

- Insert(n), inserisce il numero naturale n all'interno della struttura dati,
- Extract(k), estrae dalla struttura dati il più piccolo multiplo di k in essa contenuto.
- Delete(k), rimuove dalla struttura dati tutti i multipli di k contenuti in essa.

Si supponga inoltre che il valori di k sia limitato da una costante c.

a Si proponga una implementatizione per la struttura dati S.

b Si analizzi la complessita' degli algoritmi proposti e se ne discuta la correttezza.

Esercizio 1: Sia $G = (V, E)$ un grafo non orientato pesato, tale che ogni arco abbia peso negativo.

a Si proponga un algoritmo che risolva il problema della determinazione dei cammini minimi da una data sorgente $s \in V$.

b Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

c Si riconsideri il precedente problema nel caso G sia orientato

§ Esame di ASD del 8-7-99

Esercizio 1: Dato un numero naturale n, si consideri un vettore A contenente n valori compresi tra 1 ed 2^n, e tale che in ogni intervallo $[2^m, 2^{(m+1)}]$ ci siano al più k valori di A, con k costante.

(a) Si scriva lo pseudo codice di un algorimo che ordini il vettore A.

(b) Si discuta la correttezza dell'algoritmo proposto e se ne valuti la complessità, nell'ipotesi ogni operazione sugli interi abbia costo costante e nell'ipotesi i numeri siano rappresentati in base 2 e le operazioni fondamentali sugli interi abbiano un costo proporzionale al loro numero di cifre.

(c) Si confronti la complessità dell'algoritmo proposto con quella di merge-sort.

(d) Si riconsiderino i punti (a) e (b) nel caso in cui ogni intervallo $[2^m, 2^{(m+1)}]$ contenga al più $\ln n$ valori di A.

Si riconsiderino i punti (a) e (b) nel caso in cui ogni intervallo $[2^m, 2^{(m+1)}]$ contenga al più $\sqrt{n}$ valori di A.

Esercizio 2: Sia $G = (V, E)$ un grafo non orientato pesato (pesi positivi e negativi sono ammessi), tale che ogni ciclo non contiene archi negativi.

(a) Si proponga un algoritmo che risolva il problema della determinazione dei cammini minimi da una data sorgente $s \in V$.

(b) Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

§ **Esame di ASD del 8-7-99**

Esercizio 1: Dati un numero naturale m e una matrice $A = (a_{i,j})$ quadrata di dimesione n, avente come valori numeri interi distinti e soddisfacente la sequente proprieta':

$$\forall i, i', j, j' \, (\, i < i' \Rightarrow a_{i,j} < a_{i',j'}),$$

(a) si scriva lo pseudocodice di un algoritmo che, considerando l'ordine sugli interi, determina l'm-esimo elemento della matrice A;

(b) si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2: Dato un RB-albero, si consideri il problema di mantenere un campo *black-height[x]* che, per ogni nodo x appartenente all'albero, contenga l'altezza nera di x.

(a) Si proponga lo pseudocodice commentato e dettagliato di opportune versioni delle procedure RB-TREE-INSERT e RB-TREE-DELETE atte a garantire il mantenimento del campo *black-height*;

(b) si discuta la correttezza e si valutino le complessità delle procedure proposte;

(c) si discuta brevemente il problema di mantenere un campo *height* che contenga l'altezza.

§ **Esame di ASD del 15-6-99**

Esercizio 1: Sia A una matrice $n \times n$ di numeri interi.

(a) Si dia lo psedocodice di un algoritmo che determina l'elemento di A che appartiene al maggior numero di righe, nell'ipotesi che ogni riga contenga elementi tutti distinti.

(b) Si risolva il punto precedente per una generica matrice.

(c) Si discuta la correttezza e si valuti la complessità degli algoritmi proposti.

Esercizio 1: Dato un B-albero di grado t si consideri il problema di mantenere un campo *height[x]* che, per ogni nodo x appartenente all'albero, contenga l'altezza di x.

(a) Si proponga lo pseudocodice commentato e dettagliato di opportune versioni delle procedure B-TREE-INSERT e B-TREE-DELETE atte a garantire il mantenimento del campo *height*;

(b) si discuta la correttezza e si valutino le complessità delle procedure proposte.

§ **Compitino di ASD del 13-5-99**

Esercizio 1: Definiamo LRB-tree (*Loose* Red-Black tree) come un albero binario di ricerca in cui:

i) tutti i nodi sono etichettati come neri o rossi,

ii) ogni foglia (*nil*) è nera,

iii) se un nodo x è rosso ed il padre di x è rosso allora entrambi i figli di x sono neri,

iv) ogni cammino semplice dalla radice ad una foglia contiene lo stesso numero di nodi neri.

(a) Si mostri un LRB-tree che non sia un RB-tree.

(b) Qual e' l'altezza massima di un LRB-tree con n nodi?

(c) Se si applica l'algoritmo RB-Insert ad un LRB-Tree l'albero risultate e' un LRB-tree?

Si motivino le risposte.

Esercizio 2: Si consideri un grafo pesato e orientato $G = (V, E)$ con funzione di peso $w : \Re \rightarrow E$

(a) Si proponga un'algoritmo che determini tutti i nodi x che si trovano su un ciclo negativo e, per ognuno di tali x, produca in output un ciclo negativo che contenga x.

(b) Si discuta la correttezza e si valuti la complessità dell'algoritmo proposto.

