Appunti del corso di Algoritmi
A.A. 2023/2024

Prof. Roberto Segala
Appunti by Nick

Indice

Introduzione

1.1 Problema dell'elemento minimo di un array

1.1.1 Trovare l'elemento minimo di un array - ragionamento intuitivo

Prendiamo in esame l'algoritmo per trovare l'elemento minimo in un array di n elementi così definito:

```
n ← LENGTH[A]
m ← A[1]
FOR i ← 1 TO n
    IF A[i] < m
        m ← A[i]
RET m
```

Possiamo dire, in modo intuitivo, che il numero minimo di confronti da eseguire, per esser certi di aver trovato il minimo elemento dell'array è $n-1$.

1.1.2 Dimostrazione per assurdo

Supponiamo ci venga detto che un algoritmo può trovare il minimo elemento di un array in con meno di $n-1$ confronti. Se consideriamo i confronti fra gli elementi come a delle *gare* a coppie in cui 'vince' l'elemento più piccolo tra i due, possiamo dire che il minimo elemento dovrà essere l'elemento che non ha perso nemmeno una gara.

Consideriamo l'array **A**, contenente gli elementi B e C, con B elemento minimo. Se diamo come input all'algoritmo l'array **A**, otteniamo B come output, che è corretto. Ma sappiamo anche che C non ha mai perso una gara, quindi abbiamo due elementi che non hanno mai perso una gara. Se ora modifichiamo C nel seguente modo $C = B-1$, otteniamo che C diventa l'elemento minimo, ma se applichiamo l'algoritmo all'array, otterremo ancora B come elemento minimo.

Questo dimostra che **non è possibile** trovare il minimo di un array in meno di $n-1$ confronti.

1.2 Compessità

1.2.1 Concetto di Complessità

L'obiettivo dello studio degli algoritmi è quello di confrontare tra loro diversi algoritmi per determinare qualse sia il migliore caso per caso. Gli algoritmi possono essere confrontati su vari criteri:

- costo di esecuzione (memoria, cpu, ...)
- tempo di esecuzione
- costo economico

Analizzeremo la *complessità legata al tempo di esecuzione*, ma la stessa teoria è applicabile alla memoria o ad altri criteri di analisi degli algoritmi.

Il tempo di esecuzione può cambiare al variare della macchina su cui l'algoritmo viene eseguito; quindi, è inutile misurare il tempo fisico di completamento di un algoritmo. Per determinarne la velocità conteremo il numero di istruzioni, soprattutto quelle che hanno un impatto significativo sul tempo di esecuzione.

La *complessità* è dunque una funzione che prende in input la *dimensione di un problema* e fornisce in output il tempo di esecuzione di un algoritmo. Per sapere se la misura della dimensione del problema è corretta, lo possiamo evincere dai dai risultati ottenuti. Ad esempio, se consideriamo una matrice, potremmo considerare come dimensione del problema il numero di elementi della matrice, oppure potremmo considerare come dimensione del problema $num.righe * num.colonne$.

1.2.2 Alcuni esempi intuitivi

Consideriamo un algoritmo come un insieme di blocchi B_i di istruzioni messi in sequenza, e la complessità $C_i(n)$ di ogni blocco:

$$\begin{array}{c|c} B_1 & C_1(n) \\ B_2 & C_2(n) \\ B_3 & C_3(n) \end{array}$$

Potremmo calcolare la complessità di ogni blocco $C_i(n)$ e considerare la complessità totale come la sommatoria delle complessità:

$$\sum_i C_i(n)$$

Ma se l'algoritmo presentasse delle condizioni, per esempio

```
IF cond
    B1
ELSE
    B2
```

allora la complessità di un blocco condizionale diventerebbe $max(C_1(n), C_2(n))$. In alcuni casi, inoltre, la complessità della condizione dipende dal problema, ad esempio nel caso della ricorsione.

Analizziamo un altro algoritmo, contenente un ciclo:

```
WHILE cond
    B
```

In questo caso, la complessità sarà data dalla formula $C(B) * (numerointerazioni)$. Non possiamo sapere a priori il numero di iterazioni, ma possiamo ricavare un limite superiore al numero di volte che il ciclo while viene eseguito.

1.2.3 Limite inferiore, limite superiore

Esistono molti modi che ci portano a sovrastimare il numero di istruzioni che viene eseguito. Il modo per capire se la previsione è accurata è quello di trovare anche un *limite inferiore*. Quanto più vicini saranno il limite superiore e il limite inferiore, tanto più precisa è la previsione sulla complessità dell'algoritmo. Qualunque sia la stima che facciamo, siamo obbligati a fare approssimazioni, per eccesso o per difetto. In questi casi è necessario avere un limite superiore e un limite inferiore del *caso pessimo*. Non sarebbe utile calcolare il tempo medio, in quanto servirebbe una conoscenza probabilistica dell'input.

