Algoritmi in Python

Sergio Greco

May 1, 2023

Chapter 1

Richiami di Matematica

1.1 Insiemi, relazioni e strutture di base

Relazioni

Una relazione su due domini A e B è un sottoinsieme del prodotto cartesiano $A \times B$.
Una relazione $R \subseteq A \times A$ è

- riflessiva se $aRa\ \forall a \in A$,
- simmetrica se aRb implica $bRa\ \forall a,b \in A$,
- transitiva se aRb e bRc implicano $aRc\ \forall a,b,c \in A$,
- antisimmetrica se aRb e bRa implicano $a = b$, $\forall a,b \in A$.

Una relazione che é riflessiva, antisimmetrica e transitiva é detta *ordine parziale*. Un insieme sul quale é definito un ordine parziale é detto insieme parzialmente ordinato. Un ordine parziale é detto totale o lineare se $\forall a,b \in A$ vale solo una delle due relazioni aRb o bRa.

Funzioni

Funzione $f : D \to C$ é una relazione su D e C tale che $f(d_1) \neq f(d_2)$ implica $d_1 \neq d_2$.

1.2 Notazione asintotica

Strict bounds:

- $O(g(n)) = \{f(n) \mid \exists c, n_0$ tale che $0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$
- $\Omega(g(n)) = \{f(n) \mid \exists cn_0$ tale che $0 \leq c \cdot g(n) \leq f(n)$
  $\forall n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0$ tale che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0\}$

Non Strict bounds:

- $o(g(n)) = \{f(n) \mid \forall c, \exists n_0$ tale che $0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$
- $\omega(g(n)) = \{f(n) \mid \forall c, \exists n_0$ tale che $0 \leq c \cdot g(n) \leq f(n),\ \forall n \geq n_0\}$

Proprietà:

- $f(n) \in o(g(n))$ implica che $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- $f(n) \in \omega(g(n))$ implica che $lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

1.2.1 Elementi di matematica combinatorica

Dato insieme S tale che $|S| = n$,

- il numero di tuple (o stringhe) su S di lunghezza k é n^k.
- Il numero di permutazioni di S (stringhe con n elementi distinti) é $n!$
- Il numero di sottoinsiemi di cardinalità $k \leq n$ é $\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} = \left(\begin{array}{c} n \\ n-k \end{array} \right)$

1.2.2 Principio di Induzione

L'insieme dei numeri naturali é dato da $\mathbb{N} = \{0, 1, 2, 3, ...\}$. $\mathbb{N}$ gode di importanti proprietá . Una delle piú importanti é la seguente.

Assioma 1 (Principio di Induzione)**.** . *Supponiamo che $S \subseteq \mathbb{N}$ sia un sottoinsieme di $\mathbb{N}$ che soddisfi le due seguenti proprietá: i) $0 \in S$, ii) $n \in S \Rightarrow n+1 \in S$. Allora $S = \mathbb{N}$.*

Il Principio di Induzione ci consente di dimostrare il seguente cruciale teorema relativo alle dimostrazioni per induzione.

Teorema 1 (Dimostrazioni per induzione)**.** . *Supponiamo che a ogni numero naturale n sia associata una proprietà (i.e. formula logica) $P(n)$. Supponiamo che le due seguenti proprietà siano soddisfatte: i) $P(0)$ è vera. ii) $P(n) \Rightarrow P(n+1)$. Allora $P(n)$ è vera per ogni $n \in \mathbb{N}$.*

Dimostrazione. Definiamo $S = \{n \in N : P(n)\}$. Per prima cosa, i) implica che $0 \in S$. Pertanto S soddisfa l'ipotesi i) del Principio di Induzione. Inoltre ii) implica che $n \in S \Rightarrow n++1 \in S$, e pertanto S soddisfa anche l ipotesi ii) del Principio di Induzione. Ma allora il Principio di Induzione implica $S = N$, e pertanto, per come S é definito, $P(n)$ é vera per ogni $n \in N$. □

Vediamo qualche applicazione.

Teorema 2 (Disuguaglianza di Bernoulli)**.** *Per ogni $n \in N$ e per ogni $a \geq 0$ abbiamo la seguente proposizione $P(n) = (1+a)^n \geq 1+na$.*

Dimostrazione. P(1) è ovviamente vera, visto che si riduce all'uguaglianza $1+a \geq 1+a$. Supponiamo ora che $P(n)$ sia vera. Abbiamo $P(n+1) = (1+a)^{n+1} = (1+a)^n(1+a) \geq (1+na)(1+a) = 1+na+a+a^2 \geq 1+(n+1)a$. Si osservi che abbiamo usato le ipotesi che $P(n)$ sia vera e che $a^2 \geq 0$. □

Teorema 3 (Somma aritmetica)**.** .

$$\Sigma_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Dimostrazione. Per $n = 1$ abbiamo $P(1) : \Sigma_{n=1}^{1} i = 1$. Supponiamo ora che $P(n)$ sia vera. Allora abbiamo che $\Sigma_{i=1}^{n+1} i = \Sigma_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + n + 1 = \frac{n^2+n+2n+2}{2} = \frac{(n+1)(n+2)}{2}$. □

Chapter 2

Funzioni elementari

2.1 La funzione fattoriale

La funzione fattoriale, denotata da $fact$, va dall'insieme dei numeri naturali all'insieme dei numeri naturali: $fact : N \to N$. Solitamente il fattoriale di un numero n viene indicato da $n!$.

Definition 1. *Funzione fattoriale:*

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & if\ n > 0 \end{cases}$$

Una definizione equivalente è la seguente:

$$n! = \begin{cases} 1 & if\ n = 0 \\ \Pi_{i=1}^{n} i & if\ n > 0 \end{cases}$$

- $n! = o(n^n)$
- $n! = \omega(n)$

2.1.1 Codice Python

Versione iterativa

```
1 def fact(n):
2     f = 1
3     for i in range(1,n+1):
4         f = f * i
5     return f
```

Listing 2.1: Funzione fattoriale - ersione iterativa

Complessità Temporale $T_{fact}(n) = \Theta(n)$

Complessità Spaziale $S_{fact}(n) = \Theta(1)$

Versione ricorsiva

```
def factR(n):
    if n <= 1:
        return 1
    else:
        return n * factR(n-1)
```

Listing 2.2: Funzione fattoriale - versione ricorsiva

Complessità Temporale

$$T_{factR}(n) = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(1) + \Theta(1)\, T_{factR}(n-1) & se\ n > 1 \end{cases} = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(n) & se\ n > 1 \end{cases} = \Theta(n)$$

Complessità Spaziale

$$S_{factR}(n) = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(1) + \Theta(1)\, S_{factR}(n-1) & se\ n > 1 \end{cases} = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(n) & se\ n > 1 \end{cases} = \Theta(n)$$

Programma principale

```
n = -1
while n < 0:
    n = input("Inserire numero naturale: ")
    if n >= 0:
        print("Il fattoriale di " + str(n) + " e': " + str(fact(n)))
```

Listing 2.3: Programma principale

2.2 Numeri di Fibonacci

Definition 2. *La successione di Fibonacci indicata con F_n o $Fib(n)$, denota una successione di numeri interi in cui ciascun numero è la somma dei due precedenti, eccetto i primi due che sono$F_0 = 0$ e $F_1 = 1$:*

$$F_n = \begin{cases} n & if\ n \leq 1 \\ F_{n-1} + F_{n-2} & if\ n > 1 \end{cases}$$

Gli elementi F_n sono anche detti numeri di Fibonacci. I primi termini della successione di Fibonacci sono: 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144 , 233 , ...

2.2.1 Codice Python

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Listing 2.4: Funzione di fibonacci - ersione ricorsiva (non lineare)

$$T_{fib}(n) = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(1) + \Theta(1)\,(T_{fib}(n-1) + T_{fib}(n-2)) & se\ n > 1 \end{cases} = \begin{cases} \Theta(1) & se\ n \leq 1 \\ o(2^n) & se\ n > 1 \end{cases} = o(2^n)$$

$$S_{fib}(n) = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(1) + \Theta(1)\,max(S_{fib}(n-1) + S_{fib}(n-2)) & se\ n > 1 \end{cases} = \begin{cases} \Theta(1) & se\ n \leq 1 \\ \Theta(n) & se\ n > 1 \end{cases} = \Theta(n)$$

```python
def fib1(n):
    F = [0,1]
    for i in range(2,n+1):
        F.append(F[i-1]+F[i-2])
    return F[n]
```

Listing 2.5: Versione iterativa con lista

Complessità Temporale $T_{fib1}(n) = \Theta(1) + n\,\Theta(1) = \theta(n)$

Complessità Spaziale $S_{fib1}(n) = \Theta(1) + \theta(1) + (n-1)\theta(1) = \Theta(n)$

```python
def fib2(n):
    if n == 0:
        return 0
    F = [0,1]
    for i in range(2,n+1):
        F[0], F[1] = F[1], F[0]+F[1]
    return F[1]
```

Listing 2.6: Versione iterativa senza lista

Complessità Temporale $T_{fib2}(n) = \Theta(1) + n\,\Theta(1) = \theta(n)$

Complessità Spaziale $S_{fib2}(n) = \Theta(1) + \Theta(1) + \theta(1) = \Theta(1)$

```python
def fib3(n):
    if n == 0:
        return 0
    F0 = 0
    F1 = 1
    for i in range(2,n+1):
        F0, F1 = F1, F0+F1
    return F1
```

Listing 2.7: Versione ricorsiva lineare

Complessità Temporale $T_{fib3}(n) = \Theta(1) + T_{fib3}(n-1) + \Theta(1) = \Theta(n)$

Complessità Spaziale $S_{fib3}(n) = \Theta(1) + S_{fib3}(n-1) + \Theta(1) = \Theta(n)$

Al fine di poter confrontare i tempi di calcolo delle diverse implementazioni sopra riportate, assumiamo di utilizzare nomi diversi per le diverse implementazioni, cioè di usare i nomi `fib0`, `fib1`, `fib2` e `fib3`.

```python
import time
n = 40
t0 = time.time()
f0 = fib(n)
t1 = time.time()
f1 = fib1(n)
t2 = time.time()
f2 = fib2(n)
t3 = time.time()
f3 = fib3(n)
t4 = time.time()
print("Risultato e Tempo Versione iterativa con lista:   ", f1, t2-t1)
print("Risultato e Tempo Versione iterativa senza lista: ", f2, t3-t2)
print("Risultato e Tempo Versione iicorsiva lineare:     ", f3, t4-t3)
print("Risultato e Tempo Versione iicorsiva non lineare: ", f,  t1-t0)
```

Listing 2.8: Versione iterativa

Chapter 3

Liste, Vettori e Matrici

3.1 Gestione di vettori

3.1.1 Somma elementi di un vettore

La somma degli elementi di un vettore $A = (a_1, ..., a_n)$, con $n > 0$, è definita come $\Sigma_{i=1}^{n} a_i$

```
def somma(A):
    n = len(A)
    s = 0
    for i in range(n):
        s = s + A[i]
    return s
```

Listing 3.1: Somma elementi di un vettore

Complessità Temporale $T_{Prod}(n,p,m) = \Theta(n)$

Complessità Spaziale $S_{Prod}(n,p,m) = \Theta(1)$

3.1.2 Somma di due vettori

La somma di due vettori $A = (a_1, ..., a_n)$ e $B = (b_1, ..., b_n)$ è un vettore $C = (c_1, ..., c_n)$ dove $c_i = a_i + b_i$, $\forall i \in [1,n]$.

```
def Somma(A,B):
    C = [];
    for i in range(len(A)):
        C.append(A[i]+B[i])
  return C
```

