LABORATORIO DI ASD
A.A. 2017/2018
APPUNTI

ALBERTO POLICRITI
(VERSIONE IN FASE DI ELABORAZIONE DI PRECEDENTI VERSIONI SCRITTE DA SIMONE SCALABRIN E DA CRISTIAN DEL FABBRO)

1. Complessità e prestazioni

1.1. **Introduzione.** Vogliamo introdurre in questo capitolo alcuni dei concetti fondamentali dello studio analitico e sperimentale del tempo d'esecuzione di un programma. Inizieremo discutendo i concetti di dimensione dei dati di un algoritmo e di tempo d'esecuzione ottimo e pessimo.

Per una buona comprensione del materiale è necessario conoscere il concetto di complessità asintotica e le relative notazioni (O-grande).

1.1.1. *Tempo di esecuzione di un programma.* Tra gli oggetti di studio di questo laboratorio, il *tempo d'esecuzione* di un programma riveste un'importanza particolare. Supporremo fissato sin dall'inizio un linguaggio di programmazione (qualsiasi linguaggio va bene) e considereremo programmi scritti in quel linguaggio. Dato dunque un programma P da eseguirsi sui dati di ingresso d, siamo interessati a determinare il tempo che trascorre dall'inizio dell'esecuzione di P su d sino al momento in cui l'esecuzione termina; per il momento possiamo pensare di indicare questo valore con $\mathcal{T}_P(d)$. In particolare, ci interesserà studiare $\mathcal{T}_P(d)$ al variare di d, determinando, se e quando possibile, una dipendenza (magari approssimata) funzionale esplicita.

Osserviamo, per cominciare, che sebbene il tempo d'esecuzione di P dipenda dai dati d, raramente dipende da tutte le informazioni contenute in d. Ad esempio, il tempo di esecuzione di un programma per la somma degli elementi di un vettore di interi, *quando gli interi non sono "troppo grandi"*, non dipende dagli elementi contenuti nel vettore ma solo dal loro numero.

Chiamiamo *dimensione* dei dati quegli aspetti di essi che influenzano il tempo d'esecuzione di un programma. Nel caso sopra citato (a patto di chiarire cosa intendiamo per interi "troppo grandi"), la dimensione dei dati è costituita dal numero di elementi del vettore stesso. In altri casi, determinare quale sia la dimensione dei dati può essere meno ovvio.

Esempio 1 (Ricerca lineare)**.** *Consideriamo l'algoritmo (1). Si tratta di una semplice ricerca lineare: si vuole determinare se il valore* x *compare nel vettore* V*, di dimensione* n*; in caso positivo, viene stampato l'indice* i *per cui* $V[i] = x$*; altrimenti viene stampato* $n + 1$*. I numeri di linea riportati nella figura non*

fanno parte del programma e sono stati inseriti solo per facilitare il riferimento ai singoli comandi.

Il tempo di esecuzione di questo programma dipende da tre fattori: la lunghezza n del vettore V, la posizione di x in V, la massima dimensione di un intero in V. Determinare la posizione di x in V è proprio lo scopo per cui il programma è stato scritto, appare pertanto irragionevole inserire questo parametro relativo ai dati nella loro dimensione (equivarrebbe a supporre risolto il problema). La dimensione massima di uno degli elementi memorizzati in V va considerata nello stimare il costo delle istruzioni a linea 2 e a linea 5. Assumendo che tale costo sia costante, anche in questo caso una ragionevole scelta per la dimensione dei dati è la lunghezza n del vettore.

Esercizio 1. *Si dimostri che ogniqualvolta il valore del massimo intero manipolato è $O(|V|)$, possiamo utilizzare il numero di elementi in V come parametro per la complessità del precedente algoritmo.*

Algorithm 1 RICERCA LINEARE(V,n,x)

```
1: i ← 1
2: while (i < n) AND (V[i] ≠ x) do
3:    i ← i + 1
4: end while
5: if V[i] ≠ x then
6:    i ← n + 1
7: end if
8: return i
```

Per programmi che manipolano strutture dati, la dimensione delle strutture in gioco è spesso una buona scelta quale dimensione dell'input (sempre a patto di non usare confronti o altre operazioni elementari su interi troppo grandi). Ad esempio, nel caso di un programma di ordinamento, il numero di dati da ordinare costituirà la scelta ovvia. In altri casi, la dimensione può dipendere dal valore dei dati. Un interessante esempio di questo tipo è un (qualunque) programma per determinare l'n-esimo numero primo. Siccome ci aspettiamo che il tempo sia una funzione crescente di n, appare naturale far dipendere dal valore n la dimensione dell'input. Un semplice ragionamento suggerisce quale sia una scelta opportuna per il parametro d rispetto al quale esprimere la complessità computazionale dell'algoritmo in questione.

Esercizio 2. *Si implementi il famoso* crivello di Eratostene *e si illustri, motivandola, una ragionevole scelta per la dimensione d dell'input.*

Come altro esempio, consideriamo il problema del commesso viaggiatore. Vi sono n città, tutte collegate tra loro; la distanza tra la città i e la città j è data da un intero positivo (rappresentabile) $d_{i,j}$. Si vuole determinare il percorso più breve che visiti tutte ed n le città senza mai passare due volte per la stessa città. In questo caso i dati del problema sono costituiti dalla matrice (quadrata $n \times n$)

delle distanze: la dimensione dei dati è dunque n^2. La soluzione del problema non è semplice e la complessità di un qualunque algoritmo che risolva il problema è $\Omega(n^2)$ (perché?).

L'esempio della ricerca lineare mostra che sebbene il tempo d'esecuzione dipenda dalla dimensione dei dati in input, nella pratica questo può non essere il solo parametro importante. Nel caso della ricerca, infatti, è cruciale la posizione nel vettore dell'elemento da ricercare. È chiaro, in altri termini, che dati con la stessa dimensione (vettori di uguale lunghezza, contenenti interi non troppo grandi) possono dare luogo a tempi d'esecuzione diversi, a seconda, appunto, della posizione di x nel vettore. Tra tutti i possibili tempi d'esecuzione che si ottengono con dati di una fissata dimensione, due sono i casi più interessanti e studiati.

- Il *caso ottimo* è quello che corrisponde ai dati che generano l'esecuzione più veloce. (Nell'esempio, si tratta del caso in cui $x = V[1]$.) Indicheremo il tempo del caso ottimo del programma P su dati di dimensione n con $\mathcal{T}_P^{ott}(n)$.
- Il *caso pessimo* è quello che corrisponde ai dati che generano l'esecuzione più lenta. (Nell'esempio, si tratta del caso in cui x non compare in V.) Indicheremo il tempo del caso pessimo del programma P su dati di dimensione n con $\mathcal{T}_P^{pes}(n)$, o più semplicemente $\mathcal{T}_P(n)$.

Il tempo del caso ottimo dà una limitazione inferiore al tempo di calcolo di un programma. Pur con eccezioni interessanti, spesso non dipende dalla dimensione dei dati e non è di grande aiuto nell'analisi di un programma. Al contrario, il tempo del caso pessimo è di grande importanza, perché stabilisce un limite superiore al tempo necessario al programma per dati della dimensione scelta. È abbastanza frequente che anche per dati che non costituiscono il caso pessimo, il programma si discosti di poco dal suo comportamento pessimo, almeno per dati di grandi dimensioni.