1.2.4 Trovare il minimo elemento di un array

Tornando sul problema della ricerca dell'elemento minimo di un array, proviamo a determinare la complessità dell'algoritmo precedentemente scritto:

```
n ← LENGTH[A]// 1 istruzione
m ← A[1]     // 2 istruzioni
FOR i ← 1 TO n          // 4*(n-1)+1+2*(n-1) istr.
    IF A[i] < m         // 2+2 istr.  (tutto il blocco if)
RET m         // 1 istruzione
```

Per come abbiamo contato le istruzioni necessarie per ogni riga dell'algoritmo, possiamo ottenere una formula di complessità di questo tipo:

$$1+6*(n-1)+2+1+1$$

Che possiamo semplificare in $6n-1$ e possiamo generalizzare, ignorando le costanti numeriche, in $an+b$, per qualche a e qualche b.

1.2.5 Confronto tra algoritmi

Avendo due algoritmi con complessità rispettivamente

- Alg. 1: $an+b$
- Alg. 2: an^2+bn+c

conviene sempre scegliere l'Alg. 1, questo perché se $n=1000$, nel caso dell'Alg. 1 stiamo moltiplicando la complessità per 1000, mentre nell'Alg. 2 la complessità viene moltiplicata per 1000000.

Quello che interessa, quindi, ai fini dell'analisi della complessità di un algoritmo è il *grado della funzione risultante*.

1.2.6 Prodotto fra due matrici

Analizziamo ora l'algoritmo che moltiplica due matrici:

```
MULT(A,B)
    n ← ROWS[A]
    m ← COLS[A]
    l ← COLS[B]
    FOR i ← 1 TO n:       // costo n*m*l*a
        FOR j ← 1 TO l:       // costo m*l*a
            C[i][j] ← 0
            FOR k ← 1 TO m     // costo m*a+b
                C[i][j] ← C[i][j] + A[i][j] * B[i][j]
    RET C
```

Possiamo notare che la complessità del ciclo più interno è data da un valore costante di istruzioni (riga 9) moltiplicato per il numero di iterazioni del ciclo (m), ottenendo quindi $m*a+b$, dove b è il numero di istruzioni legate all'esecuzione del ciclo (verifica condizione, incremento indice, ecc). Il ciclo centrale avrà una complessità legata a $m*a$ eseguito l volte, quindi otteniamo la formula $m*l*a$.

Infine, il ciclo più esterno ha come complessità la formula $n * m * l * a$, ottenuta in modo analogo alla precedente. Possiamo notare che il valore della costante a non è di nostro interesse, ma la cosa importante è che la complessità dell'algoritmo è lineare in n, l ed m.

Notazione asintotica

2.1 Notazione asintotica

Dalle funzioni di complessità ci interessa principalmente ricavare l'ordine di grandezza; ad esempio è importante sapere se al raddoppio della dimensionde del problema il tempo di completamento raddoppia oppure quadruplica. In alcuni casi il problema potrebbe avere complessità diverse sulla base di eventuali condizioni che si verificano durante l'esecuzione, ad esempio se viene eseguito il ramo `if` o il ramo `else`. In questi casi dobbiamo considerare il caso pessimo.
Esempio: la ricerca di un elemento in un array costa n, ma esiste un caso in cui costa 1, ovvero quando l'elemento che si sta cercando è il primo.

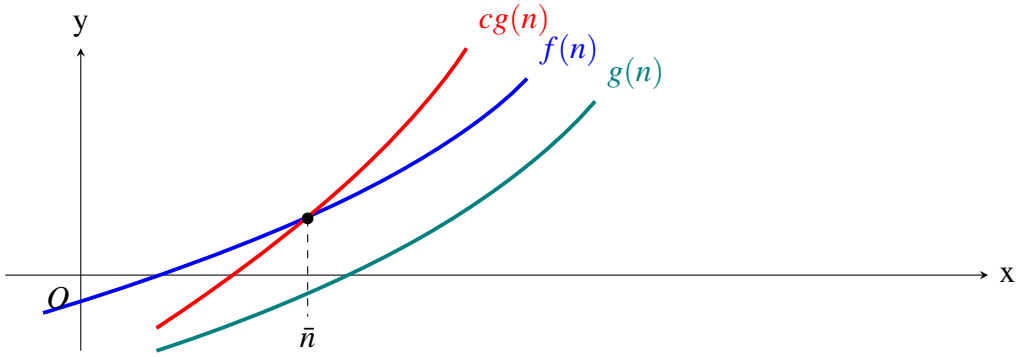
2.1.1 O-grande - $f \in O(g)$

Diciamo che una funzione f appartiene ad O-grande di g, $f \in O(g)$, quando f non cresce più in fretta di g da un certo punto in poi, a prescindere dalle costanti.

