ALGORITMI E STRUTTURE DATI

Docenti

Emanuela Merelli

Luca Tesei

Studente

Giorgio Della Roscia

Università degli studi di Camerino

Informatica per la Comunicazione Digitale (L-31)

Scuola di Scienze e Tecnologie

A.A. 2023/2024

Indice

Figure

Tabelle

Codici

Algoritmi

Approfondimenti

Sezione 1

FONDAMENTI

Definizione (algoritmo)**.** Procedimento di calcolo esplicito, descrivibile con n regole che conduce al risultato dopo n operazioni.

Il concetto appena descritto è inscindibile da quello di dato, che viene preso e manipolato per ottenere un certo output.

Definizione (programma)**.** Formulazione concreta di un algoritmo astratto tramite appositi costrutti dei linguaggi.

Ogni programma deve chiaramente risolvere un problema, ma si cerca di farlo nel modo più efficiente possibile: sia in termini di tempo che di risorse.

1.1 Analisi asintotica

La notazione asintotica aiuta a definire gli andamenti di certe funzioni.

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = \omega(g(n)) \approx a > b$$

1.1.1 O grande

Pone un limite superiore alla complessità. Siano $f(n)$ e $g(n)$ due funzioni non negative, $f(n) = O(g(n))$ se esistono due costanti positive c e n_0 tali che $0 \leq f(n) \leq cg(n) \forall n \geq n_0$.

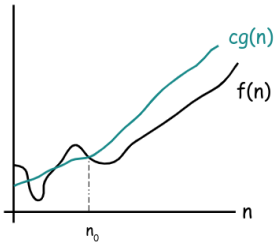


Figura 1.1: Andamento funzione O grande

La $f(n)$ evidenziata è solamente una delle molteplici funzioni limitate superiormente da $g(n)$.

1.1.2 Ω grande

Pone un limite inferiore alla complessità. Siano $f(n)$ e $g(n)$ due funzioni non negative, $f(n) = \Omega(g(n))$ se esistono due costanti positive c e n_0 tali che $0 \leq cg(n) \leq f(n) \forall n \geq n_0$.

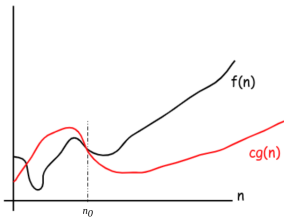


Figura 1.2: Andamento funzione Omega grande

La $f(n)$ evidenziata è solamente una delle molteplici funzioni limitate inferiormente da $g(n)$.

1.1.3 Θ grande

Pone delle delimitazioni strette superiori e inferiori alla complessità. Siano $f(n)$ e $g(n)$ due funzioni non negative, $f(n) = \Theta(g(n))$ se esistono tre costanti positive c_1, c_2 e n_0 tali che $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0$.

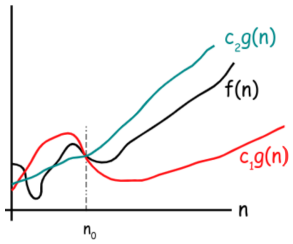


Figura 1.3: Andamento funzione Theta grandhe

Dunque da un certo n_0 in poi la funzione $f(n)$ è racchiusa fra le due funzioni $c_1 g(n)$ e $c_2 g(n)$.

1.1.4 o piccolo

Pone un limite superiore non asintoticamente stretto alla complessità. Siano $f(n)$ e $g(n)$ due funzioni non negative, $f(n) = o(g(n))$ se:

$$\forall c > 0,\ \exists n_0 > 0\ :\ 0 \leq f(n) < cg(n) \quad \forall n \geq n_0$$

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

1.1.5 ω piccolo

Pone un limite inferiore non asintoticamente stretto alla complessità. Siano $f(n)$ e $g(n)$ due funzioni non negative, $f(n) = \Omega(g(n))$ se:

$$\forall c > 0,\ \exists n_0 > 0\ :\ 0 \leq cg(n) < f(n) \quad \forall n \geq n_0$$

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

1.1.6 Proprietà asintotiche

Alcune regole vengono applicate dalla matematica:

- transitività

$$f(n) = \Theta(g(n)) \land g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \land g(n) = O(h(n)) \implies f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \land g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$$
$$f(n) = o(g(n)) \land g(n) = o(h(n)) \implies f(n) = o(h(n))$$
$$f(n) = \omega(g(n)) \land g(n) = \omega(h(n)) \implies f(n) = \omega(h(n))$$

- riflessività

$$f(n) = \Theta(f(n)) \qquad f(n) = O(f(n)) \qquad f(n) = \Omega(f(n))$$

- simmetria

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

- dualità

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \qquad f(n) = o(g(n)) \iff g(n) = \omega(f(n))$$

1.2 Algoritmi ricorsivi

In alcune situazioni viene chiamato un algoritmo all'interno di sé stesso.

1.2.1 Serie di Fibonacci

La nota serie di numeri interi positivi che inizia per 1, 1, 2, 3, 5, ... è definita come:

$$F(n) = \begin{cases} F(n-1) + F(n-2), & n \geq 3 \\ 1, & n = 1 \lor n = 2 \end{cases}$$

In questo caso sarebbe comodo calcolare direttamente $F(n)$ e si può fare

$$F(n) = \frac{1}{\sqrt{5}} (\varphi^n - \hat{\varphi}^n)$$

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1,618 \qquad \hat{\varphi} = \frac{1 - \sqrt{5}}{2} \approx -0,618$$

il problema è che i risultati ottenuti sono valori irrazionali: ingestibili dagli elaboratori. Per questo motivo si sfrutta un algoritmo ricorsivo.

Algoritmo 1**:** Funzione ricorsiva di Fibonacci

```
1. begin
2.     algoritmo fibonacci (intero n) → intero
3.     if n≤2 then
4.         output 1
5.     else
6.         output (fibonacci (n - 1) + fibonacci (n - 2))
7. end
```

Così facendo si opera esclusivamente su interi.

1.2.2 Induzione

Viene data per assodata un'affermazione e ne si verifica una sua estensione. Così se quest'ultima è vera si può accertare anche la precedente.

> **Teorema** (induzione)**.** Sia $a(n) \forall n \geq 0$ un'affermazione da dimostrare, si procede così:
>
> i. passo zero — si verifica la validità per un basico $a(k)$;
>
> ii. passo induttivo — si pone $a(n-1)$ vera e la si prova, si ha che anche $a(n)$ lo è.

1.2.3 Ricerca

Esistono molteplici algoritmi di ricerca, più o meno efficienti a seconda della costruzione. Nel caso di sequenze non ordinate si procede in modo sequenziale:

Algoritmo 2**:** Ricerca sequenziale di un elemento in un array

```
1.  begin
2.      ricercaSequenziale (array A, elemento x)
3.      for i ← to length[A] do
4.          if A[i] = x then
5.              output trovato
        output non trovato
6.  end
```

Invece, se il campione è ordinato si può velocizzare l'operazione:

Algoritmo 3**:** Ricerca binaria di un elemento in un array

```
1.  begin
2.      ricercaBinaria (array A, elemento x)
3.      n ← length[A]
4.      if n = 0 then
5.          output non trovato
6.      i ← [n/2]
7.      if A[i] = x then
8.          output non trovato
9.      else if A[i] > x then
10.         ricercaBinaria(A[1; i - 1], x)
11.     else
12.         ricercaBinaria(A[i + 1; n], x)
13. end
```

Tale algoritmo riportato antecedentemente è descritto mediante la seguente equazione:

$$T(n) = \begin{cases} c + T\left(\frac{n-1}{2}\right), & n > 1 \\ 1, & n = 1 \end{cases}$$

1.2.4 Relazioni di ricorrenza

> **Definizione** (relazione di ricorrenza)**.** Equazione che descrive una funzione in termini del suo valore con input più piccoli.

1.2.4.1 Metodo iterativo

Consiste nello srotolare la ritorsione fino ad ottenere una sommatoria dipendente da n.

Esempio (metodo iterativo)**.** Data una funzione

$$T(n) = \begin{cases} 1 + T\left(\frac{n}{2}\right), & n > 1 \\ 1, & n = 1 \end{cases}$$

si procede ad applicare il metodo iterativo

$$\begin{aligned} T(n) &= 1 + T\left(\frac{n}{2}\right) \\ &= 1 + 1 + T\left(\frac{n}{4}\right) \\ &= 2 + 1 + T\left(\frac{n}{8}\right) \\ &= 3 + 1 + T\left(\frac{n}{16}\right) \\ &\vdots \\ &= k + T\left(\frac{n}{2^k}\right) \end{aligned}$$

Si srotola la ricorsione fino a $\frac{n}{2^k} = 1$ ottenendo:

$$2^k = n \implies k = \log_2 n \qquad T(n) = \log_2(n) + 1 = O(\log_2 n)$$

Esistono delle uguaglianze per quanto riguarda le serie:

$$\sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}$$

$$\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1 = \frac{n \cdot (n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j = \frac{n \cdot (n-1)}{2} - 1 = \frac{n^2}{2} - \frac{n}{2}$$

1.2.4.1.1 Alberi di ricorsione

Vengono usati in alternativa al metodo iterativo.

Definizione (alberi di ricorsione)**.** Albero in cui ogni nodo rappresenta il costo di un sotto-problema da qualche parte nell'insieme delle chiamate ricorsive.

