Algoritmi

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2024/2025

Indice

1 Introduzione

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

1.1 Confronto tra algoritmi

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

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

1.2 Rappresentazione dei dati

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

2 Calcolo della complessità

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

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

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

2.1 Linguaggi di programmazione

Ogni linguaggio di programmazione è formato da diversi blocchi:

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

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

2.1.1 Blocchi iterativi

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

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

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

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

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

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:

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

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

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

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

$$m$$

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

La complessità totale alla fine sarebbe stata:

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

Ma ciò che ci interessa veramente è:

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

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

$$nml$$

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

2.3 Ordine di grandezza

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

$$f \in O(g)$$

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

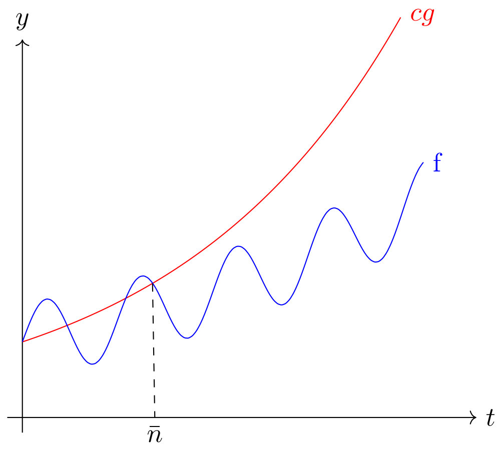

Figura 1: Esempio di funzione $f \in O(g)$

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

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

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:

$$n \leq c2n$$

Quindi è vero

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

$$2n \leq 2n$$

Quindi è vero

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

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

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

Trasformo la disequazione:

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

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

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

Se la definizione di $\Omega(g)$ è:

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

sappiamo che:

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

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

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

Trasformo la disequazione:

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

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

Esempio 2.5.

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

Dimostrazione:
Ipotesi

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

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

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

Quindi:

$$c = (c_1 + c_2)$$

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

Esempio 2.6. Se

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

è vero che:

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

Dimostrazione:
Ipotesi

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

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

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

Quindi:

$$c = c_1 \cdot c_2$$

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

3 Studio degli algoritmi

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

- **Input**: Sequenza $(a_1, \dots, a_n)$ 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 $(a'_1, \dots, a'_n)$ di $(a_1, \dots, a_n)$ tale che:

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

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

3.1 Insertion sort

Divide la sequenza in due parti:

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

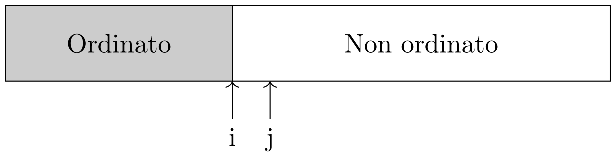

Figura 2: Parte ordinata e non ordinata

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:

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

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

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

sostituiamo nella funzione di partenza:

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

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

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

Esempio 3.2.

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

$$T(n) \leq cn$$

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

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

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

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

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

Esempio 3.3.

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

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

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

L'equazione diventa:

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

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

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

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, comsiderando il merge con complessità lineare, risulta:

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

Applicando il teorema dell'esperto si ottiene:

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

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

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

Quindi siamo nel secondo caso del teorema dell'esperto:

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

Definizione del merge sort:

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

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

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

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

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.

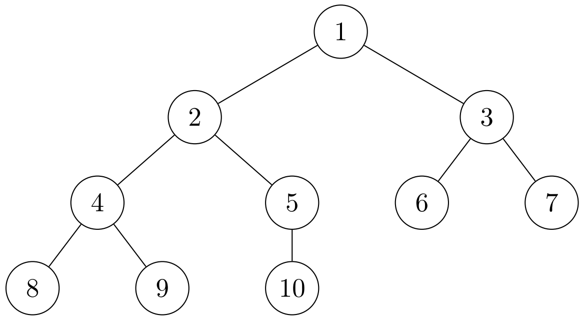

Figura 3: Heap con l'indice di un array associato ai nodi

3.5.1 Proprietà

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

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

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

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:

$$0 < c < 1$$

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

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

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

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

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

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

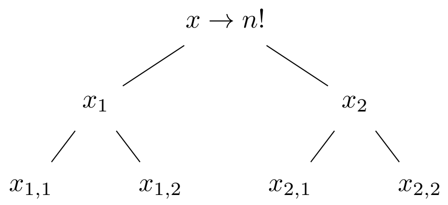

Figura 4: Heap con l'indice di un array associato ai nodi

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

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

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:

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

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

$$\dots \quad n^2 \quad n^1 \quad n^0$$

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 $a =$ `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 $\left\lfloor \frac{n}{2} \right\rfloor$ 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 $\Omega(\left\lceil \frac{3}{2}n \right\rceil - 2) = \Omega(n)$ (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:

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

  La complessita media è lineare.

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

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

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

1. Dividi gli oggetti in $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi di 5 elementi più un eventuale gruppo con meno di 5 elementi.
2. Calcola il mediano di ogni gruppo di 5 elementi (si ordina e si prende l'elemento centrale). $\Theta(n)$
3. Calcola ricorsivamente il mediano x dei mediani

$$T\left(\left\lceil \frac{n}{5} \right\rceil\right)$$

4. Partiziona con perno x e calcola k (numero di elementi a sinistra). $\Theta(n)$

5. Se $i < k$ cerca a sinistra l'elemento i, altrimenti cerca a destra l'elemento $i - k$. La chiamata ricorsiva va fatta su un numero di elementi sufficientemente piccolo, e deve risultare un proporzione di n, quindi ad esempio dividere in gruppi da 3 elementi non funzionerebbe.

$$T(?)$$

$$\begin{array}{ccccccccccccc}
m_1 & \rightarrow & m_2 & \rightarrow & m_3 & \rightarrow & \underset{x}{m_4} & \rightarrow & m_5 & \rightarrow & m_6 & \rightarrow & m_7 \\
 & & & & & & & & \downarrow & & \downarrow & & \downarrow \\
 & & & & & & & & m_{5,4} & & m_{6,4} & & m_{7,4} \\
 & & & & & & & & \downarrow & & \downarrow & & \\
 & & & & & & & & m_{5,5} & & m_{6,5} & &
\end{array}$$

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).

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

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

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 $T(n) \leq cn$, che la disequazione sia vera:

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

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

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

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

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

Quindi $T(n) \leq cn$ 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)`