Sottovettore di Somma Massimale

- Divide et Impere : $O(n \log n)$

è RICORSIVO.

```
def max_sum_rec(A, i, j):
    if (i == j):
        return max(0, A[i])

    m = (i+j)/2

    maxLL = 0          # sottovettore a partire da m andando a sinistra
    sum = 0
    for k in range(m, i-1, -1):
        sum += A[k]
        maxLL = max(maxLL, sum)
    maxRR = 0          # sottovettore a partire da m+1 andando a destra
    sum = 0
    for k in range(m+1, j+1):
        sum += A[k]
        maxRR = max(maxRR, sum)
    maxL = max_sum_rec(A, 0, m)
    maxR = max_sum_rec(A, m+1, j)
    return max(maxL, maxR, maxLL + maxRR)
def maxsum3(A):
    return maxsum_rec(A, 0, len(A)-1)
```

- Programmazione Dinamica (Algoritmo di Kadane) : $O(n)$

tenere traccia delle informazioni durante la scansione dell'array.

```
def maxsum4(A):
    maxSoFar = 0
    maxHere = 0
    start = end = 0
    last = 0          # ~~inizio~~ fine del sottovettore massimale fino ad ora?
    for i in range(0, len(A)):
        maxHere += A[i]
        if maxHere <= 0:
            maxHere = 0
```

```
        last = i+1

    if maxHere > maxSoFar :
        maxSoFar = maxHere
        start, end = last, i

return (start, end)
```

BINARY SEARCH

In un array ORDINATO, data in input un valore, si vuole restituire il suo indice nell'array, se presente.

- **Ricorsiva:** (divide et impera)

Si divide a metà e si ricerca, chiamando ricorsivamente, nella metà che contiene il valore.

```
def BinarySearch (A, low, high, value):
    if high < low :
        return ⊥
    mid = (low + high)/2
    if A[mid] > value :          # cerca a SINISTRA
        return BinarySearch (A, low, mid-1, value)
    if A[mid] < value :          # cerca a DESTRA
        return BinarySearch (A, mid+1, high, value)
    return mid
```

- **Iterativa:** (divide et impera)

Si divide a metà, ma con un <u>CICLO WHILE</u>, "giocando" con gli estremi high e low.

```
def BinarySearch (A, value):
    low = 0
    high = len (A)
    while (low <= high):
        mid = (low + high)/2
        if A[mid] > value :
            high = mid-1
        elif A[mid] < value :
            low = mid + 1
        else :
            return mid
    return ⊥
```

Relazioni di Ricorrenza

Una Relazione di Ricorrenza è un'EQUAZIONE MATEMATICA che esprime una ricorsione. Descrive a_n in funzione di $a_0, a_1, \dots, a_{n-1}$, $\forall n \geq n_0$, $n_0 \in \mathbb{N}^+$.

È LINEARE, di grado K $\iff$
$$\begin{cases} a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + F(n) \\ c_k \neq 0 \end{cases}$$

- Lineari Omogenee (1):

$a_n = c_1 a_{n-1} + c_2 a_{n-2}$ ha come soluzione: $\boxed{a_n = \alpha_1 r_1^n + \alpha_2 r_2^n}$

con r_1 ed r_2 RADICI DISTINTE del polinomio caratteristico: $r^2 - rc_1 - c_2 = 0$

$$\Rightarrow \begin{cases} a_0 = \alpha_1 + \alpha_2 \\ a_1 = \alpha_1 r_1 + \alpha_2 r_2 \end{cases} \Rightarrow \begin{cases} \alpha_1 = a_0 - \alpha_2 \\ r_2 \alpha_2 = a_1 - a_0 r_1 + r_1 \alpha_2 \end{cases} \Rightarrow \begin{cases} \alpha_1 = \dfrac{a_1 - a_0 r_2}{r_1 - r_2} \\ \alpha_2 = \dfrac{a_0 r_1 - a_1}{r_1 - r_2} \end{cases}$$

- Esempio:

$a_n = a_{n-1} + 2a_{n-2}$, con $a_0 = 2$ e $a_1 = 7$

$\Rightarrow c_1 = 1$ e $c_2 = 2$; il polinomio caratteristico è $r^2 - r - 2 = 0$

