

# Dispensa del Corso di Tecniche di Programmazione

Nome dell'Insegnante

Anno Accademico 2023-2024

## Sommario

Questa dispensa contiene un riassunto dei principali concetti trattati nel corso di Tecniche di programmazione.

## Indice

<b>1 Introduzione agli algoritmi</b>	<b>3</b>
1.1 Insertion Sort . . . . .	3
1.2 Binary Search . . . . .	6
<b>2 Notazioni asintotiche</b>	<b>8</b>
2.1 Introduzione . . . . .	8
2.1.1 Notazione Big O . . . . .	8
2.1.2 Notazione Omega . . . . .	8
2.1.3 Notazione Theta . . . . .	8
2.2 Definizioni formali . . . . .	9
2.3 Divide & Conquer . . . . .	11
2.4 Merge Sort . . . . .	11
2.5 2-Sum Problem . . . . .	15
<b>3 Strutture dati</b>	<b>16</b>
3.1 Array . . . . .	17
3.2 Lista . . . . .	17
3.3 Matrici . . . . .	17
3.4 Pila (Stack) . . . . .	17
3.5 Coda (Queue) . . . . .	18
3.6 Lista concatenata (Linked List) . . . . .	18
3.7 Tabella a indirizzamento diretto (Direct Address Table) . . . . .	19
3.8 Hash Table . . . . .	20
3.8.1 Collisioni . . . . .	20
<b>4 Algoritmi di ordinamento</b>	<b>20</b>
4.1 Binary Search Tree . . . . .	21
4.2 Interval Sorting . . . . .	23
4.3 Quick Sort . . . . .	24
4.4 Analisi Caso Atteso . . . . .	25
4.5 Merge Sort e Quick Sort . . . . .	27
4.6 Heap Sort . . . . .	27

<b>5</b>	<b>Programmazione Dinamica</b>	<b>32</b>
5.1	Un'approccio diretto: Fibonacci . . . . .	32
5.2	Interval Scheduling . . . . .	33
5.3	Weighted Interval Scheduling . . . . .	35
5.4	KnapSack Problem . . . . .	36
<b>6</b>	<b>Grafi</b>	<b>36</b>
6.1	Rappresentazione dei Grafi . . . . .	37
6.2	Visite in Grafi . . . . .	38
6.3	Shortest Path . . . . .	39
6.3.1	Algoritmo di Dijkstra . . . . .	40
6.4	Algoritmo di Bellman-Ford . . . . .	42
6.5	Minimum Spanning Tree . . . . .	43
6.5.1	Algoritmo di Prim . . . . .	44
6.5.2	Algoritmo di Kruskal . . . . .	44

# 1 Introduzione agli algoritmi

Un algoritmo è una successione di istruzioni per risolvere un problema, cioè per ottenere un preciso risultato a partire da un certo numero di dati iniziali. Inizieremo dando uno sguardo a degli esempi di algoritmi di ordinamento e ricerca, per poi passare a definire le notazioni asintotiche e i principali paradigmi di progettazione degli algoritmi.

**Osservazione** Se esiste una relazione d'ordine allora è possibile sviluppare un algoritmo di ordinamento

## 1.1 Insertion Sort

L'Insertion sort è un semplice algoritmo per ordinare un array. Non è molto diverso dal modo in cui un essere umano, spesso, ordina un mazzo di carte. Esso è un algoritmo in place, cioè ordina l'array senza doverne creare una copia, risparmiando memoria.

---

**Algorithm 1:** InsertionSort

---

**Data:** Un array  $A[1 \dots n]$

**Result:** Array  $A$

```
1 for  $i = 2$  to  $n$  do
2   key =  $A[i]$ ;
3    $j = i - 1$ ;
4   while  $j > 0$  and  $A[j] > key$  do
5      $A[j + 1] = A[j]$ ;
6      $j = j - 1$ ;
7   end
8    $A[j + 1] = key$ ;
9 end
```

---

### Esempio: Ordinamento di un array con l'Insertion Sort

Se per esempio  $A = [31, 41, 59, 26, 41, 58]$  il processo di ordinamento va come segue:

**Passo 1:** Consideriamo il secondo elemento  $A[2] = 41$ . È già nella posizione corretta rispetto al primo elemento. Quindi, l'array rimane invariato:

31	41	59	26	41	58
↑					
				$i = 2$	

**Passo 2:** Analizziamo il terzo elemento  $A[3] = 59$ , che è già in posizione corretta:

31	41	59	26	41	58
↑					
				$i = 3$	

**Passo 3:** Ora guardiamo il quarto elemento  $A[4] = 26$ . Questo deve essere inserito nella posizione corretta, spostando gli elementi più grandi verso destra:

31	41	59	26	41	58
↑					
				$i = 4$	

Dopo l'inserimento:

26	31	41	59	41	58
----	----	----	----	----	----

**Passo 4:** Consideriamo il quinto elemento  $A[5] = 41$ . Anche questo deve essere posizionato nella parte ordinata dell'array:

26	31	41	59	41	58
↑					
				$i = 5$	

Dopo l'inserimento:

26	31	41	41	59	58
----	----	----	----	----	----

**Passo 5:** Infine, esaminiamo il sesto elemento  $A[6] = 58$ . Questo va inserito giusto prima dell'ultimo elemento:

26	31	41	41	59	58
↑					
				$i = 6$	

Dopo l'inserimento:

26	31	41	41	58	59
----	----	----	----	----	----

```
# Insertion Sort in Python
def insertion_sort(Ar):
    for i in range(1, len(Ar)):
        key = Ar[i]
        j = i - 1
        while j >= 0 and Ar[j] > key:
            Ar[j + 1] = Ar[j]
            j = j - 1
        Ar[j + 1] = key
    return Ar
```

Caratteristiche di un algoritmo:

- Tempo di esecuzione
- Correttezza

## Invarianza di un ciclo

Un'invariante di ciclo è una proprietà o condizione che rimane vera all'inizio e alla fine di ogni iterazione di un ciclo. È utilizzata per dimostrare che il ciclo funziona correttamente e per aiutare a capire il comportamento del ciclo stesso.

1. Inizializzazione: È vera prima della prima iterazione del ciclo (nel nostro caso abbiamo che l'array inizialmente è ordinato, dato che il primo elemento è già ordinato rispetto a se stesso).
2. Mantenimento: Se è vera prima di un'iterazione del ciclo, allora è vera anche prima della successiva iterazione (ad ogni iterazione  $i+1$  il sottoarray  $A[1 \dots i]$  è ordinato).
3. Terminazione: Quando il ciclo termina, l'invariante di ciclo è sufficiente per dimostrare la correttezza dell'algoritmo (alla fine del ciclo, l'intero array è ordinato).

## Random Access Machine

La Random Access Machine (RAM) è un modello teorico di calcolatore che rappresenta un'astrazione del funzionamento di un computer reale. La RAM è composta da un'unità di controllo, un'unità aritmetico-logica, una memoria e un'unità di input/output. La memoria è composta da celle di memoria, ciascuna delle quali può contenere un'unità di informazione. Ogni cella di memoria ha un indirizzo univoco, che può essere utilizzato per accedere direttamente alla cella stessa. L'unità di controllo esegue le istruzioni del programma, mentre l'unità aritmetico-logica esegue le operazioni aritmetiche e logiche. L'unità di input/output è responsabile della comunicazione con l'esterno. Questo modello è utile per analizzare la complessità spazio-temporale degli algoritmi. Al momento deve essere chiaro che:

- Le operazioni sono effettuate una dopo l'altra (non ci sono carichi parallelizzati)
- Ogni operazione elementare richiede un tempo costante (e.g. 1 unità di tempo = accesso a una variabile = storing di un valore = confronto tra due valori etc.)

Per quanto riguarda il tempo di esecuzione del nostro algoritmo di “Insertion Sort” possiamo dire in primo luogo che questo dipende dall’input. Infatti si possono verificare le seguenti:

- Si dà in input un array già ordinato (best case)
- Si dà in input un array ordinato al contrario (worst case)

Definiamo il tempo di esecuzione come il numero di operazioni elementari eseguite dall’algoritmo. Riprendendo l’algoritmo 1 possiamo ipotizzare la seguente **analisi del costo**:

<b>Linea</b>	<b>Costo</b>	<b>Volte</b>
1	$c_1$	$n$
2	$c_2$	$n - 1$
3	$c_3$	$n - 1$
4	$c_4$	$\sum_{i=2}^n t_i$
5	$c_5$	$\sum_{i=2}^n (t_i - 1)$
6	$c_6$	$\sum_{i=2}^n (t_i - 1)$
7	$c_7$	$n - 1$

**Costo Totale:**

$$T(n) = c_1n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{i=2}^n t_i + c_5 \sum_{i=2}^n (t_i - 1) + c_6 \sum_{i=2}^n (t_i - 1) + c_7(n - 1)$$

**Best case**

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Si osservi come questa forma è riconducibile a una funzione lineare in  $n$   $a\mathbf{n} + b$

**Worst case**

$$T(n) = c_1n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{i=2}^n i + c_5 \sum_{i=2}^n (i - 1) + c_6 \sum_{i=2}^n (i - 1) + c_7(n - 1) =$$

Utilizzando la formula di Gauss

$$\sum_{i=2}^n i = \frac{n(n+1)}{2} - 1 \quad \sum_{i=2}^n (i-1) = \sum_{i=1}^{n-1} i$$

Si ottiene:

$$\begin{aligned} &= c_1n + c_2(n-1) + c_3(n-1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) + c_5 \left( \frac{n(n-1)}{2} \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7(n-1) = \\ &= \left( \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left( c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

Si osservi come questa forma è più complessa e riconducibile a una funzione quadratica in  $n$ , il cui tempo di esecuzione è quadratico rispetto al caso “migliore”.

## 1.2 Binary Search

Il Binary Search è un algoritmo di ricerca che trova la posizione di un valore in un array ordinato. L’array deve essere ordinato in modo crescente o decrescente. L’algoritmo confronta il valore da cercare con il valore centrale dell’array. Se il valore cercato è uguale al valore centrale, l’algoritmo restituisce la posizione del valore centrale. Se il valore cercato è minore del valore centrale, l’algoritmo cerca nella metà inferiore dell’array. Se il valore cercato è maggiore del valore centrale, l’algoritmo cerca nella metà superiore dell’array. L’algoritmo continua a dividere l’array in due parti e a cercare nella metà corretta fino a quando il valore è stato trovato o fino a quando l’array è vuoto.

---

**Algorithm 2:** BinarySearch

---

**Data:** Un array  $A[1 \dots n]$  ordinato in modo crescente, un valore target  $v$

**Result:** La posizione di  $v$  in  $A$

```
1 low = 1;
2 high = n;
3 while low ≤ high do
4     mid = ⌊(low+high)/2⌋;
5     if A[mid] = v then
6         return mid;
7     else
8         if A[mid] > v then
9             high = mid - 1;
10        else
11            low = mid + 1;
12        end
13    end
14 end
15 return "Not found";
```

---

Ad esempio, dato come input l'array  $A = [1, 3, 7, 11, 13, 14, 18]$  e il valore  $v = 7$ , l'algoritmo restituirà la posizione 3, poiché  $A[3] = 7$ . In generale, il Binary Search ha un tempo di esecuzione logaritmico, cioè  $O(\log n)$ , dove  $n$  è la dimensione dell'array. Questo è dovuto al fatto che l'algoritmo dimezza la dimensione dell'array ad ogni iterazione.

**Lemma 1.1.**  $\forall a \in \mathbb{R}^n$  ordinato, “Binary Search” trova la posizione di  $v$  in  $a$  in tempo  $O(\log n)$  nel caso peggiore.

$$T(n) := \text{tempo di esecuzione dell'algoritmo}$$

*Dimostrazione.* Dato che l'array può essere bisecato ad ogni iterazione si ha che  $n = 2^r$  dove  $r$  è il numero di iterazioni. Quindi  $r = \log n$ .

$$T(n) = 1 + T\left(\frac{n}{2}\right) = 1 + 1 + T\left(\frac{n}{4}\right) = \dots = r + T(1) = O(r) = O(\log n)$$

□

In Python, l'algoritmo di Binary Search può essere implementato ricorsivamente come segue:

```
# Binary Search in Python
def binary_search(Ar, x: int, start=0, end=None):
    if end is None:
        end = len(Ar) - 1
    if start > end:
        return -1
    mid = (start + end) // 2
    if Ar[mid] == x:
        return mid
    elif Ar[mid] > x:
        return binary_search(Ar, x, start, mid - 1)
    else:
        return binary_search(Ar, x, mid + 1, end)
```

## 2 Notazioni asintotiche

### 2.1 Introduzione

Ogni algoritmo presenta un tempo di esecuzione che vede in primo luogo la sua dipendenza dall'input:

- **Best case:** il tempo di esecuzione è minimo
- **Worst case:** il tempo di esecuzione è massimo
- **Average case:** il tempo di esecuzione è medio

È utile concentrare la nostra attenzione sul caso peggiore, in quanto ci dà un'idea di quanto l'algoritmo possa essere lento in situazioni critiche. Inoltre il caso peggiore è spesso quello più frequente, oltre al fatto che il caso medio va spesso come il caso peggiore. Quello su cui ci focalizzeremo sarà il **running time asintotico**. Ne si deduce quindi che il caso peggiore interessa il termine che ha un ordine di grandezza maggiore, tale da rendere trascurabile i termini di ordine inferiore.

