Algoritmi

UniVR - Dipartimento di Informatica

Fabio Irimie

Corso di Roberto Segala
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 sequenziale**: 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

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

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

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

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:

```
1
2   for i <- 1 to n // n ( 5 ml + 4l + 2) + n + 1
3     for j <- 1 to l // l (5m + 2 + 1) + 1 + l
4       c[i][j] <- 0
5       for k <- 1 to m // (m + 1 + m(4))
6         // 3 (moltiplicazione, somma e assegnamento)
7         // 1 (incremento for)
8         c[i][j] += a[i][k] * b[k][j]
9
```

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:

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

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:

$$f \in O(g)$$

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

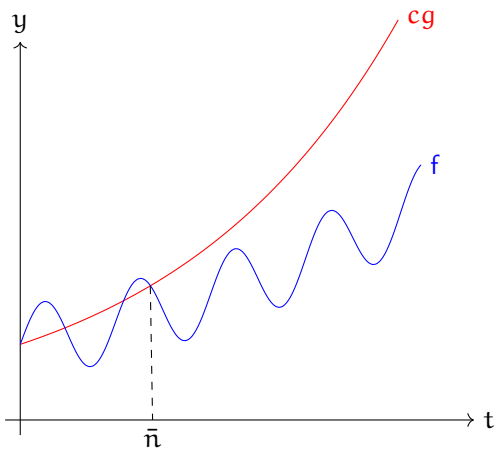


Figura 1: Esempio di funzione $f \in O(g)$

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

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

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:

$$n \leqslant c2n$$

Quindi è vero

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

$$2n \leqslant 2n$$

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:

  $$\forall n \geqslant \bar{n} \;\; f(n) \leqslant cg(n)$$

  Trasformo la disequazione:

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

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

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

  Se la definizione di $\Omega(g)$ è:

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

  sappiamo che:

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

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

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

  Trasformo la disequazione:

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

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

Esempio 2.5.

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

Dimostrazione:
Ipotesi

$$\bar{n}_1 c_1\ \forall n > n_1 \quad f_1(n) \leqslant c_1 g(n)$$
$$\bar{n}_1 c_2\ \forall n > n_2 \quad f_2(n) \leqslant c_2 g(n)$$

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

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

$$\bar{n}_1 c_1\ \forall n > \bar{n}_1 \quad f_1(n) \leqslant c_1 g_1(n)$$
$$\bar{n}_2 c_2\ \forall n > \bar{n}_2 \quad f_2(n) \leqslant c_2 g_2(n)$$

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

Quindi:

$$c = c_1 \cdot c_2$$
$$\bar{n} = \max(\bar{n}_1, \bar{n}_2)$$

3 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:

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

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

3.1 Insertion sort

Divide la sequenza in due parti:

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

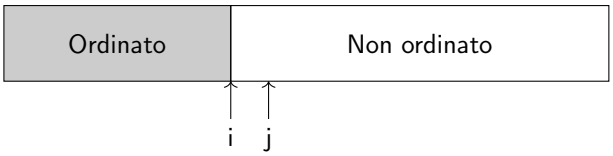


Figura 2: Parte ordinata e non ordinata

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.

3.2 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 3.1.

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

Questa funzione si può riscrivere come:

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

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:

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

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

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 3.2.

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

$$T(n) \leqslant cn$$

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

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:

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

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

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

Esempio 3.3.

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

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:

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

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

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

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:

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

3.3 Teorema dell'esperto

Teorema 3.1 (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 3.4.

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

Applico il teorema dell'esperto:

$$a=9$$
$$b=3$$
$$f(n)=n$$

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

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 3.5.

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

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 3.6.

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

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 3.7.

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

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.

3.4 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, considerando 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.

3.5 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.

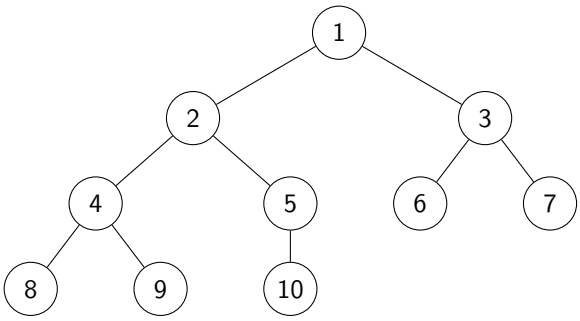


Figura 3: Heap con l'indice di un array associato ai nodi

3.5.1 Proprietà

$\forall$ nodo il contenuto del nodo è $\geqslant$ 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.

Definiamo una funzione che "aggiusta" i figli di un nodo per mantenere la proprietà di heap:

```
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)
```

Una volta definito un heap si possono fare diverse operazioni, come ad esempio estrarre il nodo massimo:

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

3.5.2 Heap sort

Heap sort è un algoritmo di ordinamento basato su heap.

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

La complessità dell'algoritmo è precisamente:

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

Per la formula di Stirling $n!$ ha ordine di grandezza n^n. Questo algoritmo è in loco e instabile.

Il caso pessimo è un array ordinato al contrario ($O(n \log n)$) e il caso migliore è un array già ordinato ($\Omega(n \log n)$), quindi la complessità è:

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

3.6 Quick sort

Il concetto di questo algoritmo è quello di mettere prima in disordine l'algoritmo e poi ordinarlo. L'algoritmo divide l'array in 2 parti e ordina ricorsivamente le due parti; a quel punto l'array è ordinato.

```
// A: Array da ordinare
// p: Indice di partenza
// r: Indice di arrivo
quick_sort(A, p, r)
  if p < r // Ordina solo se l'array ha piu' di un elemento
    q <- partition(A, p, r) // Dividi l'array in due parti
    quick_sort(A, p, q) // Ordina sinistra
    quick_sort(A, q + 1, r) // Ordina destra
```

```
partition(A, p, r)
  x <- A[p] // Elemento perno (o pivot)
  i <- p - 1
  j <- r + 1
  while true
    repeat // Ripete finche' la condizione non e' soddisfatta
      j--  // n/2
    until A[j] <= x // Trova un elemento che non puo' stare a
  destra
    repeat
      i++  // n/2
    until A[i] >= x // Trova un elemento che non puo' stare a
  sinistra
    if i < j
      swap(A[i], A[j]) // n/2
    else
      return j // alla fine j puntera' all'ultimo elemento di
  sinistra
```

Questo algoritmo è in loco e non è stabile. La sua complessità nel caso peggiore è:

$$\begin{aligned} T(n) &= T(\text{partition}) + T(q) + T(n-q) \\ &= n + T(q) + T(n-q) \\ &= n + \cancel{T(1)} + T(n-1) \\ &= n + T(n-1) \\ &= \Theta(n^2) \end{aligned}$$

Il caso peggiore è un array già ordinato.

Mediamente ci si aspetta che l'array venga diviso in 2 parti molto simili, quindi la complessità è $O(n \log n)$ perchè:

$$0 < c < 1$$

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

```
1  rand_partition(A, p, r)
2    i <- random(p, r)
3    swap(A[i], A[p])
4    return partition(A, p, r)
```

La complessità è la media di tutte le complessità con probabilità $\frac{1}{n}$

$$\begin{aligned} T(n) = n + \frac{1}{n}(T(1) + T(n-1)) + \frac{n-1}{n}(T(2) + T(n-2)) \\ + \ldots + \frac{1}{n}(T(n-1) + T(1)) \end{aligned}$$

$$\begin{aligned} T(n) &= n + \frac{1}{n}\sum_i (T(i) + T(n-i)) \\ &= n + \frac{2}{n}\sum_i T(i) \end{aligned}$$

3.7 Algoritmi di ordinamento non per confronti

Si possono avere algoritmi di ordinamento con complessità $< n \log n$?

Qualsiasi algoritmo che ordina per confronti deve fare almeno $n \log n$ confronti nel caso pessimo

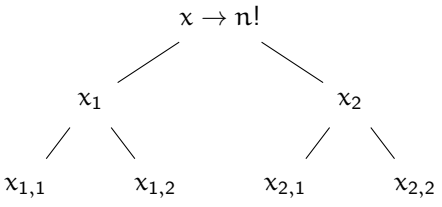


Figura 4: Heap con l'indice di un array associato ai nodi

Le foglie rappresentano ogni singola combinazione possibile. Il numero di foglie è $n!$ e l'altezza sarà sempre

$$h \geqslant \log_2 n! = n \log n$$

3.7.1 Counting sort

Si vogliono ordinare n numeri con valori da 1 a k. L'idea di questo algoritmo è quella di creare un'array che contiene il numero di occorrenze di un certo valore (rappresentato dall'indice).

```
counting_sort(A, k)
  for i <- 1 to k // k
    C[i] <- 0 // Inizializzazione di un array a 0

  for j <- 1 to length[A] // n
    C[A[j]]++ // Conteggio delle occorrenze

  // k
  for i <- 2 to k          // In ogni indice metto il numero di
    C[i] <- C[i] + C[i-1] // elementi minori o uguali
                           // al numero dell'indice
  // Alla fine l'array C conterra' l'ultima posizione di occorrenza
     per ogni elemento

  for j <- length[A] downto 1 // n
    B[C[A[j]]] <- A[j] // Inserimento dell'elemento in posizione
    C[A[j]]--          // Decremento della posizione di occorrenza
```

La complessità di questo algoritmo è $O(n + k)$ e siccome sappiamo che k è una costante fissata a priori la complessità è $O(n)$. Non è in loco, ma è stabile

3.7.2 Radix sort

Il radix sort è un ordinamento lessico grafico, cioè si ordinano le cifre partendo da quella meno significativa e se sono uguali si passa a quella più significativa.

La complessità dell'algoritmo è:

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

dove:

$$l = \text{numero di cifre} = \log_k n$$
$$n = \text{numero di elementi}$$
$$k = \text{numero di valori possibili}$$

Se rappresentiamo i numeri in base n, allora si avrà la seguente rappresentazione:

$$\ldots \; n^2 \; n^1 \; n^0$$

e ad esempio per rappresentare $n^2 - 1$ valori possibili serviranno 2 cifre. cifre.

3.7.3 Bucket sort

Dato un array di numeri con **supporto infinito** e **distribuzione nota**, si può dividere l'array in k parti (bucket) uguali (equiprobabili) e ordinare ricorsivamente. Ogni coppia di gruppi deve essere totalmente ordinata, cioè ogni elemento del primo gruppo deve essere minore di ogni elemento del secondo gruppo. Una volta ordinati i gruppi (con un algoritmo di ordinamento a scelta) si concatenano in modo ordinato.

