LCS(X, Y)

```
m = X.[length]
n = Y.length
for i = 0 to m
    L[i,0] = 0
for j = 0 to n
    L[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i = y_j
            L[i,j] = L[i-1, j-1] + 1
            B[i,j] = '↖'
        else if L[i-1,j] ≥ L[i,j-1]
            L[i,j] = L[i-1,j]
            B[i,j] = '↑'
        else
            L[i,j] = L[i,j-1]
            B[i,j] = '←'
return (L[m,n], B)
```

X, Y

INIT.

$$l(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 & \text{(caso 0)} \\ l(i-1, j-1) + 1 & \text{se } i,j > 0 \text{ e } x_i = x_j & \text{(caso 1)} \\ \max\{l(i, j-1), l(i-1, j)\} & \text{se } i,j > 0 \text{ e } x_i \neq x_j & \text{(caso 2)} \end{cases}$$

ESAUSTIVO / BRUTE FORCE

$Y \setminus X = (n^2)$

$n \cdot n \simeq n^2$

$X = \langle b, d, c, d \rangle$

$Y = \langle a, b, c, b, d \rangle$

Restituisci $LCS(X, Y)$ e $|LCS(X, Y)|$

$$L = \begin{array}{c} \\ b \\ d \\ c \\ d \end{array} \overset{\begin{array}{cccccc} & a & b & c & b & d \end{array}}{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 1 & \mathbf{2} & 2 & 2 \\ 0 & 0 & 1 & 2 & 2 & \mathbf{3} \end{bmatrix}} \qquad B = \begin{array}{c} b \\ d \\ c \\ d \end{array} \overset{\begin{array}{ccccc} a & b & c & b & d \end{array}}{\begin{bmatrix} \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \\ \uparrow & \uparrow & \nwarrow & \leftarrow & \uparrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \end{bmatrix}}$$

Esercizio 1: Problema dello scheduling con profitti

Un'azienda deve eseguire n lavori su una singola macchina. Ogni lavoro `j` ha un tempo di esecuzione `t[j]` e un profitto `p[j]` se viene completato. I lavori possono essere eseguiti in qualsiasi ordine, ma una volta iniziato un lavoro deve essere completato senza interruzioni. L'obiettivo è massimizzare il profitto totale entro un tempo limite T.

i. Formalizzare la nozione di soluzione per il problema e il relativo profitto. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `schedule(t, p, n, T)` che, dati in input gli array dei tempi `t[1..n]` e dei profitti `p[1..n]`, il numero di lavori n e il tempo limite T, restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

iv. Dimostrare la correttezza dell'algoritmo.

→ GREEDY

n

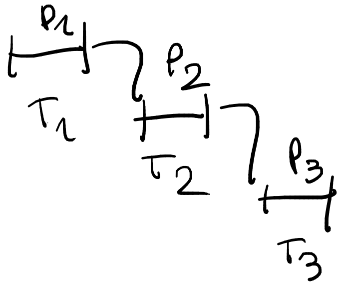
$L_1 \rightarrow T[1], P[1]$

$L_2 \rightarrow T[2], P[2]$

Un'azienda deve eseguire n lavori su una singola macchina. Ogni lavoro `j` ha un tempo di esecuzione `t[j]` e un profitto `p[j]` se viene completato. I lavori possono essere eseguiti in qualsiasi ordine, ma una volta iniziato un lavoro deve essere completato senza interruzioni. L'obiettivo è massimizzare il profitto totale entro un tempo limite T.

i. Formalizzare la nozione di soluzione per il problema e il relativo profitto. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

$\boxed{\text{CUT-AND-PASTE}}$



→ PROPRIETÀ DI SOTTOSTRUTTURA OTTIMA

$M^* =$ OTTIMO $\rightarrow M^* = M^* \cup \{P_1\}$

(ORDINA E PRENDI IL PRIMO)

[1][2][3][4] 5 6 → [MAX = 12]

OTTIMO!

