

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(integer[] a, integer[] b, integer n)
integer[] w = new integer[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 la 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 $O((n + k)k^n)$.

```
colora(GRAPH G, integer k, integer[] S, integer u)
```

```

SET  $C \leftarrow \{1, \dots, k\}$ 
foreach  $v \in u.\text{adj}()$  do
    if  $S[v] \neq 0$  then
         $C.\text{remove}(S[v])$ 

foreach  $c \in C$  do
     $S[u] \leftarrow c$ 
    if  $u = G.n$  then
        printSolution( $S, G.n$ )
        return true
    else
        if colora( $G, k, S, u + 1$ ) then
            return true
     $S[u] \leftarrow 0$ 
return false

```

Esercizio 4

Caso $k = 2$

Nel caso $k = 2$, è possibile progettare un algoritmo che lavora in tempo $\Theta(n)$ calcolando progressivamente la somma parziale V_t dei primi t e calcolando la somma parziale dei restanti $n - t$ elementi come differenza fra V_n (la somma di tutti gli elementi) e V_t . Essendo composta da due cicli **for** di costo n , il costo della procedura è $\Theta(n)$.

```
2-partition(integer[] V, integer n)
```

```

integer tot  $\leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
    tot  $\leftarrow$  tot +  $V[i]$ 
integer sofar  $\leftarrow 0$ 
integer min  $\leftarrow +\infty$ 
for  $i \leftarrow 1$  to  $n - 1$  do
    sofar  $\leftarrow$  sofar +  $V[i]$ 
    min  $\leftarrow \min(\min, \max(\sofar, \text{tot} - \sofar))$ 
return min

```

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 $V[1 \dots i]$, $V[i + 1, j]$, $V[j + 1, n]$. 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 $T[0 \dots n]$ tale per cui $T[i]$ contiene la somma dei primi i elementi di V :

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

Il valore del sottovettore $V[a \dots b]$ è pari a $T[b] - T[a - 1]$.

Il codice seguente ha quindi costo $O(n^2)$:

```
integer 3-partition(integer[] V, integer n)
```

```
integer[][] T ← new integer[0...n]
T[0] ← 0
for i ← 1 to n do
    T[i] ← T[i - 1] + V[i]
integer min ← +∞
for i ← 1 to n - 2 do
    for j ← i + 1 to n - 1 do
        integer temp ← max(T[i], T[j] - T[i], T[n] - T[j])
        min ← min(min, temp)
return min
```

Caso generico k

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

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

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

L'idea è la seguente: si consideri una t -partizione del vettore $V[1 \dots i]$ e sia $V[j + 1 \dots n]$ l'ultimo sottovettore di essa, con $1 \leq j < i$. E' possibile vedere che tale t -partizione come composta da una $(t - 1)$ -partizione di $V[1 \dots j]$ e un sottovettore $V[j + 1 \dots n]$ (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 $T[i] - T[j]$ (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.

```
integer partition(integer[] V, integer n, integer k)
```

```
integer[][] M ← new integer[1...n][1...k] % Inizializzato a ⊥
integer[][] T ← new integer[0...n]
T[0] ← 0
for i ← 1 to n do
    T[i] ← T[i - 1] + V[i]
return partition-rec(V, T, M, n, k)
```

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

```
integer partition-rec(integer[] V, integer[] T, integer[][] M, integer i, integer t)
```

```
if  $t > i$  then return  $+\infty$ 
```

```
if  $t = 1$  then return  $T[i]$ 
```

```
if  $M[i, t] = \perp$  then
```

```
    integer  $M[i, t] \leftarrow +\infty$ 
    for  $j \leftarrow 1$  to  $i - 1$  do
        integer  $temp \leftarrow \max(\text{partition-rec}(V, T, M, j, t - 1), T[i] - T[j])$ 
        if  $temp < M[i, t]$  then  $M[i, t] \leftarrow temp$ 
```

```
return  $M[i, t]$ 
```