§ **Compitino di ASD del 23-2-99**

Esercizio 1: Sia dato un vettore $A[1..n]$ di interi positivi.

(a) Si scriva lo pseudo-codice di un algoritmo efficiente che trovi un indice k tale che:

$$| \sum_{i=1}^{k-1} A[i] - \sum_{j=k+1}^{n} A[j] \ | \leq \ A[k].$$

(b) Si giusitfichi e si determini la complessità dell'algoritmo proposto.

(c) Si riconsiderino i quesiti (a) e (b) nell'ipotesi di dover cercare un elemento A_k tale che:

$$| \sum_{A[i]<A[k]} A[i] - \sum_{A[k]<A[j]} A[j] \ | \leq \ A[k].$$

(d) (Facoltativo) Una algoritmo soddisfacente per il problema al punto (c) richiede tempo $\Theta(n \log n)$, nel casa peggiore. Si discuta l'esistenza di soluzioni con un limite di complessità media più basso.

Esercizio 2: Si considerino gli algoritmi *build_heap*, *extract_min* e *insert* definiti per operare su una *min_heap* H di profondità h e contenente n nodi.

(a) Si proponga una variante della struttura dati *min_heap* atta a supportare le suddette procedure e anche una procedura *extract_max* di estrazione del massimo.

(b) Si discuta la correttezza e si valutino le complessità degli algoritmi proposti in funzione dei parametri h ed n.

§ **Esame di ASD del 10-2-99**

Esercizio 1: Sia dato un vettore A di interi, diversi da 0, di dimensione n.

(a) Si scriva lo pseudo-codice (commentandolo dettagliatamente) di un algoritmo efficiente ed *in–place* che modifichi il vettore A in modo tale che al termine il (sotto)insieme dei numeri positivi sia ordinato in modo crescente e quello dei numeri negativi in modo decrescente.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

(c) Si riconsiderino i quesiti (a) e (b) nell'ipotesi che venga richiesto anche che al termine dell'algoritmo nel vettore A ogni elemento positivo sia seguito da un elemento negativo, fino all'esaurimento dei numeri positivi o dei numeri negativi, e quindi compaiano tutti i numeri, positivi o negativi, rimanenti.

Esercizio 2: Sia $G = (V, E)$ un grafo pesato non orientato in cui non ci sono due archi di peso uguale e sia T un *minimum spanning tree* di G.

(a) Dato un arco $e = (i, j) \in E$ si dimostri che e appartiene a T se e solo se ogni cammino di lunghezza maggiore o uguale a 2 da i a j contiene un arco di peso maggiore del peso di e. Inoltre si provi o si refuti l'unicitá di T.

(b) Si descriva un algoritmo che dati G e T nelle ipotesi di cui sopra e dato un arco pesato $\bar{e} \notin E$ di peso diverso dal peso di ogni arco in E, ritorni un minimum spanning tree T' per $G' = (V, E \cup \{\bar{e}\})$?

(c) Si provi la correttezza e si determini la complessità dell'algoritmo di cui al punto precedente.

§ **Esame di ASD del 26-1-99**

Esercizio 1: Si considerino n numeri interi $a_1, ..., a_n$ la cui somma sia n.

1. Si disegni un algoritmo efficiente per ordinare $a_1, ..., a_n$.
2. Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 2: Sia H una heap contenente gli elementi $\{a_1, ..., a_n\}$ e dato A sottoinsieme di $\{a_1, ..., a_n\}$, sia $H(A)$ la più piccola sotto-heap di H contenente gli elmenti di A.

1. Si proponga un algoritmo efficiente che, dati $A_1, ..., A_k$ sottoinsiemi di $\{a_1, ..., a_n\}$, li ordini in modo tale che $A_i < A_j \Rightarrow H(A_i) \subseteq H(A_j)$
2. Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

§ **Esame di ASD del 28-9-98**

Esercizio 1: Sia dato un generatore di sequenze (localmente) ordinate di numeri naturali, di valore compreso tra 0 e c (con c costante prefissata), il cui comportamento può essere descritto come una successione infinita di passi che si articolano nelle seguenti due fasi:

(fase 1) generazione di sequenze ordinate di lunghezza esponenzialmente crescente 2^i, con $i = 0, 1, \ldots, k$;

(fase 2) generazione di sequenze ordinate di lunghezza esponenzialmente decrescente 2^i, con $i = k, k-1, \ldots, 0$.

Si chiede di:

(1) proporre lo pseudo-codice, adeguatamente commentato, di un algoritmo che riceva in input le $2 \cdot (k+1)$ sequenze ordinate generate in un singolo passo e restituisca in output un'unica sequenza ordinata;

(2) provare la correttezza e determinare la complessità dell'algoritmo proposto.

Esercizio 2: Sia $G = (V, E)$ un grafo non-orientato e connesso. Diciamo che un vertice $v \in V$ è un *cut-vertice* se la rimozione da G di v (e di tutti gli archi che lo contengono) sconnette G.

(1) Proporre lo pseudo-codice, adeguatamente commentato, di un algoritmo che ricevuto in input G restituisca in output tutti i cut-vertici di G;

(2) provare la correttezza e determinare la complessità dell'algoritmo proposto.

Suggerimento: usare la visita in profondità e la classificazione degli archi di G che si ottiene eseguendo tale visita.