Definizione:

$$\exists c > 0\ \exists \bar{n} > 0\ \forall n > \bar{n}\ f(n) \leq cg(n)$$

Grafico:



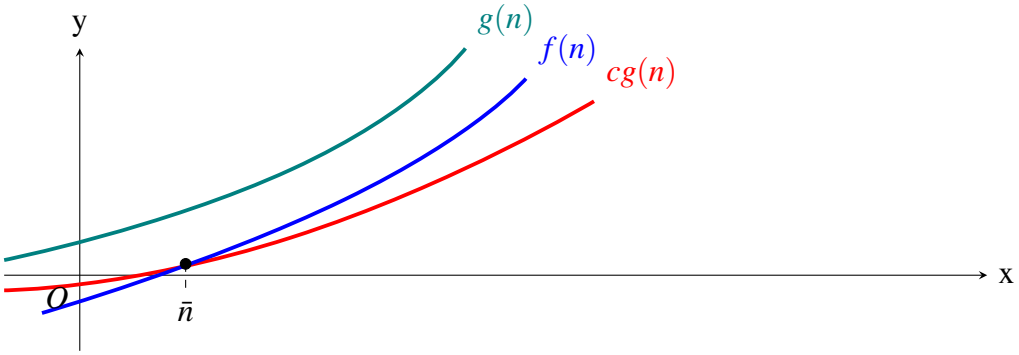
2.1.2 Omega - $f \in \Omega(g)$

Diciamo che una funzione f appartiene ad Omega di g, $f \in \Omega(g)$, quando da un certo punto in poi, a meno di una costante moltiplicativa, f non va mai al di sotto di g.

Definizione:

$$\exists c > 0 \; \exists \bar{n} > 0 \; \forall n > \bar{n} \; f(n) \geq cg(n)$$

Grafico:



2.1.3 Theta - $f \in \Theta(g)$

Una funzione f appartiene a Theta di g se e solo se f appartiene ad O-grande di g ed f appartiene a Omega di g. Formalmente scriviamo:

$$f \in \Theta(g) \leftrightarrow f \in O(g) f \in \Omega(g)$$

2.1.4 Esercizio

Dimostrare che $f \in O(g) \leftrightarrow g \in \Omega(f)$

Dimostrazione:

- Dimostriamo l'implicazione a destra: $f \in O(g) \rightarrow g \in \Omega(f)$. Usiamo la definizione:

  $$\exists c \; \exists \bar{n} \; \forall n > \bar{n} \; f(n) \leq cg(n)$$

  Siano c ed $\bar{n}$ due valori che soddisfano la formula $\forall n > \bar{n} \; f(n) \leq cg(n)$. Vogliamo dimostrare che vale

  $$\exists c' \; \exists \bar{n}' \; \forall n > \bar{n}' \; g(n) \geq c' f(n)$$

quindi

$$f(n) \leq cg(n)$$

$$\frac{1}{c}f(n) \leq g(n)$$

$$g(n) \geq \frac{1}{c}f(n)$$

Fissiamo $c' = \frac{1}{c}$ e otteniamo che:

$$g(n) \geq c'f(n)$$

$\forall n > \bar{n}$, con $\bar{n}' = \bar{n}$.

- Dimostriamo l'implicazione a sinistra: $g \in \Omega(f) \rightarrow f \in O(g)$

$$\exists c\ \exists \bar{n}\ \forall n > \bar{n}\ g(n) \geq cf(n)$$

Siano c ed $\bar{n}$ due valori che soddisfano la formula $\forall n > \bar{n}\ g(n) \geq cf(n)$. Vogliamo dimostrare che vale

$$\exists c''\ \exists \bar{n}''\ \forall n > \bar{n}''\ f(n) \leq c''g(n)$$

quindi

$$g(n) \geq cf(n)$$

$$\frac{1}{c}g(n) \geq f(n)$$

$$f(n) \leq \frac{1}{c}g(n)$$

Fissiamo $c'' = \frac{1}{c}$ e otteniamo che:

$$f(n) \leq c''g(n)$$

$\forall n > \bar{n}$, con $\bar{n}'' = \bar{n}$.

□

2.1.5 Notazione di uso comune

Sia A un algoritmo.

- Affermare che $A \in O(f)$ significa che $\exists g \in O(f)$ tale che A *termina sempre entro tempo* g.
- Affermare che $A \in \Omega(f)$ significa che *esiste uno schema di input* tale che su quello schema A richiede almeno g operazioni con $g \in \Omega(f)$.
- Se combaciano allora diciamo che $A \in \Theta(f)$.

Sia P un problema.

- Dire che $P \in O(f)$ significa che $\exists A$ che risolve P, con $A \in O(f)$.
- Dire che $P \in \Omega(f)$ significa che $forallA$ che risolve P, con $A \in \Omega(f)$.
- Se combaciano diciamo che $P \in \Theta(f)$.

2.1.6 Equazioni di ricorrenza

Calcolo del fattoriale di un numero

```
FATT(n)
    IF n = 0 THEN
        RET 1
    ELSE
        RET n * FATT(n-1)
```

L'equazione di ricorrenza corrispondente è:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ 1 + T(n-1) & \text{se } n > 0 \end{cases} \tag{2.1}$$

2.2 Risoluzione di e quazioni di ricorrenza

2.2.1 Risoluzione di equazioni di ricorrenza - Metodo iterativo

Prendendo in considerazione l'equazione di ricorrenza del fattoriale:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ 1 + T(n-1) & \text{se } n > 0 \end{cases} \tag{2.2}$$

Possiamo risolvere l'equazione come segue:

$$T(n) = 1 + T(n-1)$$
$$= 1 + (1 + T(n-2))$$
$$= 2 + T(n-2)$$
$$= 2 + (1 + T(n-3))$$
$$= \cdots = i + T(n-i)$$
$$= \cdots = n + t(n-n) = n + 1$$

Questa serie di passaggi intuitivi possono essere verificati per induzione.

