Algoritmi e Strutture Dati - 05/06/14

Esercizio 1

Questo problema è molto simile al problema dell'insieme indipendente di intervalli pesati visto a lezione, dove il peso $w[i]$ è pari a $b[i] - a[i]$. Si risolve quindi con una singola chiamata a quella soluzione, in un tempo pari a $\Theta(n \log n)$.

```
SET segmentcover(int[] a, int[] b, int n)
  int[] w = new int[1...n]
  for i = 1 to n do
    w[i] = b[i] - a[i]
  return maxinterval(a, b, w, n)
```

Esercizio 2

L'algoritmo opera ricorsivamente, utilizzando l'approccio divide-et-impera: si verifica che i nodi che si stanno analizzando contengano lo stesso valore (oppure che siano entrambi **nil**) e si richiama la funzione sul sottoalbero sinistro e destro, restituendo **false** nel caso uno di queste verifiche diano risultato negativo. Il costo è ovviamente $O(n)$.

```
boolean equal(TREE T, TREE S)
  if S == nil and T == nil then
    return true
  if S == nil or T == nil then
    return false
  if S.key ≠ T.key then
    return false
  return equal(S.left, T.left) and equal(S.right, T.right)
```

Esercizio 3

Per risolvere un problema di questo tipo, è necessario utilizzare un algoritmo di tipo backtrack. La procedura colora() prende in input il grafo G, l'intero k, il vettore delle scelte S, l'indice della scelta da effettuare u (che rappresenta anche l'identificatore di un nodo). Nelle prime righe della funzione, viene calcolato l'insieme C dei colori disponibili per l colorazione. Per ogni $c \in C$, l'u-esimo nodo del grafo viene colorato di c; se tutti i nodi sono stati colorati, viene stampata tale colorazione dalla printSolution(). Altrimenti, si continua ricorsivamente incrementando u.
Nel caso pessimo, la funzione richiede $O(k^n)$ chiamate ricorsive; ogni chiamata costa $O(n + k)$ (derivanti dal calcolo dell'insieme C). Il

costo complessivo è quindi $O((n+k)k^n)$.

```
boolean coloring(GRAPH G, int k, int[] S, int u)
    SET C = {1, ..., k}
    foreach v ∈ u.adj() do
        if S[v] ≠ 0 then
            C.remove(S[v])
    foreach c ∈ C do
        S[u] = c
        if u == G.n then
            printSolution(S, G.n)
            return true
        else
            if coloring(G, k, S, u + 1) then
                return true
        S[u] = 0
    return false
```

Esercizio 4

Le soluzioni proposte nei casi $k = 2$ e $k = 3$ servono a prendere confidenza con il problema, ma per risolverlo in modo efficiente nel caso generale bisogna fare un passo in più.
In generale, una soluzione banale a tutti i problemi con k deciso a priori è fatta in questo modo:

```
int partitionK(int[] V, int n)
    int minSoFar = +∞
    for i_1 = 1 to n do
        for i_2 = i_1 + 1 to n do
            for i_3 = i_2 + 1 to n do
                for ... = ... + 1 to n do
                    for i_{k-1} = i_{k-2} + 1 to n do
                        tot_1 = sum(V, 1, i_1)
                        tot_2 = sum(V, i_1 + 1, i_2)
                        tot_3 = sum(V, i_2 + 1, i_3)
                        ...
                        tot_k = sum(V, i_{k-1} + 1, n)
                        minSoFar = min(minSoFar, max(tot_1, ..., tot_k))
```

Ci sono $k-1$ cicli annidati di costo $O(n)$; per calcolare i valori tot_j, è necessario sommare tutti i valori con costo $O(n)$. Il costo di questo algoritmo è quindi $O(n^k)$. Ovviamente, si può fare meglio di così.

Caso $k = 2$

Nel caso $k = 2$, si può utilizzare un trucco simile a quanto fatto nel problema di somma massimale visto all'inizio delle lezioni. Calcoliamo, utilizzando una variabile $sumSoFar$, la somma dei primi i valori nel vettore:

$$sumSoFar_i = \sum_{j=1}^{i} V[j]$$

La somma dei restanti $n - i$ valori è data dalla somma totale tot meno $sumSoFar$:

$$tot - sumSoFar_i = \sum_{j=i+1}^{n} V[j]$$

Il valore tot non lo conosciamo, ma possiamo calcolarlo all'inizio sommando tutti i valori.
Essendo composta da due cicli **for** non annidati di costo $\Theta(n)$, il costo della procedura è $\Theta(n)$.

```
int partition2(int[] V, int n)
    int tot = 0
    for i = 1 to n do
        tot = tot + V[i]
    int sumSoFar = 0
    int minSoFar = +∞
    for i = 1 to n − 1 do
        sumSoFar = sumSoFar + V[i]
        minSoFar = min(minSoFar, max(sumSoFar, tot − sumSoFar))
    return minSoFar
```

E' una soluzione di programmazione dinamica? Nel senso che utilizza la somma ottenuta sinora per calcolare la somma successiva, sì. Ma ovviamente è una versione molto semplice della programmazione dinamica.

