Algoritmi

Università di Verona
Imbriani Paolo -VR500437
Professor Roberto Segala

November 5, 2024

Contents

1 Introduzione agli algoritmi

Ci sono diverse definizioni di 'algoritmo' ma quella più semplice per definirlo è una sequenza di istruzioni volta a risolvere un problema. In maniera più semplice, possiamo definirlo come una ricetta. Le ricette sono delle istruzioni per creare dei piatti: queste istruzioni sono precise tuttavia possono avere anche delle varianti. Il problema non è trovare la ricetta, ma capire quale sia quella gisuta da utilizzare in base al problema posto.

In questo corso impararemo come *decidere*. Studiare i diversi approcci e il metodo migliore per affrontare un problema. Capiremo come confrontare gli algoritmi fra di loro.

1.1 Complessità

Classifichiamo gli algoritmi in base alla loro complessità ovvero quanto tempo ci mettono per essere completati. Il nostro obiettivo è quello non di misurare il tempo in sè (poiché può dipendere dal tipo di macchina che utilizziamo) che impiega un programma a finire ma piuttosto capire il numero di istruzioni elementari che vengono impiegate per risolvere il problema.

La complessità non è un numero bensì una funzione, poiché mappa la dimensione del problema in numero di istruzioni da eseguire. Per esempio, quando si parla di dimensione di una matrice ci si viene più comodo vedere il numero di colonne e righe piuttosto che contare il numero di elementi all'interno della matrice. La dimensione del problema è un insieme di oggetti, tipicamente un insieme di numeri che ci permette di avere un idea chiara di capire quanto sia grande il problema. Se riusciamo a misurarlo bene, potremmo facilitarci la vita nel risolvere il problema.

1.2 Complessità dei costrutti e ordini di grandezza

È doveroso stare attenti a cosa moltiplichiamo: se moltiplichiamo due vettori, il numero di operazioni da eseguire è esattamente n. Mentre se fossero orgnanizzate in matrici quadrate con lunghezza del lato $\sqrt{n}$ la sua complessità andrebbe ad aumentare a $O(n\sqrt{n})$. Come si rappresentano le istruzioni di un programma? Se sono in serie:

$$\begin{array}{ll} I_1 & c_1(n) \\ I_2 & c_2(n) \\ \vdots & \\ I_l & c_l(n) \end{array}$$

Dove la complessità totale allora sarà:

$$\sum_{i=1}^{l} c_i(n)$$

oppure all'interno di un costrutto **if-else:**

$$\begin{array}{cc} \text{if cond} & c_{\text{cond}}(n) \\ I_2 & c_1(n) \\ \text{else} & \\ I_l & c_2(n) \end{array}$$

In questo caso, come faccio a sapere la complessità totale di questi istruzioni? Per capirlo, studiamo il caso nel *worst case scenario* ovvero nel caso peggiore. A volte vengono mostrati anche i casi migliore ma di solito si calcola il tempo peggiore. Quindi ci interessa specialmente il *limite superiore* dell'algoritmo, quindi piuttosto che dire che la complessità è esattamente **uguale** questo numero, invece noi diremo che è **minore o uguale** del numero calcolato.

$$C(n) = c_{\text{cond}}(n) + max(c_1(n), c_2(n))$$

Mentre per un while loop?

$$\begin{array}{cc} \text{while cond} & c_{cond}(n) \\ I & c_0(n) \end{array}$$

Sia m limite superiore numero di iterazioni che esegue l'algoritmo. La complessità di questo algoritmo quindi sarà:

$$C(n) = c_{cond}(n) + m(c_{cond}(n) + c_0(n))$$

Proviamo ora a calcolare la molteplicazioni tra matrici. Siano due matrici A e B rispettivamente con dimensione n x m e m x l.

```
For i <- 1 to n
    For j <- 1 to l
        C[i][j] <- 0
        For k <- 1 to m
            C[i][j] += A[i][j] * B[k][j]
```

Avere un risultato del tipo $5m + 1$ (riguardo il for più interno) è inutile perché non ci da informazioni realmente utili sulla complessità dell'algoritmo. Ad ogni modo contando tutti i for, avremo un risultato del tipo:

$$n(5ml + 4l + 2) + n + 1 = 5nml + 4nl + 3n + 1$$

Ci sono alcuni elementi in questa operazioni che non influiscono realmente sul risultato finale. Indi per cui, possiamo anche ometterlo all'interno del calcolo della complessità di un algoritmo. In realtà ciò che ci interessa realmente nel risultato finale è:

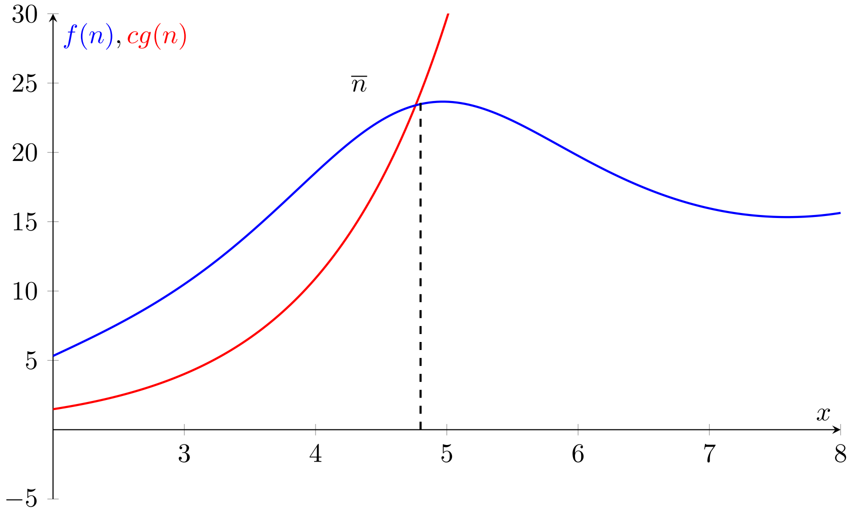
$$5\,\textbf{nml} + 4nl + 3n + 1$$

Infatti nml è capace di dirci tutto sulla complessità dell'algoritmo e da cosa dipende. Quando si parla di ordine di grandezza si parla in realtà del comportamento asintotico della funzione calcolata.