$\Rightarrow (r-2)(r+1) = 0 \Rightarrow \begin{matrix} r_1 = 2 \\ r_2 = -1 \end{matrix} \Rightarrow a_n = \alpha_1 \cdot 2^n + \alpha_2 \cdot (-1)^n$

$$\begin{matrix} a_0 = \\ a_1 = \end{matrix} \begin{cases} \alpha_1 + \alpha_2 = 2 \\ 2\alpha_1 - \alpha_2 = 7 \end{cases} \Rightarrow \begin{cases} \alpha_1 = 2 - \alpha_2 \\ 4 - 2\alpha_2 - \alpha_2 = 7 \end{cases} \Rightarrow \begin{cases} \alpha_1 = 3 \\ \alpha_2 = -1 \end{cases}$$

$\Rightarrow$ Soluzione: $\boxed{a_n = 3 \cdot 2^n - (-1)^n}$

- Lineari Omogenee (2)

$a_n = c_1 a_{n-1}$ e $c_2 a_{n-2}$ $\longrightarrow$ Pol. Caratteristico: $r^2 - c_1 r - c_2 = 0$

$\rightarrow$ Se il polinomio ha una RADICE con MOLTEPLICITÀ 2, detta r_0:

$\Rightarrow$ La soluzione cercata è: $\boxed{a_n = \alpha_1 r_0^n + n \cdot \alpha_2 r_0^n}$

• LINEARI NON OMOGENEE:

Se $\{a_n^{(p)}\}$ è una soluzione particolare di $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} + F(n)$

e $\{a_n^{(h)}\}$ è soluzione dell' OMOGENEA ASSOCIATA $\Rightarrow$ Tutte le soluzioni sono: $\{a_n^{(h)} + a_n^{(p)}\}$

• ESEMPIO:

$a_n = 3a_{n-1} + 2n$, con $a_1 = 3$

Dunque, $c_1 = 3$ ed $F(n) = 2n$. $\Rightarrow$ Polinomio Caratteristico: $r - 3 = 0 \Rightarrow r_0 = 3$

$\Rightarrow a_n^{(h)} = \boxed{\alpha 3^n}$;

Ora, cerchiamo $a_n^{(p)}$ col METODO della SOMIGLIANZA:

sia $q(n) = bn^2 + cn + d$

$$\Rightarrow bn^2 + cn + d = 3bn^2 - 6bn + 3b + 3cn + 3d + 2n - 3c$$

$$\Rightarrow \begin{cases} b = 0 \\ -2cn = 2n \\ -2d = -3c \end{cases} \Rightarrow \begin{cases} b = 0 \\ c = -1 \\ d = -\frac{3}{2} \end{cases} \Rightarrow q(n) = a_n^{(p)} = -n - \frac{3}{2}$$

$$\Rightarrow \text{Soluzione è } a_n = \underbrace{\alpha 3^n}_{a_n^{(h)}} \underbrace{- n - \frac{3}{2}}_{a_n^{(p)}}$$

Applichiamo ora le condizioni iniziali: $a_1 = 3 \Leftrightarrow 3\alpha - 1 - \frac{3}{2} = 3$,

quindi $\alpha = \frac{11}{6}$ $\Rightarrow$ $\boxed{a_n = \frac{11}{6} \cdot 3^n - n - \frac{3}{2}}$

Analisi Asintotica

Siano $f(n): \mathbb{N} \mapsto \mathbb{R}$ e $g(n): \mathbb{N} \mapsto \mathbb{R}$ due funzioni tali che $f, g > 0$ e NON DECRESCENTI:

- O GRANDE:

$$f(n) = O(g(n)) \iff \exists c \in \mathbb{R},\ c > 0 \text{ ed } n_0 \in \mathbb{N},\ n_0 \geq 1 \text{ tali che: } \boxed{f(n) \leq c \cdot g(n)},\ \text{per } n > n_0$$

Cioé, da un certo punto in poi, $f(n) \leq c \cdot g(n)$, quindi $g(n)$ è un UPPER BOUND per f.

- Ω GRANDE:

$$f(n) = \Omega(g(n)) \iff \exists c \in \mathbb{R},\ c > 0 \text{ ed } n_0 \in \mathbb{N},\ n_0 \geq 1 \text{ tali che: } \boxed{f(n) \geq c \cdot g(n)},\ \text{per } n > n_0$$

Da un certo punto in poi, $f(n)$ sta sempre sopra $c \cdot g(n)$, che è un LOWER BOUND.