#### 2.1.1 Notazione Big O

La notazione Big  $O$  è una notazione asintotica che caratterizza un **upper bound** sul comportamento asintotico di una funzione. In altre parole ci dice che una funzione non cresce più velocemente di un certo tasso di crescita.

**Esempio 2.1.** Sia  $f(n) = 7n^3 + 100n^2 + 200n + 6$ . Individuiamo che il termine di ordine superiore è  $7n^3$ . Quindi  $f(n)$  non cresce più velocemente di  $O(n^3)$ , ma si può anche dire che non cresce più velocemente di  $O(n^4)$  o  $O(n^5)$  etc.

#### 2.1.2 Notazione Omega

La notazione Omega  $\Omega$  è una notazione asintotica che caratterizza un **lower bound** sul comportamento asintotico di una funzione. In altre parole ci dice che una funzione cresce **almeno** alla velocità di un certo tasso di crescita.

**Esempio 2.2.** Se riconsideriamo la funzione  $f(n) = 7n^3 + 100n^2 + 200n + 6$ , possiamo dire che  $f(n)$  cresce almeno quanto  $n^3$ , quindi  $f(n) \in \Omega(n^3)$ , ma si anche dire che cresce almeno quanto  $n^2$  o  $n$  etc.

#### 2.1.3 Notazione Theta

La notazione Theta  $\Theta$  è una notazione asintotica che caratterizza un **tight bound** (o limite stretto) sul comportamento asintotico di una funzione. Ci dice che una funzione cresce **precisamente** a un certo tasso di crescita. Se dimostriamo che una funzione è sia  $O(g(n))$  che  $\Omega(g(n))$ , allora possiamo dire che è anche  $\Theta(g(n))$ .

**Esempio 2.3.** Se riconsideriamo la funzione  $f(n) = 7n^3 + 100n^2 + 200n + 6$ , possiamo dire che  $f(n)$  è sia  $O(n^3)$  che  $\Omega(n^3)$ , e quindi  $f(n) \in \Theta(n^3)$ . Si può dimostrare che questa è una relazione di equivalenza.

## 2.2 Definizioni formali

### $O$ -notation

**Definizione 2.1.** Data una funzione  $g(n)$ , si denota con  $O(g(n))$  l'insieme delle funzioni

$$O(g(n)) = \{f(n) \text{ se } \exists c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

**Esempio 2.4.**  $f(n) = 4n^2 + 100n + 500 \leq cn^2$  per un certo  $c$ .

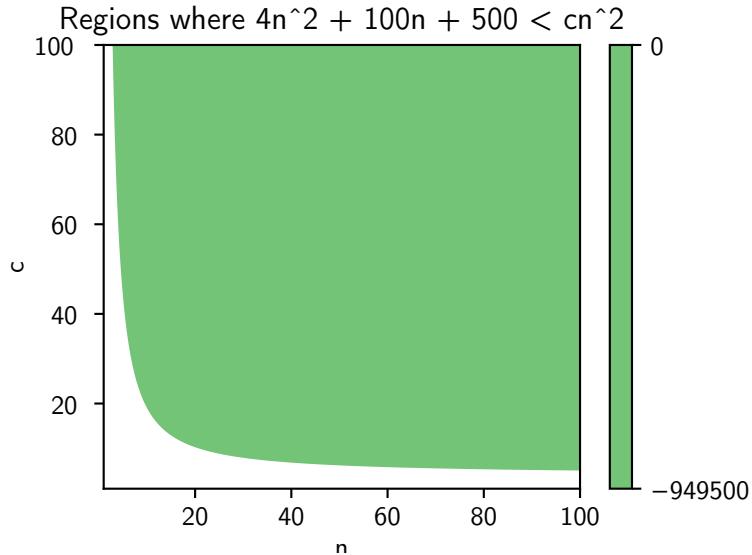


Figura 1: Curva di livello Big  $O$

### $\Omega$ -notation

**Definizione 2.2.** Data una funzione  $g(n)$ , si denota con  $\Omega(g(n))$  l'insieme delle funzioni

$$\Omega(g(n)) = \{f(n) \text{ se } \exists c, n_0 > 0 \mid 0 \leq cg(n) \leq f(n) \forall n \geq n_0\}$$

**Esempio 2.5.**  $f(n) = 4n^2 + 100n + 500 \geq cn^2$  per un certo  $c$ .

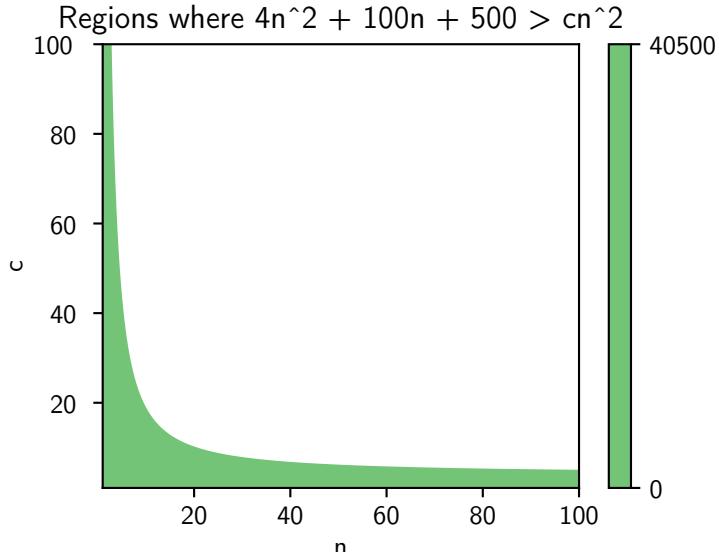


Figura 2: Curva di livello  $\Omega$

### $\Theta$ -notation

**Definizione 2.3.** Data una funzione  $g(n)$ , si denota con  $\Theta(g(n))$  l'insieme delle funzioni

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

**Teorema 1.** Date due funzioni  $f(n)$  e  $g(n)$ , si ha che  $f(n) = \Theta(g(n))$  se e solo se  $f(n) = O(g(n))$  e  $f(n) = \Omega(g(n))$ .

### $o$ -notation

**Definizione 2.4.** Data una funzione  $g(n)$ , si denota con  $o(g(n))$  l'insieme delle funzioni

$$o(g(n)) = \{f(n) \text{ se } \forall c > 0, \exists n_0 > 0 \mid 0 \leq f(n) < cg(n) \forall n \geq n_0\}$$

Ovvero  $g(n)$  domina sulla crescita di  $f(n)$ . In corsi di analisi si può trovare la seguente definizione, che tuttavia presenta dei casi limite e non è sempre utilizzabile:

$$f(n) = o(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

**Esempio 2.6.**  $2n = o(n^2)$  ma  $2n^2 \neq o(n^2)$ .

### $\omega$ -notation

**Definizione 2.5.** Data una funzione  $g(n)$ , si denota con  $\omega(g(n))$  l'insieme delle funzioni

$$\omega(g(n)) = \{f(n) \text{ se } \forall c > 0, \exists n_0 > 0 \mid 0 \leq cg(n) < f(n) \forall n \geq n_0\}$$

Ovvero  $f(n)$  domina sulla crescita di  $g(n)$ . In corsi di analisi si può trovare la seguente definizione, che tuttavia presenta dei casi limite e non è sempre utilizzabile:

$$f(n) = \omega(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

**Esempio 2.7.**  $n^2 = \omega(n)$  ma  $\frac{1}{2}n^2 \neq \omega(n^2)$ .

**Osservazione** Si ricorda che  $T(n)$  è la funzione che associa al problema di dimensione  $n$  il running time dell'algoritmo nel suo caso peggiore.

L'algoritmo di “Insertion Sort” ha un tempo di esecuzione  $O(n^2)$  nel caso peggiore, e dato che l'ordine di un'array sarebbe essere ordinato in modo casuale si arriverebbe ad un caso in cui per un  $i$ -esimo elemento si avrebbero  $i - 1$  confronti.

## 2.3 Divide & Conquer

Il paradigma Divide & Conquer è un metodo di risoluzione di problemi che prevede la suddivisione di un problema in sottoproblemi più piccoli, la risoluzione dei sottoproblemi e la combinazione delle soluzioni per ottenere la soluzione del problema originale. Questo metodo è spesso utilizzato per progettare algoritmi efficienti per problemi di ricerca, ordinamento e ottimizzazione. Gli step principali di un algoritmo Divide & Conquer sono:

1. **Divide:** Il problema è diviso in sottoproblemi più piccoli che hanno la stessa struttura del problema originale.
2. **Conquer:** I sottoproblemi sono risolti in modo ricorsivo.
3. **Combine:** Le soluzioni dei sottoproblemi sono combinate per ottenere la soluzione del problema originale.

Gli algoritmi che fanno uso del paradigma Divide & Conquer fanno spesso uso di un approccio ricorsivo per risolvere i sottoproblemi.

## 2.4 Merge Sort

Il Merge Sort è un algoritmo di ordinamento che utilizza il paradigma Divide & Conquer. L'algoritmo divide l'array in due metà, ordina le due metà in modo ricorsivo e combina le due metà ordinate per ottenere l'array ordinato. L'algoritmo è efficiente per ordinare grandi array e ha un tempo di esecuzione  $O(n \log n)$  nel caso peggiore.

---

**Algorithm 3:** Merge

---

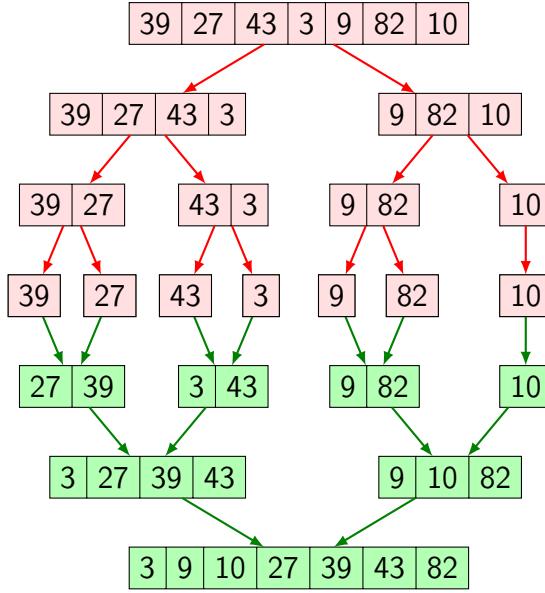
**Input:** Un array  $A$ , e tre indici  $p$  (primo indice),  $q$  e  $r$  (ultimo indice)

**Output:** Array  $A$  ordinato tra gli indici  $p$  e  $r$

```
1  $n_l = q - p + 1;$ 
2  $n_r = r - q;$ 
3 Creare  $L[0 \dots n_l - 1]$  e  $R[0 \dots n_r - 1]$  due nuovi array;
4 for  $i = 0$  to  $n_l - 1$  do
5   |  $L[i] = A[p + i];$ 
6 end
7 for  $j = 0$  to  $n_r - 1$  do
8   |  $R[j] = A[q + j + 1];$ 
9 end
10  $i = 0;$ 
11  $j = 0;$ 
12  $k = 0;$ 
13 while  $i < n_l$  and  $j < n_r$  do
14   | if  $L[i] \leq R[j]$  then
15     |   |  $A[k] = L[i];$ 
16     |   |  $i = i + 1;$ 
17   | else
18     |   |  $A[k] = R[j];$ 
19     |   |  $j = j + 1;$ 
20   | end
21   |  $k = k + 1;$ 
22 end
23 while  $i < n_l$  do
24   |  $A[k] = L[i];$ 
25   |  $i = i + 1;$ 
26   |  $k = k + 1;$ 
27 end
28 while  $j < n_r$  do
29   |  $A[k] = R[j];$ 
30   |  $j = j + 1;$ 
31   |  $k = k + 1;$ 
32 end
```

---

Ecco una schematizzazione dell'algoritmo di Merge:




---

**Algorithm 4:** MergeSort

---

**Data:** Un array  $A$ , e due indici  $p, r$   
**Result:** Array  $A$  ordinato

```

1 if  $p \geq r$  then
2   | return;
3 else
4   |    $q = \left\lfloor \frac{p+r}{2} \right\rfloor$ ;
5   |   MergeSort( $A, p, q$ );
6   |   MergeSort( $A, q + 1, r$ );
7   |   Merge( $A, p, q, r$ );
8 end

```

---

Ecco come appare l'algoritmo di Merge Sort in Python:

```

def merge(left_A, right_A):
    output = []
    while left_A and right_A:
        if left_A[0] <= right_A[0]:
            new = left_A.pop(0)
        else:
            new = right_A.pop(0)
        output.append(new)
    output += left_A + right_A
    return output

```

```

def merge_sort(Arr):
    if len(Arr) <= 1:
        return Arr
    mid = len(Arr) // 2
    left = merge_sort(Arr[:mid])
    right = merge_sort(Arr[mid:])
    return merge(left, right)

```

**Osservazione** Merge sort e Merge sono due cose diverse:

- Merge sort è un algoritmo di ordinamento che utilizza il paradigma Divide & Conquer.
- Merge è una funzione che combina due array ordinati in un unico array ordinato.