Definizione

$$f \in O(g) \Longleftrightarrow \exists c > 0,\ \exists \overline{n},\ \forall n \geq \overline{n} \,|\, f(n) \leq cg(n)$$

Figure 1: $f \in O(g)$

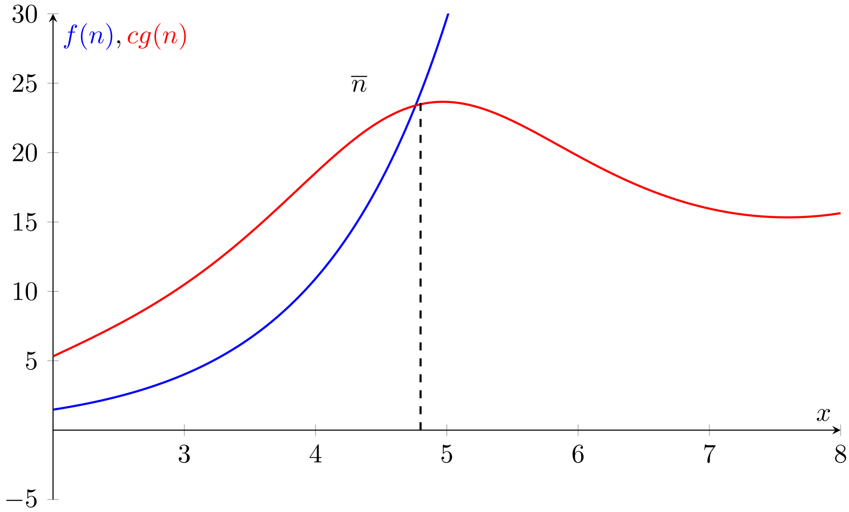


Definizione

$$f \in \Omega(g) \iff \exists c > 0,\ \exists \overline{n},\ \forall n \geq \overline{n} \,|\, f(n) \geq cg(n)$$

Che rispettivamente è l'inverso della funzione O grande.

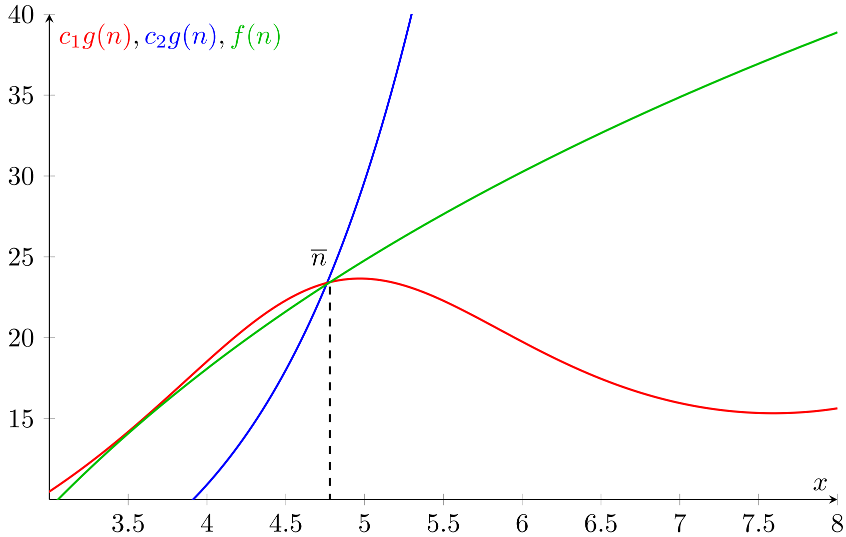
Figure 2: $f \in \Omega(g)$



Definizione

$$f \in \Theta(g) \iff f \in O(g) \wedge f \in \Omega(g)$$

Figure 3: $f \in \Theta(g)$



Esempio 1

Questa affermazione vale?

$$n \in O(2n)$$

Sì, perché esiste un c che soddisfa $n \leq c2n$.

Quest'altra affermazione vale?

$$2n \in O(n)$$

Sì perché esiste un c che soddisfa $2n \leq 2n$.

