05/01

Domanda A (8 punti) Si consideri la funzione ricorsiva `search(A,p,r,k)` che dato un array A, ordinato in modo crescente, un valore k e due indici p, q con $1 \le p \le r \le A.length$ restituisce un indice i tale che $p \le i \le r$ e $A[i] = k$, se esiste, e altrimenti restituisce 0.

```
search(A,p,r,k)
   if p <= r
      q = (p+r)/2
      if A[q] = k
         return q
      elseif A[q] < k
         return search(A,q+1,r,k)      T(n/2)
      else
         return search(A,p,q-1,k)      T(n/2)
   else
      return 0
```

$\to T(n) = T(n/2) + c$

Dimostrare induttivamente che la funzione è corretta. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e risolverla con il Master Theorem.

$$T(n) = T\left(\frac{n}{2}\right) + c \quad \to \Theta(\log(n))$$

$$T(n) \le d(n) \qquad (\Omega)$$

$$d\left(\frac{n}{2}\right) + c \le d(n)$$

$$\frac{dn}{2} + c \le dn$$

$$\frac{dn}{2} - dn \le -c$$

$$\frac{dn - 2dn}{\cancel{2}} \le -2c$$

$$-dn \le -2c$$

$$d \ge 2c$$

$$\forall n \ge 0 \quad c, d > 0$$

DUALE PER O

Soluzione: La prova di correttezza avviene per induzione sulla lunghezza $l = r - p + 1$ del sottoarray $A[p..r]$ di interesse. Se $l = 0$, ovvero $p > r$, la funzione ritorna correttamente 0, dato che non ci sono elementi nel sottoarray e quindi non ci possono essere elementi $= k$. Se invece $l > 0$ si calcola $q = \lfloor (p + r)/2 \rfloor$ e si distinguono tre casi:

- se $A[q] = k$ si ritorna correttamente k
- se $A[q] < k$, dato che l'array è ordinato certamente $A[j] \leq A[q] < k$ per $p \leq j \leq q$. Quindi il valore k può trovarsi solo nel sottoarray $A[q + 1, r]$. La chiamata $search(A, q + 1, r, k)$, dato che il sottoarray ha lunghezza minore di l, per ipotesi induttiva restituisce un indice i tale che $q + 1 \leq i \leq r$ e $A[i] = k$, se esiste, e altrimenti restituisce 0. Per l'osservazione precedente, questo è il valore corretto da restituire anche per $search(A, p, r, k)$ e si conclude.
- se $A[q] > k$ si ragiona in modo duale rispetto al caso precedente.

Per quanto riguarda la ricorrenza, ignorando i casi base, dato che la funzione ricorre su di un array la cui dimensione è la metà di quello originale, si ottiene:

$$T(n) = T(n/2) + c$$

Rispetto allo schema standard del master theorem ho $a = 1$, $b = 2$ e $f(n) = c$. Ho dunque che $f(n) = 1 = \Theta(n^{\log_2 1} = n^0 = 1$. Pertanto si conclude che $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Esercizio 1 (9 punti) Realizzare una funzione `intersect(A1,A2,n)` che dati due array di interi `A1` e `A2`, organizzati a min-heap, con capacità n, restituisce un nuovo array `A`, ancora organizzato a min-heap, che contiene l'intersezione dei valori contenuti in `A1` e `A2`. Nel caso gli array contengano più occorrenze dello stesso valore v, l'intersezione mantiene il numero minimo di occorrenze di v (ad es. se `A1` contiene i valori `1,2,2,2` e `A2` contiene i valori `1,1,2,2` allora A conterrà `1,2,2`). Valutarne la complessità.

```
Intersect(A_1, A_2, n)

while(A_1.heapsize > 0) and (A_2.heapsize > 0)

if(Min(A_1) == Min(A_2)
        A[i] = ExtractMin(A_1)
        i++

if(Min(A_1) < Min(A_2))
        A[i] = ExtractMin(A_1)
        A[i+1] = A_2[i]

if(Min(A_2) < Min(A_1))
        A[i] = ExtractMin(A_2)
        A[i+1] = A_1[i]
```

```
intersect(A1,A2,n)
   allocate A[1..n]
   i=0                        // ultimo elemento occupato in A
   while (A1.heapsize > 0) and (A2.heapsize > 0)
         v1 = Min(A1)
         v2 = Min(A2)
         if (v1 = v2)
            i++
            A[i]=1
            ExtractMin(A1)
            ExtractMin(A2)
         elseif (v1 < v2)
            ExtractMin(A1)
         else
            ExtractMin(A2)

   A.heapsize=i
   return A
```