Il caso peggiore è quello in cui tutti gli elementi finiscono in un singolo bucket, la probabilità che questo accada è molto bassa:

$$\underbrace{\frac{1}{n} \cdot \frac{1}{n} \cdot \dots \cdot \frac{1}{n}}_{n-1} = \frac{1}{n^{n-1}}$$

e la sua complessità diventa:

$$O(n^2)$$

Nel caso medio si ha che per creare i bucket si ha una complessità $O(n)$ e per assegnare gli elementi ai bucket si ha una complessità $O(n)$. Per ogni bucket ci si aspetta che il numero di elementi al suo interno sia una **costante**, quindi **indipendente dal valore di** n. Per ordinare un bucket si ha una complessità $O(1)$ siccome il numero di elementi è costante. La complessità totale è quindi:

$$\Theta(n)$$

Formalizzando si ha:

Sia X_{ij} la variabile casuale 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 può scrivere:

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

La complessità di questo algoritmo sarà quindi:

$$C = \sum_j (N_j)^2$$

Per ottenere il valore medio della complessità:

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

sappiamo che $N_j = \sum_i X_{ij}$, quindi la media è:

$$E[N_j] = \sum_j^n E[X_{ij}] = \sum_j^n \frac{1}{n} = 1$$

e la varianza è:

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

La complessità diventa:

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

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 $a = \texttt{max(b+1,a)}$ 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:

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

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 se l'array è dispari

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:

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

  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).

$$\#\text{left} \leqslant 3 \cdot \left(\underbrace{\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil}_{\text{verdi + blu + rosso}} - \underbrace{2}_{\text{rosso + blu}} \right) = \frac{7}{10}n + 6$$

$$\#\text{right} \geqslant 3 \cdot \left(\underbrace{\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil}_{\text{verdi + blu + rosso}} - \underbrace{2}_{\text{rosso + blu}} \right) = \frac{7}{10}n + 6$$

Da ogni parte si hanno almeno $\frac{7}{10}n + 6$ elementi.

Quindi abbiamo trovato $T(?)$:

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

Dimostriamo con il metodo di sostituzione, supponendo $T(n) \leqslant cn$, che la disequazione sia vera:

$$T(n) \leqslant n + c\left\lceil \frac{n}{5} \right\rceil + c(\frac{7}{10}n + 6)$$

Non sappiamo se $\frac{7}{10}n + 6$ è minore di n, quindi lo calcoliamo:

$$\begin{aligned} \frac{7}{10}n + 6 &\leqslant n \\ 7n + 60 &\leqslant 10n \\ 3n &\geqslant 60 \\ n &\geqslant 20 \end{aligned}$$

Quindi per valori di $n \leqslant 20$ la disequazione non è vera. Consideriamo quindi $\bar{n} > 20$ e $n > \bar{n}$. Togliendo l'approssimazione per eccesso si ha:

$$\begin{aligned} T(n) &\leqslant n + c + \frac{c}{5}n + \frac{7}{10}n + 6c \\ &\leqslant \frac{9}{10}cn + 7c + n \\ &\overset{?}{\leqslant} cn \\ &= cn + \left(-\frac{1}{10}cn + 7c + n\right) \leqslant cn \text{ quando} \\ &\left(n + 7c - \frac{1}{10}cn\right) \leqslant cn \end{aligned}$$

Quindi $T(n) \leqslant cn$ e quindi $T(n) = O(n)$. Abbiamo trovato un limite superiore e un limite inferiore, quindi la complessità è $T(n) = \Theta(n)$. Il problema è che le costanti sono così alte che nella pratica è meglio il `randomized_select`.

Esiste un modo per strutturare meglio le informazioni nel calcolatore per trovare l'elemento cercato in tempo $\log n$? Si possono implementare delle **Strutture dati** che permettono di fare ricerche in tempo logaritmico.

5 Strutture dati

Una struttura dati è un modo per organizzare i dati in modo da poterli manipolare in modo efficiente. Bisogna avere un modo per comunicare con le strutture dati, senza dover sapere come sono implementate.

5.1 Stack

Ad esempio se consideriamo uno stack, si possono individuare le seguenti operazioni:

- `new()`: Crea uno stack vuoto
- `push(S,x)`: Inserisce un elemento x nello stack S
- `top(S)`: Restituisce l'elemento in cima allo stack S
- `pop(S)`: Rimuove l'elemento in cima allo stack S
- `is_empty()`: Restituisce vero se lo stack è vuoto

Da queste operazioni si possono definire certe proprietà dello stack:

- `top(push(S,x)) = x`
- `pop(push(S,x)) = S`

Questo ci dice che lo stack è LIFO (Last In First Out).

Abbiamo quindi definito un'algebra dei termini da cui possono definire tutte le operazioni possibili, ad esempio uno stack è definito come una sequenza di push:

```
push(push(push(empty(),1),2),3)
```

5.2 Queue

Una coda è una struttura dati in cui si possono inserire elementi in fondo e rimuoverli dall'inizio. Le operazioni possibili sono:

- `new()`: Crea una coda vuota
- `enqueue(Q,x)`: Inserisce un elemento x in fondo alla coda Q
- `dequeue(Q)`: Rimuove l'elemento in cima alla coda Q
- `front(Q)`: Restituisce l'elemento in cima alla coda Q
- `is_empty()`: Restituisce vero se la coda è vuota

5.3 Lista doppiamente puntata

Una lista doppiamente puntata è una struttura dati in cui ogni nodo ha un puntatore al nodo precedente e al nodo successivo.

Per rimuovere un elemento x dalla lista bisogna:

1. Trovare il nodo x
2. Collegare il nodo precedente a x con il nodo successivo a x

Il problema di questa soluzione è che bisogna anche gestire i casi degli estremi separatamente. Per evitare di gestire i casi specifici si possono aggiungere dei nodi sentinella all'inizio e alla fine della lista.

5.4 Albero binario

Un albero binario è una struttura dati in cui ogni nodo ha al massimo 2 figli.

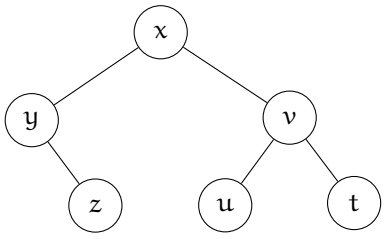


Figura 5: Albero binario

Le operazioni possibili su un albero binario sono:

- `new()`: Crea un albero vuoto
- `insert(T, x)`: Crea un nuovo nodo con valore x e lo aggiunge all'albero T
- `extract(T, x)`: Rimuove il nodo con valore x dall'albero T
- `is_empty()`: Restituisce vero se l'albero è vuoto
- `left(T)`: Restituisce il figlio sinistro dell'albero T

- `right(T)`: Restituisce il figlio destro dell'albero T
- `value(T)`: Restituisce il valore del nodo dell'albero T

Un albero è **bilanciato** quando la differenza tra l'altezza del sottoalbero sinistro e di quello destro è al massimo 1. Un albero è **completo** quando tutti i livelli sono completi, cioè tutti i nodi sono presenti, tranne l'ultimo, che può essere incompleto.

La profondità di un albero di n nodi è:

$$P(n) = 1 + P\left(\left\lceil \frac{n-1}{2} \right\rceil\right) = \Theta(\log n)$$

5.4.1 Albero binario di ricerca

Un albero binario di ricerca è un albero binario in cui per ogni nodo x valgono le seguenti proprietà:

- Tutti i nodi nel sottoalbero sinistro di x hanno valore minore di x
- Tutti i nodi nel sottoalbero destro di x hanno valore maggiore di x

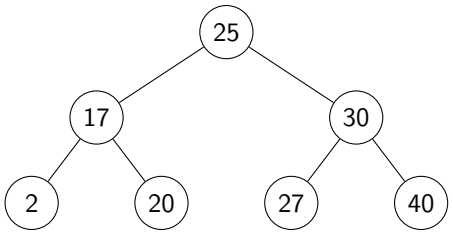


Figura 6: Albero binario di ricerca

Le operazioni possibili su un albero binario di ricerca sono:

- `insert(T, x)`: Inserisce un nodo con valore x nell'albero T
- `extract(T, x)`: Rimuove il nodo con valore x dall'albero T
- `search(T, x)`: Restituisce vero se il valore x è presente nell'albero T
- `min(T)`: Restituisce il valore minimo dell'albero T
- `max(T)`: Restituisce il valore massimo dell'albero T

Non è possibile inserire un nodo in un albero binario di ricerca in tempo logaritmico perchè potrebbe dover essere inserito in modo da sbilanciare l'albero, quindi per riottenere un albero bilanciato non rimane che spostare tutti gli elementi in una posizione diversa.

Per ottenere la complessità logaritmica, bisogna utilizzare un albero che si può sbilanciare, ma non troppo.

- **Inserimento** Per inserire un nodo in un albero binario di ricerca si può procedere nel seguente modo:

1. Si parte dalla radice e si scende fino a trovare il nodo in cui inserire il nuovo nodo rispettando le proprietà dell'albero binario di ricerca
2. Si inserisce il nodo in quel punto

- **Rimozione** Per rimuovere un elemento da un albero binario di ricerca si possono avere 3 casi:

1. Il nodo da rimuovere è una foglia: si può rimuovere direttamente
2. Il nodo da rimuovere ha un solo figlio: si può sostituire il nodo con il figlio
3. Il nodo da rimuovere ha due figli: si può sostituire il nodo con il minimo del sottoalbero destro o con il massimo del sottoalbero sinistro

Una volta che un nodo viene aggiunto o rimosso non si può più essere certi che l'albero sia ancora bilanciato, quindi bisogna fare in modo che l'albero venga bilanciato ogni volta che viene modificato e la complessità non sarà più logaritmica.

5.4.2 RB-Tree (Red-Black Tree)

Gli alberi rosso-neri sono alberi binari di ricerca in cui ogni nodo può essere rosso o nero.

- **Nero**: Il nodo nero indica che il nodo è a posto
- **Rosso**: Il nodo rosso è un nodo ausiliario

Questo tipo di albero ha le seguenti proprietà:

1. Ogni nodo è rosso o nero
2. Ogni foglia è nera
3. I figli di un nodo rosso sono neri
4. Ogni cammino dalla radice ad una foglia contiene lo stesso numero di nodi neri

Esempio 5.1. Abbiamo un albero con l nodi neri nel sotto albero destro, quindi per la proprietà 4, il sotto albero sinistro deve avere l nodi neri:

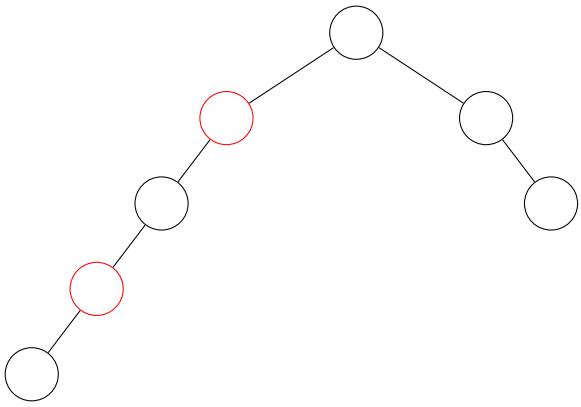


Figura 7: Albero rosso-nero

Il sotto albero sinistro avrà $2l$ nodi totali.

