Algoritmi e Strutture Dati - 04/06/2018

Esercizio 1

L'algoritmo può essere risolto banalmente con un approccio greedy. E' sufficiente negare i k valori più bassi:

```
negate(int[] X, int n, int k)
    sort(X)
    int tot = 0
    for i = 1 to n do
        tot = tot + iif(i ≤ k, −X[i], X[i])
    return tot
```

L'algoritmo proposto ha complessità $O(n \log n)$ per l'ordinamento, $O(n)$ se il vettore è già ordinato.
La correttezza dell'algoritmo può essere dimostrata nel modo seguente: si consideri l'insieme di indici S che identificano i k valori più bassi. Si consideri un insieme di indici M che formano una negazione k-massimale. Se $S = M$, la nostra soluzione è k-massimale. Supponiamo per assurdo che S non sia k-massimale.

- Sia $s \in S - M$ un indice presente in S ma non in M
- Sia $m \in M - S$ un indice presente in M ma non in S
- Entrambi questi indici esistono, perchè $S \neq M$ e $|S| = |M| = k$
- Ovviamente, $s \neq m$ per come sono estratti dagli insiemi
- $X[m] > X[s]$, perchè s è uno degli indici contenenti i k valori più bassi e tutti i valori sono distinti
- Sia $M' = M - \{m\} \cup \{s\}$; allora,

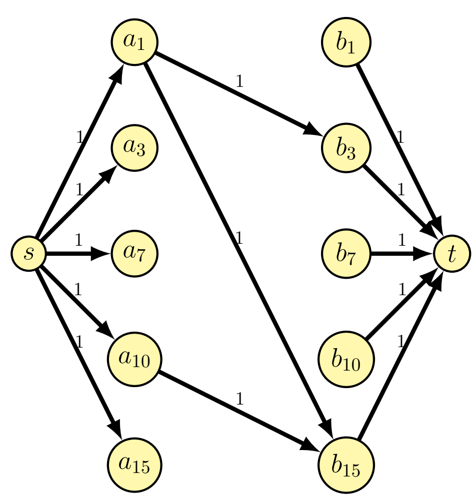
$$\sum_{i=1}^{n} \overline{X}_{M'}[i] = \sum_{i=1}^{n} \overline{X}_M[i] + 2X[m] - 2X[s] > \sum_{i=1}^{n} \overline{X}_M[i]$$

  assurdo in quanto M è k-massimale. Il fattore due deriva dal fatto che si passa da $-X[m]$ a $+X[m]$ e da $+X[s]$ a $-X[S]$.

Esercizio 2

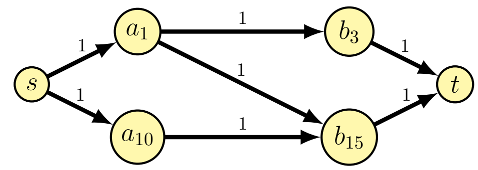
Il problema può essere risolto, fra l'altro, tramite una rete di flusso, così costruita:

- Si crea un nodo sorgente s e un nodo pozzo t
- Per ogni elemento di X, si crea una coppia di nodi, $a_{x[i]}$ e $b_{x[i]}$
- Si crea un arco orientato dalla sorgente s ad ogni nodo a_i
- Si crea un arco orientato da ogni nodo b_i al pozzo t
- Dati due elementi a_i e b_j, si crea un arco da a_i a b_j se e solo se $X[i] < X[j]$ e $X[i], X[j]$ sono quadrabili.
- Tutti gli archi hanno capacità 1



L'idea è la seguente: i nodi corrispondenti a due valori distinti sono collegati da un arco se e solo se il primo è più piccolo del secondo (per evitare ripetizioni) e se sono quadrabili. In questo modo, il valore flusso massimo $|f^*|$ sarà pari al numero massimo di coppie quadrabili, senza ripetizioni. Gli archi fra nodi a_i e b_i con flusso uguale a 1 rappresentano le coppie inquadrabili selezionate.
Il numero di nodi è pari a $|V| = 2n+2$; il numero di archi $|E|$ è limitato superiormente da $O(n^2)$. Il valore flusso è limitato superiormente da $n/2$ (numero massimo di coppie). Per il limite di Ford-Fulkerson, la complessità è $O(|f^*|(|V| + |E|))$, pari a $O(n^3)$. Esiste un algoritmo (Hopcroft-Karp), non visto a lezione, che lavora in tempo $O(n^2\sqrt{n})$ in questo caso.

