

# Giorno 1

## 1 Notazioni asintotiche

**Def. 1.1.** Date due funzioni  $f(n)$  e  $g(n)$ , si dice che  $f(n) = O(g(n))$  se esistono  $c \neq 0$  ed  $n_0$  tali che:

$$\forall n > n_0 : 0 \leq f(n) \leq cg(n)$$

**Def. 1.2.** Date due funzioni  $f(n)$  e  $g(n)$ , si dice che  $f(n) = \Omega(g(n))$  se esistono  $c \neq 0$  ed  $n_0$  tali che:

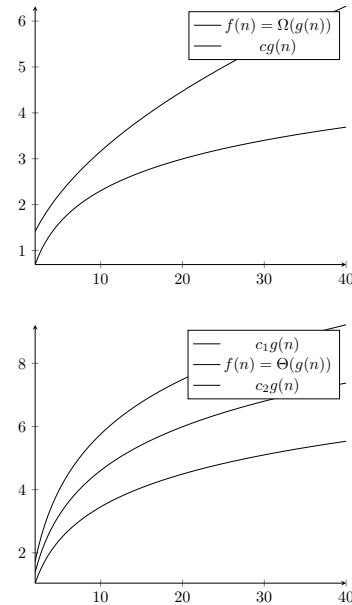
$$\forall n > n_0 : 0 \leq cg(n) \leq f(n)$$

**Def. 1.3.** Date due funzioni  $f(n)$  e  $g(n)$ , si dice che  $f(n) = \Theta(g(n))$  se

$$f(n) = O(g(n)), f(n) = \Omega(g(n))$$

Ossia se:

$$\exists c_1, c_2 \neq 0, n_0 : \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$



## 2 Limiti Superiori ed Inferiori

### 2.1 Definizioni

**Def. 2.1.** Dato un problema  $\Pi$ , se esiste un algoritmo  $A$  che risolve  $\Pi$  in tempo  $t(n)$ , allora  $t(n)$  è un *limite superiore* alla complessità in tempo di  $\Pi$ .

**Def. 2.2.** Dato un problema  $\Pi$ , se ogni algoritmo  $A$  che lo risolve deve impiegare necessariamente almeno tempo  $t(n)$ , allora  $t(n)$  è un *limite inferiore* alla complessità in tempo di  $\Pi$ .

**Def. 2.3.** Dato un problema  $\Pi$ , se un algoritmo  $A$  che lo risolve impiega tempo  $t(n)$ , e  $t(n)$  è un limite inferiore alla complessità in tempo di  $\Pi$ , allora si dice che  $A$  è un *algoritmo ottimo*.

### 2.2 Metodi per individuare limiti inferiori

1. **Dimensione dei dati:** Se per risolvere un problema si deve necessariamente accedere a tutti i dati in input, la loro dimensione è un limite inferiore alla complessità in tempo del problema.

*Esempio:* Per cercare l'elemento massimo di un array  $A$  di dimensione  $n$  si deve scansionare l'intero  $A$  (il minimo potrebbe essere in ognuna delle  $n$  posizioni).

2. **Eventi contabili:** Se al fine della risoluzione del problema è necessario che un particolare evento si verifichi un numero  $f(n)$  di volte, allora  $f(n)$  è un limite inferiore alla complessità in tempo del problema.

*Esempio:* Se voglio generare le permutazioni di un insieme di  $n$  numeri, devo, ovviamente, generare *ogni* permutazione. La generazione di una delle  $n!$  permutazioni è perciò un evento contabile, e  $n!$  è un limite inferiore alla complessità del problema.

3. **Albero di decisione:** Si rappresentano tutte le possibili "decisioni" nei nodi interni di un albero. I possibili esiti di ogni nodo-decisione sono i suoi figli ed i risultati sono rappresentati nelle foglie. L'altezza dell'albero è un limite inferiore alla complessità del problema.

*Esempio:* Il limite inferiore per la ricerca binaria dell'elemento  $k$  su un array di  $n$  elementi è uguale all'altezza di un albero binario con  $n$  foglie ( $\log n$ ), poiché ogni scelta ha due possibili esiti (due figli), ed ognuno degli  $n$  elementi può essere  $k$ .

## 3 Analisi di algoritmi

### 3.1 Ricerca

#### 3.1.1 Ricerca sequenziale

Sia  $A$  un array di lunghezza  $n$ . Devo cercare la chiave  $k$ .

```
1 SeqSearch(A, k):
2   for(i = 1 to n):  $O(n)$ 
3       if(i == k) return i;
4   return -1;
```

Costo al caso ottimo  $O(1)$ ;

Costo al caso pessimo  $O(n)$ , algoritmo ottimo.