Talvolta sarà interessante studiare anche il tempo d'esecuzione per casi "intermedi", cioè che non sono né quello ottimo né quello pessimo. Introdurremo di volta in volta la notazione che useremo in questi casi. Infine, un altro valore di grande importanza, e che rivestirà un ruolo di primo piano in questo scritto, è il tempo medio d'esecuzione, $\mathcal{T}_P^{med}(n)$, dove la media è intesa su tutti i dati della stessa dimensione (dunque comprendendo sia quelli del caso ottimo che quelli del caso pessimo che di tutti i casi intermedi). Rimandiamo al seguito una discussione più approfondita relativa al tempo medio.

1.2. **Analisi della complessità.** Scopo dell'analisi di un programma è quello di determinare (o stimare sperimentalmente) la forma della funzione $\mathcal{T}_P(n)$ al variare di n.

Un momento di riflessione mostra subito, però, che l'effettivo tempo d'esecuzione pessimo $\mathcal{T}_P(n)$ dipende in realtà non solo dai parametri P e n, ma anche da una miriade di altri fattori, solo alcuni dei quali noti al momento in cui un programma viene scritto. Tra questi ricordiamo i seguenti.

(1) Il *calcolatore* su cui il programma viene mandato in esecuzione: è chiaro che un processore di ultima generazione eseguirà P più velocemente di un processore di qualche anno fa. Anche due macchine che eseguano (in media) lo stesso numero di istruzioni macchina per secondo, possono eseguire P in tempi diversi. Questo perché il repertorio delle istruzioni (e dei modi di indirizzamento, e dei bus utilizzati, ecc.) influenza non poco l'efficienza di uno specifico programma.
(2) Il *compilatore* o l'*interprete* utilizzato: P, per poter essere eseguito su uno specifico calcolatore, deve essere tradotto nel suo linguaggio macchina. Due compilatori/interpreti diversi per lo stesso linguaggio generano sequenze diverse di istruzioni macchina e, dunque, influenzano in modo diverso il tempo di esecuzione.
(3) Il *carico del sistema*: durante l'esecuzione di P possono succedere molte cose, per esempio può arrivare un'interruzione ad alta priorità che richiede di essere gestita. Il fenomeno è ancora più chiaro in un calcolatore (e relativo sistema operativo) che permetta l'esecuzione "simultanea" di più programmi (che si risolve, in realtà, nell'esecuzione a rotazione di piccole quote dei vari programmi in gioco). È chiaro che tutti questi eventi hanno un'influenza tutt'altro che trascurabile su $\mathcal{T}_P(n)$.
(4) Il *sistema operativo*: il modo in cui il sistema operativo gestisce la memoria può avere grande influenza sul tempo d'esecuzione, specie quando questa richieda grosse quantità di memoria (che potrebbe essere allocata tutta insieme all'inizio dell'esecuzione, oppure a blocchi durante l'esecuzione stessa; inoltre la memoria potrebbe essere virtuale, cioè vista come illimitata dall'utente, ma gestita dal sistema operativo con accessi alla memoria secondaria).

Ciò che abbiamo espresso come $\mathcal{T}_P(n)$, dunque, è una funzione di molti altri parametri, oltre ad n. Così tanti che, a prima vista, appare difficile poterla ragionevolmente approssimare con una funzione del solo n. Osserviamo, tuttavia, che se si eccettuano gli ultimi due punti sopra menzionati, gli altri fattori elencati non dipendono dai dati. Partendo da questa osservazione si può dimostrare che, con ragionevole approssimazione e a meno dell'influenza dei fattori (3) e (4) (e di altri aspetti che dipendano non dal programma ma dal *contesto* nel quale è eseguito), possiamo stabilire per che $\mathcal{T}_P(n)$ esistono delle costanti a_1, a_2, b_1 e b_2, tali che vale la relazione:

$$a_1 \cdot \mathcal{T}_P(n) + b_1 \leq \mathcal{T}_P(n) \leq a_2 \cdot \mathcal{T}_P(n) + b_2. \tag{1}$$

Nella precedente relazione $\mathcal{T}_P(n)$ dipende solo da P e dalla dimensione dei dati n, mentre a_1, a_2, b_1 e b_2 sono costanti che dipendono da P, dal compilatore e dal calcolatore utilizzati, ma non dai dati.

1.2.1. *Passi di calcolo.* Operazioni quali assegnamento di un valore ad una variabile, confronto fra numeri, accesso ad una variabile o a una cella di un array, operazioni aritmetiche di base e simili possono essere considerate come operazioni che si eseguono in un tempo costante, cioè che vengono tutte eseguite al più

in un tempo t_{max} che dipende principalmente dal tipo di calcolatore utilizzato. Chiamiamo *passo di calcolo* questa unità convenzionale di tempo.

Esempio 2 (Tempo di esecuzione della ricerca lineare)**.** *Applichiamo il metodo descritto nel paragrafo precedente per il calcolo del numero dei passi del programma (1). Valuteremo sia $\mathcal{T}^{ott}(n)$ che $\mathcal{T}^{pes}(n)$. Il caso ottimo si ha, chiaramente, quando x compare in $V[1]$. La tabella seguente riassume i passi con cui le varie linee del programma contribuiscono al tempo d'esecuzione.*

LINEA	*PASSI*	*RIPETIZIONI*	*TOTALE*
1	*1*	*1*	*1*
2	*2*	*1*	*2*
3	*1*	*0*	*0*
5	*1*	*1*	*1*
6	*1*	*0*	*0*
PASSI TOTALI NEL CASO OTTIMO			*4*

L'assegnamento della linea 1 contribuisce con un passo di calcolo. La linea 2, la guardia del ciclo, contribuisce con due passi (per l'espressione logica composta), che vengono moltiplicati per 1, il numero di volte che la guardia viene valutata nel caso ottimo. Il corpo del ciclo (linea 3) è un assegnamento (1 passo) che non viene mai eseguito. Infine, la guardia del comando condizionale della linea 5 contribuisce per un passo di calcolo, mentre il ramo **then**, *che produrrebbe un ulteriore passo di calcolo, non è mai eseguito nel caso ottimo. Possiamo pertanto concludere che $\mathcal{T}^{ott}(n) = 4$ e che l'effettivo tempo d'esecuzione soddisfa, per ogni n, la relazione*

$$a_1 \cdot 4 + b_1 \leq \mathcal{T}^{ott}(n) \leq a_2 \cdot 4 + b_2$$

Il calcolo del numero dei passi nel caso pessimo fornisce, in modo analogo, la seguente tabella.

LINEA	*PASSI*	*RIPETIZIONI*	*TOTALE*
1	*1*	*1*	*1*
2	*2*	*n*	*2n*
3	*1*	*n-1*	*n-1*
5	*1*	*1*	*1*
6	*1*	*1*	*1*
PASSI TOTALI NEL CASO PESSIMO			*3n+2*

Dalla tabella deriviamo dunque che $\mathcal{T}^{pes}(n) = 3n + 2$.

In conclusione, a meno di fattori costanti, possiamo determinare il tempo d'esecuzione di un programma P nel caso pessimo (o ottimo, o in un qualsiasi caso intermedio) calcolando il numero di passi di calcolo, che indichiamo anch'esso con $\mathcal{T}_P^{pes}(n)$, o, più semplicemente, con $\mathcal{T}_P(n)$. Tale valore limita il tempo effettivo d'esecuzione a meno di fattori costanti. La funzione $\mathcal{T}_P(n)$ si dice complessità in tempo del programma P.