- Θ GRANDE:

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$$

Quindi, se e solo se:

$$\exists c_1, c_2 \in \mathbb{R},\ c_1, c_2 > 0 \text{ ed } \exists n_0 \in \mathbb{N},\ n_0 \geq 1,\ \text{tale che: } \boxed{c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)},\ \text{per } n > n_0$$

Analisi delle Ricorrenze

(1) Analisi dei Livelli: si sviluppa la relazione di ricorrenza fino al caso base, in un albero i cui nodi rappresentano i costi ai vari LIVELLI della ricorsione.

(2) Analisi per Tentativi o per Sostituzione: si "indovina" intuitivamente una soluzione e si dimostra che è giusta per INDUZIONE.

(3) Ricorrenze Comuni: si usa il MASTER THEOREM.

- **Analisi dei Livelli: Binary Search**

La relazione di ricorrenza è:
$$T(m) = \begin{cases} c & \text{se } m = 0 \\ T(m/2) + d & \text{se } m > 0 \end{cases}$$

Espandiamo la relazione:

$$\begin{aligned} T(m) &= d + T(m/2) \\ &= d + d + T(m/4) \\ &= d + d + d + T(m/8) \\ &\dots \\ &= d + d + d + \dots + d + T(1) \overset{\text{Se } m = 2^k}{=} d \log m + c \end{aligned}$$

$$\Rightarrow \boxed{T(m) = \Theta(\log m)}$$

- **Analisi dei Livelli: 2° Esempio**

$$T(m) = \begin{cases} 1 & \text{se } m = 0 \\ 4T(m/2) + m & \text{se } m > 0 \end{cases}$$

Espandendo, e supponendo sempre $m = 2^k$:

$$T(m) = 4T(m/2) + m = m + \frac{4m}{2} + 16T(m/4) = m + 2m + \frac{16m}{4} + 64T(m/8) = \dots$$

$$= m + 2m + 4m + 8m + \dots + 2^{\log m - 1} \cdot m + 4^{\log m} \cdot T(1) =$$

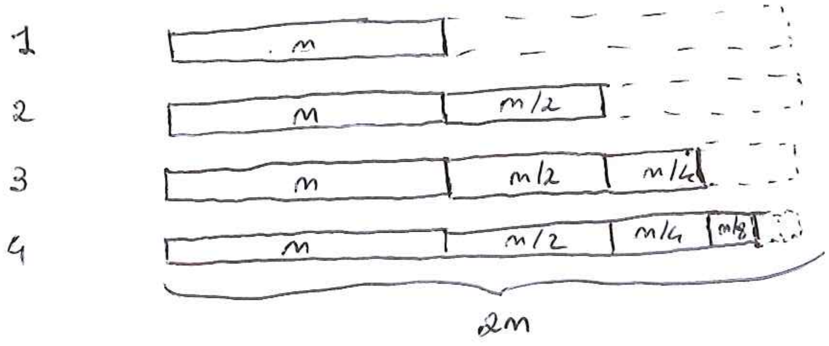
$$= m \cdot \sum_{J=0}^{\log m - 1} 2^J + 4^{\log m} = m \cdot \frac{(2)^{\log m \cancel{-1+1}} - 1}{2-1} + 2^{2\log m} = m \cdot \frac{m-1}{1} + 2^{\log m^2} =$$

$$= m^2 - m + m^2 = 2m^2 - m$$

$$\Rightarrow \boxed{T(m) = \Theta(m^2)}$$

- **Metodo di Sostituzione: esempio**

$$T(m) = \begin{cases} 1 & \text{se } m \leq 1 \\ T(\lfloor m/2 \rfloor) + m & \text{se } m > 1 \end{cases}$$



$\Rightarrow$ Si intuisce che la soluzione potrebbe essere $2m$.