§ **Esame di ASD del 9-9-98**

Esercizio 1: Si considerino le due seguenti varianti dell'algoritmo *heap-sort*. A partire da una *min-heap*,

- dopo aver restituito in output il minimo corrente x, (anziché rimuoverlo sostituendolo con la foglia più a destra dell'ultimo livello) determina l'elemento minimo fra le foglie dell'ultimo livello, scambialo con la foglia più a destra e sostituisci ad x tale elemento;
- scambia il nodo x che si trova in posizione di radice (minimo corrente) con il minimo fra i suoi figli e ripeti l'operazionione finché x non raggiunge la posizione di una foglia; a quel punto restituisci x in output e rimuovilo.

(a) Si fornisca lo pseudo-codice commentato delle due varianti di heap-sort sopra descritte;

(b) si provi la correttezza e si determini la complessità degli algoritmi proposti.

Esercizio 2: Sia $G = (V, E)$ un grafo diretto, $w : E \to \Re^+$ una funzione peso, $s \in V$ ed $(u, v) \in E$ un arco.

(a) Si proponga un algoritmo per determinare se esiste un cammino minimo da s a v che contiene l'arco (u, v);

(b) si proponga un algoritmo per determinare se esiste un cammino minimo da s a v che non contiene l'arco (u, v);

(c) si provi la correttezza e determini la complessità degli algoritmi proposti.

§ **Esame di ASD del 15-7-98**

Esercizio 1: Dato quale input un vettore V contenente $n = 3^k$ numeri interi distinti, si consideri il seguente algoritmo $\mathcal{A}$:

passo 1: se $k = 0$ l'unico elemento di V viene restituito come output, altrimenti il vettore V viene suddiviso in k gruppi contenenti 3 elementi ciascuno;

passo 2: per ognuno dei k gruppi viene determinato l'elemento avente valore mediano e viene costituito il vettore V' contenete i 3^{k-1} elementi determinati;

passo 3: $\mathcal{A}$ richiama se stesso con V' come input.

(a) Si determini la complessità di $\mathcal{A}$ in funzione di n;

(b) si descriva una implementazione *in-place* di $\mathcal{A}$;

(c) si provi o si refuti il seguente enunciato: l'elemento prodotto in output da $\mathcal{A}$ cui è stato dato in input V è l'elemento mediano di V.

Esercizio 2: Dato $\Sigma = \{0, 1\}$, siano $a = a_0, a_1, \ldots, a_p$ e $b = b_0, b_1, \ldots, b_q$, due stringhe su Σ. Diciamo che la stringa a è lessicograficamente minore della stringa b se:

1. esiste un intero j, con $0 \leq j \leq min(p, q)$, tale che $a_i = b_i$, per $i = 0, 1, \ldots, j-1$, e $a_j < b_j$, oppure
2. $p < q$ e $a_i = b_i$ per $i = 0, 1, \ldots, p$.

La struttura dati *radix tree* è un albero binario ordinato i cui archi sinistri sono etichettati con 0, i cui archi destri sono etichettati con 1 e i cui nodi possono essere bianchi o neri. Inizialmente tutti i nodi sono bianchi; una stringa s viene rappresentata in un radix tree colorando di nero il nodo x tale che la concatenazione delle etichette nel cammino dalla radice ad x produca s.

Sia S un insieme di stringhe distinte su Σ, la somma delle cui lunghezze sia pari a n.

(a) Proporre un algoritmo efficiente che ordini lessicograficamente S utilizzando la struttura dati radix tree ;

(b) provare la correttezza e determinare la complessità dell'algoritmo proposto.

§ **Esame di ASD del 30-6-98**

Esercizio 1: Dato un albero k-ario τ (generalizzazione della nozione di albero binario in cui ogni nodo interno ha k, anzichè 2, figli) perfettaente bilanciato e un nodo interno $x \in \tau$, siano $\downarrow (x,0), ..., \downarrow (x,k-1)$ rispettivamente il primo, il secondo, ... il k-esimo figlio di x.

Sull'insieme dei nodi di τ sia definita una relazione di ordinamento totale $<$ nel seguente modo: $<$ è la chiusura transitiva della relazione $\prec$, tale che:

i per ogni nodo interno x, $\downarrow (x,0) \prec x$ e $x \downarrow (x,j)$, con $j = 1, ..., k-1$;

ii per ogni nodo interno x, $\downarrow (x,j) \prec \downarrow (x,j+1)$, per $j = 1, ..., k-2$;

iii dati $x, x' \in \tau$, se $x \prec x'$ e x' non appartiene al sottoalbero radicato in x, allora $\downarrow (x,k-1) \prec x'$;

iv dati $x, x' \in \tau$, se $x \prec x'$ e x' non appartiene al sottoalbero radicato in x', allora $x \prec \downarrow (x',0)$.

Si dimostri la verità o la falsità delle seguenti proposizioni (in caso di falsità si produca un controesempio):

1a dati $x, x' \in \tau$, se profondità(x) = profondità(x') e $x < x'$, allora per ogni nodo y appartenente al sottoalbero radicato in x, e per ogni nodo y' appartenente al sottoalbero radicato in x', $y < y'$;

1b per ogni nodo interno x è vero che, per ogni nodo appartenente al sottoalbero radicato in $\downarrow (x,0)$, $y < x$ e che, per ogni nodo y' appartenente al sottoalbero radicato in $\downarrow (x,1)$, $x < y'$;

1c (generalizzazione di 1a) dati $x, x' \in \tau$, se $x < x'$ allora per ogni nodo y appartenente al sottoalbero radicato in x e per ogni nodo y' apartenente al sottoalbero radicato in x', $y < y'$.