1.2.2. *Esercizi.* Si consideri il seguente algoritmo, che ordina l'array A di n elementi tramite l'algoritmo denominato Insertion Sort:

Algorithm 2 INSERTION SORT(A)

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

- Si implementi l'algoritmo.
- Si costruisca una tabella con il numero di passi da compiere nel caso ottimo e pessimo di Insertion Sort (possibilmente spiegando quali sono tali casi).

Si consideri il seguente algoritmo, che ordina l'array A di n elementi tramite l'algoritmo denominato Bubble Sort:

Algorithm 3 BUBBLE SORT(A)

```
1: for j ← n − 1 to 1 do
2:     for i ← 0 to j − 1 do
3:         if A[i] > A[i + 1] then
4:             temp ← A[i]
5:             A[i] ← A[i + 1]
6:             A[i + 1] ← temp
7:         end if
8:     end for
9: end for
```

- Si implementi l'algoritmo.
- Si costruisca una tabella con il numero di passi da compiere nel caso ottimo e pessimo di Bubble Sort (possibilmente spiegando quali sono tali casi).

2. TEMPO DI ESECUZIONE DI UN ALGORITMO

2.1. **Misura dei tempi di esecuzione di un algoritmo.** Per misurare il tempo di esecuzione di un algoritmo P su un dato input d è sufficiente utilizzare una funzione che rilevi il tempo di inizio e di fine dell'esecuzione di P su d e bisogna infine effettuare la differenza di tali rilevazioni.

Un problema importante che si può riscontrare è quello della rilevazione di tempi su programmi efficienti che operano su input di dimensione limitata. Tale problema però non è molto importante dal punto di vista pratico perché, solitamente, non fa alcuna differenza se un algoritmo necessita di qualche millesimo o di qualche decimo

di secondo e perché le misurazioni rilevate sono troppo aleatorie, dipendono cioè più da fattori esterni (ad esempio, il carico della CPU in quel momento) che dall'algoritmo e dall'input. Inoltre tali rilevazioni sarebbero fortemente dipendenti dalla *granularità del sistema*, cioè dal minimo tempo misurabile dal sistema utilizzato.

Ci concentreremo dunque sulla misurazione dell'esecuzione di programmi su dati con dimensione elevata, vedendo in seguito come ridurre il problema sopra enunciato a quello che sappiamo risolvere.

In Java[1] esiste la funzione `System.currentTimeMillis()` che restituisce un intero di tipo `long` che rappresenta il tempo corrente (in millisecondi) rispetto alla mezzanotte del 1 gennaio 1970 UTC (ora e data sono state scelte in modo arbitrario ma condivise da quasi tutti i sistemi operativi). La durata di un algoritmo è quindi facilmente calcolabile come la differenza fra l'istante immediatamente successivo alla conclusione dell'algoritmo e l'istante immediatamente precedente all'inizio dell'algoritmo. È da rimarcare che in questo contesto non ci interessa calcolare il tempo di esecuzione del programma (che può comprendere i tempi di inizializzazione o altro) ma *solo* il tempo di esecuzione dell'algoritmo.

2.1.1. *Esercizi.*

- Si utilizzi l'orologio di macchina per misurare il tempo di esecuzione della ricerca lineare, di Bubble Sort e di Insertion Sort nel caso ottimo e nel caso pessimo, per vettori di dimensione 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 (o più ...).
- Disegnate i grafici dei tempi rilevati.
- Confrontate i risultati ottenuti per i due algoritmi di ordinamento (Bubble Sort e Insertion Sort). Quale dei due algoritmi è più veloce (a parità della dimensione dell'input)?
- Siete soddisfatti dei tempi che avete misurato? Vi sembra che ci siano delle anomalie (differenze rispetto alla teoria)? Se si, come le spiegate?

2.2. **Dominare la granularità.** Il primo problema che si presenta con la valutazione delle prestazioni è quello di misurare con una certa precisione il tempo d'esecuzione di un certo programma (o di alcune sue parti). Il problema è assai semplificato in un ambiente ad unico utente, quale un calcolatore personale, nel quale una volta mandato in esecuzione un programma, questo mantiene il controllo del calcolatore sino al termine dell'esecuzione, con l'eccezione del trattamento delle interruzioni. In un contesto del genere, il modo più semplice e diretto, per effettuare le misure che ci interessano è quello di usare esplicitamente l'orologio di macchina, un meccanismo presente su tutti i calcolatori e tutti i sistemi operativi. Si tratta, in genere, di un registro contenente un numero naturale, che viene incrementato ogni δ secondi. Il valore di questo registro viene letto con opportune chiamate di sistema, quasi sempre disponibili nei maggiori linguaggi ad alto livello. La risoluzione

[1] In C/C++ la lettura dell'orologio può essere effettuata mediante la funzione `clock()` che restituisce un valore di tipo `clock_t` (che nella maggior parte dei compilatori equivale ad un intero) corrispondente al numero di "battiti" dell'orologio trascorsi dall'inizio dell'esecuzione del programma. La costante `CLOCKS_PER_SEC` rappresenta il numero di "battiti" per secondo, e può essere utilizzata per convertire i tempi misurati in secondi.

dell'orologio, δ, viene detta *granularità del sistema* e dipende dalla macchina e dal sistema operativo usati e varia solitamente da qualche decimo a pochi millesimi di secondo. Leggendo l'orologio prima e dopo l'evento di nostro interesse otteniamo il tempo trascorso, purché l'evento abbia avuto una durata superiore alla risoluzione dell'orologio (si veda il paragrafo (2.1)).

Uno degli scopi principali della misura dei tempi di un algoritmo è quello di confrontare la complessità teorica con i tempi sperimentali. La misura dei tempi sperimentali, inoltre, ci può consentire di determinare le costanti nascoste dalla notazione asintotica (nel caso in cui sia stata eseguita l'analisi per passi possiamo determinare la costante moltiplicativa dovuta all'ambiente in cui lavoriamo). In linea di principio per determinare due costanti (ad esempio, i coefficienti di una retta) bastano due misurazioni. È opportuno, tuttavia, non limitarci a due sole misure. Infatti, come tutte le misure sperimentali, anche le nostre andranno soggette ad errore ed è quindi opportuno effettuare più rilevazioni e determinare la funzione che meglio si adatta (in un senso che sarà precisato in seguito) a tutte le misure compiute.

Se andassimo a misurare il tempo di calcolo dell'algoritmo (ricercaLineare) per dimensioni del vettore 10, 20, 30, 40, 50, 100, 500, 1000 otterremmo, con molta probabilità, risultati apparentemente misteriosi e senza parentela con la relazione $\mathcal{T}_p(n) = 3n + 2$. La loro spiegazione non è tuttavia difficile: abbiamo cercato di misurare la durata di eventi che sono di gran lunga troppo brevi per la risoluzione del nostro strumento di misura. Per questo dovremmo ottenere molti tempi pari a zero e forse qualche misurazione sarà di qualche secondo, o una sua frazione. I tempi diversi da zero saranno solamente del "rumore" che dipende dal verificarsi di qualche evento al contorno, quale per esempio un'interruzione.

