Appunti di Algoritmi e Strutture Dati

Algoritmi e Strutture Dati (prof. Pirola) - CdL Informatica Unimib - 23/24

Federico Zotti

2024-07-01

Indice

1 Introduzione

Un'**algoritmo** è una sequenza di istruzioni **elementari** (devono essere comprese e eseguite dall'esecutore) che permettono di risolvere un problema computazionale (ovvero per ogni possibile input produce l'output corretto).

Per definire un **problema** è necessario specificare:

- Il tipo del parametro in input
- Il tipo del risultato in output
- Il legame tra input e output

Un'**istanza** di un problema si ottiene specificando uno dei possibili valori in input specifico per il problema.

1.1 Algoritmo: Definizione dell'ordinamento di un vettore

Sort:

- Input: `Array Int` (`Dim` n) $\rightarrow A = \langle a_1, a_2, \ldots, a_n \rangle$
- Output: `Array Int` (`Dim` n) $\rightarrow A' = \langle a'_1, a'_2, \ldots, a'_n \rangle$

A' è una permutazione di A, tale che $a'_1 \leq a'_{i+1} \quad \forall i \,.\, 1 \leq i \leq n-1$.

1.2 Scelta di un algoritmo

L'algoritmo migliore è quello che utilizza il minor numero di risorse.

Le risorse sono:

- Il tempo di esecuzione
- Lo spazio (memoria) utilizzato

1.2.1 Tempo di esecuzione

Per calcolare il tempo utilizziamo una funzione $T(n)$. n rappresenta la quantità di dati in input.

- $T_p(n)$ rappresenta il caso peggiore
- $T_n(n)$ rappresenta il caso "medio" (non è la media dei due)
- $T_m(n)$ rappresenta il caso migliore

1.2.1.1 Esempio

- Algoritmo 1: $T(n) = 100000 \cdot n$
- Algoritmo 2: $T(n) = 10 \cdot n^3$
- Algoritmo 3: $T(n) = 1 \cdot 2^n$

In questo caso il migliore dipende dal grado di n, dunque l'algoritmo 1 risulta quello più veloce. Per numeri di n molto piccoli invece è meglio calcolare caso per caso il tempo. Nel caso ci siano più n, si considera quello con il grado maggiore.

$$T(n) = 7n^3 + 2n + 10000 \sim n^3$$

1.3 Algoritmo: Ricerca sequenziale

- V: vettore di interi
- k: intero da cercare nel vettore
- p: posizione nel vettore

```
1  Ricerca_Seq(V, k)
2      p = 1
3      while (V[p] != k) and (p <= V.length)
4          p = p + 1
5      if p > V.length
6          return -1
7      else
8          return p
```

Analisi del tempo di esecuzione:

- Caso peggiore: $k \neq V[\,] \Rightarrow T(n) = 3 + 2 \cdot n + 1 \sim n$
- Caso migliore: $k = V[1] \Rightarrow T(n) = 4 \sim c$
- Caso medio: $k = V[\frac{n}{2}] \Rightarrow 3 + 2\frac{n}{2}(\pm 1) \sim n$

Se V è ordinato ci si può fermare appena trova un numero più grande di k.

```
Ricerca_Seq(V, k)
    p = 1
    while (V[p] < k) and (p <= V.length)
        p = p + 1
    if (V[p] != k) or (p > V.length)
        return -1
    else
        return p
```

Analisi del tempo di esecuzione:

- Caso migliore: $V[p] \geq k \Rightarrow 4 \sim c$
- Caso peggiore: $k > V[p] \Rightarrow 3 + 2n \sim n$
- Caso medio: $k = V\left[\frac{n}{2}\right] \Rightarrow 3 + 2\frac{n}{2} \cdot \frac{1}{2}$

Per avere un'ottimizzazione significativa si può sfruttare il fatto che il vettore è ordinato per implementare una semplice ricerca binaria (spezzare il vettore e guardare solo una metà).

```
Ricerca_Dic(V, k)
    sx = 1
    dx = V.length
    p = (dx + sx) div 2
    while (V[p] != k) and (sx < dx)
        if k > V[p]
            sx = p + 1
        else
            dx = p - 1
        p = (dx + sx) div 2

    if V[p] = k
        return p
    else
        return -1
```

Analisi del tempo di esecuzione:

- Caso migliore: $V\left[\frac{n}{2}\right] = k \Rightarrow t_m(n) = 6 \sim c$
- Caso peggiore: $k \notin V \Rightarrow T_p(n) = 5 + 4 \cdot (\log_2 n) + 1 \sim \log_2 n$
- Caso medio: $T(n) = \frac{T_p(n)}{2} \sim \log_2 n$

2 Problema computazionale

Un problema computazionale è una relazione tra input e output.

$$P \subseteq I \times O$$

2.1 Esempio: Ricerca in un vettore

Input:

- Un vettore V di n elementi
- Un valore k

Output:

- Un intero i, t.c. $i = -1$ se $k \notin V$ altrimenti $V[i] = k$

Ci sono tanti algoritmi per risolvere questo problema. Un esempio è la **ricerca sequenziale**.

> Aggiungere algo. #todo-uni

2.2 Esempio: Ricerca in un vettore ordinato

Input:

- Un vettore **ordinato** V di n elementi
- Un valore k

Output:

- Un intero i, t.c. $i = -1$ se $k \notin V$ altrimenti $V[i] = k$

Questo problema è diverso dal precedente, perché i dati in input sono diversi. Essendo che l'algoritmo della ricerca sequenziale è corretto per tutti i vettori in

input, è corretto anche per i vettori ordinati. Esistono però algoritmi specifici per questo problema. Per esempio la **ricerca binaria** (o dicotomica).

> Inserire algo. #todo-uni

3 Trovare il miglior algoritmo

Generalmente il **tempo di esecuzione** e lo **spazio utilizzato** da un algoritmo sono legati alla grandezza dell'input. Dunque è possibile esprimere questi due dati come funzioni di n.

$$T(n)$$

$$S(n)$$

> In questo corso ci si concentrerà di più su $T(n)$.

Il tempo di esecuzione non può essere espresso in secondi, perché questi dipendono dalla velocità dell'elaboratore (da quanto tempo viene impiegato ad eseguire ogni singola istruzione). Dunque il tempo viene espresso in quante istruzioni devono essere eseguite.

Non sempre però il tempo dipende soltanto da n. Per esempio $300 + 200$ risulta molto più semplice di $345 + 783$, nonostante abbiano lo stesso numero di cifre. Dunque $T(n)$ oscilla tra il caso migliore $T_{migl}(n)$ e il caso peggiore $T_{pegg}(n)$.

3.1 Esempio

Riprendendo gli algoritmi di ricerca si possono definire i tempi di esecuzione.

- **Ricerca sequenziale:**
  - $T_{migl}(n) = a_1$

  - $T_{pegg}(n) = a_2 + b_2 n$

- **Ricerca binaria:**

  - $T_{migl}(n) = a_3$
  - $T_{pegg}(n) = a_4 + b_4 \log_2 n$

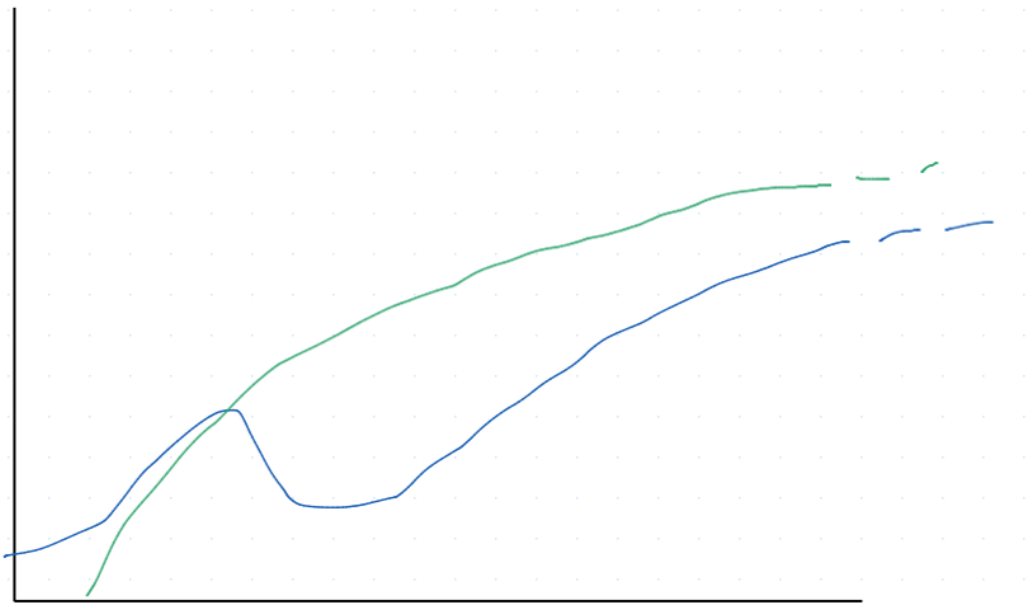
4 Notazioni asintotiche

4.1 Limite asintotico superiore

Assunzioni:

- Le funzioni $T(n)$, $S(n)$ sono definite nei numeri naturali, ma i grafici vengono definiti nei reali per semplicità
- Le funzioni sono definitivamente positive

> Aggiungere grafico con una funzione casuale e una funzione def maggiore. #todo-uni



- Blu: $T_{pegg}(n) = f(n)$

- Verde: $O(g(n))$

$$O(g(n)) = \{f(n) \mid \exists\, n_0 > 0, c > 0 : 0 \leq f(n) \leq c \cdot g(n) \forall\, n > n_0\}$$

Ovvero $O(g(n))$ è l'insieme delle funzione superiormente limitate da $g(n)$ per una costante c $(g(n) \cdot c)$.