Esempio 2

Questa affermazione vale?

$$f \in O(g) \iff g \in \Omega(g)$$

Proviamo a dimostrare. Sappiamo che esiste un c e $\overline{n}$ tale che $\forall n > \overline{n}$ $f(n) \leq cg(n)$. Isoliamo la g:

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

Esiste quindi un c' che soddisfa la disuguaglianza.

$$c' = \frac{1}{c}$$

Esempio 3

$$f_1 \in O(g), f_2 \in O(g) \implies f_1 + f_2 \in O(g)$$

Dobbiamo dimostrare che esiste c_1 e c_2, $\overline{n}_1$ e $\overline{n}_2$ tali che:

$$\forall n > \overline{n}_1 \mid f_1(n) \leq c_1 g(n)$$

$$\forall n > \overline{n}_2 \mid f_2(n) \leq c_2 g(n)$$

Quindi

$$\overline{n} = max(\overline{n}_1, \overline{n}_2)$$

$$f_1(n) + f_2(n) \leq (c_1 + c_2)g(n)$$

Esempio 4

$$f_1 \in O(g_1), f_2 \in O(g_2) \mid f_1 f_2 \in O(g_1 g_2)$$

Quindi esiste c_1 e c_2, $\overline{n}_1$ e $\overline{n}_2$ tali che:

$$\forall n > \overline{n}_1 \mid f_1(n) \leq c_1 g(n)$$

$$\forall n > \overline{n}_2 \mid f_2(n) \leq c_2 g(n)$$

$$f_1(n) f_2(n) \leq c_1 g_1(n) c_2 g_2(n)$$

$$f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$$

Quindi:

$$c = c_1 c_2$$

$$\overline{n} = max(\overline{n}_1, \overline{n}_2)$$

1.3 Ordini di grandezza per le funzioni

L'algoritmo di ricerca A termina entro n. Immaginiamo che l'algoritmo A sia il seguente:

```
For i <- 0 to length(a) - 1
    if a[i] = x
        ret i
ret -1
```

Capiamo che la sua complessità è uguale a:

$$A \in O(n)$$

Per appurarci che la stima di complessità sia accurata dobbiamo controllare che $A \in \Omega(g)$ vuol dire che esiste uno schema di input tale per cui se $g(n)$ è il numero di passi necessari per risolvere l'istanza n, allora $g \in \Omega(f)$. Se riusciamo a fare questo allora $g \in \Theta(f)$ e quindi possiamo dire che la stima è accurata.

$P \in O(f)$ vuol dire che il problema P riesco a risolvere in tempo f. Supponiamo per assurdo che esista un algoritmo riesca a capire se l'elemento si trova nell'array o no. Possiamo capire velocemente che per contraddizione, essendo che esiste almeno un elemento dove l'algoritmo non ha guardato, siamo certi del malfunzionamento dell'algoritmo e quindi non è vero che esiste una stima di complessità più bassa di f.

Ora andiamo ad affrontare i tipi di algoritmi che si definiscono di "ordinamento".

Definizione

Input: Sequenza $(a_1, ..., a_n)$ di oggetti su cui è definita una relazione di ordinamento (in questo caso ordinamento per confronti).

Output: Permutazione $(a'_1, ..., a'_n)$ di $(a_1, ..., a_n)$ t.c. $\forall\, i > j,\;\; a'_i \leq a'_j$.

Andiamo ora a vedere i diversi tipi di algoritmi di ordinamento.

1.4 insertion_sort (A)

In questo caso la j sarà chiamata variabile "invariante" ovvero che mantiene una proprietà che continua a valere nel run-time dell'algoritmo. In questo caso, la j è invariante perchè tutti gli oggetti "a sinistra" di essa saranno considerati già ordinati. (Parte da 2 perché abbiamo deciso che la posizione all'interno dell'array parte da 1).

```
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--
    a[i+1] <- key
```

Ora dobbiamo capire quale sia la complessità di questo algoritmo. Se le cose vanno bene, l'algoritmo potrebbe terminare in $O(n)$. Tuttavia, seppur giusto, non è preciso. Infatti, nel peggiore dei casi, l'algoritmo termina in $O(n^2)$. Infatti il caso peggiore di input che mi può arrivare è un array ordinato al contrario con complessità uguale a:

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Questo algoritmo, quanto spazio di memoria usa in più rispetto allo spazio occupato dai dati?

Lo spazio di memoria di questo algoritmo rimane costante a prescindere dalla dimensione del problema. Un algoritmo del genere si dice che ***ordina in loco*** se la quantità di memoria extra è costante. Si parla di ***stabilità*** di un algoritmo di ordinamento quando l'ordine relativo di elementi uguali non viene scambiato dall'algoritmo. Quindi:

$$\text{Se } a_i = a_j \; i < j, \text{mantiene l'ordinamento}$$

2 Concetto di "Divide et impera"

2.1 Fattoriale e funzioni ricorsive

```
Fatt(n)
    if n = 0
        ret 1
    else
        ret n * fatt(n - 1)
```

L'argomento della funzione ci fa capire la complessità dell'algoritmo:

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

Con problemi ricorsivi si avrà una complessità con funzioni definite ricorsivamente. Questo si risolve induttivamente:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &= \underbrace{1 + 1 + \ldots + 1}_{i} + T(n-i) \end{aligned}$$

La condizione di uscita è: $n - i = 0 \quad n = i$

$$\begin{aligned} &= \underbrace{1 + 1 + \ldots + 1}_{n} + T(n-n) \\ &= n + 1 = \Theta(n) \end{aligned}$$

Questo si chiama passaggio iterativo.

Esempio 1

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

Questa funzione si può riscrivere come:

$$T(n) = \begin{cases} \text{Costante} & \text{se } n < a \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & \text{se } n \geq a \end{cases}$$

Se la complessità fosse già data bisognerebbe soltanto verificare se è corretta. Usando il metodo di sostituzione:

$$T(n) = cn \log n$$

sostituiamo nella funzione di partenza:

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \log\left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq \cancel{2}c\frac{n}{\cancel{2}} \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &\stackrel{?}{\leq} cn \log n \quad \text{se } n - cn \log 2 \leq 0 \end{aligned}$$

$$c \geq \frac{n}{n \log 2} = \frac{1}{\log 2}$$

Il metodo di sostituzione dice che quando si arriva ad avere una disequazione corrispondente all'ipotesi, allora la soluzione è corretta se soddisfa una certa ipotesi.