Esempio 5.2. Prendiamo ad esempio il seguente albero RB:

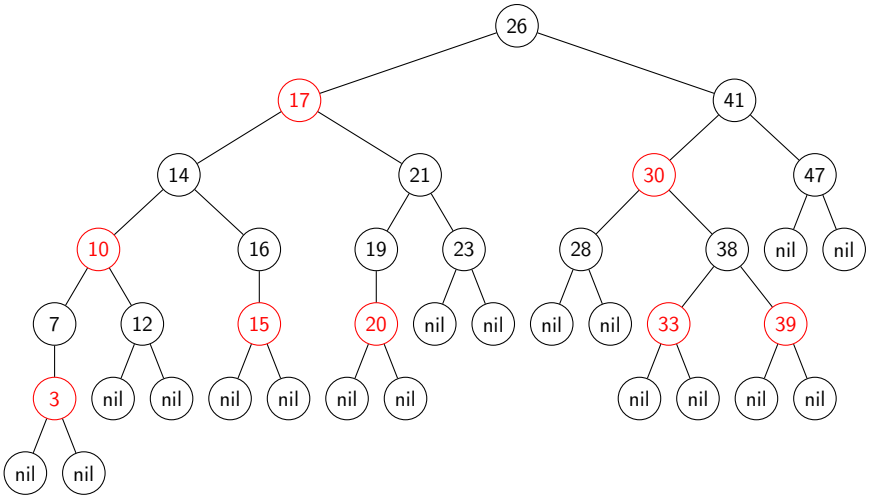


Figura 8: Albero rosso-nero

Tutte le proprietà sono verificate, però tutte le foglie sono dei `nil` (nero) e questo è uno spreco di memoria, perchè metà dei nodi di un albero sono foglie e quindi metà dei nodi sono utilizzati per la sentinella.

Questo si può risolvere facendo puntare tutte le sentinelle allo stesso valore `nil`, però facendo così non si riesce più a risalire all'elemento padre di una foglia.

Il concetto principale di questo tipo di alberi è quello della **black height**, cioè il numero di nodi neri che si incontrano lungo un cammino dalla radice ad una foglia. Prendendo in considerazione l'esempio 5.2 i nodi:

- **nil**: Hanno black height 0
- **3**: Ha black height 1
- **19**: Ha black height 1
- **14**: Ha black height 2
- **26**: Ha black height 3
- ecc...

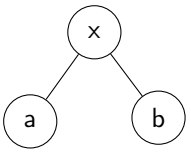
Lemma:
Per ogni nodo x dell'albero, il sottoalbero radicato in x contiene almeno $2^{bh(x)} - 1$ nodi interni (nodi che non sono una foglia).

Dimostrazione:
Dimostriamo per induzione su $bh(x)$:

- **Caso base**: Se x è una foglia, allora $bh(x) = 0$ e il sottoalbero radicato in x contiene 0 nodi interni.

$$2^{bh(x)} - 1 = 2^0 - 1 = 0$$

- Se x è un nodo con un figlio destro b e un figlio sinistro a allora:



$$bh(a) \geqslant bh(x) - 1 \quad \text{e} \quad bh(b) \geqslant bh(x) - 1$$
$$\#nodi(x) \geqslant \#nodi(a) + \#nodi(b) + 1$$
$$\Downarrow$$
$$\begin{aligned}\#nodi(x) &\geqslant 2^{bh(a)} - 1 + 2^{bh(b)} - 1 + 1 \\ &\geqslant 2^{bh(x)-1} - 1 + 2^{bh(x)-1} \cancel{-1+1} \\ &= 2 \cdot 2^{bh(x)-1} - 1 \\ &= 2^{bh(x)} - 1\end{aligned}$$

Quindi
$$\#nodi(x) \geqslant 2^{bh(x)} - 1$$

Complessità: Consideriamo un albero RB di altezza h (distanza tra la radice e la foglia più lontana, escludendo la radice) e radice x, si ha che $bh(x)$ vale:

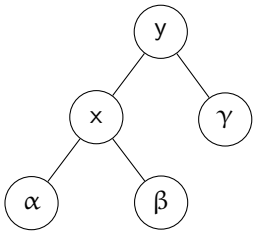
$$bh(x) \geqslant \frac{h}{2}$$

Il numero di nodi n (per il lemma) vale $2^{\frac{h}{2}}$, si ha quindi che l'altezza di un RB albero è **almeno** il doppio dell'altezza di un albero binario:

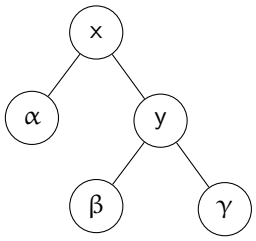
$$\begin{aligned} n &\geqslant 2^{\frac{h}{2}-1} \\ 2^{\frac{h}{2}} &\leqslant n+1 \\ \frac{h}{2} &\leqslant \log_2(n+1) \\ h &\leqslant 2\log_2(n+1) \end{aligned}$$

- **Inserimento**: L'inserimento di un nodo in un albero RB si fa allo stesso modo di un albero binario di ricerca. Per non violare la proprietà 4, si fa di colore rosso e si salva un puntatore al nuovo oggetto, questo perchè il nuovo albero è un RB albero, **tranne** per l'oggetto puntato dal puntatore perchè potrebbe avere un padre rosso. Si propaga l'anomalia verso la radice dell'albero per poter cambiare il colore della radice mantenendo tutte le proprietà. Per risolvere il problema si può fare una serie di **rotazioni** a destra o a sinistra.

  Prendiamo in considerazione il seguente albero:

  La rotazione a destra della coppia x e y è la seguente:

  Questa operazione mantiene tutte le proprietà dell'albero binario di ricerca. L'operazione opposta è la rotazione a sinistra. Il tempo di esecuzione di una rotazione è costante.

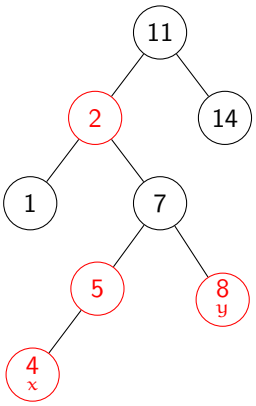
  Lo pseudocodice dell'algoritmo per risistemare l'albero è il seguente:

```
// Albero con radice sicuramente nera
// x: Nodo da cui si propaga l'anomalia

while x != root and color(parent(x)) == red
  // Se il padre e' figlio sinistro del nonno
  if parent(x) == left(parent(parent(x)))
```

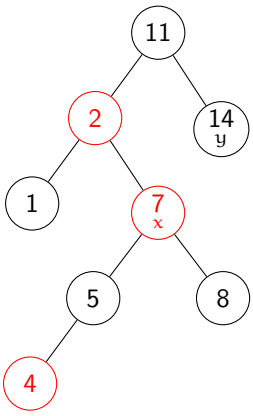
```
    y <- right(parent(parent(x))
    if color(y) == red
      color(parent(parent(x))) <- red
      color(parent(x)) <- black
      color(y) <- black
      x <- parent(parent(x))
    else
      if x == right(parent(x))
        x <- parent(x)
        left_rotate(x)

      color(parent(x)) <- black
      color(parent(parent(x))) <- red
      right_rotate(parent(parent(x)))
      x <- root

  // Se il padre e' figlio sinistro del nonno
  // Si fanno le stesse operazioni con le direzioni invertite
```

La complessità di questo algoritmo è $O(\log n)$ perchè è un inserimento in un albero binario di ricerca con delle rotazioni per sistemare l'anomalia, ma il numero di rotazioni è costante.

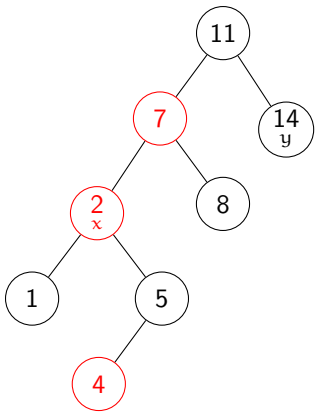
Esempio 5.3. Un esempio del predcedente algoritmo è il seguente:



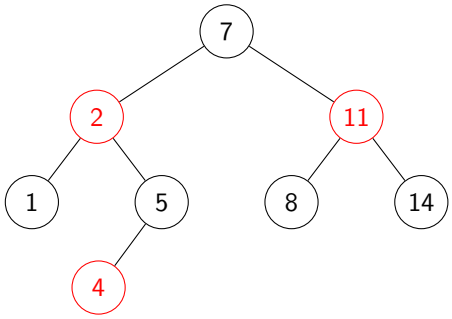
Si cambiano i colori dei nodi $parent(x)$ e y e si sposta l'anomalia verso l'alto



Prendiamo la coppia 2 e 7 e ruotiamo a sinistra



Si cambiano i colori e si effettua una rotazione a destra



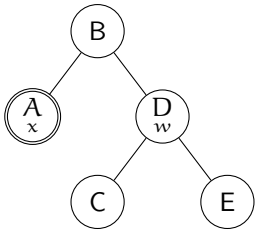
L'albero ora non ha più l'anomalia.

- **Rimozione**: La rimozione di un nodo da un albero RB si fa allo stesso modo di un albero binario di ricerca. Bisogna quindi distinguere i 3 casi:

1. **Il nodo da rimuovere è una foglia**: si può rimuovere direttamente e salviamo in un puntatore il nodo sentinella che sostituirà il nodo rimosso
2. **Il nodo da rimuovere ha un solo figlio**: si può sostituire il nodo con il figlio e si salva in un puntatore il figlio
3. **Il nodo da rimuovere ha due figli**: si può sostituire il nodo con il minimo del sottoalbero destro o con il massimo del sottoalbero sinistro

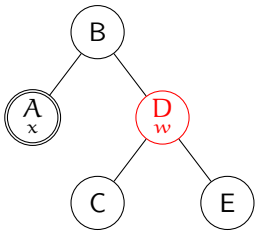
L'anomalia sarà un nodo rosso che dovrà essere nero per mantenere la proprietà 4, oppure un nodo nero che dovrà valere come due neri per mantenere la proprietà 4. Questa è un anomalia locale che si può risolvere propagandola verso la radice dell'albero lavorando su un sottoinsieme di nodi.

Visto che ci sono i nodi sentinella i primi due casi sono risolti dallo stesso codice. L'algoritmo è il seguente:

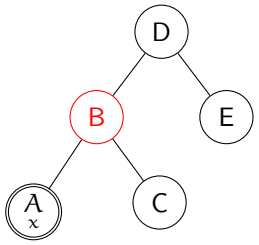


Consideriamo un nodo x non radice e **doppiamente nero** (anomalia), prendiamo il fratello w di x e distinguiamo i seguenti casi:

– **Se w è rosso**:



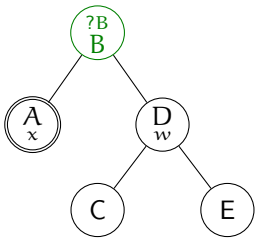
Per forza il padre di x deve essere nero (per mantenere la proprietà 3), di conseguenza lo sono anche i figli di w. Si scambia il colore tra il padre di x e w e si fa una rotazione a sinistra sul padre di x.



Questa trasformazione non viola nessuna proprietà dell'RB-albero, tranne per l'anomalia di x.

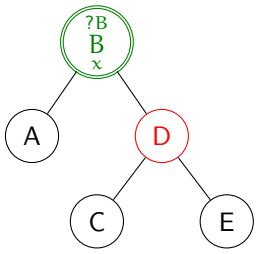
Non siamo più nel caso in cui il fratello di x è rosso.

