**Algoritmi e Strutture Dati**
**8 Giugno 2018**

**Cognome** ..... **Nome** ..... **Matricola** .....

*Note*

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea soggiacente e motivarne la correttezza.
3. L'efficienza è un criterio di valutazione delle soluzioni proposte.
4. Si consegnano tutti i fogli, con nome, cognome, matricola e l'indicazione *bella copia* o *brutta copia*.

# Domande

**Domanda A** (5 punti) Data la ricorrenza $T(n) = T(n/2) + T(n/3) + \sqrt{n} + 2$ dimostrare che ha soluzione $T(n) = O(n)$. Il limite è stretto, ovvero vale anche $T(n) = \Omega(n)$? [Questa seconda parte della domanda, per un errore nella scelta dei coefficienti risultava più complessa di quanto preventivato, quindi non è stata considerata nella valutazione. Allego comunque la soluzione per completezza.]

**Soluzione:** Per dimostrare che $T(n) = O(n)$ occorre trovare $c > 0$ e $n_0 \in \mathbb{N}$ tale che per ogni $n \geq n_0$, $T(n) \leq cn$. Procediamo con un ragionamento induttivo

$$\begin{aligned}
T(n) &= T(n/2) + T(n/3) + \sqrt{n} + 2 \\
&\leq cn/2 + cn/3 + \sqrt{n} + 2 \\
&= cn/2 + cn/3 + \sqrt{n} + 2 \\
&= 5/6cn + \sqrt{n} + 2 \\
&= 5/6cn + n + 2 \\
&\leq cn
\end{aligned}$$

vale se $1/6cn \geq n + 2\text{m}$, quindi, ad esempio, per $c \geq 7$ e $n \geq 12$.

Il limite non è stretto. Un tentativo di dimostrazione induttiva che $T(n) \geq dn$ per qualche $d > 0$ fallisce. Tuttavia questo non consente di concludere.

Si può invece dimostrare che $T(n) = O(n^\alpha)$ con $\alpha = 5/6$. Infatti, ancora in modo induttivo:

$$\begin{aligned}
T(n) &= T(n/2) + T(n/3) + \sqrt{n} + 2 \\
&\leq cn/2 + cn/3 + \sqrt{n} + 2 \\
&= c(n/2)^\alpha + c(n/3)^\alpha + \sqrt{n} + 2 \quad [\text{dato che } \sqrt{n} + 2 \leq n^\alpha \text{ per } n \geq 5] \\
&\leq cn^\alpha((1/2)^\alpha + (1/3)^\alpha) + n^\alpha \\
&\leq cn^\alpha
\end{aligned}$$

vale se $c(1 - (1/2)^\alpha - (1/3)^\alpha)n^\alpha \geq n^\alpha$, quindi, osservando che la quantità $1 - (1/2)^\alpha - (1/3)^\alpha > 0$, è sufficiente scegliere $c \geq 1/(1 - (1/2)^\alpha - (1/3)^\alpha)$ e $n \geq 5$.

Questo prova che $T(n) = O(n^\alpha)$ e quindi non può essere $T(n) = \Omega(n)$.

**Domanda B** (4 punti) Fornire lo pseudocodice della funzione `Prec(T,x)` che dato un albero binario di ricerca `T` e un suo nodo `x`, ritorna il predecessore di `x` (oppure `nil` se non c'è).

**Soluzione:** Si veda il libro.

**Domanda C** (5 punti) Scrivere una funzione `diff(T)` che dato in input un albero binario di ricerca `T` determina la massima differenza di lunghezza tra due cammini che vanno dalla radice ad un sottoalbero vuoto. Ad esempio sull'albero ottenuto inserendo 1, 2 e 3 produce 2, su quello ottenuto inserendo 2, 1, 3 produce 0. Valutarne la complessità.