#### 3.1.2 Ricerca binaria

Sia  $A$  un array **ordinato** di lunghezza  $n$ .  
Devo cercare la chiave  $k$ .

```
1 BinSearch(A, k):
2   p = 1, r = n + 1;
3   while(p <= r):
4       q =  $\left\lfloor \frac{p+r}{2} \right\rfloor$ ;
5       if(A[q] == k) return
6           q;
7       if(k > A[q]) p = q;
8       else r = q;
9   return -1;
```

Costo al caso ottimo  $O(1)$ ;

Costo al caso pessimo  $O(\log n)$ , ottimo.

### 3.2 Insertion, Selection Sort

#### 3.2.1 Insertion Sort

```
1 InsSort(A):
2   for(j = 2 to n):
3       k = A[j];
4       i = j - 1;
5       while(i > 0 && A[i] >
6           k):
7           A[i+1] = A[i];
8           i--;
9       A[i+1] = k;
```

**Invariante di ciclo:** Ad ogni iterazione si ha che i primi  $j - 1$  elementi sono ordinati.

1. Inizializzazione: Un solo elemento è banalmente ordinato
2. Conservazione: Informalmente – si inserisce il  $j$ -esimo elemento nel sottoarray  $A[1, \dots, j - 1]$ .
3. Alla fine il sottoarray corrisponde all'intero array, che è perciò ordinato.

Costo al caso ottimo:  $O(n)$ ;

Costo al caso pessimo:  $O(n^2)$ , non ottimo.

#### 3.2.2 Selection Sort

```
1 SelSort(A):
2   for(i = 1 to n):
3       posmin = i;
4       for(j = i+1 to n):
5           if(A[j] < A[
6               posmin]):
7               posmin = j;
8       if(A[i] > A[posmin]):
9           tmp = A[i];
10          A[i] = A[posmin];
11          A[posmin] = tmp;
```

**Invariante di ciclo:** Ad ogni iterazione si ha che i primi  $i - 1$  elementi sono ordinati e sono nella loro posizione finale

Costo al caso ottimo:  $O(n^2)$ ;

Costo al caso pessimo:  $O(n^2)$ , non ottimo.

1. Il sottoarray vuoto è banalmente ordinato

2. Conservazione: Ad ogni iterazione si prende il minimo del sottoarray  $[i, \dots, n]$  e lo si scambia con l' $i$ -esimo elemento.

Il sottoarray  $A[1, \dots, i - 1]$  è ordinato per ipotesi induttiva, e dato che ogni suo elemento è nella sua posizione finale si ha che tutti gli elementi di  $A[i, \dots, n]$  sono maggiori di  $A[i - 1]$ . Se ad  $A[1, \dots, i - 1]$  si *appende* il minimo di  $A[i, \dots, n]$ , allora il sottoarray  $A[1, \dots, i]$  è necessariamente ordinato, e l'elemento  $A[i]$  è nella sua posizione finale (non esiste elemento  $< A[i]$  in  $A[i + 1, n]$ ).

3. Alla fine il sottoarray  $[1, i]$  corrisponde all'intero array, che è perciò ordinato.

## 4 Limite inferiore per l'ordinamento (per confronti)

### 4.1 Limite inferiore $n \log n$

$n \log n$  è limite inferiore per l'ordinamento.

*Dimostrazione.* Sia  $A$  un array di  $n$  elementi. Usiamo il metodo dell'albero di decisione per stabilire un limite inferiore per l'ordinamento, ed assumiamo w.l.o.g. che tutti gli elementi di  $A$  siano distinti.

1. Usiamo come confronto la relazione  $\leq$ . Questa ha due possibili esiti, perciò l'albero sarà binario.
2. Ogni algoritmo di ordinamento deve essere in grado di generare ognuna delle  $n!$  permutazioni degli elementi di  $A$ .

Perciò si ha che il numero di risultati  $l$  (rappresentati dalle foglie dell'albero di decisione) possibili è compreso tra  $n!$  e  $2^h$  (massimo di foglie per un albero binario):

$$\begin{aligned} n! &\leq l \leq 2^h \\ \implies \log(n!) &\leq \log(l) \leq h \\ (\log \text{ monotona continua}) &\implies h > \log(n!) \\ (\text{Approx di Stirling}) &\implies h > \Omega(n \log(n)) \end{aligned}$$

■