Esempio di risoluzione con metodo iterativo

Data l'equazione $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$, applichiamo il metodo iterativo per risolverla.

$$= 3\left(3T\left(\left\lfloor \frac{\lfloor \frac{n}{4} \rfloor}{4} \right\rfloor\right) + \left\lfloor \frac{n}{4} \right\rfloor\right) + n$$
$$= 3\left(3T\left(\left\lfloor \frac{n}{4^2} \right\rfloor\right) + \left\lfloor \frac{n}{4} \right\rfloor\right) + n$$
$$= 3^2 T\left(\left\lfloor \frac{n}{4^2} \right\rfloor\right) + 3\left\lfloor \frac{n}{4} \right\rfloor + n$$
$$= 3^2\left(3T\left(\left\lfloor \frac{\lfloor \frac{n}{4^2} \rfloor}{4} \right\rfloor\right) + \left\lfloor \frac{n}{4^2} \right\rfloor\right) + 3\left\lfloor \frac{n}{4} \right\rfloor + n$$
$$= 3^3 T\left(\left\lfloor \frac{n}{4^3} \right\rfloor\right) + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + 3\left\lfloor \frac{n}{4} \right\rfloor + n$$

Dopo i iterazioni otteniamo:

$$3^i T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right) + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor + 3^{i-2}\left\lfloor \frac{n}{4^{i-2}} \right\rfloor + \cdots + 3\left\lfloor \frac{n}{4} \right\rfloor + n$$

$$= 3^i T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right) + \left(\sum_{k=1}^{i-1} 3^k \left\lfloor \frac{n}{4^k} \right\rfloor\right) + n$$

Se $4^i \geq n \rightarrow i \geq log_4 n$ sono sicuramente in un caso base.
Otteniamo:

$$= 3^{log_4 n} T\left(\left\lfloor \frac{n}{4^{log_4 n}} \right\rfloor\right) + \left(\sum_{k=1}^{log_4 n - 1} 3^k \left\lfloor \frac{n}{4^k} \right\rfloor\right) + n$$

Quando siamo nel caso base possiamo dire che $T\left(\left\lfloor \frac{n}{4^{log_4 n}} \right\rfloor\right)$ è una costante, quindi la sostituiamo con c.

$$c3^{log_4 n} + \left(\sum_{k=1}^{log_4 n - 1} 3^k \left\lfloor \frac{n}{4^k} \right\rfloor \right) + n$$

Ora consideriamo $n = \left(\frac{3}{4}\right)^0 n$:

$$\leq c3^{log_4 n} + \sum_{k=1}^{log_4 n - 1} 3^k \left\lfloor \frac{n}{4^k} \right\rfloor + \left(\frac{3}{4}\right)^0 n$$

$$= c3^{log_4 n} + \sum_{k=0}^{log_4 n - 1} 3^k \left(\frac{3}{4}\right)^k n$$

La costante c non è rilevante, quindi possiamo eliminarla. Possiamo inoltre notare che $\sum_{k=0}^{log_4 n-1} \left(\frac{3}{4}\right)^k n$ è una somma parziale di una serie geometrica di ragione $\frac{3}{4}$. Perciò possiamo dire che

$$n \sum_{k=0}^{log_4 n - 1} \left(\frac{3}{4}\right)^k \leq n \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = \Theta(n) \quad \textit{(lineare)}$$

Ora dobbiamo analizzare l'altra parte della formula, ovvero $3^{log_4 n}$

$$3^{log_4 n} = n^{log_n 3^{log_4 n}}$$

$$^{1} = n^{log_4 n * log_n 3}$$

$$= n^{log_4 3}$$

Otteniamo

$$T(n) = n^{log_4 3} + \Theta(n)$$

Quindi la soluzione dell'equazione di ricorrenza è $\Theta(n)$.

2.2.2 Risoluzione di equazioni di ricorrenza - Metodo di sostituzione

Data un'ipotesi andiamo a verificare se è corretta, sostituendo tutte le ocorrenze di $T(n)$ sulla destra con l'ipotesi da verificare.

[1] formula del cambio di base del logaritmo.

Esempio di risoluzione con metodo di sostituzione

Data l'equazione di ricorrenza $T(n) = 2T\left(\left\lfloor \frac{2}{n} \right\rfloor\right) + n$. Supponiamo ci venga detto che $T(n) \in O(nlogn)$.

Sostituiamo le ocorrenze di destra con la nostra ipotesi:

$$T(n) \leq 2 * c * \left\lfloor \frac{n}{2} \right\rfloor log \left\lfloor \frac{n}{2} \right\rfloor + n$$

$$\leq \cancel{2} * c * \frac{n}{\cancel{2}} \log \frac{n}{2} + n$$

(scegliamo per il logaritmo la base che ci fa comodo)

$$\leq cnlogn - cnlog_2 2 + n$$

$$\leq cnlogn - (cn - n) \leq cnlogn$$

quando $(cn - n) \geq 0$, ovvero:

$$n(c-1) \geq 0$$

$$\forall c \geq 1,\ \bar{n} = \text{qualsiasi}$$

Nota: se applicando il metodo di sostituzione, sostituendo l'ipotesi si ottiene l'ipotesi, allora abbiamo verificato che la nostra ipotesi è vera.

