Indice

I Algoritmi ricorsivi e analisi della loro complessità . . . **5**

I.1 L'approccio dividi-et-impera . . . 5

I.1.1 L'algoritmo di merge-sort . . . 5

I.1.2 Analisi dell'albero di ricorsione . . . 7

I.1.3 Il metodo di sostituzione . . . 7

I.1.4 Master theorem . . . 8

I.2 Progetto di algoritmi ricorsivi . . . 10

I.2.1 Pronostico sportivo (confronto di ordinamenti) . . . 10

I.2.2 Moltiplicazione di interi a n cifre . . . 11

I.2.3 Minima distanza tra punti del piano . . . 13

II Randomized algorithms and performance analysis . . . **17**

II.1 Worst case and average case complexity . . . 17

II.1.1 QUICK-SORT algorithm . . . 17

II.2 Randomized algorithms: basic notation and tools . . . 20

II.2.1 Uniformity of the input: randomization . . . 20

II.2.2 Indicator Random Variables . . . 23

II.3 Las Vegas algorithms . . . 25

II.3.1 Average cost of the hiring problem . . . 25

II.3.2 Randomized Quick-Sort sorting algorithm . . . 25

II.3.3 Finding the median . . . 27

II.4 Montecarlo algorithms . . . 32

II.4.1 Matrix multiplication test . . . 32

II.4.2 Minimum cut of a graph . . . 34

II.4.3 Polynomial identity test . . . 36

II.4.4 Determinant of a matrix . . . 38

II.4.5 Fingerprint of a file . . . 40

II.4.6 Application to Pattern Matching . . . 41

II.5 The probabilistic method . . . 43

II.6 3-SAT problem . . . 44

II.6.1 Randomized algorithm for MAX-3-SAT . . . 44

II.7 Cache memory management . . . 46

II.7.1 Marking algorithms . . . 46

II.7.2 A randomized marking algorithm . . . 48

II.8 The "engagement ring" problem . . . 50

III Tecniche di valutazione di complessità ammortizzata **53**

Indice degli algoritmi

I.1 MERGE-SORT(A,p,r) . . . 5
I.2 MERGE(A,p,q,r) . . . 6
I.3 SORT-AND-COUNT(A,p,r) . . . 11
I.4 MERGE-COUNT(A,p,q,r) . . . 12
I.5 MULTIPLY(x,y) . . . 13
I.6 COLSEST-PAIR(P) . . . 15
I.7 COLSEST-PAIR-REC(P_x, P_y) . . . 15
II.1 QUICK-SORT(A,p,r) . . . 18
II.2 PARTITION(A,p,r) . . . 18
II.3 PERMUTE-BY-SORTING(A) . . . 21
II.4 RANDOMIZE-IN-PLACE(A) . . . 22
II.5 RAND-QUICK-SORT(A,p,r) . . . 25
II.6 RAND-PARTITION(A,p,r) . . . 26
II.7 RANDOMIZED-SELECT(A, p, r, i) . . . 28
II.8 DETERMINISTIC-SELECT(A,i) . . . 31
II.9 MATRIX-MUL-TEST(A,B,C) . . . 33
II.10 RAND-CONTRACT(G) . . . 35
II.11 REC-RAND-CONTRACT(G) . . . 36
II.12 POLY-TEST(Q,R) . . . 37
II.13 RAND-PERFECT-MATCHING(A,p,r) . . . 39
II.14 FILE-EQUALITY-ON-NETWORK() . . . 40
II.15 STRAIGHTFORWARD-PATTERN-MATCHING(x,y) . . . 41
II.16 KARP-RABIN(x,y) . . . 42
II.17 MARKING ALGORITHM(σ) . . . 47

Capitolo I

Algoritmi ricorsivi e analisi della loro complessità

I.1 L'approccio dividi-et-impera

Il concetto alla base della progettazione di numerosi algoritmi ricorsivi è il seguente. Si suddivide il problema principale in sottoproblemi di dimensioni minori, fino a che la loro soluzione possa essere considerata elementare; successivamente, si ricompone il problema principale a partire dalle soluzioni parziali.

I.1.1 L'algoritmo di merge-sort

Un esempio di approccio *divide et impera* al problema dell'ordinamento di dati è dato dall'algoritmo di MERGE-SORT. Merge sort opera nel modo seguente:

1. Suddividi la sequenza di n elementi in due sottosequenze di lunghezza $\frac{n}{2}$;
2. Ordina le due sottosequenze applicando ricorsivamente l'algoritmo di MERGE-SORT;
3. Fondi le due sottosequenze ordinate in un'unica sequenza ordinata.

In pratica, l'algoritmo non fa che suddividere il vettore di dati iniziali in due fino a che non si ottengano vettori di lunghezza unitaria, mentre deroga il lavoro di ordinamento alla fase di ricombinazione (MERGE).

Algoritmo I.1 MERGE-SORT(A,p,r)

```
1: if p < r then
2:     q ← ⌊(p+r)/2⌋
3:     MERGE-SORT(A, p, q)
4:     MERGE-SORT(A, q + 1, r)
5:     MERGE(A, p, q, r)
```

La procedura MERGE si occupa della ricombinazione (con ordinamento) dei sottovettori creati dalle chiamate ricorsiva a MERGE-SORT. Per fare ciò si utilizzano due array di supporto, L ed R, in cui vengono copiati i due sottovettori creati dall'ultima chiamata ricorsiva. Si noti come, al termine di ciascun vettore di supporto, si pone un valore simbolico ∞, nella pratica un valore relativamente grande, per semplificare la casistica dei confronti. Infatti in questo modo in L e in R rimarrà sempre almeno un elemento e non dovremo preoccuparci di controllare che gli indici utilizzati per scandire i vettori vadano fuori dal dominio di definizione.

Algoritmo I.2 MERGE(A,p,q,r)

1: $n_1 \leftarrow q - p + 1$
2: $n_2 \leftarrow r - q$
3: CREA$(L[1 \ldots n_1 + 1]), (R[1 \ldots n_2 + 1])$
4: $L[1 \ldots n_1] \leftarrow A[p \ldots q], R[1 \ldots n_2] \leftarrow A[q + 1 \ldots r]$
5: $L[n_1 + 1] \leftarrow \infty, R[n_2 + 1] \leftarrow \infty$
6: $i \leftarrow 1$, $j \leftarrow 1$
7: **for** $k \leftarrow p \ldots r$ **do**
8:   **if** $L[i] \leq R[j]$ **then**
9:     $A[k] \leftarrow L[i]; i \leftarrow i + 1$
10:   **else**
11:     $A[k] \leftarrow R[j]; j \leftarrow j + 1$

Esempio. Sia dato il vettore

$$\begin{matrix} [& 1 & 6 & 7 & 9 & 2 & 4 & 5 & 8 &] \\ & p & & & q & & & & r & \end{matrix}$$

dove sono indicati indice iniziale, finale e intermedio. I passi dell'algoritmo, che innanzitutto suddivide il vettore in 8 blocchi unitari, sono i seguenti:

[1 6 7 9 2 4 5 8]

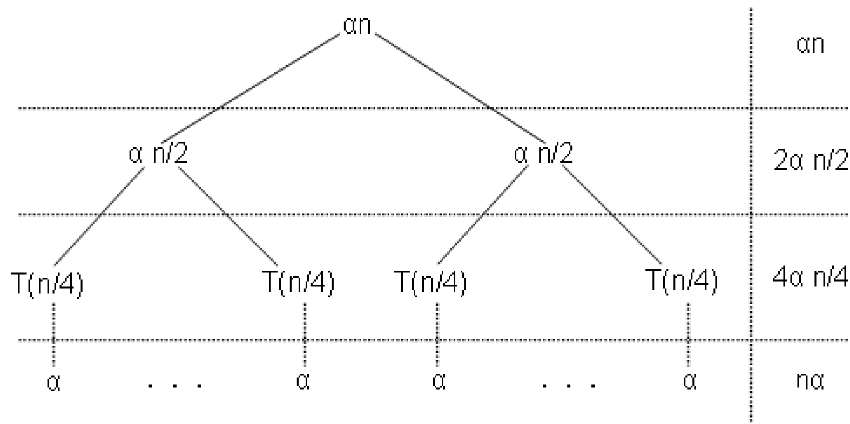
[1 6 7 9] [2 4 5 8]

[1 6] [7 9] [2 4] [5 8]

[1] [6] [7] [9] [2] 4] [5] [8]

[1 6 7 9] [2 4 5 8]

[1 2 4 5 6 7 8 9]



Complessità. Nel caso di vettore di input unitario, la complessità temporale è, banalmente, $O(1)$. Altrimenti, si tratta di scomporre il problema di cardinalità n in due sottoproblemi di cardinalità $n/2$, aggiungendo il costo di ricombinazione, che è lineare.

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

La formula trovata è però ricorsiva. Per aver maggiori indicazioni sulla reale complessità dell'algoritmo occorre trasformarla in una formula chiusa.

I.1.2 Analisi dell'albero di ricorsione

Data una formula ricorsiva del tipo $aT(n/b) + \alpha n$, è possibile ottenere la formula chiusa analizzando l'albero generato dalle chiamate ricorsive dell'algoritmo. Si sviluppano i primi livelli dell'albero, si individua la regola di generazione dei livelli successivi, quindi si sommano i costi per livello. Nel caso di merge-sort, $2T(n/2) + \alpha n$, si sviluppa nel modo seguente (si assume che n sia potenza esatta di 2).

Ad ogni livello, si raddoppia il numero di nodi, ciascuno con costo dimezzato rispetto al nodo genitore. All'ultimo livello, si avranno n foglie (tale è la cardinalità del problema iniziale) con costo α: il numero di livelli totale sarà quindi $\lg n$, e la complessità $\Theta(n \lg n)$.

I.1.3 Il metodo di sostituzione

Un metodo più rigoroso per l'analisi della complessità prevede l'utilizzo del principio di induzione. L'utilizzo di tale tecnica però ha un costo, ed è quello di dover conoscere preventivamente la struttura della formula chiusa che vogliamo dimostrare equivalente alla ricorrenza data. Utilizzando sempre il caso del MERGE-SORT, dimostriamo per induzione che $T(n) \leq cn \lg n$, con $c > 0$. Per questa dimostrazione, verrà assunto che la cardinalità n sia potenza esatta di 2; in caso contrario andrebbe introdotta la notazione di intero inferiore e il procedimento si complicherebbe, pur rimanendo, nella sostanza, identico.

Base induttiva. per $n = 2$ abbiamo $T(2) \leq 2c$. Passo induttivo. La diseguaglianza valga per $m < n$. Allora

$$T(n) = 2T(\frac{n}{2}) + cn \leq 2\frac{cn}{2} \cdot \lg\frac{n}{2} + 2\frac{n}{2} = cn(\lg n - 1) + n = cn\lg n - cn + n$$

per $c \geq 1$ abbiamo che $T(n) \leq cn\lg n$.

I.1.4 Master theorem

Utilizzando il *master theorem*, si può risalire alla forma chiusa di numerose ricorrenze della forma
(I.1)

$$T\left(n\right) = aT\left(\frac{n}{b}\right) + f\left(n\right)$$

purché le costanti a e b e la funzione f soddisfino alcune proprietà.

Teorema I.1. Master Theorem *Siano $a \leq 1$, $b > 1$ due costanti, $f(n)$ una funzione e $T(n)$ una misura di complessità definita su interi nonnegativi dalla ricorrenza (I.1), allora $T(n)$ ha limite asintotico secondo quanto segue:*

1. *se $f(n) = O(n^{\log_b a-\varepsilon})$ per $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$;*
2. *se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a}\lg n)$;*
3. *se $f(n) = \Omega(n^{\log_b a+\varepsilon})$ per $\varepsilon > 0$, e se $af(\frac{n}{b}) \leq cf(n)$, per $c < 1$ ed n sufficientemente grande, allora $T(n) = \Theta(f(n))$.* □

Nell'enunciato abbiamo sottinteso che $n = b^k$ per qualche intero $k > 0$. Nel caso in cui n non sia una potenza di b il risultato può venire esteso a costo di qualche piccola complicazione nella dimostrazione. Per semplicità ci concentriamo nel caso più semplice.

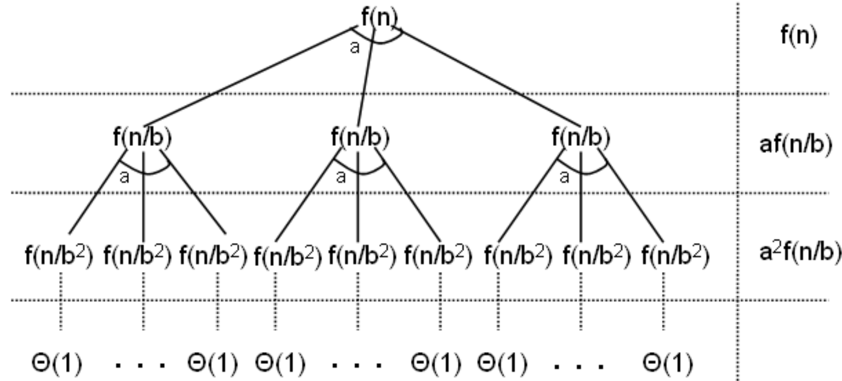
L'interpretazione dei tre casi del Master Theorem deriva direttamente dal confronto tra il costo della ricombinazione dei risultati (funzione $f(n)$) e il costo della soluzione dei singoli sottoproblemi (componente $n^{\log_b a}$). Se $f(n)$ è polinomialmente minore di $n^{\log_b a}$, allora il suo contributo è asintoticamente ininfluente. In caso invece fosse polinomialmente maggiore, esso domina la complessità. Mentre in caso di sostanziale "pareggio" interviene un fattore aggiuntivo $\lg n$ dovuto ai livelli dell'albero di ricorsione.

