Algoritmi e Strutture Dati

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

ufficio: U14 primo piano stanza 10-10
ricevimento: mandare mail dando tre alternative
nelle mail ricordare matricola
Libro: Cormen, Leiserson, Rivest, Stein "Introduzione agli algoritmi e alle strutture dati" (circa i primi 13 capitoli)
Eserciziario: Sedgewick
Esame: scritto (circa 4 esercizi di scrittura di algoritmi e domande di teoria), ci sono i parziali, a Luglio posso recuperarne uno. No orale
Nel corso si ricerca l'efficienza in a termini di tempo di esecuzione e spazio nella memoria (questa è la parte di strutture dati).
Questi appunti sono presi a lezione e sono stati anche integrati con gli appunti di *Chiodini* (`https://mega.nz/#F!LoRHWSQK!yaWdV5a478P5mcrOwHT3Jg!7kpFyAZZ`). Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

Iterazione

Definizione 1. *Un algoritmo è una sequenza di istruzioni elementari che consentono di risolvere un problema computazionale. Elementari perché l'esecutore (nel nostro caso il compilatore) può comprendere i vari step dell'algoritmo. Un algoritmo prende un input ed elabora i dati, dando un output.*
Un problema computazionale è una domanda a cui dare una risposta. Dipende dalle proprietà dell'input, proprietà dell'output, da come sono legati input e output (le proprietà della soluzione)
Un'istanza è uno dei casi del problema (una certa sequenza di numeri etc)

Esempio 1. *Ordinamento:*
input: $< a_1, ..., a_n >$ *output:* $< a_1^{'}, ..., a_n^{'} >$*, con* $a_1^{'} < ... < a_n^{'}$ *, che è una permutazione dell'input*

Dato un problema ci sono più algoritmi di soluzione (anche aggiungere codice inutile cambia comunque la natura dell'algoritmo, senza modificarne il funzionamento, facendogli comunque risolvere il problema). Ci sono ovviamente algoritmi diversi in tempi di tempi e spazio. Devo cercare il migliore.

Esempio 2 (divisione)**.** *Si ha:*

```
dimmi A
dimmi B
  sottrai B da A finché non arrivi a 0
  Conta le volte che fai la sottrazione
```

Ma non funziona se c'è un resto, il compilatore non vede 0 e va avanti all'infinito. ***L'algoritmo non è corretto, perché c'è almeno un input che non da risposta corretta***

Riprovo:

```
dimmi A
dimmi B
  sottrai B da A finché non arrivi a un valore <=0
  Conta le volte che fai la sottrazione
```

Ma anche questo non va, con A o $B \leq 0$

Se scrivo un Algoritmo di ordinamento A e uno B, come scelgo il più veloce? uno può essere più veloce su array più piccoli o magari non capisce se uno è già ordinato. Ho quindi:

$$tempo(n) \;\; funzione\ di\ n = |x| \in \mathbb{N}$$

con x input e n numero di dati in input. La funzione mi dice il numero di istruzioni svolte. Per esempio $A \to tempo(n) = 10 \cdot n$ e $B \to tempo(n) = 40 \cdot n^2$, quindi vince A.
Se ho tipo $A \to tempo(n) = 100000000 + 10 \cdot n$ e $B \to tempo(n) = 40 \cdot n^2$, il migliore dipende da n, ma a priori non so n, B è più veloce con n basso ma poi A risulta migliore; A è migliore quando comunque i tempi sono già ampi, su pochi dati sono tutti veloci; quindi vince A. Si sceglie il migliore su tempi importanti.
Si hanno cambiamenti anche in base agli x (penso ad un vettore ordinato ed uno no entrambi di n elementi). Si danno un *inf* (caso migliore, che non è il migliore in assoluto, quello sarebbe un algoritmo che non deve far niente, ma potrebbe essere, nell'esempio di ordinamento (non in tutti), un vettore già ordinato; si ricorda che per alcuni algoritmi il caso migliore potrebbe non verificarsi mai) e un *sup* (caso peggiore) dei tempi e si fa un tempo medio basato sulla frequenza delle casistiche.

Esempio 3. *Ho tre algoritmi, con la loro funzione tempo:*

- $A = 10000 \cdot n$
- $B = 100 \cdot n^2$
- $C = 2^n$
  Avrò il seguente esempio al variare di n ($''$ sono secondi):

$$\begin{pmatrix} \setminus & n = 20 & n = 100 \\ A = 10000 \cdot n & 0,2'' & 1'' \\ B = 100 \cdot n^2 & 0,4'' & 1'' \\ C = 2^n & 1'' & 3 \times 10^6 anni \end{pmatrix}$$

Algoritmi esponenziali non sono evidentemente fattibili e ragionevoli.

L'hardware aiuta a parità di algoritmo. A noi interessa la funzione in n, se è lineare, parabolica, esponenziale etc... $1 \cdot n \sim 100 \cdot n$ per noi (si vedrà più avanti il ragionamento coi limiti asintotici), ci interessa la differenza tra, per esempio le funzioni n, n^2 e α^n. Il tempo dato dalla *funzione tempo* è dato in *istruzioni*, non in *secondi*.

Esempio 4. *Dato un vettore V di elementi e un intero k, trovo k in V:*

```
TrovaValore(V[] int, k int){
  for i=1 to length(V)
    if V[i]==k
       return(i) //trovato
  return(-1) //caso non trovato
}
```

ho tempo$(n) = n$ ma col while ho già la condizione di blocco, e non devo cercare i vari break ad ogni istruzione del for. Il for lo uso quando so il numero di istruzioni. Nel nostro esempio potrei avere anche solo un istruzione, trovando k subito. Avrò quindi;

```
TrovaValore(V[] int, k int){
  i=1
  while(V[i]!=k) and (i<=length(V)) /* formalmente i controlli vanno invertiti *
    i++
    if(i>n)
       return(-1) /* l'elemento non è nel vettore */
     else
      return(i) /* l'elemento è trovato alla posizione i */
}
```

(nel corso i vettori partono da 1 e vanno a n per comodità nel conto della funzione tempo) Calcolo il tempo di esecuzione: conto le istruzioni:

- *i=1, una volta*
- *while(...),* $1 + t(while)$*, le volte in cui il while è vero + 1*
- *i++,* $1 \cdot t(while)$
- *if* $(i > n)$*, una volta*
- *return(-1), una volta o zero volte, ovvero:* $1 \cdot t(if)$
- *return(i),* $1 \cdot f(if)$

Si ha quindi:

$T(n) = 1+(1+t(while))+t(while)+1+1\cdot t(if)+1\cdot f(if) = 3+2\cdot t(while)+1\cdot t(if)+1\cdot f(if)$

Si hanno:

- ***caso migliore:*** *k è in posizione 1 ($t(while) = 0$ e primo if falso) quindi $T_{migliore} = 3 + 2 \cdot 0 + 1 \cdot 0 + 1 \cdot 1 = 4$*
- ***caso peggiore****: non trovo k ($t(while) = n$ e secondo if falso) quindi $T_{peggiore} = 3 + 2 \cdot n + 1 \cdot 1 + 1 \cdot 0 = 4 + 2 \cdot n$*
- ***caso medio:*** *ipotizzo un $t(while) = \frac{n}{2}$ e quindi un $T_{medio} \sim n$*

2.1 Ricerca Dicotomica o Ricerca Binaria

Nel calcolo nella qualità di un algoritmo bisogna considerare anche il tempo medio dello stesso. Nel caso della ricerca binaria, considerando equivalente la probabilità del dato ricercato si può calcolare che il tempo medio è $\frac{n}{2}$. Oltre al caso migliore e al caso peggiore si ha quindi anche il *caso medio*, che è quello che accade la maggior parte delle volte.
Nell'algoritmo in questione ad ogni iterazione elimino metà degli elementi tra i quali sto cercando l'elemento. Ecco l'algoritmo:

```
sx=1
dx=length(A)
do
  m=(sx+dx)/2
  if A[m]==k
    trovato=true
  else
    if A[m]>k
      dx=m-1
    else
      sx=m+1
while (trovato==false) and (sx<=dx)
if trovato
  return(m)
else
  return(-1)
```

passiamo ora all'analisi dei tempi:

```
sx=1 ---> 1
dx=length(A) ---> 1
do
  m=(sx+dx)/2 ---> (T_w+1)
  if A[m]==k ---> (T_w+1)
    trovato=true ---> 0/1 (=T_if1)
  else
    if A[m]>k ---> (T_w+1)
      dx=m-1
    else ---> (T_w+1)
      sx=m+1
while (trovato==false) and (sx<=dx) ---> (T_w+1)
if trovato ---> 1
  return(m)
else ---> 1
  return(-1)
```

Si ha quindi: $T(n) = 5 + 5 \cdot (T_w + 1) + 1 \cdot T_{if\,1}$ Si hanno quindi:

- **Caso Migliore**: l'elemento è a metà, $T_w = 0$ quindi $T_{migliore}(n) = 5 + 1 + 1 = 11$
- **Caso Peggiore**: l'elemento non c'è e $T(n) \sim \log_2 n$

Si ha quindi che:

passo	**Elementi Rimasti**
$1°$	$\frac{n}{2}$
$2°$	$\frac{n}{4}$
$3°$	$\frac{n}{8}$
$\cdots$	$\cdots$
$r°$	$\frac{n}{2^r}$

Inoltre si ha che $\frac{n}{2^r} = 1$ in quanto all'ultimo passo avrò solo un elemento, quindi: $n = 2^r \rightarrow r = \log_2 n$, ignoro gli altri moltiplicatori, tanto non influiscono sul risultato con grandi quantità di dati.
Inoltre $\log(n)$ è meglio di n (lo si vede anche dl grafico), gerarchia degli infiniti. Inoltre il tempo medio sarà $\frac{\log_2(n)}{2}$. Non si aggiunge il controllo che il k cercato sia fuori dai limiti perché, per migliorare così due casi soltanto, sto peggiorando di molto tutti gli altri, si avrebbe un esecuzione in più ad ogni di ciclo.

2.1.1 Convenzioni dello pseudo codice

Si hanno *for, while* e *do-while* come cicli. Non si hanno cicli con *break* o *return*, al più si esce con i *boolean*. Come condizioni di test si hanno *if, then* e *else*. Il codice va indentato, con commenti che indicano cosa intendo fare:

```
condizioni
ciclo /*scopo del ciclo*/
  condizioni del ciclo
  altro ciclo
    condizioni altro ciclo
  sempre nel primo ciclo
  if
    istruzioni
  else
    istruzioni
fuori dal primo ciclo
```

Per l'assegnamento si usa il simbolo "=", che non è il test, quello è "==".
Si usano solo variabili locali invece array e metodi sono globali.
Gli array vanno da 1 a n, per avere facile la funzione tempo. Si ha che length(vettore) mi da la lunghezza del vettore.
Si distinguono procedure (non restituisce nulla, è una *void*) e funzioni (restituiscono un tipo, *int, String, boolean, double, float etc...*)e si indicano come funzione/procedura(input), esempio: *somma(int a, int b)* e se nel codice ho *c=somma(1, 2)* avrò *a=1* e *b=2*. Volendo potrei lavorare anche con gli indirizzi di memoria. Il programma gira sulla *RAM*, che qui però è *Random Access Machine* e non è esattamente la RAM che si usa normalmente.Il vantaggio è che in questa posso accedere ad ogni zona di memoria alla stessa velocità, la memoria è quindi sempre ad accesso diretto, inoltre non ho idealmente limiti di memoria. Questa macchina ha input, output, operatori matematici, può fare salti condizionati e può modificare le zone di memoria. Inoltre è a singolo processore. Questa macchina rappresenta un calcolatore standard.