Esempio:

- $f(n) = 3n^2$
- $3n^2 \in O(n^3)$? ovvero
- $\exists\, n_0, c > 0 : 0 \leq 3n^2 \leq c \cdot n^3 \forall n > n_0$?

$n \geq \frac{3}{c}$? Questo è verificato per esempio con $c = 3, n_0 = 1$.
È possibile però che esista una funzione $g(n)$ "minore": $3n^2 = O(n^2)$?
$c \geq 3$? Questo è verificato per esempio con $c = 4, n_0 = 1$.
Dunque $3n^2 = O(n^2)$.
Si può provare con una funzione ancora più "bassa", ma facendo i calcoli la condizione non viene soddisfatta.

Esempio:

- $f(n) = 5n^3 + n^2$
- $f(n) \in O(n^3)$? ovvero
- $\exists\, n_0, c > 0 : 0 \leq 5n^3 + n^2 \leq c \cdot n^3 \forall n > n_0$?

$6n^3 \leq cn^3$? Questo è verificato per esempio con $c = 7, n_0 = 1$.
Si può provare con una funzione ancora più "bassa", ma facendo i calcoli la condizione non viene soddisfatta.

4.2 Limite asintotico inferiore

> Aggiungere grafico con funzione $T(n)$ e una funzione def inferiore.

$$\Omega(g(n)) = \{\, f(n) \mid \exists\, n_0 > 0, c > 0 \text{ t.c. } 0 \leq c \cdot g(n) \leq f(n) \,\forall\, n > n_0 \,\}$$

Es:

- $f(n) = 3n^2$
- $g(n) = n$
- $3n^2 \in \Omega(n)$?

$$\exists\, c > 0, n_0 > 0 \quad 0 \leq cn \leq 3n^2 \quad \forall\, n > n_0$$

$$3n^2 \geq cn$$

$$n \geq \frac{c}{3}$$

$$c = 3 \quad n_0 = 1$$

Dunque è verificato.

Si può dire che $3n^2 = \Omega(n)$ *(impropriamente)*.

Proseguendo, $3n^2 = \Omega(n^2)$?

$$\exists\, c > 0, n_0 > 0 \quad 0 \leq cn^2 \leq 3n^2 \quad \forall\, n > n_0$$

$$3n^2 \geq cn^2$$

$$3 \geq c$$

$$c = 3 \quad n_0 = 1$$

Dunque è verificato.

Continuando, $3n^2 = \Omega(n^3)$?

$$\exists\, c > 0, n_0 > 0 \quad 0 \leq cn^3 \leq 3n^2 \quad \forall\, n > n_0$$

$$3n^2 \geq cn^3$$

$$3 \geq cn$$

$$n \leq \frac{3}{c}$$

Questo non è verificato definitivamente. $3n^2 \notin \Omega(n^3)$. Lo stesso vale per tutti gli $\Omega(n^\varepsilon) \quad \forall\, \varepsilon > 2$.

4.3 Limite asintotico stretto

Devo trovare due costanti c_1, c_2 tale che la funzione $T(n)$ è compresa definitivamente tra $c_1 g(n)$ e $c_2 g(n)$.

> Disegnare grafico #todo-uni : funzione casuale $T(n)$ con due "rette"/"curve" che stanno def. sopra e def. sotto $T(n)$.

$$\Theta(g(n)) = \{\, f(n) \mid \exists\, n_0, c_1, c_2 > 0 \,:\, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\, n > n_0 \,\}$$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Es 1: continuando gli esempi precedenti, sappiamo che

$$3n^2 = \Omega(n^2) \land 3n^2 = O(n^2) \implies 3n^2 = \Theta(n^2)$$

Es 2:

- $f(n) = 5n^3 + n^2$
- $g(n) = n^3$
- $5n^3 + n^2 = \Omega(n^3)$?

$$5n^3 + n^2 \geq 5n^3 = \Omega(n^3)$$

$$5n^3 + n^2 = O(n^3) \implies 5n^3 + n^2 = \Theta(n^3)$$

Es 3:

- $f(n) = 5n^3 - 10n^2 - 30n$
- $g(n) = n^3$

- $f(n) = \Omega(n^3)$?

$$\exists\, c, n_0 > 0 \quad 0 \leq cn^3 \leq 5n^3 - 10n^2 - 30n \quad \forall\, n > n_0$$
$$5n^3 - 10n^2 - 30n \geq cn^3$$
$$5n^3 - cn^3 \geq 10n^2 + 30n$$
$$(5 - c)n^3 \geq 10n^2 + 30n$$

Sapendo che

- $10n^2 \leq 10n^2$
- $30n \leq 30n^2$

possiamo scrivere $10n^2 + 30n \leq 10n^2 + 30n^2$. Adesso dobbiamo stabilire se $(5 - c)n^2 \geq 10n^2 + 30n^2$ (perché implica la disuguaglianza precedente).

$$(5 - c)n^2 \geq 10n^2 + 30n^2$$
$$n \geq \frac{40}{5 - c} \qquad \text{con } c < 5$$
$$c = 1 \quad n_0 = 9$$

Dunque è verificato.
Inoltre $f(n) = O(n^3) \implies f(n) = \Theta(n^3)$.

Dim: $\Omega()$ è uguale a n al grado massimo del polinomio?

- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 \quad a_k > 0$
- $g(n) = n^k$
- $f(n) = \Omega(n^k)$?

$$\exists\, c, n_0 > 0 \quad 0 \leq cn^k \leq f(n) \quad \forall\, n > n_0$$

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \geq cn^k$$

$$(a_k - c)n^k \geq -a_{k-1} n^{k-1} - \ldots - a_0$$

$$-a_{k-1} n^{k-1} - \ldots - a_0 \leq |a_{k-1}| n^{k-1} + |a_{k-2}| n^{k-1} + \ldots + |a_0| n^{k-1}$$

$$(a_k - c)n^k \geq \sum_{i=0}^{k-1} |a_i| n^{k-1}$$

$$(a_k - c)n^k \geq n^{k-1} \sum_{i=0}^{k-1} |a_i|$$

$$(a_k - c)n \geq \sum_{i=0}^{k-1} |a_i|$$

$$n \geq \frac{\sum_{i=0}^{k-1} |a_i|}{a_k - c} \qquad \text{con } c < a_k$$

Dimostrato che un polinomio è un omega di n al grado massimo.

Dim: un'esponenziale è limitato inferiormente da un polinomio.

- $f(n) = 2^n$
- $2^n = O(2^n)$
- $2^n = \Omega(n^b) \quad \forall\, b > 0$?

$$\lim_{n \to +\infty} \frac{n^b}{2^n} = 0$$

$$\implies \exists\, n_0 \quad \forall\, n > n_0 \quad \frac{n^b}{2^n} < 1$$

$$2^n > 1 n^b$$

Quindi tutti i polinomi limitano inferiormente un esponenziale. E sapendo che $2^n = \Omega(2^n)$, si considera questo perché è "il più grande". Ciò implica $2^n = \Theta(2^n)$.

Dim: lo stesso per i logaritmi.

- $f(n) = \log_2^a n$
- $\log_2^a n = O(n^b) \quad \forall\, b > 0$?

$$\lim_{n \to +\infty} \frac{\log_2^a n}{n^b} = 0$$

$$\implies \exists\, n_0 \quad \forall\, n > n_0 \quad \frac{\log_2^a n}{n^b} < 1$$

$$\log_2^a = O(n^b)$$

Quindi tutti i polinomi limitano inferiormente un logaritmo. E sapendo che $\log_2^a = O(\log_2^a)$, si considera questo perché è "il più piccolo". Ciò implica $\log_2^a = \Theta(\log_2^a)$.

4.4 Scala degli asintoti

- 1
- $\log n$
- $n^b \quad \forall\, 0 < b < 1$
- n
- $n \cdot \log n$
- $n^{1+\varepsilon} \quad \forall\, 0 < \varepsilon < 1$
- n^2
- $n^2 \cdot \log n$
- $n^{2+\varepsilon} \quad \forall\, 0 < \varepsilon < 1$
- n^3
- $n^3 \cdot \log n$
- $n^{2+\varepsilon} \quad \forall\, 0 < \varepsilon < 1$
- $\dots$
- $a^n \quad \forall\, a > 1$ *(ogni a è una classe a se)*
- $n!$

- n^n

4.5 Esempi

Es 1: ricerca in un vettore ordinato.

Algoritmo	Tempo caso migliore	Tempo caso peggiore
Ricerca sequenziale	$a_1 = \Omega(1)$	$a_2 + b_2 n = O(n)$
Ricerca dicotomica	$a_3 = \Omega(1)$	$a_4 + b_4 \log n = O(\log n)$

5 Selection Sort

5.1 Problema: Trova minimo

Attenzione: gli indici degli array partono da 1 e non da 0.

- **Input:** un vettore V di n interi
- **Output:** una posizione i t.c. $V[i] \leq V[j] \quad \forall\, 1 \leq j \leq n$

```
1  Trova_Minimo(V)
2      posMin = 1
3      for i = 2 to V.length
4          if V[i] < V[posMin]
5              posMin = i
6      return posMin
```

- **Caso migliore:** $1 + n + (n-1) + 0 + 1 = 2n + 1 = \Omega(n)$
- **Caso peggiore:** $1 + n + (n-1) + (n-1) + 1 = 3n = O(n)$

Dunque il tempo di calcolo di questo algoritmo è $\Theta(n)$.