Prima di cercare una soluzione al problema che abbiamo appena incontrato, poniamoci una questione generale: quale deve essere la durata minima di un evento, affinché la si possa misurare con un orologio di risoluzione δ con un errore "ragionevole"? Osserviamo, innanzitutto, che ogni misurazione che fornisce un tempo t corrisponde ad un tempo effettivo che appartiene all'intervallo $[t-\delta;\, t+\delta]$. L'errore di misurazione δ non può essere eliminato se non ricorrendo ad uno strumento più preciso. D'altra parte, una cosa è fare un errore di un millimetro misurando una distanza di due millimetri; altra cosa, avere lo stesso errore misurando una distanza di un metro: nel primo caso la misura presenta un errore percentuale del cinquanta per cento; nel secondo, dell'uno per mille. L'errore di misurazione è quindi di $E = \delta/t$. Perciò, se si vuole un errore $E \leq k$, bisogna misurare un tempo $t \geq \delta/k$ (che potremmo chiamare *tempo minimo*). Ad esempio, tollerando un errore del 5% ($k = 0,05$), serve rilevare l'esecuzione per un tempo pari ad almeno 20δ. Per i nostri scopi una stima con errore massimo percentuale del 2% o del 5% è più che sufficiente.

Proponiamo di seguito l'algoritmo (4) per calcolare la granularità del sistema.

Come fare, però, per misurare le prestazioni di un nostro programma per dimensioni del problema n limitate? La risposta non è difficile: basta ripetere più volte lo stesso evento, diciamo rip volte, misurando il tempo trascorso per le rip ripetizioni. Il valore rip sarà scelto, con qualche tentativo, in modo che i tempi misurati

Algorithm 4 GRANULARITÀ()

```
1: t0 ← getTime()
2: t1 ← getTime()
3: while (t1 == t0) do
4:     t1 ← getTime()
5: end while
6: return (t1 − t0)
```

siano dell'ordine di grandezza che ci permette di ottenere la precisione desiderata. A questo punto una semplice divisione per rip ci dà il tempo del singolo evento.

Per calcolare il valore rip si può agire in maniera lineare o, meglio ancora, per dicotomia come presentato nell'algoritmo (5).

Algorithm 5 CALCOLARIP(P,d,tMin)

```
1: t0, t1 ← 0
2: rip ← 1
3: while (t1 − t0 ≤ tMin) do
4:     rip ← rip · 2 # stima di rip con crescita esponenziale
5:     t0 ← getTime()
6:     for i ← 1 to rip do
7:         EXECUTE(P,d)
8:     end for
9:     t1 ← getTime()
10: end while
11: # ricerca esatta del numero di ripetizioni per bisezione
12: # approssimiamo a 5 cicli
13: max ← rip
14: min ← rip/2
15: cicliErrati ← 5
16: while (max − min ≥ cicliErrati) do
17:     rip ← (max + min)/2 # valore mediano
18:     t0 ← getTime()
19:     for i ← 1 to rip do
20:         EXECUTE(P,d)
21:     end for
22:     t1 ← getTime()
23:     if (t1 − t0 ≤ tMin) then
24:         min ← rip
25:     else
26:         max ← rip
27:     end if
28: end while
29: return max
```

Di seguito presentiamo anche l'algoritmo (6) che esegue un programma P per rip volte (dipendente dall'errore desiderato).

Algorithm 6 CALCOLO DEI TEMPI(P,d,rip)

```
1: t_0 ← getTime()
2: for i ← 1 to rip do
3:    EXECUTE(P,d)
4: end for
5: t_1 ← getTime()
6: t_tot ← t_1 − t_0 # tempo totale di esecuzione
7: t_sing ← t_tot/rip # tempo medio di esecuzione
```

In verità, nell'algoritmo (6) bisognerebbe scorporare anche il tempo necessario per l'esecuzione del ciclo `FOR` e delle chiamate alla funzione `getTime`. In pratica, tali operazioni sono solitamente trascurabili rispetto al tempo di esecuzione di P.

Per riassumere, una volta determinato il valore di granularità δ del sistema e fissata una tolleranza E, bisogna determinare il valore del tempo minimo $tMin$ che le ripetizioni devono assicurare sul sitema. A questo punto, per ogni algoritmo P e per ogni input d bisogna calcolare il numero di ripetizioni rip necessarie. Infine si procede alla misurazione vera e proprio che consta in rip ripetizioni di P su d.

2.2.1. *Esercizi.*

- Si determini la granularità del sistema in uso.
- Si determini il numero di ripetizioni necessarie per misurare il tempo di esecuzione della Ricerca Lineare, di Insertion Sort e di Bubble Sort nei casi migliori e peggiori, con un errore massimo del 5% per vettori di dimensione 10, 100, 200, 500, 1000, 2000, 5000, 10000, ... e si determinino i tempi (medi) di esecuzione

3. TEMPO MEDIO D'ESECUZIONE

3.1. **Tempo medio e distribuzione dei dati.** La valutazione (analitica o sperimentale) del tempo pessimo lascia spesso il principiante insoddisfatto. Questi teme, infatti, che l'analisi sia sovradimensionata, e che, nella pratica, il caso pessimo si presenti talmente di rado da rendere la sua valutazione solo un caso da manuale. Ci sono algoritmi in cui la situazione è davvero quella temuta (o forse, meglio, sperata) dal pricipiante[2], e ci sono, al contrario, algoritmi (e sono probabilmente la maggioranza) per i quali la valutazione asintotica del caso pessimo non è molto diversa da quella "media". Ciò non vuol dire, ovviamente, che le costanti in gioco siano le stesse, o che il comportamento asintotico inizi a manifestarsi per dati di dimensione analoga. In questo testo ci occuperemo spesso di valutazioni medie, ed avremo modo, con ciò, di confermare proprio la validità delle valutazioni del caso pessimo.

[2]Un caso assai importante è costituito dall'algoritmo di ordinamento Quicksort, che sarà discusso più avanti.

Il primo problema che si pone con la valutazione del tempo d'esecuzione "in media" è che esso dipende, evidentemente, dai dati che vengono presentati al programma. Mentre, cioè, il caso pessimo (e quello ottimo) è perfettamente determinato, il caso medio non esiste: esistono casi (istanze del problema) che non sono né quello pessimo né quello ottimo, e siamo interessati al tempo medio necessario ad eseguire il nostro programma al variare dei dati (di dimensione fissata n) tra i due estremi costituiti dal caso ottimo e da quello pessimo. D'altra parte, tale tempo medio dipende da quali di questi casi intermedi si presentano più di frequente. È chiaro che se la ricerca lineare incerta (i.e. non siamo certi che l'elemento appaia nel vettore) è usata nel 50% dei casi per ricercare un elemento che sappiamo appartenere al vettore, il suo tempo medio è migliore (cioè minore) di quello che si otterrebbe con dati di cui sappiamo che l'elemento da ricercare appartiene al vettore solo nel 20% dei casi.

Il tempo medio d'esecuzione di un programma, dunque, ha senso solo dopo aver stabilito una certa distribuzione dei dati, cioè con quale frequenza si presentano tutte le varie istanze del problema della fissata dimensione n. Non esiste il tempo medio d'esecuzione di un programma, ma solo il tempo medio supposta una certa distribuzione[3].