**Lemma 2.1** (Merge). *Date due sequenze ordinate di lunghezza  $m$  impiega tempo  $O(m)$  per combinare le due sequenze in un'unica sequenza ordinata.*

*Dimostrazione.* Dato che le due sequenze sono ordinate, ogni volta che si confrontano due elementi si sta confrontato uno dei  $2m$  elementi. Le operazioni di base di Merge sono:

- Confronto tra due elementi
- Assegnamento di un valore a un array

□

**Lemma 2.2.** *L'altezza dell'albero di ricorsione chiamato su un array di dimensione  $n$  è al più  $\lceil \log_2 n \rceil \leq 2 \log_2 n$ . (Da ora in poi  $\log n$  sarà inteso come  $\log_2 n$ )*

*Dimostrazione.* Dimostriamo innanzitutto che  $O(\log_2 n) = O(\log_{10} n)$ . Procediamo quindi con la doppia inclusione:

$$\begin{aligned} \supseteq f \in O(\log_{10} n) &\implies \exists c, n_0 > 0 \mid f(n) \leq c \log_{10} n \leq c \log_2 n \quad \forall n \geq n_0 \\ \subseteq f \in O(\log_2 n) &\implies \exists c, n_0 > 0 \mid f(n) \leq c \log_2 n \leq c \frac{\log_{10}(n)}{\log_{10}(2)} \quad \forall n \geq n_0 \end{aligned}$$

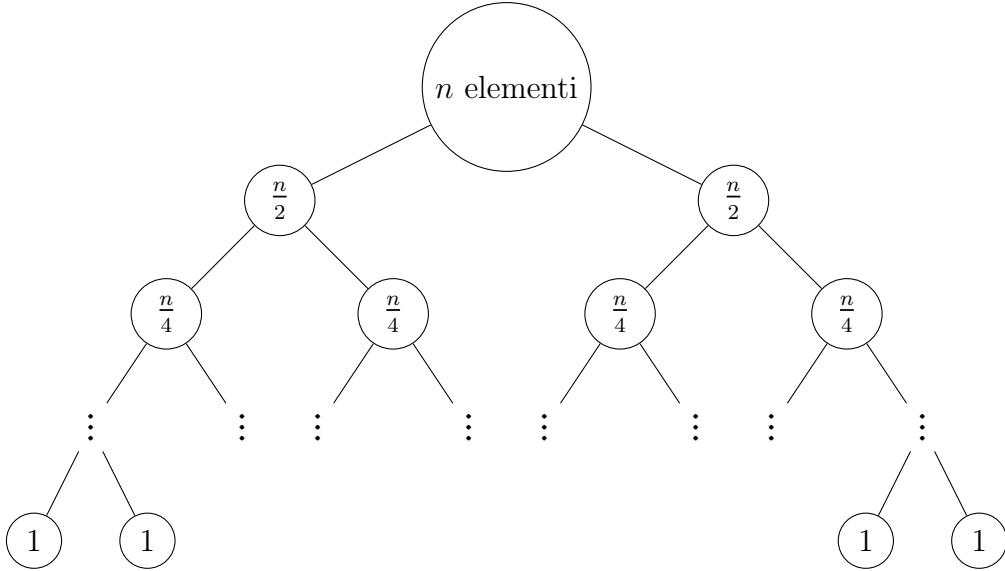
Si ha quindi che il livello  $l$  contiene al più  $\frac{n}{2^l}$  elementi. Quindi al livello  $H$  si avrà l'intero più piccolo  $H$  tale per cui  $\frac{n}{2^H} \leq 1$ . Quindi  $H = \lceil \log_2 n \rceil \leq 2 \log_2 n$ . □

**Teorema 2.** *Merge Sort su un array di dimensione  $n$  ha un running time  $T(n) \in O(n \log n)$ .*

*Dimostrazione.* Consideriamo tutto quello che accade al livello  $l$ -esimo dell'albero di ricorsione: avrà  $2^l$  nodi, ognuno dei quali avrà dimensione  $\frac{n}{2^l}$ .

- Per la prima parte si che il tempo di esecuzione al livello  $l$  sarà  $T_1(l) \leq C_1 \cdot 2^l \cdot \frac{n}{2^l} = C_1 \cdot n$
- Per la seconda parte si ha che il tempo di esecuzione al livello  $l$  sarà  $T_2(l) \leq C_2 \cdot 2^l \cdot \frac{n}{2^{l+1}} = \frac{C_2}{2} \cdot n$  (si osservi che  $2^{l+1}$  dato che riceve i due array dal livello sottostante)

Quindi il tempo di esecuzione totale sarà  $T(n) \leq (C_1 + \frac{C_2}{2})n \implies \text{Running time}(n) \leq 2(C_1 + \frac{C_2}{2})\log n \cdot n$  dove  $\log n$  è l'altezza dell'albero di ricorsione. □



**Teorema 3.** *Ogni algoritmo di ordinamento basato su confronti ha un running time nel caso peggiore  $\Omega(n \log n)$ .*

*Dimostrazione.* Si consideri un albero di decisione che rappresenta tutti i possibili ordinamenti di un array di dimensione  $n$ . Ogni percorso dall’alto verso il basso rappresenta una sequenza di confronti: ogni nodo è un confronto tra due elementi e ogni percorso termina in una foglia, e dato che ci sono  $n!$  possibili ordinamenti, l’albero avrà almeno  $n!$  foglie e quindi l’altezza dell’albero sarà nel caso peggiore  $\log(n!)$ . Si ha quindi che  $\log(n!) \in \Omega(n \log n)$ . (dimostrazione con la formula di Stirling)  $\square$

**Spazio di memoria utilizzato da Merge Sort** Merge Sort utilizza uno spazio di memoria aggiuntivo  $O(n)$  per memorizzare i due array temporanei  $L$  e  $R$ . Quindi lo spazio di memoria utilizzato da Merge Sort è  $O(n)$ .

## 2.5 2-Sum Problem

Il 2-Sum Problem è un problema di ricerca che chiede di trovare due numeri  $x, y$  in un array  $A$  tali che  $x + y = t$ , dove  $t$  è un valore target. Ecco variante del problema in forma algoritmica:

**Variante I** Input: Un array  $A$  non ordinato di interi distinti  
 Target: un intero  $x$  tale per cui  $\exists i, j \in [n] a[i] + a[j] = x$ .  
 Requisito: Trovare  $x$  in  $A$  in tempo  $T(n) \in O(n)$  e spazio  $S(n) \in O(n)$ .  
 Conosciamo la struttura dati **Hash Map** (o **Dictionary** o **Hash Table**) che permette di memorizzare e recuperare valori in tempo costante  $O(1)$ .

---

**Algorithm 5:** TwoSum

---

**Data:** Un array  $A$  di interi distinti, e un intero  $x$   
**Result:** Due indici  $i, j$  tali che  $A[i] + A[j] = x$ , se esistono

```
1  $H = \{\}$ ;
2 for  $i = 0$  to  $n - 1$  do
3   if  $x - A[i] \in H$  then
4     return  $i, H[x - A[i]]$ ;
5   else
6      $H[A[i]] = i$ ;
7   end
8 end
9 return "Nessuna coppia trovata";
```

---

In Python l'algoritmo può essere implementato come segue:

```
def two_sum(Ar: list, x: int):
    H = {}
    for i in range(len(Ar)):
        if x - Ar[i] in H:
            return i, H[x - Ar[i]]
        else:
            H[Ar[i]] = i
    return "Nessuna coppia trovata"
```

**Variante II** Input: Un array  $A$  ordinato di interi distinti  
Target: un intero  $x$  tale per cui  $\exists i, j \in [n] a[i] + a[j] = x$ .  
Requisito: Trovare  $x$  in  $A$  in tempo  $T(n) \in O(n)$  e spazio  $S(n) \in O(1)$ .

---

**Data:** Un array  $A$  di interi distinti, e un intero  $x$   
**Result:** Due indici  $i, j$  tali che  $A[i] + A[j] = x$ , se esistono

```
1  $i = 0$ ;
2  $j = n - 1$ ;
3 if  $A[i] + A[j] < x$  then
4    $i = i + 1$ ;
5 end
6 else if  $A[i] + A[j] > x$  then
7    $j = j - 1$ ;
8 end
9 else
10  return  $i, j$ ;
11 end
```

---

### 3 Strutture dati

I computer moderni presentano due componenti principali per l'esecuzione di codice: la **CPU** e la memoria **RAM**. La memoria RAM è divisa in celle di memoria, ognuna delle quali ha un indirizzo univoco (0x00). Le azioni che possiamo compiere su una cella di memoria sono:

- **Insert**: inserire un valore in una cella di memoria
- **Delete**: eliminare un valore da una cella di memoria
- **Search**: cercare un valore in una cella di memoria

### 3.1 Array

Un array è una struttura dati che memorizza un insieme di elementi dello stesso tipo in celle di **memoria contigue** e si utilizzano per memorizzare collezioni di elementi di dimensione fissa e quindi di **lunghezza predeterminata**. Tutti gli elementi di un array sono dello stesso tipo. Gli elementi di un array sono accessibili tramite un indice, che rappresenta la posizione dell'elemento nell'array.

0	1	2	3
$a_0$	$a_1$	$a_2$	$a_3$

Dove  $a_i$  è l'elemento dell'array in posizione  $i$  e tutti gli elementi sono dello stesso tipo.

### 3.2 Lista

Una lista è una struttura dati che memorizza un insieme di elementi dello stesso tipo in celle di memoria **non contigue**. Le liste sono utilizzate per memorizzare collezioni di elementi di dimensione variabile e quindi di **lunghezza dinamica**. Inoltre le liste possono memorizzare elementi di tipo diverso.

### 3.3 Matrici

Per la memorizzazione di matrici si possono utilizzare array bidimensionali. Un array bidimensionale è una struttura dati che memorizza un insieme di elementi in celle di memoria contigue, ma in questo caso le celle sono organizzate in righe e colonne. Quindi se si volesse accedere al primo elemento della prima riga dell'array  $A$  si scriverebbe  $A[0][0]$ .

0	1	2	3
$a_{00}$	$a_{01}$	$a_{02}$	$a_{03}$
$a_{10}$	$a_{11}$	$a_{12}$	$a_{13}$
$a_{20}$	$a_{21}$	$a_{22}$	$a_{23}$

### 3.4 Pila (Stack)

Una pila è una struttura dati che memorizza un insieme di elementi dello stesso tipo in celle di memoria contigue. Una pila è una struttura dati **LIFO** (Last In First Out), ovvero l'ultimo elemento inserito è il primo ad essere rimosso, ed è quindi una struttura di lunghezza **dinamica**. L'operazione di inserimento di un elemento in cima alla pila è detta **push**, mentre l'operazione di rimozione di un elemento dalla cima della pila è detta **pop**. Infine vi è l'operazione di **stack empty** che controlla se la pila è vuota.

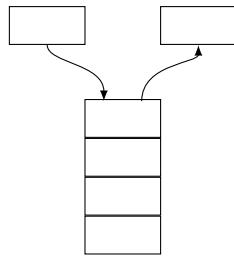


Figura 3: Rappresentazione di una pila e del suo funzionamento basato sul principio LIFO.

### 3.5 Coda (Queue)

Una coda è una struttura dati che memorizza un insieme di elementi dello stesso tipo in celle di memoria contigue. Una coda è una struttura dati **FIFO** (First In First Out), ovvero il primo elemento inserito è il primo ad essere rimosso, ed è quindi una struttura di lunghezza **dinamica**. L'operazione di inserimento di un elemento in coda alla coda è detta **enqueue**, mentre l'operazione di rimozione di un elemento dalla testa della coda è detta **dequeue**.

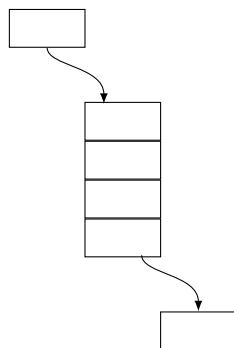


Figura 4: Rappresentazione di una coda e del suo funzionamento basato sul principio FIFO.

### 3.6 Lista concatenata (Linked List)

Una lista concatenata è una struttura dati che memorizza una serie di elementi che sono collegati tra loro da puntatori. Quindi ogni elemento punta all'elemento successivo nella lista, e l'accesso agli elementi della lista è sequenziale, ovvero si parte dall'elemento iniziale e si procede fino all'elemento cercato. Vi sono due tipi di liste concatenate:

- **Semplicemente concatenata:** ogni elemento punta all'elemento successivo nella lista.
- **Doppicamente concatenata:** ogni elemento punta all'elemento successivo e a quello precedente nella lista.

Si distinguono inoltre due tipi di liste concatenate:

- **Lista concatenata circolare:** l'ultimo elemento punta al primo elemento della lista.

- **Lista concatenata non circolare:** l'ultimo elemento punta a un valore nullo.