5.1.1 Dimostrare che l'algoritmo è corretto

Riscriviamo l'algoritmo con un `while`.


```
1 Trova_Minimo(V)
2     posMin = 1
3     i = 2
4     while i <= V.length
5         if V[i] < V[posMin]
6             posMin = i
7         i = i + 1
8     return posMin
```

- **Invariante di ciclo** *(condizione che deve essere sempre vera)*: `posMin` è la posizione che contiene il valore più piccolo di $V[1 \dots i-1]$.
- **Inizializzazione** *(prima del while)*: `posMin` è la posizione del minimo di $V[1 \dots 1]$? Sì
- **Conservazione** *(ciò che è vero alla prima istr. del ciclo deve essere vero anche alla fine)*: l'invariante di ciclo è vera alla fine se è vera anche all'inizio
- **Terminazione** *(la conseguenza dell'inizializzazione e della conservazione)*: $i = V.\text{length} + 1$. `posMin` è la posizione del minimo di $V[1 \dots V.\text{length}] = V$

5.2 Problema: Ordinamento di vettori

- **Input:** A un vettore di n interi
- **Output:** un vettore di n interi che contiene gli stessi valori di A ma ordinati in modo crescente

5.2.1 Esempio

Supponiamo di avere un vettore

$$A = (5, 2, 4, 6, 1, 3)$$

Per ordinarlo si possono seguire questi passi:

1. Trovo il minimo e la sua posizione
2. Scambio il minimo con il primo elemento

$$A = (1, 2, 4, 6, 5, 3)$$

3. Ora continuo considerando la restante parte del vettore $(2, 4, 6, 5, 3)$ trovando il minimo e la sua posizione *(il secondo elemento più piccolo)*
4. Scambio questo elemento con il secondo elemento di A
5. Continuo così finché non esaurisco il vettore

$$A = (1, 2, 3, 4, 5, 6)$$

5.2.2 Implementazione

```
1  Selection_Sort(A)
2      for i = 1 to A.length
3          // Trova l'elemento minimo
4          posmin = i
5          for j = i + 1 to A.length
6              if A[j] < A[posmin]
7                  posmin = j
8
9          scambia A[posmin] con A[i]
```

Per semplificare i calcoli il primo **for** va da 1 a `A.length` al posto che `A.length` − 1, e non entriamo nel secondo **for** quando `i = A.length`. Inoltre consideriamo i parametri come Java, quindi i vettori vengono passati per riferimento e dunque l'algoritmo non necessita di una **return**.

5.2.3 Analisi dei tempi di calcolo

> **for** a to b: $T(n) = b - a + 1 + 1$

Selection Sort	$T_{migl}(n)$	$T_{pegg}(n)$
for i = 1 to A.length	$n + 1$	$n + 1$
posmin = i	n	n

Selection Sort	$T_{migl}(n)$	$T_{pegg}(n)$
`for j = i + 1 to A.length`	$\Theta(n^2)$	$\Theta(n^2)$
`if A[j] < A[posmin]`	$\Theta(n^2)$	$\Theta(n^2)$
`posmin = j`	0	$\Theta(n^2)$
`scambia A[posmin] con A[i]`	n	n

- **Caso migliore:** $T_{migl}(n) = \Omega(n^2)$
- **Caso peggiore:** $T_{migl}(n) = O(n^2)$

Dunque l'algoritmo è $\Theta(n^2)$.

5.2.4 Considerazioni accessorie

5.2.4.1 Stabilità In applicazioni più complesse gli elementi del vettore da ordinare sono **chiavi** a strutture con dati aggiuntivi. Dunque molto raramente due elementi (considerando anche questi dati aggiuntivi) risultano uguali. Se però bisogna ordinare il vettore, il selection sort si comporta nel seguente modo:

$$(5, 6, 3^A, 1, 3^B)$$
$$(1, 6, 3^A, 5, 3^B)$$
$$(1, 3^A, 6, 5, 3^B)$$
$$(1, 3^A, 3^B, 5, 6)$$
$$(1, 3^A, 3^B, 5, 6)$$

In questo caso 3^A risulta "prima" di 3^B. In quest'altro esempio invece:

$$(3^A, 5, 6, 3^B, 1)$$

$$(1, 5, 6, 3^B, 3^A)$$

$$(1, 3^B, 6, 5, 3^A)$$

$$(1, 3^B, 3^A, 5, 6)$$

$$(1, 3^B, 3^A, 5, 6)$$

3^B risulta "prima" di 3^A.

Ciò indica che **il selection sort non è stabile** perché non preserva l'ordine degli elementi di ugual valore.

5.2.4.2 In-place

Un algoritmo è **in-place** se lavora direttamente sul vettore in input, e non su un'altra copia.

Il selection sort è in-place.

6 Insertion sort

- **Input:** un vettore A di n interi
- **Output:** un vettore di n interi che contiene gli stessi valori di A ma ordinati in modo crescente

6.1 Esempio

Supponiamo di avere un vettore

$$A = (5, 2, 4, 6, 1, 3)$$

1. Controlliamo il secondo elemento (2) con il primo elemento (5).
2. Essendo minore sposto il 2 a sinistra del 5

3. Non ci sono altri elementi a sinistra, quindi mi fermo

$$A = (2, 5, 4, 6, 1, 3)$$

4. Controllo il terzo elemento (4) con il precedente (5)
5. Essendo minore sposto il 4 a sinistra del 5
6. Controllo il 4 con il precedente (2)
7. Essendo il 4 maggiore di 2, mi fermo

$$A = (2, 4, 5, 6, 1, 3)$$

8. Controllo il quarto elemento (6) con il precedente (5)
9. Essendo maggiore, mi fermo

$$A = (2, 4, 5, 6, 1, 3)$$

10. Controllo il quinto elemento (1) con il precedente (6)
11. Essendo minore sposto l'1 a sinistra del 6
12. Controllo l'1 con il precedente
13. Essendo minore sposto l'1 a sinistra del 5
14. Continuo così finché l'elemento precedente è minore di 1 o non ci sono più elementi
15. Continuo così per tutti gli elementi restanti nel vettore

$$A = (1, 2, 3, 4, 5, 6)$$

Il vettore è ordinato.

6.2 Implementazione


```
1  Insertion_Sort(A)
2      for i = 1 to A.length
3          key = A[i]
4          j = i - 1
5
6          while (j > 0) and (A[j] > key)
7              A[j+1] = A[j]
8              A[j] = key
9              j = j - 1
```

6.3 Analisi dei tempi di calcolo

Insertion sort	$T_{migl}(n)$	$T_{pegg}(n)$
`for i = 1 to A.length`	$n+1$	$n+1$
`key = A[i]`	n	n
`j = i - 1`	n	n
`while (j > 0)and (A[j] > key)`	n	$\Theta(n^2)$
`A[j+1] = A[j]`	0	$\Theta(n^2)$
`A[j] = key`	0	$\Theta(n^2)$
`j = j - 1`	0	$\Theta(n^2)$

- **Caso migliore:** *(A è già ordinato in modo crescente)* $T_{migl}(n) = \Omega(n)$
- **Caso peggiore:** *(A è ordinato in modo decrescente)* $T_{pegg}(n) = O(n^2)$

6.4 Considerazioni accessorie

L'insertion sort è **stabile** e **in-place**.

7 Ricorsione

7.1 Problema: Esponenziale di un numero

- **Input:** un reale a e un naturale n
- **Output:** un reale b tale che $b = a^n$

7.1.1 Implementazione iterativa

```
1  EsponenzialeIt(a, n)
2      ris = 1
3      for i = 1 to n
4          ris = a * ris
5      return ris
```

$$T(n) = \Theta(n)$$

7.1.2 Implementazione ricorsiva

```
1  EsponenzialeRic(a, n)
2      if n == 0
3          return 1
4
5      return a * EsponenzialeRic(a, n-1)
```

7.1.3 Tempi dell'implementazione ricorsiva

In questo specifico esempio non ha senso calcolare i tempi migliore e peggiori perché la quantità in input invariabile.

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

Il problema è che rimane la ricorrenza anche all'interno del calcolo.

Riscrivo il caso $n \neq 0$:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ &= T(n-1) + c \\ &= \big(T(n-2) + c\big) + c \\ &= T(n-3) + 3c \\ &= \dots \\ &= T(n-i) + ic \end{aligned}$$

Ad un certo punto si arriverà che $n - i = 0$. A quel punto si sarà nel caso $T(n) = \Theta(1)$. Dunque il tutto si può riscrivere come

$$T(n) = T(0) + nc = \Theta(1) + nc = \Theta(n)$$

Attenzione: questi casi non sono il peggiore e il migliore, perché in quei casi non si può fissare n. Questi sono solo in casi in cui n è fissato.

Nonostante le due implementazioni siano asintoticamente equivalenti, nella realtà le versioni iterative sono più veloci di quelle ricorsive.