Traccia di dimostrazione. Analizziamo l'albero di ricorsione generato da $T(n)$:

errore nella figura': terzo livello: $z^2 f(n/b^2)$

Esso ha $\log_b n+1$ livelli; l'ultimo livello ha costo $\Theta(n^{\log_b a})$ essendo l'ordine del numero di foglie, mentre i livelli superiori hanno in totale costo: $g(n) = \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right)$. Quindi $T(n) = \Theta(n^{\log_b a}) + g(n)$. Studiamo $g(n)$ nei tre casi previsti dall'enunciato del teorema.

1. $f\left(\frac{n}{b^j}\right) = O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right)$ per $\varepsilon > 0$.
   In tal caso, sostituendo nella sommatoria abbiamo,



$$
\begin{aligned}
g(n) &= O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\tfrac{n}{b^j}\right)^{\log_b a-\varepsilon}\right) = \\
&= O\left(n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(\tfrac{ab^\varepsilon}{b^{\log_b a}}\right)^j\right) = \\
&= O\left(n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} (b^\varepsilon)^j\right) = \\
&= O\left(n^{\log_b a-\varepsilon} \tfrac{b^{\varepsilon \log_b n}-1}{b^\varepsilon-1}\right) = \\
&= O\left(n^{\log_b a-\varepsilon} \tfrac{n^\varepsilon-1}{b^\varepsilon-1}\right)
\end{aligned}
$$

avendo utilizzato lo sviluppo delle serie geometriche. Siccome b e ε sono costanti, possiamo dire che

$$g(n) = O\left(n^{\log_b a-\varepsilon} n^\varepsilon\right)$$

e quindi, essendo il costo dell'ultimo livello $\Theta(n^{\log_b a})$, si ha che la complessità totale è $\Theta(n^{\log_b a})$.

2. $f(n) = \Theta\left(n^{\log_b a}\right)$.
Allora, sostituendo nella sommatoria abbiamo:

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

In questo caso l'espressione di $g(n)$ risulta più semplice che in precedenza:

$$
\begin{aligned}
g(n) &= \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\tfrac{a}{b^{\log_b a}}\right)^j\right) = \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1\right) = \\
&= \Theta\left(n^{\log_b a} \log_b n\right) = \Theta\left(n^{\log_b a} \lg n\right)
\end{aligned}
$$

3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ per $\varepsilon > 0$.
   Notiamo innanzitutto che $g(n) = \Omega\left(f(n)\right)$, perché $g(n)$ contiene nella sua definizione $f(n)$ e nella somma tutti i termini sono non negativi. Avendo assunto che $af\left(n/b\right) \leq cf\left(n\right)$ per $c < 1$ ed $n > b$, si ottiene $a^j f\left(\frac{n}{b^j}\right) \leq c^j f\left(n\right)$. Andando a sostituire nella sommatoria, possiamo maggiorare $g(n)$:
   $$\sum_{j=0}^{\log_b n-1} a^j f\left(\tfrac{n}{b^j}\right) \leq \sum_{j=0}^{\log_b n-1} c^j f\left(n\right)$$
   $$\leq f\left(n\right) \sum_{j=0}^{\infty} c^j = f\left(n\right)\left(\tfrac{1}{1-c}\right)$$
   $$= \mathrm{O}\left(f\left(n\right)\right)$$
   E poiché $g\left(n\right) = \Omega\left(f(n)\right)$, abbiamo la tesi.

Applichiamo ora il master theorem al caso di MERGE-SORT. Tralasciando il caso base, $T(1) = c$, consideriamo la ricorrenza $T(n) = 2T\left(n/2\right) + cn$. Utilizzando gli stessi nomi delle costanti utilizzate nell'enunciato del master theorem, $a = 2$ e $b = 2$, mentre $f(n)$ è, ovviamente, $\Theta(n)$. Allora, rientriamo nel secondo caso dell'enunciato, e pertanto $T(n) = \Theta\left(n^{\log_2 2} \lg n\right) = \Theta(n \lg n)$. La notazione $n^{\log_b a\pm\varepsilon}$ indica che, perché valga il teorema nei casi 1) e 3), $f(n)$ deve essere polinomialmente minore (maggiore) di $n^{\log_b a}$. Si consideri la ricorrenza $T\left(n\right) = 2T\left(n/2\right) + n \lg n$. Poiché per n grande, $n < n \lg n < n \cdot n^{\varepsilon}$, ci troviamo in un caso intermedio tra il 2) ed il 3), e non possiamo applicare il teorema.

I.2 Progetto di algoritmi ricorsivi

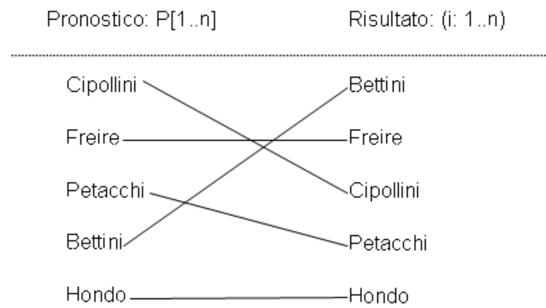
Verranno ora presentati tre problemi, risolti con i metodi di progetto degli algoritmi ricorsivi, ed in particolare con un approccio divide et impera.

I.2.1 Pronostico sportivo (confronto di ordinamenti)

Vengono raccolti da una società di scommesse pronostici relativi alla classifica finale della Milano-Sanremo (celebre competizione ciclistica primaverile). Il pronostico consiste di una classifica (vettore ordinato) di n nominativi, secondo la scelta dello scommettitore. Compito della società è, a gara svolta, assegnare un punteggio ad ogni scommettitore in base a quanto il pronostico si avvicina al risultato effettivo. Per semplicità, si assume che il numero dei partecipanti alla corsa ciclistica sia uguale al numero di posizioni nel pronostico.

Soluzione. Il fatto che il numero di ciclisti sia equivalente al numero di posizioni in classifica rende possibile assegnare un punteggio agli scommettitori in base al numero di posizioni scambiate rispetto alla classifica originale:

Per attribuire un punteggio che cresce all'aumentare delle discrepanze tra il pronostico $P[i]$ e la classifica i, possiamo assegnare una penalità ad ogni "incrocio" fra i due ordinamenti, cioè se $i > j$ ma $P[i] < P[j]$. Il problema è chiaramente polinomiale. Infatti un algoritmo rudimentale enumera tutte le coppie i, j, con $i < j$, e verifica se $P[i] < P[j]$, in caso contrario incrementa il contatore di penalità di uno. L'algoritmo ha complessità $O(n^2)$. Chiaramente se vogliamo cercare di migliorare la complessità dobbiamo trovare



un meccanismo che permetta di incrementare il contatore delle penalità di più di unità alla volta. Utilizzando la strategia divide-et-impera suddividiamo il vettore in due sezioni e contiamo gli incroci all'interno di ognuno di essi; successivamente ricomponiamo i due sottovettori considerando gli incroci che vi sono tra elementi di un sottovettore e elementi dell'altro. A tal fine possiamo utilizzare un algoritmo analogo a MERGE-SORT. Non è strettamente necessario riordinare il vettore, ma per facilitare la comprensione procediamo all'ordinamento intanto che conteggiamo le penalità.

Algoritmo I.3 SORT-AND-COUNT(A,p,r)

```
1: if p < r then
2:    q ← ⌊(p+r)/2⌋
3:    c1 ← SORT-AND-COUNT(A, p, q)
4:    c2 ← SORT-AND-COUNT(A, q + 1, r)
5:    c3 ← MERGE-COUNT(A, p, q, r)
6:    return c1 + c2 + c3
```

La funzione MERGE-COUNT è simile alla funzione MERGE vista in precedenza, con l'aggiunta di un contatore di scambi. Il contatore viene incrementato di un numero pari al numero di elementi ancora presenti in L ogni volta che un elemento di R viene copiato in A.

Esempio. Sia dato il pronostico di 8 posizioni:

$$[\ 1\ \ 4\ \ 3\ \ 6\ \ 2\ \ 5\ \ 8\ \ 7\ ]$$

dove i numeri che compongono la sequenza indicano la posizione nella classifica reale. Applicando l'algoritmo di cui sopra, si ha:

Quindi, il pronostico considerato ottiene 6 inversioni. La complessità dell'algoritmo, data la sostanziale equivalenza di questo con *merge-sort* è $O(n \lg n)$.

I.2.2 Moltiplicazione di interi a n cifre

L'algoritmo tradizionale per la moltiplicazione in colonna di numeri con n cifre ha un costo computazionale che è $O(n^2)$. Dobbiamo infatti moltiplicare ogni coppia di cifre dei

Algoritmo I.4 MERGE-COUNT(A,p,q,r)

1: $n_1 \leftarrow q - p + 1$
2: $n_2 \leftarrow r - q$
3: $c \leftarrow 0$
4: CREA$(L[1 \dots n_1 + 1]), R[1 \dots n_2 + 1])$
5: $L[1 \dots n_1] \leftarrow A[p \dots q], R[1 \dots n_2] \leftarrow A[q + 1 \dots r]$
6: $L[n_1 + 1] \leftarrow \infty; R[n_2 + 1] \leftarrow \infty$
7: $i \leftarrow 1; j \leftarrow 1$
8: **for** $k \leftarrow p \dots r$ **do**
9: **if** $L[i] \leq R[j]$ **then**
10: $A[k] \leftarrow L[i]; i \leftarrow i + 1$
11: **else**
12: $A[k] \leftarrow R[j]; j \leftarrow j + 1$
13: $c \leftarrow c + (n_1 + 1 - i)$
14: **return** c

Fase di decomposizione.

[1 4 3 6 2 5 8 7]

[1 4 3 6] [2 5 8 7]

[1 4] [3 6] [2 5] [8 7]

[1] [4] [3] [6] [2] [5] [8] [7]

Fase di "merge".

[1 4] **0**; [3 6] **0**; [2 5] **0**; [$\underline{7}$ 8] **1**;

[1 $\underline{3}$ 4 6] **1**; [2 5 7 8] **0**;

[1 $\underline{2}$ 3 4 $\underline{5}$ 6 7 8] **3 + 1**;

Figura I.1: esempio di funzionamento di MERGE-COUNT: in grassetto gli incrementi del contatore di inversioni. Vengono sottolineate le inversioni

due numeri. Anche in questo caso è possibile, con l'approccio divide-et-impera, ridurre la complessità. Siano x e y due numeri di n cifre decimali da moltiplicare. Li scomponiamo allora in due parti:
$x = 10^{n/2}x_1 + x_2$
$y = 10^{n/2}y_1 + y_2$
In tal modo il prodotto xy può essere scritto come:

(I.2)
$$xy = \left(10^{n/2}x_1 + x_2\right)\left(10^{n/2}y_1 + y_2\right) = 10^n x_1 y_1 + 10^{n/2}\left(x_1 y_2 + x_2 y_1\right) + x_2 y_2$$

In realtà, il guadagno in complessità finora ottenuto è nullo: applicando il master theorem alla formula ricorsiva che dà la complessitaà del prodotto $T(n) = 4T(n/2) + cn$, che esprime la complessità del prodotto xy (I.2 (quattro moltiplicazioni di numeri a $n/2$ cifre, più il costo della somma), otteniamo ancora $O(n^2)$. È però possibile eliminare una moltiplicazione dalla soluzione, osservando che:

$$(x_1 + x_2)(y_1 + y_2) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2$$

per cui
$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - (x_1 y_1 + x_2 y_2)$$

Ecco quindi lo pseudocodice per l'algoritmo. Nella descrizione utilizziamo la notazione binaria.

Algoritmo I.5 MULTIPLY(x,y)

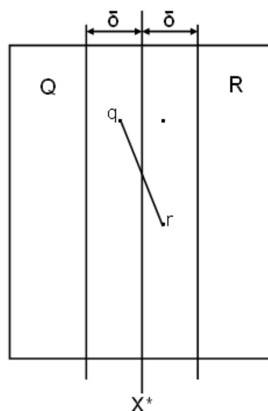
1: $x = x_1 2^{n/2} + x_2$
2: $y = y_1 2^{n/2} + y_2$
3: $z_1 \leftarrow x_1 + x_2$
4: $z_2 \leftarrow y_1 + y_2$
5: $z_3 \leftarrow$ MULTIPLY(z_1, z_2)
6: $z_4 \leftarrow$ MULTIPLY(x_1, y_1)
7: $z_5 \leftarrow$ MULTIPLY(x_2, y_2)
8: **return** $z_4 \cdot 2^n + (z_3 - z_4 - z_5) \cdot 2^{n/2} + z_5)$

Avendo eliminato una moltiplicazione, la formula ricorsiva per la complessità temporale diventa $T(n) = 3T(n/2) + cn$ e, applicando il *master theorem*, otteniamo una complessità $O\left(n^{\log_b a}\right) = O\left(n^{\log_2 3}\right) = O\left(n^{1.59\ldots}\right)$.