2.2 Algoritmi di Ordinamento

2.2.1 Selection Sort

Si ha il seguente algoritmo:
SelectionSort(v[ ])

```
for i=1 to n-1
  p=i /*salvo l'indice della posizione del primo, il più piccolo*/
  for j=i+1 to n /*il j-esimo più piccolo lo cerco da i*/
    if A[j]<A[p]
      p=j /*j è il più piccolo*/
  app=A[i]
  A[i]=A[p]
  A[p]=app
```

Controllare se un elemento è uguale a se stesso (ovvero quando scambio il numero alla stessa posizione) comporta una perdita di tempo. Lavorare coi puntatori comporterebbe una perdita di spazio. valuto i tempi:

- il primo for (c_1) cicla $n \cdot c_1$ volte, non entra l'ultima volta del for
- $p = i$ (c_2) perché non cicla come il for viene eseguito $(n-1) \cdot c_2$
- il secondo for viene eseguito circa $n \cdot n = n^2$ volte, meno perché n^2 si avrebbe con i due for che partono dallo stesso indice, ma non è questo il caso. Più in preciso fa: $n + (n-1) + (n-2) + \cdots + \underbrace{1}_{n-(n-1)}$ che è:

$$c_3 \cdot \sum_{k=n}^{1} k = c_3 \cdot \sum_{k=1}^{n} k$$

- l'if, c_4, viene eseguito $c_4 \cdot \sum_{k=1}^{n-1} k_2$, una volta meno del for
- $p = j$, c_5, viene eseguito $c_5 \cdot t_{if}$
- tutto il resto lo fa in $3 \cdot c_6 \cdot (n-1)$, c_6 per abbreviare, tanto tutte quelle iterazioni si eseguono senza esclusioni

In totale si ha:

$$T(n) = (n \cdot c_1) + ((n-1) \cdot c_2) + c_3 \cdot \sum_{k=1}^{n} k + c_4 \cdot \sum_{k=1}^{n-1} k_2 + c_5 \cdot t_{if} + 3 \cdot c_6 \cdot (n-1)$$

$$\downarrow$$

$$T(n) = (n \cdot c_1) + ((n-1) \cdot c_2) + c_3 \cdot \frac{n \cdot (n+1)}{2} + c_4 \cdot \frac{(n-1) \cdot (n)}{2} + (c_5 \cdot t_{if}) + (c_6 \cdot (n-1))$$

Inizio a ragionare con gli asintotici a $n = +\infty$ quindi si ha:

$$t(n) \sim (c_1 + c_2 + 3 \cdot c_6) \cdot n + (c_1 + c_4) \cdot \frac{n^2}{2} + c_5 \cdot t_{if}$$

- **Caso migliore:** si ha quando $t_{if} = 0$, il resto viene eseguito lo stesso, ovvero quando il primo numero del vettore è il più piccolo, poi in seconda il secondo etc...ovvero quando ho un vettore già ordinato. $T(n) \sim (c_1 + c_2 + 3 \cdot c_6) \cdot n + (c_1 + c_4) \cdot \frac{n^2}{2}$
- **Caso peggiore:** si ha quando $t_{if} = 1$ sempre, quindi quando ho un array ordinato al contrario. Ma dopo la metà ha già messo a posto a 2 a 2 tutto il vettore. Quindi: $T(n) = \sum_{i=1}^{\frac{n}{2}} i + (c_3 + c_4 + c_5) \cdot \frac{n^2}{2}$
- **Caso medio**: non ha una formula definita ma si muove come n^2

Ha il vantaggio è che ordina in loco, ovvero con un numero costante di variabili

2.2.2 Bubble Sort

Si ha un algoritmo migliore del selection. Senza analizzare l'algoritmo si ha che $T(n) = n^2$ Ha come caso peggiore il primo come più grande è l'ultimo come più piccolo

```
BubbleSort(A[])
  scambio = true
  while scambio do
    scambio = false
    for i = 0 to length(A)-1
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        scambio = true
```

- **caso migliore:** $T(n) \sim n$
- **caso peggiore:** $T(n) \sim n^2$
- **caso medio:** statisticamente metà del caso peggiore $T(n) \sim \frac{n^2}{2} \sim n^2$

2.2.3 Insertion Sort

Questo algoritmo è iterativo e agisce in loco. Smile a come si ordina un mazzo di carte. Presa la seconda carta la sistemo al posto giusto, prima o dopo la prima, presa la terza procedo come sopra, partendo dall'ultimo e tornando indietro. Non serve scambiare tutte le volte, tengo da parte il valore da inserire ordinato e spostare di uno a destra tutti i valori maggiori del valore da inserire:

```
insertionSort(A[])
  for i=2 to n
    c=A[i]
    p=i-1;
    while(c<A[p]) and p>0 /*se no outofbound */
      A[p+1]=A[p]
      p--
    A[p+1]=C
```

Tempi:

- for: $c_1 \cdot (n-1) + c - 1 \cdot 1$
- c=A[i]: $c_2 \cdot (n-1)$
- p=i-1: $c_3 \cdot (n-1)$
- while: $c_4 \cdot T_{w_i} + c_4 \cdot 1$
- A[p+1]=A[p]: $c_5 \cdot T_{w_i}$
- $p--$: $c_6 \cdot T_{w_i}$
- A[p+1]=C: $c_7 \cdot (n-1)$

quindi:

$$T(n) = c_1 \cdot (n-1) + c - 1 \cdot 1 + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot T_{w_i} + c_4 \cdot 1 + c_5 \cdot T_{w_i} + c_6 \cdot T_{w_i} + c_7 \cdot (n-1)$$

- **tempo migliore:** si ha con $T_{w_i} = 0$ quindi col vettore ordinato

- **caso peggiore:** $T_{w_i} = 1$: $(c_1 + c_2 + c_3 + c_7) \cdot (n-1) + c_1 + (c_4 + c_5 + c_6) \cdot \sum_2^n i - 1 = (c_1 + c_2 + c_3 + c_7) \cdot (n-1) + c_1 + (c_4 + c_5 + c_6) \cdot \frac{(n-1) \cdot (n)}{2}$

- **tempo medio:** mi aspetto una parte di array già ordinata, ergo $T_{w_i} \rightarrow \sum_1^n \frac{i-1}{2} = \frac{(n-1)\cdot(n)}{4}$ quindi

$$T(n) = (c_1 + c_2 + c_3 + c_7) \cdot (n-1) + c_1 + (c_4 + c_5 + c_6) \cdot \sum_1^n \frac{i-1}{2}$$

$$= (c_1 + c_2 + c_3 + c_7) \cdot (n-1) + c_1 + (c_4 + c_5 + c_6) \cdot \frac{(n-1)\cdot(n)}{4}$$

Nota 1. *Nel caso migliore InsertionSort è migliore del SelectionSort, indicativamente anche nel caso medio. Entrambi sono in loco, con un uso di memoria costante.*
Un algortitmo di ordinamento è detto ***stabile*** *se dati due numeri uguali il reciproco ordine che avevano prima dell'ordinamento viene mantenuto anche dopo.*

2.2.4 Limiti Asintotici

Per la valutazione di un algoritmo si usano gli asintotici.

- **limite superiore:** $f(n) = O(g(n))$ se

$$\{f(n) | \, \exists C,\, n_0 \geq 0 \; t.c. \; 0 \leq f(n) \leq C \cdot g(n)\}$$

  ,quindi ho una funzione che limita da sopra f per esempio, $7 \cdot n^2 = O(n^2)$.

- **limite inferiore:** $f(n) = \Omega(g(n))$ se

$$\{f(n) | \, \exists C,\, n_0 \geq 0 \; t.c. \; 0 \leq C \cdot g(n) \leq f(n)\}$$

  , quindi ho una funzione che limita da sotto f, per esempio $\frac{1}{7} \cdot n^2 = \Omega(n^2)$

- una funzione è sia omega grande che o grande della stessa funzione sse le due funzioni sono dello stesso ordine di grandezza, in questo caso si ha $f(n) = \Theta(g)(n))$

InsertionSort, per esempio, ha $O(n^2)$ come caso peggiore e $\Omega(n)$ come caso migliore, mentre SelectionSort ha $O(n) = \Omega(n)$, caso migliore e peggiore dello stesso ordine. Se ho un polinomio generico $P(k) = \alpha_k x^k + ... + \alpha_0$ si ha che $O(P(k)) = x^k$ e $\Omega(P(k)) = x^k$

Esercizio 1. *ho due vettori, calcolo in modo iterativo quanti sono in comune (se non specificato si intende anche se ripetuti). calcolo i tempi.*
v1 lungo m e v2 lungo n

```
compara(v1[],v2[])
  conta=0 /*c x 1*/
  for i=1 to length (v2) /*c x n*/
    j=1 /*c x n*/
    while v2[i]!=v1[j] and j<=length(v1) /*c x sum (da 1 a n) tw_i*/
      j++  /*c x sum (da 2 a n) tw_i*/
    if(j<=length(v1)) /*c x sum (da a a n) tif_i*/
      conta++
  return(conta) /*c x 1*/
```

Quindi: $T(n) = 2 \cdot c + 3 \cdot c \cdot n + 2 \cdot \sum_1^n t_{w_i} + c \cdot t_{if}$

- ***caso migliore:*** *v_2 contiene un solo elemento ripetuto in ogni posizione uguale al primo elemento di v_1. Quindi il while lo eseguo una volta e* $T(n) = 2 \cdot c + 4 \cdot c \cdot n = \Omega(n)$
- ***caso peggiore:*** *eseguo il while più volte possibile, ovvero quando i due vettori sono totalmente diversi. Quindi $t_{w_i} = m \, \forall i$ e l'if sempre falso e quindi:* $T(n) = 2 \cdot c + 3 \cdot c \cdot n + 2 \cdot c \cdot n \cdot m + c \cdot 0$. *Se $n \sim m$ allora* $T(n) = O(n^2)$

Esercizio 2. *ho due array con n bit (o 0 o 1) faccio la loro somma in un terzo array che parte da 0 e non da 1, ricordo che il bit meno significativo è a destra (ciclo all'inverso)*

```
SommaBitBit(A[],B[])
  riporto=0 /*c x 1*/
  for i=n downto 1 /*c x n*/
    C[i]=A[i]+B[i]+riporto /*c x n*/
    if c[i]>1 /*c x n*/
      c[i]=c[i]-2
      riporto=1 /*2 x c x t_if*/
    else
      riporto=0 /*c x f_if*/
  c[0]=riporto
```

tempi: $T(n) = 2 \cdot c + 3 \cdot c \cdot n + 2 \cdot c \cdot t_{if} + c \cdot f_{if} \sim n$

- ***Caso Peggiore:*** *if sempre vero quindi quando c'è sempre riporto, ci sono più input peggiori. Quindi:* $A[n] = B[n] = 1$ *e* $A[k]$ *or* $B[k] = 1, \ \forall 1 \leq k \leq n-1$. *Avrò comunque* $T(n) = O(n)$

- ***caso migliore:*** *non ho mai riporto... ma avrò comunque* $T(n) = \Omega(n)$

Quindi si ha $t(n) = \Theta(n)$

Esercizio 3. *Calcolo i tempi di:*

```
z=n /*c x 1*/
t=0 /*c x 1*/
while z>0  /*c x t_w ovvero log_2 z +1*/
  x=z mod 2 /*c x t_w*/
  z= z div 2 /*c x t_w*/
  if x=0 /*c x t_w*/
    for i=1 to n /*c x n x t_if*/
      t++
return(t)
```

$T(n) = 3 \cdot c + 4 \cdot c \cdot t_w + 2 \cdot c \cdot t_{if} \cdot n$

- ***caso peggiore:*** *si ha quando è vero l'if, ovvero quando* $n = 2^h$ *perché il resto della divisione ha resto 0 sempre (rappresentazione binaria fatta solo da 0). Quindi* $T(n) = 3 \cdot c + 4 \cdot c \cdot \log_2(n) + 2 \cdot c \cdot n \cdot \log_2(n) = O(n \cdot log(n))$

- ***caso migliore:*** *non entro mai nell'if, quando ho sempre resto 1 (rappresentazione binaria fatta solo da 1) quindi* $n = 2^h - 1$. *Ergo* $T(n) = 3 \cdot c + 4 \cdot c \cdot \log(n) = \Omega(log(n))$

Esercizio 4. *Calcolo i tempi di:*

```
for i=1 to n-1 /*c x (n-1)*/
  for j=i+1 to n /*c x sum (da i a n-1) i+1 */
    for k=1 to j  /*c x sum (da 1 a n) (sum (da i+1 a n) j)*/
      r++
```

conti completi su appunti di Gabriele

$$T(n) = 2 \cdot c + c \cdot (n-1) + c \cdot \sum_{i=1}^{n-1}(n-i) + 2 \cdot c \cdot \sum_{i=1}^{n-2}\sum_{j=i+2}^{n} j$$

$$\dots$$

$$= 2 \cdot c + c \cdot n - c + \left[n \cdot (n-1) - \frac{(n-1) \cdot n}{2} \right] + \cdot c \cdot \sum_{i=1}^{n-1} \left[\frac{n \cdot (n-1)}{2} - \frac{i \cdot (i+1)}{2} \right]$$

$$\sim 2 \cdot c + c \cdot n - c + c \cdot \left[\frac{(n-1) \cdot n}{2} \right] + c \cdot \left[(n-1) \cdot n^2 - \frac{(n-2) \cdot n}{2} - \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6} \right]$$

$$= \cdots = 2 \cdot c + c \cdot n - c + c \cdot \left[\frac{(n-1) \cdot n}{2} \right] + c \cdot \left[\frac{4 \cdot n^3 - 3 \cdot n^2}{6} \right] = \Theta(n^3)$$

I tempi non variano in base all'input in questo algoritmo

Esercizio 5 (matrice quadrata)**.** *Calcolo i tempi di un algoritmo che identifica una matrice simmetrica:*

```
boolean Simmetrica(M[1..n,1..n])
  r=1 /* c */
  c=2 /* c */
  do
    while M[r,c]==M[c,r] and c<=m /* c x T_w1 */
       c++ /* c x T_w1 */
       if c>m /* c x T_w2 */
         r++ /* c x T_if */
         c=r+1 /* c x T_if */
  while M[r,c]==M[c,r] and r<n /* c x T_w2 */
  if r>n /* c */
    return(true) /* c x T_if */
  else
    return false /* c x F_if */
```

Calcolo i tempi:

$$T(n) = 2 \cdot c + 2 \cdot c \cdot T_{w_1} + 2 \cdot c \cdot T_{if} + 2 \cdot c \cdot T_{w_2} + 2 \cdot c$$

- ***Caso migliore:*** $A[1,2] \neq A[2,1]$ *quindi so già che non è simmetrica e* $T_{w_1} = 0$, $T_{w_2} = 0$, $T_{if} = 0$ *quindi:*

$$T(n) = 2 \cdot c + 0 + 0 + 2 \cdot c + 2 \cdot c = 6 \cdot c = \Omega(1)$$

- ***Caso Peggiore:*** *la matrice è effettivamente simmetrica, ho* $T_{w_2} = T_{if} = n - 1$, $T_{w_1} = (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{i=2}^{n-1} i$ *e quindi si ha:*

$$T(n) = 2 \cdot c + 2 \cdot c \cdot \sum_{i=2}^{n-1} i + 2 \cdot c \cdot (n-2) + 2 \cdot c \cdot (n-1) + 2 \cdot c = O(n^2)$$

In media prevedo che calcoli un quarto della matrice.

