Algoritmi

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2024/2025

Indice

1 Introduzione

Un'algoritmo è una sequenza **finita** di **istruzioni** volta a risolvere un problema. Per implementarlo nel pratico si scrive un **programma**, cioè l'applicazione di un linguaggio di programmazione, oppure si può descrivere in modo informale attraverso del **pseudocodice** che non lo implementa in modo preciso, ma spiega i passi per farlo.
Ogni algoritmo può essere implementato in modi diversi, sta al programmatore capire qual'è l'opzione migliore e scegliere in base alle proprie necessità.

1.1 Confronto tra algoritmi

Ogni algoritmo si può confrontare con gli altri in base a tanti fattori, come:

- **Complessità**: quanto ci vuole ad eseguire l'algoritmo
- **Memoria**: quanto spazio in memoria occupa l'algoritmo

1.2 Rappresentazione dei dati

Per implementare un algoritmo bisogna riuscire a strutturare i dati in maniera tale da riuscire a manipolarli in modo efficiente.

2 Calcolo della complessità

La complessità di un algoritmo mette in relazione il numero di istruzioni da eseguire con la dimensione del problema, e quindi è una funzione che dipende dalla dimensione del problema.

La **dimensione del problema** è un insieme di oggetti adeguato a dare un idea chiara di quanto è grande il problema da risolvere, ma sta a noi decidere come misurare il problema.
Ad esempio una matrice è più comoda da misurare come il numero di righe e il numero di colonne, al posto di misurarla come il numero di elementi totali.

La complessità di solito si calcola come il **caso peggiore**, cioè il limite superiore di esecuzione dell'algoritmo.

2.1 Linguaggi di programmazione

Ogni linguaggio di programmazione è formato da diversi blocchi:

1. **Blocco iterativo**: un tipico blocco di codice eseguito sequenzialmente e tipicamente finisce con un punto e virgola.
2. **Blocco condizionale**: un blocco di codice che viene eseguito solo se una condizione è vera.
3. **Blocco iterativo**: un blocco di codice che viene eseguito ripetutamente finché una condizione è vera.

Questi sono i blocchi base della programmazione e se riusciamo a calcolare la complessità di ognuno di questi blocchi possiamo calcolare più facilmente la complessità di un intero algoritmo.

2.1.1 Blocchi iterativi

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

Se ogni blocco ha complessità $c_1(n)$, allora la complessità totale è data da:

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

2.1.2 Blocchi condizionali

$$\begin{array}{ll} \text{IF cond} & c_{cond}(n) \\ \quad I_1 & c_1(n) \\ \text{ELSE} & \\ \quad I_2 & c_2(n) \end{array}$$

La complessità totale è data da:

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

A volte la condizione è un test sulla dimensione del problema e in quel caso si può scrivere una complessità più precisa.

2.1.3 Blocchi iterativi

$$\begin{array}{ll} \text{WHILE cond} & c_{cond}(n) \\ \quad I & c_0(n) \end{array}$$

Si cerca di trovare un limite superiore m al limite di iterazioni.

Di conseguenza la complessità totale è data da:

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

2.2 Esempio

Esempio 2.1. Calcoliamo la complessità della moltiplicazione tra 2 matrici:

$$A_{n\times m} \cdot B_{m\times l} = C_{n\times l}$$

L'algoritmo è il seguente:

```
for i <- 1 to n // n ( 5 ml + 4l + 2) + n + 1
  for j <- 1 to l // l (5m + 2 + 1) + 1 + l
    c[i][j] <- 0
    for k <- 1 to m // (m + 1 + m(4))
      // 3 (moltiplicazione, somma e assegnamento)
      // 1 (incremento for)
      c[i][j] += a[i][k] * b[k][j]
```

Partiamo calcolando la complessità del ciclo for più interno. Non ha senso tenere in considerazione tutti i dati, ma solo quelli rilevanti. In questo caso avremo:

$$(m + 1 + m(4)) = 5m + 1$$

Questa complessità contiene informazioni poco rilevanti perchè possono far riferimento alla velocità della cpu e un millisecondo in più o in meno non cambia nulla se teniamo in considerazione solo l'incognita abbiamo:

$$m$$

Questo semplifica molto i calcoli, rendendo meno probabili gli errori. Siccome la complessità si calcola su numeri molto grandi, le costanti piccole prima o poi verranno tolte perchè poco influenti.

La complessità totale alla fine sarebbe stata:

$$5nml + 4ml + 2n + n + 1$$

Ma ciò che ci interessa veramente è:

$$5\textcolor{red}{nml} + 4ml + 2n + n + 1$$

Se non consideriamo le costanti inutili, la complessità finale è:

$$nml$$

Nella maggior perte dei casi ci si concentra soltanto sull'ordine di grandezza della complessità, e non sulle costanti.