I.2.3 Minima distanza tra punti del piano

Dato un insieme P di n punti nel piano, essendo ogni punto p specificato dalle coordinate cartesiane (p_x, p_y), consideriamo il problema di individuare la coppia di punti a distanza minima, dove per distanza intendiamo la distanza euclidea denotata dalla funzione $d(p, q)$. Anche in questo caso il problema è polinomiale infatti un algoritmo banale considera tutte le coppie di punti, ne calcola la distanza e tiene traccia del valore minimo, dando luogo a una complessità $\Theta(n^2)$. Una possibile strategia per ridurre la complessità è la seguente:

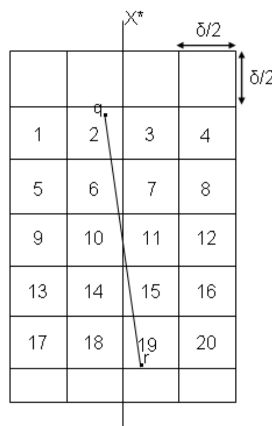
1. si suddivide ricorsivamente P in due sottoinsiemi, Q ed R, rispetto ad un valore di ascissa x^*;



2. si calcolano le minime distanze fra i punti di Q ed R, e si indica con δ il minimo fra le due;

3. si considera una striscia di larghezza 2δ centrata in x^*, e la si partiziona in quadrati di lato $\delta/2$: si garantisce così che in ogni quadrato sia contenuto al più un punto;

4. si confrontano le distanze fra punti all'interno della striscia, numerando progressivamente i quadrati per riga e per colonna. Si osserva che utilizzando tale numerazione non possono esistere punti a distanza minore di δ che giacciano nello stesso quadrato e in quadrati a più di 10 posizioni di distanza: per come è stato indotto il loro ordinamento infatti, essi sarebbero separati da almeno altri due righe di quadrati, che come sappiamo hanno lato $\delta/2$. Ne consegue che il numero di confronti da effettuare per ogni punto di S è limitato da una costante.



Algoritmo I.6 COLSEST-PAIR(P)

1: $P_x \leftarrow$ SORT(P, x) {ordina i punti secondo la coordinata x}
2: $P_y \leftarrow$ SORT(P, y){ordina i punti secondo la coordinata y}
3: $(p, p') \leftarrow$ CLOSEST-PAIR-REC(P_x, P_y)

Algoritmo I.7 COLSEST-PAIR-REC(P_x, P_y)

1: **if** $|P| \leq 3$ **then**
2: (p, p') be the closest pair computed by enumeration
3: **return** (p, p')
4: **else**
5: crea la partizione Q, R estraendo gli insiemi ordinati Q_x, R_x, Q_y, R_y
6: $(q, q') \leftarrow$ CLOSEST-PAIR-REC(Q_x, Q_y)
7: $(r, r') \leftarrow$ CLOSEST-PAIR-REC(R_x, R_y)
8: $\delta \leftarrow \min\{d(q, q'), d(r, r')\}$
9: $x^* \leftarrow \max\{q_x : q \in Q\}$
10: $S \leftarrow \{p \in P : |p_x - x^*| < \delta\}$
11: costruisci S_y estraendolo da P_y
12: $d_{min} \leftarrow \infty$
13: **for each** $s_i \in S_y$ **do**
14: **for** $s_j \leftarrow s_{i+1}, \ldots, s_{i+10}$ **do**
15: **if** $d(s_i, s_j) < d_{min}$ **then**
16: $d_{min} \leftarrow d(s_i, s_j), (s, s') \leftarrow (s_i, s_j)$
17: **if** $d(s, s') < \delta$ **then**
18: **return** s, s'
19: **else**
20: **if** $d(q, q') < d(r, r')$ **then**
21: **return** (q, q')
22: **else**
23: **return** (r, r')

Di seguito sono illustrati pseudocodice e analisi di complessità.
Complessità. Uno sguardo all'algoritmo ci permette di ottenere la forma ricorsiva:

$$\begin{cases} T(n) = c, \text{ se } n \leq 3 \\ T(n) = 2T(n/2) + O(n), \text{ se } n > 3 \end{cases}$$

Si osservi infatti che la costruzione dei sottoinsiemi ordinata può essere effettuata in tempo lineare grazie alla fase di ordinamento degli insiemi P_x e P_y fatta nel preprocessing. Il doppio `for` comporta al massimo $10 \cdot n$ calcoli della distanza euclidea. La formula ricorsiva di cui sopra è del tutto analoga a quella che descrive la complessità di MERGE-SORT. Anche in questo caso, quindi, è $\Theta(n \lg n)$.

Capitolo II

Randomized algorithms and performance analysis

II.1 Worst case and average case complexity

In the previous chapter we considered different algorithms, even sophisticated ones, and we tried to improve their worst case complexity. It may sometimes happen that an algorithm which has been extremely optimized for the worst case, in practice behaves worse than a simpler algorithm that has a worse complexity. This leads us to study the behavior of the algorithms in the average case and to adopt some appropriate tricks to make the worst case slightly probable.

Let us start with the analysis of the case of sorting algorithms. We previously studied the MERGE-SORT algorithm. This algorithm has an optimal worst case complexity but it has some relevant flaws. The first one is of practical nature. Because of its implementation, it requires to allocate memory to maintain a copy of the array that must be ordered. Moreover, the algorithm doesn't work in contiguous memory space (it's not an *in-place* or local algorithm). The second drawback is bound to its complexity: if we analyze the algorithm we can notice that, independently of the disposition of the data in the input array, the number of operations made by the algorithm doesn't change. The first drawback can be avoided with another type of algorithm known as HEAP-SORT which uses a particular data structure (a binary heap) and has the same complexity as MERGE-SORT. However, this algorithm doesn't avoid the second drawback.

In the following section we'll study the simple QUICK-SORT algorithm which has a worse worst case complexity than the other two algorithm we mentioned earlier, but that it comes out to be competitive in practice.

II.1.1 QUICK-SORT algorithm

The algorithm is recursive as MERGE-SORT; for each iteration the sorted array is considered to be the result of the sorting of two portions of that array. The heart of the algorithm is the policy used to split the original array into two sub-arrays; the method is

really simple and is based on the selection of one element, defined *pivot*, which becomes the "central" element used to partition the other elements into the two sub-arrays. All the elements that are smaller than the pivot are moved in the first partition, while the greater ones are moved in the second partition. This way the position occupied by the pivot will be its definitive position in the the sorted array and the algorithm will be repeated on the two partitions on the right and on the left of the pivot.

Algoritmo II.1 QUICK-SORT(A,p,r)

```
1: if p<r then
2:    q ← PARTITION(A, p, r)
3:    QUICK-SORT(A, p, q − 1)
4:    QUICK-SORT(A, q + 1, r)
```

Algoritmo II.2 PARTITION(A,p,r)

```
1: pivot ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:    if A[j] ≤ pivot then
5:       i ← i + 1
6:       A[i] ↔ A[j]
7: A[i + 1] ↔ A[r]
8: return i + 1
```

We indicated with the symbol "↔" an operation of swap between two entries.

You can notice how the QUICK-SORT algorithm does not require the duplication of the data and how it does always operate over contiguous memory space. To have a better understanding of the operation of the algorithm and the characteristics we observed above, let us consider a simple example:

$$\begin{array}{ccccccccccc} & p & & & & & & & r & \\ [& 2 & 8 & 7 & 1 & 3 & 5 & 6 & 4 &] \\ i & & & & & & & & \textit{pivot} & \end{array}$$

- 4 is selected as the pivot value (being the last element of the array to be sorted);
- i is initialized as index for the scanning of the sequence; in practice i indicates the boundary between the elements that are lower than the pivot and the greater ones or those yet to be examined;
- the array is scanned from the first to the second last cell through the index j and the elements which are smaller or equal than the pivot are moved in the first half through a swap.

Let us have a look at some iterations.

$$\begin{array}{ccccccccccc} [& 2 & 8 & 7 & 1 & 3 & 5 & 6 & 4 &] \\ & i & & & j & & & & \textit{pivot} & \end{array}$$

$A[j]$ is smaller than the pivot and is swapped with the first subsequent element $A[i]$ and finally i is incremented.

$$\begin{array}{ccccccccc} [& 2 & 1 & 7 & 8 & 3 & 5 & 6 & 4 &] \\ & & i & & j & & & & pivot & \end{array}$$

Eventually the pivot is placed at the boundary between the lower elements and the greater ones with the final swap.

$$\begin{array}{cccccccccc} [& 2 & 1 & 3 & 4 & 7 & 5 & 6 & 8 &] \\ & & & & pivot & & & & & \end{array}$$

Complexity

Let us consider three possible cases:

- *optimal case*: the chosen pivot is always at the center of the partition
- *worst case*: the chosen pivot is always the maximum, or the minimum, of the elements to be partitioned
- *average case*: the chosen pivot is generally collocated in the surrounding of the center of the partition

We will analyze separately the three cases, using $T(\cdot)$ to indicate the number of elementary operations executed by the algorithm.

Optimal case At each recursive step the complexity is proportional to the two generated partitions plus a number of operations which is linear in the dimension of the array due to the partition:

$$T(n) = 2T(\frac{n-1}{2}) + \Theta(n)$$

For the master theorem the complexity of the algorithm results:

$$T(n) = \Theta(n \cdot \lg(n))$$

In the optimal case we can notice that the complexity of the algorithm is similar to that of MERGE-SORT

Worst case The worst case yields an unbalance in the generation of the two partitions upon which to apply recursively the algorithm: a partition will always be empty, while the other one will contain all the elements except the pivot; the complexity results:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

which is

$$T(n) = O(n^2)$$

Average case In the average case the complexity of the algorithm varies depending on the size of the subproblems that must be solved recursively. If we consider the two partitions that are subject of the recursion, at each call of the algorithm the complexity is:

$$T(n) = T(\alpha \cdot (n-1)) + T(1 - \alpha \cdot (n-1) + \Theta(n)$$

in which $\frac{1}{2} \leq \alpha \leq 1$ represents the size of the largest of the two sub-arrays, while $\Theta(n)$ again represents the cost connected to the partition of the initial array. The value of α can obviously vary at each iteration. We will deal with the average case more formally later on, when we will introduced the necessary analysis tools. For the moment we will limit our consideration to a case in which α is constant even if distant from the ideal case ($\alpha = 1/2$), for example $\alpha = \frac{9}{10}$. Developing the recursion tree we find that the depth of the tree is $O(\log_{\frac{1}{\alpha}})$ and that the number of elementary operations executed in each level of the tree is $O(n)$. We can conclude that the complexity for the unbalanced case is:

$$T(n) = O(n \cdot \log_{\frac{1}{\alpha}}(n))$$

which is even $O(n \ln n)$. We will see that this result is valid *on the average* also in the case of partitions with non constant dimensions.

II.2 Randomized algorithms: basic notation and tools

An algorithm is randomized if its behavior is determined not only by its input but also by some values produced by a random-number generator. We shall assume that we have at our disposal a random-number generator RAND. A call to RAND(a, b) returns an integer between a and b with each such integer being equally likely. For example, RAND$(0, 1)$ produces 0 with probability 1/2, and 1 with probability 1/2 whereas a call to RAND$(3, 7)$ returns either 3, 4, 5, 6 or 7, each with probability 1/5. Each value returned by RAND is independent of the previously returned values. We will always assume that an execution of RAND takes constant time.

Randomized algorithm are divided into two categories: *Las Vegas* and *Montecarlo*. A Las Vegas algorithm is a randomized algorithm that always returns the correct result; that is, it produces the correct result or it informs about the failure. In other words, a Las Vegas algorithm does not gamble with the correctness of the result; it only gambles with the resources or time used for the computation. On the other hand, a Monte Carlo algorithm is an algorithm whose running time is deterministic, but whose output may be correct only with a certain probability. In this case we may distinguish between *false positive* answers, that is when the algorithm returns a YES answer though the correct answer is NO, and *false negative* answers, that is when the algorithm returns a NO answer though the correct answer is YES.

II.2.1 Uniformity of the input: randomization

Studying the QUICK-SORT algorithm we have seen how sometimes the distribution of the input data is crucial, in particular how this can affect the pivot selection. Consider the

case of a company that applies to an agency for the selection of personnel with the purpose of hiring a new staff member. The target of the company is to hire the best candidate that is proposed by the agency. The agreed procedure with the agency is the following: the candidates are interviewed one at time, if the candidate is better than the preceding one, the person currently in the staff is dismissed and the new one is hired. The dismissal and the hiring of a new employee involves additional cost and ideally it would be preferable to minimize it. This happens only if the agency yields to the company the best candidate as the first one. Obviously the agency, being payed for each new recruitment, tends to maximize the number of recruitments. Hence if it knew the criterion used for the selection by the agency, it should provide the candidates to be interviewed in increasing order of "value".

Even in the case of the QUICK-SORT algorithm, in order to induce a pathological behavior we could provide an already sorted sequence, thus incurring in the quadratic worst case.

To avoid possible "perturbations" in the order of the input data we should try to make equiprobable all the configurations. A very simple way to do this is to randomly "shuffle" the input data. This apparently simple operation must: a) guarantee equiprobability, b) have a computational that does not affect the overall complexity of the original algorithm.

II.2.1.1 Randomization with key

The idea is to associate each element of the input with a priority uniformly generated within a range of values such as to make improbable the extraction of two equal keys. Then, such priority will be used to reorder the input elements. Through this pre-processing of the data the equiprobability of the input combinations is guaranteed.