7.1.4 Seconda implementazione ricorsiva

$$a^n = \begin{cases} 1 & \text{se } n = 0 \\ a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{se } n \text{ è pari} \\ a \cdot a^{\left\lfloor \frac{n}{2} \right\rfloor} \cdot a^{\left\lfloor \frac{n}{2} \right\rfloor} & \text{se } n \text{ è dispari} \end{cases}$$

```
1  EsponenzialeRicV2(a, n)
2      if n == 0
3          return 1
4
5      if n == 1
6          return a
7
8      m = floor(n/2)
9      ris = EsponenzialeRicV2(a, m)
10     ris = ris * ris
11
12     if dispari(n)
```

```
        ris = a * ris

    return ris
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ \Theta(1) & \text{se } n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(1) & \text{altrimenti} \end{cases}$$

Supponendo che n sia sempre pari, e che tutti i $\frac{n}{2}$ siano pari:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c \\ &= T\left(\frac{n}{4}\right) + 2c \\ &= \dots \\ &= T\left(\frac{n}{2^i}\right) + ic \end{aligned}$$

Il tutto termina quando $i = \log_2 n$. A quel punto si può riscrivere come

$$\begin{aligned} T(n) &= T(1) + ic \\ &= T(1) + c \log_2 n \\ &= \Theta(\log_2 n) \\ &= \Theta(\log n) \end{aligned}$$

7.2 Ricerca dicotomica

$$A = (1, 5, 7, 11, 15, 18, 21, 32, 35, 39, 40, 45, 48, 51, 57, 61)$$

Dobbiamo trovare 40 all'interno di A.

1. Trovo l'elemento a metà (pos. 8): 32
2. Lo confronto con 40: è minore, dunque scarto tutta la metà inferiore
3. Trovo l'elemento a metà della parte superiore $(35, 39, 40, 45, 48, 51, 57, 61)$ (pos. 12): 45

4. Lo confronto con 40: è maggiore, allora scarto la parte superiore
5. Trovo l'elemento a metà della parte inferiore $(35, 39, 40, 45)$ (pos. 11): 40
6. Lo confronto con 40: è minore *o uguale*, allora scarto la parte inferiore
7. Trovo l'elemento a metà della parte superiore $(40, 45)$ (pos. 11): 40
8. Continuo così finché il vettore considerato avrà dimensione 1, a quel punto l'elemento dovrebbe essere l'elemento cercato se è presente.
9. Confronto l'elemento trovato con quello che stavo cercando

7.2.1 Implementazione

- **Input:** A vettore ordinato di interi, x elemento da cercare
- **Input ricorsivi:** *input* + l indice sinistro, r indice destro
- **Output:** un valore booleano che indica se l'elemento x è presente nell'array

```
1  Ricerca_Dic_Ric(A, x, l, r)
2      if l == r
3          return x == A[l]
4
5      if l > r
6          return FALSE
7
8      mid = floor( ( l + r ) / 2 )
9
10     if x <= A[mid]
11         return Ricerca_Dic_Ric(A, x, l, mid)
12     else
13         return Ricerca_Dic_Ric(A, x, mid+1, r)
14
15 Ricerca_Dic_Entry(A, x)
16     return Ricerca_Dic_Ric(A, x, 1, A.length)
```

7.2.2 Analisi dei tempi

$n = \text{dim input} = r - l + 1$.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ \Theta(1) + T\left(\frac{n}{2}\right) & \text{altrimenti} \end{cases}$$

Semplificando la ricorsione si ottiene

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

7.3 Merge sort

Vogliamo ordinare il seguente vettore

$$A = (11, 5, 8, 7, 13, 6, 9, 1)$$

Possiamo dividere l'array in due parti e ordinarle separatamente:

$$(5, 7, 8, 11) \qquad (1, 6, 9, 13)$$

A questo punto confronto i primi elementi delle due metà. Sicuramente uno dei due è il primo elemento di A. Continuo così fino a "svuotare" le due metà e ricostruire A ordinato.

Per ordinare le due metà iniziali è possibile svolgere lo stesso algoritmo ricorsivamente.

7.3.1 Implementazione

- **Input:** A vettore di interi
- **Input ricorsivi:** *input* + l indice sinistro, r indice destro
- **Output:** stesso A ordinato in modo crescente

```
Merge(A, l, mid, r)
    T = new [(r-l+1) int]
    i = l         // Indice della parte l -> mid
    j = mid + 1   // Indice della parte mid+1 -> r
    k = 1         // Indice del vettore T dove inserire i nuovi elementi

    while (i <= mid) and (j <= r)
        if A[i] <= A[j]
            T[k] = A[i]
            i = i + 1
        if A[i] > A[j]
            T[k] = A[j]
```



```
            j = j + 1
        k = k + 1

    while i <= mid
        T[k] = A[i]
        i = i + 1
        k = k + 1

    while j <= r
        T[k] = A[j]
        j = j + 1
        k = k + 1

    for c = 1 to T.length
        A[c+l-1] = T[c]


Merge_Sort_Ric(A, l, r)
    if l < r
        mid = floor( ( l + r ) / 2)
        Merge_Sort_Ric(A, l, mid)
        Merge_Sort_Ric(A, mid+1, r)

        Merge(A, l, mid, r)


Merge_Sort_Entry(A)
    return Merge_Sort_Ric(A, 1, A.length)
```

7.3.2 Analisi dei tempi

$n = \text{dim input} = r - l + 1$

$$T_{\text{Merge}}(n) = \Theta(n)$$

$$T_{\text{MSort}}(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$
$$= \Theta(n \log n)$$

7.3.3 Differenze di implementazione

Primo tipo:

```
if l < r
    mid = floor( ( l + r ) / 2 )
    Merge_Sort_Ric(A, l, mid)
    Merge_Sort_Ric(A, mid+1, r)
    Merge(A, l, mid, r)
```


Secondo tipo:

```
if l >= r
    return
mid = floor( ( l + r ) / 2 )
Merge_Sort_Ric(A, l, mid)
Merge_Sort_Ric(A, mid+1, r)
Merge(A, l, mid, r)
```

Nel secondo caso viene evidenziato il caso base rispetto al passo ricorsivo. I due tipi sono equivalenti.

7.3.4 Altre considerazioni

Questo algoritmo non è in-place, perché la `Merge` utilizza un array di appoggio. Inoltre è stabile con questa implementazione di `Merge`.

La merge sort è un esempio classico di un algoritmo **divide et impera** *(divide, impera, combina)*. Questa tipologia di algoritmi divide un problema in più sottoproblemi. Vengono risolti questi sottoproblemi e vengono ricombinati per ottenere la soluzione al problema iniziale.

Questi algoritmi hanno

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \text{ suff. piccolo } \forall\, n \leq n_0 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{se } n > n_0 \end{cases}$$

7.4 Teorema dell'esperto

Se abbiamo un algoritmo **divide et impera** (come il merge sort precedente), dunque avente la formula

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \text{ suff. piccolo } \forall\, n \leq n_0 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{se } n > n_0 \end{cases}$$

con $a \geq 1, b > 1$ e $f(n)$ asintoticamente non negativa, possiamo semplificare la

formula con i seguenti casi:

1. Se $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ per $\varepsilon > 0 \implies T(n) = \Theta\left(n^{\log_b a}\right)$
2. Se $f(n) = \Theta\left(n^{\log_b a}\right) \implies T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right)$
3. Se $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ per $\varepsilon > 0$ e $\exists c < 1$ t.c. $af\left(\frac{n}{b}\right) \leq cf(n)$ per n suff. grandi $\implies T(n) = \Theta(f(n))$

7.4.1 Esempi

Es 1: $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$

- $a = 1$
- $b = 2$
- $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- $f(n) = \Theta(1) = c$

Vediamo i casi:

1. $\exists \varepsilon > 0$ t.c. $c = O(n^{0-\varepsilon})$? *NO*
2. $c = \Theta(n^0) = \Theta(1)$? *SÌ*

Dunque $T(n) = \Theta(n^0 \cdot \log n) = \Theta(\log n)$.

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

- $a = 2$
- $b = 2$
- $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
- $f(n) = \Theta(n) = cn$

Vediamo i casi:

1. $\exists \varepsilon > 0$ t.c. $c = O(n^{1-\varepsilon})$? *NO*
2. $n = \Theta(n)$? *SÌ*

Dunque $T(n) = \Theta(n \log n)$.

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

- $a = 1$
- $b = \frac{3}{2}$
- $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$
- $f(n) = \Theta(1) = c$

Vediamo i casi:

1. $\exists \varepsilon > 0$ t.c. $c = O(n^{0-\varepsilon})$? *NO*
2. $c = \Theta(n^0) = \Theta(1)$? *SÌ*

Dunque $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Es 3: $T(n) = 3T\left(\frac{n}{2}\right) + n^3$

- $a = 3$
- $b = 2$
- $n^{\log_b a} = n^{\log_2 3} = n^{1,\dots}$
- $f(n) = n^3$

Vediamo i casi:

1. $\exists \varepsilon > 0$ t.c. $c = O(n^{1,\cdots-\varepsilon})$? *NO*
2. $n^3 = \Theta(n^{1,\cdots})$? *NO*
3. $\exists \varepsilon > 0 \quad n^3 = \Omega(n^{1,\cdots+\varepsilon})$ *SÌ*

Condizione di regolarità: $\exists c < 1$ t.c. $af\left(\frac{n}{b}\right) < cf(n)$ *SÌ*, $c = \frac{3}{8}$
Dunque $T(n) = \Theta(n^3)$.

7.5 Problema: Ricerca del minimo

- **Input:** vettore A di n interi
- **Output:** la posizione del minimo in A

Per creare un algoritmo ricorsivo è possibile utilizzare il fatto che

$$\min(A) = \min(A[0], \min(A[1 \dots n-1]))$$

Per delimitare la porzione di array su cui opera l'algoritmo usiamo due parametri `l` e `r`. Questo evita di copiare ogni volta il sotto-array da passare alla chiamata ricorsiva (con costo $\Theta(n)$). È necessaria una funzione di utilità che inizializza i parametri aggiuntivi per la chiamata ricorsiva iniziale.

