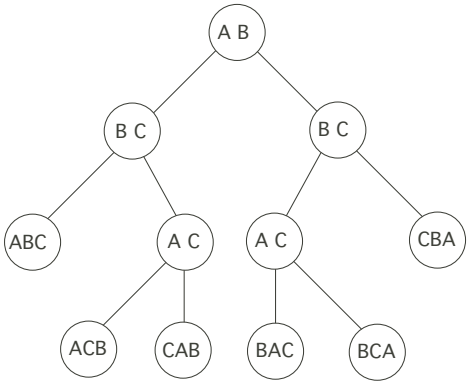
Algoritmi di ordinamento per confronto di chiavi

1 – Una limitazione inferiore sul numero di confronti

Si immagini di rappresentare tutte le permutazioni con un albero binario in cui ogni nodo interno ha due figli (quindi il numero dei nodi interni è il numero delle foglie meno uno), le foglie sono in corrispondenza uno a uno con le permutazioni e i nodi interni corrispondono ad una relazione d'ordine fra due chiavi i e j per cui in tutte le permutazioni discendenti del figlio sinistro i precede j e in tutte quelle discendenti del figlio destro i segue j. Inoltre la permutazione corrispondente ad una foglia è l'unica coerente con l'ordine stabilito dai nodi interni che si trovano sul cammino dalla radice alla foglia. Ad esempio alle sei permutazioni di tre chiavi può essere associato il seguente albero binario (l'albero non è unico e non necessariamente si fanno gli stessi confronti su nodi del medesimo livello).



In ogni caso, comunque venga costruito l'albero, il numero di nodi interni, cioè di confronti, è $n! - 1$. Un algoritmo che per determinare un ordinamento si basi su confronti, esegue, a seconda dell'istanza, i confronti corrispondenti al cammino dalla radice alla permutazione dell'ordinamento. Il caso peggiore, in termini di numero di confronti, è quindi dato dal cammino più lungo, ovvero dalla profondità d dell'albero. Si è visto che vale la relazione $d \geq \log_2(N+1) - 1$ con N numero dei nodi dell'albero. Siccome $N = 2\,n! - 1$ si ha

$$d \geq \log_2(2\,n!) - 1 = \log_2 n!$$

Siccome $n! > (n/2)^{n/2}$, che si ottiene da

$$(n!)^2 = \prod_{k=1}^{n} k \prod_{k=1}^{n} (n-k+1) = \prod_{k=1}^{n} k\,(n-k+1) > \prod_{k=1}^{n} \max\{k\,, n-k+1\} > \prod_{k=1}^{n} \frac{n}{2} = \left(\frac{n}{2}\right)^n$$

si ha

$$d > \log_2(\frac{n}{2})^{n/2} = \frac{n}{2}\,\log_2\frac{n}{2} \in \Omega(n\,\log n)$$

La dimostrazione che $n/2\log(n/2) \in \Omega(n\log n)$ si può fare nel seguente modo: si deve dimostrare che esistono una costante $K > 0$ ad un valore $\bar{n}$ tali che

$$\frac{n}{2}\,\log_2\frac{n}{2} \geq K\,n\,\log_2 n, \qquad n \geq \bar{n}$$

ovvero che

$$\frac{n}{2}\,\log_2\,n - \frac{n}{2} \geq K\,n\,\log_2 n, \qquad n \geq \bar{n}$$

Prendiamo ad esempio il valore $K = 1/4$. Allora

$$\frac{n}{2}\,\log_2\,n - \frac{n}{2} \geq \frac{n}{4}\,\log_2 n, \qquad n \geq \bar{n}$$

cioè

$$\frac{n}{4}\,\log_2\,n \geq \frac{n}{2}, \qquad n \geq \bar{n} \quad \Longrightarrow \quad \log_2\,n \geq 2, \qquad n \geq \bar{n}$$

da cui $\bar{n} = 4$.

Quindi qualsiasi algoritmo di ordinamento che si basi sul confronto di chiavi non potrà mai avere complessità inferiore a $O(n\,\log n)$ e gli algoritmi con tale complessità possono essere considerati ottimali. Questo non significa che non possono esistere algoritmi di ordinamento con complessità inferiori, ad esempio lineari. Vi sono infatti algoritmi che non si basano sul confronto di chiavi e che operano con complessità $O(n)$ se alcune ipotesi sulle chiavi sono soddisfatte. Li esamineremo più avanti.

2 – Insertion sort

Le chiavi sono disposte in un vettore K. Ad una generica iterazione j dell'algoritmo le chiavi $k_1, \ldots, k_{j-1}$ sono ordinate, la chiave k_j viene confrontata con la chiave k_{j-1}; se $k_j \geq k_{j-1}$ allora le chiavi $k_1, \ldots, k_j$ sono ordinate e si può procedere con l'iterazione successiva, altrimenti si confronta k_j con k_{j-i}, $i := 2, 3, \ldots$ fino a trovare una chiave p tale che $k_j \geq k_p$, altrimenti, se tale chiave non esiste si pone $p = 0$; poi si spostano le chiavi da $p+2$ a $j-1$ di una posizione (cioè $k_i := k_{i-1}$ con $i := j-1, j-2, \ldots, p+2$) e si sposta la chiave k_j in posizione $p+1$ (cioè $k_{p+1} := k_j$). Dopo l'ultima iterazione tutte le chiavi sono ordinate.

```
procedure insertionsort(K)
    { n := |K|;
    for j := 2 to n do
        { stop:=False;
        i := j − 1;
        while ¬ stop ∧ i ≥ 1 do
            { stop:=K[j] > K[i];
```

```
            i := i − 1;
            } ;
        if stop then p := i + 2 else p = 1;
        temp:=K[j];
        for i := j − 1 downto p do K[i + 1] := K[i];
        K[p]=temp
        }

    } .
```