Capitolo 3

Ricorsione

3.1 Ricorsione semplice

La ricorsione è legata al concetto di induzione (che permette per esempio di rappresentare i numeri naturali $\mathbb{N}$, dato $n \in \mathbb{N}$ allora anche $n+1 \in \mathbb{N}$, sapendo che $1 \in \mathbb{N}$, quindi per esempio parto da k, retrocedo a paso di $k-1$ fino a $k=1$). Nell'induzione si ha il caso base e il caso passo.
Gli algoritmi ricorsivi sono spesso di più facile scrittura di un algoritmo iterativo e sono spesso più efficienti, ma non sempre è l'esempio migliore. Possono esserci pi casi base:

Esempio 5 (Fattoriale)**.** *Si ha:*

```
int Fattoriale(n)
  if n==0
    return 1
  else
    ris=Fattoriale(n-1)*n
    return(ris)
```

Esempio 6 (Fibonacci)**.** *Si ha:*

```
int Fibonacci[n] /* n è il passo */
  if n==0
    return(0)
  if n==1
    return(1)
  else
    ris=Fibonacci(n-1)+Fibonacci(n-2)
    return(ris)
```

Un algoritmo ricorsivo ha il caso base e il caso generico. Devo essere certo di arrivare sempre al caso base, che non richiami se stesso. il caso base può non fare nulla ma c'è sempre. Possono esserci più casi base.

Esempio 7. *Si ha, per stampare un array di char:*

```
void Stampa(A[i], int i) /* A array di caratteri */
  if i<= A.length /* il caso base: i > A.length */
    print(A[i])/* se metto il print dopo lo stampa */
    Stampa(A[],i+1) /* stampo al contrario */
```

Il caso base non fa nulla ma c'è.

Esempio 8. *Si ha, per contare 5 in un array:*

```
int conta(A[], int p) /* p = posizione */
  if p>length(A) /* c */
    return(0) /* c x t_if */
  else
    r1=conta(A[],p+1) /* f_if x t_conta */
    if A[p]==5 /* c x f_if */
      r1++ /* c x t_if2 */
    return(r1) /* c x f_if */
```

Ne calcolo i tempi: $T(n) = 3 + T(n-1)$*, è un'equazione di ricorrenza*

Esercizio 6. *Somma valori array: ricorsivo − > sommo primo più gli altri, Dividi et impera − > primo più ultimo più centrali*

```
int somma(A[], int l, int r)
  if l>r /* c */
    return(0) /* c */
  if l==r /* c */
    return(A[l]) /* c */
  else
    tot=somma(A[],l+1,r-1) /* T(n-2) perché ho 2 celle in meno */
    tot=tot+A[l]+A[r] /* c */
    return(tot) /* c */
```

Calcolo i tempi: $T(n) = \begin{cases} 2 \cdot c & n = 0 \\ 3 \cdot c & n = 1 \\ 4 \cdot c + T(n-2) & n > 2 \end{cases}$ *, calcolo l'equazione di ricorrenza:* $T(n-2) = 4 \cdot c + T(n-4) = 4 \cdot c + [4 \cdot c + T(n-4)]$
$= 4 \cdot c + 4 \cdot c + [4 \cdot c + T(n-6)] = \cdots = k \cdot 4 \cdot c + T(n-2k)$ *voglio quindi* $n = 2 \cdot k \to k = \frac{n}{2}$*. Ottengo quindi* $2 \cdot n \cdot c + T(0) = 2 \cdot n \cdot c + 2 \cdot c = \Theta(n)$

3.2 Divide et impera

Con questa tecnica si dividono i problemi in sottoproblemi la cui quantità è una frazione della quantità totale. Si divide in fasi:

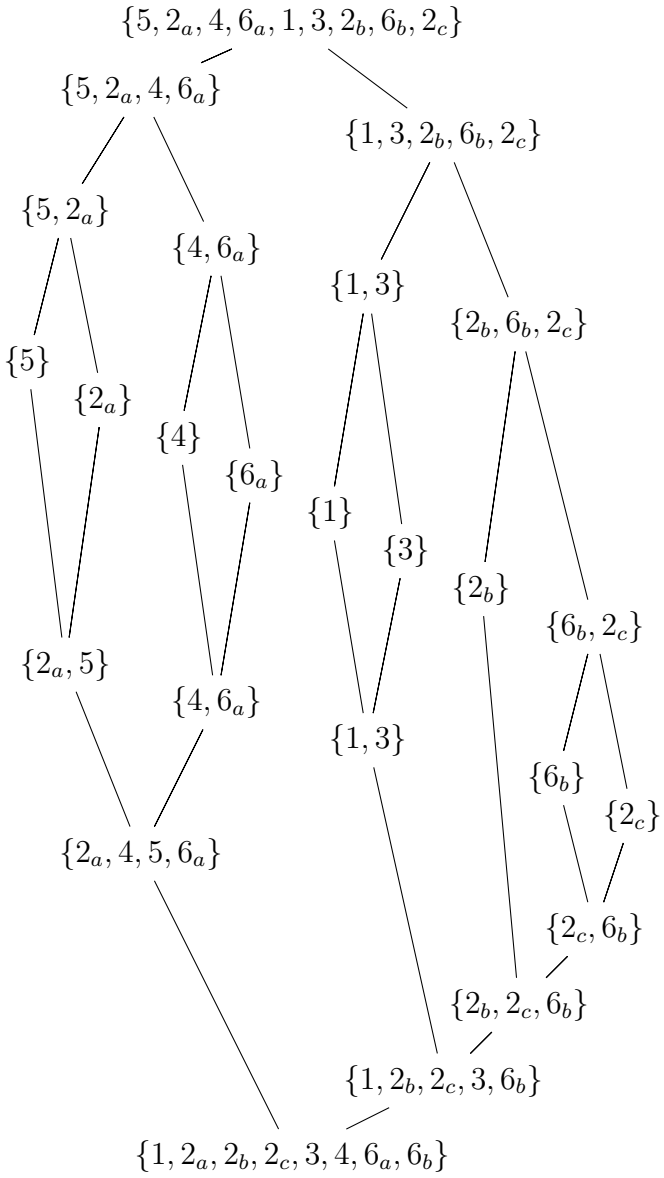
- **divide:** divido il problema in sottoproblemi, si procede iterativamente
- **impera:** si risolvono i vari sottoproblemi, si procede ricorsivamente
- **combina:** combino le soluzioni dei vari sottoproblemi per dare la risposta al problema principale, si procede iterativamente

Mergesort

le fasi sono così divise: quantità è una frazione della quantità totale. Si divide in fasi:

- **divide:** divide il problema in 2 sottoproblemi, i primi $\frac{n}{2}$ e i secondi $\frac{n}{2}$
- **impera:** ordina le due metà indipendentemente
- **combina:** fonde i due sottoarray ordinati

facciamo un esempio numerico sfruttando un dag:



Si ha quindi che MergeSort è quindi un algoritmo stabile.
Scriviamo ora l'algoritmo che è dato da 2 funzioni, con MergeSort che sarà la funzione chiamata nel main:

```
MergeSort(A[], int I, int F)
  if not I == F
    M = (I + F) / 2;
    MergeSort(A, I, M);
    MergeSort(A, M + 1, F);
    Merge(A, I, M, F);

Merge(A[], I, M, F)
  I_1 = I; I_2 = M + 1; I_B = I; /* 5c x T(while_1) */
  do
    if A[I_1] <= A[I_2]
      B[I_B] = A[I_1]
      I_1++
    else
      B[I_B] = A[I_2]
      I_2++
    I_B++
  while I_1 <= M AND I_2 <= F

  while I_1 <= M  /* 4c x T(while_2) */
    B[I_B] = A[I_1]
    I_1++
    I_B++
  while I_2 <= F  /* 3c x T(while_3) */
    B[I_B] = A[I_2]
    I_2++
    I_B++
  for I_B = I to F   /* 2c x n */
    A[I_B] = B[I_B]
```

Si ha quindi che la funzione *merge* ha

$$T(n) = t_{while_1} + t_{while_2} + t_{while_3} = F - I \sim n$$

quindi si ha: $T(n) = 5 \cdot c \cdot n + 2 \cdot c \cdot n = \Theta(n)$.

Calcolo ora i tempi dell'intero algoritmo (ricordando che è ricorsivo):

$$T_{MergeSort}(n) = \begin{cases} c & n = 1 \\ 2 \cdot c + \underbrace{2 \cdot T_{MergeSort}\left(\frac{n}{2}\right)}_{\text{2 chiamate ricorsive}} + \underbrace{c \cdot n}_{merge} \end{cases}$$

Quindi si avrà:

$$T_{MergeSort}(n) = 2 \cdot c + c \cdot n + \left[2 \cdot c + c \cdot \frac{n}{2} + 2 \cdot T\left(\frac{n}{4}\right)\right]$$

$$= 2 \cdot c + 2 \cdot c \cdot n + 2^2 \cdot c + 2^2 \cdot T\left(\frac{n}{2^2}\right) = \cdots + 2^2 \cdot \left[2 \cdot c + c \cdot \frac{n}{4} + 2 \cdot T\left(\frac{n}{2^3}\right)\right]$$

SI stanno però commettendo errori nello sviluppo dell'equazione di ricorrenza. Inoltre si sta avendo un vantaggio prestazionale a discapito dell'uso della memoria (si usa infatti un vettore d'appoggio).

Nota 2. *Si mostra ora la procedura per il calcolo del tempo di esecuzione di un Divide et Impera:*

$$T_{DivideEtImpera} \begin{cases} T_{Divide} + T_{Impera} + T_{Combina} \\ T_{CasoBase} \end{cases}$$

nel Divide et Impera si hanno a sottoproblemi con $\frac{n}{b}$ elementi (si ha quindi $a \geq 1$ e $b > 1$)

Torniamo al calcolo dei tempi del MergeSort, si ha:

$$T_{MergeSort}(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

Si ha:

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

Si mostra che $T(n) = n \cdot \log n$ per $n = 2^k$. Procedo per induzione:

- **caso base:**
$$K = 1 \longrightarrow T(n) = 2 = 2^1 \cdot \log_2 1 = 2$$

- **passo induttivo:**

$$T(2^{k+1}) = 2 \cdot T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1}$$

$$= 2 \cdot T(2^k) + 2^{k+1}$$
$$= 2 \cdot 2^k \cdot \log 2^k + 2^{k+1}$$
$$= 2^{k+1} \cdot (\log 2^k + 1)$$
$$= 2^{k+1} \cdot (\log 2?k + \log_2 2$$
$$= 2^{k+1} \cdot (\log(2^k + 1))$$
$$\downarrow$$
$$2^k = n$$
$$\downarrow$$
$$2 \cdot n \cdot \log(n+1) \sim n \cdot \log n$$

Nel MergeSort si ha il caso migliore e il caso peggiore che si muovono come $\Theta(n)$

3.2.1 Equazioni di Ricorrenza

Le equazioni di ricorrenza hanno solitamente la seguente forma:

$$\begin{cases} T(n) = T(n-1) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

Esistono tre metodi per risolvere le equazioni di ricorrenza:

- Iterativo (detto anche Albero di ricorsione)
- Sostituzione
- Esperto (detto anche Principale

Metodo Iterativo

Si può usare sia per algoritmi ricorsivi e per Divide et Impera. Ad ogni passo si prende il valore a destra dell'uguaglianza e lo si sostituisce, arrivando, dopo k passi ad una formula generale. Sempre k ci darà il caso base. Posso rappresentare questo metodo con l'albero delle chiamate ricorsive, guardando quanto è alto l'albero e quanto impiega ad ogni livello

Esempio 9. *Calcolo i tempi di:*

$$\begin{cases} T(N) = T(n-1) + 8 \\ T(1) = 6 \end{cases}$$

procedo nella seguente maniera:

$$\begin{aligned} T(n) = T(n-1) + 8 = [T(n-2) + 8] + 8 = T(n-2) + 2 \cdot 8 \\ = [T(n-3) + 8] + 2 \cdot 8 = T(n-3) + 3 \cdot 8 \\ = [T8n-4) + 8] + 3 \cdot 8 = T(n-4) + 4 \cdot 8 \\ = T(n-k) + k \cdot 8 \end{aligned}$$

per $k = n-1$ si ha:

$$T[n-(n-1)] + (n-1) \cdot 8 = T(1) + (n-1) \cdot 8 = 6 + (n-1) \cdot 8 = \Theta(n)$$

Altri esempi su sito e appunti di Chiodini

Metodo per Sostituzione

Si ipotizza un tempo di calcolo (si possono usare gli asintotivi con O e Ω lo si dimostra per induzione

Esempio 10.