More formally, consider an input sequence given by array A. To each element of A we associate a random priority chosen in a broad range (for example $[1 \dots n^3]$) such that the probability of having elements with the same priority is low. The we sort the elements of A using the generated key:

Algoritmo II.3 PERMUTE-BY-SORTING(A)

```
1: n ← length(A)
2: for i ← 1 to n do
3:     P ← RAND(1, n^3)
4: sort A using P as key
5: return
```

Let us now analyze the probability that the obtained permutation is the identity (we could repeat the same reasoning for any permutation). We call X_i the random variable that is associated to the event "$A[i]$ has the i-th priority". Hence, the identical permutation corresponds to:

$$I = X_1 \cap X_2 \cap \dots \cap X_n$$

Teorema II.1. *The probability of the event I is $\frac{1}{n!}$* □

Dimostrazione.

$$Pr\{I\} = Pr\{X_1\} \cdot Pr\{X_2|X_1\} \cdot Pr\{X_3|X_1 \cap X_2\} \cdot \ldots \cdot Pr\{X_n|X_1 \cap \ldots \cap X_{n-1}\}$$

Estimating the single probabilities:

- $Pr\{X_1\} = \frac{1}{n}$ since each element has the same probability of getting the minimal priority;
- $Pr\{X_2|X_1\} = \frac{1}{n-1}$ since each of the remaining elements has the same probability of getting the second priority when the first one is already assigned.
- $\ldots$
- $Pr\{X_i|X_1 \cap X_2 \cap \ldots \cap X_{i-1}\} = \frac{1}{n-i+1}$

Multiplying the various calculated priorities we prove the thesis. □

The process illustrated above, despite being correct (as proved in the theorem II.1), has two principal flaws. The first one concerns the complexity, which is lower bounded by the computational cost of the sorting; thus in case of the application to algorithms whose complexity is lower may cause a worsening of the performances. The second flaw is concerned with the usage of memory. In fact the process, in addition to requiring that an array of the same size of the input is used for the memorization of the priorities, it does not work locally.

II.2.1.2 Linear in-place randomization

Given a vector of n elements it is possible to permute the single elements by exchanging only once position at a time. The algorithm that we propose below (RANDOMIZE-IN-PLACE) has the advantage of not requiring additional memory for its execution, except the memory needed to perform the swap of two elements in the array.

Algoritmo II.4 RANDOMIZE-IN-PLACE(A)

```
1: n ← length(A)
2: for i ← 1 to n do
3:     A[i] ↔ A[RAND(i, n)]
```

Notice that RAND could also return i itself, in which case the subsequent swap instruction would not have any effect and the element $A[i]$ would remain in the i-th position. Observe that the complexity of RANDOMIZE-IN-PLACE, according to above hypothesis, is $\Theta(n)$. Less trivial is to prove that it produces any permutation with probability $\frac{1}{n!}$. The proof is based on the following result.

Lemma II.2. *At the beginning of iteration i of* RANDOMIZE-IN-PLACE $A[1, \ldots, i]$ *contains the identity permutation with probability* $\frac{(n-i+1)!}{n!}$. □

Dimostrazione. The proof is made by induction over the iteration index i.

- *inductive hypothesis* At the step $i = 1$ $A[1..0] = \varnothing$ that corresponds to the null identity permutation and this happens with probability $\frac{(0-1+1)!}{0!} = 1$. Analyzing also the step with $i = 2$, $A[1]$ contains the identity permutation only if $A[1]$ has not been exchanged during the previous iteration, which happens with probability $\frac{1}{n}$.

- *inductive step* Let us suppose that the identity permutation of $A[1, \ldots, i-1]$ before the iteration i can happen with probability $\frac{(n-i+1)!}{n!}$. We want to prove that at the end of i-th iteration the probability to have the identity permutation in $A[1, \ldots, i-1]$ is $\frac{(n-i)!}{n!}$.

  At the i-th iteration, in the array A in positions from 1 to i we have the following elements:

  $$[a_1, a_2, \ldots, a_{i-1}, a_i]$$

  that is equivalent to state that the permutation $[a_1, a_2, \ldots, a_{i-1}]$ is followed by the element a_i. Denoting with I_i the identity till the i-th element, for the inductive hypothesis, we have that:

  $$Pr\{[a_1, a_2, \ldots, a_{i-1}] = I_{i-1}\} = \frac{(n-i+1)!}{n!}$$

  Furthermore

  $$Pr\{[a_1, a_2, \ldots, a_i] = I_i\} = Pr\{a_i \text{ in posizione } i | [a_1, a_2, \ldots, a_{i-1}] = I_{i-1}\}$$

  Notice that $Pr\{a_i \text{ in posizione} i | [a_1, a_2, \ldots, a_{i-1}] = I_{i-1}\} = \frac{1}{n-i+1}$ because a_i is chosen between the $(n-i+1)$ elements in $A[i, \ldots n]$. Calculating the product the thesis follows. □

Using the result of lemma II.2 and applying it for $i = n$ we obtain that the identity permutation occurs with probability $\frac{1}{n!}$, as in the case of the rPERMUTE-BY-SORTING. Hence the two methods are equivalent, but RANDOMIZE-IN-PLACE is more efficient.

II.2.2 Indicator Random Variables

Let us formally introduce a tool that we will use for the complexity analysis of the average case. For each random event A we can introduce a random variable $I\{A\}$ that takes values 0 or 1 with the following criterion:

$$I\{A\} = \begin{cases} 1, & \text{if the event A takes place} \\ 0, & \text{otherwise} \end{cases}$$

This variable is defined as *Indicator Random Variable*.

Let us consider a simple example to understand its use. In the tossing of a coin the two events T="tail comes out", e H="head comes out" occur with probability $1/2$. We can introduce the indicator random variable $I\{T\}$.

The following result is valid:

Lemma II.3. *Given a space of events S and an event $A \in S$, considered the variable X_A that gives the number of occurrences of the event A, then we have that $X_A = I\{A\}$ and $E[X_A] = Pr\{A\}$.* □

Dimostrazione.

$$E[X_A] = E[I\{A\}] = 1 \cdot Pr\{A\} + 0 \cdot Pr\{\bar{A}\} = Pr\{A\}$$

where $\bar{A}$ is the complement of A. □

Applying the lemma II.3 to the example of the coins, and willing to calculate the expected value of the variable X_T that gives the number of T occurring in a toss, we have that

$$E[X_T] = \frac{1}{2}.$$

If we consider n tossing of the same coin, indicating with X the number of occurrences of T in the n tossing and with X_i the number of occurrences of T at the i-th toss we have:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$$

and by the linearity of $E[\cdot]$ it follows that:

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}.$$

Another simple example of Indicator Random Variables application is the following. In an international conference n professors resides in the same hotel having n single rooms, each assigned to a professor. After the social dinner, professors return to the hotel and being slightly drunk they cannot remember their room number, thus each of them picks randomly a key and sleeps in a randomly chosen room. We can apply the Indicator Random Variables to estimate the expected number of professors that enters in the correct room. Let us define the following variable:

$$X_i = \begin{cases} 1, & \text{if professor } i \text{ sleeps in the correct room} \\ 0, & \text{otherwise} \end{cases}$$

We can now calculate the expected value:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$$

For the linearity of $E[\cdot]$ follows that:

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Pr\{\text{professor } i \text{ sleeps in the correct room}\} = \sum_{i=1}^{n} \frac{1}{n} = \frac{n}{n} = 1.$$

This means that we can expect that only one professor will wake up in the morning in the original room.

II.3 Las Vegas algorithms

We will apply the indicator random variables to the detailed analysis of the two algorithms that we formally discussed previously. In both cases, using randomization techniques, we can assume that input data are uniformly distributed and that any permutation is equiprobable. Moreover we will consider the case of the median selection problem presenting a Las Vegas algorithm with linear average complexity and we will exploit this result to devise a deterministic linear worst case algorithm.

II.3.1 Average cost of the hiring problem

In this case the time complexity of the algorithm is not an issue. We are rather interested in evaluating the average number of times that we hire a new employee. Let X be the random variable giving the number of hirings; we want to evaluate the expected value $E[X]$. Instead of proceeding in the usual way doing $\sum_{x=1}^{n} xPr\{X = x\}$ we use indicator random variables:

$$X_i = I\{\text{hired the } i\text{-th candidate}\}$$

$$X = \sum_{i=1}^{n} X_i$$

By lemma II.3 $E[X_i] = Pr\{\text{hired the } i\text{-th candidate}\}$. The candidate i is hired when i is the best among the first i candidates, hence the probability that such event occurs is $\frac{1}{i}$. Therefore:

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{i}.$$

The expected number of hirings is given by a harmonic series arrested at the n-th term and so it's $O(\ln n)$.

II.3.2 Randomized Quick-Sort sorting algorithm

In the case of the QUICK-SORT algorithm it is not necessary to apply a randomization phase of input data. We can obtain the same effect by choosing a random *pivot* at each iteration and temporarily swap it with the last element of the sequence.

Algoritmo II.5 RAND-QUICK-SORT(A,p,r)

```
1: if p<r then
2:     q ← RAND-PARTITION(A, p, r)
3:     RAND-QUICK-SORT(A, p, q − 1)
4:     RAND-QUICK-SORT(A, q + 1, r)
```

Notice that PARTITION is the same function that we described in II.1.1. The essential observation for the evaluation of the average case complexity of the algorithm is about the fact that the partitioning phase examines one *pivot* that is compared with various elements. In the successive calls, the element that was chosen as a *pivot* will not be examined any

Algoritmo II.6 RAND-PARTITION(A,p,r)

1: $i \leftarrow$ RAND(p, r)
2: $A[i] \leftrightarrow A[r]$
3: **return** PARTITION(A, p, r)

more, moreover two elements that due to the partition are put in different subsequences will be never compared. The average computational complexity is proportional to the overall number of comparison performed by the algorithm.

Let $z_1, \ldots, z_n$ be the values, considered in increasing order, appearing in the sequence, and let us indicate with Z_{ij} the set of values ranging from z_i to z_j, that is $Z_{ij} = \{z_i, \ldots, z_j\}$. z_i and zj are compared only when either z_i or z_j is chosen as a *pivot*. According to the previous observation, we know that even after one of the two is chosen as a *pivot* the two elements z_i and z_j will never be compared. Let $X_{ij} = I\{z_i \text{ is compared with } z_j\}$, then the random variable that counts the number of comparison is:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

The expected value of X, using the properties of indicator random variables, has the following expression:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared with } z_j\}.$$

Hence, we can evaluate $Pr\{z_i \text{ is compared with } z_j\}$, keeping in mind that the *pivot* are chosen in an independent and uniform way, and remember that if we choose as a *pivot* a y such that $z_i < y < z_j$, then z_i and z_j will never be compared. Thus:

$$Pr\{z_i \text{ is compared with } z_j\} = Pr\{z_i \text{ is chosen as } \textit{pivot} \text{ o } z_j \text{ is chosen as } \textit{pivot}\}$$

Because the two events are exclusive and the number of elements in Z_{ij} is $(j - i + 1)$ we get:

$$Pr\{z_i \text{ is compared with } z_j\} = \frac{2}{j - i + 1}.$$

Therefore

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \leq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k};$$

because the second summation is in fact a harmonic series, limited by $O(\ln n)$, we can conclude that the expected number of comparisons is $O(n \lg n)$, and as a consequence also the average case complexity of the randomized QUICK-SORT algorithm is $O(n \lg n)$.

II.3.3 Finding the median

Given a sequence of n elements A, if we wanted to extract the minimum it would be necessary to scan the whole sequence while keeping in memory the lower element that is found during the process; the complexity of such operation is $O(n)$. To find the second, the third, or the k-th element (fixed any constant integer k) it is sufficient to scan once again the sequence, always keeping in memory the first 2, 3 or k elements found during the process; the complexity is yet $O(nk) = O(n)$, being k a constant. Conversely, if, for example, we want to find the median (that is the element that is collocated in the position $n/2$ in the ordered sequence), the complexity of the algorithm becomes $O(n \cdot n/2) = O(n^2)$. Since there exist algorithms capable of executing the sorting of A with complexity $O(n \log n)$, we can alternatively sort the vector and then extract the element in position $n/2$; in this way we can find the median with a complexity of $O(n \log n)$. As we will see, it is possible to obtain even better results: using a randomized algorithm it is possible to reach a complexity of $O(n)$ in the average case; we will devise also a deterministic algorithm that has a linear worst case complexity taking advantage of the ideas coming from the randomized version.

II.3.3.1 Randomized algorithm

The Quick-Sort algorithm divides the sequence to be ordered into two parts at each iteration, that corresponds to the subsequences of the elements smaller or greater than a particular element (the *pivot*). By recursively sorting the subsequences, as we have seen, we obtain the sorted vector in a time that in the average case is $O(n \log n)$.

A similar technique can be used for the search of the i-th lower element. Chosen randomly an element as *pivot* we divide the sequence into two pieces, that correspond to the greater and smaller elements of the *pivot*, similarly to what happens in Quick-Sort. At this point, if the *pivot* is in the i-th position then this means that there are exactly $i-1$ lesser elements, and so the searched element is the *pivot* itself; otherwise we continue the search recursively only in the subsequence that contains it. The fundamental difference with respect to Quick-Sort is in the fact that at each iteration only one of the two produced subsequences is considered. The algorithm, whose pseudo-code is reported, regards the general case in which we want to find the i-th lower element of a sequence; the case of the search of the median is obtained by assigning to i the value $n/2$.

Notice that when the function is invoked recursively over the subsequence of the values greater than the *pivot*, the parameter i is decreased by k units; in fact, since the first k elements are discarded, the searched element will be now k more positions ahead. The functions rand-partition and partition that are referred are the same ones that are defined for the rand-quick-sort algorithm (II.5). It is also important to notice that this algorithm always returns the correct result.