Nel caso peggiore sono richiesti $j-1$ confronti all'iterazione j e globalmente si hanno quindi

$$1+2+3+\ldots+n-1=\frac{n\,(n-1)}{2}\in O(n^2)\qquad \text{confronti}$$

Per analizzare il caso medio si può ragionare che al momento di confrontare la chiave j con quelle precedenti, saranno richiesti mediamente "circa" $j/2$ confronti e quindi globalmente il numero di confronti sarà la somma di $(1+2+\cdots+n-1)/2\in O(n^2)$.

3 – Bubblesort

```
procedure bubblesort(K)
    { b := n
    while b > 1 do
        {
        a := 1;
        for i := 2 to b do
        if K[i] > K[i − 1];
            then{
                K[0] := K[i − 1]; K[i − 1] := K[i]; K[i] = K[0]; a := i
                } ;
        b := a − 1;
        }
    } .
```

Si eseguono diverse scansioni del vettore delle chiavi scambiando di posto fra loro due chiavi adiacenti se sono nell'ordine sbagliato. Conviene mantenere in memoria l'indice dell'ultimo scambio effettuato. Alla fine della scansione tutti gli elementi dall'indice dell'ultimo scambio fino alla fine sono necessariamente ordinati: l'ultimo elemento scambiato è più grande di tutti quelli che lo precedono mentre è più piccolo del successivo (altrimenti sarebbe stato scambiato) che a sua volta è più piccolo del successivo e così via. La scansione viene ripetuta fino all'indice dell'ultimo scambio, finché il vettore risulta ordinato.

Il caso peggiore si ha quando i dati sono ordinati in ordine inverso. È immediato valutare $O(n^2)$ confronti di chiavi. Il caso migliore si ha con le chiavi già ordinate che richiede solo $n-1$ confronti. L'analisi del caso medio conduce ad una complessità quadratica.

4 – Quicksort

L'idea di quicksort è di dividere i dati in due insiemi tali che tutti gli elementi di un insieme sono minori o uguali a tutti gli elementi dell'altro insieme. Questa divisione viene realizzata scegliendo un elemento a caso, detto *pivot* e ponendo in un insieme gli elementi minori o uguali al pivot e nell'altro gli elementi maggiori del pivot. Poi si ordinano i due insiemi ricorsivamente usando la stessa tecnica. L'insieme finale ordinato si ottiene semplicemente elencando prima il primo insieme, poi il pivot e poi il secondo insieme. Nell'algoritmo qui indicato il pivot è scelto come il primo elemento del vettore da ordinare.

```
procedure quicksort(K)
    { n := |K|;
    if n = 0 ∨ n = 1
    then return(K)
    else
        { (K1, pivot, K2)=partition(K)
        return(quicksort(K1),pivot,quicksort(K2));
        }
    } .

procedure partition(K)
    { n := |K|;
    pivot:=K[1];
    p1 := 1;p2 := 1;
    for i := 2 to n do
        if K[i] < pivot
        then K1[p1] := K[i]; p1 := p1 + 1
        else K2[p2] := K[i]; p2 := p2 + 1;
    return(K1,pivot,K2);
    } .
```

Se vi sono dati uguali è senz'altro più rapido, anziché ripartire in due insiemi ed un elemento, ripartire invece in tre insiemi, il primo di elementi tutti minori del pivot, il secondo di elementi uguali al pivot e il terzo di elementi maggiori del pivot. Ad esempio nel caso estremo di tutti dati uguali, basta un'iterazione con questa variazione, altrimenti si ricade nel caso peggiore, analizzato qui sotto.

Analisi del caso peggiore:

Le operazioni da considerare nella valutazione della complessità sono essenzialmente i confronti da effettuare nel momento della partizione. La restituzione del vettore ordinato da parte della procedura quicksort (nelle sue varie chiamate) è linearmente correlata con gli ordinamenti. Il numero di confronti nella prima applicazione di quicksort è $n-1$. Dopo avere ripartito l'insieme in due insiemi di n_1 e n_2 elementi, i confronti da effettuare sono n_1-1+n_2-1 se $n_1 \geq 1$ e $n_2 \geq 1$, e n_1+n_2-1 se $n_1 = 0$ oppure $n_2 = 0$ (entrambi nulli non possono essere perché per $n = 1$ l'insieme non viene più suddiviso). Siccome $n_1 + n_2 = n - 1$, il numero di confronti di questa seconda fase è $n-3$ oppure $n-2$. Proseguendo gli insiemi verranno a loro volta suddivisi, n_1 in due insiemi n_{11} e n_{12} e n_2 in due insiemi n_{21} e n_{22} e il numero di confronti, se i valori n_{ij} sono non nulli, è