Esercizio 2: Dato un grafo $G = (V, E)$ diretto aciclico:

2a si proponga un algoritmo efficiente che associ ad ogni nodo v la lunghezza del cammino più lungo da v ad una foglia (cioè un nodo privo di archi uscenti);

2b si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

§ **Compitino di ASD del 10-6-98**

Esercizio 1: Si consideri un grafo orientato e pesato $G = (V, E)$ e si supponga che G contenga una *radice* (cioè un nodo dal quale ogni altro nodo in V risulti raggiungibile).

(a) Si proponga un algoritmo efficiente per determinare se G contiene un ciclo negativo;

(b) si modifichi l'algoritmo proposto al passo precedente in modo da produrre in output i nodi di un ciclo negativo (quando ne esista uno);

(c) si provi la correttezza e si valuti la complessità degli algoritmi proposti.

§ **Compitino di ASD del 27-2-98**

Esercizio 1: Si considerino un insieme S di numeri naturali e le seguenti operazioni:

1. $extract\_min(S)$ che estrae (cancellandolo) l'elemento minimo da S;
2. $insert(x, S)$ che inserisce l'elemento x in S;
3. $print\_log\_sequence(S)$ che stampa una sequenza ordinata di $\log(|S|)$ elementi di S tale che il primo elemento della sequenza sia il minimo in S e l'ultimo sia il massimo in S.

(a) Si proponga una struttura dati atta a memorizzare S ed a supportare efficientemente le operazioni sopra elencate.

(b) Si provi la correttezza e si determini la complessità degli algoritmi proposti per supportare le operazioni considerate.

(c) Si discuta il problema di supportare anche l'operazione di estrazione del massimo.

§ **Esame di ASD del 12-2-98**

Esercizio 1: Sia A un vettore contenente n interi $a_1, \ldots, a_n$ a due a due distinti. Definiamo *spiazzamento* dell'elemento a_i l'intero k tale che $i + k$ sia la posizione di a_i nella versione ordinata di A.

(a) Si fornisca lo pseudo-codice di commentato dettagliatamente di un algoritmo che, preso in input il vettore A, produca come output il vettore $B = [b_1, \ldots, b_n]$ degli spiazzamenti di $a_1, \ldots, a_n$.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 2: Dato un grafo non orientato $G = (V, E)$, diciamo che G è *bipartito* se esistono V_1 e V_2 tali che

- $V_1 \cap V_2 = \emptyset$;
- $V_1 \cup V_2 = V$;
- per ogni $e = (u, v) \in E$, $u \in V_1 \Leftrightarrow v \notin V_2$.

(a) Si fornisca lo pseudo-codice di commentato dettagliatamente di un algoritmo che determini se un grafo G è bipartito.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

§ **Esame di ASD del 28-1-98**

Esercizio 1: Si assuma che la coppia di interi $[a_i, b_i]$, con $a_i \leq b_i$, rappresenti l'insieme degli interi compresi tra a_i e b_i (estremi inclusi). Si scriva un algoritmo efficiente che, ricevuto in ingresso un insieme di n coppie $[a_i, b_i]$, con $1 \leq i \leq n$, tali che $\forall i, j$ $(1 \leq i, j \leq n)$ $b_i - a_i = b_j - a_j$, restituisca un insieme di m coppie $[c_i, d_i]$, con $1 \leq i \leq m$, comprendenti tutti e soli gli interi che appartengono ad uno ed uno solo degli insiemi $[a_i, b_i]$.

(a) Si fornisca lo pseudo-codice dell'algoritmo, commentandolo dettagliatamente.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

(b) Si discutano gli effetti della rimozione della condizione $\forall i, j$ $(1 \leq i, j \leq n)$ $b_i - a_i = b_j - a_j$ sulle soluzioni proposte per i quesiti (a) e (b).

Esercizio 2: Dato un grafo non orientato (non pesato) $G = (V, E)$ ed un insieme di nodi $F \subseteq V$:

(a) si proponga lo pseudocodice di un algoritmo efficiente che determini, se esiste, un albero di supporto per G in cui ogni nodo di F risulti essere una foglia;

(b) si provi la correttezza e si valuti la complessità dell'algoritmo proposto.

§ Esame di ASD del 1-10-97

Esercizio 1: Si scriva un algoritmo efficiente che, ricevuto in ingresso un insieme di n intervalli $[a_i, b_i]$, con a_i, b_i numeri interi e $1 \leq i \leq n$, stabilisca se la loro unione è un intervallo (nel qual caso, restituisca tale intervallo) o meno.

(a) Si fornisca lo pseudo-codice commentato dell'algoritmo.