**Soluzione:** Il programma si basa su una una funzione ricorsiva `diff-rec(x)` che restituisce il la lunghezza del minimo e massimo cammino per il sottoalbero radicato in $x$

```
diff(T)
   min,max = diffRec(T.root)
   return

diff-rec(x)
   if x = nil
      return 0,0
   else
      minl, maxl = diff-rec(x.left)
      minr, maxr = diff-rec(x.right)
      return min(minl,minr)+1, max(maxl,maxr)+1
```

Per quanto riguarda la complessità si osservi che si tratta di una visita dell'albero, quindi $\Theta(n)$.

# Esercizi

**Esercizio 1** (7 punti) Si dica che un array `A[1..n]` è bi-ordinato se esiste un indice `k` tale che `A[1..k]` è ordinato in senso crescente e `A[k..n]` è ordinato in senso decrescente. Realizzare un algoritmo `top(A,n)` che dato in input un array `A` bi-ordinato con elementi distinti ne determina il valore massimo. Ad esempio su `A = (1 2 3 14 13 4)` restituisce `14`. Valutarne la complessità.

**Soluzione:**

```
top(A,n)
   top-rec(A,i,j)

top_rec(A,i,j)
   if (i=j)
      return A[i]
   else
      q = (i+j)/2
      if A[q] <= A[q+1]
         return top_rec(q+1,j)
      else
         return top_rec(i,q)
```

**Esercizio 2** (9 punti) Nello stato di Frattalia, si è appena insediato il nuovo governo, sostenuto da due partiti, Arancio e Cobalto. La coalizione Arancio-Cobalto ha sottoscritto un contratto di governo che prevede diversi punti, alcuni proposti dagli Arancio altri dai Cobalto. Tuttavia ogni punto ha un costo economico, così che non è possibile realizzarli tutti: il debito pubblico non può oltrepassare una certa soglia. Occorre quindi scegliere un sottoinsieme di punti da realizzare e tale

insieme deve essere *bilanciato*, ovvero devono essere scelti in egual numero punti degli Arancio e dei Cobalto.

Si supponga che i punti proposti dal partito Arancio siano `n` con costi $a_1, \ldots, a_n$ e analogamente i punti proposti dai Cobalto siano `m` con costi $c_1, \ldots, c_m$. Fornire un algoritmo di programmazione dinamica che individui un sottoinsieme bilanciato dei punti del contratto, di dimensione massima e tale che la somma dei costi non oltrepassi un limite fissato $d$. Più precisamente:

i. dare una caratterizzazione ricorsiva della cardinalità massima $p_{i,j,d}$ di un insieme bilanciato di punti di programma scelti in $a_1, \ldots, a_i$, $c_1, \ldots, c_j$ con costo $\leq d$;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina il numero massimo;

iii. trasformare l'algoritmo in modo che fornisca anche i punti del contratto scelti, non solo il loro numero;

iv. valutare la complessità dell'algoritmo.

Esiste la possibilità di applicare un algoritmo greedy?

**Soluzione:** La caratterizzazione ricorsiva della cardinalità massima $p[i, j, m]$ di un insieme bilanciato di punti di programma scelti in $a_1, \ldots, a_n$, $c_1, \ldots, c_m$ con costo $\leq d$ deriva dall'osservazione che

- se $i = 0$ oppure $j = 0$, l'unico insieme bilanciato è vuoto e quindi il numero totale sarà 0.;
- altrimenti, se $i, j > 0$, possiamo escludere il punto $a_i$, oppure escludere il punto $c_j$, oppure, se il budget lo consente, includere sia $a_i$ che $c_j$

In sintesi:

$$p[i,j,m] = \begin{cases} 0 & \text{if } i = 0 \text{ oppure } j = 0 \\ \max\{p[i-1,j,d],\ p[i,j-1,d]\} & \text{if } i,j > 0 \text{ e } a_i + c_j > d \\ \max \left\{ \begin{array}{l} p[i-1,j,d],\ p[i,j-1,d], \\ p[i-1,j-1,d-a_i-c_j]+2 \end{array} \right\} & \text{if } i,j > 0 \text{ e } a_i + c_j \leq m \end{cases}$$