$$n_{11} - 1 + n_{12} - 1 + n_{21} - 1 + n_{22} - 1 = n_{11} + n_{12} + n_{21} + n_{22} - 4 =$$

$$n_1 - 1 + n_2 - 1 - 4 = n_1 + n_2 - 2 - 4 = n - 1 - 2 - 4 = n - 7$$

Quindi il numero di confronti, nel caso migliore, cala ad ogni iterazione di un fattore esponenziale ovvero (se ad esempio $n > 32$)

$$n - 1 + n - 3 + n - 7 + n - 15 + n - 31 + \dots$$

fino ad arrivare a 0. Nel caso migliore, nel quale ad ogni iterazione vengono prodotti due insiemi che differiscono, per cardinalità, al più per un elemento, nell'iterazione k-ma si eseguono $n - (2^k - 1)$ confronti. Quindi il numero m di iterazioni è il massimo valore di k per cui $n - (2^k - 1) \geq 0$, ovvero $\log_2(n+1) \geq k$, da cui $m = \lfloor \log_2(n+1) \rfloor$. Quindi il numero globale di confronti è dato da

$$\sum_{k=1}^{\lfloor \log_2(n+1) \rfloor} n + 1 - 2^k = (n+1)\ \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2 \in \Theta(n \log n - n) = \Theta(n \log n)$$

Se invece i dati sono già ordinati (e il pivot viene scelto come primo elemento) o in generale, se la scelta casuale del pivot è l'elemento minimo (o massimo), i due sottoinsiemi sono sbilanciati con $n_1 = 0$ e $n_2 = n - 1$. Proseguendo, il primo insieme è vuoto e quindi non viene suddiviso, e il secondo insieme viene suddiviso in due insiemi ancora sbilanciati $n_{21} = 0$ e $n_{22} = n_2 - 1 = n - 2$. In questo caso il numero confronti è

$$n - 1 + n_2 - 1 + n_{22} - 1 = n - 1 + n - 2 + n - 3 + \dots = \frac{n\,(n-1)}{2} \in \Theta(n^2)$$

Il caso peggiore porta quindi a considerare una complessità quadratica. Siccome il caso migliore ha una complessità $\Theta(n \log n)$ è interessante valutare il caso medio.

Analisi del caso medio:

Inizialmente bisogna valutare la probabilità con cui un insieme di n dati viene suddiviso in due insiemi di n_1 e n_2 dati. Possiamo, per semplicità, assumere che la probabilità che due dati siano uguali sia nulla. Comunque va sottolineato che la presenza di dati uguali diminuisce il numero di confronti (purché venga adottata la variazione indicata precedentemente), per cui l'analisi di caso medio che viene ora fatta è pessimistica rispetto al caso con dati non necessariamente diversi.

Sotto l'ipotesi di dati diversi $\Pr\{n_1 = k\} = 1/n$, per cui, se indichiamo con $A(n)$ il valore medio di confronti per n dati, si ricava l'espressione ricorsiva

$$A(n) = n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} A(k) + A(n-1-k) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} A(k)$$

$$n\,A(n) = n\,(n-1) + 2 \sum_{k=0}^{n-1} A(k) \quad \Longrightarrow \quad (n-1)\,A(n-1) = (n-1)\,(n-2) + 2 \sum_{k=0}^{n-2} A(k)$$

sottraendo

$$n\,A(n) - (n-1)\,A(n-1) = 2\,(n-1) + 2\,A(n-1) \quad \Longrightarrow \quad n\,A(n) = (n+1)\,A(n-1) + 2\,(n-1)$$

dividendo per $n\,(n+1)$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2\,(n-1)}{n\,(n+1)}$$

$B(n) := A(n)/(n+1)$

$$B(n) = B(n-1) + \frac{2\,(n-1)}{n\,(n+1)}$$

siccome $B(0) = 0$

$$B(n) = \sum_{k=1}^{n} \frac{2\,(k-1)}{k\,(k+1)} = -\sum_{k=1}^{n} \frac{2}{k} + \sum_{k=1}^{n} \frac{4}{k+1} = -\sum_{k=1}^{n} \frac{2}{k} + \sum_{k=2}^{n+1} \frac{4}{k} =$$

$$-2 + \sum_{k=2}^{n} \frac{2}{k} + \frac{4}{n+1} = -4 + \sum_{k=1}^{n} \frac{2}{k} + \frac{4}{n+1} = \sum_{k=1}^{n} \frac{2}{k} - \frac{4\,n}{n+1} = 2\,H(n) - \frac{4\,n}{n+1}$$

dove $H_n = \sum_{k=1}^{n} 1/k$ sono i numeri armonici, visti precedentemente. Dalla relazione asintotica

$$H_n \approx \ln n + \frac{1}{2\,n} + \gamma$$

si ha

$$B(n) = \sum_{k=1}^{n} \frac{2}{k} - \frac{4\,n}{n+1} \approx 2\,\ln n + \frac{1}{n} - \frac{4\,n}{n+1}$$

$$A(n) \approx (n+1)\,(2\,\ln n + \frac{1}{n} - \frac{4\,n}{n+1}) = 2\,n\,\ln n + 2\ln n + \frac{n+1}{n} - 4\,n \in \Theta(n\,\log n)$$

5 – BST sort

Per ordinare i dati si può pensare di costruire un BST (secondo un ordine qualsiasi delle chiavi) e poi operare una visita in-order del BST. Si è già visto che la visita in-order ha complessità $O(n)$. Tuttavia la costruzione del BST ha complessità $O(n\,d)$ con d profondità del BST. Se si riesce a mantenere bilanciato il BST durante la sua costruzione si ha a disposizione un algoritmo di ordinamento di complessità $O(n\,\log n)$. Non presentiamo qui le operazioni di bilanciamento per un BST e neppure presentiamo uno speciale tipo di albero binario, i cosiddetti red-black trees, che sono per definizione bilanciati e per i quali le operazioni di inserzione ed eliminazione sono simili a quelle di bilanciamento di un BST.