Complexity. Let us consider the (particularly unlucky) case in which at each iteration it's chosen as a *pivot* the maximum element of the sequence (the same holds if the minimum one is always chosen): the dimension of the sequence to deal with decreases by only one element (the *pivot*) at each iteration, and so it's necessary to perform $\Theta(n)$ iterations. Since at each iteration we have to compare $\Theta(n)$ elements with the *pivot*, the

Algoritmo II.7 RANDOMIZED-SELECT(A, p, r, i)

```
1:  if p = r then
2:     return A[p]
3:  q ← RAND-PARTITION(A,p,r)
4:  k ← q − p + 1
5:  if i = k then
6:     return A[q]
7:  if i < k then
8:     return RANDOMIZED-SELECT(A,p,q − 1, i)
9:  if i > k then
10:    return RANDOMIZED-SELECT(A,q + 1,r, i − k)
```

total complexity in the *worst case* is $\Theta(n^2)$. Conversely, in the average case the described algorithm behaves much better.

To perform the complexity analysis in the more general case, we denote the execution time of the algorithm with a random variable, $T(n)$. The decisive element is the choice of the *pivot*. We denote with X_k the probability of choosing as *pivot* the k-th element: $X_k = I\{A[p \dots q] \text{ has } k \text{ elements}\}$
The *pivot* ie randomly chosen between the n elements, therefore:

$$E[X_k] = \text{ Pr } \{k\text{-th element chosen as } pivot\} = 1/n$$

Since we want to find an upper bound for $T(n)$, we'll consider the case in which, at each iteration, the search should be continued over the bigger partition among the two that are created, with dimension respectively $k-1$ and $n-k$:

$$T(n) \leq \sum_{k=1}^{n} X_k T(max\{k-1, n-k\}) + O(n)$$

where the last term is due to the scanning of the array to compare all the elements with the *pivot*.

We can now calculate an upper bound for the average value of the execution time:

$$E[T(n)] \leq E\left[\sum_{k=1}^{n} X_k T(max\{k-i, n-k\})\right] + O(n) =$$

for the linearity of the expected value:

$$= \sum_{k=1}^{n} E[X_k T(max\{k-i, n-k\})] + O(n)$$

since the variables are independent between each other:

$$= \sum_{k=1}^{n} E[X_k] E[T(max\{k-i, n-k\})] + O(n)$$

substituting the value of X_k :

$$= \frac{1}{n} \sum_{k=1}^{n} E[T(max\{k-i, n-k\})] + O(n)$$

We observe that:

$$max\{k-1, n-k\} = \begin{cases} k-1, & \text{if } k > \lceil \frac{n}{2} \rceil \\ n-k, & \text{if } k \leq \lceil \frac{n}{2} \rceil \end{cases}$$

If n is even, all the terms from $T(\lceil \frac{n}{2} \rceil)$ to $T(n-1)$ will appear in the summation exactly two times; if n is odd they will all appear two times, except $T(\lceil \frac{n}{2} \rceil)$, that will appear once. Hence we can write:

$$E[T(n)] \leq 1/n \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} 2E[T(k)] + O(n)$$

where the equality is valid in the case of n odd, and the unequality in the case of n even.

Now we need to formulate an induction hypothesis; Let us assume that, for some constant c:

$$T(n) \leq c \cdot n, n > h$$
$$T(n) = O(1), n \leq h$$

where h is a proper constant. Let us take also a constant a such that $a \cdot n$ be an upper bound for $O(n)$. Substituting the terms in the unequality we can write:

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} (ck) + an$$

The summation from $\lfloor \frac{n}{2} \rfloor$ to $n-1$ can be written as the difference between two summations:

$$E[T(n)] \leq 2/n \left(\sum_{k=1}^{n-1} c - \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor - 1} c \cdot k \right) + a \cdot n$$

Remembering that $\sum_{k=1}^{n} k = \frac{(n-1)n}{2}$:

$$\begin{aligned} E[T(n)] \leq & \frac{2}{n}\left(\frac{c(n-1)n}{2} - \frac{c(\lfloor n/2 \rfloor - 1)\lfloor \frac{n}{2} \rfloor}{2}\right) + a \cdot n \leq \\ \leq & \frac{c}{n}\left((n-1)n - \left(\frac{n}{2} - 1\right)\frac{n}{2}\right) + a \cdot n = \\ = & c\left(n - 1 - \frac{n}{4} + \frac{1}{2}\right) + a \cdot n = \\ = & c\left(n - \frac{n}{4} - \frac{1}{2}\right) + a \cdot n \end{aligned}$$

from which, isolating the term $c \cdot n$:

$$E[T(n)] \leq c \cdot n - \left(c\frac{n}{4} - \frac{c}{2} - a \cdot n\right)$$

To avoid the contradiction of the initial assumption ($(n) \leq c \cdot n$) we need that, for sufficiently high values of n, the last expression doesn't overcome the value of $c \cdot n$, and so that the term between parenthesis isn't negative. Therefore we impose:

$$\frac{c \cdot n}{4} - \frac{c}{2} - an \geq 0$$

According to choosing a constant $c > 4a$, we can make explicit the relation with n:

$$n \geq \frac{\frac{c}{2}}{\frac{c}{4} - a} = \frac{2c}{c - 4a}$$

In conclusion, if we assume $T(n) = O(1)$ for $n < \frac{2c}{c-4a}$, we can conclude that the average time for the execution of the algorithm is linear.

II.3.3.2 Deterministic algorithm

The RANDOMIZED-SELECT algorithm has a linear execution time in the average case, but quadratic in the worst case. It's possible to make the algorithm deterministic and maintaining the complexity linear even in the worst case. This imply a burdening of the algorithm that in the randomized case was extremely simple. The method that we will apply is just an example of a more general technique, known as *derandomization technique* that can also be applied to many other algorithms. The critical point of the RANDOMIZED-QUICK-SORT algorithm is related to the choice of the *pivot* that may fall over the minimal or maximal element, deteriorating the performances. The ideal choice would be to choose a pivot as much as possible close to the median. Here's how it's possible to improve the choice of the *pivot* to limit the impact of the worse case:

Notice that while at step 2 the median is calculated in constant time since the groups are of constant and little dimension, the algorithm contains a recursive call at step 3 (when the median of the medians is calculated) and at steps 8 or 10.

Complexity. Let us now evaluate what is the advantage introduced by the first three steps of the algorithm in the worst case. Because of the way the value of X is chosen,

Algoritmo II.8 DETERMINISTIC-SELECT(A,i)

1: divide A in $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements each, and a group with $n \mod 5$ elements
2: for each group of 5 elements calculate the median
3: calculate the median of the medians X
4: partitionate A using X as *pivot*; let $k-1$ the number of elements contained in the lower subsequence (X is the k-th element of A)
5: **if** $k = i$ **then**
6: **return** X
7: **if** $k > i$ **then**
8: search in the first subsequence
9: **if** $k < i$ **then**
10: search in the second subsequence

it's possible to calculate the minimal number of elements of A that will be surely greater than X. In fact, at least half of the groups has at least 3 elements greater than X (in particular all the groups that have a value greater than X for the median). Starting from these groups we have to subtract no more than two of them (the last one and the one that contains X itself that contribute with less than three elements). Hence, formally we have that the number of elements greater than X is, at minimum:

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

The same reasoning can be applied to calculate the minimal number of elements that are lesser than X, that is the same that we calculated.

In the worst case, which consists in having the maximum unbalance between the two partitions, we will have a subsequence of $(\frac{3}{10}n - 6)$ elements and one with the complementary number, that is $(\frac{7}{10}n + 6)$. Therefore, in the worst case, at each iteration will be performed the recursive call over $(\frac{7}{10}n + 6)$ elements; it's now possible to calculate deterministically the execution time of the algorithm in this case:

$$T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left(\frac{7}{10}\right) n + 6\right) + O(n)$$

Suppose that we can consider $T(n)$ as a constant for values of $n \leq 140$ and proceed the calculation for values of $n > 140$.

Let us introduce an induction hypothesis: $T(n) \leq c \cdot n$ and one constant a such that:

$O(n) \leq a \cdot n$. We can now rewrite the previous equation as:

$$\begin{aligned} T(n) \leq & c \left\lceil \frac{n}{5} \right\rceil + c \left(\frac{7}{10} n + 6 \right) + an \\ \leq & c \left(\frac{n}{5} + 1 \right) + \frac{7}{10} cn + 6c + an \\ \leq & \frac{9}{10} cn + 7c + an \\ \leq & cn - \frac{1}{10} cn + 7c + an \end{aligned}$$

To avoid the contradiction of the initial hypothesis, it's needed that the last expression has a value $\leq c \cdot n$. This corresponds to imposing the condition:

$$-\frac{1}{10} c \cdot n + 7c + a \cdot n \leq 0$$

from which:

$$c \left(7 - \left(\frac{n}{10} \right) \right) \leq -a \cdot n$$

and, with $n > 70$:

$$c \geq 10a \left(\frac{n}{n - 70} \right)$$

Since we assumed that $n \geq 140$, we have to choose $c \geq 20a$. It's possible to have lower values of n (providing they are strictly greater than 70), according to choose an adequate constant c. In conclusion, we proved that, for very high values of the constant c, the algorithm has a linear complexity in the worst case.

II.4 Montecarlo algorithms

II.4.1 Matrix multiplication test

When we deal with matrices, many simple operations come out to have a high complexity in practice. We will now consider a simple Montecarlo algorithm used to test the equivalence between a matrix and the product of other two matrices.

Let us consider three $n \times n$ matrices: A, B and C. Our goal is to determine whether $C = A \cdot B$. The deterministic approach implies to calculate the multiplication $A \cdot B$ and then make the comparison between the result and C. While the second phase would have a fixed complexity of $O(n^2)$, the first phase could have different complexities based on the implementation: $O(n^3)$ (standard row $\times$ columns algorithm), $O(n^{2.807})$ (Strassen's algorithm) or $O(n^{2.376})$ (Coppersmith's algorithm).

In order to improve the time complexity we could apply the following randomized approach. We generate a random vector r of size n and make the comparison between $A \cdot (B \cdot r)$ and $C \cdot r$. If the two resulting vectors are different we can be sure that $C \neq A \cdot B$,

Algoritmo II.9 MATRIX-MUL-TEST(A,B,C)

1: generate the vector $r = [r_1 \cdots r_n]$ where each r_i is randomly picked from $S, |S| \geq 2$
2: **if** $A \cdot (B \cdot r) \neq C \cdot r$ **then**
3: **return** The equality does not hold
4: **else**
5: **return** The equality holds in probability

otherwise we cannot be sure that $C = A \cdot B$, but the probability that they are different is reasonably small, depending on how we generate vector r.

Notice that, being a Montecarlo algorithm, MATRIX-MUL-TEST can give an answer which is not correct. In particular, if $A \cdot (B \cdot r) = C \cdot r$ we cannot be sure that the equality $C = A \cdot B$ holds.

Complexity. The complexity analysis for this algorithm is rather simple: the algorithm has to perform two matrix×vector products to evaluate the left hand side of the expression and another matrix×vector product to evaluate the right hand side. Finally, a comparison between two vectors is needed. Therefore, the final complexity is: $O(2n^2 + n^2 + n)$ that is equal to $O(n^2)$.

Error Rate. The most important analysis of Montecarlo algorithms is the evaluation of the error rate. The algorithm that we proposed above can return a false positive when the equality does not actually hold but the value of r satisfies $A \cdot (B \cdot r) = C \cdot r$. Hence, our goal is to evaluate the probability:

$$\Pr\{\text{false positive}\} = \Pr\{A \cdot (B \cdot r) = C \cdot r \mid C \neq A \cdot B\}$$

Supposing that $C \neq A \cdot B$, we can define the matrix $D = A \cdot B - C$ such that $D \neq 0$. With this formulation, the probability is:

$$\Pr\{\text{false positive}\} = \Pr\{D \cdot r = 0\}$$

Without loss of generality, we can suppose suppose that $d_{11} \neq 0$, where D_{11} is the element of D in the first column and in the first row. If we impose that $D \cdot r = 0$, we need to impose that each product $D_i \cdot r = 0$, where D_i is the i-th row of D. Considering the first row we have:

$$D_1 \cdot r = 0$$

That is:

$$d_{11} \cdot r_1 + d_{12} \cdot r_2 + \cdots + d_{1n} \cdot r_n = 0$$

Isolating r_1 we have:

$$r_1 = -\frac{1}{d_{11}} \cdot (d_{12} \cdot r_2 + \cdots + d_{1n} \cdot r_n)$$

This means that, for each choice of $r_2, r_3, \ldots, r_n$, there exists only one value of r_1 that satisfies $D_1 \cdot r = 0$. Hence, we can conclude that the probability of having $D \cdot r = 0$ is equal to the probability of choosing the value for r_1 from the set S that makes zero the first component of D (according that such value is contained in S). Hence:

$$\Pr\{D \cdot r = 0\} \approx \frac{1}{|S|}$$

Since S contains at least 2 elements:

$$\Pr\ \{\ D \cdot r = 0\ \} \leq \frac{1}{2}$$

If we want to improve the error rate we can:

- increase the number of elements in the set S, thus having an error rate of $\approx \frac{1}{|S|}$
- execute t times the algorithm, thus obtaining an error rate of $\approx \frac{1}{|S|^t}$

The idea of executing the same algorithm many times on the same input is a classical technique used to overcome the problem of fallibility of Montecarlo algorithms.

