Esercizio. Un'auto può percorrere K Km con un litro di carburante, e il serbatoio ha una capacità di C litri. Tale auto deve percorrere un tragitto lungo il quale si trovano $n+1$ aree di sosta indicate con $0, 1, \ldots n$, con $n \geq 1$. L'area di sosta 0 si trova all'inizio della strada, mentre l'area di sosta n si trova alla fine. Indichiamo con $d[i]$ la distanza in Km tra le aree di sosta i e $i + 1$. Nelle $n - 2$ aree di sosta intermedie $\{1, 2, \ldots n - 1\}$ si trovano delle stazioni di servizio nelle quali è possibile fare il pieno. Tutte le distanze e i valori di K e C sono numeri reali positivi. La auto parte dall'area 0 con il serbatoio pieno, e si sposta lungo la strada in direzione dell'area n senza mai tornare indietro. Progettare un algoritmo in grado di calcolare il numero minimo di fermate che sono necessarie per fare il pieno e raggiungere l'area di servizio n senza restare a secco per strada, se ciò è possibile. Nel caso in cui la destinazione non sia in alcun modo raggiungibile senza restare senza carburante, l'algoritmo restituisce -1.

Soluzione. È possibile utilizzare il seguente algoritmo che considera il tragitto dalla stazione di partenza 0 alla stazione di arrivo n utilizzando una variabile res che quantifica il numero di chilometri residui ancora percorribili in base allo stato attuale del serbatoio. Inizialmente res contiene il valore $K \times C$. Per ogni stazione che si incontra durante il percorso, res viene modificato in due possibili modi: decrementato del numero di chilometri per raggiungere la prossima stazione oppure reinizializzato a $K \times C$ se la prossima stazione non risulta raggiungibile in base agli attuali chilometri residui. In questo modo si effettua la scelta *greedy* di fare il pieno solo quando strettamente necessario per raggiungere la prossima stazione. Ogni volta che viene fatto il pieno si incrementa una variabile f che conta il numero di fermate.

Algorithm 1: MinFermate(*Real* $d[0..n]$, *Real* K, *Real* C) $\rightarrow$ *Intero*

```
Real res = K × C
Intero i = 0, f = 0
while i < n do
    if res < d[i] then
        res = K × C
        f = f + 1
    res = res − d[i]
    if res < 0 then
        return −1
    i = i + 1
return f
```

Si noti che nel caso in cui la distanza per raggiungere la prossima fermata risulta essere superiore al chilometraggio massimo $K \times C$, l'algoritmo restituisce immediatamente -1. Per quanto riguarda il costo computazionale, si nota che tutte le operazioni hanno costo costante e che nel caso pessimo il corpo del ciclo *while* viene eseguito n volte. Il costo computazionale risulta quindi essere $T(n) = O(n)$.

Esercizio. Disponiamo di un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al più n tubi più corti, aventi rispettivamente lunghezza $T[1], T[2], \ldots T[n]$. Il tubo viene segato sempre a partire da una delle due estremità, quindi ogni taglio riduce la lunghezza del tubo iniziale della misura del tubo $T[i]$ appena tagliato. Scrivere un algoritmo efficiente per determinare il numero massimo di tubi che è possibile ottenere. Formalmente, tra tutti i sottoinsiemi degli n tubi la cui lunghezza complessiva sia minore o uguale a L, vogliamo determinarne uno con il numero massimo di elementi e restituirne la cardinalità.

Soluzione. Ordiniamo i tubi in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima. Procediamo quindi a segare prima il tubo più corto, poi quello successivo e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere scritto in questo modo:

Algorithm 2: MAXNUMTUBI(*Intero* L, *Intero* $T[1..n]$) $\rightarrow$ *Intero*

```
OrdinaCrescente(S)
Intero i = 1
while i ≤ n and L ≥ S[i] do
    L = L − S[i]
    i = i + 1
return i − 1
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$ usando un algoritmo di ordinamento ottimale. Il successivo ciclo *while* ha costo $O(n)$. Il costo complessivo dell'algoritmo risulta quindi $\Theta(n \log n)$. Si noti che l'algoritmo di cui sopra restituisce l'output corretto sia nel caso in cui gli n tubi abbiano complessivamente lunghezza minore o uguale a L, sia nel caso in cui nessuno abbia lunghezza minore o uguale a L (in questo caso l'algoritmo restituisce zero).

Esercizio. Lungo una linea, a distanze costanti (che per comodità indichiamo con distanza 1), sono presenti $2n$ punti, n dei quali neri ed n bianchi. È necessario collegare ogni punto nero ad un corrispondente punto bianco tramite fili; ad ogni punto deve essere collegato uno ed un solo filo. Scrivere un algoritmo efficiente che stampa la quantità minima di filo necessaria. La distribuzione dei punti bianchi e neri viene passata all'algoritmo sotto forma di un array $p[1..2n]$ di booleani: se $p[i]$ è *true* allora in posizione i c'è un punto bianco, altrimenti se $p[i]$ è *false* allora in posizione i c'è un punto nero.

Soluzione. È possibile risolvere il problema con un semplice algoritmo greedy, leggendo i punti da sinistra a destra, e collegando ogni punto incontrato al primo fra i successivi di colore diverso.

Algorithm 3: COLLEGAPUNTI(*Boolean* $p[1..2n]$)

```
Intero i, filo = 0
Queue bianchi = new Queue() Queue neri = new Queue()
for i = 1 to 2n do
    if p[i] then
        // i-esimo punto bianco
        if neri.empty() then
            bianchi.enqueue(i)                    // non ci sono precedenti punti neri liberi
        else
            filo = filo + (i − neri.dequeue())    // collega ad un precedente punto nero
    else
        // i-esimo punto nero
        if bianchi.empty() then
            neri.enqueue(i)                       // non ci sono precedenti punti bianchi liberi
        else
            filo = filo + (i − bianchi.dequeue()) // collega ad un precedente punto bianco
print("Lunghezza minima filo:"+filo)
```

Considerando costo costante per le operazioni di *new*, *empty*, *enqueue* e *dequeue* sulle code, il costo dell'algoritmo è $\Theta(n)$, visto che il corpo del ciclo *while* viene eseguito $2n$ volte.

The End.