Listing 3.2: Somma di due vettori

Complessità Temporale: $T_{Somma}(n) = \Theta(n)$

Complessità Spaziale: $S_{Somma}(n) = \Theta(n)$

3.1.3 Prodotto scalare di due vettori

Il prodotto scalare di due vettori $A = (a_1, ..., a_n)$ e $B = (b_1, ..., b_n)$ è definito come $A \cdot B = \Sigma_{i=1}^{n} a_i \cdot b_i$.

```
1 def ProdS(A,B):
2     s = 1;
3     for i in range(len(A)):
4         s = s + A[i] * B[i]
5   return s
```

Listing 3.3: Prodotto scalare di due vettori

Complessità Temporale: $T_{ProdS}(n) = \Theta(n)$

Complessità Spaziale: $S_{ProdS}(n) = \Theta(1)$

3.1.4 Concatenazione di due liste

```
1 def concat(A,B):
2     n = len(A)
3     m = len(B)
4     C = []
5     for i in range(n):
6         C.append(A[i])
7     for i in range(m):
8         C.append(B[i])
9     return C
```

Listing 3.4: Concatenazione di due liste

Complessità Temporale $T_{Prod}(n, p, m) = \Theta(n + m)$

Complessità Spaziale $S_{Prod}(n, p, m) = \Theta(n + m)$

3.1.5 Fusione di due liste ordinate

```
def merge(A,B):
    n = len(A)
    m = len(B)
    C = []
    i = 0
    j = 0
    while i < n and j < m:
        if A[i] <= B[j]:
            C.append(A[i])
            i = i+1
        else:
            C.append(A[j])
            j = j+1
    while i < n:
        C.append(A[i])
        i = i + 1
    while j < m
        C.append(B[j])
        j = j + 1
    return C
```

Listing 3.5: Fusione di due liste ordinate

Complessità Temporale $T_{Prod}(n,p,m) = \Theta(n+m)$

Complessità Spaziale $S_{Prod}(n,p,m) = \Theta(n+m)$

3.2 Matrici

3.2.1 Visualizzazione

```
def printMatrix(M):
    for i in range(len(M)):
        print(M[i])
```

Listing 3.6: Stampa matrice

```
from __future__ import print_function

def printChess(chess):
    for i in range(len(chess)):
        print('[% 3d' % chess[i][0], end='')
        #print("[ ", chess[i][0], end='')
        for j in range(1,len(chess)):
            print(',% 3d' % chess[i][j], end='')
        print(" ]")
```

Listing 3.7: Stampa scacchiera

3.2.2 Somma di due matrici

La somma di due matrici $A(n,m)$ e $B(n,m)$ è una matrice $C(n,m)$ dove $c_{ij} = a_{ij} + b_{ij}$, $\forall i \in [1,n]$ e $\forall j \in [1,m]$.

```
def Somma(A,B):
    C = [];
    for i in range(len(A)):
      C.append([])
      for j in range(len(A[0])):
          C[i].append(A[i][j]+B[i][j])
    return C
```

Listing 3.8: Somma di due vettori

Complessità Temporale: $T_{Somma}(n,m) = \Theta(n \cdot m)$

Complessità Spaziale: $S_{Somma}(n,m) = \Theta(n \cdot m)$

3.2.3 Prodotto di due matrici

Il prodotto di due matrici $A(n,p)$ e $B(p,m)$ è una matrice $C(n,m)$ dove $c_{ij} = \sum_{k=1}^{p} a_{ik} + b_{kj}$, $\forall i \in [1,n]$ e $\forall j \in [1,m]$.

```
def Prod(A,B):
    C = []
    for i in range(len(A)):
      C.append([])
      for i in range(len(B[0])):
          C[i].append(0)
          for k in range(len(B)):
            C[i][j] = C[i][j] + B[i][k] * B[k][j]
    return C
```

Listing 3.9: Somma di due vettori

Complessità Temporale $T_{Prod}(n,p,m) = \Theta(n \cdot m \cdot p)$

Complessità Spaziale $S_{Prod}(n,p,m) = \Theta(n \cdot m)$

Chapter 4

Ricorsione

4.0.1 Simulazione della iterazione con la ricorsione

```
def sum(A, i):
    if i >= len(A):
        return 0
    else:
        return A[i]+sum(A, i+1)
```

Listing 4.1: Somma elementi di una lista - Versione ricorsiva

La funzione precedente verrà invocata con una chiamata del tipo `sum(V,0)`. Cioè, l'indice parte dal primo elemento e cresce.

```
def sum(A, i):
    if i < 0:
        return 0
    else:
        return A[i]+sum(A, i-1)
```

Listing 4.2: Somma elementi di una lista - Versione ricorsiva

La funzione precedente verrà invocata con una chiamata del tipo `sum(V,len(V)-1)` Cioè, l'indice parte dall'ultimo elemento e decresce.

```
def ProdSRec(A,B,k):
    if k < 0:
        return 0
    else:
        return ProdSRec(A,B,k-1) + A[k]*B[k]
```

Listing 4.3: Prodotto scalare - Versione ricorsiva

La funzione precedente verrà invocata con una chiamata del tipo `ProdSRec(U,V,len(V)-1)`. Cioè, l'indice parte dall'ultimo elemento e decresce.

4.0.2 Somma di vettori e matrici

```
def SumVectRec(A,B,k):
    if k < 0:
        return []
    else:
        C = SumVectRec(A,B,k-1)
        C.append(A[k]+B[k])
        return C
```

Listing 4.4: Somma vettori - Versione ricorsiva

La funzione precedente verrà invocata con una chiamata del tipo `SumVectRec(U,V,len(V)-1)`. Cioè, l'indice parte dall'ultimo elemento e decresce.

```
def sumMatRec(A, B, i):
    if i < 0:
        return []
    else:
        L = SumVectRec(A[i],B[i],len(A[i])-1)
        C = sumMatRec(A,B,i-1)
        C.append(L)
        return C
```

Listing 4.5: Somma Matrici - Versione ricorsiva

4.1 Torri di Hanoi

La Torre di Hanoi ÃÍ un rompicapo matematico composto da tre paletti e un certo numero di dischi di diametro decrescente, che possono essere posti in uno qualsiasi dei paletti. Il gioco inizia con tutti i dischi incolonnati in ordine decrescente su un paletto, in modo da formare un cono. Lo scopo del gioco é portare tutti i dischi su un altro paletto, potendo spostare solo un disco alla volta e potendo mettere un disco solo su un altro disco piú grande, mai su uno piú piccolo.

Il gioco fu inventato dal matematico francese Edouard Lucas nel 1883. Una leggenda, forse inventata dal matematico stesso, parla di un tempio Indù dove alcuni monaci sono costantemente impegnati a spostare, su tre colonne di diamante, 64 dischi d'oro secondo le regole della Torre di Hanoi (a volte chiamata Torre di Brahma). La leggenda narra che quando i monaci completeranno il lavoro, il mondo finirá . L'antica leggenda indiana recita cosí: $<<$nel grande tempio di Brahma a Benares, su di un piatto di ottone, sotto la cupola che segna il centro del mondo, si trovano 64 dischi d'oro puro che i monaci spostano uno alla volta infilandoli in un ago di diamanti, seguendo l'immutabile legge di Brahma: nessun disco può essere posato su un altro più piccolo. All'inizio del mondo tutti i 64 dischi erano infilati in un ago e formavano la Torre di Brahma. Il processo di spostamento dei dischi da un ago all'altro é tuttora in corso. Quando l'ultimo disco sará finalmente piazzato a formare di nuovo la Torre di Brahma in un ago diverso, allora arriverá la fine del mondo e tutto si trasformerá in polvere$>>$

Il problema può essere risolto facilmente utilizzando una semplice funzione ricorsiva basata sulla definizione induttiva della soluzione.

```
def moveDisk(X,Y):
    global h
    h = h+1
    print(str(h) + "> Muovi un disco da " + str(X) + " a " + str(Y))
```

Listing 4.6: Muovi un disco

```
h = 0
def hanoi(n,X,Y,Z):
    if n > 0:
        hanoi(n-1,X,Z,Y)
        moveDisk(X,Y)
        hanoi(n-1,Z,Y,X)
```

Listing 4.7: Torri di Hanoi

```
hanoi(4,'A','B','C')
```

Listing 4.8: Chiamata funzione

Complessità Temporale $T_{hanoi}(n) = \Theta(1) + T_{hanoi}(n-1) + T_{hanoi}(n-1) = \Theta(2^n)$

Complessità Spaziale $S_{hanoi}(n) = \Theta(1) + S_{hanoi}(n-1) = \Theta(n)$

4.2 Permutazioni di un vettore

```
1  def permute(A,k):
2      if k == len(A):
3          global h
4          h = h+1
5          print(h, ") ", A)
6      else:
7          for i in range(n,len(V)):
8              A[k], A[i] = A[i], A[k]
9              permute(A,k+1)
10             A[k], A[i] = A[i], A[k]
```

Listing 4.9: Permutazioni di un vettore

```
1 h = 0
2 V = [0,1,2]
3 permute(V,len(V))
```

Listing 4.10: Programma principale

Complessità Temporale $T_{permute}(n) = \begin{cases} \Theta(n) & \text{se } n = 0 \\ \Theta(1) + n(\Theta(1) + T_{permute}(n-1)) & sen > 0 \end{cases}$

$$= \begin{cases} \Theta(n) & \text{se } n = 0 \\ n\,\Theta(1) + n\,T_{permute}(n-1) & sen > 0 \end{cases} = \begin{cases} \Theta(n) & \text{se } n = 0 \\ n!\,T_{permute}() & sen > 0 \end{cases} = n!\,\Theta(n) = \Theta(n!)$$

Complessità Spaziale $S_{permute}(n) = \Theta(1) + S_{permute}(n-1) = \Theta(n)$

4.3 Determinante di una matrice

Il calcolo del determinante con il metodo di Laplace è una tecnica che risulta efficiente solo per matrici molto piccole o contenenti un gran numero di zeri. Data una matrice $A(n,n)$, si procede scegliendo una riga, la i-esima, tramite la formula:

$$det(A) = \sum_{j=1}^{n} a_{i,j}\, C_{i,j}$$

dove $C_{i,j}$ è il determinante (minore) di ordine $n-1$ ottenuto dalla matrice A eliminando la riga i-esima e la colonna j-esima. Esiste uno sviluppo analogo anche lungo la j-esima colonna.

```
def riduciV(V,k): # theta(k)
    U = []
    for i in range(0,k):
        U.append(V[i])
    for i in range(k+1,len(V)):
        U.append(V[i])
    return U

def riduci(A,k):
    C = []
    for i in range(1,len(A)):
        C.append(riduciV(A[i],k))
    return C

def det(A):                                  # Theta(1)
    if len(A) == 1:                          # Theta(1)
        return A[0][0]                       # Theta(1)
    d = float(0)                             # Theta(1)
    for j in range(len(A)):                  # n Theta(1)
        C = riduci(A,j)                      # n Theta(nxn)
        d = d + pow(-1,j)*A[0][j]*det(C).    # n (Theta(1)+T_{det}(n-1))
    return d
```