(b) Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 2: Si dia una risposta motivata ad ognuna delle seguenti domande:

1. Sia T un *minimum spanning tree* di un grafo indiretto G con pesi positivi. Se aggiungiamo a G esclusivamente un vertice v ed archi (di peso positivo) uscenti da v, il peso di un *minimum spanning tree* per il grafo ottenuto è sicuramente superiore al peso di T?
2. Tutti gli *spanning trees* (sia minimi che non) di un grafo G hanno lo stesso numero di archi?
3. Se aggiungiamo archi ad un grafo diretto aciclico, il grafo ottenuto è ancora un grafo diretto aciclico?
4. Se cancelliamo archi da un grafo diretto aciclico, il grafo ottenuto è ancora un grafo diretto aciclico?
5. Il grafo indiretto ottenuto eliminando la direzione degli archi da un grafo diretto aciclico, è aciclico?
6. Supponiamo che durante la visita in profondità (DFS) di un grafo diretto G vengano assegnati ad ogni vertice v due numeri: $pre(v)$ e $post(v)$. Tali valori corrispondano, rispettivamente, alle posizioni di v nelle liste in pre-ordine ed in post-ordine dei nodi nel DFS-albero/i ottenuto dalla visita. Sono vere le seguenti affermazioni?
   - se $pre(v) < pre(w)$ e $post(v) > post(w)$ allora c'è un cammino da v a w in G;
   - se $pre(v) < pre(w)$ e $post(v) < post(w)$ allora non c'è un cammino da v a w G;

§ Esame di ASD del 18-9-97

Esercizio 1: Si consideri un cammino dalla radice ad una delle foglie in un albero binario di ricerca. Siano A l'insieme delle chiavi associate ai nodi alla sinistra del cammino, B l'insieme delle chiavi associate

ai nodi appartenenti al cammino, C l'insieme delle chiavi associate ai nodi alla destra del cammino. Sia inoltre x la chiave associata alla radice.

Si dimostri la verità o falsità delle seguenti proposizioni (in caso di falsità, si produca un controesempio):

1-a $\forall y, z((y \in B \wedge z \in C) \rightarrow y \leq z)$;

1-b $\forall y(y \in A \rightarrow y \leq x) \vee \forall y(y \in C \rightarrow y \geq x)$;

1-c $\forall y \in A \forall z \in C(y \leq z)$.

Esercizio 2: Sia $G = (V, E)$ un grafo diretto e aciclico, e siano s e t due vertici distinti in V. Si consideri il problema di determinare un insieme *massimale* di cammini disgiunti (privi di nodi in comune) da s a t.

2-a Si producano G, s, e t tali che esiste un insieme massimale ma non di cardinalità massima, di cammini disgiunti da s a t in G.

2-b Si proponga lo pseudo codice commentato di un algoritmo che risolva il problema proposto.

2-c Si valuti la complessità dell'algoritmo di cui al punto precedente.

§ **Esame di ASD del 30-7-97**

Esercizio 1: Si scriva un algoritmo efficiente che, ricevuto in ingresso un multi-insieme (insieme con ripetizioni) $E \subseteq N \times N$ (con N insieme dei numeri naturali) di n coppie che coinvolgono $m \leq n$ elementi distinti di N tale che $\forall (x, y) \in E \; \exists z \in N \; ((y, z) \in E)$, restituisca l'insieme $E' = \{(f(x), f(y)) : (x, y) \in E\}$, dove f è una funzione che mappa gli m elementi distinti di N che compaiono nelle coppie di E nei primi m numeri naturali $1, \ldots, m$ in modo che $\forall (x, y), (z, w) \in E \; (x < z \rightarrow f(x) < f(z))$.

1-a Si fornisca lo pseudo-codice commentato dell'algoritmo.

1-b Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

1-c Si riconsiderino i quesiti (a) e (b) nell'ipotesi aggiuntiva che valga la condizione $\exists k \in N \; \forall (x, y) \in E \; (x \leq k)$.

1-d E' possibile fornire un limite inferiore (lower bound) per la complessità dell'algoritmo di cui al punto (a)?

Esercizio 1: Sia $G = (V, E)$ un grafo diretto e aciclico.

2-a Si proponga un algoritmo che determinini un ordinamento $<$ dei nodi di V tale che per $u, v \in V$, se esiste un cammino da u a v in G allora $u < v$.

2-b Si dimostri la correttezza e si valuti la complessitá dell'algoritmo proposto.

§ **Esame di ASD del 23-6-97**

Esercizio 1: Si descriva un algoritmo efficiente che, ricevuto in input un vettore A di n interi positivi, restituisca in output l'insieme degli elementi che occupano una delle posizioni comprese tra la k-esima posizione e la $(k+m)$-esima posizione (estremi inclusi) nella permutazione che ordina A, con $1 \leq k \leq k+m \leq n$.

1-a Si fornisca lo pseudo-codice commentato dell'algoritmo.

1-b Si provi la correttezza e si determini la complessità dell'algoritmo proposto.

Esercizio 2: Sia $G = (V, E)$ un grafo indiretto e siano s, t ed u tre vertici distinti in V.

2-a Si proponga un algoritmo che determini se ogni cammino da s a t passa per u.

2-b Si proponga un algoritmo che determini se esiste un cammino da s a t passante per u.

2-c Si provi la correttezza e si valuti la complessità degli algoritmi proposti.

§ **Esame di ASD del 3-6-97**

Esercizio 1: Sia A un array di interi positivi o negativi, di dimensione n e tale che

$$A[1] < A[2] < \ldots < A[n].$$

1-a Scrivere lo pseudocodice commentato di un algoritmo che, preso in input A, produca come output un indice i tale che $A[i] = i$, se tale i esiste, o NIL altrimenti.

1-b Dimostrare la correttezza e determinare la complessità dell'algoritmo proposto.

Esercizio 2: Si consideri la tecnica di implementazione di una min-heap mediante un array discussa a lezione. Si consideri l'operazione, che chiameremo $insert\_cut(x)$, che consiste nell'inserire l'elemento x nella heap e, contemporaneamente, cancellare dalla heap tutti gli elementi maggiori di x.