- Dim. che sia $O(m)$:

Se $T(m) = O(n) \Rightarrow \exists c>0, \exists m_0 \geq 1 : T(m) \leq c \cdot m, \forall m > m_0$

CASO BASE: $T(1) = 1 \leq 1 \cdot c, \forall c \geq 1$

PASSO INDUTTIVO: Sia che $\forall k < m: T(k) \leq ck$

$$T(m) = T(\lfloor m/2 \rfloor) + m \leq c \cdot \lfloor m/2 \rfloor + m \leq c \cdot (m/2) + m = m \cdot \left(c/2 + 1\right) \leq c \cdot m$$

Se e solo se $\frac{c}{2} + 1 \leq c \iff c \geq 2$

$\Rightarrow$ Nel caso base $T(m) \leq cm$ per $c \geq 1$, nel passo induttivo $T(m) \leq cm$ per $c \geq 2$

e vale per $m \geq 1 \Rightarrow c = 2, m_0 = 1$

$\Rightarrow \boxed{T(m) = O(m)}$

- Dim. che sia $\Omega(m)$:

Se $T(m) = \Omega(m) \Rightarrow \exists c>0, \exists m_0 \geq 1 : T(m) \geq c \cdot m, \forall m > m_0$

CASO BASE: $T(1) = 1 \geq 1 \cdot c \iff c \leq 1$

PASSO INDUTTIVO: Sia che $\forall k < m: T(k) \geq ck$

$$T(m) = T(\lfloor m/2 \rfloor) + m \geq c \lfloor m/2 \rfloor + m \geq cm/2 - 1 + m = \left(\frac{c}{2} - \frac{1}{m} + 1\right) \cdot m \geq cm$$

Se e solo se: $\frac{c}{2} - \frac{1}{m} + 1 \geq c \iff c \leq 2 - \frac{2}{m}$

Per i casi NON BASE, il minimo valore che $2 - \frac{2}{m}$ puo' assumere è $2 - \frac{2}{2} = 1$

$\Rightarrow$ Poiché per il caso base $c \leq 1$ e negli altri casi $c \leq 2 - \frac{2}{m}$, $\forall m \geq 2$, si puo' prendere $c = 1$ ed $m_0 = 1$

$\Rightarrow \boxed{T(m) = \Omega(m)} \Rightarrow$ Quindi, abbiamo dimostrato che $\boxed{T(m) = \Theta(m)}$

Master Theorem

Si usa per Relazioni di Ricorrenze del seguente tipo:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & \text{se } n > 1 \\ d & \text{se } n \leq 1 \end{cases}$$

dove: $a \geq 1$, $b \geq 2$ $(a, b \in \mathbb{N})$ e $c > 0$, $\beta \geq 0$ $(c, \beta \in \mathbb{R})$.

Posto $\boxed{\alpha = \log_b a}$, il Master Theorem ci dice che:

$$\boxed{T(n) = \begin{cases} \Theta(n^\alpha) & , \text{se } \alpha > \beta \\ \Theta(n^\alpha \log n) & , \text{se } \alpha = \beta \\ \Theta(n^\beta) & , \text{se } \alpha < \beta \end{cases}}$$

Praticamente l'andamento è determinato dal termine più "forte" tra n^α e cn^β.

- Esempio:

Sia $T(n) = 9T\left(\frac{n}{3}\right) + n$; quindi: $a = 9$, $b = 3$, $c = 1$, $\beta = 1$

$\Rightarrow \alpha = \log_b a = \log_3 9 = 2 \Rightarrow \alpha > \beta$: siamo nel primo caso del Teorema,

quindi: $\boxed{T(n) = \Theta(n^2)}$

- Esempio (Binary Search):

$T(n) = T(n/2) + 1 \Rightarrow a = 1$, $b = 2$, $c = 1$, $\beta = 0$

$\alpha = \log_2 1 = 0 \Rightarrow \alpha = \beta = 0 \rightarrow 2°$ caso del Master Theorem:

$T(n) = \Theta(n^\alpha \log n) = \boxed{\Theta(\log n)}$

Ricorrenze Lineari di Ordine Costante

Se abbiamo una Relazione di Ricorrenza della forma:

$$T(n) = \begin{cases} \sum_{1 \le i \le h} a_i T(n-i) + cn^{\beta} & \text{se } n > m \\ \Theta(1) & \text{se } n \le m (\le h) \end{cases}$$

dove $a_i \geq 0 \in \mathbb{N}$, $c, \beta \in \mathbb{R}$, $c > 0$, $\beta \geq 0$

Sia $\boxed{a = \sum_{i=1}^{h} a_i}$ $\Longrightarrow$ $\boxed{T(n) = \begin{cases} \Theta(n^{\beta+1}) & \text{se } a = 1 \\ \Theta(a^n \cdot n^{\beta}) & \text{se } a > 1 \end{cases}}$

- Esempio 1:

Se $T(n) = T(n-10) + n^2$, si può scrivere come $T(n) = \sum_{i=1}^{10} a_i \cdot T(n-i) + 1 \cdot n^2$

Si ha che $a_1, a_2, \ldots, a_9 = 0$, $a_{10} = 1$, $c = 1$, $\beta = 2$.