Esempio 2

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \qquad \in O(n)$$

$$T(n) \leq cn$$

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= cn + 1 \overset{?}{\leq} cn \end{aligned}$$

Il metodo utilizzato non funziona perchè rimane l'1 e non si può togliere in alcun modo. Per risolvere questo problema bisogna risolverne uno più forte:

$$T(n) \leq cn - b$$

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) - b + c\left(\left\lceil \frac{n}{2} \right\rceil\right) - b + 1 \\ &= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) - 2b + 1 \\ &= cn - 2b + 1 \overset{?}{\leq} cn - b \\ &= \underbrace{cn - b}_{} + \underbrace{1 - b}_{\leq 0} \leq cn - b \quad \text{se } b \geq 1 \end{aligned}$$

Se la proprietà vale per questo problema allora vale anche per il problema iniziale perchè è meno forte.

Esempio 3

$$\begin{aligned}
T(n) &= 3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + n \\
&= n + 3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right) \\
&= n + 3\left(\left\lfloor\frac{n}{4}\right\rfloor + 3T\left(\left\lfloor\frac{\lfloor\frac{n}{4}\rfloor}{4}\right\rfloor\right)\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2T\left(\left\lfloor\frac{n}{4^2}\right\rfloor\right) \\
&\leq n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2\left(\left\lfloor\frac{n}{4^2}\right\rfloor + 3T\left(\left\lfloor\frac{\lfloor\frac{n}{4^2}\rfloor}{4}\right\rfloor\right)\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2\left\lfloor\frac{n}{4^2}\right\rfloor + 3^3T\left(\left\lfloor\frac{n}{4^3}\right\rfloor\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + \ldots + 3^{i-1}\left\lfloor\frac{n}{4^{i-1}}\right\rfloor + 3^iT\left(\left\lfloor\frac{n}{4^i}\right\rfloor\right)
\end{aligned}$$

Per trovare il caso base poniamo l'argomento di T molto piccolo:

$$\begin{aligned}
\frac{n}{4^i} &< 1 \\
4^i &> n \\
i &> \log_4 n
\end{aligned}$$

L'equazione diventa:

$$\leq n + 3\left\lfloor\frac{n}{4}\right\rfloor + \ldots + 3^{\log_4 n-1}\left\lfloor\frac{n}{4^{\log_4 n-1}}\right\rfloor + 3^{\log_4 n}c$$

Si può togliere l'approssimazione per difetto per ottenere un maggiorante:

$$\begin{aligned}
&\leq n\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \ldots + \left(\frac{3}{4}\right)^{\log_4 n-1}\right) + 3^{\log_4 n}c \\
&\leq n\left(\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\right) + c3^{\log_4 n}
\end{aligned}$$

Per capire l'ordine di grandezza di $3^{\log_4 n}$ si può scrivere come:

$$3^{\log_4 n} = n^{\left(\log_n 3^{\log_4 n}\right)} = n^{\log_4 n \cdot \log_n 3} = n^{\log_4 3}$$

Quindi la complessità è:

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

Si ha che una funzione è uguale al termine noto della funzione originale e l'altra che è uguale al logaritmo dei termini noti. Se usassimo delle variabili uscirebbe:

$$\begin{aligned} T(n) &= aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) \\ &= O(f(n)) + O(n^{\log_b a}) \end{aligned}$$

2.2 Master Theorem o Teorema dell'esperto

Data una relazione di occorrenza di questa forma:

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

Distinguiamo tre casi:

1. $$f(n) \in O(n^{\log_b a - \epsilon}) \Longrightarrow T(n) \in \Theta(n^{\log_b a})$$

2. $$f(n) \in \Theta(n^{\log_b a}) \Longrightarrow T(n) \in \Theta(f(n) \log n)$$

3. $$f(n) \in \Omega(n^{\log_b a} + \epsilon) \Longrightarrow T(n) \in \Theta(f(n))$$

Esempio 1

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

$$a = 3, b = 3, f(n) = n$$

Basta che trovo un ϵ che mi dia n.

$$n^{\log_b a} = n^{\log_3 9} = n^2 * n^{-\frac{1}{2}}$$

In questo caso $\epsilon = n^{-\frac{1}{2}}$ e ci troviamo nel **PRIMO CASO** e la soluzione è $T(n) \in \Theta(n^2)$.

Esempio 2

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

$$a = 1, b = \frac{3}{2}, f(n) = n^0$$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0$$

Ci troviamo nel **SECONDO CASO** e la soluzione è $T(n) \in \Theta(\log n))$

Esempio 3

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

$$a = 3, b = 4, f(n) = n \log n$$

Ci troviamo nel **TERZO CASO** quindi basta qualsiasi valore di ϵ basta che sia contenuto tra $\log_3 4 \leq \epsilon \leq 1$. La soluzione è $T(n) \in \Theta(n \log n)$.

Esempio 4

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

$$a = 2, b = 2, f(n) = n \log n$$

$$n \log n \in \Omega(n^{1+\overset{?}{\epsilon}})$$

$$\log n \in \Omega(n^\epsilon) \text{ NON VALE}$$

Poiché un logaritmo è sempre più piccolo di un polinomio. Questo è un caso dove il teorema *non* si applica

2.3 Merge Sort (A, n)

Questo algoritmo di ordinamento *ricorsivo* utilizza il concetto di *divide et impera*.