```java
// JAVA Class (LAB)

// Funzione ricorsiva
private static int _findPosMin(int[] A, int l, int r){
    // Caso base: ho un solo valore
    if (l + 1 == r)
        return l;
    // Chiamata ricorsiva
    int rest = _findPosMin(A, l+1, r);
    if (A[l] < A[rest])
        return l;
    return rest;
}

// Funzione di inizializzazione parametri
public static int findPosMin(int[] A) {
    return _findPosMin(A, 0, a.length);
}
```

7.5.1 Selection sort ricorsivo

Con la funzione appena definita, è possibile riscrivere anche il selection sort in maniera ricorsiva

```java
// JAVA Class (LAB)

// Funzione ricorsiva
private static void _selectionsort(int [] A, int l, int r) {
    // Caso base: array vuoto o con un elemento
    if (l + 1 >= r)
        return;
    // Passo ricorsivo:
    // Cerca posizione minimo
    int posMin = _findPosMin(A, l, r);
    // Scambia
    int tmp = A[posMin];
    a[posMin] = a[l];
    a[l] = tmp;
    // Prosegui ricorsivamente
    _selectionsort(A, l + 1, r);
}

```



```
19  // Funzione di inizializzazione parametri
20  public static void selectionsort(int[] A) {
21      _selectionsort(A, 0, A.length);
22  }
```

8 Quick Sort

> Completare #todo-uni

> Noi abbiamo guardato la partizione di Hoare, ma si può seguire tranquillamente quella di Lomuto. Hai fini dell'esame non è importante.

> Viene dimostrata anche la correttezza.

8.1 Tempi di calcolo

Nel caso peggiore (array ordinato) l'equazione di ricorrenza è

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

con k costante.

Nel caso migliore invece è

$$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n) = \Theta(n \log n)$$

con $0 < \alpha < 1$.

Dunque il Quick Sort è $\Omega(n \log n)$ e $O(n^2)$.

Se l'input è casuale, il tempo atteso però è $\Theta(n \log n)$.

8.2 Random partition

```
Rnd_Partition(A, l, r):
    Scambia A[l] con un A[l-c] scelto casualmente
    return Partition(A, l, r)

Rnd_QS(A, l, r):
    if l<r then
        cut = Rnd_Partition(A, l, r)
        Rnd_QS(A, l, cut)
        Rnd_QS(A, cut+1, r)
```

9 Problema di selezione

- **Input:** un array A di n interi distinti, un intero i $\in [1, n]$
- **Output:** il valore di A tale che è maggiore esattamente di i-1 elementi

9.1 Esempio

$$15 \quad 4 \quad 8 \quad 12 \quad 22 \quad 26 \quad 9 \quad 19$$

- Con $i = 1$ l'output sarà 4 (minimo)
- Con $i = 8$ l'output sarà 26 (massimo)
- Con $i = \frac{n}{2} = 4$ l'output sarà 12 (mediana)

9.2 Implementazioni

9.2.1 Naive

```
Naive(A, i):
    Ordina A
    return A[i]
```

9.2.2 Selez

```
Cut(A, l, r):
    // Lo stesso del quick sort

```



```
Selez(A, i, l, r):
    if l == r then
        return A[l]

    cut = Partition(A, l, r)
    dim1 = cut - l + 1
    if i <= dim1 then
        return Selez(A, i, l, cut)
    else
        return Selez(A, i-dim1, cut+1, r)
```

Questa implementazione è una ricerca binaria mischiata al quick sort.

9.3 Tempi

Se divide a metà

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

Se invece toglie un elemento per volta

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

Quindi

$$O(n^2) \qquad \Omega(n)$$

10 Counting Sort

- **Input:** A vettore di interi e k intero t.c. $\forall i . 0 \leq A[i] \leq k$

```
CountingSort(A, k):
    C = nuovo vettore con indici da 0 a k
    for i = 0 to k:
        C[i] = 0

    for i = 1 to A.length:
        C[A[i]] = C[A[i]] + 1

    for i = 1 to k:
```



```
10          C[i] = C[i] + C[i-1]
11
12      B = nuovo vettore da 1 a A.length
13      for i = A.length down to 1:
14          B[C[A[i]]] = A[i]
15          C[A[i]] = C[A[i]] - 1
16
17      return B
```

10.1 Calcolo dei tempi

$$T(n,k) = \Theta(n+k)$$

11 Radix Sort

- **Input:** A vettore di interi, d massimo numero di cifre di ogni elemento

```
1  RadixSort(A, d):
2      for i = 1 to d:
3          aplica un alg. di ord. stabile per ordinare A sulla i-esima cifra meno
               significativa
```

11.1 Calcolo dei tempi

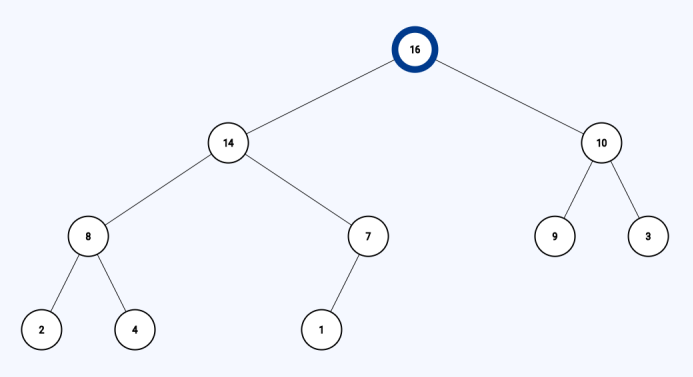
$$T(n,d) = \Theta(d \cdot T_0(n))$$

12 Binary Heap

Un **binary heap** è una struttura dati memorizzata in un array e un certo campo accessorio (**heap size**).

- Essa può essere vista come albero binario quasi completo
- Tutti gli elementi tranne il primo soddisfano la **proprietà dello heap**

16	11	10	8	7	9	3	2	4	1



$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

$$\text{left}(i) = 2i$$

$$\text{right}(i) = 2i + 1$$

$$A.\text{length} = 10$$

$$A.\text{heap\_size} = 10$$

12.1 Proprietà dello heap

$$\forall i \qquad 1 < i \leq A.\text{heap\_size}$$

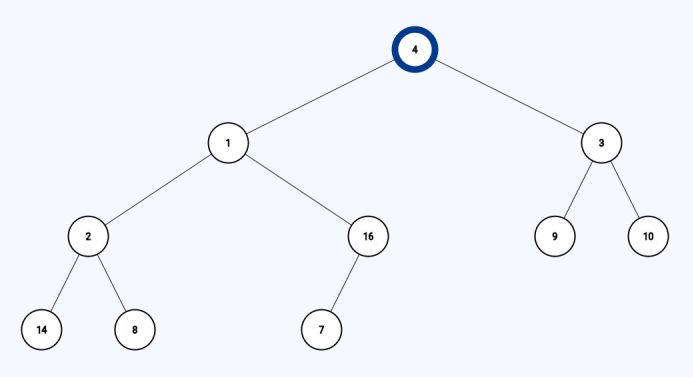
Se A è:

- **Max heap:** $A[parent(i)] \geq A[i]$
- **Min heap:** $A[parent(i)] \leq A[i]$

Oss: se A è ordinato in modo decrescente, allora è un heap. Il contrario invece non è sempre vero.

12.2 Esempi

$$A = \quad 4 \quad 1 \quad 3 \quad 2 \quad 16 \quad 9 \quad 10 \quad 14 \quad 8 \quad 7$$



12.3 Generare un binary heap

Per rendere un normale albero un binary heap abbiamo bisogno delle seguenti procedure:

- `BuildMaxHeap` per rendere un array un max heap
- `MinHeapify` per rendere un albero un min heap

12.3.1 MaxHeapify

```
MaxHeapify(A, i):
    l = left(i)
    r = right(i)
    largest = i
    if l <= A.heap_size AND A[largest] < A[l]:
        largest = l
    if r <= A.heap_size AND A[largest] < A[r]:
        largest = r
    if i != largest:
        A[i], A[largest] = A[largest], A[i]
        MaxHeapify(A, largest)

BuildMaxHeap(A):
    A.heap_size = A.length
    for i = floor(A.heap_size / 2) down to 1 do
        MaxHeapify(A, i)
```

12.3.1.1 Tempi

I tempi della `MaxHeapify` sono:

$$T(h) = \Theta(1) + T(h-1) = O(h)$$

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

I tempi di `BuildMaxHeap` sono:

$$T(n) \leq \frac{n}{2} \cdot O(\log n) = O(n \log n)$$
$$T(n) = \Theta(n)$$

12.4 Heap Sort

```
1 HeapSort(A):
2     BuildMaxHeap(A)
3     for i = A.length to 2:
4         scambia A[1] con A[i]
5         A.heap_size = A.heap_size - 1
6         MaxHeapify(A, 1)
```

$$\text{HS} = \text{BH} + n \cdot \big(\text{Estraz. Max} + \text{Heapify}\big)$$
$$T(n) = \Theta(n) + O(n \log n) = O(n \log n)$$

Questo è un algoritmo **in-place** ma **non stabile**.

13 Insiemi dinamici

Gli insiemi manipolati dagli algoritmi possono crescere, ridursi o cambiare nel tempo. Questi insiemi sono detti **dinamici**.

Molti algoritmi richiedono soltanto la capacità di **inserire** e **cancellare** degli elementi da un insieme e di **verificare l'appartenenza** di un elemento ad un insieme.

Un insieme dinamico che supporta queste operazioni è un **dizionario**.

13.1 Dizionari

Un dizionario può essere implementato tramite un array in cui, oltre all'attributo `length`, viene memorizzato anche l'attributo `size` che specifica quanti valori

sono effettivamente presenti nel dizionario.