Listing 4.11: Versione ricorsiva

```
A = [ [1, -4,  2],
      [3,  1, -6],
      [1, -1, -1] ]
print(determinante(A))
```

Listing 4.12: Versione ricorsiva

Complexity

- $T_{riduciV}(n) = \Theta(n)$; $S_{riduciV}(n) = \Theta(n)$;
- $T_{riduci}(n) = \Theta(n^2)$; $S_{riduci}(n) = \Theta(n^2)$;

4.4 Risoluzione sistema di equazioni lineare

Ilseguente programma risolve un sistema di equazioni lineari di primo grado con il metodo di Laplace.

```
def replace(A,B,k):
    for i in range(len(A)):
        A[i][k], B[i] = B[i], A[i][k]

def sistema(A,B):
    d = determinante(A)
    X = []
    print(determinante(A))
    for i in range(len(A)):
        replace(A,B,i)
        print(determinante(A))
        X.append(float(det(A)/d))
        replace(A,B,i)
    return X
```

Listing 4.13: Versione ricorsiva

```
# programma principale

A = [ [1, -4,  2],
      [3,  1, -6],
      [1, -1, -1] ]
B = [9,1,3]
print(sistema(A,B))
```

Listing 4.14: Versione ricorsiva

Chapter 5

Ordinamento di Vettori

5.1 Insertion Sort

```
1 def insertionSort(A):
2     for i in range(1,len(A)):
3         minV = A[i]
4         j = i
5         while j > 0 and A[j-1] > minV:
6             A[j] = A[j-1]
7             j = j-1
8         A[j] = minV
```

Listing 5.1: Insertion Sort

Assumendo che il tempo per eseguire una singola istruzione relativa alls riga di codice i sia t_j, la complessità è definita come segue:

Caso peggiore $T(n) = t_1 + (n-1) \times (t_2 + t_3 + t_4 + t_8) + O(\sum_{j=1}^{n-1}(t_5 + t_6 + t_7)) = O(1+2+\cdots+n-1) = O((n-1) \cdot n/2) = \Theta(n^2)$

Caso migliore $T(n) = t_1 + (n-1) \times (t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + t_8) = \Theta(n)$

Quindi: $T_{insertionSort}(n) \in O(n^2)$ e $T_{insertionSort}(n) \in \Omega(n^2)$

5.2 Selection sort

È un algoritmo estremamente intuitivo e sempliceNella pratica è utile quando l'insieme da ordinare è abbastanza piccolo e dunque può essere utilizzato anche un algoritmo non molto efficiente con il vantaggio di non rendere troppo sofisticata codifica del programma che implementa

Idea: ripetere $n-1$ volte la procedura che durante la i-esima iterazione seleziona l'i-esimo elemento più piccolo dell'insieme e lo scambia con quello che di trova in posizione i

```
1 def selectionSort(A):
2     for i in range(len(A)-1):
3         minp = i
4         for j in range(i+1,len(A)):
5             if A[j] < A[minp]:
6                 minp = j
7         A[i], A[minp] = A[minp], A[i]
```

Listing 5.2: Selection Sort

Caso peggiore $T(n) = t_1 + (n-1) \times (t_2 + t_3 + t_4 + t_8) + O(\sum_{j=1}^{n-1}(t_5 + t_6 + t_7)) = O(1+2+\cdots+n-1) = O((n-1) \cdot n/2) = O(n^2)$

Caso migliore $T(n) = t_1 + (n-1) \times (t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + t_8) = \Omega(n)$

5.3 Bubble sort

L'idea di base di questo algoritmo è di far risalire gli elementi più grandi verso l'alto (i.e. nelle posizioni di indice più alto) e, nel contempo, far ridiscendere gli elementi più piccoli verso verso il basso (i.e. posizioni di indice più basso)

La strategia consiste nello scorrere più volte la sequenza in input confrontando, ad ogni passo, l'ordinamento reciproco di elementi contigui e scambiando le posizioni di eventuali coppie non ordinate.

```
1 def bubbleSort(A):
2     n = len(A)-1
3     for i in range(n):
4         for j in range(n-i):
5             if A[j] > A[j+1]:
6                 A[j], A[j+1] = A[j+1], A[j]
```

Listing 5.3: Bubble Sort- versione 1

```
1  def bubbleSort2(A):
2      n = len(A)-1
3      swap = True
4      i = 0
5      while i < n and swap:
6          swap = False
7          for j in range(n-i):
8              if A[j] > A[j+1]:
9                  A[j], A[j+1] = A[j+1], A[j]
10                 swap = True
11     i = i+1
```

Listing 5.4: Bubble Sort - versione 2

5.4 Ricerca lineare

```
def linearSearch(x,A):
    for i in range(len(A)):
        if x == A[i]:
            return i
    return -1
```

Listing 5.5: Ricerca lineare

Complessità Temporale $T_{linearSearch}(n) = O(n)$

5.5 Ricerca binaria

```
def binarySearch(k,A):
    first = 0
    last = len(A)-1
    while first <= last:
        medium = (int)((first+last)/2)
        if k == A[medium]:
            return medium
        if k < A[medium]:
            last = medium-1
        else:
            first = medium+1
    return -1
```

Listing 5.6: Ricerca lineare

Complessità Temporale $T_{linearSearch}(n) = O(\log_2 n)$

```
V = [7, 9, 1, 5, 2, 6, 19, 42, 3, 13, 18, 17, 21, 8, 14, 13]
print(V)
print(linearSearch(18,V))
bubbleSort2(V)
print(V)
print(binSearch(18,V))
print(binSearchR(18,V,0,len(V)-1))
```

Listing 5.7: Versione ricorsiva

Chapter 6

Divide-et-Impera

La tecnica Divide-et-Impera è una tra le tecniche più importanti con una vasta gamma di applicazioni. Essa consiste nel dividere un problema in sotto-problemi di dimensione più piccola, risolvere ricorsivamente tali sotto-problemi, e quindi combinale le soluzioni dei sotto-problemi per ottenere la soluzionedel problema globale. In un algoritmo ottenuto attraverso tale tecnica si individuano 3 fasi:

1. *Suddivisione del problema originale in sotto-problemi.* Il problema originale di dimensione n Âţe decomposto in k sotto-problemi di dimensione minore;
2. *Soluzione ricorsiva dei sotto-problemi.* Gli k sotto-problemi sono risolti ricorsivamente;
3. *Combinazione delle soluzioni.* Le soluzioni degli a sotto-problemi sono combinate per ottenere la soluzione del problema originale.

6.1 Ricerca binaria

```
1  def binSearchR(k,A,first,last):
2      if first <= last:
3          middle = (first+last)/2
4          if k == A[middle]:
5              return middle
6          if k > A[middle]:
7              return binSearchR(x,A,middle+1,last)
8          else:
9              return binSearchR(x,A,first,middle-1)
10     return -1
```

Listing 6.1: Ricerca binaria - versione ricorsiva

Complessitá caso peggiore

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ \Theta(1) + T(n/2). & if\ n > 1 \end{cases} = O(\log_2 n)$$

6.2 Fusine di liste ordinate

```
1  def mergeV(A,B):
2      n = len(A)
3      m = len(B)
4      i = 0
5      j = 0
6      C = []
7      while i < n and j < m:
8          if A[i] <= B[j]:
9              C.append(A[i])
10             i = i+1
11         else:
12             C.append(B[j])
13             j=j+1
14     while i < n:
15         C.append(A[i])
16         i = i+1
17     while j < m:
18         C.append(B[j])
19         j = j+1
20     return C
```

Listing 6.2: Versione ricorsiva

6.3 Merge sort

```
def merge(A, first , middle , last ):
    i = first
    j = middle+1
    C = []
    while i <= middle and j <= last:
        if A[i] <= A[j]:
            C.append(A[i])
            i = i+1
        else:
            C.append(A[j])
            j=j+1
    while i <= middle:
        C.append(A[i])
        i = i+1
    while j <= last:
        C.append(A[j])
        j = j+1
    for k in range(len(C)):
        A[first+k] = C[k]
```

Listing 6.3: Funzione Merge

```
def mergeSort(A, first , last ):
    if first < last:
        middle = int((first+last)/2)      # Identificaz. 2 sotto-probl.
        mergeSort(A, first , middle)
        mergeSort(A, middle+1, last)
        merge(A, first , middle , last)
```

Listing 6.4: Funzione Mergesort

Complessitá:

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T(n/2) + \Theta(n) & if\ n > 1 \end{cases}$$

$$T_{mergeSort}(n) = \begin{cases} 2 & if\ n = 2 \\ 2T(n/2) + n & if\ n = 2^k\ (con\ k > 1) \end{cases} = n \times log_2 n$$

$$T_{mergeSort}(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ \Theta/1) + T_{mergeSort}(\lceil n/2 \rceil) + T_{mergeSort}(\lfloor n/2 \rfloor) + T_{merge}(n) & if\ n > 1 \end{cases}$$

$$T_{mergeSort}(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ T_{mergeSort}(\lceil n/2 \rceil) + T_{mergeSort}(\lfloor n/2 \rfloor) + \Theta(n) & if\ n > 1 \end{cases} = \Theta(n \cdot log_2\ n)$$

6.4 Quick sort

fbox Inventato nel 1962 da Charles Anthony Richard Hoare[1], all'epoca exchange student presso la Moscow State University, vincitore del Turing Award (l'equivalente del Nobel per l'informatica) nel 1980 per il suo contributo nel campo dei linguaggi di programmazione.

È un algoritmo ricorsivo divide-et-impera:

1. Scegli un elemento pivot del vettore A, e partiziona il vettore in due parti considerando gli elementi maggiori o uguali di pivot e quelli minori di pivot,
2. Ordina ricorsivamente le due parti,
3. Restituisci il risultato concatenando le due parti ordinate.

```
def partition(A, first, last):
    i = first
    j = last
    pivot = A[first]
    while i < j:
        while i < last and A[i] < pivot:
            i = i+1
        while j > first and A[j] >= pivot:
            j = j-1
        if i < j:
            A[i], A[j] = A[j], A[i]
            i = i+1
            j = j-1
    if i == j and A[j] > pivot:
        j = j-1
    return j
```

Listing 6.5: Funzione Partition

Complessità:

$T_{partition}(n) = \Theta(n)$

```
def quickSort(A, first, last):
    if first < last:
        middle = partition(A, first, last)
        quickSort(A, first, middle)
        quickSort(A, middle+1, last)
```

Listing 6.6: Funzione QuickSort

```
X = [4, 6, 2, 11, 17, 3]          # theta(n)
Y = [12, 4, 11, 3, 7, 14, 6]      # theta(m)
Z = concat(X,Y)                   # theta(n+m)
print(Z)                          # theta(n+m)
quickSort(Z,0,len(Z)-1)           #
```

[1]Hoare, C. A. R., Quicksort. Computer Journal, 5(1): 10-15. (1962).

```
6 print(Z)                                    # theta(n+m)
```

Listing 6.7: Programma Principale

6.5 Master Theorem

Teorema 4. *Siano $a \geq 1$ e $b > 1$ due costanti costanti, $f(n)$ una funzione e $T(n)$ una funzione ricorrente definita sui numeri naturali come segue:*

$$T(n) = a \cdot T(n/b) + f(n)$$

dove interpretiamo n/b sia per indicare $\lfloor n/b \rfloor$ che per indicare $\lceil n/b \rceil$. Allora i limiti asintotici di $T(n)$ sono definiti come segue:

1. *Se $\exists \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon})$, allora $T(n) = \Theta(n^{\log_b a})$.*
2. *Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *Se $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon})$ e $\exists c < 1 : a \cdot f(n/b) < c \cdot f(n)$, allora $T(n) = \Theta(f(n))$.*

Example 1. *Si considerino le seguenti equazioni ricorrenti:*

- $T(n) = 2\,T(n/2) + n$.
  Abbiamo $a = 2$, $b = 2$, $f(n) = n$ e, quindi, $n^{\log_b a} = n^{\log_2 2} = n$.
  Poiché $f(n) = n = \Theta(n^{\log_2 2}) = \Theta(n)$, allora $T(n) = \Theta(n \log n)$ (Caso 2).
- $T(n) = 9\,T(n/3) + n$.
  Abbiamo $a = 9$, $b = 3$, $f(n) = n$ e, quindi, $n^{\log_b a} = n^{\log_3 9} = n^2$.
  Poiché $f(n) = n = O(n^{\log_b a - \varepsilon}) = O(n^{2-1})$, con $\varepsilon = 1$, allora $T(n) = \Theta(n^2)$ (Caso 1).
- $T(n) = 1\,T(n/3) + 1$.
  Abbiamo $a = 1$, $b = 3$, $f(n) = 1$ e, quindi, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$.
  Poiché $f(n) = 1 = \Theta(n^{\log_b a}) = \Theta(n^0) = \Theta(1)$, allora $T(n) = \Theta(\log n)$ (Caso 2).
- $T(n) = 3\,T(n/4) + n \log n$.
  Abbiamo $a = 3$, $b = 4$, $f(n) = n \log n$ e, quindi, $n^{\log_b a} = n^{\log_4 3} = n^{0{,}793}$.
  Poiché $f(n) = n\,logn = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{0.8+0.2})$, con $\varepsilon \approx 0.2$ e $\exists c : a \cdot f(n/b) < c \cdot f(n)$, cioè $3 \cdot n/4 \log(n/4) < c \cdot n \log n$ per $c > 3/4$, allora $T(n) = \Theta(f(n)) = \Theta(n \log n)$ (Caso 3).

Chapter 7

Priority queue

Una coda di priorità è una struttura dati astratta che permette di rappresentare un insieme di elementi su cui è definita una relazione d'ordine.

Sono definite almeno le seguenti operazioni:

- $insert(Q,k)$: inserimento di una chiave k;
- Min(Q) [risp.Max(Q)]: restituisce l'elemento minimo o massimo (a seconda dell'ordinamento scelto);
- $deleteMin(Q)$ [risp. $deleteMax(Q)$]: estrae dalla coda l'elemento minimo o massimo (a seconda dell'ordinamento scelto);

Spesso sono definite anche le seguenti operazioni:

- $decreaseKey(Q,i,v)$ [risp. $increaseKey(Q,i,v)$]: aggiorna il valore di priorità dell'elemento i-esimo, se necessario, al valore v;
- $delete(Q,i)$: rimuove l'elemento I-esimo dalla coda.

Le code a priorità trovano applicazioni in molti contesti, come ad esempio Ordinamento di sequenze di elementi, Problemi di schedulazione ottima di attività, Problemi di compressione di dati (codici di Huffman), Problemi di ottimizzazione su grafi pesati (percorsi minimi, alberi di copertura, ...), Algoritmi euristici di ricerca ottima su alberi/grafi.

Le code a priorità possono essere facilmente implementate tramite:

- Liste puntate o vettori disordinati:
  - Operazione di inserimento molto semplice e a tempo costante;
  - Le altre operazioni sono però onerose, lineari nella dimensione dell'insieme.
- Liste puntate o vettori ordinati:
  - Le operazioni di ricerca ed estrazione del minimo (massimo) risultano molto efficienti (tempo costante); Poiché è necessario mantenere l'ordine totale tra gli elementi, le altre operazioni (es. inserimento, cancellazione) possono risultare computazionalmente più costose (tempo lineare)
- In genere si preferiscono implementazioni più efficienti, in cui gli elementi sono mantenuti in coda secondo un ordine parziale (ad es. heap binari o vettori parzialmente ordinati)

Nel seguito viene presentata l'implementazione mediante vettori parzialmente ordinati.

7.1 Priority queue

```
def create():
    return []
```

Listing 7.1: Creazione priority queue vuota

```
def min(Q):
    if len(Q) == 0:
        print("Error: Empty Queue")
        return None
    return Q[0]
```

Listing 7.2: Valore minimo

```
def insert(Q,x):
    Q.append(x)
    i = len(Q)-1
    while i > 0 and Q[i] < Q[(i-1)/2]:
        Q[i], Q[(i-1)/2] = Q[(i-1)/2], Q[i]
        i = (i-1)/2
```

Listing 7.3: Inserimento chiave

```
def deleteMin(Q):
    if len(Q) == 0:
        print("Error: Empty Queue")
        return None
    min = Q[0]
    last = len(Q)-1
    Q[0] = Q[last]
    Q.pop(last)
    last = last-1
    i = 0
    while (2*i+1 <= last and Q[i] > Q[2*i+1]) or (2*i+2 <= last and Q[i
    ] > Q[2*i+2]):
        if 2*i+2 > last or Q[2*i+1] <= Q[2*i+2]:
            Q[i], Q[2 * i + 1] = Q[2 * i + 1], Q[i]
            i = 2*i+1
        else:
            Q[i], Q[2 * i + 2] = Q[2 * i + 2], Q[i]
            i = 2*i+2
    return min
```

Listing 7.4: Cancellazione chiave minima

```
A = [8, 11, 17, 5, 12, 2, 7]
B = []
for i in range(len(A)):
    insert(B,A[i])
print(B)
```

Listing 7.5: Programma principale

7.2 Heap sort

```
def heapSort(A):
    B = []
    for i in range(len(A)):
        inserisci(B, A[i])
    print(B"heap:", B)
    for i in range(len(A)):
        A[i] = deleteMin(B)
```

Listing 7.6: Heapsort

```
print("--- heapSort ---")
A = [8, 11, 17, 5, 12, 2, 7]
print(A)
heapSort(A)
print(A)
```

Listing 7.7: Programma principale

Chapter 8

Liste e Alberi

8.1 Liste

```
def creaLista():
    return []

def inserisciCoda(L,x):
    if L == []:
        L.append(x)
        L.append([])
    else:
        inserisciCoda(L[1],x)

def inserisciTesta(L,x):
    if L == []:
        L.append(x)
        L.append([])
    else:
        # [key,next] = L
        [key,next] = [L[0], L[1]]
        L[0] = x
        L[1] = [key,next]

def lista2Array(L,A):
    if L == []:
        return
    A.append(L[0])
    lista2Array(L[1],A)
```

Listing 8.1: Lista

```
L1 = creaLista()
inserisciTesta(L1,5)
inserisciTesta(L1,7)
inserisciTesta(L1,4)
inserisciTesta(L1,2)
print(L1)
A1 = []
lista2Array(L1,A1)
print(A1)
```

Listing 8.2: Programma principale

8.2 Alberi binari di ricerca

Un albero binario di ricerca (noto anche come *Binary Search Tree - BDT*) è una particolare struttura dati che permette di effettuare in maniera efficiente operazioni come: ricerca, inserimento e cancellazione di elementi.

In particolare, un albero binario di ricerca è un albero binario che soddisfa la seguente proprietà per ogni nodo v: $key(left(v)) \leq key(v) \leq key(right(v))$.

```
def createTree():
    return []

def empty(A):
    return A == []

def value(A):
    return A[0]

def left(A):
    return A[1]

def right(A):
    return A[2]

def setValue(A,x):
    A[0] = x

def findMax(A):
    if empty(right(A)):
        return value(A)
    return findMax(right(A))

def addNode(A,x):
    A.append(x)
    A.append([])
    A.append([])

def deleteNode(A):
    A.pop()
    A.pop()
    A.pop()

def copyNode(A,B):
    A[0] = B[0]
    A[1] = B[1]
    A[2] = B[2]
```

Listing 8.3: Funzioni base

```
def insert(A,x):
    if empty(A):
        addNode(A,x)
    else:
        if x <= value(A):
            insert(left(A),x)
        else:
            insert(right(A),x)
```

Listing 8.4: Inserimento di una chiave

```
def search(A,x):
    if empty(A):
        return False
    elif x == value(A):
      return True
    elif x < value(A):
        return search(left(A),x)
    else:
      return search(right(A),x)
```

Listing 8.5: Ricerca di una chiave

```
def delete(A,x):
    if empty(A):
        return
    if value(A) == x:
        if empty(left(A)) and empty(right(A)):
            deleteNode(A)
        elif empty(left(A)) and not empty(right(A)):
            copyNode(A, right(A))
        elif empty(right(A)) and not empty(left(A)):
            copyNode(A, left(A))
        else:
            y = findMax(left(A))
            setValue(A,y)
            delete(left(A),y)
    elif x < value(A):
        delete(left(A),x)
    else:
        delete(right(A),x)
```

Listing 8.6: Cancella una chiave

```
def infix(A,B):
    if not empty(A):
        infix(left(A),B)
        B.append(value(A))
        infix(right(A),B)

def prefix(A,B):
    if not empty(A):
      B.append(value(A))
      prefix(left(A),B)
      prefix(right(A),B)

def postfix(A,B):
    if not empty(A):
      postfix(left(A),B)
      postfix(right(A),B)
      B.append(value(A))

def breadthfirst(Q,B):
    # Q e' una lista di alberi che devono essere visitati
    while Q != []:
        A = Q[0]
        B.append(value(A))
        if not empty(left(A)):
            Q.append(left(A))
        if not empty(right(A)):
            Q.append(right(A))
        Q.pop(0)
```

Listing 8.7: Visita dell'albero

```
A = createTree()
C = [10, 5, 20, 3, 8, 1, 4, 7, 9, 15, 25, 12, 18, 22, 27]
for i in range(len(C)):
    insert(A,C[i])
print(C)
print(A)

B = []
infix(A,B)
print("infix: ", B)
B = []
prefix(A,B)
print("prefix: ", B)
B = []
postfix(A,B)
print("postfix: ", B)
B = []
breadthfirst([A],B)
print("breadthfirst: ", B)

delete(A,9)
print(A)
delete(A,8)
print(A)
delete(A,20)
print(A)
```

Listing 8.8: Programma principale

La complessità delle operazioni è la seguente:

- $T_{search}(n) = T_{insert}(n) = T_{delete}(n) = O(n)$,

poiché l'albero potrebbe generare in una lista.

8.3 Alberi AVL

Un albero binario di ricerca si dice bilanciato se per ciascun nodo le altezze del sotto-albero sinistro e del sotto-albero destro differiscono al più di 1. L'altezza di un albero binario di ricerca bilanciato con n chiavi è al massimo $O(\log n)$.

Determiniamo $n(h)$, ovvero il numero minimo di nodi presenti in un albero di altezza h.

- *Caso base:* $n(1) = 1, n(2) = 2$.
- *Caso induttivo:* Per $h \geq 3$, i sotto-alberi possono differire al massimo di 1. Quindi $n(h) \geq n(h-1) + n(h-2)$.