Per risolvere la ricorrenza occorre rispondere a queste domande:

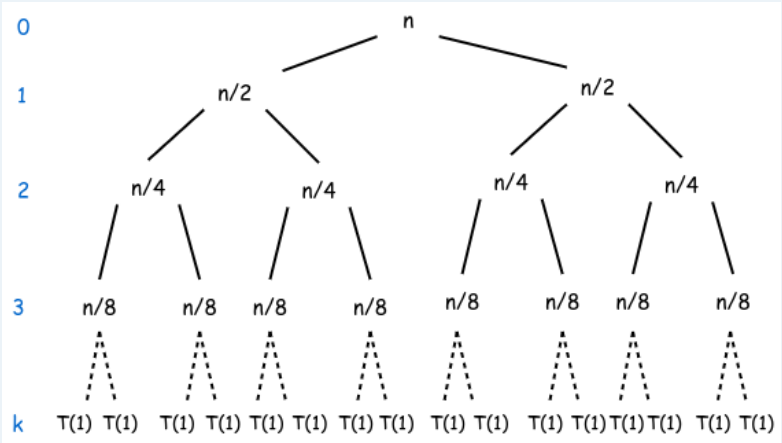
1. quanti nodi al livello i?
2. qual è il costo di un nodo al livello i?
3. qual è il costo complessivo dei nodi al livello i?
4. qual è il numero di livelli?
5. qual è il costo complessivo di $T(n)$?

Il costo complessivo di un qualunque livello i è sempre n.

Esempio (alberi di ricorsione)**.** Considerando la ricorrenza

$$T(n) = \begin{cases} n + 2T\left(\frac{n}{2}\right), & n > 1 \\ 1, & n = 1 \end{cases}$$

viene costruito il rispettivo albero.



Per rispondere alle cinque domande:

1. 2^i
2. $\frac{n}{2^i}$
3. $2^i \cdot \frac{n}{2^i} = n$
4. $k + 1$
5. $(k + 1) \cdot n$

Dove, al livello $T(k)$, si ha $T\left(\frac{n}{2^k}\right) = 1$, quindi: $k = \log_2 n$.

Si nota che, seppur spezzetato in più parti, il problema ha sempre dimensione n.

1.2.4.2 Metodo di sostituzione

Si applica quando si ha l'intuizione di quale funzione asintotica rappresenta la soluzione della ricorrenza.

Esempio (metodo di sostituzione)**.** Consideriamo di nuovo

$$T(n) = \begin{cases} n + T\left(\frac{n}{2}\right), & n > 1 \\ 1, & n = 1 \end{cases}$$

e dimostriamo, applicando il metodo della sostituizione, che

$$T(n) = O(n)$$

Per $n = 2$

$$n + T\left(\frac{n}{2}\right) = O(n) \quad \rightarrow \quad n + T\left(\frac{n}{2}\right) \leq cn$$

$$2 + T\left(\frac{2}{2}\right) \leq 2c \quad \rightarrow \quad 2 + T(1) \leq 2c$$

$$2 + 1 \leq 2c \ : \ c = 2 \quad \rightarrow \quad 3 \leq 4$$

1.2.4.3 Metodo dell'esperto

Dato un problema di dimensione n, lo si generalizza dividendolo in a sottoproblemi, chiaramente con $a \geq 1$.

$$a = \frac{n}{b} \ : \ b > 1$$

Il processo si esegue in tre parti:

i. dividere il problema;

ii. risolvere i sottoproblemi ricorisivamente;

iii. ricombinare le soluzioni.

Sia allora $f(n)$ il tempo per scindere e ricombinare istanze di dimensione n, la relazione è data da:

$$\boxed{T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n), & n > 1 \\ 1, & n = 1 \end{cases}}$$

Si hanno tre casistiche con rispettive soluzioni:

a. caso 1

$$\boxed{T(n) = \Theta\left(n^{\log_b a}\right), \text{ se } f(n) = O\left(n^{\log_b a - \epsilon}\right) \ \forall \epsilon}$$

b. caso 2

$$\boxed{T(n) = \Theta\left(n^{\log_b a} \cdot \log_b n\right), \text{ se } f(n) = \Theta\left(n^{\log_b a}\right)}$$

c. caso 3

$$\boxed{T(n) = \Theta\left(f(n)\right) \ : \ af\left(\frac{n}{b}\right) \leq cf(n) \text{ con } c < 1, \text{ se } f(n) = \Omega\left(n^{\log_b a + \epsilon}\right) \ \forall \epsilon}$$

Per risolvere l'esercizio si pone $f(n) = \Theta\left(n^{\log_b a}\right)$, per poi aggiungere o rimuovere qualcosa all'esponente in caso di necessità.

Sezione 2

ALGORITMI DI ORDINAMENTO

Data una sequenza disordinata, l'obiettivo è quello di disporla in maniera crescente col minor costo possibile. La scelta del miglior algoritmo dipende da:

- numero di elementi;
- quanto gli elementi siano già ordinati;
- metodo di accesso al dispositivo di memoria.

Nell'analisi dei costi vengono evidenziati il caso migliore e quello peggiore, risulta più decisivo prendere in considerazione quest'ultimo poiché:

a. rappresenta una limitazione;
b. potrebbe essere al quanto ricorrente;
c. spesso il caso medio tende verso il peggiore.

Tabella 2.1: Complessità degli algoritmi di ordinamento

ALGORITMO	BEST	AVG	WORST
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
merge sort	$O\left(n \cdot \log_2(n)\right)$	$O\left(n \cdot \log_2(n)\right)$	$O\left(n \cdot \log_2(n)\right)$
quick sort	$O\left(n \cdot \log_2(n)\right)$	$O\left(n \cdot \log_2(n)\right)$	$O(n^2)$
heap sort	$O\left(n \cdot \log_2(n)\right)$	$O\left(n \cdot \log_2(n)\right)$	$O\left(n \cdot \log_2(n)\right)$
counting sort	$O(k+n)$	$O(k+n)$	$O(k+n)$

Un sito dove vengono descritti i passaggi in dettaglio è visualgo[1].

2.1 Algoritmi incrementali

Si assume la posizione $A[1 \dots j-1]$ del vettore ordinata e si inserisce il successivo elemento $A[j]$ nella giusta posizione.

[1] Hal11.

2.1.1 Insertion Sort

Si scorre la sequenza infilando l'elemento nella porzione ordinata a sinistra dell'iteratore.

Algoritmo 4: Pseudocodice Insertion Sort

```
begin
    for j ← 2 to length[A] do
        key ← A[j]
        i ← j - 1
        while i > 0 e A[i] > key do
            A[i + 1] ← A[i]
            i ← i - 1
            A[i + 1] ← key
end
```

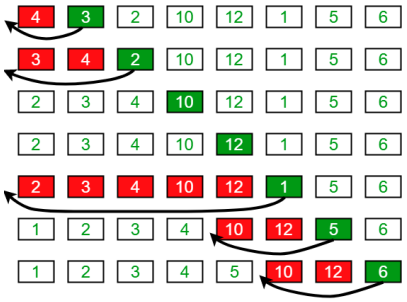


Figura 2.1: Esempio Insertion Sort

2.1.2 Selection Sort

Si scorre la sequenza spostando l'elemento minore a sinistra per poi bloccarlo.

Algoritmo 5: Pseudocodice Selection Sort

```
begin
    for i ← 1 to length[A] - 1 do
        min ← 1
        for j ← i + 1 to length[A] do
            if A[j] < A[min] then
                min ← j
        scambia A[min] ↔ A[i]
end
```

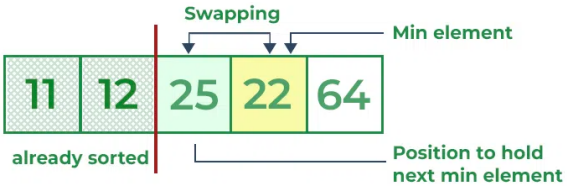


Figura 2.2: Esempio Selection Sort

2.1.3 Bubble Sort

Scorre la sequenza scambiando man mano il massimo a destra che poi viene bloccato.

Algoritmo 6: Pseudocodice Bubble Sort

```
begin
    for j ← length[A] downto 2 do
        scambi ← 0
        for i ← 1 to j - 1 do
            if A[i] > A[i + 1] then
                scambia A[i] ↔ A[i + 1]
                scambi ↔ scambi + 1
        if scambi = 0 then
            output return
end
```

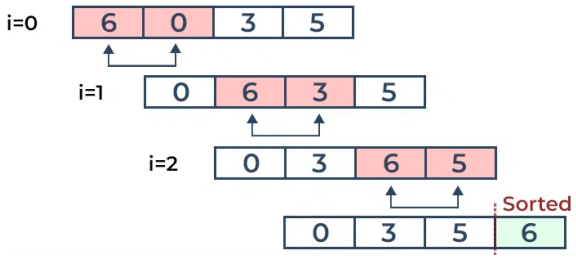


Figura 2.3: Esempio Bubble Sort

2.2 Algoritmi ricorsivi

All'interno dell'algoritmo vi è una chiamata a sé stesso.

> **Definizione** (dividi-et-impera)**.** Si suddivide il problema in un n sottoproblemi che, una volta risolti ricorsivamente e ricombinati, formano la soluzione del problema.

2.2.1 Merge Sort

Si divide a metà ricorsivamente sino ad ottenere delle coppie da riassemblare in ordinatamente.