2.2.3 Teorema dell'esperto - Master Theorem

Questo teorema permette di risolvere tutte le equazioni di ricorrenza nella forma

$$T(n) = aT(\frac{n}{b}) + f(n),\ \text{ con } a \geq 1, b > 1$$

Il teorema prevede di confrontare l'ordine di grandezza di due funzioni ricavate dala formula precedente. Le due formule sono:

- $f(n)$
- $n^{log_b a}$

Il teorema prevede tre casi:

1. Se $f(n) \in O(n^{log_b a - \varepsilon})$, con $\varepsilon > 0$ allora $T(n) \in \Theta(n^{log_b a})$
2. Se $f(n) \in \Theta(n^{log_b a})$, allora $T(n) \in \Theta(f(n)logn)$
3. Se $f(n) \in \Omega(n^{log_b a + \varepsilon}$, con $\varepsilon > 0$, allora $T(n) \in \Theta(f(n))$, con $af(\frac{n}{b}) \leq cf(n)$, con $c < 1$ e $n > \bar{n}$.

Esempi

Caso 1:

$$T(n) = 9T(\frac{n}{3}) + n$$

Ricaviamo le due funzioni da confrontare: $f(n) = n$ e $n^{log_b a} = n^{log_3 9} = n^2$. Siccome $n \in O(n^2)$ cerchiamo un $\varepsilon > 0$ tale che $n \in O(n^{2-\varepsilon})$. Prendendo $\varepsilon = 0.1$ è verificato il caso 1:

$$\frac{n^2}{n^{0.1}} \rightarrow n \in O(n^{2-0.1})$$

Caso 2:

$$T(n) = T(\frac{2n}{3}) + 1$$

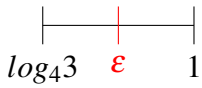
Scomponiamo l'equazione di ricorrenza nei suoi componenti, come da teorema, e otteniamo $a = 1$, $b = \frac{2}{3}$ e la funzione nota $f(n) = 1$. Ora abbiamo che $n^{log_b a} = n^{log_{\frac{2}{3}} 1} = 1$. Si può notare che $f(n) \in \Theta(n^1)$ e quindi siamo nel caso 2.

$$T(n) \in \Theta(f(n)logn) = \Theta(1logn)$$

Caso 3:

$$T(n) = 3T(\frac{n}{4}) + nlogn$$

In questo caso abbiamo $a = 3$, $b = 4$ e $f(n) = nlogn$. $n^{log_4 3} < 1$, mentre $nlogn > 1$. Dobbiamo trovare un ε tale che $(n^{log_4 3} * n^{\varepsilon}) < nlogn$. Graficamente possiamo notare che quasiasi ε compreso tra $log_4 3$ e 1, soddisfa la condizione:



Quindi possiamo prendere come $\varepsilon = \frac{1-log_4 3}{2}$. Essendo nel caso 3, dobbiamo verificare anche la condizione $af(\frac{n}{b}) \leq cf(n)$:

$$^1 a\frac{n}{b}log\frac{n}{b} \leq c(nlogn), \quad c < 1$$

$$\frac{3}{4}nlog\frac{n}{4} \leq c(nlogn), \quad c < 1$$

con $c = \frac{3}{4}$ la condizione è soddisfatta.

[1] se $a > b$, c deve essere maggiore di 1, quindi non verifica la condizione.

Caso 4: non è possibile applicare il teorema

$$T(n) = 2T(\frac{n}{2}) + nlogn$$

Provando ad applicare il teorema a questa formula ricaviamo che $n^{log_2 2} = n^1$ e che la funzione nota $f(n) = nlogn$. Dovremmo trovare un $\varepsilon > 0$ tale che $nlogn \in (n - n^{\varepsilon})$, che **non esiste**. In questo caso il teorema non è applicabile, ma potrebbe essere corretta la supposizione intuitiva che l'equazione di ricorrenza data appartenga ad uno dei tre casi. Per verificare se l'intuizione è corretta possiamo applicare il *metodo di sostituzione*.

Problema dell'ordinamento

3.1 Input, output e relazioni di ordinamento

Input: è definito come una sequenza di oggetti $a_1, \ldots, a_n$ su cui è definita una relazione di ordinamento totale.

Output: è una permutazione $a'_1, \ldots, a'_n$ di $a_1, \ldots, a_n$ tale che per ogni $i < j$ si ha $a'_i \leq a'_j$.

Relazione di ordinamento: significa che esiste un metodo per determinare, dati due oggetti, quale sia il più piccolo e quale il più grande. Se non si conosce la *relazione d'ordine*, senza confronti non siamo in grado di dire come due oggetti vanno ordinati tra loro, quindi l'unico modo che abbiamo per ordinare due oggetti è quello di confrontarli tra loro.

Nota: il problema dell'ordinamento $\in O(n^2)$. Inoltre, il problema dell'ordinamento $\in \Omega(n)$.