– **Se w è nero**:



Se w è nero non si può dire nulla sul colore del padre di x. Anche adesso si distinguono due casi:

∗ **Se i figli di w sono neri**: Si toglie 1 nero dal livello di x e w e si aggiunge 1 nero al padre di x. Si propaga l'anomalia verso l'alto:

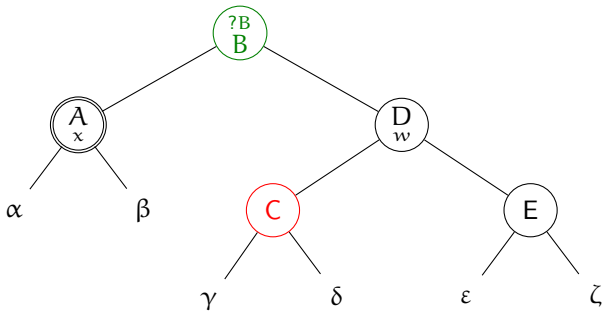


Se il nuovo x è rosso l'anomalia è risolta, altrimenti si applica la stessa procedura al nuovo x.

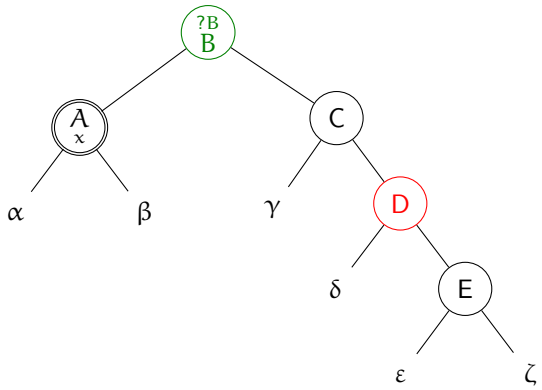
Questa trasformazione non viola nessuna proprietà dell'RB-albero; anche se x non ha un colore specifico (e se fosse rosso avrebbe un figlio rosso, di conseguenza violerebbe una proprietà dell'RB-albero), l'anomalia si è spostata sul nuovo x, di conseguenza se fosse rosso diventerebbe nero (anomalia risolta) e se fosse nero diventerebbe doppiamente nero.

∗ **Se i figli di w non sono entrambi neri**:

· **Se il figlio destro di w è nero**: Visto che entrambi i figli di w non sono neri e il destro è nero, allora il sinistro è per forza rosso. (le lettere greche sono i sottoalberi delle foglie che non vengono considerati)

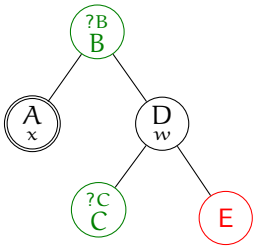


Si scambia il colore tra w e il figlio sinistro di w e si fa una rotazione a destra su w:

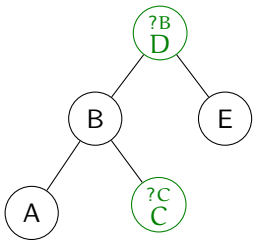


Questa trasformazione preserva l'RB-albero. Ora il figlio destro del fratello di x è rosso e si applica il caso successivo.

- **Se il figlio destro di w è rosso**: Visto che entrambi i figli di w non sono neri e il destro è rosso, allora il sinistro potrebbe essere sia rosso che nero.



Si assegna a w il colore del padre di x, al padre di x e al figlio destro di w il colore nero. Infine si fa una rotazione a sinistra sul padre di x e si porta l'anomalia alla radice.



Questa trasformazione preserva l'RB-albero.
L'anomalia punta alla radice dell'albero, in questo caso non punta a niente perchè non sappiamo se il nodo più in alto è la radice, quindi l'anomalia è risolta.

L'algoritmo è finito quando l'anomalia punta alla radice dell'albero. Lo pseudocodice dell'algoritmo è il seguente:

```
// x: Nodo da cui si propaga l'anomalia
while x != root and color(x) == black
  w <- right(parent(x))
  // Se x e' figlio sinistro del padre
  if x == left(parent(x))
    // Se w e' rosso (caso 1)
    if color(w) == red
      color(w) <- black
      color(parent(x)) <- red
      left_rotate(parent(x))
    else // Se w e' nero
      // Se i figli del w sono neri (caso 2, propaga x in alto)
      if color(left(w)) == black and color(right(w)) == black
        color(w) <- red
        x <- parent(x)
      else
        // Se il figlio destro di w e' nero (caso 3, termina)
        if color(right(w)) == black
          color(left(w)) <- black
          color(w) <- red
          right_rotate(w))

        // Se il figlio destro di w e' rosso (caso 4, termina)
        color(w) <- color(parent(x))
        color(parent(x)) <- black
        color(right(w)) <- black
        left_rotate(parent(x))
        x <- root
  // Se x e' figlio destro del padre
  // Si fanno le stesse operazioni con le direzioni invertite
```

La complessità di questo algoritmo è $\Theta(\log n)$ si fa un massimo di 3 rotazioni.

- **Selezione**: La selezione di un nodo in un albero RB si fa allo stesso modo di un albero binario di ricerca. Si parte dalla radice e si scende fino a trovare il nodo desiderato. La complessità di questo algoritmo è $O(\log n)$. Se riesco a trovare un modo per decidere se andare a sinistra o a destra in tempo costante allora è sufficiente a far diventare la complessità logaritmica.

  – **Dato l'indice da trovare**: Per sapere la posizione dell'elemento della

radice si può salvare la grandezza del sottoalbero radicato nel nodo x e usare la seguente formula:

$$size(left(x)) + 1$$

Usando questa formula si può decidere se andare a sinistra o a destra in tempo costante e lo pseudocodice dell'algoritmo è il seguente:

```
// x: Nodo da cui si parte la selezione
// i: Indice dell'elemento da selezionare
select(x,i)
  r <- size(left(x)) + 1
  if i == r
    return x
  else if i < r
    return select(left(x),i)
  else
    return select(right(x),i-r)

```

La complessità di questo algoritmo è $O(\log n)$ perchè la decisione di andare a sinistra o a destra è in tempo costante.

Questo crea il problema di dover aggiornare la grandezza del sottoalbero ad ogni inserimento e rimozione di un nodo.

* In un inserimento, il valore size del nodo aggiunto vlae 1, mentre il campo size di tutti i nodi del percorso seguito per inserire il nodo aumenta di 1.
* In un estrazione, il campo size di tutti i nodi del percorso seguito per rimuovere il nodo diminuisce di 1.
* Nella sistemazione dell'albero, i campi del colore non hanno alcuna influenza sul campo size, mentre se si fa una rotazione il valore di size deve cambiare:

  ```
  size(x) <- size(left(x)) + size(right(x)) + 1

  ```

  Questo funziona perchè il campo size è la somma dei campi size dei figli più 1, cioè viene ricalcolato da capo utilizzando il valore salvato nei figli.
  In ogni caso dopo una rotazione gli unici nodi che cambiano il campo size sono i nodi alla radice che si scambiano il valore.

Abbiamo dimostrato che si può mantenere il campo size mantenendo la complessità logaritmica.

– **Dato un nodo da trovare**: Lo pseudocodice dell'algoritmo è il seguente:

```
// x: Nodo da cui si parte la selezione
// y: Nodo da trovare
rank(x)
  count <- size(left(x)) + 1
  while x != root
    // Se x e' figlio destro del padre
    if x == right(parent(x))
      // Si aggiunge la grandezza del sottoalbero sinistro
     del padre
```

```
      count <- count + size(left(parent(x)) + 1

    x <- parent(x)

  return count
```

La complessità di questo algoritmo è $O(\log n)$ perchè la decisione di andare a sinistra o a destra è in tempo costante.

Se si volessero inserire tutti gli elementi di un RB-albero in un array ordinato, si potrebbe usare un algoritmo di visita in-order dell'albero. Lo pseudocodice dell'algoritmo è il seguente:

```
// x: Nodo da cui si parte la visita
// A: Array in cui inserire x
// i: Indice dell'array in cui inserire x (passato per riferimento)
in_visit(x,A,i)
  if x != nil
    in_visit(left(x),A,i)
    visit(x,A,i) // Inserisce x nell'array
    in_visit(right(x),A,i)


// x: Nodo da inserire nell'array
// A: Array in cui inserire x
// i: Indice dell'array in cui inserire x (passato per riferimento)
visit(x,A,i)
  A[i] <- value(x)
  i++
```

Questo algoritmo ha complessità $\Theta(n)$ perchè visita tutti i nodi dell'albero.

$$T(n) = T(n_{left}) + 1 + T(n_{right}) = n_{left} + 1 + n_{right} = n$$

Ci sono anche le seguenti alternative:

```
pre_visit(x,A,i)
  if x != nil
    visit(x,A,i) // Inserisce x nell'array
    pre_visit(left(x),A,i)
    pre_visit(right(x),A,i)

post_visit(x,A,i)
  if x != nil
    post_visit(left(x),A,i)
    post_visit(right(x),A,i)
    visit(x,A,i) // Inserisce x nell'array
```

Se si volesse creare un RB-albero partendo da un array (non necessariamente ordinato) si potrebbero fare n inserimenti in un RB-albero vuoto. La complessità di questo algoritmo è $\Theta(n \log n)$.

Oppure si potrebbe prima ordinare l'array e poi usare l'algoritmo di visita in-order per inserire tutti gli elementi in un RB-albero, per fare ciò però bisogna creare un "impalcatura", cioè un RB-albero con tutti i nodi nil, e poi inserire i nodi ordinati. Si fanno tutte le foglie rosse e tutti gli altri nodi neri. In pseudocodice l'algoritmo è il seguente:

```
// Crea un albero bilanciato con n nodi nil
build_empty_tree(n)
  if n == 0
    return nil
  else
    x <- new_node(nil)
    left(x) <- build_empty_tree(floor((n-1)/2))
    right(x) <- build_empty_tree(ceil((n-1)/2))
    return x
```

Questo algoritmo non si può fare con complessità minore di $n \log n$.

Dimostrazione:
Supponiamo che `array_to_rb` abbia complessità $f < n \log n$ e consideriamo:

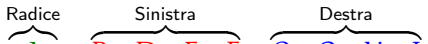
```
sort(A)
  T <- array_to_rb(A)
  A <- rb_to_array(T)
```

La complessità di questo algoritmo sarebbe:

$$f + n < n \log n$$

e ciò vorrebbe dire che è stato creato un algoritmo di ordinamento più veloce di $n \log n$, cosa impossibile, quindi l'algoritmo `array_to_rb` non può avere complessità $f < n \log n$.

Se visitiamo un albero con gli algoritmi in-visit e pre-visit ottieniamo una stringa di nodi, è possibile ricostruire l'albero originale partendo dalla radice?

- **Pre**: 

- **In**: 

Dalle stringhe si possono individuare quali sono i sottoalberi sinistro e destro e qual'è la radice. L'algoritmo per ricostruire l'albero è il seguente:

```
// P: Stringa di visita pre-ordine
// I: Stringa di visita in-ordine
// Supponiamo che le stringhe abbiano la stessa lunghezza
str_to_tree(P,I)
  if length(I) == 0
    return nil
  else
    x <- new_node(P[1])
    i <- find(I,P[1])
    left(x) <- str_to_tree(P[2..i],I[1..i-1])
    right(x) <- str_to_tree(P[i+1..length(P)],I[i+1..length(I)])
    return x
```

L'albero sarà quindi:

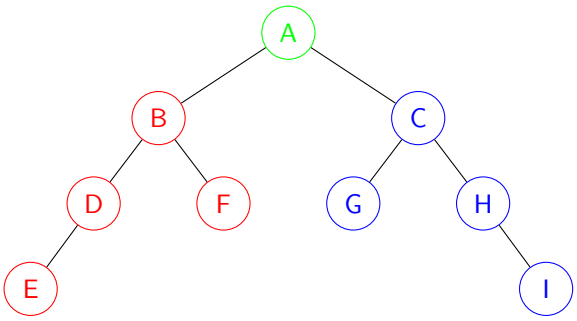


Figura 9: Albero ricostruito

È possibile implementare un algoritmo che unisce 2 RB-alberi in un unico RB-albero in tempo $\log n$?

```
A1 <- rb_to_array(T1)
A2 <- rb_to_array(T2)
A <- merge(A1,A2)
T <- array_to_rb(A)
```

Questo ha complessità $O(n)$ perchè è l'unione di più algoritmi con complessità lineare.