Algoritmo 7: Pseudocodice Merge Sort

```
begin
    Merge (A, left, mid, right)
    m1 ← mid - left + 1
    m2 ← right - mid
    B[1, . . . , m1] ← A[left..mid]
    C[1, . . . , m2] ← A[mid + 1..right]
    i, j ← 1, k ← left
    while i ≤ m1 and j ≤ m2 do
        if B[i] ≤ C[j] then
            A[k] ← B[i]
            i ← i + 1
        else
            A[k] ← C[j]
            j ← j + 1
        k ← k + 1
    if i ≤ m1 then
        A[k, . . . , right] ← B[i..m1]
    else
        A[k, . . . , right] ← C[j..m2]
    MergeSort (A, left, right)
    if left < right then
        mid = ⌊(left + right) / 2⌋
        MergeSort(A, left, right)
        MergeSort(A, mid + 1, right)
        Merge(A, left, mid, right)
end
```

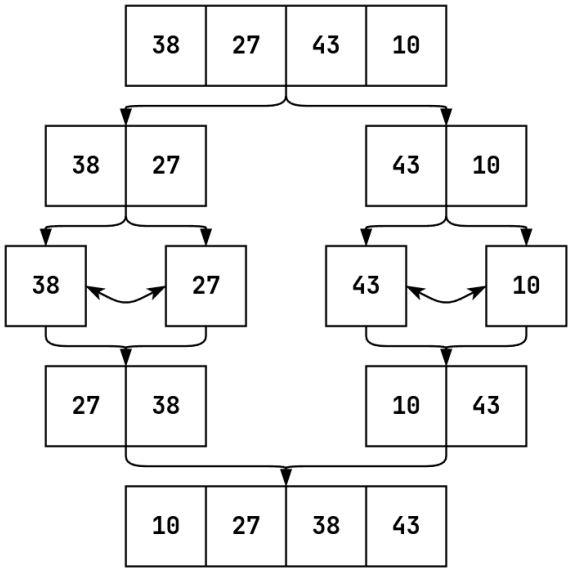


Figura 2.4: Esempio Merge Sort

2.2.2 Quick Sort

Applica sempre il concetto di dividi-et-impera:

a. *dividi*, si divide l'array scegliendo un elemento pivot tale che

$$A[p..r] \begin{cases} \nearrow A[p..q-1] \\ \searrow A[q+1..r] \end{cases}$$

b. *impera*, ordina le due metà per poi rinserire il pivot al centro

Algoritmo 8: Pseudocodice Quick Sort

```
1.  begin
2.      Partition(A, p, r)
3.      x ← A[r]
4.      i ← p - 1
5.      for j ← p to r - 1 do
6.          if A[j] ≤ x then
7.              i ← i + 1
8.              A[i] ↔ A[j]
9.      scambia A[i + 1] ← A[r]
10.     output return i + 1
11.     QuickSort(A, p, r)
12.     if p < r then
13.         q ← Partition(A, p, r)
14.         QuickSort(A, p, q - 1)
15.         QuickSort(A, p + 1, r)
16. end
```

2.2.3 Heap Sort

L'array viene considerato come un albero binario, si crea un max, o min, heap per poi ordinare tramite max, o min, extract. Il numero di nodi è dato da:

d: profondità $\qquad$ h: altezza

$$\#\text{nodi} = 2^d \qquad \#\text{nodi} = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Il numero di foglie è sempre la metà, o la metà+1 nel caso in cui dispari, dei nodi totali.

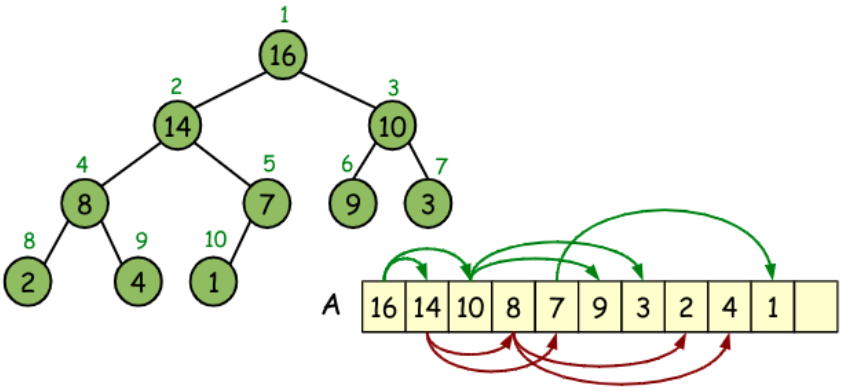


Figura 2.5: Albero Heap Sort

L'array ha due attributi:

- length[A], numero di elementi;
- heapSize[A], numero di elementi dell'heap memorizzati.

Algoritmo 9**:** Pseudocodice Heap Sort

```
1.  begin
2.      BuildMaxHeap(A)
3.      heapSize[A] ← length[A]
4.      for i ← ⌊length[A]/2⌋ downto 1 do
5.          MaxHeapify(A, i)
6.      MaxHeapify(A, i)
7.      l ← left[i]
8.      r ← right[i]
9.      if l ≤ heapSize[A] and A[l] > A[i] then
10.         max ← l
11.     else
12.         max ← i
13.     if r ≤ heapSize[A] and A[r ] > A[max] then
14.         max ← r
15.     if max 6 = i then
16.         A[i] ↔ A[max]
17.         MaxHeapify(A, max)
18.     HeapSort(A)
19.     BuildMaxHeap(A)
20.     for i ← length[A] downto 2 do
21.         A[1] ↔ A[i]
22.         heapSize[A] ↔ heapSize[A] - 1
23.         MaxHeapify(A, 1)
24. end
```

2.2.3.1 Max-heap e min-heap

Esistono due tipi di heap binari:

- max-heap
  ogni nodo i diverso dalla radice $A[1]$ è tale che $A[\text{parent}[i]] \geq A[i]$. Dunque l'array è ordinato in modo decrescente e l'elemento più grande viene memorizzato nella radice.
- min-heap
  ogni nodo i diverso dalla radice $A[1]$ è tale che $A[\text{parent}[i]] \leq A[i]$. Dunque l'array è ordinato in modo crescente e l'elemento più piccolo viene memorizzato nella radice.

Dopo avere l'albero in max-heap si può procedere ad estrarre il massimo.

2.2.3.1.1 Priority queues

Come gli heap, esistono due tipi di code: di max-priorità e min-priorità. Esse supportano le seguenti operazioni:

Tabella 2.2: Operazioni delle priority queues

MAX-PRIORITY QUEUE	MIN-PRIORITY QUEUE
`Insert(A, x)`	`Insert(A, x)`
`Maximum(A)`	`Minimum(A)`
`Extract-Max(A)`	`Extract-Min(A)`
`Increase-Key(A, x, k)`	`Decrease-Key(A, x, k)`

L'estrazione del massimo consiste, dato un albero max-heap, nel rimuovere la radice per poi sostituirla con l'elemento dall'indice più grande.

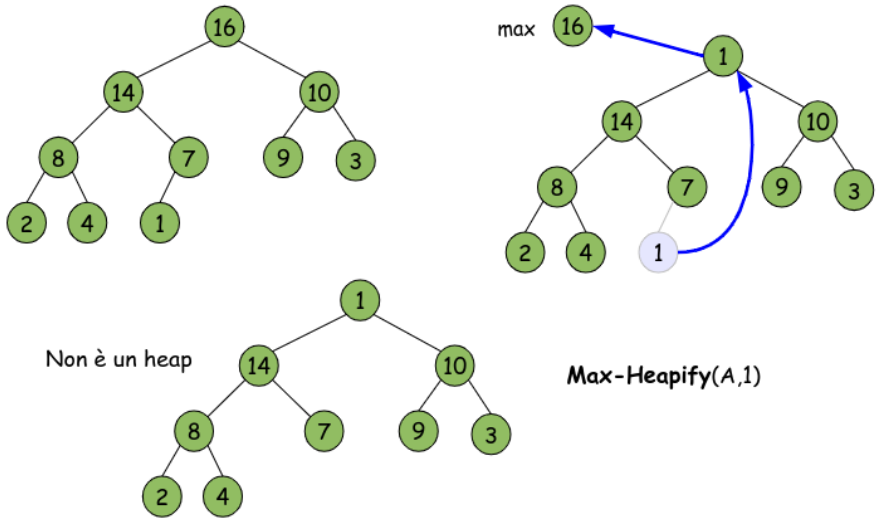


Figura 2.6: Esempio operazione di max-heapify

Nel far riscendere l'elemento sostituito alla radice si predilige lo switch-down con l'elemento maggiore.

2.3 Algoritmi lineari

Anziché basarsi solamente su confronti fra elementi, si vanno ad escludere soluzioni in base ad altri fattori.

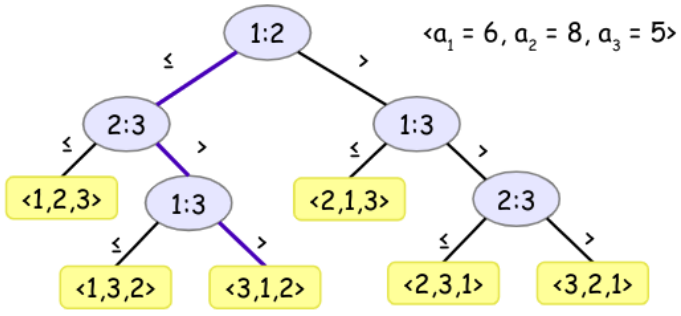


Figura 2.7: Ordinamento tramite albero di decisione

2.3.1 Counting Sort

Si hanno tre vettori: input (*I*), count (*C*) ed output (*O*). In *C* vanno le occorrenze dei valori di *I* posti come indici; poi si effettua la somma a carico dal primo elemento; infine si hanno gli elementi di *I* ordinati in posizione data da *C* in *O*.

