Binary Search

Se un vettore non è ordinato, il numero di confronti è lineare
worst case: *n* confronti
best case: 1 confronto

ma se il vettore è ordinato possiamo usare la ricerca dicotomica, che, nel **worst case** effettua $\log_2 n$ confronti.
Dunque è **importante** avere un **vettore ordinato!**

```
BinSearch-Ric(x, A, i, j)
      ▷ Pre: A[i..j] ordinato
      ▷ Post: true se x ∈ A[i..j]
if i > j then        ▷ A[i..j] = ∅
    return false
else
    m ← ⌊(i + j)/2⌋
    if x = A[m] then
        return true
    else
        if x < A[m] then
            return BinSearch-Ric(x, A, i, m − 1)
        else      ▷ A[m] < x
            return BinSearch-Ric(x, A, m + 1, j)
        end if
    end if
end if
```

*Algoritmo della ricerca dicotomica(simile al **Peak Finding DI**)*
*Nota: provare a farlo in **C** visto che può venir chiesto all'esame*

Ordinamento

Ordinamento come problema computazionale:
Input: una sequenza di *n* numeri $a_1, a_2, .., a_n$
Output: una permutazione $a_i1, a_i2, .., a_in$ della sequenza in input tale che $a_i1 \leq a_i2 \leq \ldots \leq a_in$

Forza bruta

```
SORTED(A)
for i ← 2 to length(A) do
    if A[i − 1] > A[i] then
        return false
    end if
end for
return true
```

```
TRIVIAL-SORT(A)
for all A' permutazione di A do
    if SORTED(A') then
        return A'
    end if
end for
```

Questo algoritmo(**Sorted**) controlla che il vettore ricevuto in input sia ordinato o meno.
Dunque controlla tutte le possibili permutazioni(**Trivial Sort**) di un dato vettore.
Tuttavia il numero di permutazioni di un vettore è $n!$, il quale è altissimo

Infatti, $n!$ cresce persino più velocemente di una esponenziale(ad esempio 2^n).
$n < 2^n < n!$

Ordinamento per inserimento (Insertion Sort)

Idea di base per ordinare un vettore A[1..n]:

- Se $A[1...i-1]$ è già ordinato, posso **inserire** $A[i]$ nella parte ordinata tramite scambi
    - Se $A[i] \geq A[i-1]$ allora $A[i..i]$ è ordinato e ci si ferma
      altrimenti si **scambia** $A[i]$ con $A[i-1]$
    - Se $A[i-1] \geq A[i-2]$ allora $A[i..i]$ è ordinato e ci si ferma
      altrimenti si **scambia** $A[i-1]$ con $A[i-2]$
    - ...

- Alla fine $A[i..i]$ è ordinato
- Partiamo dal presupposto che $A[1..1]$ è ordinato
- Inseriamo nella parte ordinata prima $A[2]$, poi $A[3]$ fino ad inserire $A[n]$

Esempio:

(5,4,7,3,6,6)
(**4**, 5,7,3,6,6)
(4,5, **3**, 7,6,6)
(4, **3**, 5,7,6,6)
(**3**, 4,5,7,6,6)
(3,4,5, **6**, 7,6)
(3,4,5,6, **6**, 7)

Il vettore A all'inizio contiene (5,4,7,3,6,6)

Pseudocodice:

INSERTION-SORT(A)
for $i \leftarrow 2$ **to** $length(A)$ **do**
  $\triangleright$ inserisce $A[i]$ in $A[1..i-1]$
  $j \leftarrow i$
  while $j > 1$ **and** $A[j-1] > A[j]$ **do**
    scambia $A[j-1]$ con $A[j]$
    $j \leftarrow j - 1$
  end while
end for
return A

La variabile i viene inizializzata a 2 visto che si parte dal presupposto che $A[1..1]$ è ordinato
e cicla per la lunghezza di A ($length(A)$).
Nel ciclo più interno(**while(..)** con indice $j = i$), ciclo per tutto l'array finchè persiste la condizione $A[j-1] > A[j]$, ovvero il mio vettore **non è ordinato** per la porzione $A[1..j]$, e finchè $j > 1$ ovvero finchè ho elementi nel sub-array da controllare.
Finchè ciclo inoltre scambio $A[j-1]$ con $A[j]$.

Dunque ci spostiamo **da destra verso sinistra**, visto che scambio $A[j]$ con $A[j-1]$

Correttezza dell'algoritmo

L'**insertion sort** usa due cicli ⇒ usiamo le invarianti di ciclo per verificarne la sua correttezza.