In base all'analisi fatta precedentemente la complessità media in un BST sort è $O(n\,\log n)$.

6 – Mergesort

Nell'algoritmo quicksort il fattore chiave per ottenere una buona complessità computazionale era la possibilità di suddividere in modo bilanciato un insieme. Per sfruttare quest'idea in modo sistematico si può pensare di suddividere l'insieme in due insiemi arbitrari ma di uguale cardinalità, di ordinarli e poi fonderli assieme in un unico insieme ordinato. Si può sfruttare il fatto che, dati due insiemi già ordinati, si ottiene velocemente un unico insieme ordinato. L'ordinamento dei due sottoinsiemi viene eseguito secondo la stessa tecnica in maniera ricorsiva.

procedure mergesort(K)

```
    { n := |K|; m = ⌊n/2⌋ ;
    for i := 1 to m do A[i] := K[i]
    for i := m + 1 to n do B[i − m] := K[i]
    C = mergesort(A)
    D = mergesort(B)
    return(merge(C, D))
    } .

procedure merge(A, B)
    { n_a := |A|; n_b := |B|;
    p_a := 1; p_b := 1; p_c := 1;
    while p_a ≤ n_a ∧ p_b ≤ n_b do
        { if A[p_a] < B[p_b]
        then C[p_c] := A[p_a]; p_a := p_a + 1
        else C[p_c] := B[p_b]; p_b := p_b + 1;
        p_c := p_c + 1;
        } ;
    if p_a > n_a
    then for i := 0 to n_b − p_b do C[p_c + i] := B[p_b + i]
    else for i := 0 to n_a − p_a do C[p_c + i] := A[p_a + i];
return(C);
    } .
```

Analisi del caso peggiore:

La procedura merge richiede un numero lineare di operazioni. Pertanto, se indichiamo con $T(n)$ il numero di operazioni richiesto per ordinare un insieme di n elementi, possiamo stabilire la seguente equazione ricorsiva.

$$T(n) = 2\,T(\frac{n}{2}) + n \tag{1}$$

Per risolverla si operi la sostituzione $n = 2^m$ per cui (1) diventa

$$T(2^m) = 2\,T(2^{m-1}) + 2^m \tag{2}$$

e si definisca $f(m) := T(2^m)$ per cui (2) diventa

$$f(m) = 2\,f(m-1) + 2^m = 2\,(f(m-1) + 2^{m-1}) \tag{3}$$

Per risolvere la ricorsione (3) calcoliamo i primi termini in funzione di $f(0)$:

$$f(1) = 2\,f(0) + 2; \qquad f(2) = 2\,f(1) + 2^2 = 2\,(2\,f(0) + 2) + 2^2 = 2^2\,f(0) + 2\,2^2$$

$$f(3) = 2\,f(2) + 2^3 = 2\,(2^2\,f(0) + 2\,2^2) + 2^3 = 2^3\,f(0) + 3\,2^3$$

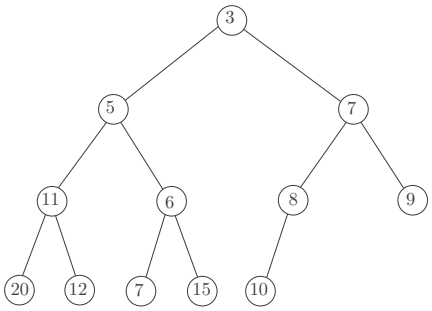
Si induce l'espressione generale (che può essere facilmente dimostrata per induzione)

$$f(m) = 2^m\,f(0) + m\,2^m$$

Siccome $f(0) = T(1) = 0$ si ha $f(m) = m\,2^m$ ovvero $T(n) = n\,\log_2 n$.

7 – Heapsort

Uno heap è una struttura dati che permette di determinare in tempo costante il minimo di un insieme ed in tempo logaritmico la rimozione del minimo o in generale l'aggiornamento di una qualsiasi chiave. Uno heap è un albero binario bilanciato (con le foglie del livello massimo tutte a sinistra) con le regola che la chiave di un nodo è non peggiore delle chiavi dei figli (se presenti).