Algoritmo 10: Pseudocodice Counting Sort

```
begin
    CountingSort(A, B, k)
    for i ← 0 to k do
        C[i] ← 0
    for j ← 1 to length[A] do
        C[A[j]] ← C[A[j]] + 1
    for i ← 1 to k do
        C[i] ← C[i] + C[i - 1]
    for j ← length[A] downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
end
```

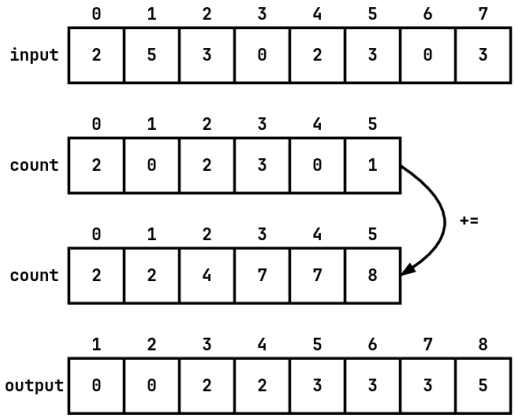


Figura 2.8: Passaggi Counting Sort

Sezione 3

STRUTTURE DATI

I dati vengono organizzati in modo da supportare le operazioni in maniera efficiente.

> **Definizione** (dati)**.** Valori effettivi sommati all'insieme delle operazioni che consentono di manipolarli.

3.1 Strutture dati astratte

Viene generalizzato il concetto di struttura dati avendo un insieme di valori da manipolare tramite operazioni.

$$\text{SD} = \begin{cases} S = "A[1, \ldots, n]" \text{ vettore} \\ \text{Read}(i) = "A[i]" \\ \text{Size}(A) = "n" \\ \text{Modify}(i, x) = "A[i] = x" \end{cases} \qquad \text{SDA} = \begin{cases} \text{insieme } S \text{ di numeri} \\ + \\ \text{Read, Size, Modify} \end{cases}$$

3.1.1 Pila

Gli elementi vengono gestiti secondo la politica LIFO (*Last In First Out*). Le operazioni supportate sono:

- *isEmpty()* $\rightarrow$ *boolean*, restituisce true se la sequenza è vuota e false altrimenti;
- *push(elem e)*, aggiunge *e* come ultimo elemento;
- *pop()* $\rightarrow$ *elem*, toglie l'ultimo elemento e lo restituisce;
- *top()* $\rightarrow$ *elem*, restituisce l'ultimo elemento senza eliminarlo.

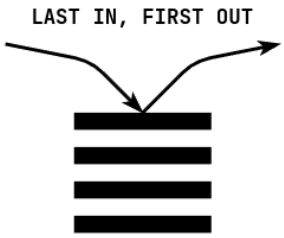


Figura 3.1: Rappresentazione pila

3.1.2 Coda

Gli elementi vengono gestiti secondo la politica FIFO (*First In First Out*). Le operazioni supportate sono:

- *isEmpty()* → *boolean*, restituisce true se la sequenza è vuota e false altrimenti;
- *enqueue(elem e)*, aggiunge *e* come ultimo elemento;
- *dequeue()* → *elem*, toglie il primo elemento e lo restituisce;
- *first()* → *elem*, restituisce il primo elemento senza eliminarlo.

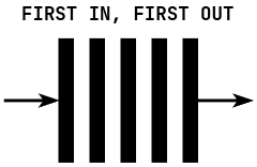


Figura 3.2: Rappresentazione coda

3.2 Strutture dati non astratte

Per catalogare i dati si possono seguire due approcci:

Tabella 3.1: Tecniche di rappresentazione dei dati

INDICIZZATE	COLLEGATE
array	record e puntatori
accesso diretto	accesso sequenziale
dimensione fissa	dimensione variabile

3.2.1 Liste

Ogni elemento trasporta il puntatore al successivo.

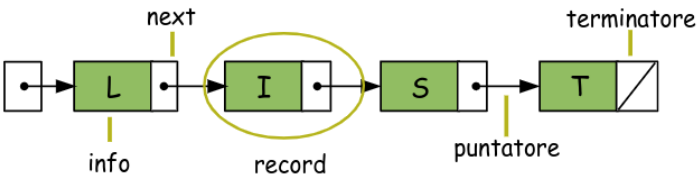


Figura 3.3: Struttura di una lista di elementi a puntatore singolo

In alcuni casi si ha anche il puntatore all'elemento precedente.

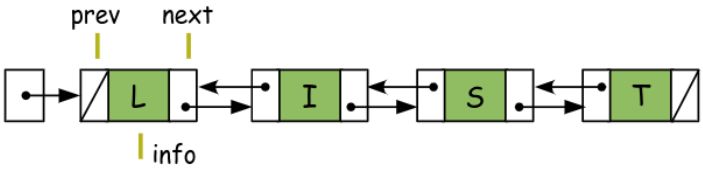


Figura 3.4: Struttura di una lista di elementi a doppio puntatore

3.2.2 Alberi

Ricercando un'organizzazione gerarchica, si ricorre agli alberi:

$$\underset{\text{albero}}{T} = (N,\ A) \quad \begin{cases} N = \text{insieme di nodi} \\ A \subseteq N \times N = \text{coppie di nodi, detti archi} \end{cases}$$

Dato che A è una coppia di nodi, l'albero è binario. Le caratteristiche sono:

- ogni nodo v, tranne la radice, ha un solo padre u. Nodi con lo stesso padre sono detti fratelli;
- un nodo u può avere tanti figli come zero, il numero di v è detto grado del nodo u;
- un nodo senza figli è detto foglia, mentre gli altri, esclusa la radice, interni;
- la profondità, o livello, di un nodo è dato dal numero di archi che lo separano dalla radice;
- l'altezza dell'albero è la massima profondità.

3.2.2.1 Visite di alberi

Analizziamo algoritmi che consentono l'accesso sistematico ai nodi e agli archi degli alberi.

3.2.2.1.1 Algoritmo di visita generica

Visita il nodo r e tutti i suoi discendenti.

Algoritmo 11**:** Pseudocodice visita generica

```
1. begin
2.     visitaGenerica (nodo r)
3.     S ← r
4.     while S ≠ ∅ do
5.         estrai un nodo u da S
6.         visita il nodo u
7.         S ← S ∪ (figli di u)
8. end
```

3.2.2.1.2 Algoritmo di visita in profondità

L'algoritmo DFS (*Depth First Search*) parte da r e procede visitando i nodi di figlio in figlio fino a raggiungere una foglia, per poi tornare al primo antenato con figli non visitati e ripete il procedimento.

Algoritmo 12**:** Pseudocodice visita in profondità iterativa

```
1. begin
2.     visitaDFS (nodo r)
3.     Pila S
4.     S.push(r)
5.     while not S.isEmpty() do
6.         u ← S.pop()
7.         if u ≠null then
8.             visita il nodo u
9.             S.push(figlio dx di u)
10.            S.push(figlio sx di u)
11. end
```

Algoritmo 13**:** Pseudocodice visita in profondità ricorsiva

```
1. begin
2.     visitaRicorsivaDFS (nodo r)
3.     if r ≠null then
4.         visita il nodo r
5.         visitaRicorsivaDFS(figlio sx di r)
6.         visitaRicorsivaDFS(figlio dx di r)
7. end
```

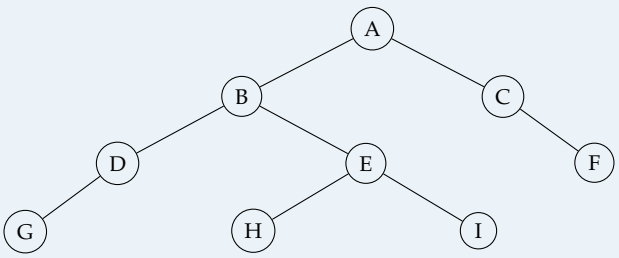
Esistono tre variazioni dello stesso modus-operandi:

i. **visita in pre-ordine**, prima la radice poi i figli sx e dx;

ii. **visita simmetrica**, prima il figlio sx, la radice, poi il figlio dx;

iii. **visita in post-ordine**, prima i figli sx e dx, e poi la radice;

Esempio (visite di alberi)**.** Dato il seguente albero



ecco le tre possibili visite

visita in pre-ordine:	A, B, D, G, E, H, I, C, F
visita simmetrica:	G, D, B, H, E, I, A, C, F
visita in post-ordine:	G, D, H, I, E, B, F, C, A

Algoritmo 14**:** Pseudocodice visita simmetrica

```
1. begin
2.     visitaSimmetrica (nodo r)
3.     if r ≠null then
4.         visitaRicorsivaDFS(figlio sx di r)
5.         visita il nodo r
6.         visitaRicorsivaDFS(figlio dx di r)
7. end
```

Algoritmo 15**:** Pseudocodice visita in post-ordine

```
1. begin
2.     visitaPostOrdine (nodo r)
3.     if r ≠null then
4.         visitaRicorsivaDFS(figlio sx di r)
5.         visitaRicorsivaDFS(figlio dx di r)
6.         visita il nodo r
7. end
```

3.2.2.1.3 Algoritmo di visita in ampiezza

L'algoritmo BFS (*Breadth First Search*) parte da r e procede visitando i nodi per livelli successivi. Un nodo al livello i può esser visitato solo dopo tutti i nodi al livello $i-1$.

Algoritmo 16**:** Pseudocodice visita in ampiezza

```
1.  begin
2.      visitaBFS (nodo r)
3.      Coda C
4.      C.enqueue(r)
5.      while not C.isEmpty() do
6.          u ← C.dequeue()
7.          if u ≠null then
8.              visita il nodo u
9.              C.enqueue(figlio sx di u)
10.             C.enqueue(figlio dx di u)
11. end
```

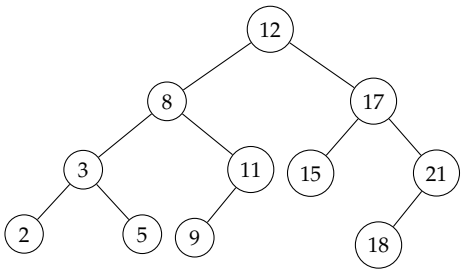
3.2.2.2 Alberi BST

Un *Binary Search Tree* è dato ordinato per confronto: nel sottoalbero sinistro gli elementi minori della radice e nel destro i maggiori, e così per ogni nodo. Dato un nodo x, allora:

- se y è un nodo del sottoalbero sx di x si ha che key$[y] \leq$ key$[x]$;
- se y è un nodo del sottoalbero dx di x si ha che key$[y] \geq$ key$[x]$.