Questo problema è simile al concetto del merge sort, cioè prendere 2 metà, risolverle ed unire il risultato.

```
array_to_rb(A,p,q)
  if p > q
    return nil
  else
    r <- (p+q)/2
    T1 <- array_to_rb(A,p,r)
    T2 <- array_to_rb(A,r+1,q)
    return union(T1,T2)
```

Questo algoritmo ha complessità:

$$T(n) = 2T(n/2) + C(\texttt{union})$$

Supponiamo che $C(\texttt{union}) = \log n$, quindi $T(n) = \Theta(n)$, ma sappiamo che in tempo minore di $n \log n$ non si può costruire un RB-albero, di conseguenza non è possibile che la union abbia complessità $\log n$.

5.5 Campi aggiuntivi

Si possono aggiungere dei campi aggiuntivi alle strutture dati per velocizzare alcuni algoritmi, ma bisogna stare attenti a mantenere la complessità delle operazioni inalterata.

Teorema 5.1. Sia F un campo aggiuntivo, se

$\exists f.\forall x F(x) = f(key(x), F(left(x)), F(right(x)), key(left(x)), key(right(x)))$

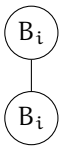
allora F è mantenibile in $\log n$.

Cioè se il campo aggiuntivo è calcolabile utilizzando le operazioni definite in f allora la complessità di F è mantenibile.

5.6 Albero Binomiale

Gli alberi binomiali sono definiti ricorsivamente rispetto ad un valore chiamato **valore binomiale**. Un albero di dimensione 0 sarà da un solo nodo:

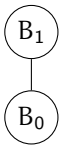
Per costruire un albero binomiale di dimensione $i + 1$ si fa diventare il secondo albero un sottoalbero del primo albero:



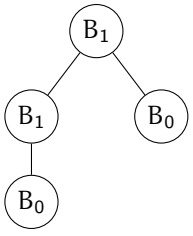
Quindi un albero di dimensione:

- B_0:

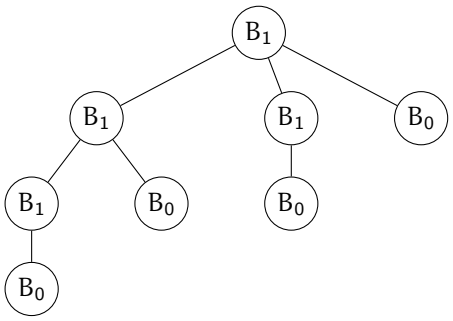
- B_1:



- B_2:



- B_3:



Teorema 5.2.

- Il numero di nodi è 2^k
  - Dimostrazione:

    Faccio vedere che la proprietà vale per 0 (caso base). Il numero totale di nodi in un albero binomiale di dimensione 0 è 1, cioè 2^0.
    Supponiamo per induzione che in B_k ci siano 2^k nodi. Per dimostrare che in B_{k+1} ci sono 2^{k+1} nodi, utilizziamo la costruzione dell'albero binomiale B_k e aggiungiamo un altro albero binomiale B_k come figlio sinistro della radice di B_k. Quindi il numero di nodi in B_{k+1} è $2^k + 2^k = 2^{k+1}$.
- L'altezza dell'albero è k
  - Dimostrazione:

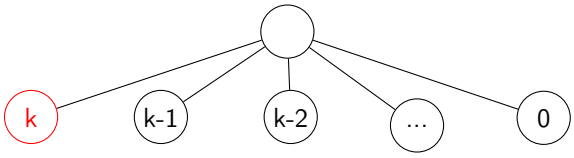
    L'altezza di un albero binomiale di dimensione 0 è 0.
    Supponiamo per induzione che la dimensione di un albero binomiale B_k sia k. Per dimostrare che la dimensione di un albero binomiale B_{k+1} sia $k+1$, sappiamo che l'altezza di un albero è il cammino più lungo radice-foglia, quindi il cammino più lungo dalla radice al figlio sinistro è k più un arco, quindi l'altezza è $k+1$.
- A profondità i ci sono $\binom{k}{i}$ nodi: $(1+i)^k$
  - Dimostrazione:

    Se consideriamo il triangolo di targaglia, abbiamo che l'elemento alla riga k e colonna i è $\binom{k}{i}$. Quindi il numero di nodi alla profondità i è:
    $$\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$$
- I figli della radice da sinistra a destra sono radici di:

$$B_{k-1}, B_{k-2}, \ldots, B_0$$

– Dimostrazione:

Un albero di dimensione 0 non ha figli, quindi la radice di B_k ha come figli le radici di $B_{k-1}, B_{k-2}, \ldots, B_0$. Per $k+1$ basta aggiungere un sottoalbero k come figlio sinistro della radice



5.7 Heap Binomiale

Un heap binomiale è una lista di alberi binomiali, dove:

- I contenuti dei nodi sono oggetti su cui è definita una relazione di ordinamento
- Per ogni dimensione c'è al più un albero binomiale
- I vari nodi soddisfano la proprietà di uno heap, cioè:

```
key(x) <= keys(children(x))
```

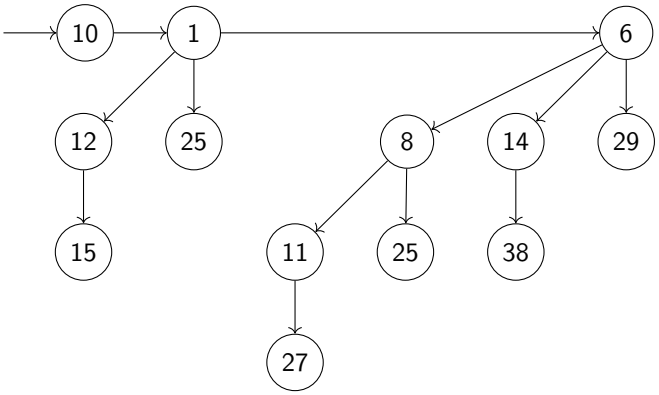


Figura 10: Esempio di heap binomiale

La complessità per cercare il minimo è data dal numero di radici.
In un heap di n nodi il numero di alberi binomiali è $\log n$.

$$2^k > 2^{\log_2 n} = n$$

Di conseguenza la complessità per trovare il minimo è $O(\log n)$.

Per questa struttura non esiste un algoritmo di visita in ordine, perchè se ci fosse vorrebbe dire che esisterebbe un algoritmo di ordinamento in tempo logaritmico e sappiamo che non esiste.

5.7.1 Creazione di un heap binomiale

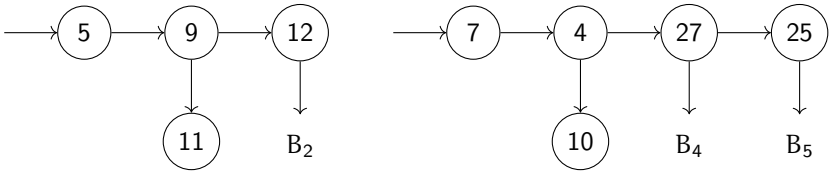
Per costruire un heap binomiale di dimensione n si può trasformare n in binario e considerare la notazione polinomiale:

$$n = \overset{2^5 2^4 2^3 2^2 2^1 2^0}{1\,0\,0\,1\,0\,1}$$

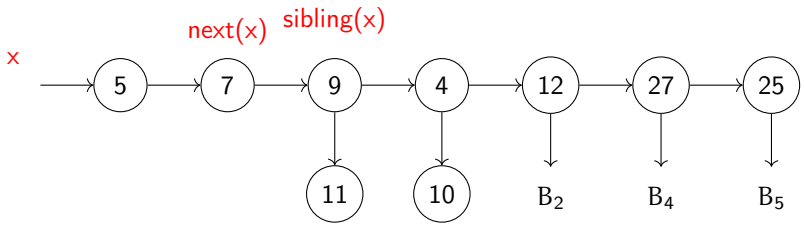
Si può esprimere un qualsiasi numero n come somma di potenze di 2. Se la cifra è 1 allora si crea un albero binomiale di quella dimensione, altrimenti si passa alla cifra successiva.

5.7.2 Unione di due heap binomiali

Si vogliono unire i seguenti heap binomiali:



Si scrive un unica lista che è il merge delle due liste:

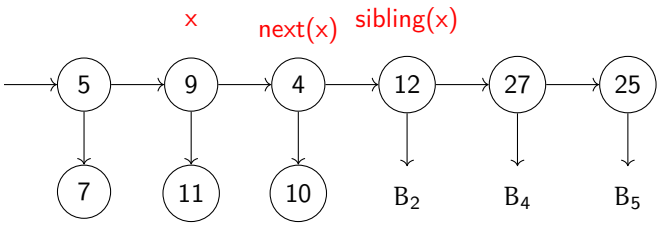


Le due liste rappresentate in binario sono:

$$L1 = 111 \quad L2 = 110011$$

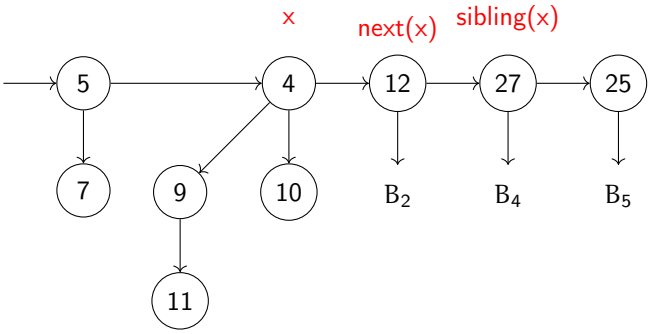
L'unione delle due liste si fa sommando i numeri binari: La nuova lista diventa:

1. Passo 1:



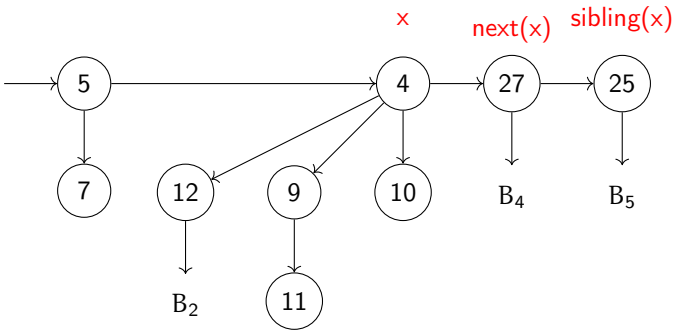
$$\begin{array}{cccccc} & & & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ \hline & & & & & 0 \end{array}$$

2. Passo 2:



$$\begin{array}{cccccc} & & & 1 & 1^1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ \hline & & & & 1 & 0 \end{array}$$

3. Passo 3:



$$\begin{array}{cccccc} & & 1 & 1^1 & 1^1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

L'unione di due heap binomiali ha complessità $O(\log n)$.