  Da ciò deriviamo:

  $$n(h) \geq 2 \cdot n(h-2) \equiv n(h) \geq 4 \cdot n(h-4) \equiv n(h) \geq 8 \cdot n(h-6) \equiv \cdots \equiv n(h) \geq 2^i \cdot n(h-2\,i)$$

  per ogni i tale che $h - 2 \cdot i \geq 1$.

Scegliendo i in modo tale che $h-2i=1$ oppure $h-2i=2$ troviamo la soluzione al nostro problema. Infatti, scegliendo $i=\lceil h/2\rceil-1$ e sostituendo il valore di i nella equazione precedente $n(h)>2^i+n(h-2i)$, otteniamo $n(h)>2^{\lceil h/2\rceil-1}+n(h-2\lceil h/2\rceil-1)\geq 2^{\lceil h/2\rceil-1}\cdot n(1)\geq 2^{\lceil h/2\rceil-1}$.

Da $n(h)>2^{\lceil h/2\rceil-1}$ otteniamo $\log n(h)>\lceil h/2\rceil-1$, cioè $h<2\log n(h)+2$. Concludiamo quindi che $h=O(\log n(h)$ e $n(h)=\Omega(2^h)$.

Bilanciamento dell'albero

Un nodo con il coefficiente di bilanciamento diverso da 1, 0 o -1 è considerato sbilanciato e viene ribilanciato grazie alle rotazioni. Ne esistono quattro tipi:

A ciascun nodo x associamo un fattore di bilanciamento $b(x)$ pari alla differenza tra le altezze del sotto-albero sinistro e del sotto-albero destro, definito anche nel seguente modo:

$$b(x)=\begin{cases}0 & \text{se } x \text{ è un nodo foglia}\\ b(left(x))-b(right(x)) & \text{se } x \text{ non è un nodo foglia}\end{cases}$$

Rotazione a destra Si esegue quando un nodo ha un coefficiente di bilanciamento di $+2$ e il suo figlio sinistro un coefficiente di bilanciamento uguale a $+1$ o 0.

Rotazione a sinistra Si esegue quando un nodo ha un coefficiente di bilanciamento di -2 ed il suo figlio destro un coefficiente di bilanciamento uguale a -1 o 0.

Rotazione Sinistra-Destra Si esegue quando un nodo ha un coefficiente di bilanciamento di $+2$ e il suo figlio sinistro un coefficiente di bilanciamento uguale a -1.

Rotazione Destra-Sinistra Si esegue quando un nodo ha un coefficiente di bilanciamento di -2 e il suo figlio destro un coefficiente di bilanciamento uguale a $+1$

```
def createTree():
    return []

def empty(A):
    return A == []

def value(A):
    return A[0]

def left(A):
    return A[1]

def right(A):
    return A[2]

def setValue(A,x):
    A[0] = x

def setLeft(A,p):
    A[1] = p

def setRight(A,p):
    A[2] = p

def addNode(A,x):
    A.append(x)
    A.append([])
    A.append([])

def deleteNode(A):
    A.pop()
    A.pop()
    A.pop()

def copyNode(A,B):
    A[0] = B[0]
    A[1] = B[1]
    A[2] = B[2]

def findMax(A):
    if empty(right(A)):
        return value(A)
    return findMax(right(A))
```

Listing 8.9: Funzioni di base

```
def depth(A):
    if empty(A):
        return 0
    return max(depth(left(A)),depth(right(A))) + 1

def bal(A):
    if empty(A):
        return 0
    return depth(left(A)) - depth(right(A))

def rightRotate(A):
    T = A
    A = left(A)
    setLeft(T,right(A))
    setRight(A,T)
    return A

def leftRotate(A):
    T = A
    A = right(A)
    setRight(T,left(A))
    setLeft(A,T)
    return A

def rotate(A):
    if bal(A) == 2 and not empty(left(A)) and bal(left(A)) >= 0:
        A = rightRotate(A)
    if bal(A) == -2 and not empty(right(A)) and bal(right(A)) <= 0:
        A = leftRotate(A)
    if bal(A) == 2 and not empty(left(A)) and bal(left(A)) < 0:
        setLeft(A,leftRotate(left(A)))
        A = rightRotate(A)
    if bal(A) == -2 and not empty(right(A)) and bal(right(A)) > 0:
        setRight(A,rightRotate(right(A)))
        A = leftRotate(A)
    return A
```

Listing 8.10: Funzione Rotate

```
def search(A,x):
    if empty(A):
        return False
    if x == value(A):
        return True
    if x < value(A):
        return search(left(A),x)
    else:
        return search(right(A),x)

def insert(A,x):
    if empty(A):
        addNode(A,x)
    elif x <= value(A):
        insert(left(A),x)
    else:
        insert(right(A),x)
    return rotate(A)
```

Listing 8.11: Funzioni search e insert

```
def delete(A,x):
    if empty(A):
        return A
    if x == value(A):
        if empty(left(A)) and empty(right(A)):
            deleteNode(A)
        elif empty(left(A)) and not empty(right(A)):
            copyNode(A, right(A))
        elif not empty(left(A)) and empty(right(A)):
            copyNode(A, left(A))
        else:
            y = findMax(left(A))
            setValue(A,y)
            setLeft(A, delete(left(A),y))
    elif x < value(A):
        setLeft(A, delete(left(A),x))
    else:
        setRight(A, delete(right(A),x))
    return rotate(A)
```

Listing 8.12: Funzione delete

```
A = createTree()
C = [10, 9, 8,  7, 6, 5, 4, 3, 2, 1]
for i in range(len(C)):
    A = insert(A,C[i])
print(C)
print("A: ", A)
B = createTree()
C = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in range(len(C)):
    B = insert(B,C[i])
print(C)
print("B: ", B)

A = delete(A,8)
print(A)
A = delete(A,9)
print(A)
A = delete(A,10)
print(A)
```

Listing 8.13: Programma principale

8.4 Heap

Un (min-)heap (resp. (max-)-heap) è una struttura dati basata sugli alberi che soddisfa la "proprietà di heap": se A è un genitore di B, allora la chiave (il valore) di A è minore (risp. maggiore) o uguale rispetto alla chiave di B.

Un heap è definito come una coppia: albero binario e numero di elementi, dove la relazione d'ordine è definita tra coppie di nodi "padre/figlio". Nel codice riportato la relazione d'ordine è chiave padre $\leq$ chiave figlio, per cui il nodo radice contiene sempre l'elemento minimo.

```
def createTree():
    return []

def key(T):
    return T[0]

def left(T):
    return T[1]

def right(T):
    return T[2]

def emptyTree(T):
    return T == []

def setKey(T,x):
    T[0] = x

def swapKeys(A1,A2):
    k = key(A1)
    setKey(A1,key(A2))
    setKey(A2,k)

def addLeafNode(T,x):
    T.append(x)
    T.append([])
    T.append([ ])

def deleteLeafNode(T):
    T.pop()
    T.pop()
    T.pop()
```

Listing 8.14: Primitive per alberi completi

```
def createHeap():
    return [ createTree(), 0 ]

def size(H):
    return H[1]

def tree(H):
    return H[0]

def setSize(H, n):
    H[1] = n

def emptyHeap(H):
    return size(H) == 0

def min(H):
    if emptyHeap(H):
        print("Errore: heap vuoto")
        return None
    return key(tree(H))
```

Listing 8.15: Funzioni di base

Per trovare il punto dove inserire temporaneamente una nuova chiave si considera il numero di chiavi presenti nell'albero. Tale numero viene convertita in una sequenza di bit che denotano il percorso per individuare il punto in cui inserire la chiave.

```
def int2string(i):
    s = [ ]
    while i > 0:
        s.append(i % 2)
        i = int(i / 2)
    s.pop()
    return s

def last(s):
    return s[len(s)-1]
```

Listing 8.16: Converti numero in stringa di bit

Inserimento di una chiave

La seguente funzione *insertHeap* inserisce una chiave x nell'heap H. Questa funzione calcola prima il percorso s che bisognerà seguire nell'albero per inserire la chiave e quindi chiama la funzione *insertH* che inserirła chiave x nell'heap H seguendo il percorso s

```
def insertTree(T , s , x):
    if s == [ ]:
        addLeafNode(T,x)
    elif last(s) == 0:
        s.pop()
        insertTree(left(T),s,x)
        if key(T) > key(left(T)):
            swapKeys(T, left(T))
    else:
        s.pop()
        insertTree(right(T),s,x)
        if key(T) > key(right(T)):
            swapKeys(T, right(T))

def insertHeap(H,x):
    setSize(H, size(H)+1)
    s = int2string(size(H))
    insertTree(tree(H),s,x)
```

Listing 8.17: funzione insertTree

La complessità è pari all'altezza $O(h)$ dell'albero. Poichè $h = log_2 n$, la complessità è $O(log : 2n)$

Estrazione minimo

```
def deleteMin(H):
    if n < 1:
        print("Errore: heap vuoto")
        return None
    min= key(A)
    A = tree(H)
    n = size(H)
    if n == 1:
        setSize(H, 0)
        deleteLeafNode(A)
        return min
    s = int2string(n)
    p = tree(H)
    while len(s) > 0:
        if last(s) == 0:
            p = left(p)
        else:
            p = right(p)
        s.pop()
    x = key(p)
    deleteLeafNode(p)
    setSize(H, size(H)-1)
    setKey(A,x)
    p = A
    while (not emptyTree(left(p)) and key(p) > key(left(p))) or \
            (not emptyTree(right(p)) and key(p) > key(right(p))):
        if emptyTree(right(p)) or (key(left(p)) <= key(right(p))):
            swapKeys(p,left(p))
            p = left(p)
        else:
            swapKeys(p,right(p))
            p = right(p)
    return min
```

Listing 8.18: funzione insertHeap

La complessità è pari all'altezza $O(h)$ dell'albero. Poichè $h = log_2 n$, la complessità è $O(log : 2n)$

Mapping di un Heap in una Priority Queue. Alternativamente è anche possible ottenere lo stesso risultato con una visita in ampiezza dell'albero.

```
def insertPQ(H,i,Q):
    if H != []:
        Q[i] = H[0]
        insertPQ(H[1],2*i+1,Q)
        insertPQ(H[2],2*i+2,Q)

def tree2list(H):
    Q = []
    for i in range(H[1]+1):
        Q.append(0)
    insertPQ(H[0],0,Q)
    return Q
```

Listing 8.19: Funzione tree2list

Heap Sort

```
def heapSort(A):
    H = createHeap()
    for i in range (len(C)):
        insertHeap(H,C[i])
    for i in range(len(A)):
        A[i] = deleteMin (H)
```

Listing 8.20: funzione tree2list

Programma principale

```
C = [10, 5, 20, 3, 8, 1, 25, 7, 27, 15, 5, 12, 18, 22, 9]
print ("Input list:  ", C)
heapSort(C)
print("Sorted list: ", C)
```

Listing 8.21: Versione ricorsiva

```
Input list:   [10, 5, 20, 3, 8, 1, 25, 7, 27, 15, 5, 12, 18, 22, 9]
Sorted list:  [1, 3, 5, 5, 7, 8, 9, 10, 12, 15, 18, 20, 22, 25, 27]
```

Listing 8.22: Output

Chapter 9

Grafi

I grafi sono strutture con un un'ampissima gamma di campi applicativi. Un grafo è un insieme di elementi detti nodi o vertici che possono essere collegati fra loro attraverso archi. Formalmente, un grafo è coppia $G = (V,E)$ dove V è un insieme dei nodi ed $E \subseteq (V \times V)$ un insieme degli archi. Nel caso E fosse un multi-insieme, diremo che G è un multigrafo.