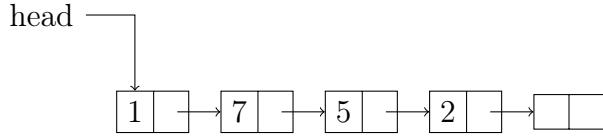


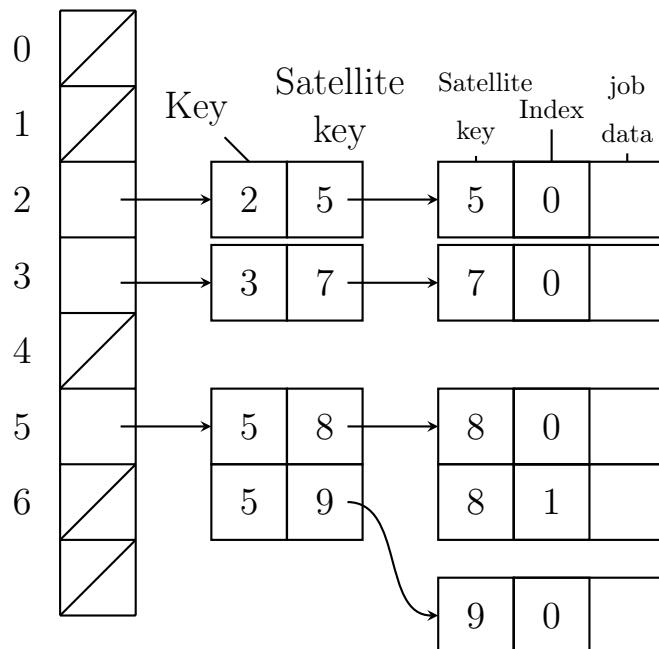
Figura 5: Rappresentazione di una lista concatenata.

Le operazioni che si possono compiere su una lista concatenata sono:

- **Insert:** inserire un elemento in una posizione specifica della lista ( $O(1)$  se si conosce la posizione)
- **Delete:** eliminare un elemento dalla lista
- **Search:** cercare un elemento nella lista ( $O(n)$  dato che l'accesso è sequenziale)

### 3.7 Tabella a indirizzamento diretto (Direct Address Table)

Trovare quindi un particolare elemento all'interno di un Array richiede un tempo  $O(n)$ , quindi potrebbe non essere la struttura dati più adatta per questo tipo di operazioni. Possiamo immaginare un insieme “universo”  $U$  delle chiavi possibili (numerabile, **ragionevolmente piccolo**, assioma molto forte), e quindi il suo sottoinsieme delle chiavi effettivamente di interesse presenti nella tabella  $T$ . Per ogni chiave di questo insieme è quindi corrisposta una cella nella tabella  $T$  che ha lo stesso indirizzo della chiave. Una possibile idea è quella di immaginare in un aula di 100 studenti (insieme  $U$ ) il sottoinsieme delle chiavi degli studenti in prima fila.



Ora dato che l'insieme  $U$  non potrà essere troppo piccolo ( $m \ll |U|$  considerando  $m$  la lunghezza dell'Array), si utilizzerà una Hash Table per memorizzare le chiavi.

## 3.8 Hash Table

Una Hash Table è una struttura dati che memorizza un insieme di elementi in celle di memoria contigue e che permette di memorizzare e recuperare valori in tempo costante  $O(1)$ . La funzione che calcola l'indirizzo di una cella di memoria in cui memorizzare un elemento è detta **funzione di hash** così definita:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

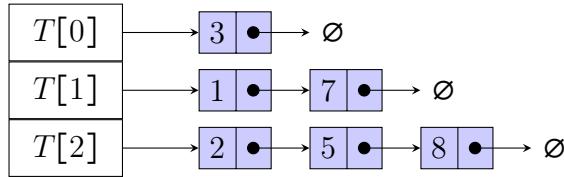
Dove  $U$  è l'insieme delle chiavi possibili e  $m$  è la dimensione della tabella hash. Una possibile funzione di hash è:

$$h(k) = k \mod m$$

Dove  $k$  è la chiave da memorizzare e  $m$  è la dimensione della tabella hash.

### 3.8.1 Collisioni

Così definita la funzione di hash, si potrebbero avere delle collisioni, ovvero due chiavi diverse che vengono mappate nella stessa cella di memoria. Ad esempio se definiamo una funzione di hash  $h(k) = k \mod 10$  e abbiamo due chiavi  $k_1 = 3$  e  $k_2 = 13$ , entrambe verranno mappate nella cella  $h(3) = h(13) = 3$ . Questo problema può essere risolto grazie al **concatenamento** delle chiavi nella stessa cella di memoria, ovvero memorizzare le chiavi in una lista concatenata che si trova nella cella di memoria. Una buona funzione di hash è quindi una funzione che minimizza le collisioni, ovvero che ha il valore atteso di collisioni minore possibile.



Nella cella di memoria è memorizzato quindi il puntatore alla lista concatenata che contiene le chiavi.

## 4 Algoritmi di ordinamento

**Esempio 4.1** (Dutch Flag Problem). È dato in input un array  $A$  di  $k$  elementi noti. Si vuole implementare un algoritmo di ordinamento che abbia un running time  $O(n)$  e uno spazio di memoria  $O(n)$ . Nel caso di un array in cui ho 3 elementi (0,1,2)

```

def sort_colors_1(arr):
    dict_colors = {0: 0, 1: 0, 2: 0}
    for _ in arr:
        dict_colors[_] += 1
    i = 0
    while i < dict_colors[0]:
        arr[i] = 0
        i += 1
    while i < dict_colors[0]+dict_colors[1]:
        arr[i] = 1
        i += 1
    while i < dict_colors[0]+dict_colors[1]+dict_colors[2]:
        arr[i] = 2
        i += 1
  
```

```

        arr[i] = 1
        i += 1
    while i < dict_colors[0]+dict_colors[1]+dict_colors[2]:
        arr[i] = 2
        i += 1
    return arr

```

Questo algoritmo ha tuttavia la falla di eseguire il tutto in due passaggi.

```

def sort_colors_2(arr):
    l, i = 0, 0
    hi = len(arr)-1
    while i <= hi:
        if arr[i] == 0:
            arr[l], arr[i] = arr[i], arr[l] # swapping elements
            l += 1
        if arr[i] == 2:
            arr[hi], arr[i] = arr[i], arr[hi]
            hi -= 1 # non ci interessiamo dell'ultima posizione
            i -= 1
        i += 1
    return arr

```

**Esempio 4.2** (Wiggle Sort). È dato in input un array  $A$  di dimensione  $n$  disordinato. Come output si vuole un array tale che  $A[i] \leq A[i+1] \geq A[i+2] \leq A[i+3] \geq \dots$ . L'algoritmo deve lavorare in place e avere un running time  $O(n)$  e uno spazio di memoria  $O(1)$ .

```

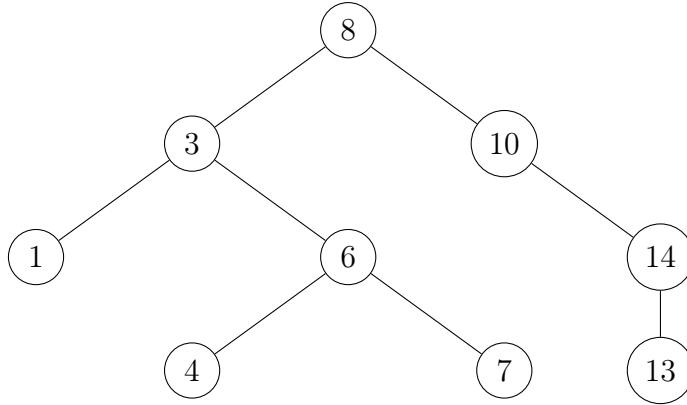
def wiggle_sort(arr):
    for i in range(1, len(arr) - 1):
        if i % 2 == 1 and arr[i] < arr[i - 1] or i % 2 == 0 and arr[i] >
           ↪ arr[i - 1]:
            arr[i], arr[i - 1] = arr[i - 1], arr[i] # indici dispari a
            ↪ monte, pari a valle
    return arr

```

In questo algoritmo è insita una relazione di transitività: se  $A[i] < A[i-1]$  e  $A[i-1] < A[i-2]$  allora  $A[i] < A[i-2]$ .

## 4.1 Binary Search Tree

Sia  $x$  un nodo di un albero binario di ricerca binario. Allora se  $y$  è un nodo nel sottoalbero destro di  $x$ , allora  $\text{key}[y] \geq \text{key}[x]$ . Se  $y$  è un nodo nel sottoalbero sinistro di  $x$ , allora  $\text{key}[y] \leq \text{key}[x]$ .



Distinguiamo diversi tipi di cammini in un albero binario di ricerca: **Inorder Tree Walk** e **Search Tree**.

---

**Algorithm 6:** Inorder Tree Walk

---

```

1 if  $x \neq NIL$  then
2   | ITW( $x.left$ );
3   | print( $x.key$ );
4   | ITW( $x.right$ );
5 end

```

---

Questo algoritmo ha complessità  $O(n)$ .

---

**Algorithm 7:** Search Tree

---

**Data:** Un nodo  $x$  e una chiave  $k$

```

1 if  $x = NIL$  or  $k = x.key$  then
2   | return  $x$ ;
3 end
4 if  $k < x.key$  then
5   | return Search( $x.left$ ,  $k$ );
6 end
7 else
8   | return Search( $x.right$ ,  $k$ );
9 end

```

---

Il caso average è  $O(h)$  dove  $h$  è l'altezza dell'albero (ricerca di minimo o massimo), mentre il caso peggiore è  $O(n)$ , ovvero quando l'albero è molto sbilanciato.

## Insertion

Se si volesse inserire un nodo in un albero binario di ricerca si potrebbe utilizzare l'algoritmo **Insertion**, tenendo conto di 4 possibilità:

- L'elemento è maggiore della radice
- L'elemento è minore della radice
- L'elemento è uguale alla radice
- L'elemento è NIL

## Remove

Possiamo dividere la rimozione di un nodo in 2 passaggi:

- **Find Phase**

- L'elemento è maggiore della radice
- L'elemento è minore della radice
- L'elemento è uguale alla radice
- L'elemento è NIL

- **Remove Phase**

- Se il nodo da rimuovere è una foglia (caso facile)
- Uno tra il figlio sinistro o destro è un sottoalbero
- Entrambi i figli sono sottoalberi (caso più complesso, o si mette come radice il minimo del sottoalbero destro o il massimo del sottoalbero sinistro)

## 4.2 Interval Sorting

**Esempio 4.3** (Interval Sorting). Dato un array di intervalli  $[(s_1, l_1), \dots, (s_n, l_n)]$  ed è garantito che  $\forall i \ l_i < s_{i+1}$ . Si vuole implementare un algoritmo che restituisca una nuova lista  $L'$  che contiene  $I=(s, l)$  (l'intervallo da inserire) e non ha overlaps. L'algoritmo deve avere un running time  $O(n)$ .

```
def real_interval_searching(A: list, x: tuple):
    res = []
    for i in range(len(A)):
        if A[i][1] < x[0]:
            res.append(A[i])
        elif A[i][0] > x[1]:
            res.append(x)
            return res + A[i:]
        else:
            x = (min(A[i][0], x[0]), max(A[i][1], x[1])) # riguarda
    res.append(x)
    return res
```

**Esempio 4.4** (Remove counter). Dato un array di interi  $L$  (non ordinata) tali per cui si può costruire una lista  $L'$  senza overlaps e in complessità temporale  $n \log n$ .

```
def del_interval(A: list):
    A_new = sorted(A)
    prev_end = A_new[0][1]
    deletions = 0
    for i in range(len(A)):
        if A[i][0] > prev_end:
            prev_end = A[i][0]
        else:
```

```

    deletions += 1
    prev_end = min(prev_end, A[i][1])
return deletions

```

### 4.3 Quick Sort

Quick Sort è un algoritmo di ordinamento (sorting) basato su confronti che utilizza il paradigma **Divide et Impera**, suddiviso nelle fasi **Divide**, **Conquer** e **Combine**. Baseremo la nostra spiegazione sull'assunzione che l'array non contenga elementi duplicati. L'algoritmo funziona in modo simile a Merge Sort, ma anziché dividere l'array in due parti uguali, sceglie un elemento chiamato **pivot** (solitamente si preferisce l'ultimo numero dell'array), e divide l'array in due parti in modo che tutti gli elementi a sinistra del pivot siano minori di esso, e tutti gli elementi a destra del pivot siano maggiori di esso. Quindi si procede a ordinare le due parti dell'array in modo ricorsivo. Quindi l'algoritmo è diviso in tre fasi:

- **Divide:** Scegliere un pivot e dividere l'array in due parti  
 $A[:p]$  ovvero tutti gli elementi minori del pivot  
 $A[p+1:]$  ovvero tutti gli elementi maggiori del pivot
- **Impera:** Ordinare le due parti dell'array in modo ricorsivo
- **Combine:** Le foglie sono già ordinate, quindi non c'è bisogno di fare nulla

#### Algorithm 8: QuickSort

**Data:** Un array  $A$  di interi,  $p=1$ ,  $r=n$

**Result:** Un array ordinato

```

1 if  $p < r$  then
2   | q = Partition( $A$ ,  $p$ ,  $r$ );
3   | QuickSort( $A$ ,  $p$ ,  $q-1$ );
4   | QuickSort( $A$ ,  $q+1$ ,  $r$ );
5 end
6 return  $A$ ;

```

#### Algorithm 9: Partition

**Data:** Un array  $A$  di interi,  $p=1$ ,  $r=n$

**Result:** un indice  $i$  tale che  $A[p \dots i-1] \leq A[i] \leq A[i+1 \dots r]$

```

1 x =  $A[r]$ ;
2 i =  $p-1$ ;
3 for  $j = p$  to  $r-1$  do
4   | if  $A[j] \leq x$  then
5   |   | i =  $i + 1$ ;
6   |   | swap( $A[i]$ ,  $A[j]$ );
7   | end
8 end
9 swap( $A[i+1]$ ,  $A[r]$ );
10 return  $i+1$ ;

```

Valutiamo ora la correttezza dell'algoritmo analizzando l'invariante di ciclo che osserva queste regole:

1. Se  $p \leq k \leq i$  allora  $A[k] \leq x$
2. Se  $i + 1 \leq k \leq j - 1$  allora  $A[k] > x$
3. Se  $k = r$  allora  $A[k] = x$  ed è il pivot

- **Inizializzazione:** All'inizio del ciclo abbiamo  $i = p - 1$  e  $j = p$ , ed è soddisfatta
- **Mantenimento:** Se  $A[j] > x$  scorreremo  $j$  (soddisfatta per  $A[J - 1]$ ), altrimenti incrementeremo  $i$ , scambieremo  $A[i]$  con  $A[j]$  e poi incrementeremo anche  $j$
- **Terminazione:** Dopo  $r - p$  iterazioni l'algoritmo termina

**Worst Case** Il caso peggiore di Quick Sort si ha quando l'array è già ordinato e il pivot è sempre l'ultimo elemento dell'array. In questo caso l'algoritmo ha complessità  $O(n^2)$ .