5.7.3 Inserimento

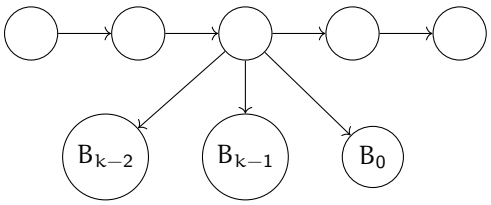
L'inserimento si può vedere come l'unione tra un heap binomiale e un heap binomiale con un solo nodo.

$$\begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Anche l'inserimento ha complessità $O(\log n)$.

5.7.4 Rimozione

- Se si vuole rimuovere il minimo elemento di un heap binomiale bisogna rimuovere una radice. Sappiamo che i figli di una radice sono tutti heap binomiali di dimensione minore.



La complessità per rimuovere il minimo elemento è $O(\log n)$.

Per rimuovere un nodo arbitrario si usa la `reduce` diminuendo il valore della chiave del nodo e facendolo salire fino a che non arriva alla radice eliminando il nodo.

5.7.5 Riduzione

La riduzione prende un nodo, lo abbassa (diminuendo il valore della chiave) e gli assegna la chiave a, mantenendo le proprietà di un heap.

La proprietà dello heap viene mantenuta rispetto ai figli, ma non rispetto al padre. Se la chiave del padre è maggiore di quella del figlio, si scambiano le chiavi e si fa salire il nodo fino a che non si rispettano le proprietà dello heap.

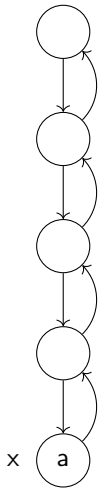


Figura 11: Riduzione della chiave di un nodo

La complessità della riduzione è $O(\log n)$.

5.7.6 Divisione

Per dividere un heap binomiale mantenendo la differenza di nodi tra le due parti al più uno bisogna mantenere da parte il nodo di dimensione 0 e dividere il resto. Il nodo da parte verrà poi riaggiunto ad una delle due lista utilizzando la union.

5.8 Struttura dati facilmente partizionabile

Farebbe comodo avere una struttura dati che permetta di partizionare una collezione di oggetti in complessità costante. Si vogliono implementare i seguenti algoritmi:

- `make_set(x)` crea un insieme di un solo elemento
- `union(x,y)` unisce due insiemi a cui appertengono due oggetti
- `find_set(x)` restituisce il **rappresentante** dell'insieme a cui appartiene x.

Sappiamo soltanto quello che vogliamo fare, non sappiamo come implementarlo.

5.8.1 Complessità

Un modo per rappresentare un insieme è utilizzare una lista concatenata.

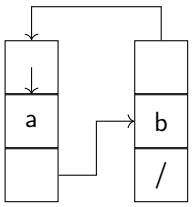


Figura 12: Esempio di una lista concatenata

Si potrebbe dire che il rappresentante dell'insieme è il primo elemento della lista. Con queste informazioni possiamo dire che le complessità degli algoritmi saranno:

- `make_set(x)`: $O(1)$ perchè basta creare un nodo che punta a se stesso perchè è il rappresentante e non ha altri elementi
- `find_set(x)`: $O(1)$ perchè basta restituire il valore del puntatore che punta al rappresentante
- `union(x,y)`: $O(n)$ perchè bisogna aggiornare tutti i rappresentanti degli insiemi. Se viene aggiunto un puntatore all'elemento in fondo alla lista, la complessità diventa $O(1)$.

Se vengono effettuate m operazioni di cui n sono `make_set` (cioè il numero di oggetti, di conseguenza il numero di union possibili), la complessità nel caso pessimo è $O(m * n)$. Di tutte queste operazioni, le `union` saranno al massimo $n - 1$ perchè non si possono unire più di n insiemi. Tutte le rimanenti saranno operazioni di tempo costante, di conseguenza un altro limite superiore corretto sarebbe $O(m + n^2)$. Un

limite superiore migliore si può ottenere calcolando la complessità media di tutte le operazioni e quindi la complessità finale diventa: $O(\frac{m+n^2}{m}) = O(1 + \frac{n^2}{m})$.

Per capire se questa complessità è accurata bisogna trovare un insieme di operazioni da fare che portino ad avere quella complessità.

1. Si fanno n `make_set` creando n insiemi
2. Si uniscono le coppie di insiemi. Per la prima coppia il costo è 1, per la seconda 2 e così via. Il costo totale è:

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

Per fare meglio di $\Theta(n^2)$ bisogna ottimizzare la union, cioè l'aggiornamento di tutti i puntatori. Si potrebbe unire l'insieme più piccolo a quello più grande, al posto di unire quello più grande a quello più piccolo e a questo punto bisogna cambiare soltanto un puntatore, nel caso in cui l'elemento più piccolo abbia un elemento. Questo metodo è chiamato **unione per rango**. Il numero massimo di volte che si può modificare il puntatore al rappresentante dopo questa tecnica diventa $\log_2 n$ perchè:

1. Se ad un insieme di un elemento viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà minimo 2 elementi:

$$\geqslant 2$$

2. Se ad un insieme di due elementi viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà al minimo 4 elementi (perchè sappiamo che bisogna unire l'insieme più piccolo ad un altro insieme, di conseguenza se sappiamo che l'insieme di due elementi è il più piccolo, l'altro insieme avrà al massimo 2 elementi):

$$\geqslant 4 = 2^2$$

3. ...
4. Se ad un insieme di i elementi viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà al minimo 2^i elementi:

$$\geqslant 2^i$$

Questa costruzione si chiama **iterative squaring**. La complessità con l'unione per rango diventa:

$$\frac{m + n \log n}{m} = 1 + \frac{n \log n}{m} \leqslant 1 + \log n$$

Per fare meglio si può rappresentare in un modo alternativo gli insiemi, quindi con degli alberi.

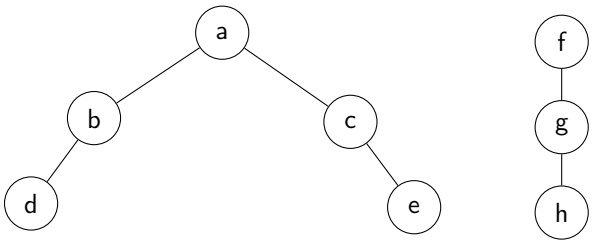


Figura 13: Esempio di alberi come insiemi

- `make_set(x)`: si crea un albero con un solo nodo

$$O(1)$$

- `find_set(x)`: si risale l'albero fino alla radice

$$O(n)$$

- `union(x,y)`: si fa diventare la radice di un albero il figlio dell'altra

$$O(n)$$

Come prima si può usare l'unione per rango per migliorare la complessità. Ogni volta che un nodo aumenta di profondità (grazie all'unione ad un altro albero), il numero di nodi raddoppia. Facendo l'unione per rango sappiamo che la complessità della ricerca del rappresentante è $O(\log n)$ e la complessità dell'unione è $O(\log n)$.

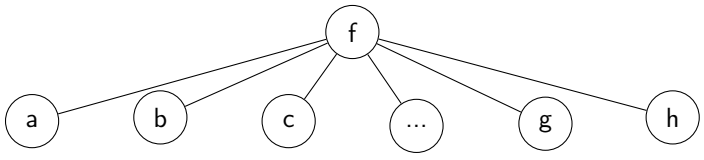
Siccome la `find_set` è $O(\log n)$ siamo comunque peggio di prima, quindi si può far puntare ogni nodo direttamente al rappresentante, ma questo viene fatto soltanto quando viene richiamata la `find_set`, im modo da non farlo più alle chiamate successive. L'algoritmo è il seguente:

```
find_set(x):
  if parent(x) == x
    return x
  else
    parent(x) = find_set(parent(x))
    return parent(x)
```

oppure più compatto:

```
find_set(x):
  if parent(x) != x
    x <- find_set(parent(x))
  return parent(x)
```

Man mano che vengono eseguite le `find_set` la struttura si comprime (**tecnica di compressione dei cammini**):



Di conseguenza la complessità della `find_set` diventa costante.

La nuova complessità sapendo di avere m operazioni di cui n `make_set` diventa:

$$O(m\alpha(m,n))$$

dove α è l'inversa della funzione di Ackermann:

$$\alpha(m,n) = m, n\left\{i \geqslant 1, \left|A\left(i, \left\lfloor \frac{m}{n} \right\rfloor > \log n\right)\right.\right\}$$

$$A(i,1) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad \text{(i volte)}$$

Siccome la funzione di Ackermann cresce molto velocemente, la sua inversa cresce molto lentamente. Di conseguenza non si riuscirà mai a superare una certa costante, quindi la complessità diventa costante.

Esercizio 5.1. Si vuole gestire un'agenda che contiene appuntamenti con un orario di inizio e un orario di fine, si vuole:

- Inserire un appuntamento (un intervallo di tempo $[min_t, max_t]$)
- Rimuovere un appuntamento
- Dato un intervallo di tempo trovare tutti gli appuntamenti in conflitto con questo intervallo di tempo

Questo può essere rappresentato come un RB-albero in cui i nodi rappresentano gli appuntamenti.

Il tempo di fine è un dato importante per capire se è presente un'intersezione, se ordinassimo per il tempo di inizio, ciò che ci dice se c'è stata un intersezione con la radice è se il tempo di fine dei nodi a sinistra è maggiore del tempo di inizio della radice.

Si può aggiungere un campo aggiuntivo che memorizza il massimo tempo di fine del sottoalbero di un nodo e questo si può mantenere in tempo logaritmico perchè dipende solo dal nodo stesso e dai propri figli

L'algoritmo che cerca l'eventuale intersezione è il seguente

```
// x e' il nodo radice
// I e' l'intervallo di tempo
search(x,I)
  if x == nil && I.intersects(x.interval)
    return x
  if left(x) != nil && left(x).max_end > I.start
    return search(left(x),I)
  else
    return search(right(x),I)
```

Quando sappiamo che non ci sono intervalli che intersecano a sinistra sappiamo anche che non ce ne sono a destra perchè se l'intervallo da controllare fosse all'interno dell'intervallo alla radice, allora sarebbe stato a sinistra. Ciò vuol dire che questo intervallo si trova almeno alla fine dell'intervallo alla radice e

di conseguenza tutti gli altri intervalli si troveranno a destra della fine della radice e quidi non ci sono intersezioni.

5.9 Tabelle Hash

Le tabelle hash sono strutture dati che permettono di memorizzare elementi associandoli ad un valore univoco. Si vuole accedere ad un elemento inserendo una chiave per ottenere il valore associato. Questa tabella si può creare idealmente utilizzando un array indicizzato da caratteri:

```
1  T["element"]
```

Se si utilizza un array indicizzato da una stringa lunga massimo 256, si dovranno allocare 26^{256} celle di memoria. Si può allora creare una struttura che abbia la stessa funzione, ma allocando meno memoria? L'idea è quella di creare uno spazio più piccolo associato all'array dei dati e poi creare una funzione chiamata **hash** che associa lo spazio più grande, cioè quello delle stringhe, a quello più piccolo.

