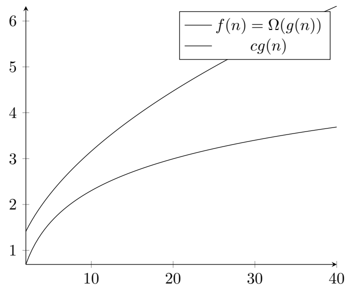
Giorno 1

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:

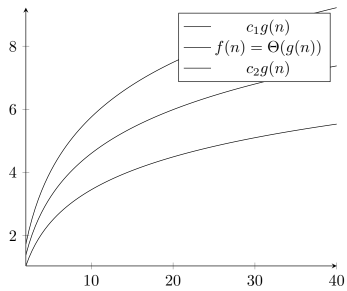
$$\forall n > n_0 : 0 \leq cg(n) \leq f(n)$$

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 Π, se esiste un algoritmo A che risolve Π in tempo $t(n)$, allora $t(n)$ è un *limite superiore* alla complessità in tempo di Π.

Def. 2.2. Dato un problema Π, 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 Π.

Def. 2.3. Dato un problema Π, se un algoritmo A che lo risolve impiega tempo $t(n)$, e $t(n)$ è un limite inferiore alla complessità in tempo di Π, 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.

```
SeqSearch(A, k):
        for(i = 1 to n): O(n)
            if(i == k) return i;
        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.

```
BinSearch(A, k):
        p = 1, r = n + 1;
        while(p ≤ r):
            q = ⌊(p+r)/2⌋;
            if(A[q] == k) return
                q;
            if(k > q) p = q;
            else r = q;
        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

```
InsSort(A):
        for(j = 2 to n):
            k = A[j];
            i = j-1;
            while(i > 0 && A[i] >
                k):
                A[i+1] = A[i];
                i--;
            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 $A[1, ..., j-1]$.

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

```
SelSort(A):
        for(i = 1 to n):
            posmin = i;
            for(j = i+1 to n):
                if(A[j] < A[
                    posmin])
                    posmin = j;
            if(A[i] > A[posmin]):
                tmp = A[i];
                A[i] = A[posmin];
                A[posmin] = tmp;
```

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

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

1. Il sottoarray vuoto è banalmente ordinato

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

   Il sottoarray $A[1, ..., i-1]$ è ordinato per ipotesi induttiva, e dato che ogni suo elemento è nella sua posizione finale si ha che tutti gli elementi di $A[i, ..., n]$ sono maggiori di $A[i-1]$. Se ad $A[1, ..., i-1]$ si *append*e il minimo di $A[i, ..., n]$, allora il sottoarray $A[1, ..., i]$ 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):

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

■