$a = \sum_{1 \le i \le 10} a_i = 1$ $\Rightarrow$ Poiché $a = 1 \Rightarrow T(n) = \Theta(n^{\beta+1}) = \boxed{\Theta(n^3)}$

- Esempio 2: Fibonacci

$$T(n) = T(n-2) + T(n-1) + 1$$

$\Rightarrow a_1 = 1$, $a_2 = 1$, $c = 1$, $\beta = 0$ $\Rightarrow$ $a = 1 + 1 = 2$

$\Rightarrow$ Poiché $a > 1 \Rightarrow T(n) = \Theta(2^n \cdot n^0) = \boxed{\Theta(2^n)}$ costo ESPONENZIALE

Analisi Ammortizzata

Per il <u>CASO PESSIMO</u> su Strutture Dati.

- <u>Metodo dell'aggregazione</u>: si calcola matematicamente la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo.
- <u>Metodo degli accantonamenti</u>: ogni operazione ha un costo AMMORTIZZATO, in modo da equilibrare operazioni più e meno costose.
- <u>Metodo del potenziale</u>: si associa ad una struttura dati D una funzione potenziale $\Phi(D)$. Il COSTO AMMORTIZZATO è la somma del costo effettivo e della variazione di potenziale (è equivalente, grossomodo, al metodo degli accantonamenti).

Programmazione Dinamica

Molto utile per spezzare problemi (spesso ricorsivi) in sottoproblemi, le cui soluzioni vengono salvate in una tabella (di accesso $O(1)$) rendendo veloce la risalita delle chiamate ricorsive che avevamo prima.

Vediamola su <u>Fibonacci</u>:

$$D_n = \begin{cases} 1 & \text{se } n \leq 1 \\ D(n-2) + D(n-1) & \text{se } n > 1 \end{cases}$$

Si utilizza un array che memorizza solo gli ultimi 2 elementi calcolati, dato che sono quelli che servono. (bastano 3 variabili anziché un array):

```
def fibonacci_din(n):
    F0 = 1
    F1 = 1
    F2 = 1

    for i in range(2, n):
        F0 = F1
        F1 = F2
        F2 = F1 + F0
    return F2
```

→

Complessità temporale:

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

Complessità spaziale:

$S_n = \Theta(1)$, anziché $\Theta(n)$ nel memorizzare tutto l'array.

* Utilizzando delle matrici, con una variante del DIVIDE ET IMPERA, si riesce a ridurre il tempo d'esecuzione a $T(n) = \Theta(\log n)$

KNAPSACK

Si ha uno zaino di capacità C ed un insieme di oggetti, ciascuno con un PESO ed un PROFITTO. Si vuole ottenere il MASSIMO PROFITTO possibile, tale che gli oggetti scelti siano contenibili nello zaino.

Abbiamo: (1) Vettore dei pesi $\underline{w}$ ($w[i]$ = peso i-esimo)

(2) Vettore dei profitti $\underline{p}$ ($p[i]$ = profitto i-esimo)

(3) Capacità C dello zaino.

Sia n il numero di oggetti e sia $DP[i][c]$ il MASSIMO PROFITTO OTTENIBILE dai primi i oggetti con uno zaino di capacità $c < C$.

Il MASSIMO PROFITTO che cerchiamo sarà $\boxed{DP[n][C]}$.

- Esaminiamo cosa succede quando prendiamo o no un oggetto: (l'ultimo)

(1) Se NON lo prendiamo: $\boxed{DP[i][c] = DP[i-1][c]}$

la capacità è la stessa (c) ed è come se non lo considerassimo e vorremmo il max profitto per i primi $(i-1)$ oggetti con capacità disponibile c.

(2) Se LO prendiamo: $DP[i][c] = DP[i-1][c-w[i]] + p[i]$

Prendendolo ~~aumenta~~ diminuisce la capacità disponibile (di $w[i]$), ma aumenta il profitto ($p[i]$). Ora il problema si riduce ad $(i-1)$ oggetti, MA con capacità disponibile pari a $(c-w[i])$.