L'insieme dei dati sia ad esempio:

$$K = \{6,\ 10,\ 8,\ 3,\ 7,\ 11,\ 20,\ 12,\ 9,\ 7,\ 5,\ 15\}$$

Una possibile inserzione di K in uno heap è rappresentata in figura. Uno heap viene normalmente implementato come un vettore elencando le chiavi livello per livello (in questo caso abbiamo bisogno anche di puntatori che, nota la posizione nello heap della chiave, forniscano la posizione della chiave in K e viceversa). Alternativamente lo heap può essere costituito direttamente da puntatori a K. Questa seconda opzione fa risparmiare un vettore ed è preferibile. Useremo questa seconda opzione anche se nelle figure, unicamente per semplicità didattica, le chiavi appaiono residenti nello heap. In entrambi i casi è indispensabile avere dei puntatori che da K indirizzano allo heap. Ad esempio, per lo heap in figura la sua implementazione può essere il vettore (con chiavi nello heap)

$$H = \{3,\ 5,\ 7,\ 11,\ 6,\ 8,\ 9,\ 20,\ 12,\ 7,\ 15,\ 10\}$$

oppure il vettore di puntatori

$$H = \{4,\ 11,\ 5,\ 6,\ 1,\ 3,\ 9,\ 7,\ 8,\ 10,\ 12,\ 2\}$$

In entrambi i casi è richiesto anche il vettore inverso di puntatori

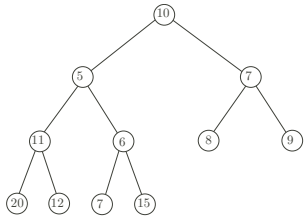
$$H^{-} = \{5,\ 12,\ 6,\ 1,\ 3,\ 4,\ 8,\ 9,\ 7,\ 10,\ 2,\ 11\}$$

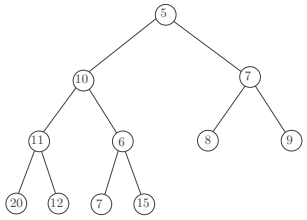
Si noti che i figli dell'elemento $K[H[i]]$ sono gli elementi $K[H[2\,i]]$ e $K[H[2\,i+1]]$ (purché ovviamente $2\,i \leq |H|$ e $2\,i+1 \leq |H|$). Viceversa l'elemento padre di un elemento $K[H[i]]$ è l'elemento $K[H[\lfloor i/2 \rfloor]]$. Quindi i figli

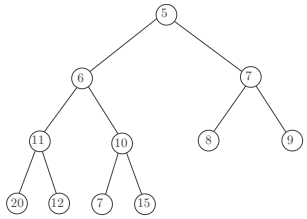
di $K[j]$ sono $K[H[2\,H^-[j]]]$ e $K[H[2\,H^-[j]+1]]$ (purché $2\,H^-[j] \leq |H|$ e $2\,H^-[j]+1 \leq |H|$). Esaminiamo in dettaglio le operazioni eseguibili su uno heap.

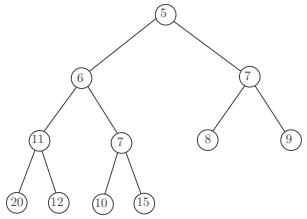
Lettura e prelevamento del minimo

Per leggere il minimo elemento di K basta ovviamente leggere $H[1]$ con complessità $O(1)$. Se, oltre a leggere il minimo si vuole eliminarlo da K, bisogna aggiornare lo heap. Siccome la struttura deve avere un elemento in meno, l'ultima posizione dello heap non può essere occupata. Quindi si può pensare di spostare la chiave presente nell'ultima posizione e spostarla nella prima, rimasta vacante dopo la rimozione dell'elemento minimo (prima delle figure qui sotto). Questa azione però può violare la proprietà dello heap che ogni nodo deve essere non peggiore dei suoi figli. Per ripristinare la proprietà si confronti la chiave in posizione 1 con i suoi due figli. Se il migliore dei due figli è migliore del padre si deve operare uno scambio fra questi (seconda figura). L'operazione va ripetuta ricorsivamente finché la proprietà dello heap è ripristinata (altre due figure). Il numero di confronti necessario in questa fase è limitato dalla profondità dello heap e quindi la complessità globale dell'operazione è $O(\log n)$.









La discesa verso il basso delle chiavi può essere realizzata dalla seguente procedura che data una posizione nello heap in cui la chiave è eventualmente mal posta perché peggiore di uno dei figli, sposta verso il basso le chiavi fino a ripristinare la condizione. Nelle successive procedure K è il vettore degli elementi da confrontare, H è lo heap di puntatori a K, n è il numero di elementi (attivi) nello heap, H^- è il vettore di puntatori da K in H. K, H e H^- sono vettori di dimensione sufficientemente ampia da memorizzare un elevato di dati. Il sottoinsieme di K di cui si valuta il minimo (sottoinsieme attivo) corrisponde ai puntatori presenti in H fino alla posizione n. Gli elementi di H successivi alla posizione n sono senza significato. Gli elementi di H^- non corrispondenti ad elementi attivi sono senza significato.

function checkdown(k)
{ **if** $2\,k \leq n$

then firstson:=$K[H[2\,k]]$
else firstson:=∞;
if $2\,k+1 \leq n$
then secondson:=$K[H[2\,k+1]]$
else secondson:=∞;
if firston $<$ secondson
then bestson:= firstson; $h := 2, k$
else bestson:= secondson; $h := 2, k+1$
if bestson $< K[H[k]]$
then return(true,h)
else return(false,nil)
} .

procedure switch(i, j)
{ $H^-[H[i]] := j$;
$H^-[H[j]] := i$;
temp:=$H[i]$;
$H[i] := H[j]$;
$H[j]$:=temp;
} .

procedure movedown(k)
{ (misplaced,h):=checkdown[k]
while $\neg$ misplaced **do**
{ switch[k, h];
$k := h$
(misplaced,h):=checkdown[k]
}
} .

procedure getdelmin(K)
{ **return**($K[H[1]]$);
switch(1,n);
$n := n - 1$;
movedown(1)
} .