2-a Si dimostri che l'array (rappresentante la heap) ottenuto a partire dall'array vuoto dopo m operazioni di tipo $insert\_cut(x)$ è ordinato.

2-b Si proponga una implementazione atta a supportare esclusivamente operazioni di tipo $insert\_cut(x)$.

2-c Si discuta una modifica della tecnica usata per l'implementazione che permetta di trattare anche operazioni di tipo *delete_min*, in modo che n operazioni *delete_min* ed m operazioni di tipo $insert\_cut(x)$ abbiano costo globale $\theta(m+n)$.

§ **Esame di ASD del 27-1-97**

Esercizio 1: Sia A un vettore contenente $2n$ numeri interi, di cui n positivi ed n negativi.

1-a Si scriva lo pseudo-codice commentato di un algoritmo efficiente che dati x, y con $x < y$ determina se esiste una coppia (i, j) tale che $A[i] < 0, A[j] > 0$ e $x < A[i] + A[j] < y$.

1-b Si dimostri la correttezza e si valuti la complessità dell'algoritmo proposto.

Esercizio 2: Sia H una min-heap di altezza h contenente $2^{h+1} - 1$ elementi. Sia $H(p)$ l'insieme dei nodi di H aventi profondità p:

$$H(p) \quad = \quad \{a \in H \mid \text{ depth}(a) = p\}.$$

Si assuma che H sia rappresentata mediante un vettore e che, per ogni p, il sotto-vettore contenente la lista dei nodi in $H(p)$ sia ordinata.

2-a Si dimostri che il vettore contenente H non è necessariamente ordinato.

2-b Si proponga lo pseudo-codice commentato di un algoritmo efficiente che produca la lista ordinata dei nodi in H.

[**Suggerimento**: Si consideri un algoritmo ricorsivo.]

2-c Si dimostri la correttezza e si valuti la complessità dell'algoritmo proposto.

§ **Esame di ASD del 14-10-96**

Esercizio 1: Sia A un vettore di $n = 2^k$ interi. Una coppia (i, j) si dice *un'inversione di* A se $i < j$ e $A[i] > A[j]$.

1-a Si scriva lo pseudocodice commentato di un algoritmo efficiente che determina il numero di inversioni del vettore A.

1-b Si valuti la complessità e dimostrare la correttezza dell'algoritmo proposto.

Esercizio 1: Sia $G = \langle V, E \rangle$ un grafo indiretto, connesso e aciclico con n nodi.

2-a Si dimostri che esiste sempre un cammino che percorre tutti gli archi del grafo G in entrambe le direzioni.

2-b Si proponga lo pseudo-codice commentato di un algoritmo che produca un cammino con le caratteristiche di cui al punto precedente.

2-b Si valuti la complessità dell'algoritmo proposto.

§ **Esame di ASD del 30-9-96**

Esercizio 1: Sia A un vettore di $n = 2^k$ interi. Una coppia (i, j) si dice *un'inversione di* A se $i < j$ e $A[i] > A[j]$.

1-a Si scriva lo pseudocodice commentato di un algoritmo efficiente che determina il numero di inversioni del vettore A.

1-b Si valuti la complessità e dimostrare la correttezza dell'algoritmo proposto.

Esercizio 2: Sia T_{rb} un red-black tree di altezza h.

2-a Si disegni e si valuti la complessità di un algoritmo che trasformi T_{rb} in un B-albero T_B di grado 2 e di altezza $\Theta(h)$.

2-b Si dimostri che T_B è un B-albero.

§ **Esame di ASD del 2-9-96**

Esercizio 1: Sia k una costante intera maggiore di 1 e sia A un vettore di n interi tali che $0 \leq A[i] < kn$.

1-a Scrivere lo pseudo-codice commentato di un algoritmo che ordina il vettore A in tempo lineare e che usa non più di $3n + 3$ locazioni ausiliarie di memoria.

1-b Dimostrare la correttezza dell'algoritmo proposto.

Esercizio 2: Si dica albero binario un albero in cui ogni nodo ha $0, 1$ o 2 figli, si dica inoltre albero binario *perfetto* un albero in cui ogni nodo ha 0 o 2 figli.

2-a Si dimostri che il numero di foglie f di un albero binario perfetto T è uguale ad $i + 1$, con i numero di nodi interni in T.

2-b Si proponga lo pseudo-codice commentato di un algoritmo che, dato un albero binario di ricerca, lo trasformi in un albero binario perfetto di ricerca, se questo è possibile, oppure ritorni NIL.

2-c Si provi la correttezza e si valuti la complessità dell'algoritmo proposto al punto precedente.

§ **Esame di ASD del 22-7-96**

Esercizio 1: Sia A un vettore di n interi e sia m il numero di posizioni i tali che $A[i] > A[i+1]$.

1-a Scrivere lo pseudo-codice **commentato**, dimostrare la correttezza e determinare la complessità di un algoritmo efficiente che ordina il vettore A nel caso in cui m è una costante.

1-b Scrivere lo pseudo-codice **commentato**, dimostrare la correttezza e determinare la complessità di un algoritmo efficiente che ordina il vettore A nel caso in cui $m = O(\log n)$.

1-c Gli algoritmi proposti sono stabili? Motivare la risposta e, in caso negativo, indicare come rendere stabili i suddetti algoritmi.

Esercizio 2: Si supponga di dover eseguire una sequenza di operazioni di tipo MAKE-SET, UNION e FIND. Si assuma che tutte le operazioni di tipo MAKE-SET precedano quelle di tipo UNION e che queste, a loro volta, precedano quelle di tipo FIND.