Concettualmente, un merge sort funziona come segue:

1. **Dividi** l'array non ordinato in n sottoarray, ognuno contenente un elemento (un array di un elemento è considerato ordinato).

2. **Unisci** ripetutamente i sottoarray per produrre nuovi sottoarray ordinati finché non ne rimane solo uno. Questo sarà l'array ordinato.

La sua complessità considerando il merge con complessità lineare risulta:

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

Utilizzando il **Master Theorem** e cadendo del *secondo caso* possiamo confermare che il risultato è:

$$= \Theta(n \log n)$$

```
// A e' l'array mentre p ed r sono rispettivamente l'indice di partenza e di arrivo
mergeSort(A, p, r) // O(n log n)
    if (p < r)
        q <- floor((p+r)/2)
        mergeSort(A, p, q)
        mergeSort(A, q+1, r)
        merge(A, p, q, r)
```

```
merge(A, p, q, r)
    i <- 1
    j <- p
    k <- q+1
    // Ordina gli elementi di A in B
    // O il lato sinistro ha finito
    while(j <= q or k <= r) // O(n)
        if j <= q and (k > r or A[j] <= A[k])
            B[i] <- A[j]
            j++
        else
            B[i] <- A[k]
            k++
        i++
    // Copia gli elementi di B in A
    for i <- 1 to r-p+1  // O(n)
        A[p+i-1] <- B[i]
```

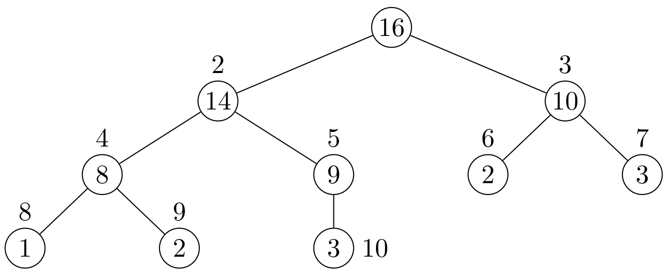
L'algoritmo è **stabile** poiché non vengono scambiati gli elementi uguali. Tuttavia non ordina **in loco** poiché utilizza uno spazio di memoria aggiuntivo.

2.4 Heap

L'Heap è un albero semicompleto (ogni nodo ha 2 figli ad ogni livello tranne l'ultimo che è completo solo fino ad un certo unto) in cui i nodi contengono oggetti con relazione di ordinamento.

Proprietà Heap:

$$\forall \text{ nodo, il contenuto è } \geq \text{ del contenuto dei figli}$$



La complessità dell'algoritmo è in base al numero di livelli dell'albero.

$\longrightarrow$ albero con n livelli:

$$\#\text{Nodi} = 2^0 + 2^1 + 2^2 + ... + 2^{n-1} = \frac{1-2^n}{1-2} = 2^n - 1$$

$\longrightarrow$ albero con n nodi:

$$\#\text{Livelli} = \log_2 n$$

$$\#\text{Foglie} = \frac{n}{2}$$

Le foglie di un albero sono la metà dei nodi dell'albero.

```
extractMax(H) // O(log n)
    h[1] <- H[H.heap_size()]
    H.heap_size()--
    heapify(H, 1)
```

```
heapify(H, 1) // O(log n)
    l <- left[i] //
    r <- right[i]
    if(l < h.heap_size() AND H[l] > H[i]
        largest <- l
    else
        largest <- i
    if m <= h.heap_size() and H[r] > H[largest]
        largest <- r
    if largest != i
        swap(H[i], H[largest])
        heapify(H, largest)
```

Creare uno heap da un array:

```
buildHeap(A)
    A.heap_size() <- length[A]
        for i <- length[A]/2 down to 1
            heapify(A, i)
```

Immagino tutte le foglie come heap con un solo nodo. L'indice del primo nodo che non è heap corrisponde a $\frac{length(A)}{2}$ su questo allora chiamo `heapify`.

2.5 Heapsort

```
HeapSort(A) // O(n log n)
    buildHeap(A)
    for i <- length(A) to 1
        scambia(A[1], A[i])
        heapsize(A)--
        heapify(a, i)
```

L'Heap Sort è un algoritmo che lavora **in loco** tuttavia non è **stabile**. Tuttavia riusciamo a fare una stima migliore e più corretta?

$$n \log i = \sum_{i=1}^{n} \log i = \log \prod_{i=1}^{n} i = \log n! = \Theta(\log n^n) = \Theta(n \log n)$$

In questo caso, essere accurati non aiuta, ma abbiamo avuto la certezza che non esiste una stima migliore.

2.6 Quicksort

Come funziona l'algoritmo?

1. Dividi prima l'array in due parti. (Partizione)
2. Devi essere sicuro che tutti gli elementi di sinistra siano $\leq$ di quelli di destra. Ricorsivamente ordina la parte sinistra e la parte destra.
3. A questo punto l'array è ordinato.

```
quickSort(A, p, r)
    if (p < r)
        q <- partition(a, p, r)
        quickSort(A, p, q)
        quickSort(A, q+1, r)
```

```
partition(A, p, r) // O(n)
    x <- A[p] // Elemento Perno
    i <- p-1
    k <- r+1
    while true
        repeat j-- until a[j] <= x
        repeat i++ until a[i] >= x
        if i < j
            scambia(a[i], a[j])
        else
            ret j
```

Scegliamo un elemento a caso in base a quello comparato rispetto all'elemento perno tale che:

$$sx \leq perno \leq dx$$

Questo algoritmo non è **stabile** ma lavora **in loco**. La sua complessità?