Tabella 3.2: Caratterizzazione nodi di un albero BST

CAMPO	FUNZIONE
key	contenuto
left	puntatore al figlio sinistro
right	puntatore al figlio destro
parent	puntatore al padre



L'inserimento di un valore uguale alla radice va posto sul sottoalbero sinistro.

3.2.2.2.1 Operazioni sugli alberi BST

Il vantaggio di ordinare tali alberi si ha con le seguenti operazioni:

- *search*: ricerca facilitata da confronti, numero di confronti al più pari all'altezza dell'albero;
- *insert*: ricerca il valore da inserire e poi lo aggiunge come figlio del valore raggiunto;
- *delete*: se l'elemento eliminato ha due figli viene sostituito con il successore;
- *minimum*: seguendo il puntatore left si arriva all'elemento minimo;
- *maximum*: seguendo il puntatore right si arriva all'elemento massimo;
- *predecessor*: prossimo elemento dell'array ordinato;
- *successor*: precedente elemento dell'array ordinato.

Dunque tutte le operazioni possono essere eseguite su alberi con altezza h in tempo $O(h)$, solo se questi sono bilanciati.

3.2.2.3 Alberi AVL

Per gli array ordinati si mantiene il bilanciamento effettuando un inserimento binario.

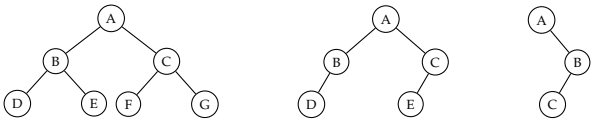
Definizione (fattore bilanciamento)**.** Differenza fra l'altezza del suo sottoalbero sinistro e quella del suo sottoalbero destro.

$$\beta(v) = h\,[\text{left}[v]] - h\,[\text{right}[v]]$$

Definizione (bilanciamento in altezza)**.** Un albero è bilanciato in altezza se, $\forall$ nodo v, vale: $|\beta| \leq 1$.

Tabella 3.3: Tipologie di alberi

COMPLETI	BILANCIATI	DEGENERATI
$\forall v\ \beta = 0$	$\forall v\ \lvert\beta\rvert \leq 1$	nodo $v\ \beta = h$



3.2.2.4 Altezza di alberi AVL

L'altezza h di un albero AVL T è logaritmica:

$$h = \Theta(\log_2 n), \quad N(h) \leq n \leq 2^{h+1} - 1$$

Il dislivello fra il sottoalbero di destra e quello di sinistra può essere al più uno, altrimenti l'albero si sbilancia. Il numero minimo di nodi è denotato da:

$$N(h) = \begin{cases} 1, & h = 0 \\ 2, & h = 1 \\ 1 + N(h-1) + N(h-2), & h \geq 2 \end{cases}$$

3.2.2.5 Rotazioni di alberi AVL

Si hanno quattro operazioni rotazionali che prendono diversa nomenclatura in base al sottoalbero u:

3.2.3 Tabelle

Si ha un database di elementi E_i costituiti da una chiave K_i e da un'informazione I_i. Le tre operazioni sono:

a. inserimento (elemento e, chiave k)

b. eliminazione (elemento e, chiave k)

c. ricerca (chiave k)

In particolare, la ricerca è caratterizzata da una lunghezza S_i che indica i tentativi effettuati prima di trovare K_i.

3.2.3.1 Tabelle ad indirizzamento diretto

Si utilizzano quando l'universo U delle chiavi è piccolo. All'insieme K viene corrisposto un'array T dove vengono allocate le chiavi utilizzate.

$$U = \{0,\ 1,\ \ldots,\ n-1\} \longrightarrow T = [0,\ 1,\ \ldots,\ n-1]$$

DirectAccessInsert($T,\ x$) { $T\,[key\,[x]] \leftarrow x$ }

DirectAccessDelete($T,\ x$) { $T\,[key\,[x]] \leftarrow$ NIL }

DirectAccessSearch($T,\ k$) { return $T\,[k]$ }

Gli inconveniente di queste tabelle sono:

i. dato che $n = |U|$, se l'insieme universo è grando l'implementazione si complica;

ii. lo spazio allocato resta costante indipendentemente dalle chiavi memorizzate;

Per questo secondo motivo si introduce il fattore di carico:

$$\alpha = \frac{m}{n} \begin{matrix} \nearrow \\ \searrow \end{matrix} \begin{matrix} m = |K| & \text{numero di chiavi utilizzate} \\ n = |U| & \text{dimensione tabella} \end{matrix}$$

Questo valore α è una percentuale e più sarà bassa, maggiore è lo spreco di memoria.

3.2.3.2 Tabelle hash

Vengono sfruttate le funzioni hash che associano la key dell'elemento all'indice dove si trova.

$$h:\ U \mapsto [0,\ 1,\ \ldots,\ m-1] \qquad m < |U|$$

Questa soluzione risparmia spazio in memoria, ma può riscontrare il problema delle collisioni. La probabilità che una chiave venga estratta è:

$$\sum_{h(k)=j} P(k) = \frac{1}{m},\ \text{per}\ j = 0,\ \ldots,\ m-1$$

3.2.3.2.1 Collisioni

Due chiavi K_1 e K_2 collidono quando corrispondono allo stesso indice:

$$h(K_1) = h(K_2)$$

Per risolvere tale problema si cerca di scegliere una funzione hash perfetta, ossia iniettiva:

$$\forall K_1,\ K_2 \in U \qquad K_1 \neq K_2 \implies h(K_1 \neq h(K_2))$$

Ciò è possibile solo se $m < |U|$. Nei casi in cui è matematicamente impossibile scegliere una funzione hash iniettiva si risolvono le collisioni in altri modi:

1. *liste di collisione*, la cella contiene un puntatore ad una lista;
2. *indirizzamento aperto*, si sfruttano le successive celle vuote.

3.2.3.2.2 Metodi di approssimazione

Per realizzare funzioni hash quanto più adatte si usano delle euristiche.

– *Metodo della divisione*
Viene associata la chiave al valore hash:

$$h(k) = k \mod m\ :\ m \neq 2^i\ \forall i \in \mathbb{N}$$

Si sceglie come m un numero lontano ad una potenza di due.

– *Metodo della moltiplicazione*
Anche in questo caso si associa la chiave al valore hash:

$$h(k) = \lfloor m \cdot (kA - \lfloor kA \rfloor) \rfloor$$

In questo modo m può essere potenza di due mentre A è una costante ed equivale a $\frac{\left(\sqrt{5}-1\right)}{2}$.

Il modulo restituisce il resto di una divisione. Avendo $\frac{\alpha}{\beta} \ : \ \beta \neq 0$ si può ricorrere alla seguente equazione:

$$\alpha \mod \beta \implies \alpha - (\beta \cdot q) = r \quad \begin{cases} q = \text{quoziente} \\ r = \text{resto} \end{cases}$$

3.2.3.2.3 Indirizzamento aperto

Per verificare la validità di un operazione si hanno varie tecniche di scansione:

– *scansione lineare*;

$$\boxed{h(k,\ i) = (h'(k) + i) \mod m \quad \begin{cases} h'(k) = \text{hash precalcolato} \\ i = \text{valore lineare} \end{cases}}$$

– *scansione quadratica*;

$$\boxed{h(k,\ i) = \left(h'(k) + c_1 i + c_2 i^2\right) \mod m : c_1,\ c_2 \neq 0}$$

– *hash doppio*.

$$\boxed{h(k,\ i) = (h_1(k) + i h_2(k)) \mod m}$$

Sezione 4

GRAFI

Definizione (grafo)**.** Si ha una coppia del tipo

$$G = (V, E) \begin{cases} V\text{: insieme finito non vuoto di nodi} \\ E \subseteq V \times V\text{: insieme di coppie di nodi, dette archi} \end{cases}$$

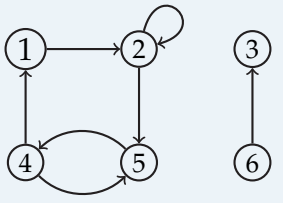
Definizione (grafo orientato)**.** L'insieme degli archi E è formato da coppie ordinate. Sono ammessi cappi.

Definizione (cappio)**.** Arco che esce ed entra dallo stesso nodo.

Definizione (grafo non orientato)**.** L'insieme degli archi E è formato da coppie non ordinate. Non sono ammessi cappi.

Esempio (grafo orientato)**.**

$$G = (V, E) \begin{cases} V = \{1, 2, 3, 4, 5, 6\} \\ E = \{(1, 2), (2, 2), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\} \end{cases}$$



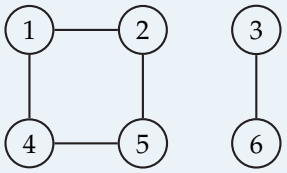
Matrice di adiacenza:

	1	**2**	**3**	**4**	**5**	**6**
1	0	1	0	0	0	0
2	0	1	0	0	1	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Per carpire eventuali cappi basta evidenziare la diagonale.