Ecco descritta la dinamica di H e H^- durante l'iterazione. Scambio di 1 e $n = 12$ (in corsivo gli elementi senza significato)

$$H = \{2,\ 11,\ 5,\ 6,\ 1,\ 3,\ 9,\ 7,\ 8,\ 10,\ 12,\ \mathit{4}\}$$

$$H^- = \{5,\ 1,\ 6,\ \mathit{12},\ 3,\ 4,\ 8,\ 9,\ 7,\ 10,\ 2,\ 11\}$$

Scambio di 1 e 2

$$H = \{11,\ 2,\ 5,\ 6,\ 1,\ 3,\ 9,\ 7,\ 8,\ 10,\ 12,\ \mathit{4}\}$$

$$H^- = \{5,\ 2,\ 6,\ \mathit{12},\ 3,\ 4,\ 8,\ 9,\ 7,\ 10,\ 1,\ 11\}$$

Scambio di 2 e 5

$$H = \{11,\ 1,\ 5,\ 6,\ 2,\ 3,\ 9,\ 7,\ 8,\ 10,\ 12,\ \mathit{4}\}$$

$$H^- = \{2,\ 5,\ 6,\ \mathit{12},\ 3,\ 4,\ 8,\ 9,\ 7,\ 10,\ 1,\ 11\}$$

Scambio di 5 e 10

$$H = \{11,\ 1,\ 5,\ 6,\ 10,\ 3,\ 9,\ 7,\ 8,\ 2,\ 12,\ \mathit{4}\}$$

$$H^{-} = \{2,\ 10,\ 6,\ \mathit{12},\ 3,\ 4,\ 8,\ 9,\ 7,\ 5,\ 1,\ 11\}$$

$$K = \{6,\ 10,\ 8,\ 3,\ 7,\ 11,\ 20,\ 12,\ 9,\ 7,\ 5,\ 15\}$$

Ordinamento dei dati

Basta eseguire n volte l'operazione di prelevare il minimo. Siccome si tratta di eseguire n volte un'operazione di complessità $O(\log n)$, la complessità dell'ordinamento è $O(n \log n)$. Si può notare che all'avanzare dell'algoritmo il numero di dati attivi nello heap diminuisce per cui la complessità potrebbe diminuire. Un'analisi più accurata che faremo più avanti, porta comunque al valore $O(n \log n)$.

procedure heapsort(K)
{ $m := n$;
for $k := 1$ **to** m **do** $S[k]$:=getdelmin(K) ;
return(S)
} .

Inserimento di un dato

Per inserire un dato in uno heap, l osi posiziona dapprima nell'ultima posizione possibile e poi si verifica se la proprietà dello heap è verificata. In questo caso la proprietà va verificata rispetto al nodo padre e ci possono essere scambi di nodi verso l'alto fino alla radice. La complessità è ovviamente $O(\log n)$. La procedura di inserire un dato potrebbe essere usata anche per costruire uno heap inserendo i dati uno alla volta a partire dal primo. In questo modo uno heap con n dati potrebbe essere costruito con complessità $O(n \log n)$. Vedremo tuttavia più avanti che per costruire uno heap è più conveniente usare una tecnica diversa che presenterà complessità $O(n)$. Nella procedura qui sotto indicata si assume che K contiene n dati tutti inseriti nello heap e che il nuovo dato viene aggiunto al vettore K.

function checkup(k)
{ $h := \lfloor k/2 \rfloor$;
if $K[H[k]] < K[H[h]]$
then return(true,h)
else return(false,nil)
} .

procedure moveup(k)
{ (misplaced,h):=checkup[k]
while $\neg$ misplaced **do**
{ switch[k, h];
$k := h$
(misplaced,h):=checkup[k]
}
} .

procedure insertkey(a)
{ $n := n + 1$;
$K[n] := a$;
$H[n] := n$;

$H^-[n] := n$;
moveup(n)
} .

Variazione di un dato

Se i dati vengono variati la loro posizione nello heap può cambiare. In particolare se un dato viene diminuito la sua posizione nello heap si sposterà verso l'alto e se invece viene aumentato la posizione si sposterà verso il basso. Le operazioni hanno complessità $O(\log n)$. Nelle procedure qui sotto delineate si assume che il dato k-mo di K è stato modificato.

procedure decreasekey(k)
{ $h := H^-[k]$;
moveup(h)
} .

procedure increasekey(k)
{ $h := H^-[k]$;
movedown(h)
} .

Costruzione di uno heap

Per costruire uno heap, anziché inserire i dati ad uno ad uno e ripristinando lo heap ad ogni inserimento con uno spostamento verso l'alto delle chiavi, conviene inserire dapprima tutti i dati in modo disordinato e poi verificare dato per dato la proprietà a partire dal fondo dello heap con degli spostamenti verso il basso. Il vantaggio deriva dal fatto che la maggior parte delle chiavi si trova a livelli bassi dello heap e per questi sono richiesti pochi spostamenti verso il basso.