GREEDY → SOLUZIONE SOB CON TEMPO PIÙ GRANDE PER PRIMO

$M^* = M^* \setminus \{T_n\}$

1 2 3 4 [5 6]

$4 > \cancel{2}$

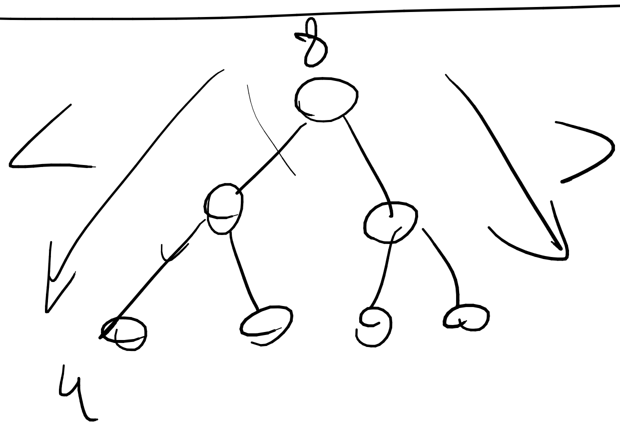
$$M^* = M^* \cup \{OPT\}$$

↘ GREEDY = OTTIMO

$$M'' = M^* \setminus \{M_0\} \Rightarrow$$ RESS. NON IN M^*

↑

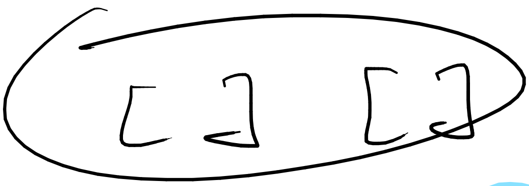
NON PIÙ OTTIMO



DIVIDE ET IMPERA

PROG. DINAMICA

GREEDY → IF

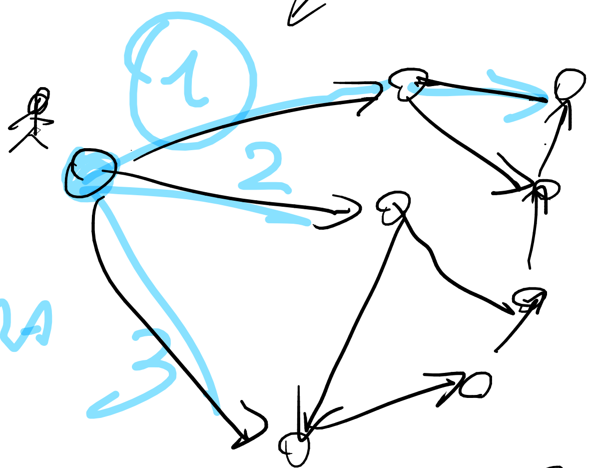


↙ GREEDY = PRENDI DISTANZA MINIMA

TSP

↙

TRAVELING SALESPERSON PROBLEM



DIJKSTRA

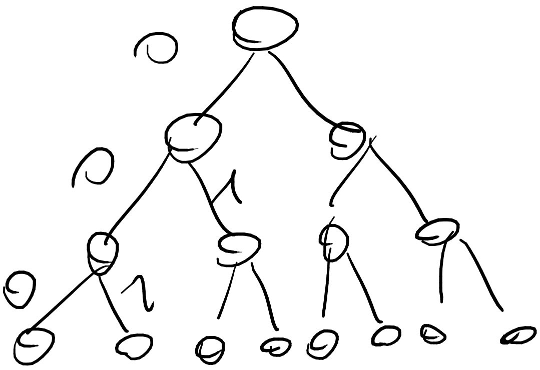
/

BELLMAN-FORD

$$n^2 \simeq n \log n$$

HUFFMAN → CARATTERI CON FREQ. MINORE

POI VAI AVANTI



$M^* = M \cup \{c_i\}$

$M' = M^* \setminus \{c_j\}$

↓

NON OTTIMO

Esercizio 1: Problema dello scheduling con profitti

Un'azienda deve eseguire n lavori su una singola macchina. Ogni lavoro `j` ha un tempo di esecuzione `t[j]` e un profitto `p[j]` se viene completato. I lavori possono essere eseguiti in qualsiasi ordine, ma una volta iniziato un lavoro deve essere completato senza interruzioni. L'obiettivo è massimizzare il profitto totale entro un tempo limite T.