2.3 Ordine di grandezza

L'ordine di grandezza è una funzione che approssima la complessità di un algoritmo:

$$f \in O(g)$$

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

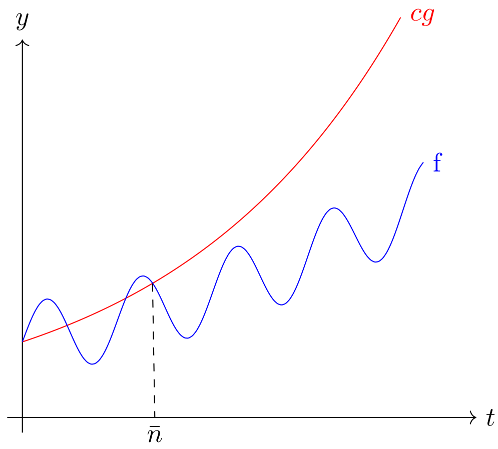

Figura 1: Esempio di funzione $f \in O(g)$

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

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

Per gli algoritmi:

Definizione 2.1.

$$A \in O(f)$$

So che l'algoritmo A termina entro il tempo definito dalla funzione f. Di conseguenza se un algoritmo termina entro un tempo f allora sicuramente termina entro un tempo g più grande. Ad esempio:

$$A \in O(n) \Rightarrow A \in O(n^2)$$

Questa affermazione è **corretta**, ma **non accurata**.

$$A \in \Omega(f)$$

Significa che esiste uno schema di input tale che se $g(n)$ è il numero di passi necessari per risolvere l'istanza n allora:

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

Quindi l'algoritmo non termina in un tempo minore di f.

Calcolando la complessità si troverà lo schema di input tale che:

$$g \in O(f)$$

cioè il limite superiore di esecuzione dell'algoritmo.
Successivamente ci si chiede se esistono algoritmi migliori e si troverà lo schema di input tale che:

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

cioè il limite inferiore di esecuzione dell'algoritmo.
Se i due limiti coincidono allora:

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

abbiamo trovato il tempo di esecuzione dell'algoritmo.

Teorema 2.1 (Teorema di Skolem)**.** Se c'è una formula che vale coi quantificatori esistenziali, allora nel linguaggio si possono aggiungere delle costanti al posto delle costanti quantificate e assumere che la formula sia valida con quelle costanti.

2.3.1 Esempi di dimostrazioni

Esempio 2.2. È vero che $n \in O(2n)$?
Se prendiamo $c = 1$ e $\bar{n} = 1$ allora:

$$n \leq c2n$$

Quindi è vero

Esempio 2.3. È vero che $2n \in O(n)$?
Se prendiamo $c = 2$ e $\bar{n} = 1$ allora:

$$2n \leq 2n$$

Quindi è vero

Esempio 2.4. È vero che $f \in O(g) \iff g \in \Omega(f)$?
Dimostro l'implicazione da entrambe le parti:

- $\rightarrow$: Usando il teorema di Skolem:

$$\forall n \geq \bar{n} \;\; f(n) \leq cg(n)$$

Trasformo la disequazione:

$$\forall n \geq \bar{n} \;\; \frac{f(n)}{c} \leq g(n)$$

$$\forall n \geq \bar{n} \;\; g(n) \geq \frac{f(n)}{c}$$

$$\forall n \geq \bar{n} \;\; g(n) \geq \frac{1}{c} f(n) \quad \square$$

Se la definizione di $\Omega(g)$ è:

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

sappiamo che:

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

- $\leftarrow$: Usando il teorema di Skolem:

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

Trasformo la disequazione:

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

$$\forall n \geq \bar{n}' \;\; f(n) \leq \frac{1}{c'} g(n) \quad \square$$

Esempio 2.5.

$$f_1 \in O(g)\ f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$$

Dimostrazione:
Ipotesi

$$\bar{n}_1 c_1\ \forall n > n_1 \quad f_1(n) \leq c_1 g(n)$$

$$\bar{n}_1 c_2\ \forall n > n_2 \quad f_2(n) \leq c_2 g(n)$$

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

Quindi:

$$c = (c_1 + c_2)$$

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

Esempio 2.6. Se

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

è vero che:

$$f_1 \cdot f_2 \in O(g_1 \cdot g_2)$$

Dimostrazione:
Ipotesi

$$\bar{n}_1 c_1\ \forall n > \bar{n}_1 \quad f_1(n) \leq c_1 g_1(n)$$

$$\bar{n}_2 c_2\ \forall n > \bar{n}_2 \quad f_2(n) \leq c_2 g_2(n)$$

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

Quindi:

$$c = c_1 \cdot c_2$$

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

2.4 Studio degli algoritmi

Il problema dell'ordinamento si definisce stabilendo la relazione che deve esistere tra **input** e **output** del sistema.