II.4.2 Minimum cut of a graph

Consider an undirected multigraph $G = (N, A)$ hence in G we may have multiple copies of arcs. Let $n = |N|$ and $m = |A|$ and suppose that G is connected. A *cut* C of G is a subset of arcs such that the new graph $G' = (N, A \setminus C)$ is not connected. The minimum cut problem consists in finding a cut C of minimum cardinality. It can be noticed that the problem has a combinatorial nature, indeed in order to define a cut it is sufficient to find a partition of N into two subsets N_1 and N_2, neither of the two is empty, and the cut is given by the set of arcs having one endpoint in N_1 and the other in N_2. Determining the partition corresponding to the minimum cut by enumeration is not practical as it would require an exponential time.

Gomory and Hu devised the first polynomial deterministic algorithm for solving the problem. The basic idea of the algorithm is to repeat a sequence of $n - 1$ maximum flow computations on a suitable directed graph thus yielding a complexity of $O(n^4)$. Successively other algorithms, simpler to be implemented, have been proposed by Gusfield and by Talluri.

A possible randomized Montecarlo algorithm is extremely simple with respect the deterministic ones. The algorithm is based on the graph contraction operation.

Let (x, y) be one arc of A the contraction of G with respect to (x, y), denoted by $G/(x, y)$ is a new graph where nodes x and y are substituted by a new node z, arc (x, y) is removed, each arc (x, v) or $(y, v) \in A$ is substituted by a new arc (z, v). Note that this operation may generate multiple copies of arcs that are maintained in the graph.

The result of the contraction of a subset of arcs is independent from the order in which arcs are considered. The contraction operation can be implemented in $O(n)$.

The Montecarlo algorithm for determining a minimum cut applies a sequence of contractions on arcs picked randomly from A and stops when only two nodes are left. The two nodes, corresponding to set of nodes of the original graph, identify the partition corresponding to the searched cut. The pseudocode is the following:

Provided that the contraction operation can be carried out in $O(n)$ time, the overall complexity is $O(n^2)$, since we repeat the contraction $(n - 2)$ times.

Let us analyze the probability that the algorithm outputs the correct solution, that is it identifies the minimum cardinality cut of the graph.

Let us denote by k the cardinality of the minimum cut, the following properties hold trivially:

Algoritmo II.10 RAND-CONTRACT(G)

1: $H \leftarrow G$
2: **while** H has more than two nodes **do**
3: randomly select one arc (x, y) of H
4: $H \leftarrow H/(x, y)$
5: **return** $C = \{(v, w) : v, w$ are the two remaining nodes of $H\}$

- RAND-CONTRACT outputs the minimum cut if and only if it never contracts one arc of the minimum cut.
- In G no node has degree less than k
- $m \geq \frac{nk}{2}$
- Consider one arc $(x, y) \in A$, the minimum cardinality cut of $G/(x, y)$ is at least as large as the minimum cardinality cut of G.

Let us analyze the i-the iteration of the algorithm. At the beginning of the iteration the number of nodes in H is $n_i = n - i + 1$. If in previous iterations we did not contract any arc of the minimum cut K k is still a minimum cut of H and k is the cardinality of the minimum cut. This implies that $fracn_ik2$ is the minimum number of arcs left in H, thus the probability of picking one arc of K is $frac2k$.

Note that at the beginning of the algorithm the probability of making a wrong choice is small, while it increases with the number of iterations.

We can now derive the probability p of obtaining the minimum cut. This happens only if we never select a wrong arc, thus

$$p \geq \prod_{i=1}^{n-2}(1 - \frac{2}{n_i}) = \prod_{i=1}^{n-2}(1 - \frac{2}{n - i + 1}) =$$

$$= \prod_{i=1}^{n-2} \frac{n - i + 1 - 2}{n - i + 1} = \prod_{i=1}^{n-2} \frac{n - i - 1}{n - i + 1} = \prod_{i=3}^{n} \frac{i - 2}{i}$$

$$= \frac{1}{\binom{n}{2}} = \Omega(n^{-2})$$

This means that the probability of generating the correct solution is small but not negligible. In order to increase the success probability it is sufficient to iterate the algorithm. With $n^2/2$ iterations the overall complexity is comparable with that of an efficient deterministic algorithm, though in practice it is much simpler, and the error probability is less than $1/e$.

As we observed previously, the error probability increases with the number of iterations. It is thus possible to improve the performance of the algorithm at a relatively small computational cost using two invocations of the algorithm on small graphs. The resulting algorithm is as follows. Where RAND-CONTRACT(G, t) denotes the contract algorithm

Algoritmo II.11 REC-RAND-CONTRACT(G)

```
1: if H has n ≤ 6 then
2:     return the minimum cut C computed by enumeration
3: else
4:     t ← ⌈1 + n/√n⌉
5:     H_1 ← RAND-CONTRACT(G, t)
6:     H_2 ← RAND-CONTRACT(G, t)
7:     C_1 ← REC-RAND-CONTRACT(H_1)
8:     C_2 ← REC-RAND-CONTRACT(H_1)
9:     if |C_1| < |C_2| then
10:        return C_1
11:    else
12:        return C_2
```

arrested at iteration t. The complexity of the algorithm is given by the following recursive formula:

$$T(n) = 2T(\frac{n}{\sqrt{2}}) + O(n^2)$$

which in closed form results:

$$T(n) = O(n^2 \lg n)$$

Thus the overhead due to the double invocation is a $\lg n$ factor, while it is possible to prove that the success probability becomes

$$\Omega(\frac{1}{\lg n})$$

II.4.3 Polynomial identity test

Many mathematical applications reduce to the comparison of two or more polynomials in order to find out if they are identical. For this kind of problem a deterministic approach can result very time-consuming in terms of computational time.

Let us consider two polynomials Q and R, defined in n variables $(x_1, x_2, \ldots, x_n)$, for which we want to check the identity. A deterministic approach could be the following: first we expand both Q and R as sums of monomials and then we make a comparison between each term of the same grade of the two polynomials. This algorithm has an exponential complexity, as we can infer from the expansion of the following polynomial:

$$Q = \sum_{i=1}^{n-1} (x_i + x_{i+1}) = (x_1 + x_2) \cdot (x_2 + x_3) \cdot (x_3 + x_4) \cdots$$

If we calculate the products in their order, at the k-th multiplication Q is reduced to the product of a polynomial with 2^{k+1} terms with the remaining binomials. Hence the complexity for this deterministic algorithm is $O(2^n)$.

A more efficient way to perform the identity test between polynomials is a randomized algorithm (POLY-TEST) that assigns randomly generated values to the n variables and tests if this instantiation of variables nullify the polynomial $Q - R$.

Algoritmo II.12 POLY-TEST(Q,R)

1: $P = Q - R$
2: generate a set $S = \{r_1, r_2, \dots, r_n\}$ of randomly chosen values
3: **if** $P(r_1, r_2, \dots, r_n) = 0$ **then**
4: **return** $Q = R$ (with high probability)
5: **else**
6: **return** $Q \neq R$ (surely)

Note that the algorithm may produce a false positive answer (thus state the identity of the polynomials although they are different). This algorithm is known as *Schwartz-Zippel Algorithm.*

Complexity. Supposing that the generation of the random values r_i can takes constant time, the complexity of the algorithm is dominated by the substitution of those values in the variables of the polynomial P, that takes a time which is proportional to the number of variables n. Therefore the algorithm has a linear complexity: $O(n)$.

Error Rate. Since the algorithm can give the wrong result, we need to evaluate the probability of a false positive. Thus we have to evaluate:

$$\Pr \{ \ P(r_1, r_2, \dots, r_n) = 0 \mid P \neq 0 \ \}$$

Teorema II.4. *If $P \neq 0$ then*

$$Pr \{ \ P(r_1, r_2, \dots, r_n) = 0\} \leq \frac{d}{|S|}$$

where d is the degree of P. □

Dimostrazione. The proof is by induction over the number of variables n.

- *inductive hypothesis* If P is a polynomial in $n = 1$ variables then the number of its distinct roots is at most equal to its degree d. Hence:

$$\Pr \{ \ P(r_1) = 0\} = \frac{\text{number of distinct roots}}{\text{number of trials}} \leq \frac{d}{|S|}$$

- *inductive step* Let us suppose that the desired property holds for polynomials with $n - 1$ variables, that is

$$\Pr \{ \ P(r_1, r_2, \dots, r_{n-1}) = 0\} \leq \frac{d}{|S|}$$

  Let us now consider a polynomial P with n variables. If k is the maximum degree of the variable x_1 in P we can write the polynomial in this form:

$$P(x_1, x_2, \dots, x_n) = M(x_2, \dots, x_n) \cdot x_1^k + N(x_1, x_2, \dots, x_n)$$

  Where:

  - the variable x_1 in M has a maximum degree of $d - k$
  - the variable x_1 in N has a maximum degree of $k - 1$

Let us now consider the event: $\epsilon =$ "$M(r_2, \ldots, r_n) = 0$". We have that:

$$\Pr\{P(r_1, r_2, \ldots, r_n) = 0\} =$$

$$= \Pr\{P(r_1, r_2, \ldots, r_n) = 0|\epsilon\} \Pr\{\epsilon\} + \Pr\{P(r_1, r_2, \ldots, r_n) = 0|\bar{\epsilon}\}\Pr\{\bar{\epsilon}\}$$

(where $\bar{\epsilon}$ is the complement of the event ϵ). From the inductive hypothesis:

$$\Pr\{\epsilon\} \leq \frac{d-k}{|S|}$$

The probability $\Pr\{P(r_1, r_2, \ldots, r_n) = 0|\bar{\epsilon}\}$ can be obtained using this reasoning: we have fixed the values for $r_2, \ldots, r_n$ and we have seen that they do not nullify M; thus, the only way to nullify P is to choose the value of x_1 that nullifies $P(x_1, r_2, \ldots, r_n) = P'(x_1)$. Hence:

$$\Pr\{P'(r_1) = 0\} \leq \frac{k}{|S|}$$

where we used the fact that the maximum degree of x_1 in P' is k. Considering that the remaining probabilities must obviously be less than 1, the probability is:

$$\Pr\{P(r_1, r_2, \ldots, r_n) = 0\} \leq 1 \cdot \frac{d-k}{|S|} + \frac{k}{|S|} \cdot 1 = \frac{d}{|S|} \qquad \square$$

As we proved, the error rate is: $\Pr\{\text{false positive}\} \leq \frac{d}{|S|}$ Therefore, if we want to decrease the probability of errors we can choose a large set S. In fact, if $|S| \geq 2 \cdot d$ then $\Pr\{\text{false positive}\} \leq \frac{1}{2}$. Again, the algorithm can be executed many times over the same input to improve the error rate. It is also possible to derandomize this algorithm, but there are some theoretical bounds to its complexity (*Shpilka* and *Kabonets/Impagliazzo*).

II.4.4 Determinant of a matrix

A simple application of Schwartz-Zippel algorithm is in testing matrix singularity. Given a square matrix A

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix}$$

the determinant is defined as follows:
(II.1)

$$\det(A) = \sum_{\sigma \in \Sigma} sgn(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)}$$

where Σ denotes the set of $n-$permutations. This expression can be seen as a polynomial where a_{ij} are the variables, we can apply the POLY-TEST algorithm to find if A can be singular for any value of entries a_{ij}. Clearly this method is not very practical to test singularity for a particular matrix instance, since there are efficient method to compute the determinant, however it can be used to solve other problems very efficiently.

II.4.4.1 Perfect Matching

One possible application is related to the problem of the *Perfect Matching*. We are given a bipartite undirected graph: $G = (N_1 \cup N_2, E)$ with $E \subseteq N_1 \times N_2$. A *matching* is a set of edges $M \subseteq E$ such that no two edges incident in the same node are contained in M. A matching is *perfect* if it has maximum cardinality. In the case of $|N_1| = |N_2| = n$ the perfect matching should have a cardinality of n (that is: $|M| = n$). Given an undirected graph G we can define its "Tutte's matrix" A_G in this way:

$$a_{ij} = \begin{cases} x_{ij}, & \text{if (i,j)} \in E \\ 0, & \text{otherwise} \end{cases}$$

where x_{ij} is a variable. If we consider the case of $|N_1| = |N_2| = n$ we can apply the following theorem:

Teorema II.5. *The bipartite undirected graph G has a perfect matching if and only if the Tutte's matrix A_G is non singular.* □

The proof is intuitive. Indeed expression II.1 is different from zero if at least one of its terms is different from zero and this happens only when the corresponding permutation σ correspond to a perfect matching of G. Moreover notice that, since two permutations differ by at least a pair of elements, non null terms in II.1 cannot cancel.

We can apply this result to obtain a randomized algorithm for the Perfect Matching problem. First, we can notice that we can analyze the existence of a perfect matching by applying POLY-TEST to the determinant of A_G that is:

$$det(A_G) = \sum_{\sigma \in \Sigma} sgn(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)}$$

This is a polynomial in n^2 variables and degree equal to n.
This algorithm can also be used to determine a perfect matching of the graph G. The idea is to remove an edge of G at each iteration and to verify if the new graph has a perfect matching with the test described above. If a Perfect Matching doesn't exists, it means that the removed edge was essential in the solution and therefore it is added to the solution.

Algoritmo II.13 RAND-PERFECT-MATCHING(A,p,r)

```
1: M ← 0
2: while (M is not a perfect matching) and (G ≠ 0) do
3:    select (i, j) ∈ E
4:    G′ ← G\{(i, j)}
5:    Test if G′ has a perfect matching (using POLY-TEST)
6:    if NO perfect matching then
7:       M ← M ∪ {(i, j)}
8:    G ← G′
```

Since the algorithm performs an iteration for each edge of the graph G, its complexity is $O(mn^2)$ where m is the number of edges and n^2 is the number of variables of the

polynomial associated to the determinant. It's also important to notice that this is a Montecarlo algorithm and so it can make mistakes in its execution. The technique of executing the algorithm on the same input various times to decrease the error rate is applicable in this case too.