i. Formalizzare la nozione di soluzione per il problema e il relativo profitto. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `schedule(t, p, n, T)` che, dati in input gli array dei tempi `t[1..n]` e dei profitti `p[1..n]`, il numero di lavori n e il tempo limite T, restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

iv. Dimostrare la correttezza dell'algoritmo.

```
schedule(t, p, n, T)

sort(t, p)
OPT = {p_1}
last = 1 // "pivot" index to select OPT
while i < n && t_i < T
        if t_i ≥ t_last && p_i ≥ p_last
                OPT = OPT U {p_i} // cut and paste
                last = i
return OPT
```

$O(n)$

Esercizio 1 (10 punti) Realizzare una funzione `union(A1,A2,n)` che dati due array di interi `A1` e `A2`, organizzati a max-heap, con capacità n, restituisce un nuovo array `A`, ancora organizzato a max-heap con capacità $2n$, che contiene l'unione insiemistica dei valori contenuti in `A1` e `A2`. Si assuma che `A1` e `A2` non contengano duplicati e si faccia in modo anche anche l'array ottenuto come unione non contenga duplicati. Ad es. se `A1` contiene i valori `3,1,2` e `A2` contiene i valori `5,2` allora l'unione A conterrà i valori `5,3,1,2`, possibilmente non in questo ordine, ovvero l'elemento `2` non è duplicato. Valutare la complessità della funzione definita.

Qualora il risultato `A` potesse contenere duplicati ci sarebbero soluzioni più efficienti?

$A_1 \to n = 3\ 1\ 2$

$A_2 \to n = 5, 2$

$A = A_1 \cup A_2$

$= 2n$

```
union(A1, A2, n)

A.heapsize = n * 2

while(i < A.heapsize)

        m1 = Max(A1)
        m2 = Max(A2)

        if(m1 > m2)
                A[i] = m1
                A[2i] = m2

        if(m2 > m1)
                A[i] = m2
                A[2i] = m1

i++

return A
```

$[\boxed{5}\ \boxed{3}, 1, 2]$

1 2 3 4

```
Funzione union(A1, A2, n):
    // Creo nuovo array con capacità 2n
    A = nuovo array[2n]
    i = 0

    // Copio A1 in A evitando duplicati
    Per ogni elemento x in A1:
        A[i] = x
        i = i + 1

    // Copio A2 in A evitando duplicati con A1
    Per ogni elemento x in A2:
        Se x non è presente in A[0...i-1]:
            A[i] = x
            i = i + 1

    // Costruisco max-heap su A
    Per j da floor(i/2) fino a 0 con passo -1:
        heapify(A, j, i)

    Restituisci A
```

$= O(n \log(n))$

= HEAPIFY

=

CHIAMA MAX-HEAPIFY

PIÙ EFFICIENTE $\Rightarrow$ USI SOLO ARRAY...

$O(n)$

$\Downarrow$

CONCATENAZIONE

Domanda A (6 punti) Realizzare una funzione booleana di tipo divide et impera `Ord(A,p,r)` che verifica se l'array `A[p,r]` è ordinato in senso crescente. Scrivere lo pseudocodice e valutare la complessità con il master theorem.

p [q] R

1 2 $\boxed{3}$ 4 5 6

p q R

```
Ord(A, p, r)

        if(p == r || p > r)
                return true

        q = floor(p + r) / 2

        while(p < r && ordl && ordr)
                if(A[p] < A[q])
                        ordl = Ord(A, p, q-1)
                if(A[q] < A[r])
                        ordr = Ord(A, q+1, r)

        return (ordl, ordr) // 1 = True, 0 = False
```

$\Rightarrow 2 \cdot T\left(\frac{n}{2}\right) \simeq \Theta(n \log(n))$

$T(n) \leq c(n \log(n))$

$\downarrow$

$\forall n, \exists c > 0 \mid$ "IT WORKS"

$\uparrow$

$Q(\vec{x}, y)$

Domanda A (7 punti) Dare la definizione della classe $\Theta(f(n))$. Mostrare che la ricorrenza

$$T(n) = \frac{3}{4}T(n/3) + T(2n/3) + 2n$$

ha soluzione in $\Theta(n)$.