Si noti inoltre che tutti i nodi a_i che non hanno archi uscenti e tutti i nodi b_i che non hanno archi entranti possono essere rimossi, riducendo di molto la complessità.



Soluzione errata - greedy Alcune persone hanno utilizzato un approccio greedy, che però non funziona.

```
squarable(int[] X, int n)
    int k = 0
    boolean[] used = new boolean[1...n]
    for i = 1 to n do
        used[i] = false
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if not used[i] and not used[j] and ⌊√(X[i] + X[j])⌋ == √(X[i] + X[j]) then
                used[i] = true
                used[j] = true
                k = k + 1
    return k
```

Per esempio, $X = [1, 15, 3, 10]$ "consuma" 1+15 subito e restituisce 1; mentre si può restituire 2 ($1 + 3$, $15 + 10$).

Soluzione errata - programmazione dinamica Sia $DP[i][j]$ il massimo numero di coppie contenute nell'intervallo $DP[i \dots j]$. E' stata proposta la seguente formula ricorsiva per la programmazione dinamica:

$$DP[i][j] = \begin{cases} 0 & i \geq j \\ \max(1 + DP[i+1][j-1], DP[i+1][j], DP[i][j-1]) & i < j \wedge \left\lfloor \sqrt{X[i] + X[j]} \right\rfloor = \sqrt{X[i] + X[j]} \\ \max(DP[i+1][j], DP[i][j-1]) & \text{altrimenti} \end{cases}$$

Il problema iniziale è dato da $DP[1][n]$

E' facile vedere che non funziona con $[1, 3, 6, 19]$. Infatti:

$$\begin{aligned} DP[1][4] &= \max(DP[2][4], DP[1][3]) = 1 \\ DP[1][3] &= \max(DP[1][2], DP[2][3]) = 1 \\ DP[2][4] &= \max(DP[2][3], DP[3][4]) = 1 \\ DP[1][2] &= 1 \\ DP[2][3] &= 1 \\ DP[3][4] &= 1 \end{aligned}$$

mentre la risposta è 2 ($1 + 3$, $6 + 19$).

Il problema è che i sottoproblemi non sono indipendenti, e quindi non è possibile dimostrare la sottostruttura ottima.

Esercizio 3

Utilizziamo la tecnica di backtrack, enumerando tutti i possibili sottoinsiemi.
La funzione ricorsiva printSums() prende in input il vettore X e la sua dimensione; prende inoltre in input il valore target v, che viene decrementato tutte le volte che si sceglie un valore. Per realizzare il backtrack, si passa anche una maschera di booleani (valori 0/1) che rappresenta l'insieme di valori scelti. Il parametro i invece rappresenta l'attuale elemento del vettore i che viene considerato, analizzato dall'ultimo al primo. La chiamata iniziale è printSums$(X, n, v, 0, S)$.
Per ogni elemento $X[i]$, ci sono due possibilità: lo utilizziamo oppure no in una possibile somma. Questo è rappresentato dalla linea **foreach** $c \in \{0, 1\}$. Memorizziamo la scelta in $S[i]$ e chiamiamo printSums() ricorsivamente, modificando v in maniera opportuna.