Dato un arco (a,b) diremo che a e b sono gli estremi dell'arco e che a è il nodo sorgente, mentre b è il nodo destinazione. Se E è una relazione simmetrica allora si dice che il grafo è non orientato (o indiretto), altrimenti si dice che è orientato (o diretto).

Un arco che ha due estremi coincidenti si dice *self-loop* (cappio). Un grafo non orientato, sprovvisto di self-loop si dice grafo semplice.

Un grafo si dice finito se entrambi A ed E sono finiti. Nel seguito il termine grafo, se non specificato diversamente, si riferisce a grafi finiti. L'ordine di un grafo è $|V|$, mentre $!E|$ è la dimensione. In un grafo non orientato, il numero di archi incidenti un nodo v costituisce il grado di v. In un grafo orientato il *grado* di ingresso (risp. uscita) è dato dal numero di archi entranti nel (risp. uscenti dal) nodo v. Gli *adiacenti* di un nodo u sono i nodi v tali che esiste un arco da u a.v.

Un grafo è definito *completo* se due qualsiasi dei suoi vertici sono adiacenti (esiste un arco che li connette). La massima cardinalità di un sottografo completo del grafo si chiama *densità* del grafo.

Un *percorso* di lunghezza n in G è dato da una sequenza di vertici $v_0, v_1, ..., v_n$ (non necessariamente tutti distinti) e da una sequenza di archi che li collegano $(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)$. I vertici v_0 e v_n si dicono estremi (sorgente e destinazione, rispettivamente) del percorso. Un percorso senza archi ripetuti prende il nome di *cammino*. Un cammino chiuso ($v_0 = v_n$) senza archi ripetuti viene detto *circuito*. Un cammino chiuso ($v_0 = v_n$) senza nodi ripetuti viene detto *ciclo*.

In un grafo $G = (V,E)$ $v,u \in V$ si dicono 'connessi" se esiste un cammino con estremi v e u (in entrambe le direzioni). Se tale cammino non esiste, v e u sono detti "sconnessi" o connessi in una sola direzione. La relazione di connessione tra vertici è una relazione di equivalenza.

Per $i = 1, ..., k$ (k classi di equivalenza) sono definibili i sottografi $G_i = (V_i, E_i)$ come i sottografi massimali che contengono tutti gli elementi connessi tra loro, che prendono il nome di componenti connesse di G, la cui cardinalitè spesso si indica con $\gamma(G) = k$. Se $\gamma(G) = 1$, G si dice "connesso". Un "nodo isolato" è un vertice che non è connesso a nessun altro vertice. Un nodo isolato ha grado 0. Un "ponte" e uno "snodo" sono, rispettivamente, un arco ed un vertice che se soppressi sconnettono il grafo.

9.1 Implementazione tramite matrice di adiacenza

Di seguito viene presentata l'implementazione di un insieme di operazioni su un grafo rappresentato mediante una matrice di adiacenza. Il seguente codice è memorizzato in un file dal nome "GraphA" che verra

importato per operare su grafi.

```
def createGraph(n):
    M = []
    for i in range(n):
        M.append([])
        for j in range(n):
            M[i].append(0)
    return M
```

Listing 9.1: Creazione di un grafo

Alternativamente, un grafo vuoto può essere implementato tramite la seguente funzione:

```
def createGraph(n):
    M = [ [0] * n ] * n
    return M
```

Listing 9.2: Creazione di un grafo

Di seguito vengono riportate alcune funzioni base per operare su un grafo.

```
def copyGraph(G):
    n = len(G)
    C = []
    for i in range(n):
        C.append([])
        for j in range(n):
            C[i].append(G[i][j])
    return C

def printGraph(G):
    for i in range(len(G)):
        print(G[i])

def size(G):
    return len(G)

def nodes(G):
    return list(range(size(G)))

def isEdge(G, i, j):
    return G[i][j] == 1

def insertEdge(G, i, j):
    G[i][j] = 1

def deleteEdge(G, i, j):  # theta(1)
    G[i][j] = 0

def outDegree(G, i):
    s = 0;
    for j in range(len(G)):
```

```
32          s = s + G[i][j]
33      return s
34
35  def inDegree(G, j):
36      s = 0;
37      for i in range(len(G)):
38          s = s + G[i][j]
39      return s
40
41  def neighbors(G, i):
42      L = []
43      for j in range(len(G[i])):
44          if G[i][j] == 1:
45              L.append(j)
46      return L
```

Listing 9.3: Funzione base

9.2 implementazione tramite lista di adiacenza

Di seguito viene presentata l'implementazione di un insieme di operazioni su un grafo rappresentato mediante una lista di adiacenza. Il seguente codice è memorizzato in un file dal nome "GraphL" che verra importato per operare su grafi.

```
1  def createGraph(n):
2      G = []
3      for i in range(n):
4          G.append([])
5      return G
```

Listing 9.4: Creazione di un grafo

```
1  def copyGraph(G):
2      n = len(G)
3      C = []
4      for i in range(n):
5          C.append([])
6          for j in G[i]:
7              C[i].append(j)
8      return C
9
10 def graph2matrix(G):
11     n = len(G)
12     M = []
13     for i in range(n):
14         M.append([])
15         for j in range(n):
16             M[i].append(0)
17     for i in range(n):
```

```
18          for j in G[i]:
19              M[i][j]=1
20      return M
21
22  def printGraph(G):
23      M = graph2matrix(G)
24      for i in range(len(M)):
25          print(M[i])
```

Listing 9.5: Stampa di un grafo

L'implementazione di alcune funzioni base per operare su grafi è riportata di seguito.

```
1   def size(G):
2       return len(G)
3
4   def nodes(G):
5       return list(range(size(G)))
6
7   def isEdge(G, x, y):
8       return (y in G[x])
9
10  def insertEdge(G, x, y):
11      if not (y in G[x]):
12          G[x].append(y)
13
14  def deleteEdge(G, i, j):
15    G[i].remove(j)
16
17  def neighbors(G, i):
18      L = []
19      for y in G[i]:
20          L.append(y)
21      return L
```

Listing 9.6: Funzioni base

9.3 Chiusura transitiva

```
1   import GraphA as g
2
3   def closure(G):
4       C = g.copyGraph(G)
5       for i in g.nodes(C):
6           g.insertEdge(C,i,i)
7       for k in g.nodes(C):
8           for i in g.nodes(C):
9               for j in g.nodes(C):
10                  if g.isEdge(C,i,k) and g.isEdge(C,k,j):
```

```
                    g.insertEdge(C,i,j)
    return C
```

Listing 9.7: Chiusura transitiva

```
import GraphA as g

G = g.createGraph(5)
print("--- Empty Graph ---")
g.printGraph(G)

g.insertEdge(G, 0, 1)
g.insertEdge(G, 0, 4)
g.insertEdge(G, 1, 2)
g.insertEdge(G, 2, 3)
g.insertEdge(G, 3, 1)
g.insertEdge(G, 4, 3)

print("--- Input Graph ---")
print(G)
g.printGraph(G)

C = closure(G)
print("--- Graph Closure ---")
g.printGraph(C)
```

Listing 9.8: Programma principale

Cammino semplice, euleriano, Raggiungibilità

9.4 Grafi pesati

Un grafo pesato è un grafo dove ogni arco ha associato un peso (valore numerico). L'implementazione precedente non funziona perchè non permette di distinguere l'assenza di un arco (equivalente ad avere un arco con peso infinito) da un arco con peso zero.

9.4.1 Implementazione tramite matrice di adiacenza

Di seguito viene riportata una nuova implementazione mediante matrice di adiacenza. La differenza rispetto all'implementazione precedente consiste nel fatto che una cella della matrice (i, j), denotante l'arco da i a j, è una lista (quindi un puntatore) contenente solo il peso dell'arco. Se il puntatore è nullo (cioè lista vuota) l'arco è assente, altrimenti il suo peso è memorizzato in una lista con un solo elemento. Ovviamente, alcune operazioni (*insertEdge, deleteEdge*) richiedono di indicare anche il peso dell'arco.

```
import math

def createGraph(n):
    M = [ [ math.inf for x in range(n) ] for x in range(n) ]
    return M

```

```
7  def copyGraph(G):
8      n = len(G)
9      C = []
10     for i in range(n):
11         C.append([])
12         for j in range(n):
13             C[i].append(G[i][j])
14     return C
15
16 def printGraph(G):
17     for i in range(len(G)):
18         print(G[i])
19
20 def size(G):
21     return len(G)
22
23 def nodes(G):
24     return list(range(size(G)))
25
26 def isEdge(G, x, y):
27     return G[x][y] != math.inf
28
29 def weight(G, x, y):
30     return G[x][y]
31
32 def insertEdge(G, i, j, w):
33     G[i][j] = w
34
35 def deleteEdge(G, i, j):
36     G[i][j] = math.inf
37
38 def outDegree(G, i):
39     s = 0;
40     for j in range(len(G)):
41         if G[i][j] != math.inf:
42             s += 1
43     return s
44
45 def inDegree(G, j):
46     s = 0;
47     for i in range(len(G)):
48         if G[i][j] != math.inf:
49             s += 1
50     return s
51
52 def neighbors(G, x):
53     # return G[i]
54     L = []
55     for y in nodes(G):
```

```
56            if G[x][y] != math.inf:
57                L.append([y,G[x][y]])
58        return L
```

Listing 9.9: Grafo pesato - Implementazione tramite matrice di adiacenza

9.4.2 implementazione tramite lista di adiacenza

L'implementazione tramite liste di adiacenza richiede di inserire nelle liste di adiacenze al posto di ciascun nodo (terminale di un arco) una coppia nodo/peso dell'arco.

```
import math

def createGraph(n):
    G = []
    for i in range(n):
        G.append([])
    return G

def copyGraph(G):
    n = len(G)
    C = []
    for i in range(n):
        C.append([])
        for [j,w] in G[i]:
            C[i].append([j,w])
    return C

def printGraph(G):
    for i in range(size(G)):
        print(i, G[i])

def size(G):
    return len(G)

def nodes(G):
    return list(range(size(G)))

def isEdge(G, x, y):
    for [j,w] in G[x]:
        if j == y:
            return True
    return False

def weight(G,x,y):
    for [j,w] in G[x]:
        if j == y:
            return w
    return math.inf

def insertEdge(G, x, y, w):
    if not isEdge(G,x,y):
        G[x].append([y,w])
    else:
        for p in G[x]:
            if p[0] == y:
```

```
                p = [y,w]

def deleteEdge(G, x, y):
    for k in range(len(G[x])):
        [j,w] = G[x][k]
        if j == y:
            G[x].pop(k)

def neighbors(G, x):
    L = []
    for [y,w] in G[x]:
        L.append([y,w])
    return L
```

Listing 9.10: Grafo pesato - Implementazione tramite liste di adiacenza

9.4.3 Creazione di grafo

```
import GraphA as g

G = g.createGraph(6)
print("Grafo con 6 nodi e un insieme vuoto di archi")
g.printGraph(G)
E = [ [0,1,2], [0,5,9], [1,2,6], [1,3,8], [1,5,5], [2,3,1], [4,2,7],
    [4,3,3], [5,4,3] ]
for [x,y,w] in E:
    g.insertEdge(G,x,y,w)
print("Grafo con 6 nodi e 9 archi")
g.printGraph(G)
```