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

Ipotizzo $O(n \cdot \log n)$ e dimostro per induzione:

$$T(n) = O(n \cdot \log n) \leq c \cdot n \cdot \log n$$

Serve una dimostrazione forte: ipotizzo $T(m)$ vera per $1 \leq m \leq n-1$ quindi si ha:

$$\begin{aligned} T(n) = 2 \cdot T\left(\frac{n}{\lfloor 2 \rfloor}\right) + n \leq 2 \cdot \left[c \cdot \frac{n}{\lfloor 2 \rfloor} \cdot \log \frac{n}{\lfloor 2 \rfloor}\right] + n \\ = c \cdot n \cdot \log \frac{n}{2} + n = c \cdot n \cdot (\log_2 n - \log_2 2) + n \\ = c \cdot n \cdot \log_2 n - c \cdot n + n \leq c \cdot n \cdot \log n \text{ se } c \geq 1 \end{aligned}$$

Analizzo ora il caso base:
$T(1) = 1$ *quindi voglio* $1 \leq c \cdot \log_2 1$ *ovvero* $1 \leq c \cdot 0$ *ovvero mai. testo fino a che non trovo* $T(3) = 2 \cdot T(1) + 3 = 29 + 3 = 5$ *che mi va bene, infatti* $5 \leq c \cdot 3 \cdot \log_2 3$

Metodo dell'Esperto

Posso usare questo metodo solo nel caso di un'equazione di ricorrenza di questo tipo:

$$\begin{cases} T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

dove:

- $a \cdot T\left(\frac{n}{b}\right)$ è l'Impera ed è $\sim n^{\log_b a}$
- $f(n)$ è il divide e il combina (ovvero la parte iterativa)

Si definiscono tre casi:

- **caso 1:** $n^{\log_b a} > f(n)$ quindi $T(n) \sim n^{\log_b a}$. Si hanno le seguenti condizioni necessarie: $f(n) = O(n^{log_b a - \epsilon})$ (con $\epsilon > 0$) e quindi $T(n) = \Theta(n^{log_b a - \epsilon})$
- **caso 2:** $n^{\log_b a} \cong f(n)$ quindi $T(n) \sim f(n) \cdot \log n$. Si hanno le seguenti condizioni necessarie $f(n) = \Theta(n^{\log_b a})$ e quindi $T(n) = \Theta(n^{\log_b a})$
- **caso 3:** $n^{\log_b a} < f(n)$ quindi $T(n) \sim f(n)$. Si hanno le seguenti condizioni necessarie: $f(n) = \Omega(n^{\log_b a + \epsilon})$ (con $\epsilon > 0$) e $a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$ (con $k < 1$) quindi $T(n) = \Theta(f(n))$

Esempio 11. *Risolvo:*

$$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n$$

Si ha: $f(n) = n$, $a = 9$ *e* $b = 3$.
Ho che $n^{\log_3 9} = n^2$ *quindi ho il primo caso:*
$f(n) = O(n^{\log_b a - \epsilon}) = O(n^{2-\epsilon})$ *Posso dire che* $\exists \epsilon : O(n^{2-\epsilon}) = n$*?*
Si $\forall \epsilon < 1$*, per esempio* $\epsilon = \frac{1}{2}$*. Quindi il Metodo dell'esperto è applicabile (nel primo caso) e si ha quindi* $T(n) = \Theta(n)$

Esempio 12. *Si può analizzare meglio il MergeSort:*

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

Si ha: $f(n) = \Theta(n)$ *e* $n^{\log_b a} = n^{\log_2 2} = n$
Posso applicare il Metodo dell'esperto nel secondo caso avendo così:

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

Esempio 13.

$$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + n \cdot \log n$$

Si ha: $f(n) = n \cdot \log n$ *e* $n^{\log_b a} = n^{\log_4 3}$ *e siamo nel terzo caso:*

$$f(n) = \Omega(n^{\log_4 3 + \epsilon}$$

se pongo $\epsilon = 1 - \log_4 3$ *ottengo* n*. Il terzo caso richiede una doppia verifica:*

$$3 \cdot \frac{n}{4} \cdot \log \frac{n}{4} \leq k \cdot n \log n$$

che vale per $k = \frac{3}{4}$ *infatti si ha:*

$$\frac{3}{4} \cdot n \cdot \log \frac{n}{4} \leq \frac{3}{4} \cdot n \cdot \log n$$

Si hanno quindi entrambi i requisiti e si può asserire che $T(n) = \Theta(n \cdot \log n)$

Esempio 14. *Calcolo i tempi di:*

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

Si ha: $n^{\log_b a} = n^{\log_2 2} = n$ *e* $f(n) = n \cdot \log n$*. Provo a procedere col terzo caso, dimostrando che:*

$$n \cdot \log n = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{1+\epsilon}) = \Omega(n \cdot n^{\epsilon})$$

Ma tale ϵ *non esiste in quanto* $n^{\epsilon} > \log n$ *infatti:*

$$\lim_{n \to \infty} \frac{n \cdot \log n}{n \cdot n^{\epsilon}} = 0, \quad \forall \epsilon > 0$$

Bisogna quindi applicare un altro metodo per risolvere l'equazione di ricorrenza

3.2.2 QuickSort

Differente dal MergeSort questo algoritmo divide l'array in due sottoarray (non a metà) "piccoli" e "grandi". La Impera ordina ricorsivamente i sottoarray mentre la combina non fa nulla quanto alla fine degli ordinamenti di "piccoli" e "grandi" il vettore sarà ordinato. La scelta degli elementi da mandare in "piccoli" e "grandi" viene fatta per mezzo di un pivot, scelto casualmente tra i vari valori. I più piccoli del pivot andranno in "piccoli" e i più grandi in "grandi". In generale si hanno le seguenti caratteristiche:

- ha un tempo medio di $n \cdot \log n$, ma con costanti nascoste dall'asintotico più piccole del MergeSort
- ha un tempo peggiore di n^2
- non è un algoritmo di ordinamento stabile
- Ordina in loco, usa solo una variabile di appoggio, quindi è efficiente in termini di memoria

La Divide è costituita dalla funzione *partition*. Esiste un metodo "originario", detto *metodo di Hoare* dal nome dell'inventore. Si propone l'algoritmo che usa il primo elemento come pivot:

```
void QuickSort(A[], Inizio, Fine)
  if Inizio < Fine
    M = Partition(A, Inizio, Fine)
    QuickSort(A, Inizio, M)
    QuickSort(A, M + 1, Fine)
int Partition(A[], I, F)
  pivot = A[I] /* sceglie pivot, ad esempio il primo elemento */
  sx = I - 1
  dx = F + 1
  while sx < dx
    do
      sx++
    while (A[sx] < pivot)
    do
      dx--
    while (A[dx] > pivot)
    if sx < dx
       scambia(A, sx, dx)
  return dx
```

Si osservano due cose:

- non serve controllare dx e sx, non andranno mau out of bound per come è fatto l'algoritmo
- la scelta del pivot potrebbe essere fatta mediante una media tra i vari elementi ma costerebbe $O(n)$. Qualsiasi elemento va bene tranne l'ultimo (si rischia di andare in loop se è il più grande di tutti, taglierei il vettore in uno nullo e uno con tutto il vettore).

Si calcolano i tempi di*Partition*, osservando che sx viene sempre incrementato almeno una volta e che dx viene sempre decrementato almeno una volta, entrambi fino ad un massimo di $\frac{n}{2}$ volte in contemporanea. Se invece il primo viene incrementato del tutto allora il secondo viene decrementato solo una volta. **In generale non so cosa accade ma so che la somma di incrementi e decrementi è n**. SI ha quindi che *Partition* richiede $T(n) = \Theta(n)$.
Calcoliamo ora *QuickSort* mediante la sua equazione di ricorrenza:

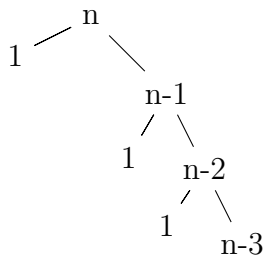
$$T_{QuickSort} = \begin{cases} c & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \end{cases}$$

Per il metodo dell'esperto si ha quindi: $n^{log_b a} = n^{\log_2 2} = n$ e quindi (secondo caso) $T_{QuickSort}(n) = n \cdot \log n$. Come il MergeSort.
Nel caso di una scelta di pivot totalmente sfavorevole si avrebbe:

$$\begin{aligned} T(n) &= T(1) + T(n-1) + c \cdot n = T(n-1) + c \cdot n + c \\ &= T(n-2) + c \cdot (n-1) + c + c \cdot n + c \\ &= T(n-3) + c \cdot (n-2) + c + c \cdot (n-1) + 2 \cdot c \\ &= T(n-k) + k \cdot c + c \cdot \sum_{i=0}^{n} i = O(n^2) \end{aligned}$$

Il caso migliore è dato quando il pivot è la media dei vari elementi, il caso peggiore quando il pivot è sempre l'elemento più piccolo, ovvero quando è sempre ordinato.

Mostriamolo anche con l'albero di ricorsione:



Si ha che:

$$T(n) = n + \sum_{i=2}^{n} i = n + \frac{n \cdot (n+1)}{2} = O(n^2)$$

Si ha quindi che QuickSort è compreso tra $n \cdot \log n$ e n^2. Lo si vede scrivendo la vera equazione di ricorrenza del QuickSort:

$$T(n) = T(Q) + T(n-Q) + \Theta(n) \text{ con } 1 \leq Q \leq n-1$$

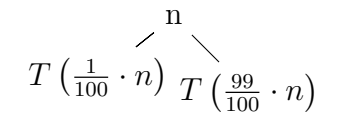
Si dimostra per induzione che $n+2$ limita i vari tempi di esecuzione, ovvero $T(n) \leq c \cdot n^2$.
Cerchiamo un Q che massimizza $T(n)$, per ipotesi si ha:

$$T(n) \leq c \cdot Q^2 + c \cdot (n-Q)^2 + \Theta(n)$$

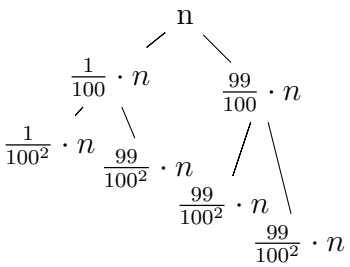
si hanno:

$$f: \ c \cdot Q^2 + c \cdot (n^2 - 2 \cdot n \cdot Q + Q^2) + \Theta(n)$$
$$f^{'}: \ 2 \cdot c \cdot Q + 2 \cdot c \cdot q - 2 \cdot n$$

e $f^{'} = 0$ con $Q = \frac{n}{2}$ ovvero il caso migliore.
Vogliamo ora domandarci se impiega più spesso $n \cdot \log n$ o n^2. Per farlo ipotizziamo il seguente albero:



che diventa:



Quindi mi basta valutare quando è profondo il ramo più a destra.
Il caso base volendo (nel caso estremo e più improbabile) è: $\left(\frac{99}{100}\right) \cdot n = 1 \rightarrow \frac{n}{\left(\frac{100}{99}\right)^k} = 1 \rightarrow n = \left(\frac{100}{99}\right)^k \rightarrow k = \log_{\frac{100}{99}} \cdot n$.
Quindi il tempo totale è: $T(n) = n \log_{\frac{100}{99}} n = \Theta(n \log n)$, tanto la base non mi interessa. Si ha quindi che anche se i tagli sono sbilanciati (restano una frazione) il tempo resta quasi sempre $n \cdot \log n$. Quindi in generale l'algoritmo è asintotico a $n \cdot \log n$.
Si ha però che il vero QuickSort è randomizzato e sceglie un pivot random:

```
Partition(A[], I, F)
  Q = random(I, F)
  scambia(A, I, Q)
  ... // resto della Partition
```

Dopo la *Partition* il pivot è sistemato e la *partition* torna$sx + 1$ e le chiamate ricorsive saranno:

```
QuickSort(A, I, r - 1)
QuickSort(A, I, r + 1)
```

Parte fatta a lezione: Quindi se l'array viene spezzato perfettamente in 2 parti ogni volta si ha il caso migliore:

$$T(n) = w \cdot T\left(\frac{n}{2}\right) + \Theta(n) \rightarrow \Omega(n \cdot \log m)$$

mentre l'opposto, il caso peggiore, con un array spezzato con un array di un elemento e l'altro con tutti gli altri, si ha:

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

Statisticamente sono più gli input che portano al caso migliore che quelli che portano al caso peggiore. Quindi è in generale meglio del MergeSort (volendo sul libro c'è il conto statistico).
Ragiono ora sulla veridicità di questa statistica, ecco un caso sbilanciato:

$$T(n) = T\left(\frac{n}{100}\right) + T\left(\frac{99 \cdot n}{100}\right) + \Theta(n)$$