$\Theta(f(n)) = \{g(n) : \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0, c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$

$$T(n) = \frac{3}{4}T(n/3) + T(2n/3) + 2n$$

$a = \frac{3}{4}, b = 3, f(n) = 2n$

$$\lim_{n \to \infty} \frac{f(n)}{n^{\log_b(a)}} = \frac{2n}{n^{\log_3(\frac{3}{4})}} \simeq$$

CASO 3 $\leftarrow \Omega(n^{\log_b(a)+\varepsilon})$

$a(f(\frac{n}{b})) \leq K \cdot f(n) \qquad 0 < K < 1$

$$T(n) = \frac{3}{4}T(n/3) + T(2n/3) + 2n$$

$\frac{\cancel{3}}{\cancel{4}_2}\left(\frac{\cancel{2}n}{\cancel{3}}\right) \leq K \cdot 2n$

$\forall n \geq 0, \; K \geq \frac{1}{4}$

$\frac{\cancel{n}}{2} \leq K \cdot 2\cancel{n}$

$\frac{1}{2} \cdot \frac{1}{2} \leq \frac{\cancel{2}K}{\cancel{2}}$

Domanda A (8 punti) Si consideri la seguente funzione ricorsiva con argomento un intero $n \geq 0$

```
val(n)
   if n <= 2
      return 1
   else
      return val(n-1) + val(n-2) + val(n-2)
```

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

Determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è $\Omega(2^n)$. La complessità è anche $O(2^n)$? Motivare le risposte.

$0 \Rightarrow (-1) + (-2) + (-2)$

≤ 0

$n+1 \Rightarrow (n+\cancel{1-1})\ (n+1-2)$

$(n+1-2)$

$\approx (n-1)(n-1)$

$(n-1)^2$

$\Omega(2^n)$

$\Downarrow$

$T(n-1) + 2T(n-2) \geq d(2^n)$

$2^n - 1 + 2(2^n - 2) \geq d2^n$

$2^n - 1 + 2^{n+1} - 4 \geq d2^n$

$2^n + 2^{n+1} - 5 \geq d2^n$

$2^n(1+2) - \cancel{5} \geq d2^n$

$d \leq 3$

Dimostriamo con il metodo di sostituzione che $T(n) = \Omega(2^n)$, ovvero dimostriamo che esistono $d > 0$ e n_0 tali che $T(n) \geq d \cdot 2^n$, per $n \geq n_0$. Si procede per induzione su n:

$$\begin{aligned}
T(n) &= T(n-1) + 2T(n-2) + c && \text{[dalla definizione della ricorrenza]}\\
&\geq d\,2^{n-1} + 2\,d\,2^{n-2} + c && \text{[ipotesi induttiva]}\\
&\geq d\,2^{n-1} + 2\,d\,2^{n-2} && \text{[poiché } c > 0\text{]}\\
&= d(2^{n-1} + 2 \cdot 2^{n-2})\\
&= d\,2^n
\end{aligned}$$

```
for i=1 to n-2 do
   for j=n-2 downto i do
      C[i,j] <- C[i-1,j-1] * C[i,j+1]
      m <- MIN(m,C[i,j])
return m
```

RISOLVI LA SOMMA PIÙ ESTERNA

(b)

$$T(n) = \sum_{i=1}^{n-2}\sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-2} n-1-i = \sum_{k=1}^{n-2} k = (n-1)(n-2)/2.$$

$\rightarrow K = \frac{n(n-1)}{2}$ ↑ GAUSS

BOTTOM - UP → ALGORITMO

MEMOIZZAZIONE → INIT/REC

Esercizio 2 (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo n, l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia n. Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

(a) Fornire un valore di n per cui l'algoritmo greedy *non* restituisce una soluzione ottima.

(b) Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo $M^\star$ di monete di valore totale n contiene la scelta greedy.

(a) $\frac{20}{1}\ \frac{20}{1}\ \frac{20}{1} > \frac{50}{1} + \frac{1 \cdot 10}{10}$

(b) $[10 - 5 - 1] \rightarrow$ MULTIPLI $\rightarrow |M^*| = n$

$\rightarrow n = 20$

$M^* =$ insieme ottimo

2 da 10

ottimo $= \left[\frac{20}{10}, \frac{20}{5}, \frac{20}{1}\right] = 20$ (2, 4)

$$M^* = M^* \left[\frac{m}{m_i} \right]$$

RAPPORTO OTTIMO...

$$M' = M^* \setminus M' \quad , m_s \notin M^*$$

$$M^* = M^* \cup m_s \rightarrow \text{NO OTTIMO}$$

COSTRUISCI CONTROESEMPIO...