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

E tramite una sommatoria si ottiene  $O(n^2)$ . Tuttavia questi casi rappresentano una piccola percentuale rispetto ai casi medi. È possibile ridurre la probabilità di avere un caso peggiore utilizzando un pivot casuale, questa variante è denominata **Randomized Quick Sort**.

## 4.4 Analisi Caso Atteso

**Lemma 4.1.** *Se  $X$  è il numero di confronti eseguibili, allora su un array di  $n$  elementi, il tempo di esecuzione atteso di Quick Sort è  $O(n + X)$ .*

*Dimostrazione.* Ci sono al massimo  $n$  chiamate a **Partition**, e ognuna di queste chiamate chiama al massimo 2 volte **QuickSort**. Quindi **Partition** fuori dal ciclo “for” ha costo  $O(1)$ , mentre all'interno del ciclo dipenderà dal numero di confronti. Quindi il costo totale è di  $O(n + X)$ . Voglio quindi analizzare il caso atteso di  $X$ .  $\square$

**Lemma 4.2.** *Data una procedura di R-QuickSort che sceglie un pivot casuale, e dato un insieme  $Z_{ij} = \{Z_i, Z_{i+1}, \dots, Z_j\}$  (con la caratteristica che  $Z_i < Z_{i+1} < \dots < Z_j$ ) sottoinsieme di  $A$ . Un elemento  $Z_j$  viene confrontato con  $Z_i$  solo se uno dei due è scelto come primo pivot  $x \in Z_{ij}$*

*Dimostrazione.* Abbiamo due casi:

- Se  $x \neq Z_i \wedge x \neq Z_j \wedge x \in Z_{ij}$  non avvengono confronti
- Se  $x = Z_j \vee x = Z_i$  avviene esattamente un confronto

$\square$

**Lemma 4.3.** *La probabilità che i due elementi  $Z_i$  e  $Z_j$  vengano confrontati è  $\frac{2}{j-i+1}$*

*Dimostrazione.*

$$\begin{aligned} \mathbb{P}(Z_i \text{ e } Z_j \text{ vengano confrontati}) &= \mathbb{P}(Z_i \text{ primo pivot}) + \mathbb{P}(Z_j \text{ primo pivot}) = \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \end{aligned}$$

$\square$

**Teorema 4.** *R-QuickSort ha complessità attesa  $O(n \log n)$*

*Dimostrazione.* Definiamo una funzione indicatrice  $X_{ij}$  che è verificata se  $Z_i$  e  $Z_j$  vengono confrontati.

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n x_{ij}$$

Siamo quindi interessati a calcolare il valore atteso di  $X$ , ovvero  $\mathbb{E}[X]$ :

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n x_{ij}\right] = \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[x_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{P}(X_{ji} = 1) = \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \end{aligned}$$

Ora possiamo effettuare un cambio di variabile, ovvero  $k = j - i$ , quindi la sommatoria diventa:

$$\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

Questa sommatoria è certamente minore di  $\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k}$  che a sua volta diventa:

$$\sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

□

Si può dimostrare che R-QuickSort ha complessità attesa  $O(n \log n)$  con una probabilità di  $1 - \frac{1}{n}$ .

Segue un implementazione di R-QuickSort in Python:

```
def Partition(A, p, r):
    num = random.randint(p, r) # Randomized Quicksort
    x = A[num]
    A[num], A[r] = A[r], A[num]
    i = p - 1
    for j in range(p, r):
        if A[j] <= x:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1]
    return i + 1

def QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q - 1)
        QuickSort(A, q + 1, r)
```

```

QuickSort(A, p, q - 1)
    QuickSort(A, q + 1, r)
return A

```

## 4.5 Merge Sort e Quick Sort

- Merge Sort

- **Vantaggi**

- \* Complessità  $O(n \log n)$  nel caso peggiore
    - \* Stabile

- **Svantaggi**

- \* Richiede spazio aggiuntivo
    - \* Complessità  $O(n \log n)$  nel caso medio

- Quick Sort

- **Vantaggi**

- \* Complessità  $O(n \log n)$  nel caso medio
    - \* Non richiede spazio aggiuntivo

- **Svantaggi**

- \* Complessità  $O(n^2)$  nel caso peggiore (array già ordinato)
    - \* Non è stabile (se ho due elementi uguali possono essere scambiati)

Algoritmo	Caso	Tempo di Esecuzione	Complessità Spaziale
HeapSort	Best	$O(n \log n)$	$O(1)$
	Average	$O(n \log n)$	$O(1)$
	Worst	$O(n \log n)$	$O(1)$
QuickSort	Best	$O(n \log n)$	$O(\log n)$
	Average	$O(n \log n)$	$O(\log n)$
	Worst	$O(n^2)$	$O(n)$
MergeSort	Best	$O(n \log n)$	$O(n)$
	Average	$O(n \log n)$	$O(n)$
	Worst	$O(n \log n)$	$O(n)$
InsertionSort	Best	$O(n)$	$O(1)$
	Average	$O(n^2)$	$O(1)$
	Worst	$O(n^2)$	$O(1)$

Tabella 1: Confronto tra algoritmi di ordinamento con inclusione dei casi migliori

## 4.6 Heap Sort

Heap Sort è un algoritmo di ordinamento basato su confronti che utilizza una struttura dati chiamata **Heap** (o coda di priorità). Un Heap è un albero binario completo in cui ogni nodo è maggiore (o minore) dei suoi figli. Un Heap può essere rappresentato come un array, dove il figlio sinistro di un nodo  $i$  si trova in posizione  $2i$ , mentre il figlio destro si trova in posizione  $2i + 1$ .

## Navigazione

La navigazione di un albero binario completo è molto semplice, infatti se si ha un nodo  $i$  si può calcolare il padre con la formula  $\lfloor i/2 \rfloor$ , il figlio sinistro con  $2i$  e il figlio destro con  $2i + 1$ .

### Proprietà Max-Heap

$$A[\text{parent}(i)] \geq A[i]$$

Dobbiamo quindi garantire che la proprietà di Max-Heap sia rispettata, e se non lo è, dobbiamo costruirlo con un **Build-Max-Heap** (complessità  $O(n)$ ). Questa proprietà andrà poi mantenuta utilizzando l'algoritmo **Max-Heapify** (complessità  $O(\log n)$ ). Infine possiamo ordinare l'array con l'algoritmo **Heap-Sort** (complessità  $O(n \log n)$ ). Procediamo quindi a scrivere l'algoritmo di Max-Heapify (heap-size rappresenta la lunghezza della lista):

---

#### Algorithm 10: Max-Heapify

---

**Data:** Un array  $A$  di interi, un indice  $i$ ,  $A.\text{heap-size}$  la lunghezza dell'array  
**Result:** Un array che rappresenta un albero binario completo

```
1 l = left(i) #(left: 2i e right: 2i + 1);
2 r = right(i);
3 largest = i;
4 if l ≤ A.heap-size and A[l] > A[i] then
5   | largest = l;
6 end
7 if r ≤ A.heap-size and A[r] > A[largest] then
8   | largest = r;
9 end
10 if largest ≠ i then
11   | swap(A[i], A[largest]);
12   | Max-Heapify(A, largest, A.heap-size);
13 end
```

---

Ora possiamo scrivere l'algoritmo di Build-Max-Heap:

---

#### Algorithm 11: Build-Max-Heap

---

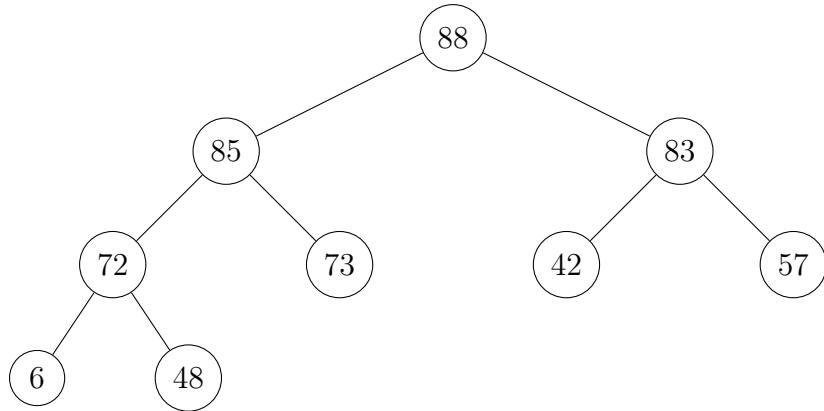
**Data:** Un array  $A$  di interi

**Result:** Un array che rispetta la proprietà di Max-Heap

```
1 A.heap-size = A.length;
2 for i = ⌊ A.length/2 ⌋ to 1 do
3   | Max-Heapify(A, i, A.heap-size);
4 end
```

---

Ecco la rappresentazione dell'albero completo dell'array [88, 85, 83, 72, 73, 42, 57, 6, 48], dove si può notare che il nodo padre è sempre maggiore dei nodi figli, e questa assunzione è il fulcro dell'algoritmo di Heap-Sort.



Quindi possiamo scrivere a scrivere l'algoritmo di Heap-Sort:

---

**Algorithm 12:** Heap-Sort

---

**Data:** Un array  $A$  di interi,  $n$  la lunghezza dell'array

**Result:** Un array ordinato

```

1 Build-Max-Heap(A);
2 for i = n to 2 do
3     swap(A[1], A[i]);
4     A.heap-size = A.heap-size - 1;
5     Max-Heapify(A, 1, A.heap-size);
6 end

```

---

Segue quindi l'implementazione in Python:

```

def max_heapify(A: list, i: int, heap_size: int):
    left_child = 2 * i + 1 # In python indici partono da 0
    right_child = 2 * i + 2
    larger_index = i

    if left_child <= heap_size and A[left_child] > A[larger_index]:
        larger_index = left_child
    if right_child <= heap_size and A[right_child] > A[larger_index]:
        larger_index = right_child
    print(A[i], A[larger_index])
    if larger_index != i:
        A[i], A[larger_index] = A[larger_index], A[i]
        max_heapify(A, larger_index, heap_size)

def build_max_heapify(A: list):
    heap_size = len(A) - 1
    for _ in range(len(A) // 2, -1, -1):
        max_heapify(A, _, heap_size)

def heap_sort(A: list):
    build_max_heapify(A)

```

```

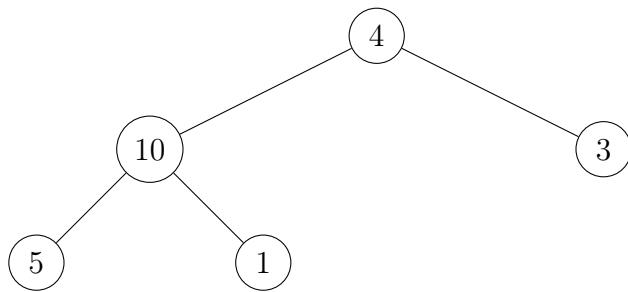
heap_size = len(A)-1
for _ in range(heap_size, 0, -1):
    A[0], A[_] = A[_], A[0] # ultimo messo alla fine e poi da non
        ↪ considerare
    heap_size -= 1
    max_heapify(A, 0, heap_size)
return A

```

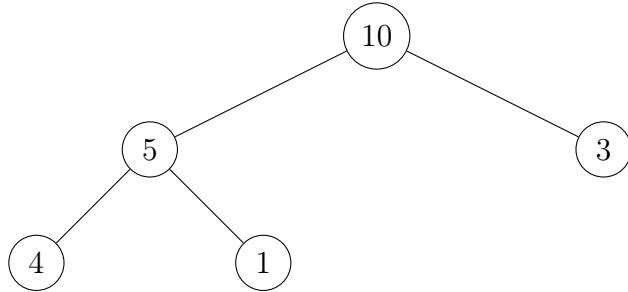
Le operazioni dell'algoritmo di Heap-Sort sono:

- A partire da un binario albero  $T$ , trasformarlo in un heap  $T'$
- Data una lista  $A$ , ordinarla attraverso l'algoritmo di Heap-Sort

**Definizione 4.1** (Heap). Un **max-heap** è un albero tale per cui per ogni nodo  $i$ , il suo valore  $v_i \geq \text{child}(v_i) \forall i$ . Ad esempio, dato un array  $[4, 10, 3, 5, 1]$ , il suo albero è:



Il suo albero **heap** è:



Quindi la funzione **Max-Heapify** prende in input un array  $A$  (albero) e un indice  $i$ , per cui il valore  $v_i$  è quanto più in basso. Dato che **Max-Heapify** parte da  $\lfloor \frac{n}{2} \rfloor$ , faremo circa  $\frac{n}{2}$  chiamate a **Max-Heapify**. Si dimostra per induzione che si passa da un piccolo heap (un nodo con due figli) a un grande heap (un nodo con due sottoalberi).