(guardo l'albero fatto sopra).
Quindi il QuickSort impazzisce quando prendo un array ordinato ne prendo come pivot il primo elemento, in questo caso scelgo un altro pivot. Quindi non voglio usare il primo come pivot se è anche il primo elemento dell'array. La soluzione migliore è scegliere un pivot random (controllando non prenda l'ultima). L'algortimo del QuickSort non cambia mentre la *Partition* diventa:

```
PartitionRandom(A[],I,F)
  Q=random(I,F)
  scambia(a,I,Q) /* evito che il pivot sia l'ultimo elemento */
  ... /* come la Partition normale */
```

Potevo anche imporre un controllo a random senza fare lo scambia. Così ho un tempo $n \cdot \log n$ quasi in ogni caso, con costanti nascoste più piccole del MergeSort. In generale è quindi migliore.
non si può ottenere un algoritmo di ordinamento più veloce di $n \cdot \log n$

Esempio 15 (QuickSort random)**.** ***DA SISTEMARE***

$$1, 7, 10, 4, 2$$

random = 3 quindi pivot = 10

$$10, 7, 1, 4, 2$$

inverto 2 e 10 e spezzo perché 10 decresce ma non scambia mai, l'indice di due incrementa ma sarà sempre più piccolo del pivot 10

$$2, 7, 1, 4 \quad 10$$

pivot = 4

$$2, 1, 7, 4 \quad 10$$

pivot = 2

$$2, 1 \quad 7, 4 \quad 10$$

e infine:

$$1, 2, 4, 7, 10$$

esercizi

Esercizio 7. *ho un divide et impera che calcola così b:*

$$b = (2 + a_1) \times \cdots \times (2 + a_n)$$

Calcolo i tempi di una funzione che fa ciò, ovvero una

```
int calcola(A[],I,F)
  if I==f
    return (2+A[I])
  else
    m=(I+F)/2
    sx=calcola(A[],I,m)
    dx=calcola(A[],m+1,dx)
    tot=sx*dx
    return(tot)
```

tempi:

$$T(n) = \begin{cases} 2 \cdot c & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 4 \cdot c & \end{cases}$$

Esercizio 8. *trovo se c'è un valore nella posizione pari al suo valore in un array ordinato:*

```
boolean Posizione (A[],I,F)
  if(I==F)
    if A[i]==I
      return(true)
    else
      return(false)
  else
    m=(I+F)/2
    if(A[m]==m)
      return(true)

    sx=Posizione(A[],I,m-1)
      if(sx== true) /*non è nella prima metà non sarà nella seconda è ordinato*/
        return(true)
    else
      dx=Posizione(A[],m+1,F)
      return (dx)
```

Occhio che i conti sono su un algoritmo ottimizzato, da cercare *tempi del caso peggiore, nessun valore sta nella posizione con lo stesso valore come indice:*

$$T(n)\begin{cases}3\cdot c & n=1\\ T\left(\frac{n}{2}\right)+\Theta(1) & n>1\end{cases}$$

Caso migliore:

$$T(n)\begin{cases}3\cdot c & n=1\\ 4\cdot c & n>1\end{cases}$$

ovvero quando il valore medio è a metà dell'array.

Esercizio 9. *Algoritmo per vedere se vettore ha un numero pari di vocali: (assumo tutto maiuscolo)*

```
boolean Vocali(A[],I,F)
  if(I==F)
    if(A[i]=='A' or A[i]=='I'...) /* solo una vocale, ovvero dispari */
      return(false)
    else
      return(true)
  else
    m=(I+F)/2
    sx=vocali(A[],I,m)
    dx=vocali(A[],m+1,F)
    if(sx==dx) /* se entrambi hanno valori solo pari o solo dispari */
      return(true) /* daranno un pari */
    else
      return(false)
```

per i tempi si sempre il solito dei divide et impera con $b=2$

3.3 Sempre sugli algoritmi di ordinamento

Abbiamo notato che alcuni algoritmi di ordinamento, come *BubbleSort*, *SelectionSort* e *InsertionSort*, si muovono come n^2 mentre altri, *MergeSort* e *Quicksort*, si muovono come $n \cdot \log n$. Ci si chiede se è possibile scendere ancora col tempo, trovando un algoritmo, che sfrutta i confronti, che si muova come n. Si dimostra che non è possibile scendere sotto $n \cdot \log n$, si cerca di porre un lim, *Limite Inferiore*, al problema dell'ordinamento.
Si dimostra che qualunque algoritmo basato sui confronti richiede almeno $n \cdot \log n$. Si sfrutta un *Albero di Decisione*, che è un albero dove ogni etichetta rappresenta una domanda a cui può essere data risposta vera o falsa. Su ogni dell'albero c'è uno dei possibili ordinamenti, ne segue che per k domande ho 2^k foglie. Ne segue che posso scegliere n modi di ordinamento al primo passaggio, $n-1$ al secondo e così via. Si hanno quindi $n!$ possibili ordinamenti diversi. Si ha quindi:

$$2^k > n! \longrightarrow k = \log_2(n!)$$

Usando *l'approssimazione di Stirling*, $n! = \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \theta\left(\frac{1}{n}\right)\right)$.
Possiamo quindi dire che:

$$n! \sim \left(\frac{n}{e}\right)^n$$

e di conseguenza che:

$$k = \log_2 \left(\frac{n}{e}\right)^n < O(n \cdot \log n)$$

3.3.1 CountingSort

Si introducono degli algoritmi di ordinamento non basati su confronti. Si comincia col CountingSort, un algoritmo non in loco e stabile. Con un tempo di esecuzione di $O(n+k)$, dove k è la differenza tra il più grande elemento dell'array e il più piccolo. Non è un algoritmo lineare perché k non è costante e se questo range è molto grande l'algoritmo ha un tempo maggiore di $n \cdot \log n$, se invece il range è piccolo l'algoritmo è lineare in n. L'algoritmo cerca di tenere conto delle occorrenze dei vari elementi. Nell'algoritmo si hanno ben 2 vettori d'appoggio, più il vettore effettivo da ordinare, uno, B; da 1 n e uno, C, da 1 a k, il primo per ordinare e il secondo per contare.
L'algoritmo funziona in 4 parti:

- si azzera l'algoritmo usato per contare, C
- si scorre l'array da ordinare contando nell'array dedicato al contare, C, il numero di occorrenze di un certo valore i. Alla fine in C[i] si avranno il numero di occorrenze di i

- $\forall i: \ 2 \leq i \leq k$ si ha C[i]=C[i]+C[i-1]
- passa tutti gli elementi del vettore da riordinare dalla fine all'inizio, in modo da avere un algoritmo stabile, e li piazza in B secondo il contenuto di C

Ecco l'algoritmo:

```
CountingSort(A[], B[], C[])
  for i = 1 to K /* fase 1 */
    C[i] = 0
  for i = 1 to N /* fase 2 */
    pos = A[i]
    C[pos]++
  for i = 2 to K /* fase 3 */
    C[i] += C[i - 1]
  for i = N down to 1 /* fase 4 */
    posC = A[i]
    posB = C[posC]
    B[posB] = A[i]
    C[posC]--
```

Si ha quindi il seguente tempo di esecuzione:

$$T(n,k) = 4 \cdot c \cdot k + 8 \cdot c \cdot n = \Theta(n+k)$$

3.3.2 RadixSort

Questo algoritmo di ordinamento è usato per ordinamenti su più chiavi diverse, ad esempio per ordinare un insieme di elementi in base ad una loro caratteristica, a parità di quella caratteristica si usa un'altra caratteristica etc... Si parte dall'ultima caratteristica che si usa in modo da avere un algoritmo stabile:

$$\begin{matrix} cow \\ dog \\ rug \\ ear \\ tar \\ now \\ dig \\ big \end{matrix} \longrightarrow \begin{matrix} dog \\ rug \\ dig \\ big \\ ear \\ tar \\ cow \\ now \end{matrix} \longrightarrow \begin{matrix} ear \\ tar \\ dig \\ big \\ dog \\ cow \\ now \\ rug \end{matrix} \longrightarrow \begin{matrix} big \\ cow \\ dog \\ ear \\ now \\ rug \\ tar \end{matrix}$$

Possiamo riassumere l'algoritmo così:

```
RadixSort(A[],d)
  for i=1 to d
    algoritmo di ordinamento stabile
```

dove si suppone che ogni elemento nell'array A di n elementi abbian d cifre, dove la cifra 1 è quella di ordine più basso e la cifra d è quella di ordine più alto.
Ipotizzando l'uso del MergeSort come algoritmo di ordinamento stabile si ha il seguente tempo, eseguendo il MergeSort d volte:

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

3.3.3 BucketSort

Questo è un algoritmo di ordinamento non basato sui confronti ma presuppone che i valori nel range siano equiprobabili. Il caso peggiore si muoverà come n^2 mentre quello migliore come n. Se i valori sono equamente distribuiti il tempo atteso sarà n.

Capitolo 4

Strutture dati

In linea generale abbiamo 2 tipologie di strutture dati

- **statiche:** array
- **dinamiche:** liste, pile, alberi, etc...

Con quelle statiche si alloca una certa quantità di memoria, possibilmente in aree contigue, si ha così più velocità e più sicurezza ma si ha molta memoria inutilizzata che non si può usare perché già allocata; non è efficiente in termini di spazio.
Con quelle dinamiche si chiedono e si eliminano a richiesta caselle di memoria e si occupa memoria solo se necessario, ma non si avranno celle contigue. L'unico problema sarà la fine della memoria; non è efficiente in velocità ma lo è in spazio.

Per le strutture Dati si hanno le seguenti operazioni, (*D = Struttura dati, x = elemento e k =chiave*):

- Insert(D,x)
- Delete(D,x)
- Search(D,k)
- Update(D,x) (può essere fatto componendo le 3 operazioni sopra)
- Min(D)
- Max(D)
- Pred(D,k)
- Succ(D,k)

4.0.1 Lista Dinamica

Ne esistono vari tipi:

Lista Dinamica Semplice o Lista Semplicemente Concatenata

Ho un puntatore verso la prima casella, chiamato solitamente *Head* di tipo *pointer*, che contiene l'indirizzo di memoria. Se ho più liste, per esempio n posso fare un array di *pointer*, tipo *head[n]*, che punta alla testa, e *tail[n]* che punta alla coda. Se ho un puntatore vuoto esso contiene *null* o *nil*. Ipotizzo che le caselle contengono solo il campo a noi utile, la *chiave* o *key*. Aggiungo però il campo *next*, ovvero un puntatore al prossimo indirizzo di memoria, se è l'ultimo della lista esso sarà *null* o *nil*. SI ha accesso sequenziale, per arrivare ad una certa casella devo passare per tutte le altre. Per cancellare una casella devo ricollegare la casella prima con quella dopo. Se non si fa si hanno errori logici e non errori segnalati dal compilatore, si hanno così enormi rischi. Questa lista ci permette di andare solo avanti. Posso far puntare il *next* dell'ultima alla prima, ottenendo una *Lista Semlice Circolare*

Lista Dinamica Doppia

Come quella semplice solo con anche il campo *prev*. Si può così andare anche indietro nella lista, la prima avrà *prev=null/nill*. si può anche fare un ciclo, col *prev* del primo che punta all'ultimo e il *next* dell'ultimo che punta al primo, è detta *Lista Doppia Circolare*.

Lista con Sentinella

In una lista si può avere una casella che c'è sempre e non viene mai cancellata e può essere usata per controllo. Questa casella è detta *sentinella.*

Esempi senza sentinelle e con liste doppie non circolari

Esempio 16. *si ha:*
next[l] $\longrightarrow$ [|10|] $\longleftrightarrow$ [|7|] $\longleftrightarrow$ [|3|] $\longleftrightarrow$ [|21|]

```
pointer P
P=head[l]
print(key(P))
P=next(P)
print(key(P))
P=next(P)
print(key(P))
P=next(P)
print(key(P))
```

Si avrà l'output seguente:

```
10
7
3
21
```

Esempio 17. *Voglio cercare k in una lista L:*

```
pointer List_Search(L,k)
  p=head[l]
  while key(p) != k and p != null
    p=next(p)
  return(p)
```

Con la lista vuota funziona, ritorna comunque null, come quando non trova k

Si hanno anche le seguenti operazioni che vengono svolte di default nello pseudo codice ma non in molti linguaggi, tipo in C:

- allocazione: *x=alloc(sizeof(cell))*
- deallocazione *free(x)*

List Insert

Aggiungo dati ad una lista, x già ben formata e non nulla:

```
List_Insert(L,x)
  prev(x)=null
  next(x)=head[L]
  if  head[L] != null
    prev(head[L])=x
  head[L]=x
  x=null
```

List Delete

Analizziamo ora la *List delete*:

```
void List_delete(L,x)
  if prev(x)!=null
    next(prev(x))=next(x) /* ricollego le caselle prima e dopo x */
  else
    head(L)=next(x) /* se voglio togliere la prima */
  if(next(x)!=null
    prev(next(x))=prev(x)
  free(x) /* nel codice reale va sempre fatto */
```

List min

Analizziamo ora la *List min*:

```
pointer List_Min(L)
  p=head[L]
  min=head[L]
  if min==null /* controllo lista vuota */
    return null
  while next(p)!=null /* così non ho problemi sull'ultimo */
    p=next(p)
    if key(p)<key(min)
      min=p
  return(min)
```