3.2 Insertion sort

3.2.1 Pseudocodice

```
INSERTION_SORT(A)
    FOR j ← 2 TO length(A)
        key ← A[j]
        i ← j - 1
        WHILE i > 0 AND A[i] > key
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

L'algoritmo `INSERTION_SORT` se applicato ad un array con elementi uguali, *non scambia l'ordine relativo di tali elementi*, questo significa che è **stabile**. Inoltre, analizzando *la quantità di memoria si può vedere che è costante e non dipende dalla dimensione del problema*, questa proprietà significa che l'algoritmo **ordina in loco**.

3.2.2 Complessità di `INSERTION_SORT`

Per calcolare la complessità di `INSERTION_SORT` possiamo notare che il blocco `WHILE` viene eseguito *al più n* volte, il ciclo `FOR` viene ripetuto $(n-1)$ volte. Come limite superiore, intuitivamente, abbiamo n^2, ma potrebbe essere una stima esagerata. Alla prima iterazione fa al massimo 1 confronto, alla seconda iterazione fa 2 confronti e se arriviamo all'ultima iterazione i confronti sono $n\cancel{-1}$[1].

Possiamo quindi formulare la complessità di `INSERTION_SORT` nella formula

$$1+2+3+\cdots+n=\frac{n(n+1)}{2}$$

Concludendo che $\frac{n(n+1)}{2} \in \Theta(n^2)$.

`INSERTION_SORT` è un algoritmo a complessità quadratica, ma in alcuni casi è molto veloce: se consideriamo un array di piccole dimensioni oppure un array quasi completamente ordinato, l'algoritmo termina in tempo lineare.

[1] Non ci interessano le costanti, possiamo togliere il -1 e approssimare a n.

3.3 Merge sort

3.3.1 Pseudocodice

```
MERGE_SORT(A, p, r)
    IF p < r     // quando p = r, l'array è ordinato
        q ← (p + r) / 2     // implicita l'appross.  per difetto
        MERGE_SORT(A, p, q)     // ordino la prima parte
        MERGE_SORT(A, q+1, r)     // ordino la seconda parte
        MERGE(A, p, q, r)     // unisco le due parti ordinate
```

```
MERGE(A, p, q, r)
    i ← 1     // array risultato
    j ← p     k ← q+1     WHILE j <= q || k <=r
        IF j <= q && (k > r || A[j] <= A[k])     // stabile
            B[i] ← A[j]
            j ← j+1
        ELSE
            B[i] ← A[k]             k ← k+1
        i ← i+1
    FOR i ← 1 TO r - p + 1
        A[i+p-1] ← B[i]
```

Salta subito all'occhio che `MERGE_SORT` non ordina in loco, infatti nella `MERGE` si usa un array ausiliario B, ma la stabilità è preservata.

3.3.2 Complessità di `MERGE_SORT`

`MERGE_SORT` è un algoritmo ricorsivo che agisce, ad ogni nuova chiamata, su un sotto array di dimensione $\frac{1}{2}$ rispetto all'array. La sua equazione di ricorrenza è:

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

Siamo nel secondo caso indicato dal *Master Theorem*, ovvero $T(n) \in \Theta(nlogn)$.

Esempio

Consideriamo di dover ordinare 1 milione di dati, e abbiamo due opzioni:

1. Utilizziamo `INSERTION_SORT` su un supercomputer in grado di svolgere 100 milioni di istruzioni al secondo, con implementazione ottimizzata. Complessità calcolata $2n^2$.

2. Utilizziamo `MERGE_SORT` su un PC ordinario in grado di svolgere 1 milione di istruzioni al secondo, con implementazione scarsa. Complessità calcolata $50nlogn$.

Calcoliamo il tempo di esecuzione dell'algoritmo per verificare in quale contesto il tempo richiesto per risolvere il problema è minore. Nel caso 1 abbiamo

$$t = \frac{2(10^6)^2}{10^8} = 20000s = 5.56h$$

Nel caso 2 abbiamo

$$t = \frac{50 * 10^6 * 20}{10^6} = 1000s = 16.67min$$

Possiamo notare come le costanti siano poco influenti rispetto all'ordine di grandezza.

3.4 Quick sort

3.4.1 Pseudocodice

```
QUICK_SORT(A, p, r)
    IF p < r         q ← PARTITION(A, p, r)
        QUICK_SORT(A, p, q)
        QUICK_SORT(A, q+1, r)
```

```
PARTITION(A, p, r)
    x ← A[p]
    i ← p-1
    j ← r+1
    WHILE TRUE
        REPEAT j ← j-1
        UNTIL A[j] <= x
        REPEAT i ← i+1
        UNTIL A[i] >= x
        IF i < j
            SCAMBIA (A[i], A[j])
        ELSE
            RETURN j
```

`QUICK_SORT` non è stabile perché la partition scambia gli elementi uguali, ma ordina in loco.

3.4.2 Complessità di `QUICK_SORT`

Il caso pessimo di `QUICK_SORT` è un array ordinato. Possiamo intuitivamente scrivere un'equazione di ricorrenza per `QUICK_SORT` cosi:

$$T(n) = C(partition) + T(1) + T(n-1)$$

Analizzando `PARTITION` si nota che la sua complessità è lineare, perchè deve svolgere al più n operazioni, $\frac{n}{2}$ nel primo `REPEAT-UNTIL` ed $\frac{n}{2}$ nel secondo. Otteniamo quindi l'equazione

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

quindi la complessità di `QUICK_SORT` è quadratica.

Si nota anche che, nel caso medio, quando l'array è totalmente disordinato, l'algoritmo è lineare. Se troviamo quindi un modo per prendere la x della partition in modo casuale, ed avendo casi d'uso in cui l'algoritmo è utilizzato molte volte, possiamo avvicinarci al caso medio sempre di più.