Invariante ciclo esterno: $A[1..\,i-1]$ è ordinato.

Prima e dopo del ciclo questo predicato sussiste:

- All'inizio, $i=2$ e dunque $A[1..1]$ è ordinato per definizione (**inizializzazione**)
- Durante il ciclo, devo dimostrare che $A[1..\,i-1]$ ⇒ $A[1..\,i'-1]$ con $i'=i+1$
  - Se inserisco $A[i]$ in $A[1..\,i-1]$(la porzione già ordinata) correttamente ogni volta, allora l'invariante viene mantenuto
  - il **mantenimento** dipende tuttavia dalla correttezza del ciclo interno

Invariante ciclo interno:

1. $A[1..\,j-1]$ e $A[j..i]$ sono ordinati
2. Ciascun elemento in $A[1..\,j-1]$ è $\leq$ di tutti gli elementi di $A[j+1..\,i]$

- **Inizializzazione:**
  1. Con $j=1$ l'invariante diventa $A[1..\,i-1]$ e $A[i..i]$ sono ordinati.
  2. Ciascun elemento in $A[1..\,i-1]$ è $\leq$ di tutti gli elementi di $A[i+1..\,i]=\emptyset$
- **Mantenimento:**
  - Il ciclo si esegue soltanto se $j>1 \wedge A[j-1]>A[j]$
  - Se viene eseguito, scambio $A[j-1]$ con $A[j]$ e $j'=j-1$
    - $A[1..\,j-1]$ è ordinato ⇒ $A[1..\,j'-1]=A[1..\,j-2]$ è ordinato
    - $A[j..i]$ è ordinato AND $A[1..\,j-1] \leq A[j+1..\,i] \wedge A[j-1]>A[j]$ ⇒ $A[j'..i]=A[j-1..\,i]$ è ordinato

Avendo dimostrato la **correttezza del ciclo interno**, abbiamo dimostrato anche la **correttezza dell'invariante del ciclo esterno** ⇒ **correttezza dell'insertion sort dimostrata**

Complessità temporale dell'Insertion Sort

1. **for** i← 2 **to** *length*(A)
2.   j← i
3.   **while** $j > 1$ and $A[j-1] > A[j]$
4.   **scambia** $A[j-1]$ con $A[j]$
5.   j ← j-1

La **riga 1** viene eseguita n volte con un costo c_1. Infatti assegna un valore alla variabile i fino alla condizione d'uscita del **for**, ovvero $length(A)+1$

La **riga 2** assegna un valore a j $n-1$ volte.Infatti assegna il valore fino a $j = length(A)$.

La **riga 3** viene eseguita $\sum_{i=2}^{n} t_i$ volte.
Infatti viene eseguita almeno 1 volta per ogni iterazione del **for**, fino ad un massimo di i volte per ogni iterazione del **for**.
Ciò perchè il ciclo **while** cicla al massimo i volte visto che decrementa $j = i$ finchè non diventa ≤ 1

La **riga 4** viene eseguita lo stesso numero di volte rispetto alla terza riga decrementato di 1. Infatti, rispetto alla terza riga, se la condizione del while è falsa, la quarta riga non viene eseguita, dunque $\sum_{i=2}^{n}(t_i - 1)$ volte

La **riga 5** è la stessa roba della quarta

	costo	num. volte
for $i \leftarrow 2$ **to** $length(A)$	c_1	n
$j \leftarrow i$	c_2	$n-1$
while $j > 1$ **and** $A[j-1] > A[j]$	c_3	$\sum_{i=2}^{n} t_i$
scambia $A[j-1]$ con $A[j]$	c_4	$\sum_{i=2}^{n}(t_i - 1)$
$j \leftarrow j-1$	c_5	$\sum_{i=2}^{n}(t_i - 1)$

Pic for reference

Caso Peggiore

Con $t_i = i$, ovvero il caso peggiore, la complessità temporale diventa:

$$T_{ins}(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=2}^{n} i + c_4 \sum_{i=2}^{n} (i-1) + c_5 \sum_{i=2}^{n} (i-1$$

$$= (c_1+c_2)n - c_2 + c_3 \sum_{i=2}^{n} i + (c_4 + c_5)\sum_{i=2}^{n} (i-1)$$

$$\sum_{i=2}^{n} i = 2 + 3 + \cdots + n = \frac{n+2}{2}(n-1) = \frac{n^2+n-2}{2}$$

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

$$T_{ins}(n) = \frac{c_3+c_4+c_5}{2}n^2 + \left(c_1 + c_2 + \frac{c_3-c_4-c_5}{2}\right)n - (c_2+c_3)$$

ovvero $= an^2 + bn + c$.