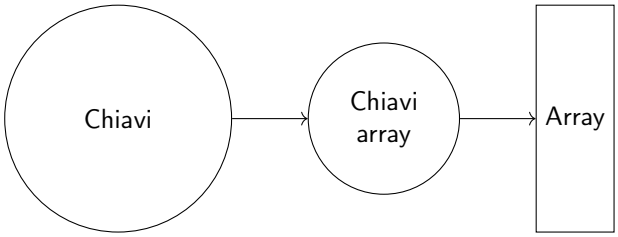


Figura 14: Struttura di una tabella hash

Questa funzione deve essere tale che due stringhe diverse abbiano un hash diverso e che due stringhe uguali abbiano lo stesso hash. Per ottenere l'elemento associato ad una stringa si avrà:

$$\mathrm{A}(\text{stringa}) \Rightarrow \mathrm{A}'\left(\mathrm{H}\left(\text{stringa}\right)\right)$$

Mappare uno spazio grande in uno più piccolo crea conflitti, cioè due stringhe diverse possono essere associate allo stesso valore e quindi bisogna trovare un meccanismo per risolvere i conflitti.

5.9.1 Risoluzione dei conflitti

- L'array A' è una lista concatenata di coppie chiave valore.

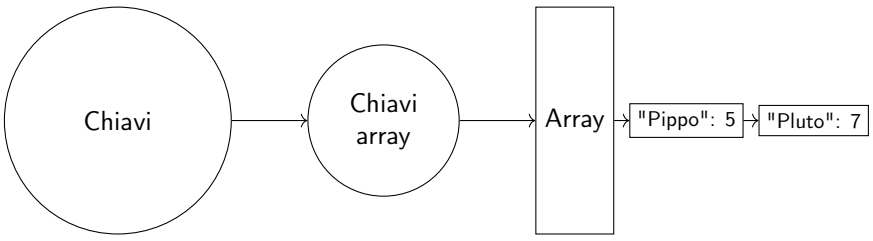


Figura 15: Risoluzione dei conflitti

La complessità di questa struttura dati è $O(n)$ perchè bisogna scorrere tutta la lista per trovare l'elemento ricercato. Se però si riesce a trovare una funzione hash che riesce a distribuire in maniera uniforme gli input nella lista di dati, allora il **fattore di carico**, cioè il rapporto tra il numero di elementi e il numero di celle allocate, sarà:

$$\frac{n}{m}$$

dove n è il numero di elementi e m è il numero di celle allocate.

- Un altro approccio è quello di mantenere tutti i dati nell'array A' e non più in una lista concatenata. Se in una cella è presente un elemento e si aggiunge un elemento diverso che viene mappato nella stessa cella si può rilevare che la cella è piena e quindi si inserisce l'elemento in una cella successiva. Per la ricerca si guarda nella cella mappata e se non si trova l'elemento si guarda nella cella successiva e così via fino a trovare l'elemento o una cella vuota. Questo metodo funziona finchè l'array non è pieno, ma questo non viene mai raggiunto perchè il fattore di carico è $\frac{1}{2}$.

  Per implementare questo concetto si usa la seguente funzione di hash che prende in input la chiave e il numero di tentativi fatti per trovare la cella giusta:

  $$H(k, i) = (h(k) + i) \bmod m$$

  dove $h(k)$ è una funzione di hash qualsiasi.

  Con questa funzione si verifica il problema di **aggregazione primaria**, cioè una sezione dell'array viene riempita prima delle altre. Per risolvere questo problema si può scalare l'offset i con una costante l per fare salti più grandi e lasciare quindi più buchi vuoti. Bisogna però scegliere l in maniera tale da non dover arrivare al punto di partenza senza prima aver passato tutti gli elementi, cioè l e m devono essere **relativamente primi tra di loro**, cioè non devono avere divisori comuni. Nonostante ci siano più buchi, la lista della ricerca rimane sempre lunga quanto gli elementi in conflitto, quindi per ridurre la lista di ricerca si può usare la tecnica dell'**hashing doppio**:

  $$H(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

  In questo modo l'offset per fare le ricerche non è più costante, ma dipende dalla chiave. In questa funzione di hash m e $h_2(k)$ sono sempre relativamente primi tra di loro.

  Per eliminare un elemento bisogna scorrere tutta la lista per trovare l'elemento e dopo averlo eliminato bisogna far scorrere tutti gli elementi successivi per riempire il buco lasciato.

Se il numero di celle allocate è il doppio del numero di elementi, allora il fattore di carico sarà $\frac{1}{2}$ e quindi la complessità sarà $O(1)$ perchè si avrà metà elemento per ogni cella.

Data la funzione di hash si possono trovare subito dei conflitti, quindi per evitare ciò bisogna introdurre un metodo di crittografia.

5.9.2 Funzioni di hash

Alcune funzioni di hash sono:

1. $H(k) = k \bmod m = k\%m$. Questa funzione è molto semplice, ma non è molto buona. Ad esempio se $m = 2^l$ tutte le chiavi che hanno gli ultimi l bit uguali avranno lo stesso hash. Gli m che funzionano bene sono i numeri primi.

2. $H(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$ dove $A \in \mathbb{R}$. Questo rappresenta un numero compreso tra 0 e 1 moltiplicato per m, quindi un numero tra 0 e $m-1$. Il valore più critico in questa funzione è il valore di A. Il valore ottimale di A è:

$$A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887 = \phi - 1 = \text{Golden ratio} - 1$$

6 Tecniche di programmazione

Fin'ora abbiamo utilizzato la tecnica del **divide et impera**, cioè dividere il problema in parti più piccole e risolverle con lo stesso algoritmo per poi unirle per ottenere il risultato. Questa non è l'unica tecnica di programmazione esistente, ce ne sono molte altre, ad esempio la **programmazione greedy**, cioè prendere le decisioni il prima possibile e sperare che siano le migliori, oppure la **programmazione dinamica**, cioè creare un'infrastruttura prima di poter prendere una decisione.

6.1 Programmazione dinamica

Prendiamo ad esempio il problema della **moltiplicazione di matrici**. Sappiamo che se abbiamo due matrici A di dimensione $n \times m$ e B di dimensione $m \times l$, il prodotto tra le due matrici avrà complessità $\Theta(nml)$.

Se volessimo moltiplicare 3 matrici: $A_1 \cdot A_2 \cdot A_3$ di dimensione:

$$A_1 : 10 \times 100$$
$$A_2 : 100 \times 5$$
$$A_3 : 5 \times 50$$

possiamo sfruttare la proprietà associativa:

$$(A_1 \cdot A_2) \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$$

Solo che il numero di operazioni nei due casi è diverso, quindi per rendere minimo il numero di operazioni è più conveniente fare: $(A_1 \cdot A_2) \cdot A_3$ perchè:

$$(A_1 \cdot A_2) \cdot A_3 \quad 10 \times 100 \times 5 + 10 \times 5 \times 50 = 5000 + 2500 = 7500$$
$$A_1 \cdot (A_2 \cdot A_3) \quad 100 \times 5 \times 50 + 10 \times 100 \times 50 = 25000 + 50000 = 75000$$

Supponiamo di avere un insieme di matrici $A_1, A_2, \ldots, A_n$, vogliamo trovare la **parentesizzazione** ottimale per minimizzare il numero di operazioni.

Un modo per affrontare il problema è quello di provare tutte le combinazioni di parentesi e vedere quale è la migliore.

C'è per forza una moltiplicazione che dovrà essere eseguita per ultima, quindi distinguiamo k come il punto in cui verrà fatta l'ultima moltiplicazione.

$$(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)$$

Il numero modi per moltiplicare le matrici è dato dal numero di modi per moltiplicare le matrici $A_1 \dots A_k$ e le matrici $A_{k+1} \dots A_n$.

$$P(n) = \sum_{k=1}^{n-1} P(k) + P(n-k) \quad \in \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

La complessità è troppo alta, quindi provare tutte le combinazioni possibili non è efficace.

Supponiamo che qualcuno ci dica il valore ottimo di k, allora è sicuro che il modo in cui vengono moltiplicate le matrici $(A_1 \cdots A_k)$ e $(A_{k+1} \cdots A_n)$ è ottimale:

$$\underbrace{(A_1 \cdots A_k)}_{\text{Ottimo}} \cdot \underbrace{(A_{k+1} \cdots A_n)}_{\text{Ottimo}}$$

Questa tecnica si chiama **Sottostruttura ottimale**, un problema si può dividere in problemi della stessa natura, ma più piccoli.

Il numero di moltiplicazioni effettuate è dato da:

$$N(A_1 \cdots A_n) =$$

$$= N(A_1 \cdots A_k) + N(A_{k+1} \cdots A_n) + \text{Costo ultima moltiplcazione}$$
$$= N(A_1 \cdots A_k) + N(A_{k+1} \cdots A_n) + rows(A_1) \cdot cols(A_k) \cdot cols(A_n)$$

Se k è ottimo, allora $N(A_1 \cdots A_k)$ e $N(A_{k+1} \cdots A_n)$ sono ottimi perchè il risultato viene sommato e se non fossero ottimi il risultato finale non sarebbe ottimo. Il problema è che non sappiamo quale sia il valore di k ottimo.

Un possibile algoritmo per trovare k è il seguente:

```
// P e' un vettore che descrive le matrici come:
//   - P[0] = rows(A1)
//   - P[i] = cols(Ai)
// Quindi Ai ha dimensione P[i-1] x P[i]
// i e' l'indice di partenza
// j e' l'indice di fine
matrix_chain_order(P,i,j)
  if i == j
    return 0

  m <- +inf
  for k <- i to j-1
    m <- min(m,
      matrix_chain_order(P,i,k) +
      matrix_chain_order(P,k+1,j) +
      P[i-1]*P[k]*P[j])
  return m
```

Questo algoritmo ha una complessità esponenziale.

Cerchiamo di migliorare l'algoritmo, implementando l'**albero di ricorrenza**, cioè un alberi i cui nodi rappresentano le chiamate ricorsive dell'algoritmo, identificate dai parametri i e j.

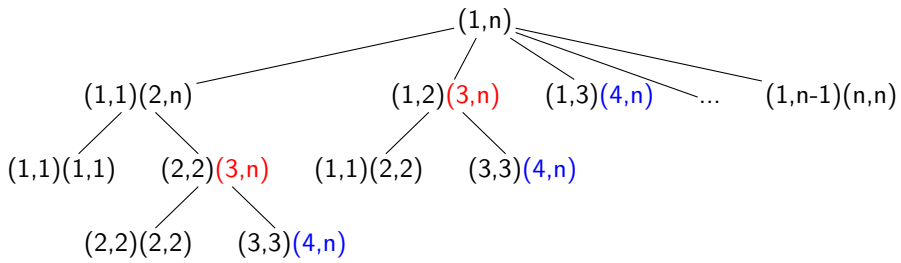


Figura 16: Albero di ricorrenza

Notiamo che ci sono molte chiamate ricorsive ripetute, ad esempio $(4, n)$, e questo aumenta la complessità dell'algoritmo inserendo istanze del problema che abbiamo già affrontato. Al massimo ci saranno n^2 istanze di un problema.