```
RANDOMIZED_PARTITION(A, p, r)
    i <- RANDOM (p, r)
    SCAMBIA (A[i], A[p])
    RETURN PARTITION (A, p, r)
```

A questo punto $T(n)$ non è più definito ma casuale, e abbiamo uno spazio campione di tutte le possibili esecuzioni dell'algoritmo. Per ogni esecuzione sappiamo quanto tempo richiede:

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

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

E otteniamo che $T(n) \leq nlogn$.

3.4.3 Rimozione della ricorsione di coda

Analizzando la complessità di QUICK_SORT rispetto alla memoria notiamo che lo stack di attivazione può crescere molto a causa delle chiamate ricorsive. Possiamo ridurle sostituendo l'ultima con un ciclo:

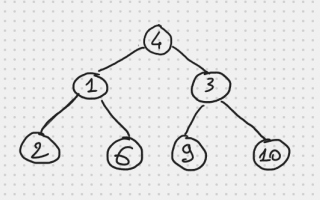
```
QUICK_SORT(A, p, r)
    WHILE p < r
        q ← PARTITION(A, p, r)
        QUICK_SORT(A, p, q)
        p ← q+1
```

Se ora si riuscisse a trasformare questo algoritmo in modo tale da eseguire la chiamata ricorsiva sul sottoarray più piccolo tra i due generati dalla partition, ridurremmo ulteriormente lo stack di attivazione.

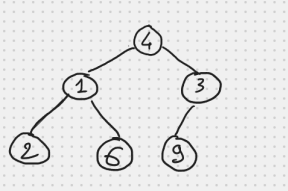
3.5 Struttura dati: Heap

Lo heap è una struttura dati ad albero binario, divisibile in livelli. Ogni livello contiene dei nodi, se un nodo non ha figli viene detto foglia. Quando un livello, partendo da sinistra, non ha tutti i nodi, si dice che il livello è semicompleto. Un albero che come ultimo livello ha un livello semicompleto, si dice albero semicompleto. Naturalmente, un albero che presenta tutti i nodi anche sull'ultimo livello si dice completo.

Albero completo

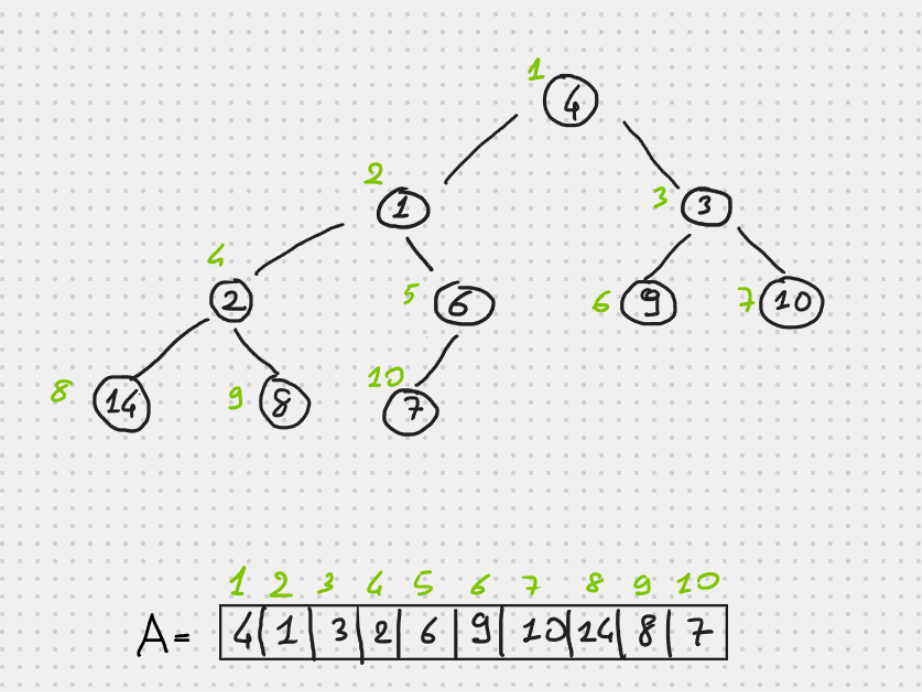


Albero semicompleto

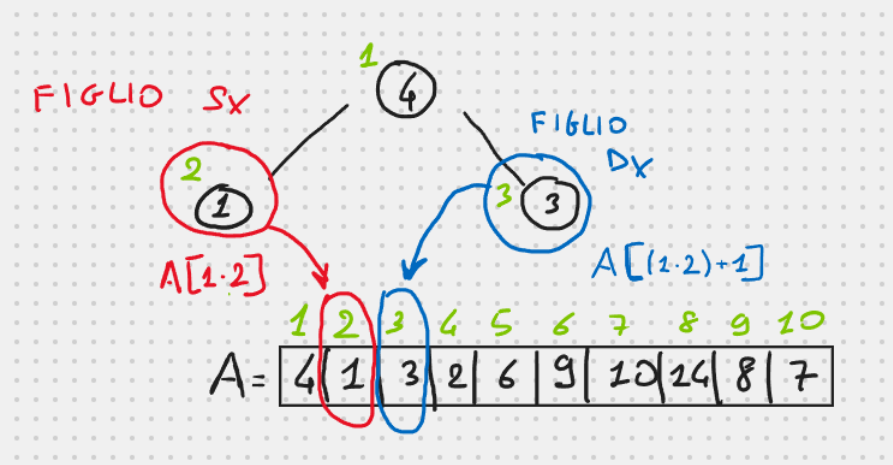


3.5.1 Rappresentazione di un albero come array

Gli alberi binari semicompleti possono essere rappresentati come un array. Prendendo un qualsiasi nodo, dividendone il suo indice per 2 si ottiene l'indice del nodo genitore, mentre per ottenere gli indici dei figli basta prendere l'indice del genitore e moltiplicarlo per 2 per il figlio sinistro, e sommare 1 per il figlio destro.



Per esempio, se il nodo con valore 4 ha indice 1, il figlio sinistro, nodo di valore 1 sarà `A[1*2]`, mentre il figlio destro (nodo di valore 6) `A[(1*2)+1]`.



3.5.2 Heap

Come ulteriore vincolo, perché un albero sia uno heap, dobbiamo imporre che gli oggetti che fanno parte dell'albero devono avere una relazione di ordinamento tra loro e i figli devono essere minori o uguali del nodo genitore. In uno heap completo con h livelli ci sono $log_2 h$ nodi.

$$2^0 + 2^1 + 2^3 + \cdots + 2^h = 2^{h+1} - 1$$

$$n \geq 2^h \Rightarrow logn \geq h$$

In questo modo ho la garanzia che nello heap l'elemento più grande, o il più piccolo se cambio la relazione di ordinamento, lo trovo alla radice. Quindi in uno heap memorizzato in un array, l'elemento di indice 1 sarà il massimo (o il minimo).

3.6 Heap sort

Utilizzando lo heap possiamo creare un algoritmo di ordinamento:

```
HEAPIFY(A, i)
    l ← LEFT(i)     // ret.  indice figlio sx
    r ← RIGHT(i)     // ret.  indice figlio dx
    IF l <= HEAPSIZE(A) AND A[l] > A[i]
        largest ← l     // se figlio sx > genitore
    ELSE
        largest ← i     // se genitore > figlio sx
    IF r <= HEAPSIZE(A) AND A[r] > A[largest]
        largest ← r     // se figlio dx > A[largest]
    IF i != largest     // se il genitore non è il maggiore
        SCAMBIA (A[i], A[largest])     // scambio il genitore con
il maggiore dei figli
        HEAPIFY (A, largest)
```

```
BUILD_HEAP(A)
    HEAPSIZE(A) ← LENGTH(A)
    FOR i ← LENGTH(A) / 2 DOWNTO 1
        HEAPIFY(A, i)
```

```
HEAP_SORT(A)
    BUILD_HEAP (A)
    FOR i ← LENGTH(A) DOWNTO 2
        SCAMBIA(A[1], A[i])
        HEAPSIZE(A) ← HEAPSIZE(A) - 1
        HEAPIFY(A, 1)
```

`HEAP_SORT` ordina in loco, ma non è stabile.

3.6.1 Complessità di `HEAP_SORT`

La complessità massima di `HEAPIFY` è la profondità massima dell'albero, ovvero $logn$ (dove n è il numero di nodi). In uno heap di n nodi ci sono $\frac{n}{2}$ foglie al massimo. Perciò la ricerca di un elemento minimo in uno heap richiede tempo lineare (devo confrontare tutte le foglie).

La complessità di `BUILD_HEAP` è $\Theta(n)$.

$$0\left(\frac{n}{2}\right)+1\left(\frac{n}{2^2}\right)+2\left(\frac{n}{2^3}\right)+\cdots+\frac{n}{2^{logn}}logn$$

Che si può scrivere come

$$\sum_{i=0}^{logn}\frac{n}{2^{i+1}}i = n\sum_{i=0}^{logn}\frac{i}{2^{i+1}} = n\sum_{i=0}^{logn} i\left(\frac{1}{2}\right)^{i+1} \leq n\sum_{i=0}^{\infty} i\left(\frac{1}{2}\right)^{i+1}$$

Questa formula è associabile alle serie geometriche, e otteniamo che la `BUILD_HEAP` ha complessità lineare $\Theta(n)$.

La complessità di `HEAP_SORT` è data dalla formula

$$n+log(n-1)+log(n-2)+\cdots+log(2)$$

e applicando le proprietà dei logaritmi la possiamo trasformare in

$$n+log((n-1)!) \Rightarrow logn! \in \Theta(log(n^n)) \text{ ovvero } \Theta(nlogn)$$