- COME SCEGLIERE COSA FARE?

Si fa RICORSIVAMENTE: $\boxed{DP[i][c] = \max\Big(DP[i-1][c],\ DP[i-1][c-w[i]] + p[i]\Big)}$

- Se gli oggetti finiscono, o finisce la capacità, il profitto è zero.

Se la capacità è negativa, il profitto è $-\infty$:

$$DP[i][c] = \begin{cases} 0 & \text{se } i=0 \text{ oppure } c=0 \\ -\infty & \text{se } c<0 \end{cases}$$

Dunque, abbiamo:

$$DP[i][c] = \begin{cases} -\infty & \text{se } c < 0 \\ 0 & \text{se } i = 0 \text{ o } c = 0 \\ \max\left(DP[i-1][c],\ DP[i-1][c-W[i]] + p[i]\right) & \text{altrimenti} \end{cases}$$

- **Algoritmo in Pseudocodice:**

```
KNAPSACK (w, p, m, C):
   DP = [0...m][0...C]
   for i ← 0 to m:                    # se i=0, DP[i][c]=0
       DP[i][0] = 0
   for c ← 0 to C:                    # se c=0, DP[i][c]=0
       DP[0][c] = 0

   for i ← 0 to m:
       for c ← 0 to C:
           if W[i] ≤ c then:
               DP[i][c] = max (DP[i-1][c], DP[i-1][c-W[i]] + p[i])

           else:                      # se il peso è troppo grande, conserva le
               DP[i][c] = DP[i-1][c]  # scelte fatte finora, le alternative sono
                                      # analizzate più avanti.
   return DP[m][C]
```

L'algoritmo consiste nel memorizzare i risultati di tutte le possibili scelte in una MATRICE. Il massimo è l'ultimo elemento.

- **ANALISI DELLA COMPLESSITÀ**

Ha complessità $T(m) = O(mC)$ poiché è una matrice $m \cdot C$.

<u>NON</u> è un ALGORITMO POLINOMIALE.

C, per essere rappresentato, ha bisogno di $k = \lceil \log_2 C \rceil \Rightarrow C \simeq 2^k$

$\Rightarrow \boxed{T(m) = O(m \cdot 2^k)}$ → Algoritmo <u>PSEUDO-POLINOMIALE</u>

3. Ordinamento

• Insertion Sort:

Prende gli elementi a partire dal primo e li inserisce nella posizione corretta fino a quel punto, tramite degli swap tra elementi adiacenti fino a giungere alla posizione giusta.

```
InsertionSort (A, n):
  for i ← 0 to n-1:
    key ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > key:
        A[j+1] ← A[j]          # swap fin quando key ≥ A[j]
        j ← j-1
    A[j+1] ← key
```

* Costo: $\Theta(n^2)$

• Stupid Sort:

```
StupidSort (A):
  while not sorted(A):
    A ← random_permutation (A)
```

• Bozo Sort:

```
BozoSort (A)
  while not sorted (A):
    A ← invert_two_elements (A)
```

• Selection Sort (naïf):

Cerco il MINIMO e lo metto con uno swap nella prima posizione ancora da ordinare:

```
SelectionSort (A, n):
  for i ← 0 to n-1:
    min ← i
    for j ← (i+1) to n-1:
      if A[j] < A[min] then:
        min ← j                 # indice del minimo
    if min != i then:
      tmp ← A[i]                # swap
      A[i] ← A[min]
      A[min] ← tmp
```

Analisi Costo

Valuto l'operazione DOMINANTE: doppio ciclo for, il secondo da i ad n-1.

$$T(n) = \sum_{i=0}^{n-1} (n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

- **BUBBLE SORT :**

Confronta ogni elemento col successivo, se il primo è più grande, li scambia. C'è una variabile che controlla se in un ciclo è stato fatto almeno 1 scambio, se no, l'algoritmo termina.

```
BUBBLESORT(A, n):
    scambio ← true
    while scambio :
        scambio ← false
        for i ← 0 to n-1 :
            if A[i] > A[i+1] :
                swap(A[i], A[i+1])
                scambio ← true
```

ANALISI COSTO

L'OPERAZIONE DOMINANTE è l'if, che, sia nel caso medio che nel caso pessimo, viene eseguito $\frac{n^2}{2}$ volte.