$$T(n) = T(\text{partition}) + T(q) + T(n-q)$$

Se il quicksort è perfettamente diviso in due, allora la sua complessità è $O(n \log n)$. Se invece l'array è già ordinato la sua equazione di ricorrenza sarà:

$$= n + T(1) + T(n-1) = \Theta(n^2)$$

Tuttavia non ci aspettiamo che questo caso sia frequente e quindin nella stra grande maggioranza dei casi allora:

$$T(n) = n + T(cn) + T((1-c)n)$$

Un equazione di questo tipo sappiamo che ha come complessità $\Theta(n \log n)$.

```
rand_Partition(A, p, r)
    i <- rand(p .. r) // Ora l'elemento perno e' un elemento a caso
    scambio(A[p], A[i]
    ret partition(A, p, r)
```

$$\begin{aligned} T(n) = n + \frac{1}{n}(T(1) + T(n-1) + \frac{1}{n}(T(2) + (T-2)) + \ \dots \ + \\ \frac{1}{n}(T(n-2) + T(2)) + \frac{1}{n}(T(n-1) + T(1) = \\ = n + \frac{1}{n}\sum_i (T(i) + T(n-i)) \\ = n + \frac{2}{n}\sum_i T(i) \in \boxed{O(n \log n)} \end{aligned}$$

Qualsiasi algoritmo che *lavora per confronti* deve fare almeno $O(n \log n)$.

3 Algoritmi di ordinamento in tempo lineare

3.1 Algoritmi non basati su confronti

3.1.1 Counting Sort

Tuttavia possiamo trovare algoritmi che come tempo di esecuzione hanno tempo lineare. Come? Non lavorando a **confronti**.

Come ordinare n numeri con valori da 1 a k?

```
countingSort(A, k)
    for i <- 1 to k
        C[i] <- 0
    for j <- 1 to len(A)
        C[A[j]]++
    for i <- 2 to k
        C[i] <- C[i-1]+C[i]
    for j <- len(A) down to 1
        B[C[A[j]]] <- a[j]
        C[A[J]]--
```

La complessità di questo algoritmo è $O(n + k)$ dove n è la lunghezza dell'array e k e il range di valori.

3.1.2 Radix Sort

Il radix sort è un algoritmo di ordinamento che ordina gli elementi confrontando i singoli bit. Quello che fa è ordinare per la cifra meno significativa, poi per la seconda cifra meno significativa e così via.

```
radixSort(A, d) // O(d(n+k))
  for i <- 1 to d
      countingSort(A, n)
```

La complessità di questo algoritmo è

$$\Theta(d(n+k))$$

dove d è il numero di cifre e k è il range di valori. Se si vuole invece ordinare n valori da 1 a n^2-1, le costanti nascoste all'interno del Θ sono molto alte e quindi non è un algoritmo efficiente. Tuttavia si posson rappresentare i numeri in base n e quindi ottenere un algoritmo lineare.

3.1.3 Bucket Sort

Il Bucket Sort è un algoritmo di ordinamento che funziona bene quando i dati sono distribuiti uniformemente. Quindi su un array di n elementi **distribuiti uniformemente** su $[0,1)$, si può dividere l'intervallo in n sottointervalli con probabilità $\frac{1}{n}$ e poi ordinare i singoli sottointervalli chiamate anche *"Bucket"*. Infatti se i dati sono distribuiti uniformemente allora la complessità dell'algoritmo è lineare:

$$\Theta(n)$$

Questo perché in ogni bucket ci si aspetta un valore costante e quindi indipendente dal valore di n.

Il caso pessimo però è quando tutti gli elementi ricadono nello stesso bucket. La probabilità che questo accada è molto bassa infatti è:

$$\underbrace{\frac{1}{n} * \frac{1}{n} * \cdots * \frac{1}{n}}_{n-1} = \frac{1}{n^{n-1}}$$

e la sua complessità diventa:

$$O(n^2)$$

Sia X_{ij} la variabile aleatoria che vale:

$$\begin{cases} 1 & \text{Se l'elemento } i \text{ va nel bucket } j \\ 0 & \text{altrimenti} \end{cases}$$

Per esprimere il numero di elementi nel bucket j si ha:

$$N_j = \sum_i X_{ij}$$

La complessità di questo algoritmo quindi può essere espressa come:

$$C = \sum_j {N_j}^2$$

Dove il valore atteso è:

$$E[C] = E\left[\sum_j {N_j}^2\right] = \sum_j E[{N_j}^2] = \sum_j \left(Var(N_j) - E[N_j]^2\right)$$

Dove $E[N_j]$ è:

$$E[N_j] = \sum E[X_{ij}] = \sum_{j=1}^{n} \frac{1}{n} = 1$$

$$Var[N_j] = \sum Var(X_{ij}) = \sum \frac{1}{n} * \left(1 - \frac{1}{n}\right) = 1 - \frac{1}{n}$$

E quindi possiamo svolgere il calcolo precedente dove:

$$\begin{aligned}\sum_j \left(Var(N_j) - E[N_j]^2\right) &= \sum_j \left(\left(1 - \frac{1}{n}\right) + 1\right) \\ &= \sum_j 2 - \frac{1}{n} \\ &= 2n - 1\end{aligned}$$

Le distribuzioni possono essere arbitrarie ma basta che tutti i bucket abbiano la stessa probabilità. Prendiamo:

$$n_1, ..., n_2, ..., n_l$$

$$\frac{1}{n_1}, \dots, \frac{1}{n_2}, \dots, \frac{1}{n_l}$$

La *turing-riduzione* è un algoritmo che riduce un problema ad un altro problema.