Ha dunque una **complessità temporale quadratica**

Caso Migliore

Con $t_i = 1$, ovvero il caso migliore, la complessità temporale diventa:

$$T_{ins}(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=2}^{n} 1 + c_4 \sum_{i=2}^{n} (1-1) + c_5 \sum_{i=2}^{n} (1-1)$$

$$= (c_1+c_2)n - c_2 + c_3 \sum_{i=2}^{n} 1$$

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

$$T_{ins}(n) = (c_1 + c_2 + c_3)n - (c_2 + c_3) = dn + e$$

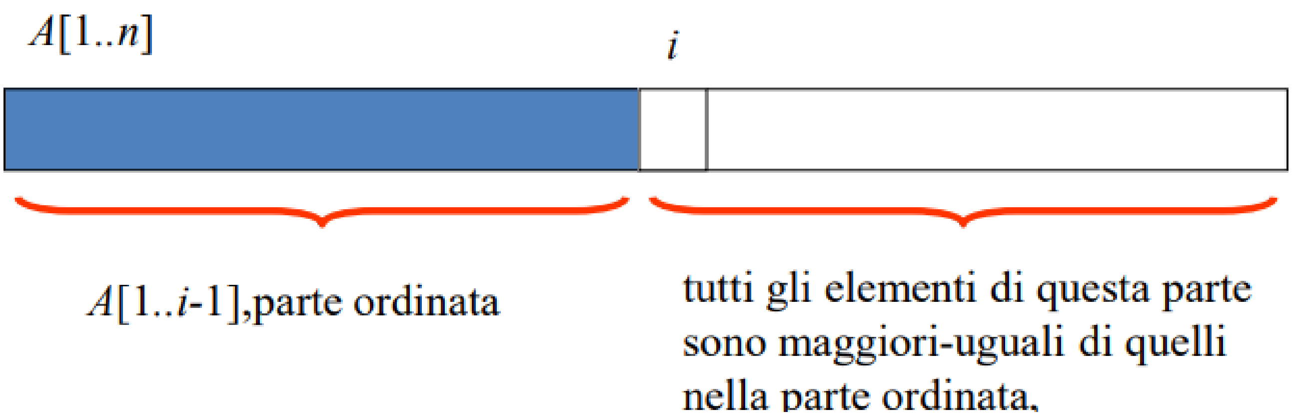
Ha dunque una **complessità temporale lineare**

Selection Sort

Idea di base:

- Assumiamo che:
    - la parte sinistra del vettore è ordinato
    - la parte destra del vettore contiene elementi maggiori o uguali

Graficamente:



- Cerchiamo l'elemento minimo in $A[i..n]$ e lo scambiamo con $A[i]$
- In questo modo la parte ordinata si allarga($A[1..i-1]$ *nella pic*) e quella disordinata diminuisce

Pseudocodice:

```
SELECT-SORT(A)
for i ← 1 to length(A) − 1  do        ▷ n = length(A)

    k ← i
    for j ← i + 1 to length(A) do
        if A[k] > A[j] then
            k ← j
        end if
    end for
    scambia A[i] con A[k]
end for
return A
```

Correttezza del Selection Sort

Invariante del Ciclo Esterno:

- $A[1..i-1]$ è ordinato
- Se x è in $A[i..n]$ e y in $A[1..i-1]$ allora $x \geq y$
- **Inizializzazione:**

- Con $i=1$, $A[1..i-1]$ corrisponde a $A[1..0]$ ovvero è vuoto dunque è per forza di cose vere la proposizione

- **Mantenimento:**

  - Assumiamo che $A[k]$ contenuto in $A[i..n]$ sia l'elemento minimo e deleghiamo la correttezza di ciò al ciclo interno
  - $A[k]$ minimo in $A[i..n]$ ma $\geq$ di qualunque valore in $A[1..i-1]$
  - Scambiando $A[k]$ con $A[i]$ e aumentando i di 1, l'invariante si mantiene

Invariante del Ciclo Interno:

- $A[k]$ è minimo in $A[i..j-1]$

- **Inizializzazione:**

  - All'inizio $j=i+1$ dunque $A[i..j-1]$ si riduce a $A[i..i]$, dunque è banalmente vero che $A[k]$ sia minimo.