Analizziamo ora in dettaglio la complessità computazionale della costruzione di uno heap. Richiamiamo dapprima alcuni risultati preliminari sulle sommatorie $\sum_{k=0}^{n} a^k$ e $\sum_{k=0}^{n} k\,a^k$. La prima somma è equivalente alla seguente espressione

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

che si verifica immediatamente da

$$(a-1)\sum_{k=0}^{n} a^k = \sum_{k=0}^{n} a^{k+1} - \sum_{k=0}^{n} a^k = \sum_{k=1}^{n+1} a^k - \sum_{k=0}^{n} a^k = a^{n+1} - 1$$

Quindi se $|a| < 1$ e $n \to \infty$

$$\sum_{k\geq 0} a^k = \frac{1}{1-a}$$

Per la seconda sommatoria si ha

$$\sum_{k=0}^{n} k\,a^k = a\sum_{k=0}^{n} k\,a^{k-1} = a\,\frac{d}{da}\sum_{k=0}^{n} a^k = a\,\frac{d}{da}\frac{a^{n+1}-1}{a-1} =$$

$$a\,\frac{(n+1)\,a^n\,(a-1)-(a^{n+1}-1)}{(a-1)^2} = \frac{a}{(a-1)^2}\,(n\,a^n(a-1)-a^n+1) \tag{4}$$

Come sempre definiamo livello 0 quello della radice dello heap. Quindi la chiave in posizione j dello heap si trova al livello $\lfloor \log_2 j \rfloor$. Se gli elementi sono n il livello massimo dello heap è $\lfloor \log_2 n \rfloor$. In uno spostamento verso il basso dalla chiave j il numero massimo di scambi è il massimo valore k tale $2^k\, j \leq n$, quindi è $\lfloor \log_2(n/j) \rfloor$. In uno spostamento verso l'alto il numero di scambi è al più $\lfloor \log_2 j \rfloor$.

Allora per costruire uno heap il numero di scambi globale è al più

$$T(n) := \sum_{j=1}^{\lfloor n/2 \rfloor} \left\lfloor \log_2 \frac{n}{j} \right\rfloor \tag{5}$$

Possiamo trovare una limitazione superiore a $T(n)$ in due modi alternativi.

1 – L'espressione $\lfloor \log_2(n/j) \rfloor$ è uguale al valore costante k per tutti i valori j tali che

$$2^k \leq \frac{n}{j} < 2^{k+1} \qquad \Longrightarrow \qquad j\, 2^k \leq n < j\, 2^{k+1} \qquad \Longrightarrow \qquad j\, 2^k \leq n \leq j\, 2^{k+1} - 1$$

ovvero

$$\left\lceil \frac{n+1}{2^{k+1}} \right\rceil \leq j \leq \left\lfloor \frac{n}{2^k} \right\rfloor$$

Sia

$$J_k := \left| \left\{ j : \left\lceil \frac{n+1}{2^{k+1}} \right\rceil \leq j \leq \left\lfloor \frac{n}{2^k} \right\rfloor \right\} \right| = \left\lfloor \frac{n}{2^k} \right\rfloor - \left\lceil \frac{n+1}{2^{k+1}} \right\rceil + 1$$

Quindi si ha

$$T(n) = \sum_{j=1}^{\lfloor n/2 \rfloor} \left\lfloor \log_2 \frac{n}{j} \right\rfloor = \sum_{k=1}^{\lfloor \log_2 n \rfloor} k\, J_k$$

Sia $2^p \leq n \leq 2^{p+1} - 1$. Quindi $p := \lfloor \log_2 n \rfloor$. Siano $(n_0, n_1, \ldots, n_p)$ le cifre binarie di n (con $n_p = 1$), ovvero

$$n = n_0 + 2\, n_1 + 2^2\, n_2 + \ldots + 2^p\, n_p = \sum_{i=0}^{p} n_i\, 2^i$$

Allora

$$J_k = \left\lfloor \frac{n}{2^k} \right\rfloor - \left\lceil \frac{n+1}{2^{k+1}} \right\rceil + 1 = \left\lfloor \frac{\sum_{i=0}^{p} n_i\, 2^i}{2^k} \right\rfloor - \left\lceil \frac{\sum_{i=0}^{p} n_i\, 2^i + 1}{2^{k+1}} \right\rceil + 1 =$$

$$\sum_{i=k}^{p} n_i\, 2^{i-k} + \left\lfloor \frac{\sum_{i=0}^{k-1} n_i\, 2^i}{2^k} \right\rfloor - \sum_{i=k+1}^{p} n_i\, 2^{i-k-1} - \left\lceil \frac{\sum_{i=0}^{k} n_i\, 2^i + 1}{2^{k+1}} \right\rceil + 1 =$$