Questo ci porta ad esplorare un albero di decisione di dimensione 2^n. In generale, nel caso pessimo ogni volta che si realizza la somma vengono stampati fino ad n valori, quindi la complessità è $O(n2^n)$.

```
printSums(int[] X, int n, int v, int i, int[] S)
  if i == n + 1 then
      if v == 0 then
          printVector(X, S, n)
      return
  foreach c ∈ {0, 1} do
      S[i] = c
      printSums(X, n, v − c · X[i], S, i + 1)
```

La funzione printVector() stampa i valori; questa è una versione abbastanza sofisticata, che stampa anche i segni + (tranne l'ultimo), ma nel compito non è necessario andare così nei dettagli.

```
printVector(int[] X, int[] S, int n)
  int i = 1
  boolean first = false
  while i < n do
      if S[i] == 1 then
          print X[i]
          if not first then
              print " + "
              first = true
      println
```

Una possibile funzione in Python per printVector() (ma sono sicuro che ci saranno modi ancora più compatti per fare una cosa del genere) è la seguente:

```python
def printVector(X, S):
      L = [ x for i,x in enumerate(X) if S[i]==1 ]
      print(*L, sep = " + ")
```

Soluzione errata Un possibile errore, molto comune, è quello di separare il controllo su quando il valore v raggiunge il valore 0 dal controllo su quando ho esaurito gli elementi da scegliere ($i = n + 1$). Se scritti separatamente, la stessa stringa verrà stampata più volte, tutte le volte che i viene incrementato e viene scelto un valore 0.

```
printSums(int[] X, int n, int v, int i, int[] S)
  if i == n + 1 then
      return
  if v == 0 then
      printVector(X, S, n)
  foreach c ∈ {0, 1} do
      S[i] = c
      printSums(X, n, v − c · X[i], S, i + 1)
```

Esercizio 4

Il problema è una variante del problema subset-sum e può essere risolto con programmazione dinamica.
Sia $DP[i][s][v]$ il massimo valore ottenibile utilizzando i primi i elementi, scegliendo al massimo s valori, non dovendo superare v.

$$DP[i][s][v] = \begin{cases} 0 & v = 0 \ \vee i = 0 \ \vee s = 0) \\ DP[i-1][s][v] & X[i] > v > 0 \wedge i > 0 \ \wedge s > 0 \\ \max\{DP[i-1][s][v], DP[i-1][s-1][v - X[i]] + X[i]\} & \text{altrimenti} \end{cases}$$

L'idea è la seguente:

- Il valore massimo che si può ottenere se $v = 0$ (non deve superare il valore 0), se $i = 0$ (ho finito gli elementi da cui scegliere), se $s = 0$ (ho finito le scelte da fare) è ovviamente 0.
- Altrimenti, se $X[i] > v$, allora non è selezionabile; quindi l'unica possibilità è ignorarlo, riducendo il numero di elementi $(i-1)$ e lasciando intatti s e v.
- Altrimenti, ci sono due possibilità: l'elemento viene preso oppure no. Nel caso venga preso, bisogna ridurre il numero di elementi prendibili $(s-1)$ e il valore di v $(v - X[n])$, ma bisogna sommare il valore $+X[i]$.

Questo può essere tradotto tramite memoization:

```
kwConstraint(int[] X, int n, int k, int w)
  int[][][] DP = new int[0...n][0...k][0...w]
  for i = 0 to n do
    for s = 0 to k do
      for v = 0 to w do
        DP[i][s][v] = −1
  kwConstraintRec(X, n, k, w, DP)
  return DP[n][k]
```

```
kwConstraintRec(int[] X, int i, int s, int v, int[][][] DP)
  if i == 0 or s == 0 or v == 0 then
    return 0
  if DP[i][s][v] < 0 then
    if X[i] > w then
      DP[i][s][v] = kwConstraintRec(X, n − 1, k, w, DP)
    else
      DP[i][s][v] = max ( { kwConstraintRec(X, i − 1, s, v, DP),
                            kwConstraintRec(X, i − 1, s − 1, v − X[i], DP) + X[i] )
  return DP[i][s][v]
```

La complessità è $O(nkw)$; poichè $k < n$, la complessità può anche essere letta come $O(n^2w)$.