Ne segue l'algoritmo che riceve in input gli array di punti $a[1, n]$ e $c[1, m]$, il costo massimo $d$ e usa una matrice $P[1..n, 1..m, 0..d]$ dove $P[i, j, d']$ rappresenta il numero massimo bilanciato di punti scelti in $a_1, \ldots, a_i$, $c_1, \ldots, c_m$ con costo $\leq d'$.

```
governo (a, c, n, m, d)
  for i=1 to n
    for d' = 0 to d
    p[i,0, d'] = 0

  for j=0 to m
    for d' = 0 to d
    p[0,j, d'] = 0

  for i=1 to n
    for j = 1 to m
        for d'=1 to to d
           if p[i-1,j,d'] >= p[i,j-1,d']
              max = p[i-1,j,d']
           else
              max = p[i,j-1,d']
```

```
          if a[i]+c[j] <= d' and p[i-1,j-1,d'-a[i]-c[j]] + 2 > max
             max  = p[i-1,j-1,d'-a[i]-c[j]] + 2

          p[i,j,d']=max

   return p[n,m,d]
```

Se vogliamo anche i punti scelti, occorre ricordare dove il massimo viene raggiunto. Dato che i punti sono scelti a coppie usiamo un array S[i,j,d'] = (i',j') che da' la prima coppia di punti scelta in una soluzione ottima (o (0,0) se non ne viene scelto nessuno)

```
governo (a, c, n, m, d)
  # i punti sono scelti a coppie
  # usa quindi S[i,j,d'] = (i',j') prima coppia di punti scelta
  # in una soluzione ottima, (0,0) se non e' possibile nessuna scelta

  for i=1 to n
    for d' = 0 to d
    p[i,0, d'] = 0
    S[i,0, d'] = (0,0)

  for j=0 to m
    for d' = 0 to d
    p[0,j, d'] = 0
    S[0,j, d'] = (0,0)


  for i=1 to n
    for j = 1 to m
        for d'=1 to to d
           if p[i-1,j,d'] >= p[i,j-1,d']
              max = p[i-1,j,d']
              S[i,j,d'] = S[i-1,j,d']
           else
              max = p[i,j-1,d']
              S[i,j,d'] = S[i,j-1,d']

           if a[i]+c[j] <= d' and p[i-1,j-1,d'-a[i]-c[j]] + 2  > max
              max  = p[i-1,j-1,d'-a[i]-c[j]] + 2
              S[i,j,d'] = (i,j)

           p[i,j,d']=max


   i=n, j=m, d'=d
   while (i>0 and j>0)
         i, j = S[i,j,d']

         if (i>0)
             print "a[", i, "], "c[", j, "]"
             d' = d' - a[i] - c[j]

        i--; j--
```

La complessità è $\Theta(nmd)$.

È facile vedere che si può utilizzare un algoritmo greedy che sceglie ad ogni passo coppie di punti di costo minimo. In primo luogo, si osserva, come fatto sopra, che vale la proprietà della sosttruttura ottima, ovvero una soluzione ottima per il problema consiste in una scelta di una coppia di punti, uno Arancio e uno Cobalto, diciamo $a_i$ e $c_j$ e un ottimo del sottoproblema con punti $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ e $b_1, \ldots, b_{j-1}, a_{j+1}, \ldots, b_m$ con limite di spesa $d - a_i - b_j$.

Inoltre, vale la scelta greedy che cosiste nello scegliere i punti Arancio e Cobalto di costo minimo (se la somma è inferiore a $d$), altrimenti, nessun punto.

Ovvero, assumendo che gli array siano ordinati in modo decrescente

$$p[i,j,m] = \begin{cases} 0 & \text{if } i = 0 \text{ oppure } j = 0 \text{ oppure } a_i + c_j > d \\ p[i-1, j-1, d - a_i - c_j] + 2 & \text{altrimenti} \end{cases}$$

Questo permette di usare l'algoritmo che ordina le gli array dei punti per costo crescente, con complessità $n \log n + m \log m$. Quindi si costruisce la soluzione prendendo, ad ogni iterazione, i punti di costo minimo, finchè possibile, in tempo $O(\min(m,n))$. La complessità è dunque $\Theta(n \log n + m \log m)$, quasi invariabilmente preferibile a $\Theta(nmd)$.