4 Algoritmi di selezione

Dato in input un array A di oggetti su cui è definita una relazione di ordinamento e un indice i compreso tra 1 e n (n è il numero di oggetti nell'array), l'output dell'algoritmo è l'oggetto che si trova in posizione i nell'array ordinato.

```
selezione(A, i)
  ordina(A) // O(n log n)
  return A[i]
```

Quindi la complessità di questo algoritmo nel caso peggiore è $O(n \log n)$ (limite superiore). È possibile selezionare un elemento in tempo lineare? Analizziamo un caso particolare dell'algoritmo di selezione, ovvero la ricerca del minimo (o del massimo).

4.1 Ricerca del minimo o del massimo

In tempo lineare si può trovare il minimo e il massimo di un array:

```
minimo(A)
  min <- A[1]
  for i <- 2 to length[A]
    if A[i] < min
      min <- A[i]
  return min
```

trovare il minimo equivale a trovare `selezione(A, 1)` e trovare il massimo equivale a trovare `selezione(A, n)`. Si può però andare sotto la complessità lineare?

Per trovare il massimo (o il minimo) elemento n di un array bisogna fare **almeno** $n-1$ confronti perchè bisogna confrontare ogni elemento con l'elemento massimo (o minimo) trovato per poter dire se è il massimo (o minimo). Di conseguenza, non è possibile avere un algoritmo per la ricerca del massimo (o minimo) in cui c'è un elemento che non "perde" mai ai confronti (cioè risulta sempre il più grande) e non viene dichiarato essere il più grande (o più piccolo).

Dimostrazione: Per dimostrarlo si può prendere un array in cui l'elemento a non perde mai ai confronti, ma l'algoritmo dichiara che il massimo è l'elemento b. Allora si rilancia l'algoritmo sostituendo l'elemento a con $a =$ `max(b+1,a)` e si ripete l'algoritmo con questo secondo array in cui a è l'elemento più grande. Si ha quindi che i confronti in cui a non è coinvolto rimangono gli stessi e i confronti in cui a è coinvolto non cambiano perchè anche prima a non perdeva mai ai confronti, di conseguenza l'algoritmo dichiarerà che il massimo è b e quindi l'algoritmo non è corretto, dimostrando che non esiste un algoritmo che trova il massimo in meno di $n-1$ confronti.

Abbiamo quindi trovato che la complessità del massimo (o minimo) nel caso migliore è $\Omega(n)$ (limite inferiore) e nel caso peggiore è $O(n)$ (limite superiore). Di conseguenza la complessità è $\Theta(n)$.

4.1.1 Ricerca del minimo e del massimo contemporaneamente

Si potrebbe implementare unendo i 2 algoritmi precedenti:

```
min_max(A)
  min <- A[1]
  max <- A[1]
  for i <- 2 to length[A]
    if A[i] < min
      min <- A[i]
    if A[i] > max
      max <- A[i]
  return (min, max)
```

Questo algoritmo esegue $n-1+n-1 = 2n-2$ confronti.

- **Limite inferiore**: Potenzialmente ogni oggetto potrebbe essere il minimo o il massimo. Sia m il numero di oggetti potenzialmente minimi e M il numero di oggetti potenzialmente massimi. Sia n il numero di oggetti nell'array.
  - All'inizio $m+M = 2n$ perchè ogni oggetto può essere sia minimo che massimo.

- Alla fine $m + m = 2$ perchè alla fine ci sarà un solo minimo e un solo massimo.

Quando viene fatto un confronto $m + M$ può diminuire.

- Se si confrontano due oggetti che sono potenzialmente sia minimi che massimi, allora $m + M$ diminuisce di 2 perchè:

  $$a < b$$

  b non può essere il minimo e a non può essere il massimo e si perdono 2 potenzialità.
- Se si confrontano due potenziali minimi (o massimi), allora $m + M$ diminuisce di 1 perchè:

  $$a < b$$

  b non può essere il minimo e si perde 1 potenzialità.

Un buon algoritmo dovrebbe scegliere di confrontare sempre 2 oggetti che sono entrambi potenziali minimi o potenziali massimi.

Due oggetti che sono potenzialmente sia minimi che massimi esistono se $m+M > n+1$ perchè se bisogna distribuire n potenzialità ne avanzano due che devono essere assegnate a due oggetti che hanno già una potenzialità. Quindi fino a quando $m + M$ continua ad essere almeno $n + 2$ si riesce a far diminuire $m + M$ di 2 ad ogni confronto.

Questa diminuzione si può fare $\lfloor \frac{n}{2} \rfloor$ volte, successivamente $m + M$ potrà calare solo di 1 ad ogni confronto.

Successivamente il numero di oggetti rimane:

$$\begin{cases} n+1 & \text{se } n \text{ è dispari} \\ n & \text{se } n \text{ è pari} \end{cases}$$

- n dispari:

$$\begin{aligned} & n + 1 - 2 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= n - 1 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= \left\lfloor \frac{3}{2} n \right\rfloor - 1 \\ &= \left\lceil \frac{3}{2} n \right\rceil - 2 \end{aligned}$$

- n pari:

$$\begin{aligned} & n - 2 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= n - 2 + \frac{n}{2} \\ &= \frac{3}{2} n - 2 \\ &= \left\lceil \frac{3}{2} n \right\rceil - 2 \end{aligned}$$

Quindi la complessità è $\Omega(\lceil \frac{3}{2}n \rceil - 2) = \Omega(n)$ (limite inferiore). Meglio di così non si può fare, ma non è detto che esista un algoritmo che raggiunga questo limite inferiore.