Il caso più semplice e studiato è quello in cui tutti i dati si suppongono distribuiti in modo uniforme, ovvero tutti identicamente probabili. Supponiamo, pertanto, che esistano k_n istanze diverse del problema di dimensione n, individuate ciascuna da un intero i, $1 \leq i \leq k_n$. Indichiamo con $\mathcal{T}_P(n, i)$ il tempo necessario ad eseguire il programma P sull'istanza i (di dimensione fissata n). In ipotesi di distribuzione uniforme, il tempo medio si ottiene con la semplice media aritmetica dei tempi:

$$\mathcal{T}_P^{med}(n) = \frac{1}{k_n} \sum_{i=1}^{k_n} \mathcal{T}_P(n, i) \tag{2}$$

Esempio 3 (Ricerca lineare certa)**.** *Si supponga di voler cercare un elemento all'interno di un vettore, nell'ipotesi che tale elemento effettivamente compaia all'interno del vettore stesso. L'algoritmo che risolve questo problema è l'algoritmo (1) escluse le righe* $5-7$*. Una semplice analisi dei passi di calcolo mostra che* $\mathcal{T}_P^{pes}(n) = 3n$*, che si ottiene quando il valore cercato* x *compare come ultimo elemento del vettore* V *dato in input (questa è l'istanza, o caso, pessimo). Le varie istanze dei dati sono determinate dalla posizione con cui* x *compare in* V*; possiamo pertanto identificarle con questa informazione. L'istanza 1, dunque, è quella in cui* x *compare in* $V[1]$ *(è l'istanza del caso ottimo); nell'istanza* i*,* x *compare in* $V[i]$*. Indicando con* $T(n, i)$ *il numero di passi di calcolo necessari per l'istanza* i *di dimensione* n*, un semplice conto mostra che*

$$T(n, i) = 3i.$$

[3] Questo costituisce anche una limitazione delle valutazioni del tempo medio, visto che raramente i dati "nella pratica" si conformeranno a quelle distribuzioni che assumiamo durante la valutazione in sede di analisi.

Supponiamo ora i dati distribuiti in modo uniforme. Assumiamo, cioè, che i dati vengano presentati al programma in modo che la frequenza del caso "x è in V[i]" sia la stessa per ogni i; ovvero, tutti i casi si presentano con la stessa probabilità. In tali ipotesi il tempo medio d'esecuzione del programma si ottiene con l'equazione (2), dove, nel caso in esame, $k_n = n$*:*

$$\mathcal{T}_P^{med}(n) = \frac{1}{n}\sum_{i=1}^{n} 3i. \tag{3}$$

È utile cercare una forma chiusa per il numero medio dei passi di calcolo. In questo caso particolarmente semplice, sfruttando la nota formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

si ha

$$\mathcal{T}_P^{med}(n) = \frac{3}{2}(n+1).$$

Per questo problema molto semplice, se n *è dispari il tempo medio coincide col tempo dell'istanza* $\lceil n/2 \rceil$*, mentre se* n *è pari il tempo medio coincide con la media dei tempi delle due istanze* $n/2$ *e* $n/2+1$*. Si osservi dunque che, in generale, il tempo medio non coincide col tempo di calcolo di qualche istanza.*

Esempio 4 (Ricerca lineare incerta)**.** *Riprendiamo ora la ricerca lineare incerta dell'algoritmo (1), di cui abbiamo studiato i casi ottimo e pessimo nell'esempio (2). Vi sono in questo caso* $n+1$ *istanze diverse di dimensione* n*: le prime* n *istanze corrispondono ai casi in cui* x *compare in una posizione del vettore; l'istanza* $n+1$ *corrisponde al caso in cui* x *non è presente in* V *. Col solito metodo tabellare usato nell'esempio otteniamo, per il numero di passi, la seguente tabella:*

LINEA	*PASSI*	*RIP 1*	*RIP* $i \le n$	*RIP* $n+1$
1	*1*	*1*	*1*	*1*
2	*2*	*1*	i	n
3	*1*	*0*	$i-1$	$n-1$
5	*1*	*1*	*1*	*1*
6	*1*	*0*	*0*	*1*
$T(n,i)$		*4*	$3i+1$	$3n+2$

Se ora supponiamo tutte le $n+1$ *istanze equiprobabili, possiamo calcolare il numero medio di passi di calcolo usando la (2):*

$$\mathcal{T}^{med}(n) = \frac{1}{n+1}\sum_{i=1}^{n+1} T(n,i) = \frac{1}{n+1}\sum_{i=1}^{n}(3i+1) + \frac{3n+2}{n+1}.$$

Con semplici calcoli, ed usando ancora la (3), si ottiene

$$\mathcal{T}^{med}(n) = \frac{n}{n+1} + \frac{3}{2}n + \frac{3n+2}{n+1} = \frac{3}{2}n + 2 + \frac{2n}{n+1}.$$

La relazione appena ottenuta, lo ricordiamo, è il numero medio di passi di calcolo necessari alla ricerca lineare incerta nell'ipotesi in cui le $k_n = n+1$ *istanze di dimensione* n *sono tutte equiprobabili. In questa ipotesi il caso in cui* x *non compare nel vettore è assai raro: esso è supposto presentarsi al programma con frequenza* $1/(n+1)$*. La probabilità del caso in cui* x *compare nel vettore è invece assai elevata:* $1 - 1/(n+1) = n/(n+1)$*: per* n *abbastanza grande si tratta di un valore prossimo a 1. Ricordiamo che questa è una ipotesi che stiamo facendo, che riguarda l'uso che del programma verrà fatto (e di conseguenza, fornisce un valore del tempo di calcolo medio valido solo in quell'ipotesi). In molti casi si tratterà di una ipotesi irrealistica.*

Si potrebbe per esempio argomentare che un caso assai più interessante è quello in cui i due casi "x *appartiene a* V*" e "*x *non appartiene a* V*" sono tra loro equiprobabili; quandi poi* x *compare in* V*, sono a loro volta equiprobabili tutti i vari casi "*x *compare in* $V[i]$*" al variare di* i *tra 1 e* n*. Ricapitolando, l'istanza* $n+1$ *si presenta nel 50% dei casi; nel rimanente 50%, si presentano in modo equiprobabile le istanze tra 1 e* n*. Come calcolare il tempo medio in questo caso? La prossima sezione affronta il problema.*

3.1.1. *Media pesata.* Quando i dati non sono distribuiti in modo uniforme, ma, come alla fine dell'esempio precedente, la frequenza delle diverse istanze non è la stessa, occorre sostituire la media aritmetica della (2) con una relazione più generale. La media pesata è data da una sommatoria in cui "pesano di più" quegli addendi che sono supposti presentarsi più frequentemente di altri. Il tempo d'esecuzione relativo ad una particolare istanza viene pesato con la frequenza (o probabilità) con la quale supponiamo presentarsi quell'istanza. Supponendo di avere a che fare con k_n istanze di dimensione n, ognuna di esse verrà pesata con una probabilità p_i, per $1 \leq i \leq k_n$. La somma di tali probabilità, ovviamente, deve essere uguale ad uno (le k_n istanze esauriscono infatti tutte le possibilità che si possono presentare):

$$\sum_{i=1}^{k_n} p_i = 1. \tag{4}$$

Identificando, come prima, ogni istanza con il proprio indice $i, 1 \leq i \leq k_n$, la media pesata del tempo d'esecuzione è allora definita come:

$$\mathcal{T}_p^{med}(n) = \sum_{i=1}^{k_n} p_i \mathcal{T}_p(n, i). \tag{5}$$

Si noti che la media aritmetica è un caso particolare della media pesata quando tutti i pesi sono uguali a $1/k_n$.