List Successor

Analizziamo ora la *List Successor*, ovvero cerco il minimo elemento più grande di k:

```
pointer List_Successor(L,k)
  p=head[L]
  Pmin=null
  Vmin=MaxInt
  if p!=null and key(p)>k
    Pmin=p
    Vmin=key(p)
  else
    return null
  while next(p)!=null
     p=next(p)
    if key(p)<Vmin and key(p)>k
      Pmin=p
      Vmin=key(p)
  return(Pmin)
```

Esercizio 10. *Creo una lista che contiene gli elementi di altre due liste:*

```
pointer List_Union(L1,L2)
  if head[L1]==null /* L1 o entrambe vuote */
    head[L3]=head[L2]
    head[L2]=null /* può non controllare se L2 è vuota */
    return (head[L3])
  else if head[l2]==null /* solo L2 vuote */
    head[L3]=head[L1]
    head[L1]=null
    return head[L3]
  p=head[L1]
  while next(p)!=null
    p=next(p)
  next(p)=head[L2]
  head[L3]=head[L1]
  head[L1]=null
  head[L2]=null
  return head[L3]
```

Il tempo dipende dal numero di elementi anche della seconda:

$$T(n,m) = \Theta(1) + \overbrace{2 \cdot c \cdot n}^{while} + \Theta(1) = \Theta(n)$$

All'esame le operazioni vanno scritte col loro codice, non solo nominate

Esercizio 11. *Data una lista doppia circolare creare una lista con gli elementi al contrario:*

```
pointer List_Invert(L)
  p=head[L]
  if p==null
    return null
  while next(p)!=null
    papp=next(p) /* papp = p di appoggio */
    list_Insert(L2,p) /* L2 è la lista invertita */
    p=papp
  list_insert(L2,p) /* se no il puntatore di coda va a null */
  head[L2]=p
  return head[L2]
```

Esercizio 12. *conto in una lista doppia la presenza di k:*

```
int chiave(int k, pointer p)
 if p==null
   return 0
 else
   tot = chiave(k, next(p)
   if key(p)==k
     tot++
   return tot
```

4.1 Pile

La **pila**, detta anche **stack**, è un'altra struttura dati. Si tratta di una struttura dati di tipo dinamico che sfrutta il così detto **LIFO** (*Last In First Out*). In una pila posso prelevare dati solo dall'ultimo elemento inserito.
Si hanno le seguenti operazioni principali:

- *push(P,k)* che inserisce un elemento
- *pop(P)* che preleva dalla pila l'ultimo elemento
- *stackempty(P)* che restituisce *true* sse la pila è vuota
- *top(P)* come *pop* da il valore dell'ultimo elemento ma non lo rimuove dalla pila

Se eseguo una *pop* su una pila vuota avrò errore di underflow, se aggiungo un elemento ad una pila pieno avrò un overflow (che non posso controllare in quanto è causato dalla fine della memoria).
Una pila può essere implementata per mezzo di array o liste dinamiche.

4.1.1 Pile tramite array

Questa soluzione presenta alcuni problemi, devo shiftare tutti gli elementi ogni volta che aggiungo o tolgo un elemento (operazione che ha tempo lineare) e non è semplice capire se si sta lavora do su una lista vuota. Per ovviare a questo problema si usa una variabile di appoggio *t[S]* che ci indica la posizione dell'ultimo elemento inserito, ottenendo così un tempo costante sia per inserire che per rimuovere, infatti, per esempio, quando eseguo un *pop* non cancello effettivamente il valore ma decremento quella variabile, controllando che non vada a 0 evito anche l'underflow.

push

Ecco l'operazione di *push* usando un array:

```
int push(S, k)
  if t[S] == length(S)
    return error(-1)
  else
    t[S]++
    S[t[S]] = k
```

pop

Ecco l'operazione di *pop* usando un array:

```
<tipo> pop(S)
  if t[S] == 0 // underflow
    error(underflow)
  else
    R = S[t[S]]
    t[S]--
    return R
```

stackempty

Ecco l'operazione di *stackempty* usando un array:

```
boolean stackempty(P)
  if head[P] == null
    return true
  else
    return false
```

4.1.2 Pile tramite liste

Basta usare una lista semplice con una gestione "in testa"

push

Ecco l'operazione di *push* usando una lista:

```
push(L, x) // elemento x già esistente
  next(x) = head[L]
  head[L] = x
```

pop

Ecco l'operazione di *pop* usando una lista:

```
<tipo> pop(P)
  if head[P] == null
    error(underflow)
  else
    R = key(head[P])
    p_t = head[P]
    head[P] = next(head[P])
    // free(p_t)
    return R
```

stackempty

Ecco l'operazione di *stackempty* usando una lista:

```
boolean stackempty(P)
  if head[P] == null
    return true
  else
    return false
```

4.1.3 Esercizi

Esercizio 13. *Data una pila P e una chiave k eliminare tutte le occorrenze di k*

```
Elimina(P, K)
  while not(stackempty(P))
    R = pop(P)
    if R != K
      push(Papp, R)
  while not(stackempty(Papp))
  R = pop(Papp)
  push(P, R)
```

Calcolo i tempi:

$$T(n) = 3 \cdot c \cdot n + t_{if} \cdot c + 3 \cdot c \cdot t_{w_2}$$

Si ha il t_{if} se $k \notin P$ e vale 0 sse tutti i valori di P sono esattamente k. Le stesse cose valgono per t_{w_2}. Si ha:

- ***caso migliore:*** *ogni valore di P è uguale a k:*

$$T(n) = 3 \cdot c \cdot n = \Omega(n)$$

- ***caso peggiore:*** *nessun valore di P è uguale a k:*

$$T(n) = 3 \cdot c \cdot n + c \cdot n + 3 \cdot c \cdot n = 7 \cdot c \cdot n = O(n)$$

Quindi possiamo dire che si ha complessità $\Theta(n)$

Esercizio 14. *Data una pila P stabilire se l'elemento k è presente al suo interno*

```
boolean trova(P, K)
  trovato = false
  while not(stackempty(P)) && not(trovato)
    R = pop(P)
    if R == K
      trovato = true
  return trovato
```

Si ha una soluzione distruttiva usando la pop. Si ha:

- ***caso migliore:*** *k è in testa alla pila, si ha $T(n) = \Omega(1)$*
- ***caso peggiore:*** *nessun elemento di P è uguale k, si ha $T(n) = O(n)$*

Esercizio 15. *Invertire una stringa contenuta in un array usando una pila*

```
Inverti(P)
  while not(stackempty(P))
    R = pop(P)
    push(Papp1, R)
  while not(stackempty(Papp1))
    R = pop(Papp1)
    push(Papp2, R)
  while not(stackempty(Papp2))
    R = pop(Papp2)
    push(P, R)
```

SI ha:

$$T(n) = 3 \cdot c \cdot n + 3 \cdot c \cdot n + 3 \cdot c \cdot n = 9 \cdot c \cdot n = \Theta(n)$$

Esercizio 16. *Data una stringa verificare che le parentesi siano correttamente annidate*

```
boolean check(S[])
  i = 1
  ok = true
  while i <= length(S) && ok
    if S[i] == '(' || S[i] == '['
      push(P, S[i])
    if S[i] == ')'
      if stackempty(P) || pop(P) != '('
        ok = false
    if S[i] == ']'
        if stackempty(P) || pop(P) != '['
          ok = false
    i++
  if not(stackempty(P))
    ok = false
  return ok
```

- ***caso migliore:*** *il primo carattere è una parentesi chiusa di qualunque tipo*
- ***caso peggiore:*** *si hanno più parentesi aperte che chiuse*

Esercizio 17. *Date due pile a valori crescenti e una terza pila vuota inserire in modo ordinato le prime due nella terza*

```
Unisci(P1, P2)
  while not(stackempty(P1)) && not(stackempty(P2))
    if top(P1) <= top(P2)
      R = top(P1)
    else
      R = top(P2)
    push(Papp, R)
  while not(stackempty(P1))
    R = pop(P1)
    push(Papp, R)
  while not(stackempty(P2))
    R = pop(P2)
    push(Papp, R)
  while not(stackempty(Papp)) /* sono invertite quindi bisogna sistemare */
```

```
    R = pop(Papp)
    push(P, R)
```

Si ha:

$$T(m,n) = 4 \cdot c \cdot t_{w_1} + 3 \cdot c \cdot t_{w_2} + 3 \cdot c \cdot t_{w_3} + 3 \cdot c \cdot t_{w_4}$$

Si ha però che $t_{w_1} + t_{w_2} + t_{w_3} = m + n$ *e che* $t_{w_4} = m + n$*. La differenza tra il caso peggiore e quello migliore è minima e trascurabile:*

$$T(m,n) \sim 3 \cdot c \cdot (m+n) + 3 \cdot c \cdot (m+n) = \Theta(m+n)$$

Esercizio 18. *Si inseriscano da tastiera dati fino al carattere nullo, dire se la sequenza è palindroma utilizzando una pila (non è consentito contare i caratteri inseriti)*

```
boolean IsPalindroma
  do
    R = read(C)
    if R != ' '
      push(P, R)
      push(Papp, R)
  while R != ' ';
  while not(stackempty(Papp))
    R = pop(Papp)
    push(Papp2, R)
  while not(stackempty(P)) && top(P) == top(Papp2)
    pop(P)
    pop(Papp2)
  if stackempty(P) && stackempty(Papp)
    return true
  else
    return false
```

Si ha:

$$T(n)5 \cdot c \cdot n + 3 \cdot c \cdot n + 3 \cdot c \cdot t_{w_1} + 2 \cdot c$$

- ***caso migliore:*** *il primo e l'ultimo carattere sono diversi, si ha* $t_{w_1} = 0$ *quindi:*

$$T(n) = 8 \cdot c \cdot n + 2 \cdot c = \Omega(n)$$

- ***caso peggiore:*** *la stringa è palindroma, si ha:*

$$T(n) = 8 \cdot c \cdot n + 3 \cdot c \cdot n + 2 \cdot c = O(n)$$

Quindi si ha

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

4.2 Code

Le pile non sono la soluzione perfetta per ogni problema. Si supponga di dover gestire la coda si stampa attraverso una pila, potrebbe verificarsi che alcune stampe non vengano mai effettuate. Si cambia approccio, passando dalla **LIFO** alla **FIFO** *First On First Out*. Si ha una nuova struttura dati detta **Coda** o **queue**. Si hanno le seguenti operazioni:

- *Enqueue(Q,x)* che aggiunge un elemento in fondo alla coda
- *Dequeue(Q)* che prende il primo elemento della coda, lo toglie e lo restituisce
- *EmptyQueue(Q)* che ritorna *true* se la coda è vuota, *false* se è piena

4.2.1 Code con le liste

Implementiamo le operazioni base con le liste, è sufficiente una lista semplice e si ha:

Dequeue

Ecco l'operazione di *dequeue* usando una lista:

```
Dequeue(L)
  if head[L] == null
    return error(underflow) /* o null */
  else
    X = head[L]
    head[L] = next(head[L])
    return X
```

Enqueue

Ecco l'operazione di *enqueue* usando una lista, con un puntatore *tail* alla coda della lista:

```
Enqueue(L, X)
  if (tail[L]) != null)
    next(tail[L]) = X
  else
    head[L] = X
  tail[L] = X
```

EmptyQueue

Ecco l'operazione di *emptyqueue* usando una lista, c0on un puntatore *tail* alla coda della lista:

```
EmptyQueue(L)
  if head[L] == null
    return true
  else
    return false
```

4.2.2 Code con array

Si implementano ora le code con gli array, si usano array a 2 indici, uno per il primo elemento e uno per la prima posizione libera. Se i due indici si sovrappongono si ha una coda vuota. Si tiene una posizione libera per capire quando la coda è piena. Tutte le operazioni si fanno con tempo costante.