Un algoritmo che raggiunge il limite inferiore è il seguente:

1. Dividi gli oggetti in 2 gruppi:

$$\underbrace{\begin{array}{cc} a_1 & b_1 \\ a_2 & b_2 \\ \vdots & \vdots \\ \underbrace{a_{\lfloor \frac{n}{2} \rfloor}}_{\text{Potenziali minimi}} & \underbrace{b_{\lceil \frac{n}{2} \rceil}}_{\text{Potenziali massimi}} \end{array}}_{\text{Potenziali sia minimi che massimi}}$$

2. Confronta a_i con b_i, supponendo $a_i < b_i$ (mette a sinistra i più piccoli e a destra i più grandi)
3. Cerca il minimo degli a_i e cerca il massimo dei b_i:
4. Sistema l'eventuale elemento in più

4.2 Randomized select

Si può implementare un algoritmo che divide l'array in 2 parti allo stesso modo in cui viene effettuata la `partition` di quick sort:

```
// A: Array
// p: Indice di partenza
// r: Indice di arrivo
// i: Indice che stiamo cercando (compreso tra 1 e r-p+1)
randomized_select(A, p, r, i)
  if p = r
    return A[p]
  q <- randomized_partition(A, p, r)
  k <- q - p + 1 // Numero di elementi a sinistra
  // Controlla se l'elemento cercato e' a sinistra o a destra
  if i <= k
    return randomized_select(A, p, q, i) // Cerca a sinistra
  else
    return randomized_select(A, q+1, r, i-k) // Cerca a destra
```

- Se dividessimo sempre a metà si avrebbe:

$$T(n) = n + T\left(\frac{n}{2}\right) = \Theta(n) \text{ (terzo caso del teorema dell'esperto)}$$

- Mediamente:

$$\begin{aligned} T(n) &= n + \frac{1}{n} T\left(max(1, n-1)\right) + \frac{1}{n} T\left(max(2, n-2)\right) + \dots \\ &= n + \frac{2}{n} \sum_{i=\frac{n}{2}}^{n-1} T\left(i\right) \end{aligned}$$

La complessita media è lineare.

Si esegue un solo ramo, che nel caso pessimo è quello con più elementi. La risoluzione è la stessa del quick sort.

Esiste un algoritmo che esegue la ricerca in tempo lineare anche nel caso peggiore?

Si potrebbe cercare un elemento perno più ottimale, cioè che divida l'array in **parti proporzionali**:

1. Dividi gli oggetti in $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi di 5 elementi più un eventuale gruppo con meno di 5 elementi.
2. Calcola il mediano di ogni gruppo di 5 elementi (si ordina e si prende l'elemento centrale). $\Theta(n)$
3. Calcola ricorsivamente il mediano x dei mediani
$$T\left(\left\lceil \frac{n}{5} \right\rceil\right)$$
4. Partiziona con perno x e calcola k (numero di elementi a sinistra). $\Theta(n)$
5. Se $i < k$ cerca a sinistra l'elemento i, altrimenti cerca a destra l'elemento $i - k$. La chiamata ricorsiva va fatta su un numero di elementi sufficientemente piccolo, e deve risultare un proporzione di n, quindi ad esempio dividere in gruppi da 3 elementi non funzionerebbe.
$$T(?)$$
$$\begin{array}{ccccccccccccc} m_1 & \to & m_2 & \to & m_3 & \to & \underset{x}{m_4} & \to & m_5 & \to & m_6 & \to & m_7 \\ &&&&&&&& \downarrow && \downarrow && \downarrow \\ &&&&&&&& m_{5,4} && m_{6,4} && m_{7,4} \\ &&&&&&&& \downarrow && \downarrow && \\ &&&&&&&& m_{5,5} && m_{6,5} && \end{array}$$
Gli elementi verdi sono maggiori dell'elemento x e ogni elemento verde avrà 2 elementi maggiori di esso (tranne nel caso del gruppo con meno di 5 elementi rappresentato in blu).
$$3 \cdot \left(\underbrace{\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil}_{\text{verdi + blu + rosso}} - \underbrace{2}_{\text{rosso + blu}} \right) \geq \frac{3}{10}n - 6$$
Da ogni parte si hanno almeno $\frac{3}{10}n - 6$ elementi, quindi al massimo si hanno $n - \left(\frac{3}{10}n - 6\right) = \frac{7}{10}n + 6$ elementi.

Quindi abbiamo trovato $T(?)$:
$$T(n) = \Theta(n) + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right)$$

Supponiamo $T(n) \leq cn$:

$$\begin{aligned}
T(n) &\leq c'n + c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7}{10}n + 6\right) \\
&\leq c'n + c\frac{n}{5} + c + c\left(\frac{7}{10}n + 6\right) \\
&= c\frac{7}{10}n + \frac{c}{5}n + 7c + c'n \\
&= cn - \left(\frac{3}{10}cn - \frac{c}{5}n - 7c - c'n\right) \\
&\stackrel{?}{\leq} cn
\end{aligned}$$

$$\begin{aligned}
\frac{3}{10}cn - \frac{1}{5}cn - 7c - c'n &\geq 0 \\
\frac{1}{10}cn - c'n - 7c &\geq 0 \\
\frac{1}{10}c &> c' \\
c &> 10c'
\end{aligned}$$

Quindi $T(n) \leq cn$ e quindi $T(n) = O(n)$. Il problema è che le costanti sono così alte che nella pratica è meglio il `randomized_select`.