*Dimostrazione.* La chiamata a **Max-Heapify** con indice  $\lfloor \frac{n}{2} \rfloor$  è un heap, e lo stesso per tutti gli alberi di dimensione tre nel layer  $h - 1$  (dove  $h$  è l'altezza dell'albero). Una volta che abbiamo un sottoalbero che è un max-heap, possiamo costruire un max-heap più grande. Il numero di scambi è al più  $h$ , che è  $\leq \log n - \log i + 1 = \log \frac{n}{i} + 1$  operazioni  $\forall i$ .  $\square$

Analizziamo la complessità dell'operazione che costruisce un heap da un albero binario:

$$T(n) \leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \log \frac{n}{i} + 1 \leq \frac{n}{2} + \sum_{i=1}^{\frac{n}{2}} =$$

Operiamo un cambio di variabile  $k = \frac{n}{2}$

$$= k + \sum_{i=1}^k \log \frac{n}{i}$$

Vale che  $\sum_{i \in D} f(i) \leq \int_{D'} f(x) dx$ , con  $D \subseteq \mathbb{N}$  e  $D' \subseteq \mathbb{R}^+$ . Quindi:

$$\leq k + \int_1^k \log \frac{n}{x} dx \leq k + 2k - \log n = 3\frac{n}{2} - \log n \leq \frac{3n}{2} \in O(n)$$

Vediamo nel complesso l'iter di **Heap-Sort**: Prima costruisce l'heap con **Build-Max-Heap**

- Prendiamo la radice e scambiamo con una foglia
- Rimuoviamo nuova foglia
- Ricostriamo l'heap chiamando **Max-Heapify**

Al più di tutto questo ciclo avremo complessità  $O(n \log n)$ .

**Teorema 5.** *Quick Select randomizzato ha complessità attesa  $O(n)$*

*Dimostrazione.* □

**Lemma 4.4.** *Alla  $i$ -esima chiamata ricorsiva il running time sarà  $O(|S^t|)$ .*

**Lemma 4.5.** *Se l' $i$ -esimo pivot allora, supponendo che  $S^t$  sia ordinato (non lo è), dico che il pivot è buono se prende un elemento che ricade tra il primo quarto e l'ultimo quarto, e quindi si ha che  $|S_{t+1}| \leq \frac{3}{4}|S_t|$ . Quindi al termine l'albero finale avrà altezza  $O(\log n)$ , e la dimensione dell'array si ridurrà così velocemente che la complessità sarà effettivamente  $O(n)$ .*

*Dico che  $S^t$  è del  $j$ -esimo gruppo  $G_j$  se:*

$$\left(\frac{3}{4}\right)^{j+1} n < |S^t| \leq \left(\frac{3}{4}\right)^j n$$

**Remark** *Se il pivot è buono in  $S^t$  allora si verifica un cambio di gruppo (nella singola chiamata). Ad esempio:*

$$\begin{aligned} G_0 : \frac{3}{4}n &< |S^t| \leq n \\ G_1 : \frac{9}{16}n &< |S^t| \leq \frac{3}{4}n \\ \vdots \end{aligned}$$

Tuttavia *se il pivot non è buono, si potrebbe comunque verificare un cambio di gruppo (magari in più chiamate successive).*

**Lemma 4.6.** *Sia  $T_j$  il numero di chiamate ricorsive che si verificano nel gruppo  $G_j$ , allora:*

$$E[T_j] \leq 2$$

Utilizziamo il teorema 6:

$$\begin{aligned}
 E[T_j] &= \sum_{i=0}^{+\infty} \mathbb{P}(T_j \geq i) \\
 &\leq \sum_{i=0}^{+\infty} \mathbb{P}(\text{no pivot buoni in } i \text{ chiamate}) \\
 &\leq \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = 2
 \end{aligned}$$

Ricordiamo che per le serie geometriche si ha:

$$\sum_{i=0}^{+\infty} q^i = \frac{1}{1-q} \text{ se } |q| < 1$$

Per cui analizziamo come si sviluppa il processo per i gruppi  $G_j$ :

$$\begin{aligned}
 G_0 &= \begin{cases} X = \text{tempo di esecuzione} \\ X_j = \text{tempo di esecuzione in } G_j \\ E[X] = \sum E[X_j] \end{cases} \\
 G_1 &= \left\{ \leq c \cdot \left(\frac{3}{4}\right)^n n \cdot E[T_j] \right.
 \end{aligned}$$

Alla fine si ottiene:

$$E[X] \leq \sum_{j=0}^{+\infty} c \cdot \left(\frac{3}{4}\right)^j n \in O(n)$$

**Teorema 6.** Sia  $X > 0$  una variabile aleatoria:

$$+\infty > E[X] = \sum_{i=0}^{+\infty} \mathbb{P}(X \geq i)$$

## 5 Programmazione Dinamica

### 5.1 Un'approccio diretto: Fibonacci

La serie di fibonacci si può definire nel seguente modo:

$$f(n) = \begin{cases} 1 & \text{se } n \in \{0, 1\} \\ f(n-1) + f(n-2) & \text{altrimenti} \end{cases}$$

Abbiamo visto (se non lo abbiamo visto vedetelo), che la complessità dell'implementazione ricorsiva di Fibonacci è molto grande (esponenziale). Tuttavia, possiamo migliorare la complessità utilizzando un approccio dinamico, ottenendo una complessità lineare.

Infatti notiamo come la funziona di Fibonacci richieda solamente i valori di  $f(n-1)$  e  $f(n-2)$ , quindi possiamo memorizzare i valori di Fibonacci in un array e calcolare i valori successivi, per poi ripetere il processo fino a  $n$ .

Quindi possiamo scrivere un algoritmo che calcola il valore di Fibonacci in modo dinamico:

---

**Algorithm 13:** Fibonacci Dinamico

---

**Data:** Un intero  $n$   
**Result:** Il valore di Fibonacci di  $n$

```
1 if  $n \in \{0, 1\}$  then
2   | return 1;
3 end
4 current, previous = 1, 1;
5 for  $i = 1$  to  $n - 2$  do
6   | aux = current + previous;
7   | previous = current;
8   | current = aux;
9 end
10 return current;
```

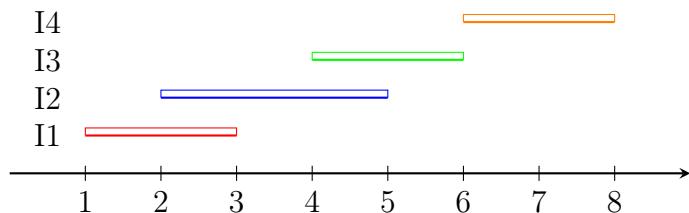
---

## 5.2 Interval Scheduling

Dato un insieme di  $n$  attività (che indichiamo come intervallo) e ognuna ha un tempo di inizio e fine ( $i \rightarrow (b(i), e(i))$ ), vogliamo trovare il sottoinsieme di attività compatibili tra loro e che massimizzino il numero di attività.

Per risolvere questo problema possiamo utilizzare un **approccio greedy**, ovvero scegliere l'attività che termina prima e che non si sovrappone con le altre attività. Lavoriamo quindi localmente senza avere una visione globale del problema (da cui il nome greedy). Possiamo descrivere l'algoritmo in questo modo:

1. Ordinare le attività in base al tempo di fine
2. Fino a che non abbiamo finito le attività:
  - (a) Scegliere l'attività che termina prima
  - (b) Rimuovere le attività che si sovrappongono



Ecco come si sviluppa l'algoritmo:

---

**Algorithm 14:** Interval Scheduling

---

**Data:** Un insieme di attività  $A$   
**Result:** Massimo numero di attività compatibili

```

1 A = sort(A);
2 S = {};
3 while A ≠ Ø do
4     a = A.pop(0);
5     S.add(a);
6     for a' ∈ A do
7         if a'.start < a.end then
8             A.remove(a');
9         end
10    end
11 end
12 return S;
```

---

L'algoritmo avrà complessità  $O(n \log n)$  dovuta alla fase di ordinamento, e lo spazio richiesto sarà  $O(n)$ .

**Lemma 5.1** (Greedy Stays Ahead). *Sia  $T$  la soluzione ottima e  $O$  la soluzione greedy, allora  $|T| \geq |O|$*

*Dimostrazione.* Chiamiamo  $t_1, t_2, \dots, t_l$  le attività della soluzione greedy e  $o_1, o_2, \dots, o_k$  le attività della soluzione ottima. Dobbiamo quindi dimostrare che  $e(t_i) \leq e(o_i)$ . Procediamo per induzione:

- **Base:**  $i = 1$ , allora  $e(t_1) \leq e(o_1)$  è vero
- **Ipotesi Induttiva:** Supponiamo che per  $i - 1$  vale  $e(t_{i-1}) \leq e(o_{i-1})$  e dimostriamo che  $e(t_i) \leq e(o_i)$
- **Passo Induttivo:** Guardiamo quindi l'attività  $o_i$  scelta dall'algoritmo ottimo, e sappiamo che a sua volta l'attività  $t_{i-1}$  è compatibile con  $o_i$ , data l'ipotesi induttiva. Quindi  $e(t_i) \leq e(o_i)$

□

**Teorema 7** (Dimostrazione del Prof. Russo).  *$\forall I |S(I)| = |OPT(I)|$  con  $S$  la soluzione dell'algoritmo greedy e  $OPT$  la soluzione ottima.*

**Remark**

1. *Supponiamo di aver selezionato un intervallo  $I$*
2. *Questo implica che dovremo rimuovere tutti gli intervalli  $J \cap I \neq \emptyset$*
3. *Fra tutti  $J$  del tipo  $J \cap I \neq \emptyset$ , soltanto un intervallo sarà selezionato da  $OPT$ :  $\forall J \cap I \neq \emptyset \Rightarrow s_j < e_i$*

*Questo tipo di dimostrazione è detta **charging argument***

*Dimostrazione.* Consideriamo la soluzione  $OPT = [I_1, \dots, I_k]$  e un intervallo  $I \in S$ , dove  $S$  è la soluzione dell'algoritmo greedy. Definiamo quindi un'applicazione  $h$  tale che:

$$h : OPT \rightarrow S \Rightarrow h(I') = I \text{ con } e_I \leq e_{I'}$$

dove  $I$  è l'intervallo che interseca  $I'$  e si conclude prima degli altri  $J \cap I' \neq \emptyset$ . Proviamo quindi l'esistenza di  $I \in S$ :

Supponiamo che non esista  $\forall I \in S$   $h(I') \neq I$ , allora  $I'$  non interseca  $S \setminus \{I\}$  e quindi l'algoritmo greedy avrebbe selezionato  $I'$ .

Dobbiamo quindi provvedere a dimostrare che  $h$  è un'applicazione biiettiva, ovvero che  $h$  è iniettiva e suriettiva.

- **Unicità**

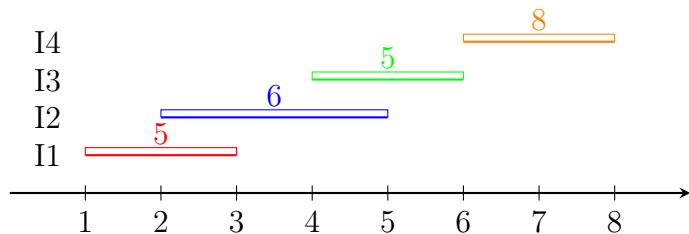
- **Iniettività:**  $\forall I_1, I_2 \in OPT$  allora  $h(I_1) = h(I_2)$ . Supponiamo che  $\exists I_1, I_2$  ( $e_1 < e_2$ ) tale per cui  $h(I_1) = h(I_2) = I$ . Consideriamo  $e$ , il tempo di fine di  $I$ , quindi per la definizione di  $h$  si ha che  $e \leq e_1$  e  $e \leq e_2$ , e quindi  $I$  finisce prima di  $I_1$ , e per cui  $I$  non potrebbe intersecarsi con  $I_2$  e l'algoritmo greedy avrebbe selezionato  $I_2$ .

□

Esiste anche una variante del problema in cui ogni attività ha un peso associato, e si vuole quindi massimizzare il peso totale delle attività compatibili. Questo problema è noto come **Weighted Interval Scheduling**, e adesso vediamo come approcciarlo.

### 5.3 Weighted Interval Scheduling

Il problema di Weighted Interval Scheduling è una variante del problema di Interval Scheduling in cui ogni attività ha un peso associato.