$\Rightarrow \boxed{T(n) = \Theta(n^2)}$

- **MERGE SORT (Divide et Impera) :**

Utilizza il DIVIDE ET IMPERA in maniera RICORSIVA sulle metà in cui viene diviso l'array.

1) Dividere l'array in 2 metà;
2) Ordinare ricorsivamente le 2 metà, se ha lunghezza 1 è già ordinata;
3) Usare la funzione MERGE per "fondere" le 2 metà confrontando ripetutamente il minimo delle due sottosequenze.

```
MERGESORT(A, left, right):
    if left < right then:
        center ← (left + right) / 2
        MERGESORT(A, left, center)
        MERGESORT(A, center+1, right)
        MERGE(A, left, center, right)
```

La funzione MERGE() opera nel seguente modo:

(1) confronto il più piccoli elementi ancora da inserire e li metto in ordine in un array d'appoggio B (i sottovettori sono già ordinati);

(2) Copio, se ci sono, i rimanenti elementi dal centro indietro fino all'ultimo elemento che ho confrontato (da center ad i)

(3) Copio gli elementi di B in A, fino a quelli inseriti in B (da left a k-1).

MERGE (A, left, center, right):

```
i ← left
j ← center + 1
k ← left
while i ≤ center and j ≤ right:
    if A[i] ≤ A[j]:
        B[k] ← A[i]
        i ← i+1
    else:
        B[k] ← A[j]
        j ← j+1
    k ← k+1

j ← right
for h ← center downto i:      # Se j > right, ma i ≤ center, si copiano gli elementi
    A[j] ← A[h]               # da i a center in fondo all'array A
    j ← j-1

for j ← left to k-1:          # Copia di B in A (fino a dove spostati)
    A[j] ← B[j]
```

ANALISI DEL COSTO:

Se $n = 2^k \Rightarrow$ numero di suddivisioni $= k = \log n$

$$\Rightarrow T(n) = \begin{cases} c & \text{se } n = 1 \\ 2(T(n/2)) + dn & \text{se } n > 1 \end{cases}$$

$\Rightarrow a = 2, \; b = 2, \; c = d, \; \beta = 1 \;;\quad \alpha = \log_b a = \log_2 2 = 1$

$\Rightarrow \alpha = \beta = 1 \Rightarrow$ Per il MASTER THEOREM: $\boxed{T(n) = \Theta(n \log n)}$

• BUCKET SORT:

Funziona molto bene se gli elementi sono distribuiti abbastanza uniformemente in un certo intervallo.

(1) Genera un numero K di BUCKETS;

(2) SCATTER: scandisci il vettore e metti gli elementi nel bucket appropriato;

(3) Ordina i singoli BUCKETS (di solito con INSERTION SORT → $O(n^2)$)

(4) GATHER: concatena i buckets ordinati ed inseriscili nel vettore.

BUCKET SORT (A, K):

buckets ← vettore di K bucket

M ← max del vettore   # costo $O(n)$ con ARRAYMAX()

for i ← 0 to len(A) - 1:

  inserisci A[i] ~~nel bu~~ in buckets $[\lfloor K * A[i] / M \rfloor]$ # fase di SCATTER

for i ← 0 to K - 1:

  SORT (buckets [i])   # di solito INSERTION SORT()

return Concatenazione di buckets [1], ..., buckets [K]   # fase di GATHER

ANALISI DEL COSTO:

* CASO PESSIMO: tutti gli elementi in 1 solo bucket → $O(n^2)$ dovuto al l'INSERTION SORT.

* CASO MEDIO: facendo un analisi probabilistica, con calcolo dei valori medi, si ottiene: $\boxed{T(n) = O\left(\frac{n^2}{k} + n + k\right)}$

⇒ Se si prende $K = \Theta(n) \Rightarrow T(n) = O(n + n + n) = O(n)$

Com'è giusto che sia, la complessità e l'efficienza dell'algoritmo dipendono dal numero di bucket impiegati.

• Quick Sort (Divide et Impera):

(1) Selezione un elemento dell'array: il PIVOT;

(2) Gli elementi più grandi del PIVOT vengono spostati a destra, i più piccoli a sinistra, tramite degli scambi;

(3) Si chiama RICORSIVAMENTE sulla parte destra e sinistra.