Esempio (grafo non orientato)**.**

$$G = (V, E) \begin{cases} V = \{1, 2, 3, 4, 5, 6\} \\ E = \{(1, 2), (4, 5), (2, 5), (4, 1), (6, 3)\} \end{cases}$$



Matrice di adiacenza:

	1	**2**	**3**	**4**	**5**	**6**
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	0	1
4	1	0	0	0	1	0
5	0	1	0	1	0	0
6	0	0	1	0	0	0

Non essendoci orientamento, la matrice è simmetrica. Non sono inoltre ammessi cappi, dunque la diagonale è nulla.

4.1 Caratteristiche dei grafi

Definizione (incidenza)**.** In un grafo non orientato, un arco (u, v) è incidente ai nodi u, v.

Definizione (grado)**.** Numero di archi che incidono sul nodo. Per i grafi orientati vi è la distinzione fra grado entrante ed uscente.

Nota bene (nodo isolato)**.** Un nodo si dice isolato se ha grado zero!

Definizione (adiacenza)**.** In un grafo, orientato o no, un arco (u, v) rende il nodo u adiancente al nodo v.

Dunque il nodo di partenza è adiacente a quello di arrivo, ciò indica che nei grafi non orientati la relazione di adiacenza è simmetrica.

Definizione (cammino)**.** Una sequenza $\langle x_0, x_1, \ldots, x_k \rangle : u = x_0, v = x_k$ è un cammino di lunghezza k fra i nodi u e v.

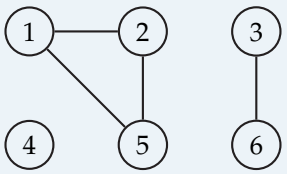
Definizione (ciclo)**.** Un cammino forma un ciclo se $x_0 = x_k$.

Se invece non viene mai ripercorso lo stesso nodo in tutto il cammino, quest'ultimo sarà detto semplice.

Se esiste un cammino p fra u e v, si dice che v è raggiungibile da u attraverso p e si indica con: $u \overset{p}{\rightsquigarrow} v$.

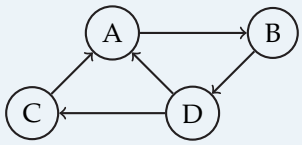
Definizione (grafo connesso)**.** Un grafo non orientato è connesso se ogni coppia è collegata da un cammino.

Esempio (componenti connesse)**.** Le componenti connesse del seguente grafo sono: {1, 2, 5}, {4} e {3, 6}.

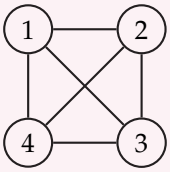


Definizione (grafo fortemente connesso)**.** Un grafo orientato è fortemente connesso se due vertici qualsiasi sono raggiungibili l'un l'altro.

Esempio (componenti fortemente connesse)**.** Il seguente grafo orientato è fortemente connesso.



Definizione (clique)**.** Un grafo avente gli elementi della matrice, tranne necessariamente la diagonale, pari a uno.



Matrice di adiacenza:

	1	**2**	**3**	**4**
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Definizione (densità di un grafo)**.** Rapporto tra il numero di archi del grafo rispetto al numero di coppie di nodi.

grafo orientato

$$\Delta = \frac{|E|}{n \cdot (n-1)}$$

grafo non orientato

$$\Delta = \frac{2 \cdot |E|}{n \cdot (n-1)}$$

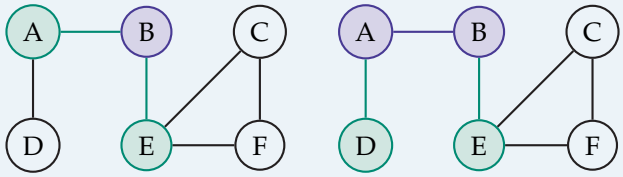
4.2 Visite di grafi

Analizziamo algoritmi che consentono l'accesso sistematico ai nodi e agli archi dei grafi.

4.2.1 Visita in ampiezza

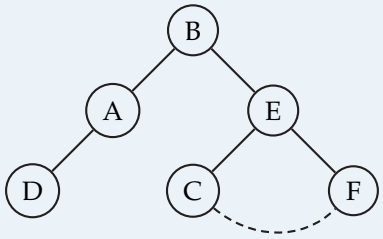
L'algoritmo BFS (*Breadth First Search*) ispezione direttamente ogni nodo adiacente a partire da uno iniziale fino a visitarli tutti, questi vengono poi salvati in una coda.

Esempio (BFS grafi)**.** Passaggi del BFS dal nodo sorgente B:



queue ← {B, A, E, D, C, F}

BFS spamming tree:



Il costo dell'algoritmo è $O(n + m)$.

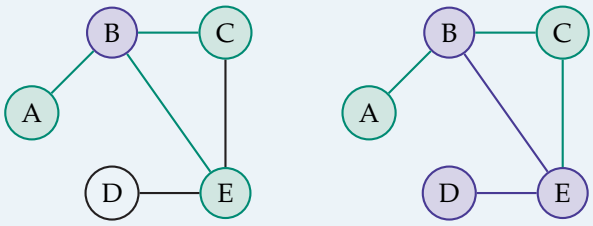
Algoritmo 17**:** Visita BFS di grafi

```
1.  begin
2.      BFS (G, s)
3.      for all nodo u ∈ V(G) − {s} do
4.          color[u] ← white
5.          d[u] ← ∞
6.          π[u] ← NIL
7.      color[s] ← GRAY
8.      d[u] ← 0
9.      π[s] ← NIL
10.     Q ← ∅
11.     ENQUEUE(Q, s)
12.     while Q ≠ 0 do
13.         u ← DEQUEUE(Q)
14.         for all v ∈ Adj[u] do
15.             if color[v] = WHITE then
16.                 color[v] ← GRAY
17.                 d[v] ← d[u] + 1
18.                 π[v] ← u
19.                 ENQUEUE(Q, v)
20.             color[u] ← BLACK
21. end
```

4.2.2 Visita in profondità

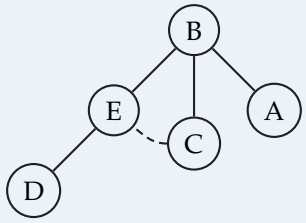
L'algoritmo DFS (*Depth First Search*) ispeziona ogni nodo scavando fino all'ultimo figlio, successivamente retrocede all'ultimo bivio e ricomincia. I nodi in attesa vengono salvati in una pila.

Esempio (DFS grafi)**.** Passaggi del DFS a partire dal nodo B:



stack ← {B, E, D, A, C}

DFS spamming tree:



Ad ogni nodo vengono associati due valori:

$$d[v] < f[v] : \begin{cases} d[v] = \text{tempo di scoperta} \\ f[v] = \text{tempo di completamento} \end{cases}$$

Il costo dell'algoritmo è $O(n + m)$.

Algoritmo 18**:** Visita DFS di grafi

```
1.  begin
2.      DFS(G)
3.      for all nodo u ∈ V[G] do
4.          color[u] ← WHITE
5.          π[u] ← NIL
6.      time ← 0
7.      for all nodo u ∈ V[G] do
8.          if color[u] = WHITE then
9.              DFSvisit(u)
10.     DFSvisit(u)
11.     color[u] ← GRAY
12.     time ← time + 1
13.     d[u] ← time
14.     for all nodo v ∈ Adj[u] do
15.         if color[v] = WHITE then
16.             π[v] ← u
17.             DFSvisit(v)
18.     color[u] ← BLACK
19.     time ← time + 1
20.     f[u] ← time
21. end
```

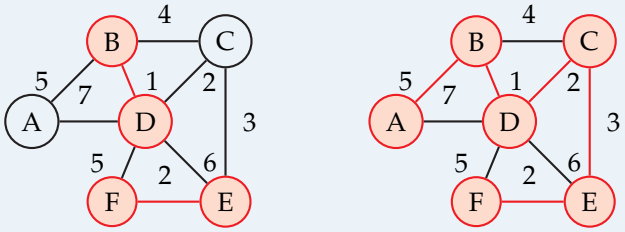
4.3 Minimo albero ricoprente

Sia $G = (V, E)$ un grafo non orientato, connesso e pesato. Si ricerca un insieme T di archi tale che ogni nodo sia raggiungibile da ogni altro. Se i pesi associati agli archi sono positivi non ci sono cicli e rappresenta un albero.

4.3.1 Algoritmo di Kruskal

Ordina gli archi per costo crescente.

Esempio (algoritmo di Kruskal)**.** Dato un grafo come segue, l'algoritmo di Kruskal agirà come vedete.



Nota bene (ambivalenza percorso)**.** Esistono dei punti critici, in questo caso C→D/E→F, dove risulta indifferente la strada scelta.

L'obiettivo è quello di costruire un MST. Si inizia dall'arco di costo inferiore per poi proseguire senza creare cicli. Non è necessario scegliere archi di nodi adiacenti!

Algoritmo 19**:** Pseudocodice algoritmo di Kruskal

```
1.  begin
2.      MST-Kruskal (G, w)
3.      T← ∅;
4.      for all nodo v ∈ V[G] do
5.          do Make-Set(v)
6.      ordina gli archi di E in senso non decrescente rispetto al peso w
7.      for all arco (u, v) ∈ E[G] preso in ordine non decrescente di peso do
8.          if Find-Set(u) ≠ Find-Set(v) then
9.              T ← T ∪ {(u, v)}
10.             Union-Set(u, v)
11.     output return T
12. end
```

- Find-Set(x) restituisce, per ogni nodo x, l'indice di W che lo contiene.
- Union-Set(u, v) costruisce l'unione di Find-Set(u) e Find-Set(v).

Il costo dell'algoritmo è pari a

$$O\left(m \cdot \log_2 n\right) : \begin{cases} m = |E| = \text{numero di archi} \\ n = |V| = \text{numero di nodi} \end{cases}$$

Un taglio di G è una partizione $\left(S, \overline{S} = V - S\right)$ dell'insieme V dei nodi. Un arco attraversa un taglio se un'estremità si trova in S e l'altra in $\overline{S}$.