2-a Si propongano strutture dati ed algoritmi che permettano di implementare efficientemente le operazioni considerate.

2-b Si analizzino le complessità degli algoritmi proposti ed il costo asintotico di una sequenza generica di operazioni che soddisfi l'assunzione relativa all'ordine delle operazioni.

§ **Esame di ASD del 24-6-96**

Esercizio 1: Si consideri un albero binario di ricerca T e un nodo x in T con chiave associata $key[x]$.

1-a Scrivere lo pseudo-codice e dimostrare la correttezza di una procedura MODIFY-KEY(T, x, key) che modifica x assegnando a $key[x]$ valore key, e ritorna T se questo è ancora un albero binario di ricerca, altrimenti ritorna NIL. Si valuti la complessità dell'algoritmo proposto.

1-b Si riconsideri il precedente punto nel caso l'albero T sia un red-black tree.

Esercizio 2:

2-a Scrivere un algoritmo di complessità $O(n \cdot \log m)$ per fondere m vettori ordinati in un vettore A, dove n è il numero totale di elementi contenuti negli m vettori.

2-b Si consideri un vettore di $n = 2^{2^k}$ elementi. Utilizzando la soluzione al punto (2-a), scrivere lo pseudo-codice e l'equazione della complessità asintotica della seguente variante di MERGESORT:

- Suddividere il vettore in $\sqrt{n}$ sottovettori;
- ordinare (ricorsivamente) i sottovettori;
- fondere i sottovettori ordinati.

2-c Facoltativo: Valutare la complessità dell' algoritmo precedente. È possibile risolvere il punto (2-a) con complessità $O(n)$?

List of Algorithms

1.1 Insertion Sort(A) . . . 6
1.2 SelectionSort(A) . . . 9
1.3 Bubble Sort(A) . . . 9
1.4 MergeSort(A, p, r) . . . 10
1.5 Merge(U, x, y, z) procedura di Merge Sort . . . 11
1.6 Parent(i) procedura di Heap sort . . . 15
1.7 Left(i) procedura di Heap Sort . . . 15
1.8 Right(i) procedura di Heap Sort . . . 15
1.9 Heapify(A, i) procedura di Heap Sort . . . 17
1.10 BuildHeap(A) procedura di Heap Sort . . . 19
1.11 HeapSort(A) . . . 22
1.12 QuickSort((A, p, r) . . . 22
1.13 Partition(A, p, r) procedura di Quick Sort . . . 23
2.1 Countign Sort(A, B, k) . . . 30
2.2 RadixSort(A, k) . . . 31
2.3 Bucket Sort(A) . . . 33
2.4 RandomizedSelect(A, p, r, i) . . . 35
3.1 . . . 42
4.1 StackEmpty(S) procedura di Stack . . . 52
4.2 Push(S, x) procedura di Stack . . . 52
4.3 Pop(S) procedura di Stack . . . 53
4.4 Dequeue(Q) procedura di Queue . . . 53
4.5 Enqueue(Q, x) procedura di Queue . . . 54
4.6 ListSearch(L, k) procedura di Linked List . . . 55
4.7 ListInsert(L, x) procedura di Linked List . . . 55
4.8 ListDelete(L, x) procedura di Linked List . . . 55
4.9 Hash Insert(T, k) procedura di Hash Table . . . 59
4.10 Hash Search(T, k) procedura di Hash Table . . . 59
4.11 InOrder(x) . . . 62
4.12 TreeSearch(x, k) . . . 63
4.13 ItTreeSearch(x, k) . . . 63
4.14 TreeMin(x) . . . 64
4.15 TreeMax(x) . . . 64
4.16 TreeSuccessor(x) . . . 65

4.17 LeftRotate(T, x) . . . 66
4.18 TreeInsert(T, z) . . . 67
4.19 TreeDelete(T, z) . . . 68
4.20 RBTreeInsert(T, x) . . . 71
4.21 RBTreeDelete(T, z) . . . 75
4.22 RBDeleteFixUp(T, x) . . . 76
4.23 BTreeSearch(x, k) . . . 79
4.24 BTreeCreate(T) . . . 80
4.25 BTreeSplit(x, i, y) . . . 81
4.26 BTreeInsert(T, k) . . . 82
4.27 BTreeInsertNonFull(x, k) . . . 83
5.1 ConnectedComponents(G) . . . 94
5.2 SameComponent(u, v) . . . 94
5.3 Make(x) . . . 97
5.4 Find(x) . . . 98
5.5 Union(x, y) . . . 98
5.6 Link(x, y) procedura di Union . . . 98
6.1 BFS(G, s) . . . 103
6.2 DFS(G) . . . 107
6.3 DFSVisit(u) . . . 108
6.4 ComponenteTrasposta(G) procedura di SCC . . . 112
6.5 SCC(G) . . . 113
6.6 MST(G) . . . 114
6.7 MST-Kruskal(G, w) . . . 116
6.8 MST-Prim(G, w, r) . . . 117
6.9 Relax(u, v, w) . . . 119
6.10 InitializeSingleSource(G, s) . . . 120
6.11 Dijkstra(G, w, s) . . . 122
6.12 Bellman-Ford(G, w, s) . . . 124
6.13 ExtendShortestPath(D, W) . . . 128
6.14 MatrixMultiply(A, B) . . . 128
6.15 SlowAPSP(W) . . . 129
6.16 FasterAPSP(W) . . . 130
6.17 Floyd-Warshall(W) . . . 130
6.18 Johnson(G) . . . 133
7.1 . . . 154