Se riprendiamo il caso della ricerca lineare incerta che abbiamo appena discusso, sappiamo che vi sono $k_n = n+1$ istanze dei dati di dimensione n. Avendo supposto che il caso pessimo si presenti nel 50% dei casi, mentre tutti gli altri casi sono tra

loro equiprobabili, concludiamo che la probabilità (peso) del caso pessimo (l'istanza $n+1$) è 0.5, mentre ogni singolo altro caso si presenta con probabilità $0.5/n$. I pesi così assegnati soddisfano alla relazione (4):

$$\sum_{i=1}^{n+1} p_i = \sum_{i=1}^{n} \frac{1}{2n} + \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1.$$

Il tempo medio d'esecuzione (in passi di calcolo) si ottiene applicando la (5):

$$\mathcal{T}^{med}(n) = \sum_{i=1}^{n+1} p_i \mathcal{T}(n,i) \tag{6}$$

$$= \sum_{i=1}^{n} \frac{1}{2n} \mathcal{T}(n,i) + \frac{1}{2} T(n,n+1) \tag{7}$$

$$= \frac{1}{2n} \sum_{i=1}^{n} (3i+1) + \frac{3n+2}{2} \tag{8}$$

$$= \frac{1}{2n} \left(n + \frac{3n(n+1)}{2} \right) + \frac{3n+2}{2} \tag{9}$$

$$= \frac{9}{4}(n+1). \tag{10}$$

Si osservi come con questa ipotesi di distribuzione dei pesi il tempo medio sia sensibilmente maggiore di quello che avevamo ottenuto in ipotesi di distribuzione uniforme.

Fissata la distribuzione dei dati (cioè i pesi p_i), la (5) può essere usata anche per determinare sperimentalmente l'effettivo tempo medio d'esecuzione: si procede alla misura dei tempi di tutte le istanze, e si calcola poi la media pesata secondo la distribuzione opportuna.

3.1.2. *Esercizi.*

- Scrivere una funzione che dato n calcoli le n^n possibili *disposizioni con ripetizione* di array di dimensione n contentente i numeri da 0 a n-1. Es. per $n = 3$ si deve creare gli array: 0,0,0
  0,0,1
  0,0,2
  0,1,0
  0,1,1
  0,1,2
  ... ecc. ...
- Determinare (e "plottare") i tempi medi sugli input generati dalla precedente funzione per la Ricerca Lineare, Insertion Sort e Bubble Sort (suggerimento: tenere n molto basso poiché n^n tende a crescere molto velocemente!). Nel caso della ricerca lineare, bisogna anche tenere conto che si deve ripetere il tutto per n volte per scegliere ogni volta un numero diverso da cercare.

- Scrivere delle funzioni per generare degli input fortemente sbilanciati e determinare i tempi medi per la Ricerca Lineare (es. il numero si trova sempre nel primo quarto, nel secondo quarto, nel terzo quarto, nel quarto quarto, oppure nel 50% delle volte non si trova mai).

3.2. **Esecuzione ripetuta di un algoritmo e preparazione dell'input.** Nelle sezioni precedenti si è visto come si possa eseguire più volte un algoritmo su un input fissato per raggiungere una determinata precisione nella misurazione dei tempi di esecuzione.

La procedura illustrata funziona correttamente nel caso della ricerca lineare di un elemento in un vettore o nel caso dell'ordinamento di un vettore già ordinato nel senso voluto (crescente o decrescente). Cosa succede, però, se l'esecuzione di un algoritmo P su un input d porta alla modifica dei dati di input (almeno nella loro disposizione)? Ad esempio, dato un array di numeri interi, l'esecuzione di un qualsiasi algoritmo di ordinamento sull'array porta alla permutazione degli elementi all'interno dell'array stesso, tranne nel caso in cui gli elementi siano già ordinati. Ciò comporta il fatto che la complessità di esecuzione dell'algoritmo possa cambiare, in quanto essa dipende dall'input stesso. Si pensi al caso di Insertion Sort: se viene dato un array di elementi ordinati in senso inverso come input, la complessità è quadratica, mentre nel caso di input con elementi ordinati nel senso voluto, la complessità è lineare.

Per sopperire a tale problema si possono progettare nuovi algoritmi che non operino *in-place*, cioè che non vadano a sovrascrivere l'input. Un metodo semplice per effettuare ciò consiste nel copiare l'input in una struttura identica a quella di input ed eseguire quindi l'algoritmo sulla copia dei dati. Ogni qualvolta rieseguiamo un dato algoritmo dobbiamo ricopiare l'input su una struttura alternativa. Questa operazione comporta, però, una maggiorazione del tempo di esecuzione che deve essere scorporata nella valutazione finale dell'algoritmo.

Possiamo vedere il tempo della copia dei dati di input come una tara da scorporare dal tempo di esecuzione dell'algoritmo vero e proprio e dalla copia dei dati (che indicheremo come tempo lordo). Nell'algoritmo (7) proponiamo dello pseudocodice per lo scorporo della tara dal lordo, dando per scontato che il numero di ripetizioni per la tara e per il lordo siano già state computate in precedenza con una variante dell'algoritmo (5). Si noti che il numero di ripetizioni della tara saranno sempre maggiori rispetto a quelle per l'esecuzione dell'algoritmo "lordo" e che quindi il numero di ripetizioni per il lordo potrebbe essere fissato, con uno spreco di tempo, pari a quello per la tara.

PREPARA è una funzione che può (ad esempio) copiare i dati, predisporre i casi migliori o peggiori, oppure generare dei numeri a caso.

3.3. **Generazione di numeri pseudo-casuali.** Le metodologie di selezione di campioni formano un capitolo assai importante della statistica. Il nostro caso, però, è particolarmente favorevole: abbiamo piena conoscenza di come è fatto l'universo delle istanze, e possiamo arbitrariamente scegliere in esso le istanze che più ci aggradano.

Algorithm 7 TempoMedioNetto(Prepara,P,d,tMin)

```
1: ripTara ←CalcolaRip(Prepara,d,tMin)
2: ripLordo ←CalcolaRip((Prepara;P),d,tMin)
3: t_0 ← getTime()
4: for i ← 1 to ripTara do
5:    e ← Prepara(d)
6: end for
7: t_1 ← getTime()
8: t_tara ← t_1 − t_0 # tempo totale di esecuzione della tara
9: t_0 ← getTime()
10: for i ← 1 to ripLordo do
11:    e ← Prepara(d)
12:    Execute(P,e)
13: end for
14: t_1 ← getTime()
15: t_lordo ← t_1 − t_0 # tempo totale di esecuzione del lordo
16: t_medio ← t_lordo/ripLordo − t_tara/ripTara # tempo medio di esecuzione
```

Per semplicità (e perché si tratta dell'unico caso che tratteremo in queste dispense) supporremo che l'universo delle istanze sia distribuito in modo uniforme: le k_n istanze di dimensione n si presentano ciascuna con probabilità $1/k_n$. Per generare un campione distribuito in modo uniforme, possiamo allora utilizzare un generatore di numeri pseudo-casuali. Si tratta di un programma che, a partire da un dato *seme* fornito come dato, genera una sequenza (determinata dal seme) di numeri razionali compresi tra 0 ed 1. La proprietà cruciale di tale sequenza è che essa "somiglia" in tutto ad una successione di numeri estratti mediante un processo davvero casuale (per esempio effettuando estrazioni con rimpiazzamento da un'urna). Qui "somiglianza" significa che eventuali test statistici effettuati sulla sequenza non sono in grado di rilevare significative regolarità nei suoi elementi, così da indicare la provenienza della successione da un algoritmo deterministico (come in effetti è un generatore di numeri pseudo-casuali). La sequenza prodotta da un generatore di numeri pseudo-casuali è in genere ciclica (cioè ad un certo punto torna a ripetersi dall'inizio), ma il periodo di tale ciclo è sufficientemente lungo da poter essere trascurato per gli scopi che ci interessano.