Nota bene (arco leggero)**.** Arco che passa dal taglio di costo minimo.

Lemma (arco leggero in soluzione ottimale)**.** Sia

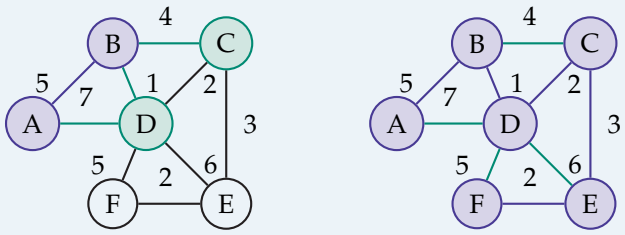
- $G = (V, E)$ un grafo non orientato, connesso e pesato;
- $\left(S, \overline{S}\right)$ un taglio di G;
- T un insieme di archi di una soluzione ottimale.

allora l'arco leggero (u, v) fa parte di T.

4.3.2 Algoritmo di Prim

Partendo da un nodo arbitrario u, si esaminano tutti gli altri del grafo tramite visita degli adiacenti. L'insieme Q dei nodi da visitare viene gestito tramite una coda di priorità.

Esempio (algoritmo di Prim)**.** Creare un MST, sfruttando l'algoritmo di Prim, a partire dal nodo A.



queue: {A, B, D, C, E, F}

L'importante è non selezionare archi che arrivano in nodi già visitati.

Algoritmo 20**:** Pseudocodice algoritmo di Prim

```
1.  begin
2.      MST-Prim (G, w, r)
3.      for all nodo v ∈ V[G] do
4.          key[v] = ∞
5.          π[v] = NIL
6.      key[r] ← 0
7.      Q ← V[G]
8.      while Q ≠ ∅ do
9.          u ← Extract-Min(Q)
10.         for all v ∈ Adj[u] do
11.             if v ∈ Q and w(u, v) < key[v] then
12.                 π[v] ← u
13.                 key[v] ← w(u, v)
14. end
```

Anche in questo caso, il costo dell'algoritmo è pari a $O\left(m \cdot \log_2 n\right)$.

4.4 Cammini minimi

Sia $G = (V,\ E)$ un grafo orientato e pesato secondo una funzione $w : E \to \mathbb{R}$, si può definire:

* *peso di un cammino* $p = \langle v_0,\ v_1,\ \ldots,\ v_n \rangle$ come la somma degli archi che lo compongono

$$w(p) = \sum_{i=1}^{n} w\left(v_{i-1},\ v_i\right)$$

* *peso di un cammino minimo* da u a v come

$$\delta\left(u,\ v\right) = \begin{cases} min\left\lfloor w\left(p\right) : u \overset{p}{\rightsquigarrow} v \right\rfloor, & \text{se esiste un cammino da } u \text{ a } v \\ \infty, & \text{altrimenti} \end{cases}$$

Teorema (sottostruttura ottima di un cammino minimo)**.** Avendo un grafo orientato $G = (V,\ E)$ e pesato secondo una funzione di peso $w : E \to \mathbb{R}$, sia inoltre $p = \langle v_0,\ v_1,\ \ldots,\ v_k \rangle$ il cammino minimo fra i nodi v_1 e v_k.

$$\forall 1 \leq i \leq j \leq k, \text{ il sottocammino } p_{ij} = \langle v_i,\ v_{i+1},\ \ldots,\ vj \rangle$$
$$\text{è un cammino minimo da } v_i \text{ a } v_j$$

Dimostrazione (sottostruttura ottima di un cammino minimo)**.** Scomponendo il cammino si ottiene che

$$w\left(p\right) = w\left(p_{1i}\right) + w\left(p_{ij}\right) + w\left(p_{jk}\right)$$

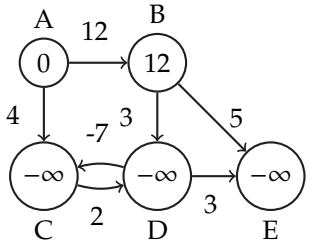
Assumendo che $w\left(p_{ij}\right)$ non sia ottimo, dunque $\exists\, p'_{ij} : w\left(p'_{ij}\right) < w\left(p_{ij}\right)$ standardora:

$$\underline{w\left(p'\right)} = w\left(p_{1i}\right) + w\left(p'_{ij}\right) + w\left(p_{jk}\right) < w\left(p_{1i}\right) + w\left(p_{ij}\right) + w\left(p_{jk}\right) = \underline{w\left(p\right)}$$

Ciò è assurdo poiché p è ottimo.

4.4.1 Archi di peso negativo

La funzione w può assegnare pesi negativi agli archi.



Tuttavia sarebbe possibile individuare cicli con peso negativo che rendono il peso del cammino minimo indefinito.

Nota bene (cicli nei cammini minimi)**.** Un cammino minimo non può contenere cicli! Dunque può avere al più $n - 1$ archi.

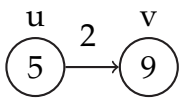
4.4.2 Rilassamento

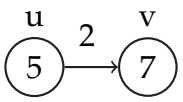
Il processo di rilassamento di un arco (u, v) consiste nel verificare, ed eventualmente migliorare, la stima del cammino minimo memorizzata in $d[u]$.

Algoritmo 21**:** Pseudocodice stima cammino minimo

```
1. begin
2.     InitializeSingleSource(G, s)
3.     for all v ∈ V[G] do
4.         d[u] ← ∞
5.         π[v] ← NIL
6.     d[s] ← 0
7. end
```

Rispetto alla stima iniziale, il peso effettivo del collegamento risulta spesso minore.





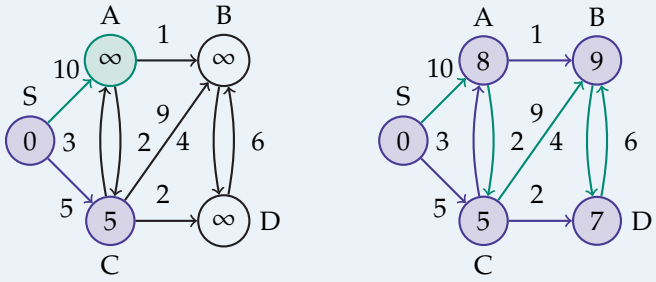
Algoritmo 22**:** Pseudocodice processo di rilassamento

```
1. begin
2.     Relax(u, v, w)
3.     if d[v] > d[u] + w(u, v) then
4.         d[v] ← d[u] + w(u, v)
5.         π[v] ← u
6. end
```

4.4.2.1 Algoritmo di Dijkstra

Il concetto alla base di questo algoritmo sta, dato un grafo orientato e pesato, nel selezionare il cammino minimo verso ogni nodo a partire da uno sorgente.

Esempio (applicazione Dijkstra)**.** Avendo un grafo orientato e pesato come quello della prima figura, si cercano i cammini minimi verso tutti i nodi a partire dal sorgente S.



La caratteristica sta nell'applicare il rilassamento in caso di percorsi di costo minore.

La complessità dell'algoritmo è pari ad $O\left(n^2\right)$.

Algoritmo 23**:** Pseudocodice algoritmo di Dijkstra

```
1.  begin
2.      Dijkstra(G, w, s)
3.      InitializeSingleSource(G, s)
4.      S ← 0;
5.      Q ← V[G]
6.      while Q ≠ ∅ do
7.          u ←Extract-Min(Q)
8.          S ← S ∪ {u}
9.          for all v ∈ Adj[u] do
10.             Relax(u, v, w)
11. end
```

Sezione 5

LABORATORIO

Ogni programma deve valutare i costi, specificamente tempo e memoria, oltre che l'effettivo funzionamento. Ai fini del corso si utilizza Java, per ogni dubbio è consigliato esaminare le rispettive API[1].

5.1 Classi e oggetti

Si programma ad oggetti, questi hanno dei metodi associati fra cui il costruttore che viene eseguito ogni volta che si instanzia un oggetto.

5.1.1 OOP

Esistono due tipologie di programmazione:

a. programmazione procedurale
   Il blocco principale può chiamare funzioni esterne ed interagire con i dispositivi di I/O.

b. Object Oriented Programming
   Si definiscono delle classi-stampo per istanziare oggetti con metodi associati.

5.1.1.1 Incapsulamento

Ogni classe dispone di variabili, o campi, ma queste sono mantenute private per evitarne la manipolazione inaspettata.

5.1.2 Business logic e Front-End

Ogni applicazione è divisa idealmente in due livelli:

i. core business logic
   Vera essenza del programma che tralascia però totalmente l'interazione con l'utente.

ii. front-end
   Gestisce il rapporto con l'utilizzatore tramite l'uso dei dispositivi di I/O.

5.1.3 Unit testing

Per controllare che il codice sia implementato correttamente si effettuano dei test. Nel nostro caso useremo il framework JUnit[2].

[1] Ora93.
[2] Tea23.

5.1.4 Uguaglianza di oggetti

Per valutare se due oggetti sono la stessa cosa è necessario effettuare un controllo sull'allocazione in memoria. Non basterà più il classico `A==B`, ma occorrerà effettuare `A.equals(B)`.

5.1.5 Ordinamento naturale tra oggetti

Gli elementi di una classe vengono ordinati secondo certi parametri scelti: ad esempio per le stringhe è lessicografico.

```
1 interface Comparable<T> {
2     public int compareTo (T o) { ... }
3 }
```

Codice 5.1: Interfaccia per comparazioni

In questo caso il metodo `compareTo(T o)` restituisce:

- < 0, se l'oggetto `this` precede `o`;
- > 0, se l'oggetto `this` segue `o`;
- 0, se `this.equals(o)`;

5.2 Polimorfismo

Definizione (variabile polimorfa)**.** Un campo che può riferire oggetti di classi diverse.