II.4.5 Fingerprint of a file

The fingerprint is a bit string of fixed length that identifies a certain file. It is used to guarantee the authenticity and security of a file and to quickly identify files distributed on the Web.

Let us consider the situation of two users A and B that are distant from each other, for example let us suppose that they are separated by the Internet. Each of them has a file. The problem is to efficiently check whether the file that is stored on the computer of the two users is the same. Schematically:

- user A holds a file $a = a_0, \ldots, a_n$
- user B holds a file $b = b_0, \ldots, b_n$
- n is very big

A straightforward algorithm could be: A sends a to B and B checks if the received a is equal to b using a bitwise comparison. This is clearly not efficient because sending a big file over the network takes a lot of time. A randomized efficient solution could be the one listed in II.14 that uses the concept of fingerprint.

Algoritmo II.14 FILE-EQUALITY-ON-NETWORK()

```
1: A chooses a prime number p ∈ {2, ..., T}
2: A computes F_p(a) = a mod p
3: A sends to B p and F_p(a)
4: B computes F_p(b) = b mod p
5: if F_p(b) ≠ F_p(a) then
6:     return a and b are surely different
7: else
8:     return a and b may be equal
```

The drawback of the algorithm is that every pair of files such that $|a - b| \bmod p = 0$ are false positive. Let us calculate the probability of choosing a prime number p that divides $|a-b|$. Considering that $|a-b|$ is a number of n bits and that a well known result states that the number of prime numbers that divide a number of n bits is at most n, we can say that:

$$Pr\{error\} \leq \frac{n}{\Pi(T)}$$

where $\Pi(T)$ is the number of primes less than or equal to T. In order to calculate explicitly the probability of error, some mathematical results are needed:

(i) $\Pi(T) \sim \frac{x}{\ln x}$ when $x \to \infty$

(ii) $\frac{x}{\ln x} \leq \Pi(x) \leq \frac{1.26 \cdot x}{\ln x}$ if $x \geq 17$

(iii) the number of primes that divide a number of n bits is at most $\Pi(n)$

Considering the result (i), the previous bound on the probability of error can be rewritten as follows:

$$Pr\{error\} \leq \frac{n}{\Pi(T)} \leq \frac{n \ln T}{T}$$

If we consider also the result (iii) we can write that the probability of error satisfies:

$$Pr\{error\} \leq \frac{\Pi(n)}{\Pi(T)} \leq \frac{1.26n}{\ln n} \frac{\ln T}{T}$$

If for example we choose $T = c \cdot n$ then we can write:

$$Pr\{error\} \leq \frac{1.26n}{\ln n} \frac{\ln(c \cdot n)}{c \cdot n} \leq \frac{1.26(\ln c + \ln n)}{c \ln n} = \frac{1.26}{c}\left(1 + \frac{\ln c}{\ln n}\right)$$

where $\ln n$ at the denominator shows that the error gets smaller as the number of bits n of the file gets higher.

Consider a numerical example, where $n = 2^{23}, T = 2^{32}$. If we apply the relation previously found we can write that:

$$Pr[error] \leq \frac{1.26n}{\ln n} \frac{\ln T}{T} = \frac{1.26 \cdot 2^{23}}{\ln 2^{23}} \frac{\ln 2^{32}}{2^{32}} \approx 0.0035$$

II.4.5.1 Application to Pattern Matching

Let us consider an application of the technique of fingerprint. The problem of pattern matching consists in finding a word $y = y_1, y_2, \ldots, y_m$ into a long text $x = x_1, x_2, \ldots, x_n$, where $n \gg m$. The straightforward solution to the problem is presented in algorithm II.15. The complexity of the algorithm is $O(mn)$, since there are $O(n)$ iterations of the

Algoritmo II.15 STRAIGHTFORWARD-PATTERN-MATCHING(x,y)

```
1: for i ← 1 to n − m + 1 do
2:    x(i) ← substring of x of size m starting from position i
3:    if x(i) = y then
4:       print("substring found")
```

loop, each of one requires a comparison between two strings of size m. This result is not very good, considering that there is a deterministic algorithm that solves the problem of pattern matching with a complexity of $O(m + n)$.

We can solve the problem in a more efficient way, using a randomized approach and the technique of fingerprint. Instead of comparing each substring of x with y, we calculate the fingerprint of the substring of x and compare it with the fingerprint of y. If we find

Algoritmo II.16 KARP-RABIN(x,y)

```
1: choose a prime number p ∈ {2, ..., T}
2: F_p(y) ← y mod p
3: for j ← 1 to n − m + 1 do
4:    x(j) ← substring of x of size m starting from position j
5:    F_p(x(j)) ← x(j) mod p
6:    if F_p(x(j)) = F_p(y) then
7:       print("substring found")
```

that the two fingerprints are equal then, according to what said before, it may be that y and the substring of x are equal, otherwise they are surely different. The algorithm is formalized in II.16. Notice that algorithm II.16 is a Montecarlo algorithm, that means that even if the algorithm finds that there is a matching, it may be a false positive. Since the calculation and the comparison of two fingerprints can be made in constant time, the complexity of the algorithm is now $O(n)$. There is also an improvement that allows us not to recalculate from scratch the fingerprint at every iteration of II.16. In fact:

$$x(j+1) = 2\left(x(j) - 2^{m-1}x_j\right) + x_{j+m}$$

$$F_p(x(j+1)) = \left(2\left(F_p(x(j)) - 2^{m-1}x_j\right) + x_{j+m}\right) \ mod\ p$$

Let us calculate the probability of error of KARP-RABIN algorithm. We can write that:

$$Pr\{error\} \leq n \cdot \frac{\Pi(m)}{\Pi(T)}$$

in fact, the probability of error during a single iteration of the algorithm is $\frac{\Pi(m)}{\Pi(T)}$, because there are $\Pi(m)$ (remark result (iii)) possible "wrong primes" among the possible $\Pi(T)$ primes that can be chosen. Since the loop in the algorithm is repeated n times the previous probability is multiplied by n. In order to refine our bound on the error, we observe that we have an error when the m bit number $|y - x(j)|$ is divisible by p, for some j. That is equivalent to say that we have an error when the number

$$h = \prod_{j=1}^{n-m} |y - x(j)|$$

is divisible by p. Since h is the product of $O(n)$ numbers of m bits, it is a number of $O(nm)$ bits. So the probability of error becomes:

$$Pr\{error\} \leq \frac{\Pi(nm)}{\Pi(T)}$$

where the factor n is now at the argument of $\Pi(\cdot)$.

The drawback of KARP-RABIN is that it is a Montecarlo algorithm. A possible way to turn it into a Las Vegas one could be making a full bitwise comparison when the fingerprint of the substring of x and that of y coincide, in order to avoid the possibility of false positive. The complexity of this approach is in the worst case $O(mn)$, as in the algorithm II.15.

II.5 The probabilistic method

The probabilistic method is a non constructive [1] method pioneered by Paul Erdõs for proving the existence of a prescribed kind of mathematical object. It works by showing that if one randomly chooses objects from a specified class, the probability that the result is of the prescribed kind is more than zero. Although the proof uses probability, the final conclusion is determined for certain, without any possible error.

Let us apply the method to the problem of *Max Cut*. Given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, we want to find a binary partition (V_1^*, V_2^*) of V such that

$$(V_1^*, V_2^*) = \underset{(V_1,V_2)}{\arg\max} \left|\{(i,j) \in E \mid i \in V_1,\ j \in V_2\}\right|$$

that is we want to find the partition of the nodes that determines the maximum cut. It can be shown that the problem is $\mathcal{NP}$-hard. However, we can show with the probabilistic method that

$$C = \left|\{(i,j) \in E \mid i \in V_1^*,\ j \in V_2^*\}\right| \geq \frac{|E|}{2}$$

that is the maximum cut C has at least $\frac{|E|}{2}$ arcs. Let us construct a random partition (V_1, V_2) where each node belongs to V_1 with probability $\frac{1}{2}$ and to V_2 with the same probability $\frac{1}{2}$. In other words, $\forall v \in V$ we assign v to V_1 with probability $\frac{1}{2}$ otherwise we assign it to V_2. We are interested in finding what is the mean number of arcs that belongs to the cut determined by such a random method of partitioning. For every $e \in E$ we construct an indicator random variable such that:

$$X_e = \begin{cases} 1, & \text{if } e \text{ belongs to the cut} \\ 0, & \text{otherwise} \end{cases}$$

We also call X the number of arcs of the cut induced by the random partition (V_1, V_2):

$$X = \sum_{e \in E} X_e$$

We can say that:

$$E[X_e] = Pr(e \in cut) = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$$

because the probability of the event that an edge e belongs to the cut can be split into two events: one event refers to the case when the first end of arc e belongs to V_1 and the other end to V_2 and the second event refers to the case when the two ends are reversed. Considering what we have just found and the linearity of the expected value we can write:

$$E[X] = E\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} E[X_e] = \frac{|E|}{2}$$

[1] A constructive proof is a method of proof that demonstrates the existence of a mathematical object by creating or providing a method for creating such an object

We have found that, extracting a random partition, the average number of arcs of the induced cut is $\frac{|E|}{2}$. This proves the existence of a cut in the graph with at least $\frac{|E|}{2}$ arcs, because otherwise the mean could not be $\frac{|E|}{2}$. We say that the calculated mean value is a *witness* of the fact that it exists a cut of at least $\frac{|E|}{2}$ arcs. Since it exists a cut with at least $\frac{|E|}{2}$ arcs, also the maximum cut C has at least $\frac{|E|}{2}$ arcs.

In the following section we will see another example of probabilistic method in order to prove properties of the mathematical object we are dealing with.

II.5.1 3-SAT problem

We can apply the probabilistic method that we explained in the above section to solve the $\mathcal{NP}$-hard problem of satisfiability. Given a boolean formula, the satisfiability problem (SAT) consists in verifying if there exist an assignment to the n boolean variables $x_1, \dots, x_n$, such that the formula is true. Let us consider formulas in the form $F = C_1 \wedge C_2 \wedge \dots \wedge C_K$, where every clause C_i is composed by exactly k *literals* which are either boolean variables or negation of boolean variables, in OR to one another: $C_i = \pm x_{i_1} \vee \pm x_{i_2} \dots \pm x_{i_k}$. Such formula is called *k-conjunctive normal form*, or k-CNF. The problem of finding an assignment to formulas in the k-CNF is called k-SAT. For example, the formula (2-CNF): $(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ is satisfiable, because the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 0$ makes the formula true.

Polynomial algorithms are known for 2-SAT, but not for 3-SAT. 3-SAT is a $\mathcal{NP}$-complete problem.

The problem of MAX-3-SAT consists in finding an assignment that maximize the number of satisfied clauses. Obviously, if the result of MAX-3-SAT coincide with the number of total clauses in the formula, the corresponding 3-SAT is satisfiable, otherwise it's not.

II.5.1.1 Randomized algorithm for MAX-3-SAT

The algorithm is straightforward: it assigns randomly T or F to each of the n variables according to uniform distribution.

While in the algorithms described in the past sections we analyzed the complexity of finding the optimal solution, in this case we find the performance of an algorithm, not necessarily optimal, in terms of quality of the solution, maintaining the complexity fixed. In order to improve the probability of generating an optimal solution, one possible method is to make multiple runs of the algorithm, generating a certain number of solutions.

Let us define $Z_i = I(C_i = V)$ the indicator random variable that is 1 if the i-th clause is satisfied by the assignment and 0 otherwise. The number of satisfied clauses is $Z = \sum_{i=1}^{K} Z_i$. Since every clause is obtained by the OR of three independent terms, the probability that a clause is not satisfied is:

$$\Pr\{C_i = F\} = \Pr\{L_{i_1} = F\}\Pr\{L_{i_2} = F\}\Pr\{L_{i_3} = F\} = \left(\frac{1}{2}\right)^3 = \frac{1}{8}.$$

Where L_{i_1}, L_{i_2} and L_{i_3} indicate the three literals of the clause C_i. So:

$$E[Z_i] = \Pr\{C_i = V\} = 1 - \Pr\{C_i = F\} = \frac{7}{8}.$$

The average number of clauses that are satisfied by a random assignment is:

$$E[z] = \sum_{i=1}^{K} E[Z_i] = \frac{7}{8}K.$$

This enables us to find a stronger result regarding the optimal solution of MAX-3-SAT. Considering the result regarding the expected value of Z, among all the realization x_i, with $i = 1 \ldots n$, there surely exits one for which the number of clauses that are satisfied is $\lceil \frac{7}{8}K \rceil$ (because $\frac{7}{8}K$ is the expected value). It should be noticed that this result is deterministic: in every 3-SAT problem it is always possible to find an assignment to the variables that makes true at least the $7/8$ of the clauses. This implies that for $K \leq 7$ every formula is satisfied. The proof just presented is another example of the probabilistic method.

Now we calculate how many iteration, in the average case, are necessary to find an assignment that satisfies at least $\frac{7}{8}K$ clauses. Let us call p_j the probability that an assignment satisfies exactly j clauses and such that $p = \sum_{j \geq \frac{7}{8}K}^{K} p_j$.