Costruire un buon generatore di numeri pseudo-casuali non è cosa semplice, né funzionano artifici tanto ingenui quanto diffusi come quello di usare l'orologio di macchina. Ciò che è ancora più sorprendente è che molti dei cosiddetti "generatori di numeri pseudo-casuali" messi a disposione come funzioni predefinite (col nome random, o simile) nei linguaggi di programmazione non sono buoni generatori! Semplici test statistici rilevano la non uniforme distribuzione delle sequenze da essi generati. D'altra parte sono noti in letteratura generatori semplici, con buone proprietà statistiche, che possono essere implementati facilmente in un qualsiasi linguaggio di programmazione. Uno di questi, quello che useremo nel seguito del testo, è quello presentato nell'algoritmo (8).

Algorithm 8 RANDOM(seed)

```
1: a ← 16807
2: m ← 2147483647
3: q ← 127773
4: r ← 2836
5: hi ← trunc(seed/q)
6: lo ← seed − q · hi
7: test ← a · lo − r · hi
8: if (test < 0) then
9:     seed ← test + m
10: else
11:    seed ← test
12: end if
13: return seed/m
```

La funzione Random è definita in un ambiente globale in cui è dichiarata la variabile reale (in doppia precisione) seed, il seme del generatore. Dopo aver inizializzato il seme, chiamate successive a random generano una successione di numeri pseudo-casuali compresi tra 0 e 1. Le proprietà di buona distribuzione della successione non dipendono dal seme scelto, ma solo dalle costanti numeriche definite nella funzione. Non è una buona idea cambiare tali valori. Tale funzione è disponibile in Java (con un esempio di utilizzo) sulla pagina del corso.

3.3.1. *Esercizi.*

- Si calcoli il tempo di esecuzione di Insertion Sort e di Bubble Sort su istanze generate casualmente, con un errore massimo del 5% per vettori di dimensione 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, ...
- Si calcoli il tempo di esecuzione della Ricerca Lineare su istanze generate casualmente, con un errore massimo del 5% per vettori di dimensione 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, ... e al variare della probabilità (10%, 30%, 50%, 70%, 90%) che il numero cercato sia all'interno del vettore.
- Si ripetano gli esperimenti dei due esercizi più volte. Cosa succede ai tempi ottenuti?

3.4. **Stima del tempo medio.** Abbiamo visto nei paragrafi precedenti che, per determinare il tempo medio e fissata una distribuzione dei dati, occorre procedere al calcolo (o alla misura) dei tempi necessari all'esecuzione di ogni singola istanza.

Se siamo interessanti ad un'analisi del comportamento asintotico del caso medio, dovremo manipolare algebricamente la somma ottenuta mediante la (5), alla ricerca, se possibile, di una sua formulazione "chiusa" (cioè che non contenga sommatorie) e che dia pertanto immediata informazione sull'ordine della funzione. Si tratta di uno studio assai importante e significativo, ma che talvolta, in specie in presenza di distribuzioni non uniformi, si presenta come molto difficile, quando non impossibile.

Se siamo interessati alla valutazione sperimentale, dovremo misurare il tempo necessario ad ogni singola istanza. Vi sono casi, tuttavia, in cui ciò non si può (o

non è conveniente) fare. Prendiamo in considerazione il caso di un algoritmo di ordinamento. Abbiamo già osservato come una buona nozione di dimensione dei dati sia in questo caso il numero, n, dei dati da ordinare. Il numero di possibili istanze di dimensione n con tutti i numeri presenti è $k_n = n^n$, una funzione che cresce molto velocemente al crescere di n. Cresce al tal punto che, per valori di n di poche decine, il tempo necessario alla misura sperimentale del tempo su tutte le possibili istanze diviene irrealisticamente lungo e, nei fatti, assolutamente fuori portata. Scartata dunque l'ipotesi di misurare tutte le istanze, se si vuole mantenere la scelta sperimentale e non analitica, non resta che ricorrere ad una *stima approssimata* del tempo medio d'esecuzione.

Possiamo ottenere una stima del tempo medio, selezionando un opportuno *campione* tra tutte le istanze dei dati, e calcolando poi la media su tale campione e non su tutto l'universo delle istanze. È chiaro che il valore così determinato costituisce un'approssimazione del valore reale, la cui bontà dipende, in sostanza, da due fattori:

- la buona distribuzione: il campione deve essere scelto in modo da rispecchiare la distribuzione dell'universo;
- la dimensione: pur essendo assai più piccolo dell'intero universo, il campione deve essere sufficientemente ampio da rappresentare adeguatamente l'universo stesso.

In una parola, il campione deve essere *rappresentativo* dell'universo da cui è stato estratto.

Ricapitolando, vogliamo stimare la media di un certo parametro X (nel nostro caso, il tempo di calcolo di un programma assegnato), al variare della dimensione dei dati, in cui l'universo dei dati di dimensione n è costituito da k_n istanze diverse. Sappiamo che la media è data da

$$E[X(n)] = \sum_{i=1}^{k_n} p_i X(n,i) \tag{11}$$

Scegliamo dunque un valore per n e generiamo un campione C, di cardinalità $c_n << k_n$, di istanze di dimensione n. Per ciascuna istanza i in C misuriamo il parametro $X(n,i)$ e calcoliamo la media di tali valori:

$$E'[X(n)] = \frac{1}{c_n} \sum_{i=1}^{c_n} X(n,i) \tag{12}$$

Un risultato fondamentale della statistica matematica ci assicura che se il campione C è distribuito nello stesso modo dell'universo da cui è estratto, allora $E'[X]$ è una ragionevole stima della media $E[X(n)]$, la cui bontà dipende dalla cardinalità del campione stesso. In altri termini, al crescere di c_n, $E'[X]$ è garantito approssimare sempre meglio il valore della media (ma la "velocità" di tale approssimazione è proporzionale alla radice quadrata di c_n: quadruplicando c_n si raddoppia soltanto la bontà della stima).

Due problemi si pongono immediatamente: come determinare (generare) C con la stessa distribuzione dell'universo? E poi, una volta calcolato $E'[X]$ per un certo campione, è possibile avere qualche informazione su quanto tale valore si discosta dal valore reale della media $E[X(n)]$? La risposta alla prima domanda sta nell'utilizzo di una buona funzione random, la seconda risposta sta nella prossima sezione.

3.5. **Intervalli di confidenza.** Abbiamo visto come, con l'ausilio di un processo di generazione pseudo-casuale, possiamo determinare un certo valore $E'[X(n)]$, che approssima il valore reale $E[X(n)]$. Sappiamo anche che, al crescere della dimensione del campione, $E'[X(n)]$ approssima sempre meglio $E[X(n)]$.