5.2.1 Interfacce

Dato che non si tratta di una classe, i metodi sono pubblici e non hanno implementazione poiché astratti.

```
1 interface I {
2     public type method();
3 }
```

Codice 5.2: Definizione interfaccia

Tabella 5.1: Differenze fra classi e interfacce

CLASSE	INTERFACCIA
estensione singola	implementazioni multiple
contiene campi e metodi	contiene metodi

5.2.1.1 Implements

Per utilizzare i metodi presenti nell'interfaccia nelle altri classi occorre implementarla:

```
1 public class C implements I { ... }
```

Codice 5.3: Implementazione di un'interfaccia

Una singola classe può includere molteplici interfacce all'evenienza:

```
1 public class C implements I, J, K { ... }
```

Codice 5.4: Implementazione di più interfacce

Se non vengono sfruttati tutti i metodi la classe va definita astratta.

5.2.1.1.1 Casting

D'accordo che viene implementata dalla classe, ma i metodi proprietari possono ricevere parametri del tipo dell'interfaccia stessa. In questi casi si effettua una conversione del tipo:

```
if (b instanceof C) {
    C c = (C) b; //cast
}
```

Codice 5.5: Controllo istanza e casting

5.3 Ereditarietà

Si crea una classe più specifica rispetto alle funzionalità già introdotte da un'altra:

```
public abstract A { ... } //superclasse
public class B extends A { ... } //sottoclasse
```

Codice 5.6: Ereditarietà fra classi

In generale tutte le classi esistenti sono figlie della superclasse `Object` e possono contenere nuovi metodi, anche sovrascritti, e campi.

5.3.1 Classi astratte

Vengono ridefiniti i metodi della superclasse nella sottoclasse. Per poter accedere alle variabili si sfrutta la keyword `super`.

```
public abstract A {
    A (a, b) { ... }
}

public class B extends A {
    B () {
        super(a, b); //deve essere la prima istruzione nel costruttore!
    }
}
```

Codice 5.7: Accedere a variabili `super`

5.4 Collections

L'array è una struttura dati basica, per questo si ricorre a metodi per raggruppare gli oggetti d'una stessa classe in un insieme dinamico; le operazioni comuni sono:

- search(primarykey) [return element]
- insert(element) [return boolean]
- delete(element) [return boolean]

Se l'insieme dinamico è ordinato, si aggiungono le seguenti operazioni:

- minimum() [return element]
- maximum() [return element]
- predecessor(element) [return element]
- successor(element) [return element]

Esistono più derivate del macrogruppo collection:

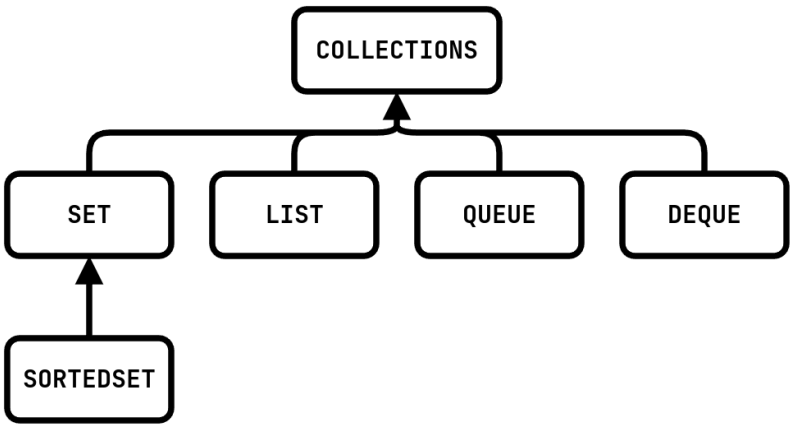


Figura 5.1: Gerarchia di collections

5.4.1 Liste

Nel pacchetto `java.util` è definita l'interfaccia `List<E>`, le classi che la implementano possono collezionare in una lista i propri oggetti.

```
1 public interface List<E> extends Collections<E> {
2     E get (int index);
3     int indexOf (Object o);
4     E set (int index, E element);
5     boolean add (E element);
6     void add (int index, E element);
7     E remove (int index);
8 }
```

Codice 5.8: Metodi interfaccia `List<E>`

È possibile scorrere tutti gli elementi della lista, ma ciò causa un errore se la si modifica nel mentre!

```
1 for (Class obj: this.list) { ... }
```

Codice 5.9: Ciclo `foreach` per scorrimento lista

Alcuni algoritmi polimorfi contenuti in `java.util.Collections` operano sulle liste:

- shuffle
- sort
- reverse

Il sorting viene effettuato per ordinamento naturale, a volte è però utile considerare altri parametri.

5.5 Eccezioni

Nei programmi possono verificarsi errori, ove possibile si cerca di gestire tali situazioni avvertendo l'utente: `throw new Exception("sample text");` Ogni eccezione è un oggetto di una classe apposita, a sua figlia della standard `Exception`.

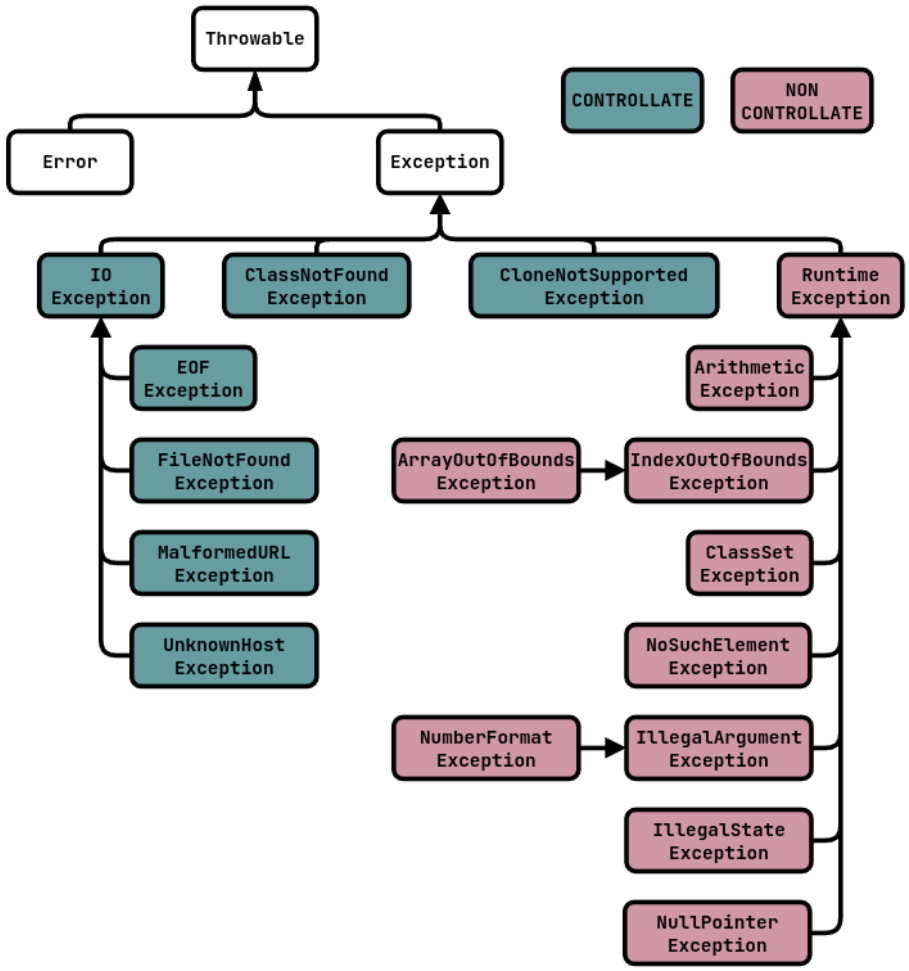


Figura 5.2: Gerarchia delle eccezioni Java

Come si vede, esiste una seconda classe padre, la `Error`, che come la `Exception` estende la `java.lang.Throwable`.

5.5.1 Try-catch-finally

Il blocco `try-catch-finally` serve a gestire le eccezioni.

```
1 try { ... }
2 catch (NullPointerException npe) { ... }
3 catch (IllegalArgumentException iae) { ... }
4 finally { ... }
```

Codice 5.10: Blocco `try-catch-finally`

Se un comando che si trova in `try` lancia un'eccezione, allora il tipo viene confrontato con quelle presenti nel `catch`.

5.6 Valutazione delle prestazioni

Per valutare un qualsiasi algoritmo A si crea un benchmark che segue questi passi:

i. inserimento di K input randomici diversi di dimensione crescente;

ii. esecuzione dell'algoritmo sugli input e registrazione, per ogni run, del tempo impiegato;

iii. aggregazione dei risultati e smistamento;

iv. visualizzazione grafica dei risultati.

Per ottenere dati quanto più veritieri è consigliato chiudere le applicazioni non interessate e disattivare la rete. La pulizia dei dati avviene tramite la rimozione dei picchi, ossia quei risultati esageratamente grandi o piccoli.

Sitografia

[Hal11] Steven Halim. VisuAlgo. Visitato in Autunno 2023. 2011. URL: www.visualgo.net.

[Ora93] Oracle. API Java 8. Visitato in Autunno 2023. 1993.
URL: https://docs.oracle.com/javase/8/docs/api.

[Tea23] The JUnit Team. JUnit 5. Visitato in Autunno 2023. 2023.
URL: https://junit.org/junit5.

Bibliografia

[Cor+98] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. 3rd ed. McGraw Hill, 1998. ISBN: 9780262033848.