By definition $E[Z] = \sum_{j=0}^{K} j \cdot p_j$. So we can write:

$$\sum_{j=0}^{K} i \cdot p_j = \frac{7}{8}K.$$

Dividing the summation in two parts:

$$\sum_{j<\frac{7}{8}K} j \cdot p_j + \sum_{j \geq \frac{7}{8}K} j \cdot p_j = \frac{7}{8}K.$$

Let K' be the greater integer $< \frac{7}{8}K$. We can find an upper bound to the formula, substituting to the term j the maximum value that it can assume, K' and K respectively in the two summations:

$$\frac{7}{8}K \leq \sum_{j<\frac{7}{8}K} K' p_j + \sum_{j \geq \frac{7}{8}K} K p_j = K'(1-p) + Kp \leq K' + Kp$$

from which:

$$Kp \geq \frac{7}{8}K - K' \geq \frac{1}{8}.$$

Taking in count that K' is the greater integer less than $\frac{7}{8}K$ we can write a *lower bound* for p:

$$p \geq \frac{1}{8k}.$$

The average number of random generations that should be conducted to find and assignment that satisfies at least $\frac{7}{8}K$ clauses is $8K$.

II.6 Randomization in on-line settings

Let us preview some issues coming from the on line setting. In this setting the input is revealed while the algorithm is executing and decisions must be taken upon the arrival of each input element.

II.6.1 Cache memory management

The cache is a small and fast memory which stores copies of the data from the most frequently used main memory locations. When it is needed to read from or write to a location in main memory, it is first checked whether a copy of that data is in the cache. If so ((hit)), data is loaded from the cache, otherwise ((miss)) the page is requested to the much slower main memory. The space in the cache is limited; let us indicate k the number of pages that can be simultaneously hosted by the cache and with σ the sequence of pages requested. When it is requested a page that is not in the cache, to make room for it another page has to be removed (and its content possibly copied to main memory).

The problem that we face is to define a replacement policy of the pages in the *cache* as new requests arrive; the aim is to find a sequence of replacements that minimize the number of cache *miss*.

There are two types of algorithms, the first type consist in replacing one page at time (*reduced* algorithms) while the second enables to replace a group of pages (*non reduced* algorithms). It can be shown that for every *non reduced* algorithm there is an equivalent *reduced* one. The algorithm Farthest-in-Future consists in evicting the page that has its next access farthest into the future. The algorithm is optimal, but requires the knowledge of the sequence of future requests. This knowledge is not possible in real cases, for which *on-line* algorithms are needed, in order to establish a replacement policy based only on past informations. An example of *on line* algorithm is LRU (Least Recently Used), that consists in replacing the page that has been used least recently, among those that are currently in the *cache*.

Online algorithms will be the topic of a subsequent chapter. Let us anticipate the analysis of a particular class of algorithms known as *marking algorithm* to apply the techniques of analysis of randomized algorithms. The class of marking algorithms includes the LRU algorithm.

II.6.1.1 Marking algorithms

The basic idea is replacing the pages that have not been recently used. The algorithm is divided in phases. Every iteration of the loop **repeat** 2 encloses a phase (i is the counter of the number of phases. At the beginning of every phase the pages are all unmarked. When a page is requested it is marked; if it is not in the cache it is inserted, replacing a page that is not marked (that has not already been requested during the current phase). When a new request arrives and all pages are marked, the request is inserted again at the head of the request sequence, all pages are unmarked and a new phase begins.

Algoritmo II.17 MARKING ALGORITHM(σ)

```
1:  i ← 0
2:  repeat
3:     unmark all pages
4:     stop ← FALSE
5:     repeat
6:        accept the request of page s
7:        mark s
8:        if s is not in the cache and ∃s' unmarked in the cache then
9:           s ↔ s'
10:       else
11:          if s is not in the cache then
12:             stop ← TRUE
13:             insert s again at the head of the sequence σ
14:    until stop
15:    i ← i + 1
16: until σ ≠ ∅
```

The previous description refers to a class of algorithms and not to a specific one; in fact is not specified how the page S' to be replaced is chosen among the unmarked ones. In the LRU policy it is chosen the "oldest" page among those that are not marked.

To analyze the performance of the algorithm we add for convenience:

- a preamble phase, where all pages that are initially in the cache are requested one time;
- a final phase where all the pages that are in the cache are requested two times, in a *round robin* manner; at the end of this phase every page will be in the cache, and will be marked.

The addition of these two steps does not modify the cost and the performance of the algorithm.

As is it structured the algorithm, the portion of the sequence σ considered in every phase consists exactly in k distinct elements. The subsequent phase starts with the $k+1$-th element. Let r be the total number of phases. To evaluate the performance of the algorithm of the cache management is is important to evaluate an upper and a lower bound to the number of *misses* that occurs for every sequence σ of requests.

Upper bound An upper bound to the number of misses can be obtained considering that, taking into consideration what was said before, during every phase, if a page is requested, it is not replaced until the next phase. So the maximum number of misses that we can have during a phase is given by the capacity of the cache, and so k. The maximum number of miss is:

$$ub(\sigma) = kr$$

Lower bound Considering what we said before, in order to estimate the optimal number of miss, we have to consider the FARTHEST-IN-FUTURE algorithm. This algorithm

can be subdivided into phases with the same criteria seen before. After the first request to a certain page s, FARTHEST-IN-FUTURE keeps the page into the cache for the whole phase. A new phase begins when there is a miss. So the minimum number of misses is given by:

$$lb(\sigma) \geq r - 1$$

Performance of the MARKING ALGORITHM Comparing $ub(\sigma)$ with $lb(\sigma)$ we can limit the loss of *competitiveness* of the MARKING ALGORITHM as opposed to the optimal algorithm that can see into the future.

$$ub(\sigma) = kr = k(r-1) + k \leq klb(\sigma) + k$$

We can conclude that MARKING ALGORITHM has a performance at most k time worse than the optimal FARTHEST-IN-FUTURE.

As stated before LRU is a particular case of the MARKING ALGORITHM, in which the choice of the page to be replaced among the unmarked ones is always on the page that has been used least recently. A drawback of this algorithm is that it is possible to create a pathological case; let us imagine a case where a program has to cyclically access to $k+1$ distinct pages, always in the same order. In this case the behavior of the LRU algorithm will be very bad, in fact after the first k requested pages there will be always a *miss*.

II.6.1.2 A randomized marking algorithm

Let us consider a randomized algorithm, that randomly chooses the page to be replaced among those that are not marked and let us evaluate the performance trying to approach *lower bound* and *upper bound* of the number of *misses*.

To make an accurate analysis let us divide the pages that can be requested in every phase into two categories:

- "fresh" pages, those pages that were not marked in the previous phase;
- "old" pages, those pages that were marked in the previous phase.

Every phase begins with all the pages unmarked, and contains accesses to k different pages, each of one is marked the first time it is requested during the phase. Among this k accesses to unmarked pages, we call c_j the number of accesses to fresh pages.

Let us call $f_j(\sigma)$ the number of *miss* encountered by the optimal algorithm during the phase j. We know that in every phase j there are requests to k different pages, and we know that in phase $j+1$ there are requests to c_{j+1} other pages. So during the phase j and $j+1$ there a total of $k + c_{j+1}$ requests to different pages. This means that the optimal algorithm must generate al least c_{j+1} *miss* during the phases j and $j+1$, so $f_j(\sigma) + f_{j+1}(\sigma) \geq c_{j+1}$. This also holds for $j = 0$ because the optimal algorithm generates c_1 *miss* during the phase 1. So we have:

$$\sum_{j=0}^{r-1}(f_j(\sigma) + f_{j+1}(\sigma)) \geq \sum_{j=0}^{r-1} c_{j+1}$$

The left-hand side of the inequality is at most $2\sum_{j=1}^{r} f_j(\sigma) = 2lb\sigma)$, while the right hand side is equal to $\sum_j c_j$, so we can write:

$$lb(\sigma) \geq \frac{1}{2}\sum_j c_j.$$

We have found a more precise estimate of the *lower bound* for the number of *misses* in which also an optimal algorithm must incur; we can now find an *upper bound* to the expected value of *misses* generated by the randomized marking algorithm, also expressed in terms of number of fresh pages in every phase.

Let us consider the phase j: in this phase there are k requests to unmarked pages, that we can distinguish in c_j requested directed to fresh pages and $k - c_j$ to old pages. Call X_j the random variable indicating the number of *miss* of the randomized algorithm in phase j. Every request to a fresh page causes a *miss* in the randomized algorithm, because by definition fresh pages were not marked in the previous phase and so they cannot be found in the cache in the current phase. Regarding the old pages, the problem is more delicate. At the beginning of every phase there are k old pages in the cache, the ones that were unmarked to begin the new phase. When an old page s is requested we have a *miss* if the page has been already replaced in the current phase; let us calculate the probability that the i-th to an old page leads to a *miss.* We suppose that in the current phase $c < c_j$ requests to fresh pages have already been submitted; so the cache contains c ex-fresh pages that are now marked, $i-1$ old pages that are now marked, and $k-c-i+1$ pages that are old and that have not already been marked in the current phase. There are in total $k - i + 1$ that were old and that have not been marked in the current phase; since $k - c - i + 1$ of these are in the cache, the remaining c pages are not in the cache any more. Each of the $k - i + 1$ old pages has the same probability of not being in the cache anymore, and so s is not in the cache with probability $\frac{c}{k-i+1} \leq \frac{c_j}{k-i+1}$; this is the probability of having a *miss* when s is requested.

Making the summation of all the unmarked pages we have:

$$E[X_j] \leq c_j + \sum_{i=1}^{k-c_j} \frac{c_j}{k-i+1} \leq c_j \left(1 + \sum_{l=c_j+1}^{k} \frac{1}{l}\right) = c_j(1 + H(k) - H(c_j)) \leq c_j \cdot H(k),$$

where the term $H(k)$ is the harmonic series truncated at the k term: $H(k) = \sum_{i=1}^{k} \frac{1}{i}$.

In conclusion, calling M_σ the random variable related to the number of *misses* encountered during the algorithm, for every sequence of requests σ the following *upper bound* for the expected number of *misses* holds:

$$E[M_\sigma] \leq H(k)\sum_{j=1}^{r} c_j.$$

Combining the *lower bound* with the *upper bound* previously found, expressed in terms of number of fresh pages requested in every phase, we can conclude that the expected number of *misses* encountered by the randomized algorithm is at most:

$$E[M_\sigma] \leq H(k) \sum_{j=1}^{r} c_j = 2H(k) \cdot lb(\sigma) = O(\log k) \cdot lb(\sigma).$$

So the performance of the mean case of the randomized marking algorithm is at most a logarithmic factor worse than the optimal algorithm. Thus, we have found an improvement. When we will deal with online algorithms, we will see that this improvement depends on the opponent behavior, that is it depends on the characteristics of the input sequence and on the way in which it can be varied by the choices of the algorithm.

II.6.2 The "engagement ring" problem

Let us analyze a variant of the hiring problem described at the beginning of the chapter and let us study an online variant.

Problem: there are n possible girlfriends but only an engagement ring is available; the ring can be given only one time and once a girl is met it is necessary to immediately decide whether or not to give the ring to her (for the sake of simplicity let us suppose that a girl does never refuse our proposal). The problem consists in choosing the girlfriend that match to the maximum according to a unique evaluation criterion.

Intuitive solution: form an opinion with the first candidates, and then continue to meet other girls and choose the one that is better than all the ones met so far; in case that no one is better we choose the last one.

Call n the number of girls, the following are the steps of the algorithm:

1. evaluation of the first k candidates
2. *best* $\leftarrow$ the best of the first k
3. choose the one that outperform *best* among the $k+1 \dots n-1$
4. if no one outperforms *best* choose the n-th.

The problem consists in finding the value of k that maximizes the probability of choosing the best girl. Call $M(j)$ the maximum "score" of the best girl among the first j: $M(j) = max\{score(i),\, i = 1 \dots j\}$ and let us define two events:

- S = event "choose the best";
- S_i = event "choose the best when she is in position i" .

Since the events S_i are all mutually exclusive, the probability of finding the best girl is:

$$Pr\{S\} = \sum_{i=1}^{n} Pr\{S_i\}.$$

If the best girl is among the first k, she will never be chosen: $Pr\{S_i\} = 0$, $i = 1 \dots k$. If she is in a subsequent position, the event S_i can be rewritten as:

$$S_i = B_i \wedge O_i$$

where:

- B_i = event "the best girl is in position i";
- O_i= event "the score of all the candidates from $k+1$ to $i-1$ is less than *best*".

The first event has probability $1/n$ for every i, assuming that the position of the best girl is random: $\Pr\{B_i\} = \frac{1}{n}$.
The second event corresponds to the probability that the best girl, among the first $i-1$, can be found in the first k positions: $Pr\{O_i\} = \frac{k}{i-1}$ The events B_i e O_i are independent, so:

$$Pr\{S_i\} = Pr\{B_i\} \cdot Pr\{O_i\} = \frac{k}{n(i-1)}$$

The probability of finding the best girl is:

$$Pr\{S\} = \sum_{i=k+1}^{n} \frac{k}{n(i-1)} = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}$$

The value of the summation is included between the two integrals:

$$\int_{k}^{n} \frac{1}{x} dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x} dx$$

Integrating the previous equation:

$$\frac{k}{n}(\ln n - \ln k) \leq Pr\{S\}$$

To find the value of k that maximize the first term of the equation we calculate the first derivative and we equal it to zero:

$$\frac{1}{n}(\ln n - \ln k - 1) = 0$$

from which we obtain:

$$\ln k = \ln (n-1)$$

and last the optimal value of k:

$$k = \frac{n}{e}$$

With ten girls the optimal value k is $\frac{10}{e} \simeq 3.6$. We can decide to try the first three or four girls and choose from the remainder the one that outperforms them.

Capitolo III

Tecniche di valutazione di complessità ammortizzata

...