Vediamo quindi come procedere passo passo:

1. Ordiniamo le attività in base al tempo di fine ( $e(i) \rightarrow e(i) \leq e(j) \forall i < j$ )
2. Costruiamo una funzione  $p(i)$ :

$$p(i) = \arg \max_{j=1, \dots, i-1} \{e(i) \mid i \text{ e } j \text{ compatibili}\}$$

3. Devo quindi definire una funzione  $\varphi(i)$  che ritorni il valore della soluzione ottima sugli intervalli  $\{1, 2, \dots, i\}$

$$\varphi(n) = \max\{w(n) + \varphi(p(n)), \varphi(n-1)\} \text{ (equazione di Bellan-Ford)}$$

Dobbiamo quindi utilizzare un vettore  $M$  per memorizzare i valori di  $\varphi(i)$  e sapremo che la soluzione ottima sarà  $M(n)$ . Questo può essere dimostrato per induzione su  $i$ . Per ricorrere alla soluzione ottima, si dovrà fare un'operazione di backtracking su  $M$ .

Si noti che l' $\arg \max$  indica l'indice dell'attività massima compatibile con  $i$ .

## 5.4 KnapSack Problem

Il problema dello zaino è un problema di ottimizzazione combinatoria, in cui si ha uno zaino con una capacità massima  $B$  e un insieme di  $n$  oggetti, ognuno con un valore  $v(i)$  e un peso  $c(i)$  ( $c_i \in \mathbb{N}$   $v_i \geq 0$ ). L'obiettivo è trovare il sottoinsieme di oggetti che massimizzano il valore totale, senza superare la capacità dello zaino. In termini matematici:

$$\max_S \sum_{i \in S} v(i) \text{ t.c. } \sum_{i \in S} c(i) \leq B$$

Anche in questo caso possiamo sfruttare la programmazione dinamica per risolvere il problema.

Definiamo quindi la funzione  $\varphi(i, b)$  che è il valore massimo che si può ottenere con i primi  $i$  oggetti e un budget  $b$ .

Possiamo quindi scrivere la funzione di ricorrenza:

$$\varphi(i, b) = \begin{cases} \varphi(i - 1, b) & \text{se } c(i) > b \\ \max\{\varphi(i - 1, b), v(i) + \varphi(i - 1, b - c(i))\} & \text{altrimenti} \end{cases}$$

Il running time dell'algoritmo è  $O(nB)$ , e la complessità spaziale è  $O(nB)$ .

Anche in questo caso possiamo utilizzare un vettore  $M$  per memorizzare i valori di  $\varphi(i, b)$  e sapremo che la soluzione ottima sarà  $M(n, B)$ . Questo può essere dimostrato per induzione su  $i$ . Per ricorrere alla soluzione ottima, si dovrà fare un'operazione di backtracking su  $M$ .

## 6 Grafi

**Definizione 6.1.** Un grafo è una coppia  $G = (V, E)$  dove  $V$  è un insieme di vertici e  $E$  è un insieme di archi. Un arco è un insieme di due vertici, ovvero  $e = \{u, v\}$  con  $u, v \in V$ . Quindi  $E \subseteq V^2$ . Esistono due tipi di grafi principali:

**Definizione 6.2** (Grafo indiretto). Un grafo è indiretto se gli archi non hanno una direzione

$$(u, v) \in E \implies (v, u) \in E$$

**Grafo diretto:** un grafo in cui gli archi hanno una direzione. Tutti i grafi che non sono indiretti sono diretti.

**Definizione 6.3** (Grafo pesato). Un grafo è pesato se gli archi hanno un peso associato ovvero  $G = (V, E, \omega)$  dove  $\omega : E \rightarrow \mathbb{R}$

**Definizione 6.4** (Cammino). Dato un grafo  $G = (V, E)$ , definiamo  $\Pi$  come:

$$\Pi = \underbrace{\{e_1, e_2, \dots, e_k\}}_{e_i \in E} \text{ è un cammino se } \forall i \in \{1, \dots, k-1\} e_i = (u, v) e_{i+1} = (\omega, v) \implies v = \omega$$

**Definizione 6.5** (Ciclo). Dato un grafo  $G = (V, E)$ ,  $\Pi$  un cammino, allora  $\Pi$  è un ciclo se  $e_1 = (u, v)$  e  $e_k = (\omega, u)$

**Definizione 6.6** (Grafo indiretto connesso). Un grafo indiretto è connesso se per ogni coppia di nodi  $u, v \in V$  esiste un cammino tra  $u$  e  $v$  che li congiunge. Un nodo è connesso a se stesso.

**Proposizione 8.** *La connessione di un grafo indiretto è una relazione di equivalenza.*

*Dimostrazione.*

$$u \sim v \iff \exists \pi : u \rightarrow v$$

- **Riflessiva:** si perché un nodo è connesso a se stesso
- **Transitiva:** se  $u \sim v$  e  $v \sim w \implies u \sim w$  dato che esiste un cammino che li congiunge
- **Simmetrica:** se  $u \sim v \implies v \sim u$  e questo vale per la definizione di grafo indiretto

□

Per i grafi diretti si parla di **fortemente connesso**.

**Definizione 6.7** (Grafo fortemente connesso). Un grafo diretto  $G = (V, E)$  è fortemente connesso se  $\forall u, v \in V$  (con  $u \neq v$ ):

1.  $\exists \pi : u \rightarrow v$
2.  $\exists \pi' : v \rightarrow u$

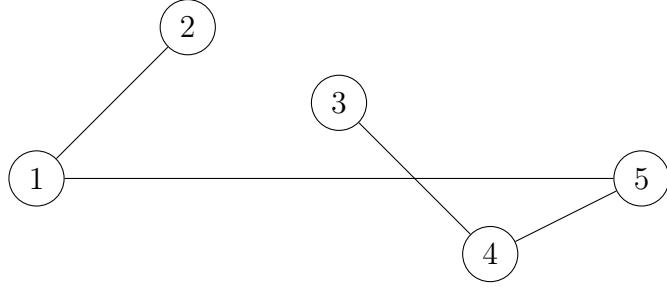
## 6.1 Rappresentazione dei Grafi

Vediamo ora come si può rappresentare un grafo in maniera computazionale.

**I opzione** Rappresentare la lista degli archi del grafo, ovvero una lista di tuple  $(u, v)$  che rappresentano gli archi del grafo. Tuttavia questa rappresentazione non è efficiente per trovare i vicini di un nodo (complessità  $O(n)$ ).

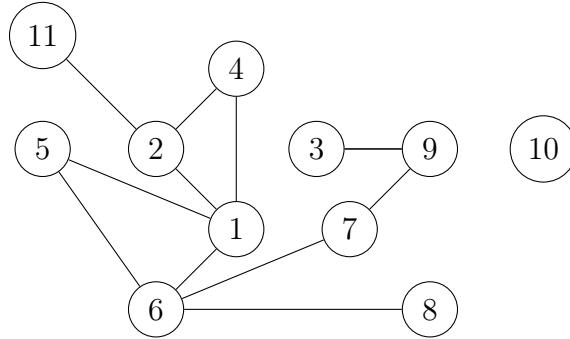
**II opzione** Lista di adiacenza, ovvero un dizionario in cui la chiave è il nodo e il valore è la lista dei nodi adiacenti. Questa rappresentazione è molto efficiente per trovare i vicini di un nodo (complessità  $O(1)$ ).

**III opzione** Matrice di adiacenza, ovvero una matrice  $A$  di dimensione  $n \times n$  in cui  $A_{ij} = 1$  se esiste un arco tra  $i$  e  $j$ , altrimenti  $A_{ij} = 0$ . Questa rappresentazione è molto efficiente per trovare se esiste un arco tra due nodi (complessità  $O(1)$ ), ma è inefficiente per trovare i vicini di un nodo (complessità  $O(n)$ ). Inoltre ha spazio di memoria  $O(n^2)$ . Si usa la matrice di adiacenza per rappresentare le catene di Markov.



## 6.2 Visite in Grafi

I grafi saranno da intendersi indiretti.



**Visita in ampiezza (BFS):** si visita il grafo in modo iterativo, visitando prima i nodi vicini al nodo di partenza, e poi i nodi più lontani. Dobbiamo quindi utilizzare una coda (FIFO) per memorizzare i nodi da visitare.

---

### Algorithm 15: BFS algorithm

---

**Data:** Un grafo  $G = (V, E)$  e un nodo di partenza  $s$

**Result:** I nodi visitati

```

1 starting node = s;
2 Q queue, enqueue s in Q;
3 while Q ≠ Ø do
4   u = dequeue Q;
5   visit u;
6   for v ∈ N(u) do
7     if v not visited and v not in Q then
8       enqueue v in Q;
9     end
10   end
11 end

```

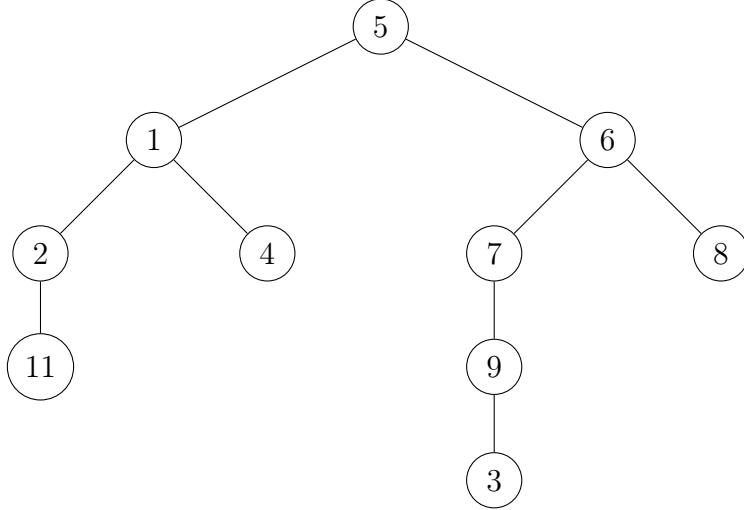
---

**Remark** La lista dei nodi visitati contiene tutti e soli gli elementi visitabili dallo starting node. Questo implica che se non tutti i nodi sono visitabili, allora il grafo **non è connesso**. Quindi il grafo può essere quoziato dalla relazione di equivalenza della connessione.

Se ipotizziamo di scorrere il grafo sopra rappresentato con un algoritmo BFS, e ipotizziamo che il nostro starting node sia 5, otterremo la seguente sequenza di nodi visitati:

$$5 \rightarrow 1 \rightarrow 6 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 11 \rightarrow 9 \rightarrow 3$$

Possiamo rappresentare l'albero di visita in questo modo:



**Visita in profondità (DFS):** si visita il grafo in modo ricorsivo, visitando prima i nodi più lontani dal nodo di partenza. Dobbiamo quindi utilizzare uno stack (LIFO) per memorizzare i nodi da visitare.

---

#### Algorithm 16: DFS algorithm

---

**Data:** Un grafo  $G = (V, E)$  e un nodo di partenza  $s$   
**Result:** I nodi visitati

```

1 starting node = s;
2 P stack, push s in P;
3 while P ≠ ∅ do
4     u = pop P;
5     visit u;
6     for v ∈ N(u) do
7         if v not visited and v not in S then
8             push v in S;
9         end
10    end
11 end
  
```

---

Anche in questo caso la lista dei nodi visitati contiene tutti e soli gli elementi visitabili dallo starting node. La complessità di BFS e DFS è  $O(m)$ , dove  $m$  è il numero di archi.

Se ipotizziamo di scorrere il grafo sopra rappresentato con un algoritmo DFS, e ipotizziamo che il nostro starting node sia 5, otterremo la seguente sequenza di nodi visitati:

$$6 \rightarrow 8 \rightarrow 7 \rightarrow 9 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 11 \rightarrow 5$$

### 6.3 Shortest Path

Ci è dato un grafo  $G = (V, E)$ . In un grafo pesato la distanza tra due nodi è la somma dei pesi degli archi che li congiungono, e la distanza minima tra due nodi è detta **shortest path**.

L'algoritmo dovrà avere in input il grafo, il nodo di partenza  $u \in V$  e il nodo di arrivo  $t \in V$  e dovrà restituire in output  $d(u, t) \forall u \in V$

**Definizione 6.8.** Dato  $\pi$  cammino  $\pi = (e_1, e_2, \dots, e_l)$  il costo di un cammino  $\pi$  è data da:

$$c(\pi) = \sum_{i=1}^l c(e_i)$$

**Definizione 6.9.** La distanza tra due nodi  $u$  e  $v$  è data da:

$$d(u, v) = \min\{c(\pi) \mid \pi \text{ cammino da } u \text{ a } v\}$$

**Remark** Noi indichiamo con  $n = |V|$  e  $m = |E|$ .

Vediamo come il problema del shortest path può essere risolto con un algoritmo greedy chiamato **algoritmo di Dijkstra**.

### 6.3.1 Algoritmo di Dijkstra

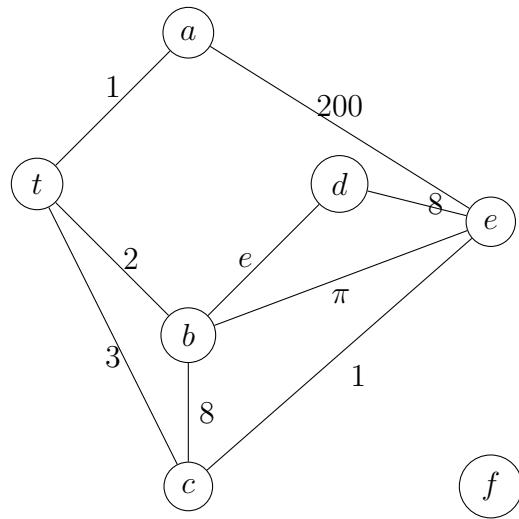


Figura 6: Esempio di grafo pesato

L'intuizione base è quella di costruire un albero di cammini minimi, e ad ogni passo aggiungere il nodo con il cammino minimo.