PRENDE IL MINIMO E FA LA HEAPIFY

Esercizio 2 (9 punti) Data una stringa $X = x_1, x_2, \ldots, x_n$, si consideri la seguente quantità $\ell(i, j)$, definita per $1 \leq i \leq j \leq n$:

$$\ell(i,j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } i = j - 1 \\ 2 + \ell(i+1, j-1) & \text{se } (i < j - 1) \text{ e } (x_i = x_j) \\ \sum_{k=i}^{j-1} (\ell(i,k) + \ell(k+1, j)) & \text{se } (i < j - 1) \text{ e } (x_i \neq x_j). \end{cases}$$

1. Scrivere una coppia di algoritmi INIT_L(X) e REC_L(X, i, j) per il calcolo memoizzato di $\ell(1, n)$.
2. Si determini la complessità *al caso migliore* $T_{\text{best}}(n)$, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

```
INIT_L(X)

n = length(X)
if(n == 1) return 1
if(n == 2) return 2

for (i = 1 to n - 1)
        L[i, i + 1] = 2
        L[i, i] = 1
L[n, n] = 1

for i = 1 to n - 2
        for j = i + 2 to n
                L[i, j] = 0

return REC_L(X, 1, n)

REC_L(X, i, j)

if(L[i,j] == 0)
        if(X[i] == X[j]) L[i,j] = 2 + REC_L(X, i+1, j-1)
        else
        for k = i to j-1
                L[i,j] = REC_L(X, i, k) + REC_L(X, k+1, j)
```

1. Pseudocodice:

```
INIT_L(X)
 n <- length(X)
 if n = 1 then return 1
 if n = 2 then return 2
 for i=1 to n-1 do
   L[i,i] <- 1
   L[i,i+1] <- 2
 L[n,n] <- 1
 for i=1 to n-2 do
   for j=i+2 to n do
     L[i,j] <- 0
 return REC_L(X,1,n)


REC_L(X,i,j)
 if L[i,j] = 0 then
   if x_i = x_j then L[i,j] <- 2 + REC_L(X,i+1,j-1)
   else for k=i to j-1 do
     L[i,j] <- L[i,j] + REC_L(X,i,k) + REC_L(X,k+1,j)
 return L[i,j]
```

$\lfloor n/2 \rfloor$ $\quad \Omega(n)$

$\lfloor n/2 \rfloor$

2. Il caso migliore è chiaramente quello in cui tutti i caratteri sono uguali, poiché l'albero della ricorsione in quel caso è unario e i suoi nodi interni corrispondono alle chiamate con indici $(1, n), (2, n-1), \ldots, (k, n-k+1)$, ognuno associato a un costo unitario. Ci sono al più $\lfloor n/2 \rfloor$ di queste chiamate, e quindi $T_{\text{best}}(n) = O(n)$.

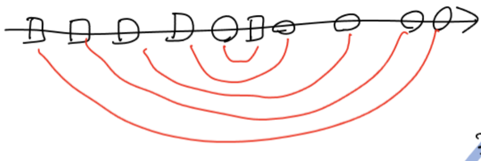
Esercizio: metric matching sulla linea

Sia $S = \{s_1, s_2, \ldots, s_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei server. Sia $C = \{c_1, c_2, \ldots, c_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei client. Il costo di assegnare un client c_i ad un server s_j è $|c_i - s_j|$. Si fornisca un algoritmo greedy che assegna ogni client ad un server distinto e che minimizzi il costo totale dell'assegnamento.

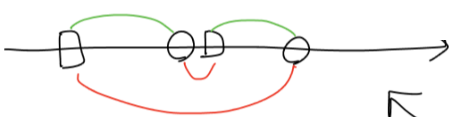
$n = 5$

1) $c_1 - s_1$, $c_2 - s_2$, ...

2)



3) trova la coppia con distanza minore



Esercizio: Sia $X = \{x_1, x_2, \dots, x_n\}$ un insieme di punti ordinato non decrescente sulla retta reale. Si fornisca un algoritmo greedy che determini un insieme I di cardinalità minima di intervalli chiusi di ampiezza unitaria ($[a,b] \in I \Rightarrow b-a=1$) tale che $\forall x_i \in X$ $\exists\, i \in I$ tale che $x_i \in i$.