Elenco delle figure

1.1 Albero delle chiamate ricorsive. 12
1.2 Esempio di albero binario completo (a sinistra) e quasi completo (a destra) 14
1.3 Implementazione di heap tramite array ed esempi di funzioni. 15
1.4 Rappresentazione delle chiamate di Heapify: a) un caso generico; b) caso peggiore. 19
1.5 Esempio di albero di ricorsione di QuickSort 25
1.6 Altro esempio di albero di ricorsione. 26
1.7 Esempio di Decision Tree 27

2.1 Esempio di algoritmo di ordinamento stabile 30
2.2 Esempio di funzionamento di Radix Sort 31
2.3 Schema di realizzazione di Bucket Sort 32
2.4 Esempio di funzionamento di Bucket Sort 33

4.1 Esempio di uso di head e tail in una coda 53
4.2 Esempi di: a) linked list; b) doubly linked list con sentinella . 54
4.3 Universo di chiavi e tabella hash 56
4.4 Schema di rotazione destra e sinistra. 65
4.5 Situazioni e azioni da intraprendere dopo un inserimento in un red black tree. 72
4.6 Situazioni di RBDeleteFixUp. 74
4.7 Schema di BTreeSplit. 80
4.8 Casi di BTreeDelete. 85

5.1 a) linked list e b)linked list modificata per soddisfare le esigenze di MAKE() e FIND(). 94
5.2 Esempio di rappresentazione di insiemi disgiunti tramite alberi di arietà variabile. 96

6.1 (a) grafo orientato; (b) rappresentazione mediante lista di adiacenza; (c) rappresentazione mediante matrice di adiacenza.101
6.2 Esempio di grafo e di come agisce DFS. 108

Indice analitico

adjacency list, 101
adjacency matrix, 101
alberi bilanciati, 67
albero
    binario, 14
    completo, 14
    quasi completo, 14
albero binario
    completo, 10
arco leggero, 115
attraversa un taglio, 115

BellMan-Ford, 124
BFS, 102
binary search tree, 61
BST
    BST Property, 61
    In Order, 62
    itTreeSearch, 63
    tree delete, 67
    tree insert, 67
    Tree Search(x, k), 63
    Tree Successor(x), 65
BST
    Left Rotate(T, x), 65
BTree
    BTreeCreate, 80
    BTreeInsert, 82
    BTreeInsertNonFull, 82
    BTreeSearch, 79
    BTreeSplit, 81
Bubble Sort, 10
bucket, 32
Bucket Sort, 32

cammino, 118
cammino minimo, 118
chaining, 56
code di priorità, 14
codice
    in place, 6
    non tight, 6
    tight, 5
complessità, 5
    if-then-else, 8
Componente Trasposta, 112
componenti connesse, 93
Connected Components, 93
correttezza, 5
Counting Sort, 30

decision tree, 27
depth-first forest, 108
depth-first tree, 108
DFS, 107
DFSVisit, 107
Dijkstra, 122
divide and conquer, 10
dynamic programming, 126

Extend Shortest Path(D, W), 127

FasterAPSP(W), 129
fattore di carico, 57
FIFO, 52
Find, 97
Floyd-Warshall(W), 130

grafo denso, 101
grafo sparso, 101
greedy stategy, 114

Hash Table

Hash Insert(T,k), 59
Hash Search(T,k), 59
heap, 14
max heap, 16
min heap, 16
Heap Sort, 21
BuildHeap, 19
Heapify, 17
Left, 14
Parent, 14
Right, 14
Heapify, 16
complessità, 18
correttezza, 17

in place, 5
in-order tree walk, 61
InitializeSingleSource, 119
Insertion Sort, 6
insiemi dinamici, 51

Johnson(G), 132

LIFO, 52
Link, 97
Linked List
ListDelete(L, x), 54
ListInsert(L, x), 54
ListSearch(L, k), 54
List
circolare, 54
doubly linked, 54
singly linked, 54

Make, 97
matrice trasposta, 102
Matrix Multiply(A, B), 128
Merge Sort, 10
Merge, 10
MST, 114
MSTKruskal, 116
MSTPrim, 117

nodi esterni, 69

overflow, 52

Pop(S)
Stack, 52
post-order tree walk, 61
pre-order tree walk, 61
predecessor sub-graph, 118
predecessor subgraph, 106
primary clustering, 60
pseudo codice, 5
Push(S, x)
Stack, 52

Queue
Dequeue(Q), 53
Enqueue, 53
Quick Sort, 22
Partition, 22

RadixSort, 31
Randomized Select, 35
rappresentante, 93
RBT
RBDeleteFixUp, 75
RBTreeDelete, 75
RBT
RBTreeInsert(T, x), 70
Red black tree, 67
red-black property, 69
relax, 119
reweighting, 131
rispetta un taglio, 115

SameComponent, 93
SCC, 112
secondary clustering, 60
Selection Sort, 8
shortest path tree, 118
simple uniform hasing, 57
SlowAPSP(W), 129
spanning tree, 114
split, 80
StatckEmpty
Stack, 52
struttura dati, 51

taglio, 115

tail recursive, 63
telescoping, 13
termine dominante, 8
TreeMax(x), 63
TreeMin(x), 63

underflow, 52
uniform hasing, 59
Union, 97