---

#### Algorithm 17: Dijkstra Algorithm

---

**Data:** Un grafo pesato  $G = (V, E)$ , un nodo di partenza  $t$

**Result:** La distanza minima tra  $s$  e tutti gli altri nodi

```

1 d[u] = +∞ ∀u ∈ V;
2 d[t]=0;
3 S = {t};
4 while S ≠ V do
5   u = argmin{d[v] ∀v ∈ V \ S};
6   d'(u) = minv∈S{d(v) + c(v, u)};
7   d[u] = d'(u);
8   add u to S;
9 end
  
```

---

Si noti che se  $(u, v) \notin E \implies c(u, v) = +\infty$  e  $d'$  è una funzione ausiliaria. Procediamo quindi a dimostrare la correttezza dell'algoritmo.

**Teorema 9.** Se la funzione  $c$  è non negativa, allora l'algoritmo di Dijkstra restituisce la distanza minima tra il nodo di partenza e tutti gli altri nodi.

*Dimostrazione.* Dimostriamo per induzione sulla cardinalità di  $S$ :

- **Base:**  $|S| = 1$ , allora  $S = \{t\}$  e  $d[t] = 0$
- **Ipotesi Induttiva:** Supponiamo che per  $|S| = i - 1$  vale la proprietà
- **Passo Induttivo:** chiamiamo  $u$  il nodo che vogliamo aggiungere e  $v$  il nodo che minimizza la distanza. Quindi l'algoritmo dovrà dare che:

$$d[u] = d[v] + c(v, u)$$

Bisogna quindi dimostrare che quello che abbiamo calcolato è effettivamente il minimo. Supponiamo per assurdo che esista un cammino  $\pi'$  più corto di  $\pi$ , quindi  $c(\pi) > c(\pi')$ . Sia  $w$  il primo nodo tale per cui si ha il minimo, quindi:

$$c(\pi') = d[w] + c(\tilde{\pi}) \geq d(u)$$

Quindi per definizione di greedy il nodo  $u$  è il minimo (ha deciso di mettere  $u$  al posto del nodo  $z$ , necessario a raggiungere  $u$ ), l'assunzione per assurdo vuole che esista un cammino più corto  $\pi'$ . Ad un certo punto si avrà che:

$$c(\pi') = \underbrace{d[w] + c(z, w)}_{\geq d[v] + c[u, v] = c(\pi)} + \underbrace{c(\tilde{\pi})}_{\geq 0}$$

che è un assurdo dato che  $c(\pi) > c(\pi')$

□

Studiamo ora la complessità dell'algoritmo di Dijkstra.

Studiamoci una modalità naïve dell'algoritmo:

- Trovare il minimo fissato un nodo:  $O(m)$
- Cercarlo per ogni nodo:  $O(nm)$

C'è un modo più efficiente per trovare il minimo, ovvero utilizzare una coda di priorità. Possiamo farlo costruendo un min-heap ( $O(n \log n)$ ) e mantenendo un vettore  $d$  con le distanze.

Dobbiamo quindi effettuare tre operazioni:

- FIND per trovare il minimo:  $O(1)$  (uso un dizionario che si tiene in memoria la posizione del nodo nel min-heap)
- DELETE per eliminare il minimo:  $O(\log n)$
- ADD per aggiungere un nodo:  $O(\log n)$

Con questa implementazione riusciamo a ridurre la complessità dell'algoritmo a  $O(m \log n)$ , dato che ogni arco sarà potenzialmente utilizzato per aggiornare la distanza di un nodo. L'algoritmo di Dijkstra fallisce tuttavia nel momento in cui ci sono archi negativi, in quanto potrebbe non terminare.

Sul grafo di Figura 6 l'algoritmo di Dijkstra restituirà la seguente distanza minima tra il nodo di partenza  $t$  e tutti gli altri nodi:

$$d = [1, 2, 3, e + 2, 4, \infty, 0] \quad S = \{t, a, b, c, d, e, f\}$$

Un'altra variante dell'algoritmo di Dijkstra è l'algoritmo di Bellman-Ford, che è in grado di gestire archi negativi attraverso l'uso della programmazione dinamica.

## 6.4 Algoritmo di Bellman-Ford

**Lemma 6.1.** *Dato un grafo  $G$  senza cicli negativi, allora il cammino minimo tra due nodi è composto da al massimo  $n - 1$  archi.*

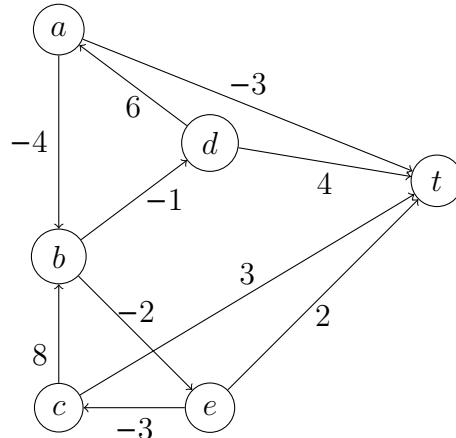
Posso quindi definire la funzione:

$\varphi(i, v) = \text{costo del cammino minimo tra } v \text{ e } t \text{ di lunghezza al più } i$

$$\varphi(i, u) = \min \begin{cases} \varphi(i - 1, u) \\ \varphi(i - 1, v) + c(v, u) \quad \forall v : (v, u) \in E \end{cases}$$

Inizialmente si ha che  $\varphi(0, u) = +\infty \quad \forall u \neq t$  e  $\varphi(i, t) = 0 \quad \forall i$ .

**Esempio 6.1.** Supponiamo di avere il grafo seguente:



Allora l'algoritmo di Bellman-Ford costruirà la seguente tabella:

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	$\infty$	-3	-3	-4	-6	-6
b	$\infty$	$\infty$	0	-2	-2	-2
c	$\infty$	3	3	3	3	3
d	$\infty$	4	3	3	2	0
e	$\infty$	2	0	0	0	0

La complessità dell'algoritmo di Bellman-Ford è  $O(n^3)$  e la complessità spaziale è  $O(n^2)$ .

**Remark** Si osservi che lo spazio di memoria può essere ridotto a  $O(n)$  se si memorizzano solo i valori di  $\varphi(i, v)$  e  $\varphi(i - 1, v)$ , tuttavia in questo caso diventa più ostico fare backtracking per ricostruire il cammino minimo.

**Remark** L'algoritmo di Dijkstra può sfruttare un vettore per salvare i cammini minimi. Nella coda di priorità vengono salvate:

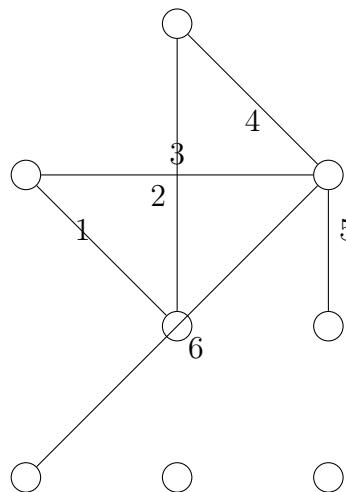
- l'identità del nodo
- il valore della distanza
- il nodo precedente

## 6.5 Minimum Spanning Tree

Dato un grafo  $G = (V, E)$  connesso e indiretto e una funzione  $c : E \rightarrow \mathbb{R}$ . Assumiamo che  $c(e) \neq c(e') \forall e, e' \in E$ . Un **minimum spanning tree** è un sottoinsieme degli archi che connette tutti i nodi senza formare cicli e la somma dei pesi degli archi è minima. Quindi l'obiettivo è:

$$\min \sum_{e \in T} c(e) \text{ t.c. } T \subseteq E(\text{connesso}) \quad \forall v \in V \exists e \in T : v \in e \text{ (quell'arco copre il nodo)}$$

Questo problema è ben posto in quanto il sottoinsieme degli archi che connette tutti i nodi è un albero.



**Definizione 6.10 (Albero).**  $T = (V, E)$  è un albero se:

1.  $T$  è connesso
2.  $T$  non ha cicli (ciclo è un cammino non banale che parte e finisce nello stesso nodo)

**Proposizione 10.** Il problema del Minimum Spanning Tree ammette una soluzione ottima  $T^*$  che è un albero.

*Dimostrazione.* Per definizione di MST il grafo è connesso. Supponiamo per assurdo che  $T^*$  non sia un albero, quindi avrà un ciclo. Se togliamo un arco da un ciclo otteniamo un albero, quindi il nostro albero non era minimo. Quindi la soluzione ottima è un albero.  $\square$

Algoritmi che risolvono il problema del Minimum Spanning Tree sono l'algoritmo di Kruskal e l'algoritmo di Prim, che sono algoritmi greedy.

### 6.5.1 Algoritmo di Prim

L'algoritmo di Prim è un algoritmo greedy che costruisce l'MST partendo da un nodo arbitrario e aggiungendo ad ogni passo l'arco con il peso minore che connette un nodo già visitato ad uno non visitato.

---

**Algorithm 18:** Prim Algorithm

---

**Data:** Un grafo pesato  $G = (V, E)$   
**Result:** Un Minimum Spanning Tree

```

1 S = {v};
2 T = {} # Archi della soluzione;
3 A = V \ S;
4 while S ≠ V do
5   | find e ∈ ∂(S) con peso minimo;
6   | e = (u, v), u ∈ S, v ∈ A;
7   | add e to T;
8   | add v to S;
9   | remove v from A;
10 end

```

---

L'implementazione Naive dell'algoritmo di Prim ha complessità  $O(n \cdot m)$ , ma possiamo ridurla a  $O(m \log n)$  utilizzando una coda di priorità (un min heap) per trovare l'arco con il peso minimo.

**Definizione 6.11.** Dto un grafo  $G = (V, E)$  e  $S ⊆ V$  definisco la frontiera di  $S$  come:

$$\partial(S) = \{e = (u, v) ∈ E \mid u ∈ S, v ∈ V \setminus S\}$$

### 6.5.2 Algoritmo di Kruskal

L'algoritmo di Kruskal è un algoritmo greedy che costruisce l'MST partendo dagli archi con il peso minore.

---

**Algorithm 19:** Kruskal Algorithm

---

**Data:** Un grafo pesato  $G = (V, E)$   
**Result:** Un Minimum Spanning Tree

```

1 E = sort E by weight;
2 T = {};
3 for e ∈ E do
4   | if T ∪ {e} does not contain a cycle then
5   |   | add e to T;
6   | end
7 end

```

---

A differenza di Prim, l'algoritmo di Kruskal crea una foresta di alberi che alla fine diventerà un albero (di cardinalità  $n - 1$ ).

Dimostriamo adesso la correttezza degli algoritmi di Prim e Kruskal.

**Teorema 11.** Sia  $G = (V, E)$  un grafo indiretto pesato e sia  $T^*$  un suo MST. Sia  $S ⊂ V$  non banale e sia  $e ∈ \partial(S)$  con peso minimo. Allora  $e ∈ T^*$ .

*Dimostrazione.* Supponiamo per assurdo che  $e \notin T^*$  (con  $e$  l'arco che connette  $(u, v)$ ). Sia  $e'$  l'arco che connette  $(u', v')$  (dato che  $u$  e  $v$  sono connessi il cammino dovrà transitare da un altro arco  $e'$ ). E dato che  $e$  è l'arco con il peso minimo, allora  $c(e) \leq c(e')$ . Quindi possiamo costruire un nuovo MST  $T' = T^* \cup \{e\} \setminus \{e'\}$ . Quindi  $c(T') \leq c(T^*)$ , ma  $T'$  è un MST (dato che i nodi presenti in  $T^*$  (tranne  $u'$  e  $v'$ , dato che  $e'$  non è considerato) erano connessi, e  $e'$  viene sostituito da  $e$ ). Quindi abbiamo trovato un MST con peso minore, assurdo. Quindi presi due nodi  $w, z$  esisterà un cammino minimo  $\pi$  tra di essi che passerà per l'arco  $e \in \partial(S)$  se  $e' \in \pi$ .  $\square$

**Corollary 12** (Correttezza di Prim). *Sia  $T$  l'output dell'algoritmo di Prim, e sia  $T^*$  un MST. Allora  $T = T^*$ .*

*Dimostrazione.* Per il teorema precedente sappiamo che se  $e \in \partial(S)$  con peso minimo allora  $e \in T^*$ . Quindi l'algoritmo di Prim sceglie sempre l'arco con il peso minimo, quindi  $T = T^*$ .  $\square$

**Corollary 13** (Correttezza di Kruskal). *Sia  $T$  l'output dell'algoritmo di Kruskal, e sia  $T^*$  un MST. Allora  $T = T^*$ .*

*Dimostrazione.* Kruskal mantiene una foresta di alberi, e ad ogni passo aggiunge l'arco con il peso minimo che non forma un ciclo.  $\square$