Per la fase (2), viene usata la seguente funzione PARTITION():

```
PARTITION (A, low, high):
    pivot ← A[low + (high-low)/2]
    i ← low
    J ← high
    loop forever:
        while A[i] < pivot:
            i ← i+1
        while A[J] > pivot:
            J ← J-1
        if i >= J:                 # quando è giunto al termine, i = J = pivot
            return J               # return the position of pivot
        SWAP(A[i], A[J])
```

```
QuickSort (A, low, high):
    if low < high:
        p ← PARTITION (A, low, high)      # p = pivot
        QUICKSORT (A, low, p)             # chiamate ricorsive
        QUICKSORT (A, p+1, high)
```

* **Analisi Costo**: CASO PESSIMO: pivot = max/min (A) → $T(n) = O(n^2)$

CASO MIGLIORE: pivot = elemento medio → $\log n$ chiamate: $T(n) = O(n\log n)$

CASO MEDIO: $T(n) = O(n) + 2T(n/2) \xrightarrow[\text{M. THEOREM}]{} \boxed{T(n) = O(n\log n)}$

- **Radix Sort:**

Ordina considerando le singole cifre dei numeri. Fa Bucket Sort per ciascuna cifra dei numeri. È controintuitivo per l'uomo, ma efficace: $\boxed{T(n) = O(nk)}$

Proprietà Algoritmi di Sort

(1) **Stabilità:** è stabile se preserva l'ordine iniziale tra due elementi già ordinati con la stessa chiave.

Sono TUTTI STABILI, tranne il <u>Quick Sort</u>.

(2) **In Place:** se <u>NON</u> crea copie dell'input per ordinarlo, gli unici <u>NON IN PLACE</u> sono il <u>Merge Sort</u> (utilizza l'array B) ed il <u>Bucket Sort</u> (che copia gli elementi nei bucket).

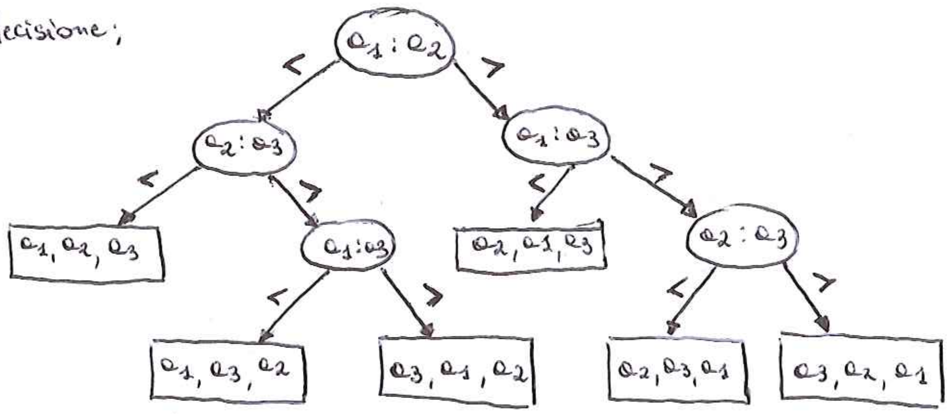
(3) **Adattatività:** se trae vantaggio dagli elementi già ordinati, in pratica lo fanno solo gli algoritmi che ordinano tramite degli scambi, quindi <u>BubbleSort</u> e <u>Insertion Sort</u>.

Problema Computazionale dell'Ordinamento

★ [La complessità temporale di un qualsiasi algoritmo di Ordinamento <u>PER CONFRONTO</u> è $T(n) = \Omega(n \log n)$, dove n è la dimensione dell'input.]

Bucket Sort ha $T(n) = O(n)$, poiché <u>NON</u> è PER CONFRONTO!

Dim: Albero di decisione:



Per ogni sequenza $a_1, \dots, a_n$ c'è un cammino nell'albero. Il numero di PERMUTAZIONI di $a_1, \dots, a_n$ è $n!$ $\longrightarrow$ $n!$ foglie $\Rightarrow$ ALTEZZA ALBERO: $h(T) = \lceil \log n! \rceil$

<u>Formula di Stirling</u>: $\boxed{\lim_{n \to +\infty} \frac{\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n}{n!} = 1}$ $\Rightarrow n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \Rightarrow n! > \left(\frac{n}{e}\right)^n$

$\Rightarrow h(T) \geq \log\left(\frac{n}{e}\right)^n = n \log n - n \log e = \underline{\Omega(n \log n)}$