Listing 9.11: Versione ricorsiva

9.5 Visita e ciclicità di un grafo

```
def edges(pred):
    E = []
    for i in range(len(pred)):
        if pred[i] != -1:
            E.append([pred[i],i])
    return E
```

Listing 9.12: Funzione ausiliare

9.5.1 Visita in ampiezza

L'algoritmo per la visita in ampiezza di un grafo (*brewdth First Search - BFS*) permette di definire quali nodi sono raggiungibili da un nodo s detto sorgente in modo simile a quanto definito per la visita in ampiezza degli alberi. Il procedimento può essere cosi sintetizzato e fa uso di due liste: una coda *queue* dei nodi da visitare e una lista V dei nodi visitati:

1. s viene inserito in *queue*
2. fino al quando la coda non è vuota viene estratto il primo elemento della coda *next* che viene inserito nella lista dei nodi visitati e, successivamente, i nodi adiacenti *next*, non presenti nella coda e non ancora visitati, vengono inseriti nella coda.

```
def breadthVisit(G,s):                    # Theta(m)
    Edges   = []
    Reached = [ False for i in range(len(G)) ]
    Reached[s] = True
    Queue   = [s]
    while Queue != []:
        next = Queue.pop(0)
        Adj = g.neighbors(G,next)
        for [j,w] in Adj:
            if not Reached[j]:
                Edges.append([next,j])
                Queue.append(j)
                Reached[j] = True
    return Edges
```

Listing 9.13: Visita in ampiezza

La complessità della precedente funzione è $O(m)$. Tale valore è determinato sotto l'assunzione che la complessità della funzione neighbors sia $\Theta(l)$, dove l è la lunghezza della lista restituita. Poiché, $\sum_i = 0^{n-1} adj(i) = m$, otteniamo che la complessità è lineare con il numero di archi contenuti nella parte di grafo raggiungibile dal nodo s. Pertanto, nel caso peggiore abbiamo $\Theta(m)$.

La successiva funzione utilizza una diversa rappresentazione degli archi che costituiscono l'albero restituito dalla funzione di visita.

```
def breadthVisit1(G,s):
    Pred    = [ -1 for i in range(len(G)) ]
    Queue   = [s]
    while Queue != []:
        next = Queue.pop(0)
        Adj = g.neighbors(G, next)
        for [y,w] in Adj:
            if Pred[y] == -1:
                Queue.append(y)
                Pred[y] = next
    return edges(Pred)
```

Listing 9.14: Visita in profondità

9.5.2 Visita in profondità

La funzione di visita in profondità di un grafo può essere ottenuta da qualle precedente sostituendo la coda con una pila, come di seguito mostrato. In tale funzione viene nuovamente introdotto il vettore `Reached` poichè la condizione che `Pred[y]` $!= -1$ non è più sufficiente a stabilire che il nodo y sia stato visitato.

```
def depthVisit(G,s):
    Pred    = [ -1 for i in range(len(G)) ]
    Reached = [ False for i in range(len(G)) ]
    Stack   = [s]
    while Stack != []:
        next = Stack.pop()
        Reached[next] = True
        Adj = g.neighbors(G,next)
        Adj.reverse()
        for [y,w] in Adj:
            if not Reached[y]:
                Stack.append(y)
                Pred[y] = next
    return edges(Pred)
```

Listing 9.15: Visita in profondità

9.5.3 Implementazione ricorsiva

```
def depthVisit1(G,s):
    Edges = []
    depthVisit2(G,s,[s],Edges)
    return Edges

def depthVisit2(G, x, Pred):
    Adj = g.neighbors(G,x)
    for [y,w] in Adj:
        if Pred[y] == -1:
            Pred[y] = x
            depthVisit2(G, y, Pred)
```

Listing 9.16: Visita in profondità

```
import WGraphL as g
import Algorithms as a

C = [ [0,1], [0,2], [0,3], [1,7], [2,3], [4,5], [5,4], [6,7], [7,5] ]
for [x,y] in C:
    D.append([x,y,1])

G = g.createGraph(8)
for [x,y,z] in D:
     g.insertEdge(G,x,y,z)

print("--- Grafo (Lista archi) ---")
g.printGraph(G)

print("Iterative breadth first search: ", a.breadthVisit(G,0))
print("Iterative breadth first search: ", a.breadthVisit1(G,0))
print("Iterative  depth  first search: ", a.depthVisit(G,0))
print("Recursive  depth  first search: ", a.depthVisit1(G,0))
```

Listing 9.17: Programma principale

9.5.4 Ciclicità

```
def inCycle(G, x, CPath, Reached):
    CPath[x] = True
    Adj = g.neighbors(G,x)
    for [y,w] in Adj:
        if not Reached[y]:
            if CPath[y]:
                return True
            elif inCycle(G, y, CPath, Reached):
                return True
    Reached[x] = True
    CPath[x] = False
    return False
```

Listing 9.18: Verifica di ciclicità

```
def isCyclic(G):
    CPath    = [ False for i in range(g.size(G)) ]
    Reached  = [ False for i in range(g.size(G)) ]
    for x in g.nodes(G):
        if not Reached[x] and inCycle(G, x, CPath, Reached):
            return True
    return False
```

Listing 9.19: Verifica di ciclicità

9.5.5 Componenti fortemente connesse

...

Chapter 10

Algoritmi golosi

La tecnica greedy (o metodo goloso) definisce un paradigma algoritmico, dove l'algoritmo cerca la soluzione ottima dal punto di vista globale (o una soluzione ammissibile (possibilmente una approssimazione dell'ottimo globale) attraverso la scelta, ad ogni passo, della soluzione migliore, secondo una metrica definita precedentemente (ottimo locale). Quando applicabili, questi algoritmi consentono di trovare soluzioni ottimali per determinati problemi in un tempo polinomiale, mentre negli altri non è garantita la convergenza all'ottimo globale.

Example 2. *Si supponga di dover restituire come resto 63 centesimi e che le unitá disponibili siano 25c, 10c, 5c, 1c. Il problema consiste nel selezionare un numero minimo di monete. Il calcolo può essere effettuato facilmente scegliendo l'unità più grande (25c) e quindi risolvendo il problema per* $(63-26)$*c. Il risultato finale sarà 6 monete (2 da 25c, 1 da 10c e 3 da 1c).*

La tecnica greedy consistere nello scegliere la soluzione locale ottima.

Tuttavia, in alcuni casi questa tecnica non funziona. Si supponga che il resto sia 15 e di avere a disposizioni monete da 11c, 5c e 1c. Il tal caso la tecnica greedy darebbe come risultato 5 (1 da 11c, e 4 da 1c), mentre la soluzione ottima è 3 (5 da 11) che potrebbe essere ottenuta mediante tecniche di programmazione dinamica.

10.1 Algoritmo di Dijkstra

Fu inventato nel 1956 dall'informatico olandese Edsger Dijkstra che lo pubblicò successivamente nel 1959.

L'algoritmo di Dijkstra è un algoritmo utilizzato per calcolare i cammini minimi in un grafo pesato con pesi non negativi. Tale algoritmo trova applicazione in molteplici contesti quale l'ottimizzazione nella realizzazione di reti (idriche, telecomunicazioni, stradali, circuitali, ecc.) o l'organizzazione e la valutazione di percorsi runtime nel campo della robotica.

Supponiamo di avere un grafo pesato con n vertici contraddistinti da numeri naturali $\{0,1,2,...,n-1\}$ e che si vogliano calcolare i cammini minimi dal nodo 0 verso tutti gli altri nodi. Le strutture dati utilizzate sono 3 vettori di dimensione n: $Visited$ (vettore di booleani), *Pred* (vettore di identificatori di nodi) e *Dist* (vettore con le distanze dei nodi dal nodo sorgente).

— SPIEGAZIONE DELL'ALGORITMO —

Esso è un algoritmo di tipo "greedy" perchè ad ogni passo si fa la scelta dell'ottimo locale, cioè del nodo che

10.1.1 Implementazione tramite liste

```
import GraphWL as g
import math

def minVertex(D,V):
    x = math.inf
    y = -1
    for i in range(len(D)):
        if (not V[i]) and (D[i] < x):
            y = i
            x = D[i]
    return y

def Dijkstra(G,s):
    n = g.size(G)
    dist    = [ math.inf for i in range(n) ]
    pred    = [ -1 for i in range(n) ]
    visited = [ False for i in range(n)]
    Edges   = []
    dist[s] = 0
    for i in range(n):
        next = minVertex(dist,visited)
        visited[next] = True
        Edges.append([pred[next], next, dist[next]])
        Adj = g.neighbors(G,next)
        for [z,w] in Adj:
            if not visited[z]:
                d = dist[next] + w          #Calcolo nuova distanza
                if (d < dist[z]):
                    dist[z] = d
                    pred[z] = next
    Edges.pop(0)
    return Edges
```

Listing 10.1: Algoritmo di Dijkstra

Complessità: $O(n^2)$.

```
import Algorithms as a
print("--- Dijkstra ---")
T = a.Dijkstra(G,0)
print(T)
```

Listing 10.2: Programma principale

10.1.2 Implementazione tramite Heap

```
import math

def createHeap():
    return []

def empty(Q):
    return Q == []

def weight(p):
    return p[1]

def insertHeap(Q,p):
    Q.append(p)
    i = len(Q)-1
    while i > 0 and weight(Q[i]) < weight(Q[(i-1)//2]):
        Q[i], Q[(i-1)//2] = Q[(i-1)//2], Q[i]
        i =(i-1)//2

def deleteMin(Q):
    if Q == []:
        print("Errore: Coda vuota")
        return [-1,math.inf]
    min = Q[0]
    if len(Q) == 1:
        Q.pop()
        return min
    Q[0] = Q[len(Q)-1]
    Q.pop(len(Q)-1)
    last = len(Q)-1
    i = 0
    while (2*i+1 <= last and weight(Q[i]) > weight(Q[2*i+1])) or (
            2*i+2 <= last and weight(Q[i]) > weight(Q[2*i+2])):
        if 2 * i + 2 > last or weight(Q[2*i+1]) < weight(Q[2*i+2]):
            Q[i], Q[2 *i+1] = Q[2*i+1], Q[i]
            i = 2 * i + 1
        else:
            Q[i], Q[2*i+2] = Q[2*i+2], Q[i]
            i = 2*i+2
    return min
```

Listing 10.3: File Heap

```
def createHeap():
    return [ [], 0 ]

def size(H):
    return H[1]

def tree(H):
    return H[0]

def setSize(H, n):
    H[1] = n

def empty (H):
    return size(H) == 0

def info(T):
    return T[0]

def left(T):
    return T[1]

def right(T):
    return T[2]

def key(N):
    return N[0]

def weight(N):
    return N[1]

def setInfo(T,x):
    T[0] = x

def emptyTree(T):
    return T == []

def min(H):
    return info(tree(H))

def addLeafNode(T,x):
    T.append(x)
    T.append([])
    T.append([ ])

def deleteNode(T):
    T.pop()
    T.pop()
    T.pop()