Siccome

$$\left\lfloor \frac{\sum_{i=0}^{k-1} n_i\, 2^i}{2^k} \right\rfloor = 0\,; \qquad \left\lceil \frac{\sum_{i=0}^{k} n_i\, 2^i + 1}{2^{k+1}} \right\rceil = 1$$

si ha

$$J_k = \sum_{i=k}^{p} n_i\, 2^{i-k} - \sum_{i=k+1}^{p} n_i\, 2^{i-k-1} = n_k + \sum_{i=k+1}^{p} n_i\, (2^{i-k} - 2^{i-k-1}) = n_k + \sum_{i=k+1}^{p} n_i\, 2^{i-k-1}$$

e allora

$$\sum_{j=1}^{\lfloor n/2 \rfloor} \left\lfloor \log_2 \frac{n}{j} \right\rfloor = \sum_{k=1}^{\lfloor \log_2 n \rfloor} k\, J_k = \sum_{k=1}^{p} k \left(n_k + \sum_{i=k+1}^{p} n_i\, 2^{i-k-1} \right) =$$

$$\sum_{k=1}^{p} k\, n_k + \sum_{k=1}^{p} \sum_{i=1}^{p} [i \geq k+1]\, k\, n_i\, 2^{i-k-1} = \sum_{k=1}^{p} k\, n_k + \sum_{i=1}^{p} n_i \sum_{k=1}^{p} [i \geq k+1]\, k\, 2^{i-k-1} =$$

$$\sum_{k=1}^{p} k\, n_k + \sum_{i=1}^{p} n_i \sum_{k=1}^{i-1} k\, 2^{i-k-1} = \sum_{k=1}^{p} k\, n_k + \sum_{i=1}^{p} n_i\, 2^{i-1} \sum_{k=1}^{i-1} k\, 2^{-k} =$$

Siccome, applicando (4), $\sum_{k=1}^{i-1} k\, 2^{-k} = 2\,(1 - 2^{-i+1} - (i-1)\, 2^{-i})$

$$\sum_{k=1}^{p} k\, n_k + \sum_{i=1}^{p} n_i\, 2^i\, (1 - 2^{-i+1} - (i-1)\, 2^{-i}) =$$

$$\sum_{k=1}^{p} k\, n_k + \left(\sum_{i=1}^{p} n_i\, 2^i \; - \sum_{i=1}^{p} n_i\, 2^i\, 2^{-i+1} - \sum_{i=1}^{p} n_i\, 2^i\, (i-1)\, 2^{-i} \right) =$$

$$\sum_{k=1}^{p} k\, n_k + n - n_0 \; - 2 \sum_{i=1}^{p} n_i - \sum_{i=1}^{p} n_i\, (i-1) =$$

$$n - n_0 - 2 \sum_{i=1}^{p} n_i + \sum_{k=1}^{p} (k - k + 1)\, n_k = n - \sum_{i=0}^{p} n_i$$

Quindi

$$T(n) = n - \sum_{i=0}^{p} n_i \; \leq n - 1$$

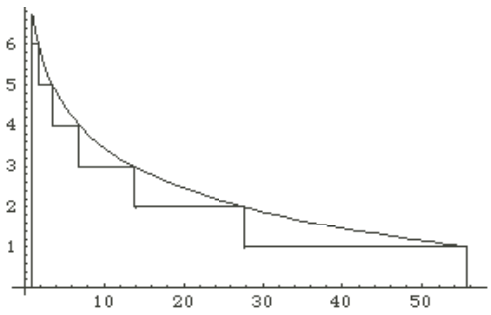
2 — L'espressione

$$\sum_{j=1}^{\lfloor n/2 \rfloor} \left\lfloor \log_2 \frac{n}{j} \right\rfloor$$

è equivalente all'area sotto la funzione a scalini (qui esemplificata per $n = 111$), che può essere maggiorata dall'integrale

$$\int_1^{n/2} \log_2 \frac{n}{x}\, dx = \Big[x\, \log_2 e + x\, \log_2 \frac{n}{x} \Big]_1^{n/2} =$$

$$\frac{n}{2}\, \log_2 e + \frac{n}{2}\, \log_2 2 - \log_2 e - \log_2 n \leq \frac{1 + \log_2 e}{2}\, n \approx 1.2213\, n$$



Analisi dell'ordinamento: Il numero di scambi globale (dovuto alle successive rimozioni della radice) è al più

$$S(n) := \sum_{j=1}^{n} \lfloor \log_2 j \rfloor$$

L'espressione $\lfloor \log_2 j \rfloor$ è uguale al valore costante k per tutti i valori j tali che $2^k \leq j \leq 2^{k+1} - 1$, quindi

$$S(n) = \sum_{k=0}^{\lfloor \log_2(n+1) \rfloor - 1} k\, 2^k + (n - 2^{\lfloor \log_2(n+1) \rfloor} + 1)$$

Sia $p = \lfloor \log_2(n+1) \rfloor - 1$, quindi

$$S(n) = \sum_{k=0}^{p} k\, 2^k + (n - 2^{p+1} + 1)$$

e applicando (4)

$$S(n) = 2\,(p\, 2^p - 2^p + 1) + (n - 2^{p+1} + 1) = p\, 2^{p+1} - 2^{p+1} + 2 + n - 2^{p+1} + 1$$

$$n \geq 1 \quad \Longrightarrow \quad \frac{n}{2} \geq \frac{1}{2} \quad \Longrightarrow \quad n - \frac{n}{2} \geq \frac{1}{2} \quad \Longrightarrow \quad n \geq \frac{n+1}{2}$$

$$p = \lfloor \log_2(n+1) \rfloor - 1 \leq \log_2(n+1) - 1 = \log_2(n+1) - \log_2 2 = \log_2 \frac{n+1}{2} \leq \log_2 n$$

$$S(n) \leq 2\, n\, \log_2 n + n + 3 \in O(n \log n)$$