Dequeue

Ecco l'operazione di *dequeue* usando un array:

```
Dequeue(A[])
  if H_A == T_A /* array vuoto */
    return error(underflow)
  else
    R = A[H_A]
    H_A++ % N /* se il totale supera la lunghezza, riporto a 1 */
    return RX
```

Enqueue

Ecco l'operazione di *enqueue* usando un array:

```
Enqueue(A[], K)
  if T_A == H_A - 1 % N /* 0 posizioni libere */
    return error(overflow)
  else
    A[T_A] = K
    T_A++ modulo N
```

EmptyQueue

Ecco l'operazione di *emptyqueue* usando un array:

```
EmptyQueue(L)
  if H_A == T_A
    return true
  else
    return false
```

4.2.3 Esercizi

Esercizio 19. *Realizzare una coda da due pile*

```
EmptyQueue(Q)
  if stackempty(Q)
    return true
  else
    return false

Enqueue(Q)
  push(Q, x)

Dequeue(Q)
  if stackempty(Q)
    return error(underflow)
  else
    while not(stackempty(Q))
      R = pop(Q)
      push(Papp, R)
    x = pop(Papp)
    while not(stackempty(Papp))
      R = pop(Papp)
      push(Q, R)
```

ma ora si ha la dequeue non costante ma lineare, può essere ottimizzata in modo che faccia solamente il pop mentre la enqueue tenga ordinata la pila. Bisogna scegliere di volta in volta cosa è meglio

Esercizio 20. *Scrivere un algoritmo che estragga il k-esimo elemento della coda. (Si scelga un valore che non compare nella coda come appoggio, nel nostro caso -1)*

```
Extract(Q, k)
  enqueue(Q, -1)
  i = 1
  do
    R = dequeue(Q)
    if i != k && R != -1
      Enqueue(Q, R) /* accodo la coda a se stessa */
    i++
  while R != -1;
  if i < k
    return error(underflow)
```

Esercizio 21. *Cancello da una coda tutte le k-esime occorrenze, usando solo code come appoggio*

```
Del_m(Q, k)
  while not(emptyqueue(Q))
    R = dequeue(Q)
    if R != k
      enqueue(Qapp, R)
  while not(emptyqueue(Q)) /* copio nella coda originale */
    R = dequeue(Qapp)
    enqueue(Q, R)
```

Si ha:

$$T(n) = 4 \cdot c \cdot t_{w_1} + 3 \cdot c \cdot t_{w_2}$$

- ***caso migliore:*** *la coda contiene solo elementi uguali a k, quindi $t_{w_1} = n$ e $t_{w_2} = 0$ e:*

$$T(n) = 4 \cdot c \cdot n = \Omega(n)$$

- ***caso peggiore:*** *k non compare mai nella lista, quindi $t_{w_1} = t_{w_2} = n$ e:*

$$T(n) = 7 \cdot c \cdot n = O(n)$$

Quindi si ha in generale:

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

Esercizio 22. *Scrivere un algoritmo che data una pila a elementi unici e una coda al elementi ripetuti elimini dalla coda gli elementi della pila*

```
Delete(Q, S)
  flag = 0
  while not(stackempty(S))
    R = pop(S)
    while (not(emptyqueue(Q)) && not(emptyqueue(Q) && emptyqueue(Qapp)
&& flag == 0) /* a coda vuota nulla da cancellare, stop */
      H = dequeue(Q)
    if H != R
      enqueue(Qapp, H)
    while not(emptyqueue(Qapp)) && flag == 1
      H = dequeue(Qapp)
      if H != R
        enqueue(Qapp, H)
    flag = not(flag) /* cambio flag */
    push(Sapp, R)
  while not(stackempty(Sapp)) /* ribalto la pila nell'originale */
    R = pop(Sapp)
    push(S, R)
  while not(emptyqueue(Qapp)) /* copo Qapp in Q per sicurezza */
    H = dequeue(Qapp)
    enqueue(Q, H)
```

Con n lunghezza pila e m lunghezza della coda si ha:

$$T(n,m) = c+4\cdot c\cdot t_{w_1}+3\cdot c\cdot t_{w_2}+3\cdot c\cdot t_{w_3}+3\cdot c\cdot t_{w_4}+3\cdot c\cdot t_{w_5}+c\cdot t_{if_1}+c\cdot t_{if_2}$$

- ***caso migliore:*** *tutti gli elementi della coda sono uguali tra loro e sono uguali al primo elemento della pila. Si ha:*

$$T(n,m) = c+4\cdot c+3\cdot c\cdot m+3\cdot c = \Omega(m)$$

- ***caso peggiore:*** *la pila e la coda non hanno elementi in comune e n è dispari, si ha:*

$$T(n,m) = c+4\cdot c\cdot n+3\cdot c\cdot n\cdot m+3\cdot c\cdot m+n\cdot m = O(n\cdot m)$$

4.2.4 insiemi

Un insieme è una collezione di insiemi. Si hanno le seguenti operazioni base:

- **appartenenza:** $x \in A$ che si risolve con una *list search* con $\Omega(1)$ e $O(n)$
- **unione:** $A \cup B = \{x \in A \; or \; x \in B\}$
- **intersezione:** $A \cap B = \{x \in A \; and \; x \in B\}$
- **differenza:** $A \cup B = \{x \in A \; and \; x \notin B\}$

Unione

Si ha:

```
Unione (La, Lb)
  if head[Lb] == null
    return head[La]
  if head[La] == null
     return head[Lb]
  Pa = head[La]
  while Pa != null /* confronto e unisco */
    Pb = head[Lb]
    while Pb != null && key(Pa) != key(Pb)
      Pb = next(Pb)
    if Pb == null /* se manca l'elemento lo aggiungo */
      insert(Lu, Pa)
  next(tail[Lb]) = head[Lu]
  head[Lu] = head[Lb]
  head[La] = null
  head[Lb] = null
```

Con:

- **caso migliore:** A e B sono uguali
- **caso peggiore:** A e B sono completamente diversi

se le liste sono ordinate si può migliorare l'algoritmo:

```
Unione(L1, L2)
  if head[L1] == null
    return L2
  if head[L2] == null
     return L1
  if key(head[L1]) < key(head(L2))
     sistema(head[L1], unione(next(head[L1]), head[L2]))
  else if key(head[L1]) > key(head[L2])
     sistema(head[L2], unione(head[L1], next(head[L2])))
  else
     sistema(head[L1], unione(next(head[L1]), next(head[L2])))
```

Con:

- **caso migliore:** la lista più corta ha anche tutti gli elementi più piccoli. Si ha:
$$T(n, m) = n$$
- **caso peggiore:** gli ultimi due elementi delle due liste sono i valori più grandi. Ci saranno $n + m$ chiamate ricorsive

4.3 Alberi e alberi binari

un albero è un grafo non orientato, aciclico e connesso. A noi interessano certi tipi di alberi, quelli radicati (ovvero con radice), comodi perché ogni nodo ha sopra un solo nodo, e la radice nessuno. Un nodo è così fatto:

parent
key
f | f | f

con f detti figli, (albero ternario implica tre figli etc). la radice ha *parent=null*. *root[p]* è un puntatore alla radice e l'antenato diretto di un nodo è il suo *parent* mentre i figli sono tutti i nodi raggiungibili scendendo da quel nodo. I nodi senza figli sono detti *foglie*, che hanno tutti i puntatori figli pari a *null*. L'albero ha grandezza pari al massimo numero di figli mentre la profondità di un nodo è la distanza di un nodo dalla radice (si contano i cammini), e l'altezza è il massimo delle profondità. Per collegare i vari figli di un albero uso una lista, ottimizzo lo spazio ma non i tempi (usare solo i punattori di un albero sprecherei troppa memoria). Un nodo dello stesso livello si chiama

brother.
Un albero binario (massimo 2 figli) è un albero vuoto o un albero con al di sotto a destra un albero binario e a sinistra un albero binario (definizione ricorsiva). esso avrà: **Riguardare**

parent	
key	
f	f

Un albero è pienamente binario si ha quando si ha ogni nodo con 0 o 2 figli, mai 1. Un albero binario completo è un albero pienamente binario si ha se si hanno nodi ad ogni livello e foglie solo all'ultimo livello e l'albero ha 2^h come altezza, h nodi a quell'altezza e li si avranno $\sum_{h=0}^{h} 2^h = 2^{h+1} - 1$ nodi. Per stampare un albero posso stampare per livelli o posso usare dei sottoalberi e usare un diverso ordine di lettura dell'albero (vedere su slide di fondamenti).

Esempio 18 (preorder)**.** *Stampo in preorder*

```
Preorder(P)
  if P!=null
    print(key(P))
    Preorder(left(P))
    Preorder(right(P))
```

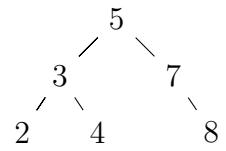
Esempio 19 (Inorder)**.** *Stampo in inorder*

```
Inorder(P)
  if P!=null
    Inorder(left(P))
    print(key(P))
    Inorder(right(P))
```

Esempio 20 (postorder)**.** *Stampo in postorder*

```
Postorder(P)
  if P!=null
    Postorder(left(P))
    Postorder(right(P))
    print(key(P))
```

Graficamente, per l'albero:



si hanno le seguenti visualizzazioni:

- **PreOrder**: $5, 3, 2, 4, 7, 8$
- **PostOrder**: $2, 4, 3, 7, 8, 5$
- **InOrder**: $2, 3, 4, 5, 7, 8$

indicativamente, per quanto riguarda i tempi, si hanno due casi:

- **albero sbilanciato**: $T(n) = T(n-1) + c \sim c \cdot n$
- **albero bilanciato**: $2 \cdot T\left(\frac{n}{2}\right) \sim \Theta(n)$ (si ricava col primo caso del Teorema dell'Esperto)

Esempio 21 (per livello)**.** *Stampo per livello, iteramente usando una pila*

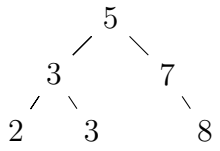
```
Stampa(P)
  if P==null
    return null
  else
    do
      print(key(P))
      push(P,key(P))
      if right(P)!=null
        push(P,right(P))
      else left(P)!=null
        push(P,left(P))
      T=pop(P)
    while T!=null
```

Si ha che un albero ben bilanciato ha altezza $\log n$, mentre un albero sbilanciato ha altezza massima $n-1$ (praticamente si ha una lista)
Con gli alberi posso ottimizzare l'operazione di ricerca. Si mediano i tempi di tutte le operazioni e si ha un tempo logaritmico. In un albero binario si hanno i valori più piccoli a sinistra della radice e i più grandi a destra, il *min* sarà il valore in basso più a sinistra e il *max* quello a destra.
$\forall$ *nodo* n tutti i nodi del sottoalbero con radice $left(n)$ sono $\leq N$ e quelli a destra $\geq N$, con N valore della radice.
$\forall$ *nodo* x si ha che:

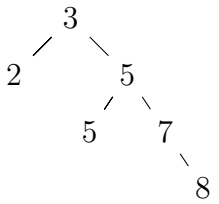
- per tutti i nodi y del sottoalbero con radice $left(y)$ si ha che: $left(y) \leq key(c)$
- per tutti i nodi y del sottoalbero con radice $right(y)$ si ha che: $right(y) \geq key(c)$

Per esempio per i valori $2, 3, 5, 5, 7, 8$ si hanno vari tipi di albero binario di ricerca:

- il primo:



- il secondo:



Ecco alcune operazioni su un albero binario (se l'albero rimane quasi bilanciato si avrà un tempo logaritmico)

Esempio 22. `Ricerca(X, K)`

```
if X == null || key(X) == K)
  return X
else
  if key(X) > K
    R = Ricerca(left(X), K)
else
    R = Ricerca(right(X), K)
return R
```

Si ha:

- ***caso migliore:*** *la radice è il valore cercato e si ha* $T(n) = \Omega(1)$
- ***caso peggiore:*** *il valore non è nell'albero e si ha* $T(n) = \Theta(h)$*, con* h *altezza dell'albero*
- ***albero bilanciato:*** $T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \sim \log n$

Esempio 23. *Ricerca del minimo, che sarà l'elemento che si trova scendendo tutto a sinistra.*

```
Min(X)
  if X == null
    return X
  else
    while left(X) != null
      X = left(X)
    return X
```

Quindi la ricerca del minimo (come del resto del massimo dove si scende a destra e non a sinistra) richiede un tempo pari alla profondità del ramo. Si ha:

- ***caso migliore:*** *la radice non ha figlio sinistro (se si cerca il maggiore se non ha figlio destro) e si ha tempo costante*
- ***caso peggiore:*** *si sta percorrendo il ramo che determina l'altezza dell'albero, si avrà quindi $T(n) = O(h)$, con h altezza dell'albero*

Esempio 24. *Si ricerca il successore:*

```
Succ(X)
  if right(X) != null /* se esiste il sottoalbero di destra è banale */
    R = Min(right(X))
    return R
  else /* ricerco il primo nodo */
    while parent(X) != null && X != left(parent(X))
      X = parent(X)
    return parent(X)
```

Si sta percorrendo un ramo, quindi si avrà $T(n) = O(h)$, con h altezza dell'albero.

Esempio 25. *vediamo l'inserimento:*

```
Insert(T, X)
  if root[T] == null
    root[T] = X
  else
    P = root[T]
    Pa = null
    while P != null
      Pa = P
      if key(P) > key(X)
        P = left(P)
      else
        P = right(P)
    if key(Pa) > key(X)
      left(Pa) = X
    else
      right(Pa) = X
```

e come prima si avrà $T(n) = O(h)$, con h altezza dell'albero.