Anzichè risolvere il problema ogni volta che lo incontriamo, possiamo risolvere ogni singolo problema una sola volta e salvare il risultato:

```
matrix_chain_order(P)
  n <- length(p) - 1
  for i <- 1 to n // n
    // La moltiplicazione di una sola matrice costa 0 perche' non c
    'e' nulla da moltiplicare
    M[i,i] <- 0 // Matrice che salva tutte le soluzioni (i,j)

  for l <- 2 to n //           n --* Problemi composti da n matrici
    for i <- 1 to n - l + 1 // n --*
      j <- i + l - 1 //            | n^3
      M[i,j] <- +inf //            |
      for k <- i to j - 1 //   n --*
        M[i,j] <- min(M[i,j],
          M[i,k] + M[k+1,j] + P[i-1]*P[k]*P[j])

  return m[1,n]
```

Questo ci dice quante moltiplicazioni sono necessarie per moltiplicare tutte le matrici, però per sapere la posizione dell'ultima moltiplicazione da fare invece si può riscrivere l'algoritmo precedente come:

```
matrix_chain_order(P)
  ...
      for k <- i to j - 1
        m <- M[i,k] + M[k+1,j] + P[i-1]*P[k]*P[j]
        if m < M[i,j]
          M[i,j] <- m // Numero moltiplicazioni
          S[i,j] <- k // Indice dell'ultima moltiplicazione
  ...
```

La complessità di questo algoritmo è $O(n^3)$.

L'algoritmo precedente scritto in modo iterativo si potrebbe anche scrivere ricorsivamente e l'idea è che prima di calcolare il risultato si controlla se è già stato calcolato e si utilizza quello, altrimenti si calcola e si salva il risultato.

```
matrix_chain_order(P)
  for i <- 1 to n
    for j <- 1 to n
      M[i,j] <- +inf

  matrix_chain_order_aux(P,1,n)


matrix_chain_order(P,i,j)
  if M[i,j] != +inf
    return M[i,j]
  else
    if i == j
      M[i,j] <- 0
    else
      m <- +inf
      for k <- i to j-1
        m <- min(m,
          matrix_chain_order(P,i,k) +
          matrix_chain_order(P,k+1,j) +
          P[i-1]*P[k]*P[j])
      M[i,j] <- m

    return M[i,j]
```

Per ogni istanza del problema si calcola una sola volta il risultato, quindi la complessità è $O(n^3)$.

Definizione 6.1. La tecnica di memorizzare i risultati già calcolati si chiama **memoizzazione**.

L'algoritmo che data la matrice S, che contiene gli indici delle parentesizzazioni ottimal e la matrice A, che contiene le matrici da moltiplicare, restituisce la moltiplicazione ottimale è il seguente:

```
optimal_multiply(S, A, i, j)
  if i == j
    return A[i]

  return optimal_multiply(S, A, i, S[i,j]) *
         optimal_multiply(S, A, S[i,j] + 1, j)
```

6.2 Programmazione greedy

La programmazione greedy è una tecnica che prende decisioni il prima possibile e spera che siano le migliori.

Esempio 6.1. Prendiamo in considerazione un insieme di attività che sono caratterizzate dal tempo di inizio e dal tempo di fine (con tempo discreto). In ogni istante di tempo c'è solo un'attività che può essere svolta, quindi i periodi di attività devono essere disgiunti. Una lista di attività è la seguente:

```
// Questa lista puo' essere rappresentata come un array
```

```
1:  [8..12]
2:  [3..5]
3:  [8..11]
4:  [3..8]
5:  [0..6]
6:  [1..4]
7:  [5..9]
8:  [2..13]
9:  [6..10]
10: [5..7]
11: [12..14]
```

Ad esempio l'attività 5 e l'attività 10 non possono essere svolte contemporaneamente perchè si sovrappongono. L'attività 5 e l'attività 11 invece si possono svolgere contemporaneamente.

Obiettivo: Bisogna trovare un insieme di attività compatibili di cardinalità massima.

Consideriamo che qualcuno ci dica che l'attività 3 fa parte dell'insieme di attività compatibili di cardinalità massima, allora possiamo cancellare tutte le attività compatibili:

```
1:  [8..12]  // X
2:  [3..5]   // V
3:  [8..11]  // <-
4:  [3..8]   // X
5:  [0..6]   // V
6:  [1..4]   // V
7:  [5..9]   // X
8:  [2..13]  // X
9:  [6..10]  // X
10: [5..7]   // V
11: [12..14] // V
```

Verificare la compatibilità per tutte le combinazioni ha un numero di istanze troppo alto e quindi bisogna trovare un altro metodo. Si può ordinare la lista di attività in base al tempo di fine in modo crescente:

```
1:  [1..4]
2:  [3..5]
3:  [0..6]
4:  [5..7]
5:  [3..8]
6:  [5..9]
7:  [6..10]
8:  [8..11]
9:  [8..12]
10: [2..13]
11: [12..14]
```

Ora è più facile trovare un insieme di attività compatibili di cardinalità maggiore di 3. C'è un teorema che dice chce tra tutte le soluzioni di cardinalità massima ce n'è una che contiene la prima attività della lista ordinata. Si prende la prima attività e si cancellano tutte le attività incompatibili:

```
1:  [1..4]   // <-
2:  [3..5]   // X
3:  [0..6]   // X
```

```
4:  [5..7]   // V
5:  [3..8]   // X
6:  [5..9]   // X
7:  [6..10]  // V
8:  [8..11]  // V
9:  [8..12]  // V
10: [2..13]  // V
11: [12..14] // V
```

Ora si può ripetere il procedimento con le attività rimanenti, perchè il problema si è ridotto ad un problema dello stesso tipo, ma più piccolo.

```
1:  [1..4]   // --
2:  [3..5]   // X
3:  [0..6]   // X
4:  [5..7]   // <-
5:  [3..8]   // X
6:  [5..9]   // X
7:  [6..10]  // X
8:  [8..11]  // V
9:  [8..12]  // V
10: [2..13]  // X
11: [12..14] // V
```

```
1:  [1..4]   // --
2:  [3..5]   // X
3:  [0..6]   // X
4:  [5..7]   // --
5:  [3..8]   // X
6:  [5..9]   // X
7:  [6..10]  // X
8:  [8..11]  // <-
9:  [8..12]  // X
10: [2..13]  // X
11: [12..14] // V
```

```
1:  [1..4]   // --
2:  [3..5]   // X
3:  [0..6]   // X
4:  [5..7]   // --
5:  [3..8]   // X
6:  [5..9]   // X
7:  [6..10]  // X
8:  [8..11]  // --
9:  [8..12]  // X
10: [2..13]  // X
11: [12..14] // --
```

Abbiamo trovato un insieme di attività compatibili di cardinalità massima:

$$\{1, 4, 8, 11\}$$

facendo una scelta greedy.

Per migliorare l'algoritmo si può evitare di controllare tutte le attività maggiori della prima attività compatibile trovata.

Il teorema menzionato sopra si può dimostrare come: Sia A un insieme di attività compatibili, sia $k = \min(A)$ e definiamo $A' = (A - \{k\}) \cup \{1\}$. Si sa che il tempo

di fine di 1 è minore del tempo di fine di k

$$f_1 \leqslant f_k$$

perchè la lista è ordinata. Per questo si sa che ogni attività che si trova nell'insieme A ha tempo di fine maggiore di f_k perchè il tempo di fine di k non è mai minore del tempo di fine di 1, quindi tutte le altre attività avranno tempo di fine maggiore di f_k e quindi non si sovrappongono con 1. Di conseguenza A' è un insieme di attività compatibili e il primo elemento fa sempre parte dell'insieme di attività compatibili di cardinalità massima.

L'algoritmo mostrato nell'esempio 6.1 è il seguente:

```
// S e' un array che contiene tutti i tempi di inizio delle
    attivita'
// F e' un array che contiene tutti i tempi di fine delle attivita'
activity_selector(s,f)
  sort s,f by f
  A <- {1}
  j <- 1
  for i <- 2 to length(s)
    if s[i] >= f[j]
      A <- A U {i}
      j <- i

  return A
```

Questo algoritmo ha complessità $O(n \log n)$ perchè viene fatto un ordinamento, ma se le attività sono già ordinate allora la complessità è $O(n)$. La complessità di questo algoritmo dipende anche dalle condizioni in cui viene utilizzato.

Esempio 6.2. Ci sono 3 tipi di sostanze che hanno un prezzo diverso in base al peso:

```
1: 20Kg -> 100$
2: 30Kg -> 120$
3: 10Kg -> 60$
```

Bisogna inserirle in uno zaino che può contenere al massimo 50Kg. Si vuole massimizzare il valore dello zaino, quindi bisogna vedere quanto costa ogni sostanza per 1Kg

```
1: 20Kg -> 100$ -> 5$/Kg
2: 30Kg -> 120$ -> 4$/Kg
3: 10Kg -> 60$ -> 6$/Kg
```

Per massimizzare il costo si può prendere 20Kg della prima sostanza, 20Kg della seconda e 10Kg della terza. Si da quindi priorità alla sostanza che vale di più per 1Kg e si riempie lo zaino con quella.

Si può ordinare la lista in base al valore al chilo e si prende il massimo della prima risorsa della lista ordinata ripetendo il procedimento sullo stesso problema, ma con uno zaino con capacità minore, cioè si è ridotto il problema ad uno dello stesso tipo, ma più piccolo.

Se queste sostanze non fossero divisibili allora il problema non ha nè una soluzione greedy, nè una soluzione dinamica. Questo tipo di problema si chiama

problema di tipo NP-completo, cioè non è stato ancora trovato un algoritmo che risolve il problema in tempo polinomiale.

Esercizio 6.1. Data una stringa si modifichi inserendo il minimo numero di caratteri per renderla palindroma:

```
  AAB     ABC    BACB
   |       |      |
   V       V      V
  BAAB   ABCBA   BACAB
```

Si parte dall'esterno e si va verso l'interno, se i caratteri sono uguali si passa ad un problema dello stesso tipo, ma più piccolo. Se i caratteri non sono uguali bisogna capire quale dei due caratteri inserire per rendere la stringa palindroma. In questo caso bisogna risolvere entrambi i problemi e vedere quale dei due ha il minor numero di caratteri da inserire.

```
// s e' la stringa
// i e j sono gli indici di inizio e fine della stringa
minimum_palindrome(s,i,j)
  if j <= i
    return 0
  if s[i] == s[j]
    return minimum_palindrome(s,i+1,j-1)
  else
    return 1 + min(
      minimum_palindrome(s,i+1,j),
      minimum_palindrome(s,i,j-1)
      )
```

Implementando la memoization la soluzione diventa:

```
minimum_palindrome(s,i,j)
  if j <= i
    return 0
  if M[i,j] != +inf
    return M[i,j]
  if s[i] == s[j]
    M[i,j] <- minimum_palindrome(s,i+1,j-1)
  else
    M[i,j] <- 1 + min(
      minimum_palindrome(s,i+1,j),
      minimum_palindrome(s,i,j-1)
      )
  return M[i,j]
```

Completare creando la matrice Z che contiene i caratteri da inserire per rendere la stringa palindroma. E implementare l'algoritmo in modo iterativo.

Esercizio 6.2 (Love calculator)**.** L'affinità tra due persone è proporzionale alla stringa più piccola tra le stringhe che contengono i due nomi come sottostringhe. Dati due nomi trovare la stringa più corta che li contiene come sottostringhe e in quanti modi si può scrivere una stringa partendo dai due nomi. Ad esempio:

```
      USA & USSR
          |
          V
        USASR
        USSAR
        USSRA
```