```java
private int[] data;
public final int length;
public int size;

C01_DictionaryArray(int length) {
    this.data = new int[length];
    this.length = length;
    this.size = 0;
}
```

13.1.1 Inserimento in fondo

La forma più semplice di inserimento è l'inserimento in fondo al dizionario.

> `size` indica direttamente in che punto fare l'inserimento in fondo.

```java
public void append(int x) {
    if (size == length)
        // Non ho abbastanza spazio per inserire un nuovo elemento
        throw new RuntimeException("overflow");

    data[size] = x;
    size = size + 1;
}
```

Nota: potrei non aver spazio per inserire un elemento (*overflow*).

Tempi: supponendo di avere n elementi, l'inserimento in fondo richiede $\Theta(1)$.

13.1.2 Inserimento in posizione data

Per inserire un elemento `x` in una data posizioni `i` devo "fargli posto" spostando a destra tutti gli elementi in posizioni comprese fra `i` (incluso) e `size` (escluso).

```java
public void insert(int i, int x) {
    if (size == length)
        throw new RuntimeException("overflow");
    if (i > size)
        throw new RuntimeException("impossibile inserire lasciando buchi");

    for (int j = size; j > i; --j)
        data[j] = data[j - 1];

    data[i] = x;
    size = size + 1;
}
```


Tempi: supponendo di avere n elementi, l'inserimento richiede $O(n)$ nel caso peggiore.

13.1.3 Inserimento in cima

L'inserimento in cima (`prepend`) potrebbe richiamare la `insert(0, x)`. La implementiamo però esplicitamente.

```
public void prepend(int x) {
    if (size == length)
        throw new RuntimeException("overflow");

    for (int j = size; j > 0; --j)
        data[j] = data[j - 1];

    data[0] = x;
    size = size + 1;
}
```

Tempi: supponendo di avere n elementi, l'inserimento in cima richiede $\Theta(n)$.

13.1.4 Cancellazione in fondo

La forma più semplice di **cancellazione** è la cancellazione dell'ultimo elemento.

```
public int deleteLast() {
    if (size == 0)
        throw new RuntimeException("underflow");

    size = size - 1;
    return data[size];
}
```

Note:

- Devo controllare che ci sia almeno un elemento (altrimenti errore di **underflow**)
- Le `delete` spesso restituiscono l'elemento cancellato

Tempi: $\Theta(1)$.

13.1.5 Cancellazione in posizione data

Per cancellare l'elemento in posizione `i` devo spostare a sinistra tutti gli elementi che sono presenti nelle posizioni comprese fra `i` e `size`. L'ultimo elemento resta fisicamente nell'array, ma non viene considerato perché decrementiamo `size`.

```
public int delete(int i) {
    if (size == 0)
        throw new RuntimeException("underflow");
    if (i >= size)
        throw new RuntimeException("impossibile eliminare");

    int ris = data[i];

    for (int j = i; j < size-1; j++)
        data[j] = data[j+1]

    size = size - 1;
    return ris;
}
```

13.1.6 Cancellazione in cima

```
public int deleteFirst() {
    if (size == 0)
        throw new RuntimeException("underflow");

    int first = data[0];

    for (int j = 0; j < size-1; j++)
        data[j] = data[j+1];

    size = size - 1;
    return first;
}
```

13.1.7 Ricerca valore

Ricerca il valore `x` nel dizionario. Restituisce l'indice dell'elemento o -1 se non è presente.

```
public int search(int x) {
    int j = 0;

    while (j < size) {
        if (x == data[j]) {
            return j;
        }
        j++;
    }

    return -1;
```



```
12  }
```

13.1.8 Cancellazione elementi uguali a un valore dato

Elimina tutti gli elementi uguali a `x`. La funzione restituisce **`true`** se il valore era presente almeno una volta, **`false`** altrimenti.

```
1   public boolean deleteAllValues(int x) {
2       int n_found = 0;
3
4       for (int i = 0; i < size; i++) {
5           if (data[i] == x) {
6               n_found++;
7           } else {
8               data[i - n_found] = data[i];
9           }
10      }
11
12      size = size - n_found;
13      return n_found > 0;
14  }
```

13.1.9 Limitazioni

Array consentono di memorizzare insiemi dinamici ma hanno capacità massima ridotta:

- Alzare la capacità riduce la probabilità di overflow ma lascia spazio non utilizzato
- Ridurre la capacità riduce lo spazio non utilizzato ma incrementa la probabilità di overflow
- Con riallocazioni dinamiche si perde tempo per la copia (analisi ammortizzata dei tempi)

13.2 Liste concatenate

Una **lista concatenata** (*linked list*) è una struttura dati in cui ciascun dato è memorizzato in un nodo e i nodi sono collegati tra loro in ordine lineare.

È necessario implementare le operazioni di accesso in modo da preservare la struttura dati.

Definiamo una classe (inner class) `Node` per rappresentare ciascun nodo e manteniamo un riferimento `head` al primo nodo.

```
public class SimpleList {
    public static class Node {
        int key;
        Node next;

        public Node(int key) {
            this.key = key;
            this.next = null;
        }
    }

    private Node head;
}
```

Una lista vuota non ha nodi, quindi `head = null`.

```
SimpleList() {
    this.head = null;
}
```

Di una lista concatenata in genere non si lavora con la sua dimensione, ma si vuole sapere solo se è vuota o se ha almeno un elemento.

```
boolean isEmpty() {
    return this.head == null;
}
```

13.2.1 Inserimento in testa

```
public Node prepend(Node x) {
    x.next = this.head;
    this.head = x;
    return x;
}
```

Tempi: $O(1)$.

13.2.2 Inserimento dopo un nodo dato

`insertAfter(Node i, Node x)` inserisce il nuovo nodo `x` come successore del nodo `i` già presente nella lista. Se `i == null` allora è inserimento in

testa.

```
public Node insertAfter(Node i, Node x) {
    if (i == null) { // inserimento in testa == prepend
        x.next = this.head;
        this.head = x;
    } else { // inserimento di x dopo i
        x.next = i.next;
        i.next = x;
    }
    return x;
}
```

Tempi: $O(1)$.

13.2.3 Inserimento in coda

Per inserire in coda `x` devo trovare il riferimento `i` all'ultimo elemento, poi procedo come da `insertAfter(i, x)`.

```
public Node append(Node x) {
    if (this.head == null) {
        this.head = x;
    } else { // Cerco l'ultimo nodo (cioè il nodo con next == null)
        Node i = this.head;
        while (i.next != null) { // dato che head!=null, allora i.next non mi
            darà errore
            i = i.next;
        } // Aggiungo x come successore di i
        i.next = x;
    }
    return x;
}
```

Tempi: $O(n)$.

13.2.4 Cancellazione in testa

```
public Node deleteFirst() {
    if (isEmpty()) {
        // se la lista è vuota non faccio nulla return null;
    }
    Node x = head;
    head = head.next; // la lista NON è vuota, quindi posso accedere a head.
        next
    x.next = null;
    return x;
}
```

Tempi: $O(1)$.

13.2.5 Cancellazione in coda

Come per l'inserimento in coda devo scorrere la lista m,a devo cercare il *penultimo* elemento (e poi assegnare `null` come successore).

Caso particolare è quando ho un solo elemento:

- Non esiste il penultimo
- Cancellarlo equivale a svuotare la lista

```
public Node deleteLast() {
    if (isEmpty())
        // se la lista è vuota non faccio nulla
        return null;
    // se head.next==null ho un solo elemento -> lo cancello
    if (head.next == null) {
        Node x = head;
        head = null;
        return x;
    }
    Node i = head;
    // Cerco il penultimo..
    while (i.next.next != null) {
        i = i.next;
    }
    Node x = i.next;
    // ...e cancello il successivo
    i.next = null;
    return x;
}
```

Tempi: $O(n)$.

13.2.6 Cancellazione di un nodo dato

Per cancellare il nodo `i` devo cercare il suo predecessore `j` per aggiornare il suo `next` a `i.next`.

```
public Node delete(Node i) {
    if (head == i) {
        // se i==head allora
        head = i.next;
        // i non ha predecessore
    } else {
        Node j = head;
        // Cerco il predecessore...
        while (j.next != i) {
            j = j.next;
        }
        // e aggiorno il suo successore "saltando" i
        j.next = j.next.next;
    }
    i.next = null;
```



```
16      return i;
17  }
```

Tempi: $O(n)$.

13.2.7 Ricerca elemento

Cerca il riferimento al primo nodo che contiene la chiave `key`. Deve restituire **`null`** se `key` non è presente.

```
1  public Node search(int key) {
2      Node i = head;
3      while (i != null && i.key != key) {
4          i = i.next;
5      }
6
7      return i;
8  }
```

13.2.8 Cancellazione nodo con valore date

Elimina il primo nodo che contiene la chiave `key`. Deve restituire il riferimento al nodo che ha cancellato o **`null`** se non presente.

```
1  public Node deleteValue(int key) {
2      Node prev = null;
3      Node i = this.head;
4
5      while (i != null && i.key != key) {
6          prev = i;
7          i = i.next;
8      }
9
10     if (i != null) {
11         if (i == this.head) {
12             // key è in head, quindi prev == null
13             this.head = i.next;
14         } else {
15             prev.next = i.next;
16         }
17         i.next = null;
18     }
19
20     return i;
21 }
```


13.2.9 Cancellazione di tutti i nodi con valore dato

Elimina tutti i nodi che contengono la chiave `key`. La funzione deve restituire **`true`** se il valore era presente almeno una volta, **`false`** altrimenti.