Vogliamo ora cercare di quantificare questa approssimazione, in termini probabilistici. È evidente che, ripetendo con altri valori l'esperimento che ci ha fornito la stima $\overline{X_1} = E'[X(n)]$, otterremo un diverso valore della media campionaria, diciamo $\overline{X_2}$. Ripetendo l'esperimento un gran numero di volte, otterremo sempre valori diversi (una successione $\overline{X_i}$) che tuttavia avranno grande probabilità di non discostarsi troppo (se i campioni sono sufficientemente grandi) dal valore reale della media $E[X(n)]$. Si può sempre presentare il caso in cui un campione generato casualmente è particolarmente "cattivo", dando luogo ad un valore della media campionaria molto distante dalla media reale, ma, se i campioni sono generati secondo la stessa distribuzione dell'universo, tale evento sarà molto raro. Vi sono dunque in gioco due parametri:

(1) un valore Δ (per ora ignoto) per il quale gran parte degli $\overline{X_i}$ distano da $E[X(n)]$ meno di Δ;
(2) la percentuale α di esperimenti "cattivi", cioè che hanno probabilità di non rientrare nella distanza Δ da $E[X(n)]$.

La statistica matematica fornisce un metodo per collegare tra loro (e calcolare) tali parametri. Fissato un valore per α (usualmente piccolo, con 0.05, 0.1, 0.01 e 0.001 in ordine decrescente di popolarità) daremo un modo per calcolare, fissato un campione sufficientemente grande, la media campionaria $E'[X(n)]$ ed un certo valore per Δ. L'intervallo così determinato

$$[E'[X(n)] - \Delta, E'[X(n)] + \Delta] \tag{13}$$

si dice *intervallo di confidenza* all'$(1-\alpha)$ ed il suo significato è il seguente: se ripetessimo l'esperimento un gran numero di volte, e per ognuna di esse calcolassimo un intervallo di confidenza, approssimativamente solo $\alpha \cdot 100$ intervalli su cento non conterrebbero il vero valore della media. Pur con qualche imprecisione, possiamo dunque dire che un intervallo di confidenza ci fornisce non solo una stima della media (data da $E'[X(n)]$), ma anche di quanto tale stima può allontanarsi dalla media reale $E[X(n)]$.

Come calcolare dunque il parametro Δ? Ci accontenteremo qui di riportare una formula, rimandando la sua giustificazione al corso di statistica[4]. Il calcolo fa uso di un ulteriore parametro del campione, la *varianza campionaria*, definita come

$$s(n)^2 = \frac{1}{c_n} \sum_{i=1}^{c_n} (X(n,i) - E'[X(n)])^2 \tag{14}$$

dove, si ricordi, c_n è la cardinalità del campione (costituito a sua volta da istanze del problema in cui i dati hanno dimensione n). Fissato il *coefficiente di confidenza* $1-\alpha$ (per esempio, il 95%, o il 90%, o il 99%), se il campione è abbastanza grande (diciamo di dimensione circa 100), si ha

$$\Delta = \frac{1}{\sqrt{c_n}} \cdot z(1-\frac{\alpha}{2}) \cdot s(n) \tag{15}$$

Il nuovo parametro $z(1-\frac{\alpha}{2})$ che compare nella formula precedente si riferisce ad una funzione molto importante nel calcolo delle probabilità che è nota come funzione di distribuzione normale. Alcuni suoi valori, al variare di α, sono riportati nella tabella (1) e sono comunque disponibili in molti testi di statistica.

α	$z(1-\frac{\alpha}{2})$
0.02	2.33
0.05	1.96
0.10	1.64

TABELLA 1. I più utilizzati tra i valori di $z(1-\frac{\alpha}{2})$.

Concludiamo questo paragrafo con un'osservazione di programmazione. Il calcolo della varianza campionaria usando la Definizione (14) richiede la memorizzazione di tutti i c_n valori $X(n,i)$ e, dunque, una discreta quantità di memoria. Con facili calcoli, sviluppando il quadrato ed usando la definizione della media $E'[X(n)]$, si osserva che

$$\begin{aligned} s(n)^2 =& \frac{1}{c_n} \sum_{i=1}^{c_n} \left(X(n,i)^2 - 2X(n,i)E'[X(n)] + E'[X(n)]^2 \right) \\ s(n)^2 =& \frac{1}{c_n} \left(\sum_{i=1}^{c_n} X(n,i)^2 - 2E'[X(n)] \sum_{i=1}^{c_n} X(n,i) + c_n E'[X(n)]^2 \right) \end{aligned}$$

da cui, ricodando che $E'[X(n)] = \sum_{i=1}^{c_n} X(n,i)$, abbiamo

$$s(n)^2 = \frac{1}{c_n} \sum_{i=1}^{c_n} X(n,i)^2 - E'[X(n)]^2$$

[4]Si veda, ad esempio, `http://www.mv.helsinki.fi/home/jmisotal/BoS.pdf` o Johnson, R.A. and Bhattacharyya, G.K., *Statistics: Principles and Methods*, 2nd Edition. Wiley, 1992.

Con questa nuova espressione, non è più necessario mantenere tutti i distinti $X(n,i)$; la memorizzazione delle due somme parziali è sufficiente per il calcolo sia della media $E'[X(n)]$, sia della varianza.

Un algoritmo per la determinazione del tempo medio con un errore minore di Δ è descritto in pseudo codice come algoritmo (9) qui di seguito.

Algorithm 9 MISURAZIONE(C,P,d,c,za,tMin,Δ)

```
1:  t ← 0
2:  sum2 ← 0
3:  cn ← 0
4:  repeat
5:    for i ← 1 to c do
6:      m ← TempoMedioNetto(C, P, d, tMin)
7:      t ← t + m
8:      sum2 ← sum2 + m^2
9:    end for
10:   cn ← cn + c
11:   e ← t/cn
12:   s ← sqrt(sum2/cn − e^2)
13:   delta ← (1/sqrt(cn)) ∗ za ∗ s;
14: until delta < Δ
15: return (e, delta)
```

Da notare che per $c = 1$ l'algoritmo termina subito poiché su una sola misurazione non può essere calcolata la varianza. Bisogna quindi usare c maggiore di 1 (ad esempio 5 o 10).

L'algoritmo risulta efficiente se si conosce a priori (a grandi linee) il tempo medio. Nei casi in cui si devono misurare tempi molti piccoli o tempi molto grandi, avere un valore inadeguatamente fissato per Δ può portare a errori di misurazione, nel primo caso, o a fare troppe iterazioni senza guadagnare molto in precisione, nel secondo caso.

3.5.1. *Esercizi.*

- Implementare l'algoritmo appena descritto e misurare i tempi per Ricerca Lineare, Insertion Sort e Bubble Sort (come nei precedenti esercizi) in modo che il Δ misurato sia inferiore a 0.2 secondi con $alpha = 0.05$ (quindi za sarà impostato pari a . . . ?).
- Modificare l'algoritmo in modo che $delta$ non sia un valore assoluto ma sia pari a 1/10 del tempo medio (ad es. se $e = 1$ allora $delta = 0.1$, se $e = 100$ allora $delta = 10$) e misurare di nuovo i tempi.
- Fate i grafici dei tempi ottenuti e confrontateli con i risultati che avete ottenuto nelle scorse lezioni.