- **Mantenimento:**
  come ipotesi induttiva assumiamo che l'invariante vale prima
  di eseguire il ciclo
  il corpo del ciclo aggiorna la posizione del massimo se
  $A[k]>A[j]$ e, in ogni caso, incrementa j
  dunque l'invariante viene mantenuto

Complessità del Selection Sort

Tralasciando vari calcoli, possiamo facilmente notare che ha una **complessità quadratica**. Infatti abbiamo un **for** interno che viene eseguito n volte.
Dunque il corpo del ciclo **for** interno viene eseguito n' volte moltiplicato per n.

Come esercizio, provare a calcolare il tempo computazionale

$$
\begin{aligned}
C^{min}(n) &= \text{n. confronti nel caso migliore} \\
C^{max}(n) &= \text{n. confronti nel caso peggiore} \\
S^{min}(n) &= \text{n. spostamenti nel caso migliore} \\
S^{max}(n) &= \text{n. spostamenti nel caso peggiore}
\end{aligned}
$$

Altri esercizi da fare. Spostamenti è da intendere come scambi, kinda

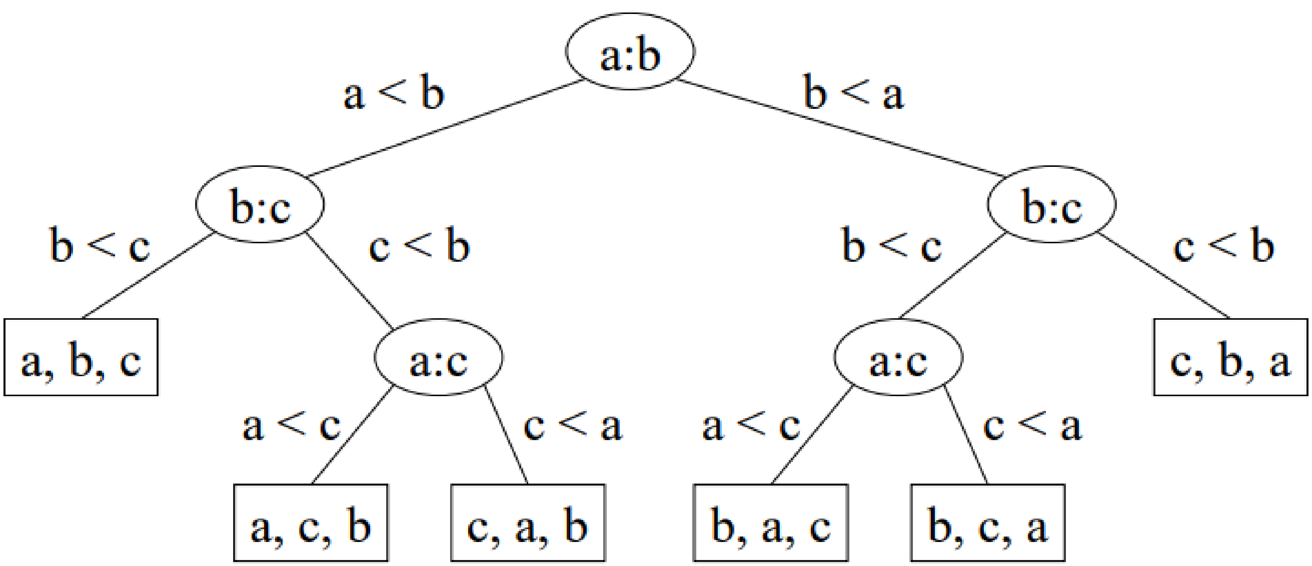
Alberi di decisione

E' un albero che rappresenta le esecuzioni di un algoritmo:

- i **nodi interni** rappresentano **decisioni da prendere**
- le **foglie** rappresentano possibili uscite (**output**)
- i **rami** rappresentano particolari **esecuzioni**.

L'albero di decisione che minimizza l'altezza rappresenta un **limite inferiore** al numero di decisioni necessarie nel caso peggiore

Esempio per l'ordinamento di 3 elementi:



Nel caso dell'ordinamento:

- $n!$ foglie (un ordinamento è una permutazione)
- i nodi interni rappresentano confronti

In un albero binario per avere k foglie ci vogliono almeno $log_2 k$ livelli.

Nel caso dell'ordinamento (*sorting*) il numero dei confronti deve essere dunque maggiore di (usando la formula di Stirling per approssimare $n!$):

$$\log_2 n! \approx \log_2\left(\sqrt{2\pi n}(n/e)^n\right) = \log_2\sqrt{2\pi n} + n\log_2(n/e) \approx n\log_2 n$$

TODO ^