```java
public boolean deleteAllValues(int key) {
    boolean deleted = false;
    Node i = this.head;
    Node prev = null;
    while (i != null) {
        if (i.key == key) {
            if (i == this.head) {
                this.head = i.next;
            } else {
                prev.next = i.next;
            }
            deleted = true;
            i = i.next;
        } else {
            prev = i;
            i = i.next;
        }
    }
    return deleted;
}
```

13.3 Riepilogo

Operazione	Array	Liste
`prepend`	$O(n)$	$O(1)$
`insert`	$O(n)$	$O(1)$
`append`	$O(1)$	$O(n)$
`deleteFirst`	$O(n)$	$O(1)$
`delete`	$O(n)$	$O(n)$
`deleteLast`	$O(1)$	$O(n)$
`search`	$O(n)$	$O(n)$

13.4 Liste doppiamente concatenate

Possiamo rendere $O(1)$ le operazioni rimanenti?

- `search` sicuramente no
- `append` avremmo bisogno di mantenere il riferimento all'ultimo nodo (quindi bisogna aggiungere `tail`)
- `delete` avremmo bisogno di conoscere il predecessore di ciascun nodo (quindi bisogna aggiungere `prev` a ciascun nodo)
- `deleteLast` avremmo bisogno di `tail` e `prev`

```java
public class DoublyLinkedList {
    public static class Node {
        int key;
        Node prev;
        Node next;

        public Node(int key) {
            this.key = key;
            this.prev = this.next = null;
        }
    }

    private Node head;
    private Node tail;

    DoublyLinkedList() {
        this.head = this.tail = null;
    }

    boolean isEmpty() {
        return this.head == null;
    }
}
```

13.4.1 Inserimento in testa

```java
public Node prepend(Node x) {
    if (this.isEmpty()) {
        this.head = this.tail = x;
    } else {
        x.next = this.head;
        this.head.prev = x;
        this.head = x;
    }

    return x;
}
```

Tempo: $O(1)$.

13.4.2 Inserimento in coda

```
public Node append(Node x) {
    if (this.isEmpty()) {
        this.head = this.tail = x;
    } else {
        this.tail.next = x;
        x.prev = this.tail;
        this.tail = x;
    }

    return x;
}
```

Tempi: $O(1)$.

13.4.3 Cancellazione in testa

```
public Node deleteFirst() {
    if (this.isEmpty())
        return null;

    Node x = head;
    if (head == tail) {
        // c'è un solo nodo
        head = tail = null;
    } else {
        // ci sono almeno due nodi
        head = head.next;
        head.prev = null;
        x.next = null;
    }

    return x;
}
```

Tempi: $O(1)$.

La cancellazione in coda è simmetrica (su `tail`).

13.4.4 Cancellazione di un nodo dato

La cancellazione di un nodo `i` dato aggiorna il riferimento `next` di `i.prev` e il riferimento `prev` di `i.next`. Se `i.prev==null` allora `i` è la testa, si aggiorna `head`. Se `i.next==null` allora `i` è la coda, si aggiorna `tail`.

Attenzione: `i` può essere sia testa che coda!

```
public Node delete(Node i) {
    if (i == this.head) {
```



```
3           this.head = i.next;
4       } else {
5           i.prev.next = i.next;
6       }
7
8       if (i == this.tail) {
9           this.tail = i.prev;
10      } else {
11          i.next.prev = i.prev;
12      }
13
14      i.prev = null;
15      i.next = null;
16      return i;
17  }
```

Tempi: $O(1)$.

13.4.5 Inserimento successore

Aggiunge `x` come successore di `i`. Se `i==null` deve aggiungere in testa. Restituisce il nodo inserito (`x`).

Tempi: $O(1)$.

13.4.6 Cancellazione in coda

Cancella l'ultimo nodo. Restituisce il nodo cancellato. Non effettua cancellazioni e restituisce `null` se la lista è vuota.

Tempi: $O(1)$.

13.5 Riepilogo 2

Operazione	Array	Liste	Liste D. Conc.
`prepend`	$O(n)$	$O(1)$	$O(1)$
`insert`	$O(n)$	$O(1)$	$O(1)$

Operazione	Array	Liste	Liste D. Conc.
`append`	$O(1)$	$O(n)$	$O(1)$
`deleteFirst`	$O(n)$	$O(1)$	$O(1)$
`delete`	$O(n)$	$O(n)$	$O(1)$
`deleteLast`	$O(1)$	$O(n)$	$O(1)$
`search`	$O(n)$	$O(n)$	$O(n)$

13.6 Confronto strutture dati per dizionari

Spazio occupato per memorizzare elementi:

- **Array**: k elementi con $k \geq n$ la capacità fissa
- **Liste semplici**: n elementi + $n + 1$ puntatori
- **Liste doppiamente concatenate**: n elementi + $2n + 2$ puntatori

Vantaggi e svantaggi:

- Gli array consentono accesso diretto e sono cache-friendly, le liste no.
- Gli array hanno, però, capacità fissa, le liste crescono e si riducono al bisogno.
- Liste dopp. concatenate consentono inserimenti e cancellazioni da entrambi gli estremi in tempo costante, array e liste semplici no.

> Qua poi dovrebbero esserci altri tipi di strutture dati simili alle liste.

14 Coda di Priorità

Una coda di priorità *(priority queue)* è una **struttura dati** con le seguenti operazioni:

- `Max(Q)-> x`: ritorna l'elemento con la priorità massima
- `Extract_Max(Q)-> x`: ritorna l'elemento con la priorità massima e lo rimuove dalla struttura
- `Insert(Q, x)`: inserisce l'elemento alla struttura
- `Increase_Key(Q, x, new_p)`: incrementa la priorità dell'elemento `x` nella struttura

Un esempio di utilizzo di questa struttura è la lista di processi in esecuzione in un computer. Viene utilizzata per gestire lo scheduling.

14.1 Implementazione

Una possibile implementazione delle code di priorità (ma non l'unica) è il **binary heap**.

```
Max(A):
    return A[1]
```

$T(n) = \Theta(1)$

```
Extract_Max(A):
    if A.heap_size == 0:
        error "underflow"

    max = A[1]
    scambia A[1] con A[A.heap_size]
    A.heap_size = A.heap_size - 1
    MaxHeapify(A, 1)
    return max
```

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

```
Insert(A, key):
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = -inf
    Increase_Key(A, A.heap_size, key)
```

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

```
Increase_Key(A, i, new_key):
    if A[i] > new_key:
        error "decr. key"

    A[i] = new_key

    while i > 1 AND A[parent(i)]:
        scambia A[i] con A[parent(i)]
```



```
9           i = parent(i)
```

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

15 Pila (Stack)

È una struttura dati che permette di memorizzare un insieme dinamico di dati nella quale la restituzione degli elementi è controllata dalla proprietà **LIFO** *(last-in, first-out)*.

Ha le seguenti operazioni:

- `Top(S)-> x`: restituisce l'ultimo elemento della pila (senza toglierlo)
- `Pop(S)-> x`: restituisce l'ultimo elemento della pila e lo rimuove
- `Push(S, x)`: aggiunge l'elemento alla pila
- `Stack_Empty(S)-> bool`: restituisce se la pila è vuota o no

15.1 Stack Search

```
1  Stack_Search(S, x):
2      found = FALSE
3      S2 = nuovo stack
4      while not Stack_Empty(S) and not found:
5          y = Pop(S)
6          if x == y:
7              found = TRUE
8          Push(S2, y)
9
10     while not Stack_Empty(S2):
11         Push(S, Pop(S2))
12
13     return found
```

Tempi: $T(n) = O(n) = \Omega(1)$

15.2 Rimozione di elementi da uno stack

```
1  Stack_Delete_All_Values(S, x):
2      S2 = nuovo stack
3      while not Stack_Empty(S):
```

```
        y = Pop(S)
        if x != y:
            Push(S2, y)

    while not Stack_Empty(S2):
        Push(S, Pop(S2))
```

Tempi: $T(n) = \Theta(n)$

15.3 Inserimento di un valore in uno stack ordinato

```
Stack_Insert_Ordered(S, x):
    S2 = nuovo stack

    while not Stack_Empty(S) AND Top(S) < x:
        y = Pop(S)
        Push(S2, y)

    Push(S, x)

    while not Stack_Empty(S2):
        Push(S, Pop(S2))
```

Tempi: $T(n) = O(n) = \Omega(1)$

16 Coda (Queue)

È una struttura dati le cui restituzioni seguono la proprietà **FIFO** *(first-in, first-out)*.

Ha le seguenti operazioni:

- `Head(Q) -> x`: ritorna il valore in testa senza modificare la coda
- `Dequeue(Q) -> x`: ritorna il valore in testa e lo rimuove dalla coda
- `Enqueue(Q, x)`: aggiunge il valore in fondo alla coda
- `Queue_Empty(Q) -> bool`: ritorna se la coda è vuota

16.1 Queue Search

```
Queue_Search(Q, x):
    found = FALSE
    Q2 = nuova coda
```



```
    while not Queue_Empty(Q):
        y = Dequeue(Q)
        if x == y:
            found = TRUE
        Enqueue(Q2, y)

    while not Queue_Empty(Q2):
        enqueue(Q, Dequeue(Q2))

    return found
```

Tempi: $T(n) = \Theta(n)$

16.2 Rimozione di elementi da una coda

```
Queue_Delete_All_Values(Q, x):
    Q2 = nuova coda
    while not Queue_Empty(Q):
        y = Dequeue(Q)
        if x != y:
            Enqueue(Q2, y)

    while not Queue_Empty(Q2):
        Enqueue(Q, Dequeue(Q2))
```

Tempi: $T(n) = \Theta(n)$

16.3 Inserimento di un valore in una coda ordinata

```
Queue_Insert_Ordered(Q, x):
    Q2 = nuova coda

    while not Queue_Empty(Q) AND Head(Q) < x:
        y = Dequeue(Q)
        Enqueue(Q2, y)

    Enqueue(Q, x)

    while not Queue_Empty(Q):
        Enqueue(Q2, Dequeue(Q))

    while not Queue_Empty(Q2):
        Enqueue(Q, Dequeue(Q2))
```

Tempi: $T(n) = \Theta(n)$

17 Alberi binari di ricerca

Un albero binario è un albero binario di ricerca se soddisfa la seguente proprietà:

Sia x un nodo dell'albero.

- Se y è un nodo del sottoalbero sinistro $\implies y.\text{Key} \leq x.\text{Key}$
- Se y è un nodo del sottoalbero destro $\implies y.\text{Key} \geq x.\text{Key}$

17.1 Ricerca su alberi binari

Cerca se il valore k è presente nell'albero binario radicato in x.