```

```
50 def int2string(i):
51     s = [ ]
52     while i > 0:
53         s.append(i % 2)
54         i = int(i / 2)
55     s.pop()
56     return s
57 
58 def last(s):
59     return s[len(s)-1]
60 
61 def insertTree(T , s , x):
62     if s == [ ]:
63         addLeafNode(T,x)
64     else:
65         if last(s) == 0:
66             s.pop()
67             insertTree(left(T),s,x)
68             if weight(info(T)) > weight(info(left(T))):
69                 z = info(T)
70                 setInfo(T,info(left(T)))
71                 setInfo(left(T),z)
72         else:
73             s.pop()
74             insertTree(right(T),s,x)
75             if weight(info(T)) > weight(info(right(T))):
76                 z = info(T)
77                 setInfo(T,info(right(T)))
78                 setInfo(right(T),z)
79 
80 def insertHeap(H,x):
81     setSize(H,size(H)+1)
82     s = int2string(size(H))
83     insertTree(tree(H),s,x)
```

Listing 10.4: Heap

```
def deleteMin (Q):
    A = tree(Q)
    n = size(Q)
    if n < 1:
        print("Errore: heap vuoto")
        return None
    min= info(A)
    if n == 1:
        setSize(Q, 0)
        deleteNode(A)
        return min
    s = int2string(n)
    p = tree(Q)
    while len(s) > 0:
        if last(s) == 0:
            p = left(p)
        else:
            p = right(p)
        s.pop()
    x = info(p)
    deleteNode(p)
    setSize(Q,size(Q)-1)
    setInfo(A,x)
    while (not emptyTree(left(A)) and \
      weight(info(A)) > weight(info(left(A)))) or \
        (not emptyTree(right(A)) and \
        weight(info(A)) > weight(info(right(A)))):
        if emptyTree(right(A)) or \
        (weight(info(left(A))) <= weight(info(right(A)))):
            z = info(A)
            setInfo(A, info(left(A)))
            setInfo(left(A), z)
            A = left(A)
        else:
            z = info(A)
            setInfo(A,info(right(A)))
            setInfo(right(A),z)
            A = right(A)
    return min
```

Listing 10.5: Algoritmo di Dijkstra

```python
import math
import GraphA as g
import Heap as h

def Dijkstra1(G,s):
    n = g.size(G)
    dist    = [ math.inf for i in range(n) ]
    pred    = [ -1 for i in range(n) ]
    visited = [ False for i in range(n)]
    Edges   = []
    dist[s] = 0
    Q = h.createHeap()
    h.insertHeap(Q,[s,0])
    while not h.empty(Q):
        [next,weight] = h.deleteMin(Q)
        if not visited[next]:
            visited[next] = True
            edges.append([pred[next],next,dist[next]])
            Adj = g.neighbors(G,next)
            for [z,w] in Adj:
                d = dist[next] + w
                if (dist[z] > d):
                    dist[z] = d
                    pred[z] = next
                    h.insertHeap(Q,[z,d])
    Edges.pop(0)
    return Edges
```

Listing 10.6: Algoritmo di Dijkstra

```python
C = [ [0,1,10], [0,2,5], [1,2,2], [1,3,1], [2,1,3], [2,3,9], [2,4,2],
    [3,4,4], [4,0,7], [4,3,6] ]
G = g.createGraph(5)
for [x,y,w] in C:
     g.insertEdge(G,x,y,w)

print("--- Grafo (Lista archi) ---")
g.printGraph(G)

print()
print("--- Dijkstra ---")
print(a.Dijkstra(G,0))
print("--- Dijkstra1 ---")
print(a.Dijkstra1(G,0))
```

Listing 10.7: Programma principale

Complessità: $O(m, \log_2 n)$

10.1.3 Algoritmo di Prim

Dato un grafo non orientato F e un nodo s, calcola un minimo albero ricoprente di G avente radice s.

L'algoritmo, a parte il fatto che opera su un grafo non orientato, differisce dall'algoritmo di Dijkstra solo nel calcolo della distanza dei nodi, che non è la distanza dal nodo sorgente s, ma la distanza minima da un nodo precedentemente scelto, cioè il peso dell'arco utilizzato per connettere il nodo ad un nodo precedentemente visitato. Pertanto, è sufficiente sostituire nell'algoritmo di Dijkstra il calcolo della distanza indicata dalla variabile `d` che non è più

```
        d = dist[z] + w
```

Listing 10.8: Codice sostituito

ma semplicemente

```
        d = w
```

Listing 10.9: Codice sostituto

```
import WGraphL as g
import Algorithms as a

C = [ [0,1,4], [0,7,8],  [1,2,8],  [1,7,11], [2,3,7], [2,5,4], [2,8,2],
      [3,4,9], [3,5,14], [4,5,10], [5,6,2],  [6,7,1], [6,8,6], [7,8,7]
    ]
G1 = g.createGraph(9)
for [x,y,w] in C:
     g.insertEdge(G1,x,y,w)
     g.insertEdge(G1,y,x,w)
print("--- Grafo (Lista archi) ---")
g.printGraph(G1)

print()
print("--- Prim ---")
print(a.Prim(G1,0))
print("--- Prim1 ---")
print(a.Prim1(G1,0))
```

Listing 10.10: Programma principale

10.2 Algoritmo di Kruskal

L'algoritmo di Kruskal è un algoritmo ottimo utilizzato per calcolare il minimo albero ricoprente di un grafo non orientato e con gli archi con costi non negativi. Prende il nome dal matematico americano Joseph Kruskal che lo ideò e propose nel 1956.

L'algoritmo parte da una foresta di alberi (cluster) dove ciascun albero contiene un solo nodo. Ripetutamente prende l'arco di costo minimo non precedentemente scelto e se i nodi terminali appartengono a due alberi diversi, unisce i due alberi aggiungendo anche l'arco scelto. Nell'unire i due alberi (cluster), i nodi del cluster di dimesione più piccola cambiano etichetta, cioè vengono trasferiti nel cluster più grande.

```
1  def Kruskal(G):
2      V = []
3      for i in g.nodes(G):
4          for [j,w] in g.neighbors(G,i):
5              if i < j:
6                  V.append([[i,j],w])
7      B = h.heapSort(V)
8      C = []
9      for i in range(len(G)):
10         C.append([i])
11     T = []
12     for [[i,j],w] in B:
13         if C[i] != C[j]:
14             T.append([i,j,w])
15             if len(C[i]) >= len(C[j]):
16                 for k in C[j]:
17                     C[i].append(k)
18                     C[k] = C[i]
19             else:
20                 for k in C[i]:
21                     C[j].append(k)
22                     C[k] = C[j]
23     return T
```

Listing 10.11: Algoritmo di Kruskal

Analisi dell'algoritmo. Le istruzioni 2-6 inseriscono gli archi del grafo all'interno di una lista `L`. Gli elementi della lista sono coppie arco/peso; l'arco a sua volta è una coppia di nodi. L'istruzione 7 effettua un ordinamento della lista. La lista ordinata è denotata da `B`. Il vettore `C` di lunghezza n (numero di nodi) indica, per ciascun nodo, il cluster di appartenenza (denotato dall'identificatore di un nodo appartenente al cluster), mentre `T` indica l'albero ricoprente (spanning tree). `C[i]` $=$ `j` significa che il nodo `i` appartiene al cluster `j`. Ciascun cluster contiene i nodi che appartengono allo stesso sotto-albero in `T`.

Inizialmente $C[i] = i$ (istruzioni 8-10), cioè ciascun cluster contiene un solo nodo o, equivalentemente, ogni nodo appartiene ad un cluster avente lo stesso identificatore. L'albero ricoprente è inizialmente vuoto: $T = []$ (istruzione 11)-

L'algoritmo procede come segue: a ciascun passo prende un arco da `B`, seguendo l'ordinamento dei pesi): Se l'arco unisce due nodi appartenenti a cluster diversi unisce i due cluster e inserisce l'arco in `T`. Nell'unire i due archi cambia l'etichetta del cluster di dimensione minore.

Complessità: $O(m \cdot \log_2 m) = O(m \cdot \log_2 n)$

```
import WGraphL as g
import Algorithms as a

C = [ [0,1,4], [0,7,8],  [1,2,8],  [1,7,11], [2,3,7], [2,5,4], [2,8,2],
      [3,4,9], [3,5,14], [4,5,10], [5,6,2],  [6,7,1], [6,8,6], [7,8,7]
    ]
G1 = g.createGraph(9)
for [x,y,w] in C:
     g.insertEdge(G1,x,y,w)
     g.insertEdge(G1,y,x,w)

print("--- Grafo (Lista archi) ---")
g.printGraph(G1)

print()
print("--- Kruskal ---")
print(a.Kruskal(G1))
```

Listing 10.12: Programma principale

10.3 Codifica di Huffman

Nella teoria dell'informazione, per codifica di Huffman si intende un algoritmo di codifica dei simboli usato per la compressione di dati, basato sul principio di trovare il sistema ottimale per codificare stringhe basato sulla frequenza relativa di ciascun carattere.

La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, risultando in un codice senza prefissi (cioè in cui nessuna stringa binaria di nessun simbolo è il prefisso della stringa binaria di nessun altro simbolo) che esprime il carattere più frequente nella maniera più breve possibile. È stato dimostrato che la codifica di Huffman è il più efficiente sistema di compressione di questo tipo: nessun'altra mappatura di simboli in stringhe binarie può produrre un risultato più breve nel caso in cui le frequenze di simboli effettive corrispondono a quelle usate per creare il codice.

La tecnica è basata sulla creazione di un albero binario di simboli:

1. Inserisci le n coppie (symbolo,frequenza) in un Heap.
2. Ripeti i seguenti passi $n-1$ volte:
   (a) Estrai due coppie (s_1, f_1), (s_2, f_2);
   (b) crea un nuova coppia $([\text{"}+\text{"}, (s_1, f_1), (s_2, f_2)], f_1 + f_2)$ dove il primo elemento è un albero (il simbolo "+" non è rilevante);
   (c) Inserisci la nuova coppia nell'Heap.
3. estrai l'unica coppia $(T, 100)$ presente nell'Heap;
4. assegna ai simboli in T una codifica in base alla loro posizione nell'albero.

```
def leaf(T):
    return T[0] in ['a','b','c','d','e','f']

def copyList(C):
    D = []
    for i in range(len(C)):
        D.append(C[i])
    return D

def codeGeneration(T,S,C):
    if leaf(T):
        D = copyList(C)
        S.append([T[0],D])
    else:
        C.append(0)
        codeGeneration(T[0][1],S,C)
        C.pop()
        C.append(1)
        codeGeneration(T[0][2],S,C)
        C.pop()
```

Listing 10.13: Funzioni ausiliarie

```
def Huffman(L):
    Q = h.createHeap()
    n = len(L)
    for z in L:
        h.insertHeap(Q,z)
    for i in range(n - 1):
        A = h.deleteMin(Q)
        B = h.deleteMin(Q)
        C = [['+', A, B], A[1] + B[1]]
        h.insertHeap(Q, C)
    T = h.deleteMin(Q)
    print(T)
    S = []
    codeGeneration(T, S, [])
    return S
```

Listing 10.14: Huffman coding

```
L = [ ['a',45], ['b',13], ['c', 12], ['d', 16], ['e',9], ['f', 5] ]
T = Huffman(L)
print(T)
```

Listing 10.15: Main

Complessità: $O(n \cdot \log_2 n)$, dove n indica il numero di caratteri.