Esempio 26. *Cancellazione di un elemento. Qui si hanno più casistiche:*

- *per eliminare una foglia basta rimuoverla*

- *per eliminare un elemento con un solo figlio sostituisco con il sottoalbero che ha come radice l'unico figlio, questa operazione è detta contrazione*
- *per eliminare un elemento con entrambi i figli trovo il minimo del sottoalbero destro (o il massimo del sinistro) per ottenere il successore (o rispettivamente il predecessore) e scambio i 2 valori. Infine cancello il nodo col successore (rispettivamente il predecessore) ottenendo uno dei 2 casi sopra*

```
Delete(T, X)
  if left(X) == null && right(X) == null /* caso 1 */
    if parent(X) == null
      root[T] = null
    else
      if X == left(parent(X)) /* X è il figlio sinistro */
        left(parent(X)) = null
      else /* X è il figlio destro */
        right(parent(X)) = null
    else if left(X) == null XOR right(X) == null /* caso 2 */
      contrazione(T, X)
    else /* caso 3 */
      S = succ(X)
      key(X) = key(S)
      Delete(T, S)

contrazione(T, X)
  if parent(X) == null
    if left(X) != null
      root[T] = left(x)
      parent(left(X)) = null
    else
      root[T] = right(X)
      parent(right(X)) = null
  else if X == left(parent(X))
    if left(X) != null
      left(parent(X)) = left(X)
      parent(left(X)) = parent(X)
    else
      left(parent(X)) = right(X)
      parent(right(X)) = parent(X)
  else
```

```
if left(X) != null
  right(parent(X)) = left(X)
  parent(left(X)) = parent(X)
else
  right(parent(X)) = right(X)
  parent(right(X)) = parent(X)
```

il terzo caso ha circa ($T(n) = O(h)$, con h altezza dell'albero. Si nota che Contrazione ha tempo costante

lo sbilanciamento viene riconosciuto mediante i valori numerici:

- 0 se è bilanciato
- negativi se il sottoalbero di sinistra è più altro di quello di destra
- positivi se il sottoalbero di destra è più alto di quello di sinistra

Esercizio 23. *Con un algoritmo dividi et impera conto quanti valori sono $N1 \leq X \leq N2$*

```
int conta(X, N1, N2)
  if X == null
    return 0
  else
    sx = conta(left(X), N1, N2)
    dx = conta(right(X), N1, N2)
    if key(X) >= N1 && key(X) <= N2
      c = 1
    else
      c = 0
    return sx + dx + c
```

Si ha la seguente equazione di ricorrenza:

$$\begin{cases} 4 \cdot c + 2 \cdot T\left(\frac{n}{2} - 1\right) \\ 2 \cdot c, \ \ conn = 0 \end{cases}$$

Quindi si ha $T(n) = O(n^{n \cdot \log_b a - \varepsilon}) \sim \Theta(n)$ per il teorema dell'esperto

Esercizio 24. *Dato un albero binario con n elementi e un vettore con m elementi li stampo in ordine crescente*

```
UnioneCrescente(T, A[])
  i = 1
  X = SBT_min(T)
  while i <= length(A) && X != null
    if A[i] < key(X)
      print(A[i])
    i++
    else if key(X) < A[i]
      X = succ(X)
    else /* non stampo ripetizioni */
      print(A[i])
      i++
      X = succ(X)
  while i <= length(A)
    print(A[i])
    i++
  while X != null
    print(key(X))
    X = succ(X)
```

Si ha $T(n, m) \sim n \log n + m$, che è solo approssimato in quanto la ricerca del successore di una foglia è 1 e solo la radice ha $n \cdot \log n$ come tempo per la ricerca del successore

Esercizio 25. *Conto i nodi di un albero binario*

```
int nodi_BT(X)
  if X == null
    return 0
  else
    S = nodi_BT(left(X))
    D = nodi_BT(right(X))
  return S + D + 1

int foglie_BT(X)
  if X == null
    return 0
  else if right(X) == null && left(X) == null
    return 1
  else
    S = foglie_BT(left(X))
    D = foglie_BT(right(X))
```

```
    return S + D

int figlidestri_BT(X)
  if X == null
    return 0
  else
    S = figlidestri_BT(left(X))
    D = figlidestri_BT(right(X))
    if right(X) != null
      C = 1
    else
      C = 0
    return S + D + C

int altezza_BT(X)
  if X == null // l'altezza di un albero vuoto è -1
    return -1
  else
    S = altezza_BT(left(X))
    D = altezza_BT(right(X))
    M = max(S, D)
    return M + 1
```

Esercizio 26. *Verificare se un albero binario è anche di ricerca:*

```
boolean is_Ricerca(X)
  if X == null
    return true
  else
    S = is_Ricerca(left(X))
    if S == false
      return false
      S_M = SBT_max(left(X)) /* controllo se l'albero è nullo */
      if S_M != null && key(S_M) > key(X)
        return false
      D = is_SBT(right(X))
        if D == false
          return false
      D_m = SBT_min(right(X))
      if D_m != null && key(D_m) < key(X)
        return false
  return true
```

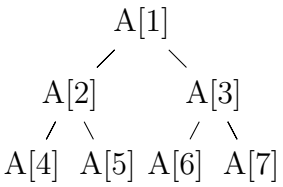
Si ha che il caso peggiore è quando è di ricerca e si ha:

$$\begin{cases} 2 \cdot c \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 \cdot \log n \end{cases}$$

per il primo caso del teorema dell'esperto si ha $T(n) = \Theta(n)$

4.4 Heap

l'*Heap* è un array visto come un albero binario quasi completo, ovvero l'array *a[i]*, $1 \leq i \leq 6$ sarà:



Nell'heap si ha che $\forall$ nodo $i \neq root$ si ha $A[parent(i)] \geq a[i]$ se *A[parent(i)]* è il max dell'heap e $A[parent(i)] \leq a[i]$ se *A[parent(i)]* è il min dell'heap. Inoltre si ha che:

- $left(i) = 2 \cdot i$
- $right(i) = 2 \cdot i + 1$
- $parent(i) = \frac{i}{2}$
- $heapsize(A) \leq length(a)$, si ha che l'heapsize varia in fase di esecuzione
- per n elementi nello heap si hanno $\log_2 nodi$, infatti alla radice ho 2^0 elementi, scendo e ne ho 2^1 e così via
- lo heap ha $\frac{n}{2}$ foglie

4.4.1 HeapSort

Si ha questo algoritmo di sort che sfrutta le proprietà dello heap ed è formato da 2 fasi:

- **BuildHeap:** si ottiene un heap da un array non ordinato, si sposta il massimo in fondo, si decremente l'heapsize e infine si richiama il BuildHeap
- **heapify:** si crea un heap senza considerare tutto l'array, considerando che il sottoalbero di sinistra e di destra sono già heap e traforma l'albero in un heap a partire da un certo valore

Si ha:

```
Heapify(A, K)
  largest = K
  if left(K) <= hs(A) && A[left(K)] > A[largest]
    largest = left(K)
  if right(K) <= hs(A) && A[right(K)] > A[largest]
    largest = right(K)
  if largest != K
    scambia(A, K, largest)
    heapify(A, largest)
```

che ha la seguente equazione di ricorrenza:

$$T(n) = 9 \cdot c + T\left(\frac{2}{3} \cdot n\right)$$

applico il secondo caso del teorema dell'esperto, avendo:

$$n^{\log_b a} = n^{\log_{\frac{2}{3}} 1} = n^0 = \Theta(1) \; e \; f(n) = (1)$$

si ha:

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

che è un risultato compatibile a quello che si otterrebbe ragionando sul fatto che l'altezza h, che determina il tempo peggiore, è circa $\log n$.
Vediamo ora la BuildHeap:

```
BuildHeap(A)
  heapsize(A) = length(A)
  for i = floor(N/2) down to 1 /* no foglie */
    heapify(A, i)
```

Per il calcolo dei tempi si ha:

$$\sum_{i=0}^{\log n} \lceil \frac{n}{s^{i+1}} \rceil \cdot i \leq \sum_{i=0}^{\infty} n \cdot \frac{i}{2^{i+1}} = n \cdot \frac{1}{1-\frac{i}{2}} = 2 \cdot n = O(n)$$

dalle due funzioni si ottiene:

```
HeapSort(A[])
  BuildHeap(A)
  for i = 1 to N - 1
    scambia(A, 1, heapsize(A))
    heapsize--
    heapify(A, 1)
```

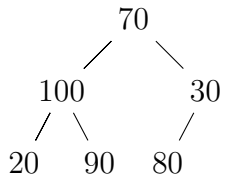
l'heapsize andrebbe decrementato fino a 0, quindi alla fine di tutto bisogna dare un ulteriore *heapsize–*. Si ha che $T(n) = O(n \cdot \log n)$ e che è in loco ma non stabile.

Rappresentazione dell'heapsort

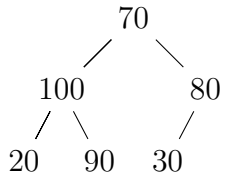
Si ha la seguente sequenza di numeri:

$$[70, 100, 30, 20, 90, 80]$$

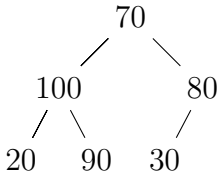
mediante l'heap viene così rappresentata:



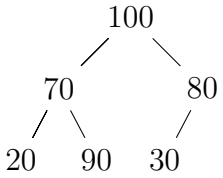
partiamo col buildheap:
Abbiamo quindi 6 cifre e quindi avremo $\frac{heapsize}{2} = 3$.
Pariamo quindi con *heapify(3)*, ovvero selezionando 30 come pivot. Tra i figli abbiamo che esiste un valore maggiore, 80, quindi effettuiamo lo scambia



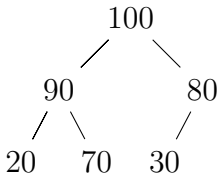
Non avendo 30 figli passiamo direttamente a *heapify(2)* selezionando 100 che come prima non ha valori superiori tra i figli quindi non si effettua nessun cambiamento.



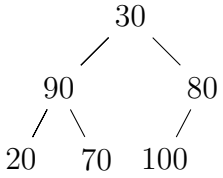
Arriviamo infine a *heapify(1)*, avendo come pivot 70. In questo caso abbiamo innanzitutto lo scambio col valore 100:



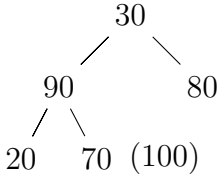
inoltre dato che 70, ora in posizione 2, ha un figlio con un valore maggiore, 90, provvedo ad effettuar lo scambio facendo *heapify(2)*:



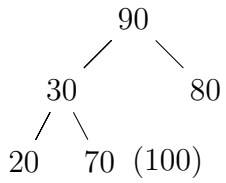
Siamo giunti alla fine del *buildheap*, avendo così un *heap* che presente il valore maggiore come radice e il valore minore come una delle foglie.
Heapsort:
Partiamo invertendo primo e ultimo valore:



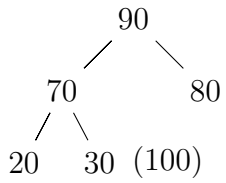
riduciamo l'*heapsize* di uno e quindi rimuoviamo "virtualmente" 100 dall'*heap*:



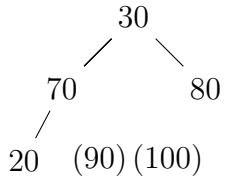
procediamo con *heapify(1)*, avendo 30 come pivot. Scambio prima con 90:



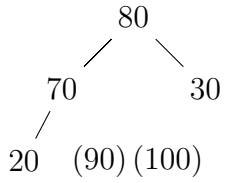
e poi 30 con 70:



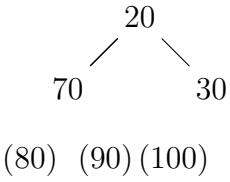
l'*heapify* è finito quindi posso nuovamente invertire il primo e l'ultimo valore. infine posso ridurre l'*heapsize* di 1 e rimuovere "virtualmente" 90:



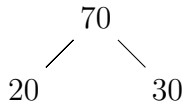
procedo nuovamente con l'*heapify(1)*. Prendo come pivot 30 e lo inverto con 80:



l'*heapify* è finito quindi nuovamente inverto il primo e l'ultimo valore, rimuovendo "virtualmente" 80:

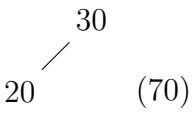


procedo nuovamente con l'*heapify(1)*, avendo 20 come pivot. Scambio 20 con 70:



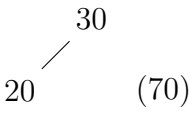
(80) (90) (100)

l'*heapify* è finito quindi nuovamente inverto il primo e l'ultimo valore, rimuovendo "virtualmente" 70:



(80) (90) (100)

effetto quindi nuovamente l'*heapify(1)*, con pivot 30. Non avviene alcuno scambio:



(80) (90) (100)

l'*heapify* è finito quindi nuovamente inverto il primo e l'ultimo valore, rimuovendo "virtualmente" 30:

20

(30) (70)

(80) (90) (100)

Abbiamo così ottenuto il vettore riordinato dal minore al maggiore:

$$[20, 30, 70, 80, 90, 100]$$

4.4.2 altro sullo heap

Funzione per trovare il massimo:

```
int ExtractMax(A)
  max=A[1]
  Scambia(A,1,heapsize(A))
  heapsize(A)--
  heapify(A,1)
  return max
```

si ha $T(n) = O(\log n)$
Per aumentare una chiave si ha:

```
HeapIncreaseKey(A,i,k)
  A[i]=k
  while i>1 and A[parent(i)<A[i]
    scambia(A[i],A[parent(i)])
    i=parent(i)
```

che ha $T(n) = O(\log n)$.
Per inserire un massimo si ha:

```
MaxHeapInsert(A,k)
  heapsize(A)=heapsize(A)+1
  A[heapsize(A)]=-inf
  HeapIncreaseMax(A,heapsize(A),k)
```

che ha sempre $T(n) = O(\log n)$