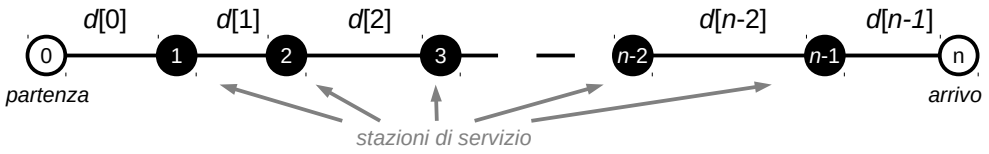
Corso di Algoritmi e Strutture Dati—Modulo 2

Esercizi su tecniche *Greedy*, 23 Aprile 2025

Esercizio 1. Un'auto può percorrere *K* Km con un litro di carburante, e il serbatoio ha una capacità di *C* litri. Lungo il tragitto si trovano $n + 1$ aree di sosta indicate con 0, 1, … *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 l'auto può fermarsi per fare il pieno (vedi figura).



Tutte le distanze e i valori di *K* e *C* sono numeri reali positivi. L'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.

```
MinFermate( real d[0..n - 1], real K, real C ) → integer
    real res ← K * C;
    integer i ← 0, f ← 0;
    while ( i < n ) do
        // res è il carburante residuo che mi rimane una volta arrivato alla stazione i.
        if ( res < d[i] ) then
            res ← C * K;
            f ← f + 1;
        endif
        res ← res – d[i];
        if ( res < 0 ) then
            errore “non raggiungibile”;
        endif
        i ← i + 1;
    endwhile
    return f;
```

Esercizio 2. Disponiamo di un tubo metallico di lunghezza *L*. Da questo tubo vogliamo ottenere al più *n* segmenti più corti, aventi rispettivamente lunghezza *S*[1], *S*[2], …, *S*[*n*]. Il tubo viene segato sempre a partire da una delle estremità, quindi ogni taglio riduce la sua lunghezza della misura asportata. Scrivere un algoritmo efficiente per determinare il numero massimo di segmenti che è possibile ottenere. Formalmente, tra tutti i sottoinsiemi degli *n* segmenti la cui lunghezza complessiva sia minore o uguale a *L*,.vogliamo determinarne uno con cardinalità massima. Determinare il costo computazionale dell'algoritmo proposto.

Soluzione. Ordiniamo le sezioni 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 segmento più corto, poi quello successiva e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere descritto in questo modo

```
MaxNumSezioni( integer L, integer S[1..n] ) → integer
    OrdinaCrescente(S);
    integer i ← 1;
    while ( i ≤ n and L ≥ S[i] ) do
        L ← L - S[i];       // diminuisce la lunghezza residua
        i ← i + 1;
    endwhile
    return i - 1;
```

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

Esercizio 3. Siete stati assunti alla Microsoft per lavorare alla prossima versione di Word, denominata Word 2030. Il problema che dovete risolvere è il seguente. È data una sequenza di *n* parole, le cui lunghezze (espresse in punti tipografici, numeri interi) sono memorizzate nel vettore *w*[1], …, *w*[*n*]. È necessario suddividere le parole in righe di lunghezza massima pari a *L* punti tipografici, in modo che lo spazio non utilizzato in ciascuna riga sia minimo possibile. Tra ogni coppia di parole consecutive posizionate sulla stessa riga viene inserito uno spazio che occupa *S* punti tipografici; nessuno spazio viene inserito dopo l'ultima parola di ogni riga. La lunghezza del testo su ogni riga è quindi data dalla somma delle lunghezze delle parole e degli spazi di separazione. *L* è maggiore della lunghezza di ogni singola parola (quindi in ogni riga può sempre essere inserita almeno una parola). Non è possibile riordinare le parole, che devono comparire esattamente nell'ordine dato.

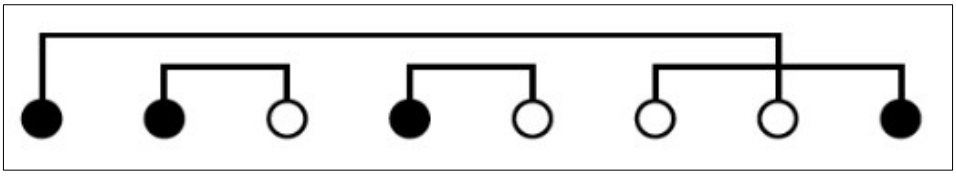
1. Scrivere un algoritmo efficiente che, dato in input il vettore *w*[1], …, *w*[*n*], e i valori *S* e *L*, stampi una suddivisione delle parole che minimizza lo spazio inutilizzato in ciascuna riga. Ad esempio, supponendo di avere 15 parole, l'algoritmo potrebbe stampare la stringa "[1 3][4 8][9 15]" per indicare che la prima riga contiene le parole da 1 a 3 (incluse), la seconda le parole da 4 a 8, e la terza le parole da 9 a 15.
2. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione. È possibile risolvere il problema con un semplice algoritmo greedy: si inseriscono in ciascuna riga le parole, nell'ordine indicato, finché non si supera la lunghezza massima consentita. Utilizziamo la variabile start per indicare l'indice della prima parola della riga corrente; Lres è la lunghezza residua (ancora da riempire) dalla riga corrente.

```
algoritmo FormattaParagrafo( array w[1..n] di int, int S, int L )
    int start := 1;
    int Lres := L - w[1];
    for i := 2 to n do
        if ( Lres ≥ S + w[i]) then   // aggiungiamo la parola i-esima alla riga corrente
            Lres := Lres – S - w[i];
        else                                  // iniziamo una nuova riga
            print "[" + start + " " + (i-1) + "]";
            start := i;
            Lres := L - w[i];
        endif
    endfor
    print "[" + start + " " + n + "]";
```

Si noti la stampa effettuata al termine del ciclo “for”, senza la quale l'algoritmo non verrebbero stampate le parole dell'ultima riga. Il costo dell'algoritmo è O(*n*).

Esercizio 4 Lungo una linea, a distanze costanti (che per comodità indichiamo con **distanza 1**), sono presenti ***n*** punti neri ed ***n*** punti 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 per determinare la quantità minima di filo necessaria. Determinare il costo computazionale dell'algoritmo proposto.
A titolo di esempio, nell'immagine sotto viene riportata una istanza del problema con 4 punti neri e 4 punti bianchi, ed un corrispondente collegamento di punti che richiede l'uso di una lunghezza complessiva di filo pari a 10. Si noti che tale collegamento non è ottimale, in quanto sarebbe possibile usare una lunghezza complessiva pari a 8.



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: nel caso in cui ci siano più punti di medesimo colore che si possono collegare ad un successivo punto di colore diverso, si dà priorità a quello più lontano.

```
algoritmo CollegaPunti( array p[1..2n] di bool )        // true = bianco,  false = nero
   int i, filo := 0;
   queue bianchi := new 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);
           else filo := filo + (i – neri.dequeue());          // collega ad un precedente nero
           endif
       else                                    // i-esimo punto nero
           if (bianchi.empty()) then neri.enqueue(i);
           else filo := filo + (i – bianchi.dequeue());       // collega ad un precedente bianco
           endif
       endif
   endfor
   print “Lunghezza minima filo: ” + filo;
```

Considerando costo costante per le operazioni di *new, empty, enqueue e dequeue* sulle code, il costo dell'algoritmo è ***O(n)***.