- **Input**: Sequenza $(a_1, \ldots, a_n)$ di oggetti su cui è definita una relazione di ordinamento, cioè l'unico modo per capire la differenza tra due oggetti è confrontarli.
- **Output**: Permutazione $(a'_1, \ldots, a'_n)$ di $(a_1, \ldots, a_n)$ tale che:

$$\forall i < j\ \ a'_i \leq a'_j$$

L'obiettivo è trovare un algoritmo che segua la relazione di ordinamento definita e risolva il problema nel minor tempo possibile.

2.4.1 Insertion sort

Divide la sequenza in due parti:

- **Parte ordinata**: Sequenza di elementi ordinati
- **Parte non ordinata**: Sequenza di elementi non ordinati

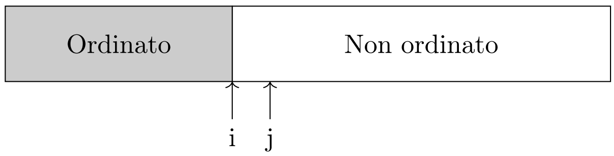

Figura 2: Parte ordinata e non ordinata

Pseudocodice:

```
insertion_sort(A)
  for j <- 2 to length[A] // A sinistra di j e tutto ordinato-
    key <- a[j]             //                              |
    i <- j - 1              //                              | O(n)
    while i > 0 and a[i] > key // --                        |
      a[i + 1] <- a[i]      //   | O(n)                     |
      i--                   // --                       -----
    a[i + 1] <- key
```

La complessità di questo algoritmo è:

$$O(n^2)$$

Per capirlo è sufficiente guardare il numero di cicli nidificati e quante volte eseguono il codice all'interno.

Se l'array è già ordinato la complessità è:

$$\Omega(n)$$

Con l'input peggiore possibile la complessità è:

$$\Omega(n^2)$$

di conseguenza, visto che vale $O(n^2)$ e $\Omega(n^2)$ vale:

$$\Theta(n^2)$$

Quanto spazio in memoria utilizza questo algoritmo?

- Variabile j
- Variabile i
- Variabile key

A prescindere da quanto è grande l'array utilizzato, di conseguenza la memoria utilizzata è costante.

- **Ordinamento in loco**: se la quantità di memoria extra che deve usare non dipende dalla dimensione del problema allora si dice che l'algoritmo è in loco.
- **Ordinamento non in loco**: se la quantità di memoria extra che deve usare dipende dalla dimensione del problema allora si dice che l'algoritmo è non in loco.
- **Stabilità**: La posizione relativa di elementi uguali non viene modificata

L'insertion sort ordina in loco ed è stabile.

2.4.2 Fattoriale

```
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 2.7.

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

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:

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

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

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 2.8.

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

$$T(n) \leq cn$$

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

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 2.9.

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

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:

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

2.5 Teorema dell'esperto

Teorema 2.2 (Teorema dell'esperto o Master theorem)**.** Per un'equazione di ricorrenza del tipo:

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

Si distinguono 3 casi:

- $f(n)\in O(n^{\log_b a-\varepsilon})$ allora $T(n);\in\Theta(n^{\log_b a})$
- $f(n)\in\Theta(n^{\log_b a})$ allora $T(n)\in\Theta(f(n)\log n)$
- $f(n)\in\Omega(n^{\log_b a+\varepsilon})$ allora $T(n)\in\Theta(f(n))$

Esempio 2.10.

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

Applico il teorema dell'esperto:

$$\begin{aligned}
a&=9\\
b&=3\\
f(n)&=n
\end{aligned}$$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Verifico se esiste un ε tale che:

$$n \in O(n^{2-\varepsilon})$$

prendo $\varepsilon = -\frac{1}{2}$ e verifico:

$$n \in O(n^2 \cdot n^{-\frac{1}{2}})$$

Quindi ho trovato il caso 1 del teorema dell'esperto.

$$T(n) \in \Theta(n^2)$$

Esempio 2.11.

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

Applico il teorema dell'esperto:

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

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

Si nota che le due funzioni hanno lo stesso ordine di grandezza, quindi siamo nel secondo caso del teorema dell'esperto.

$$T(n) \in \Theta(\log n)$$

Esempio 2.12.

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

Applico il teorema dell'esperto:

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

$$n^{\log_b a} = n^{\log_4 3}$$

$$n \log n \in \Omega(n^{\log_4 3})$$

Esiste un ε tale che:

$$n \log n \in \Omega(n^{\log_4 3+\varepsilon})$$

perchè basta che sia compreso tra $\log_4 3$ e 1.

Quindi siamo nel terzo caso del teorema dell'esperto.

$$T(n) \in \Theta(n \log n)$$

Esempio 2.13.

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

Applico il teorema dell'esperto:

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

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$n \log n \in \Omega(n)$$

Verifico se esiiste un ε, quindi divido per n:

$$\log n \in \Omega(n^{\varepsilon})$$

Quindi si nota che questa proprietà non è verificata, quindi non si può applicare il teorema dell'esperto.

2.5.1 Merge sort

Questo algoritmo di ordinamento è basato sulla tecnica divide et impera:

- **Divide**: Dividi il problema in sottoproblemi più piccoli
- **Impera**: Risolvi i sottoproblemi in modo ricorsivo
- **Combina**: Unisci le soluzioni dei sottoproblemi per risolvere il problema originale

Questo algoritmo divide la sequenza in due parti uguali e le ordina separatamente, successivamente le unisce in modo ordinato. La complessità, comsiderando il merge con complessità lineare, risulta:

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

Applicando il teorema dell'esperto si ottiene:

$$a = 2$$
$$b = 2$$
$$f(n) = n$$

$$n^{\log_b a} = n$$

$$n \in \Theta(n)$$

Quindi siamo nel secondo caso del teorema dell'esperto:

$$T(n) \in \Theta(n \log n)$$

Definizione del merge sort:

```
// A: Array da ordinare
// P: Indice di partenza
// r: Indice di arrivo
merge_sort(A, p, r)          // --
  if p < r                   //   |
    q <- floor((p + r) / 2)  //   |
    merge_sort(A, p, q)      //   | O(n log n)
    merge_sort(A, q + 1, r)  //   |
    merge(A, p, q, r)        // --
```

```
// A: Array da ordinare
// P: Indice di partenza
// q: Indice di mezzo
// r: Indice di arrivo
merge(A, p, q, r)
  i <- 1
  j <- p
  k <- q + 1
  // Ordina gli elementi di A in B
  while(j <= q and k <= r)                       // --
    if j <= q and (k > r or A[j] <= A[k])        //   |
      B[i] <- A[j]                               //   |
      j++                                        //   |
    else                                         //   | O(n)
      B[i] <- A[k]                               //   |
      k++                                        //   |
    i++                                          // --

  // Copia gli elementi di B in A
  for i <- 1 to r - p + 1                        // -|
    A[p + i - 1] <- B[i]                         // -| O(n)
```

L'algoritmo è stablie perchè non vengono scambiati elementi uguali e non è in loco perchè utilizza un array di appoggio.

2.6 Heap

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

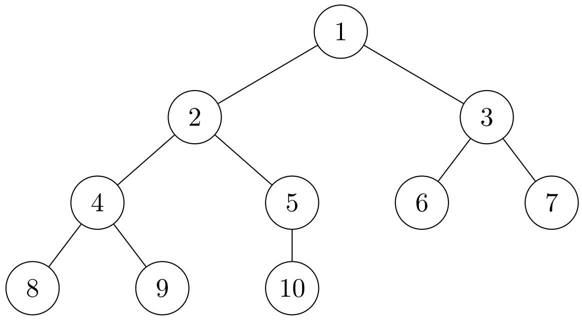

Figura 3: Heap con l'indice di un array associato ai nodi

2.6.1 Proprietà

$\forall$ nodo il contenuto del nodo è $\geq$ del contenuto dei figli. Per calcolare il numero di nodi di un albero binario si usa la formula:

$$N = 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = \frac{1-2^h}{1-2} = 2^h - 1$$

dove h è l'altezza dell'albero. Il numero di foglie di un albero sono la metà dei nodi.

```
extract_max(A)
  H[1] <- H[H.heap_size]
  H.heap_size <- H.heap_size - 1
  heapify(H,1)
```

```
heapify(A, i) // O(n)
  l <- left[i] // Indice del figlio sinistro (2i)
  r <- right[i] // Indice del figlio destro (2i+1)
  if l < H.heap_size and H[l] > H[i]
    largest <- l
  else
    largest <- i

  if r < H.heap_size and H[r] > H[largest]
    largest <- r
  if largest != i
    swap(H[i], H[largest])
    heapify(H, largest)
```

Ora si vuole definire una funzione che costruisce un heap da un array:

```
build_heap(A) // O(n)
  heapsize(a) <- length[A]
  for i <- floor(length[A]/2) downto 1
    heapify(A, i)
```