```
Bst_Search(x, k):
    if x == NULL:
        retrun NULL

    if x.key == k:
        return x

    if k < x.key:
        return Bst_Search(x.left, k)
    else
        return Bst_Search(x.right, k)
```

Alternativamente esplicitando ulteriormente la ricorsione in coda:

```
Bst_Search(x, k):
    if x == NULL OR x.key == k:
        return x

    if k < x.key:
        x = x.left
    else
        x = x.right

    return Bst_Search(x, k)
```

Questo si può trasformare in una versione iterativa.

```
Bst_Search(T, k):
    x = T.root
    while x != NULL AND x.key != k:
        if k < x:
            x = x.left
        else
            x = x.right

    return x
```

Tempi: $T(n) = \Omega(1) = O(h)$.

17.2 Visita simmetrica (in ordine)


```
Bst_InOrder(x):
    if x != NULL:
        Bst_InOrder(x.left)
        print(x.key)
        Bst_InOrder(x.right)
```

17.3 Visita anticipata (in preordine)

```
Bst_PreOrder(x):
    if x != NULL:
        print(x.key)
        Bst_InOrder(x.left)
        Bst_InOrder(x.right)
```

17.4 Visita posticipata (in postordine)

```
Bst_PostOrder(x):
    if x != NULL:
        Bst_InOrder(x.left)
        Bst_InOrder(x.right)
        print(x.key)
```

17.5 Ricerca del minimo

```
Bst_Min(x):
    if x == NULL:
        return NULL

    while x.left != NULL:
        x = x.left

    return x
```

17.6 Ricerca del massimo

```
Bst_Max(x):
    if x == NULL:
        return NULL

    while x.right != NULL:
        x = x.right

    return x
```


17.7 Ricerca del successivo

Il successivo di un nodo è il minimo del sottoalbero destro (se non è vuoto) oppure l'antenato più vicino di cui il nodo è discendente del sottoalbero sinistro.

```
Bst_Succ(x):
    if x.right != NULL:
        return Bst_Min(x.right)

    y = x.parent
    while y != NULL AND x == y.right:
        x = y
        y = y.parent

    return y
```

Tempi: $O(h)$

17.8 Ricerca del predecessore

Il predecessore di un nodo è il massimo del sottoalbero sinistro (se non è vuoto) oppure l'antenato più vicino di cui il nodo è discendente del sottoalbero destro.

```
Bst_Pred(x):
    if x.left != NULL:
        return Bst_Max(x.left)

    y = y.left
    while y != NULL AND x == y.left:
        x = y
        y = y.parent

    return y
```

Tempi: $O(h)$

17.9 Inserisci nodo

```
Bst_Insert(T, z):
    x = T.root
    y = NULL
    while x != NULL:
        y = x
        if z.key <= x.key:
            x = x.left
        else
            x = x.right

    z.parent = y
    if y == NULL:
```



```
13          T.root = z
14      else if z.key <= y.key:
15          y.left = z
16      else
17          y.right = z
```

Tempi: $O(h)$

17.10 Rimozione di un elemento

L'algoritmo cambia a seconda della posizione dell'elemento:

- Se l'elemento è una foglia, basta disconnetterlo
- Se l'elemento ha un solo sottoalbero, basta togliere l'elemento e "riattaccare" il sottoalbero
- Se l'elemento ha due sottoalberi, bisogna sostituirlo con il successore o il predecessore

```
1  Bst_Delete(T, z):
2
3      // CASO FOGLIA
4      if z.left == NULL AND z.right == NULL:
5          if z.parent == NULL:
6              T.root = NULL
7          else if z == z.parent.left:
8              z.parent.left = NULL
9          else
10             z.parent.right = NULL
11
12     // CASO UN SOTTOALBERO SINISTRO
13     else if z.right == NULL:
14         Contrazione_Nodo(T, z, z.left)
15
16     // CASO UN SOTTOALBERO SINISTRO
17     else if z.left == NULL:
18         Contrazione_Nodo(T, z, z.right)
19
20     // CASO DUE SOTTOALBERI
21     else:
22         succ = Bst_Min(z.right) //oppure Bst_Succ(z)
23         z.key = succ.key
24         Bst_Delete(T, succ)
```

Tempi: $T(n) = O(h)$

```
1  Contrazione_Nodo(T, z, child):
2
3      // Se z non ha parent, sto contraendo la radice
4      if z.parent == NULL:
5          T.root = child
6
7      // Se z è figlio sinistro
```

```
    else if z.parent.left == z:
        z.parent.left = child

    // Se z è figlio destro
    else:
        z.parent.right = child
```

Tempi: $T(n) = \Theta(1)$

18 Grafi

$$G = (V, E)$$

con $E \subseteq V^2$ l'insieme delle coppie.

Un grafo è denso se $|E|$ è vicino a $|V|^2$. È invece sparso se $|E| \ll |V|^2$.

18.1 Rappresentazione tramite liste

Considerando un grafo con 5 nodi, salviamo ogni nodo in un array. Ogni elemento di questo array sarà l'head di una linked list che contiene tutti i nodi collegati al nodo corrispondente alla posizione nell'array.

L'ordine di queste liste è spesso arbitrario.

In termini asintotici l'utilizzo di memoria è

$$\Theta(|V| + |E|)$$

18.2 Rappresentazione tramite matrici di incidenza

Considerando sempre lo stesso grafo con 5 nodi, creiamo una matrice quadrata 5×5. In ogni cella è presente 1 se l'arco è presente tra gli elementi corrispondenti alla riga e alla colonna, 0 se non è presente.

In termini asintotici l'utilizzo di memoria è

$$\Theta(|V|^2)$$

18.3 Problema: Distanza da S (BFS)

- **Input:** $G = (V, E), s \in V$
- **Output:** distanza di ciascun $v \in V$ da s

Il grafo è il seguente:

- $A \rightarrow B, E$
- $B \rightarrow A, F$
- $C \rightarrow F, G, D$
- $D \rightarrow G, C, H$
- $E \rightarrow A$
- $F \rightarrow G, C, B$
- $G \rightarrow F, C, D, H$
- $H \rightarrow G, D$

18.3.1 Implementazione della visita in ampiezza del grafo

```
Bfs_Visit(G, s):
    foreach v in G.V:
        v.color = bianco
        v.d = +INF
        v.pi = NULL
    s.color = grigio
    s.pi = NULL
    s.d = 0

    Q = nuova coda
    Enqueue(Q, s)

    while not Queue_Empty(Q):
        v = Dequeue(Q)
        foreach u in G.Adj[v]:
            if u.color == bianco:
                u.color = grigio
                u.d = v.d + 1
                u.pi = v
                Enqueue(Q, u)
        v.color = nero
```


Tempi: $T(n) = \Theta(|V| + |E|)$

```
Stampa_Percorso(v):
    if v.d == +INF:
        Print("nessun percorso")
    else if v.pi == NULL:
        Print(v)
    else
        Stampa_Percorso(v.pi)
        Print(v)
```

Tempi: $T(n) = O(?)$

18.4 DFS

Possibili colori per i nodi:

- **Bianco** se non è stato scoperto
- **Grigio** se è visitato ma non completato (non ho guardato tutti i suoi archi uscenti)
- **Nero** se completato

Inoltre si salva il predecessore (`v.pi`) di ogni vertice.

Tempi:

- `v.d`: tempo discovery
- `v.f`: tempo fine visita

$$1 \leq \text{v.d} < \text{v.f} \leq 2|V|$$

18.4.1 Implementazione

```
DFS(G):
    for each v in G.V:
        v.color = BIANCO
        v.pi = NULL
    tempo = 0
    for each v in G.V:
        if v.color == BIANCO:
            DFS_Visit(G, v)
```



```
DFS_Visit(G,v):
    tempo = tempo - 1
    v.d = tempo
    v.color = GRIGIO
    for each u in G.adj[v]:
        if u.color == BIANCO:
            u.pi = v
            DFS_Visit(G, u)
    v.color = NERO
    tempo = tempo + 1
    v.f = tempo
```

19 DAG

19.1 Ordinamento topologico di un DAG

```
Topological_Sort(G):
    Visita G in profondità
    Ogni volta che si completa la visita di un vertice v, inserisci v in testa
        a L
    return L
```