$C = \{x_1, x_1 + 1\}$

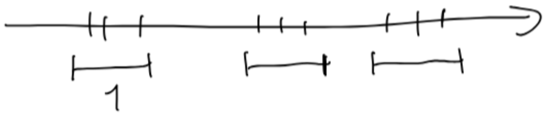
LAST = 1

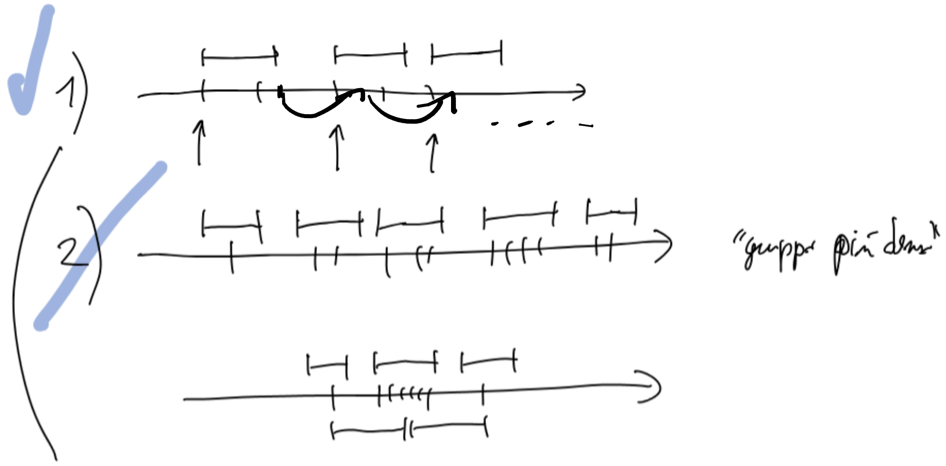
$[0, 1 \mid 1, 1] \rightarrow [0.2 \mid 1.2]$

```
MIN_COVER(X)
  n ← length(X)
  C ← {[x1, x1+1]}
  last ← 1
  for i = 2 to n do
      if (xi > x_last + 1) then
          last ← i
          C ← {[xi, xi+1]} ∪ C
  return C
```

Ci-SS

(METRIC MATCHING)





Esercizio 2 (9 punti) Lungo una strada ci sono, in vari punti, n parcheggi liberi e n auto. Un posteggiatore ha il compito di parcheggiare tutte le auto, e lo vuole fare minimizzando lo spostamento totale da fare. Formalmente, dati n valori reali $p_1, p_2, \ldots, p_n$ e altri n valori reali $a_1, a_2, \ldots, a_n$, che rappresentano

le posizioni lungo la strada rispettivamente di parcheggi e auto, si richiede di assegnare ad ogni auto a_i un parcheggio $p_{h(i)}$ minimizzando la quantità

$$\sum_{i=1}^{n} |a_i - p_{h(i)}|.$$

L'idea di questo algoritmo, se fosse in codice, sarebbe simile a Metric Matching, quindi cerco di minimizzare la differenza partendo dalla fine (quindi, vedendo quante auto e quanti parcheggi ci sono) e verificando quale, a coppie, ha la minima differenza.

1. Si consideri il seguente algoritmo greedy. Si individui la coppia (auto, parcheggio) con la minima differenza. Si assegni quell'auto a quel parcheggio. Si ripeta con le auto e i parcheggi restanti fino a quando tutte le auto sono parcheggiate. Dimostrare che questo algoritmo non è corretto, esibendo un controesempio.

2. Si consideri il seguente algoritmo greedy. Si assuma che i valori $p_1, p_2, \ldots, p_n$ e $a_1, a_2, \ldots, a_n$ siano ordinati in modo non decrescente. Si produca l'assegnazione $(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)$. Dimostrare la correttezza di questo algoritmo per il caso $n = 2$.

Questo significa che l'assegnazione parte dagli ultimi parcheggi e dalle ultime auto (quindi, massima somma al contrario significa minima differenza).

$\{a_1, p_1\} \mid \{a_2, p_2\}]$ ...

$\left[\begin{array}{l} p_2/a_1 = 10 - 9 = 1 \\ p_1/a_2 = 14 - 5 = 9 \end{array}\right] \to 10$

1. Si consideri il seguente input:

$$p_1 = 5, p_2 = 10 \quad \text{e} \quad a_1 = 9, a_2 = 14.$$

L'algoritmo produce l'assegnazione $(a_1, p_2), (a_2, p_1)$, che ha costo $1 + 9 = 10$, mentre l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $4 + 4 = 8$.

$\left[\begin{array}{l} p_1/a_1 = 9 - 5 = 4 \\ p_2/a_2 = 14 - 10 = 4 \end{array}\right] \to 8$

Se P non ordinati, allora fallisce...

2. Ci sono vari casi possibili:

Dal ragionamento detto, matematicamente, si vede che basta prendere un qualsiasi ordinamento tra le due auto e i due parcheggi di due generiche differenze e si esprime la somma in termini matematici (l'idea concreta è quella spiegata da me).

(a) Caso $a_1 \le p_1 \le p_2 \le a_2$
- l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + a_2 - p_2 = (a_2 - a_1) - (p_2 - p_1)$
- l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (a_2 - a_1) + (p_2 - p_1)$; siccome $p_2 - p_1 \ge 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

(b) Caso $a_1 \le p_1 \le a_2 \le p_2$
- l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) - (a_2 - p_1)$
- l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (p_2 - a_1) + (a_2 - p_1)$; siccome $a_2 - p_1 \ge 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

(c) Caso $a_1 \le a_2 \le p_1 \le p_2$
- l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) + (p_1 - a_2)$
- l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + p_1 - a_2 = (p_2 - a_1) + (p_1 - a_2)$, uguale a quello precedente

Tutti gli altri casi sono simmetrici e si dimostrano nella stessa maniera.