Caso $k = 3$

Nel caso $k = 3$, possiamo fare scorrere due indici i, j, con $1 \leq i < j < n$, in modo da avere tre sottovettori $V[1 \ldots i]$, $V[i+1 \ldots j]$, $V[j+1 \ldots n]$. Abbiamo bisogno di un meccanismo che ci permetta di ottenere il costo di un sottovettore in tempo $O(1)$; questo meccanismo è stato visto ad esercitazione in aula. Si calcoli preventivamente un vettore di appoggio $tot[0 \ldots n]$ tale per cui $tot[i]$ contiene la somma dei primi i elementi di V:

$$tot[i] = \begin{cases} 0 & i = 0 \\ tot[i-1] + V[i] & i > 0 \end{cases}$$

Il valore del sottovettore $V[a \ldots b]$ è pari a $tot[b] - tot[a-1]$.
Il codice seguente ha quindi costo $O(n^2)$:

```
int partition3(int[] V, int n)
    int[] tot = new int[1 ... n]
    tot[1] = V[1]
    for i = 2 to n do
        tot[i] = tot[i − 1] + V[i]
    int minSoFar = +∞
    for i = 1 to n − 2 do
        for j = i + 1 to n − 1 do
            int temp = max(tot[i], tot[j] − tot[i], tot[n] − tot[j])
            minSoFar = min(minSoFar, temp)
    return minSoFar
```

Il costo di questo algoritmo è $\Theta(n^2)$. Notate che a differenza del codice generico visto all'inizio, abbiamo applicato due semplici ottimizzazioni per gli indici i, j: il loro valore limite è $n-2$, $n-1$. Ovviamente tale ottimizzazione è applicabile anche nel caso del codice generico, ma sarebbe risultata poco chiara.

Caso generico k

Nel caso generale, invece, è possibile utilizzare la programmazione dinamica. Sia $DP[i][t]$ il costo minimo associato al sottoproblema di trovare la migliore t-partizione nel vettore $V[1 \ldots i]$. Il problema iniziale corrisponde a $DP[n][k]$ – ovvero trovare la migliore k-partizione in $V[1 \ldots n]$. Sfruttiamo un vettore di appoggio tot definito come nel caso $k = 3$.

$DP[i][t]$ può essere definito ricorsivamente in questo modo:

$$DP[i][t] = \begin{cases} +\infty & t > i \\ tot[i] & t = 1 \\ \min_{1 \leq j < i} \max(DP[j][t-1], tot[i] - tot[j]) & \text{altrimenti} \end{cases}$$

L'idea è la seguente: si consideri una t-partizione del vettore $V[1 \ldots i]$ e sia $V[j+1 \ldots n]$ l'ultimo sottovettore di essa, con $1 \leq j < i$. È possibile vedere che tale t-partizione come composta da una $(t-1)$-partizione di $V[1 \ldots j]$ e un sottovettore $V[j+1 \ldots n]$ (l'"*ultimo sottovettore*"). Il costo di tale t-partizione è quindi pari al massimo fra il costo della $(t-1)$-partizione e il costo del sottovettore, che è ottenibile in tempo $O(1)$ tramite il vettore di appoggio T grazie all'espressione $tot[i] - tot[j]$ (ovvero, la somma dei primi i valori meno la somma dei primi j valori).
Il problema è che non conosciamo il valore j, ovvero la dimensione dell'ultimo sottovettore; ma possiamo provare tutti i valori compresi fra 1 e i (escluso), e prendere il minimo fra essi.
I casi base sono i seguenti:

- Se $t > i$, significa che cerchiamo di t-partizionare un vettore che ha meno di t elementi; essendo impossibile, restituiamo $+\infty$.
- Se $t = 1$, allora possiamo semplicemente restituire la somma dei primi i valori.

Il codice può essere scritto, tramite memoization, nel modo seguente.

```
int partition(int[] V, int n, int k)
    int[][] DP = new int[1...n][1...k] = {−1}                % Initizialized to −1
    int[] tot = new int[1...n]
    tot[1] = V[1]
    for i = 2 to n do
        tot[i] = tot[i − 1] + V[i]
    return partitionRec(V, tot, DP, n, k)
```

```
int partitionRec(int[] V, int[] tot, int[][] DP, int i, int t)
    if t > i then
        return +∞
    else if t == 1 then
        return tot[i]
    else if DP[i][t] < 0 then
        int DP[i][t] = +∞
        for j = 1 to i − 1 do
            int cost = max(partitionRec(V, tot, DP, j, t − 1), tot[i] − tot[j])
            DP[i][t] = min(DP[i][t], cost)
    return DP[i][t]
```

Questo algoritmo deve riempire una matrice $n \times k$; il costo per riempire ogni elemento della matrice è pari a $O(n)$. La complessità è quindi pari a $O(kn^2)$.

Note

Potete cercare questo problema sotto il nome di "Painter partition problem"; su Google, troverete un sacco di siti che lo presentano come un classico problema da colloquio di lavoro (nelle aziende illuminate).
Notate che è possibile costruire una soluzione di costo $O(n \log T)$, dove T è la somma dei valori del vettore V, con un approccio basato sulla ricerca binaria. Tuttavia, questa soluzione è pseudo-polinomiale nel caso generale.