# Batterie

- Siete a bordo di un'auto elettrica su un'autostrada. Entrate in autostrada al km 0 con la batteria carica e dovete uscire al km $L$.
- Prima che la vostra batteria si esaurisca (dopo $r$ km), dovete fermarvi in un'area di servizio e sostituirla con una batteria di ricambio.
- Sia $D[1 \ldots n]$ un vettore di interi, dove $D[i]$ è la distanza dell'area di servizio $i$-isima dall'inizio dell'autostrada.
- Scrivete un algoritmo che prenda in input $D$, $n$, $L$ e $r$ e restituisca il numero minimo di fermate necessarie per completare il viaggio. Discutere correttezza e complessità.

# Batterie

- Siete a bordo di un'auto elettrica su un'autostrada. Entrate in autostrada al km 0 con la batteria carica e dovete uscire al km $L$.
- Prima che la vostra batteria si esaurisca (dopo $r$ km), dovete fermarvi in un'area di servizio e sostituirla con una batteria di ricambio.
- Siano $D[1 \dots n]$ e $C[1 \dots n]$ due vettori di interi, dove $D[i]$ è la distanza dell'area di servizio $i$-isima dall'inizio dell'autostrada, e $C[i]$ è il costo di una nuova batteria nell'area $i$.
- Il costo totale del viaggio è dato dalla somma dei costi delle batterie sostituite per arrivare al km $L$.
- Scrivete un algoritmo che prenda in input $D$ e $C$, $n$, $L$, $r$ e restituisca il costo totale minimo. Discutere correttezza e complessità.

# Supersequenza comune minimale

- Una stringa $P$ è una supersequenza di una stringa $T$ se $T$ è una sottosequenza di $P$.
- Scrivere un algoritmo che restituisca la lunghezza della *supersequenza comune minimale* di due stringhe $P$, $T$, ovvero la più piccola supersequenza di entrambe le stringhe.
- Discutere correttezza e complessità dell'algoritmo proposto.
- Esempio: L'unica supersequenza comune minimale di `AB` e `BC` è `ABC`, e la sua lunghezza è pari a 3.
- Esempio: Esistono due supersequenze comuni minimali di `DAB` e `DCB`, ovvero `DACB` e `DCAB`, e la loro lunghezza è pari a 4.

# Costo partizione di un vettore

- Il costo $C(i, j)$ di un sottovettore $V[i \dots j]$ di $V$ è pari alla somma dei suoi elementi
  - Esempio: $V = [1, 2, 4, 8, 16, 2]$, $C(2, 4) = 2 + 4 + 8 = 14$.
- Una $k$-partizione di $V$ è una divisione di $V$ in $k$ sottovettori non vuoti, contigui che coprono totalmente il vettore e non si sovrappongono.
  - Esempio: $V = [2, 3, 7, -7, 15, 2]$, una delle $n - 1$ 2-partizioni è $[2, 3, 7], [-7, 15, 2]$.
- Il costo della $k$-partizione è il costo massimo dei suoi sottovettori.
  - Nella 2-partizione $[2, 3, 7], [-7, 15, 2]$, il costo dei sottovettori è
    $2 + 3 + 7 = 12$,
    $-7 + 15 + 2 = 10$,
    quindi il costo della 2-partizione è pari a 12.

# Costo partizione di un vettore (Versione formale)

- Il costo $C(i,j) = \sum_{t=i}^{j} V[t]$
- $k$-partizione di $V$ è una divisione di $V$ in $k$ sottovettori contigui $V[j_0+1 \ldots j_1], V[j_1+1 \ldots j_2], \ldots, V[j_{k-1}+1 \ldots j_k]$ con $j_0 = 0, j_k = n$ e $j_t < j_{t+1}, \forall 1 \leq t < k$.
- Il costo della $k$-partizione è pari a:
$$\max\{C(j_{t-1}+1, j_t) : \forall 1 \leq t \leq k\}$$

# Costo partizione di un vettore

Scrivere un algoritmo che prenda in input un vettore $V$ contenente $n$ interi e un intero $k$ tale che $2 \leq k \leq n$ e restituisca il costo della $k$-partizione di $V$ di costo minimo.

Esempio: $V = [2, 3, 7, -7, 15, 2]$, $k = 3$
$[2, 3, 7], [-7, 15], [2]$ costo 2+3+7=12
$[2, 3], [7][-7, 15, 2]$ costo -7+15+2=10

1. Soluzione per $k = 2$ (facile: $O(n^2)$, meglio in $O(n)$).
2. Soluzione per $k = 3$ (facile: $O(n^3)$, meglio in $O(n^2)$).
3. Soluzione generale (facile: $O(n^k)$, meglio in $O(kn^2)$).

# Spoiler alert!

# Batterie (Esercizio 1.3 di 14-greedy.pdf)

```
SET minStops(int[] D, int n, int L, int r)
int deadline = r
SET stops = Set()
for i = 2 to n do
    if D[i] ≥ deadline then
        stops.insert(i − 1)
        deadline = D[i − 1] + r
if deadline < L then
    stops.insert(n)
return stops
```

# Batterie (Esercizio 1.3 di 14-greedy.pdf)

- Assumiamo che l'autostrada sia percorribile, ovvero l'autonomia $r$ sia sufficiente per andare:
  - da una stazione alla successiva
  - dal km 0 alla prima stazione
  - dall'ultima stazione al km $L$
- Per minimizzare il numero di fermate, procediamo in modo greedy: cerchiamo la prima stazione che non può essere raggiunta con autonomia $r$ e inseriamo la stazione precedente.
- Dobbiamo dimostrare:
  - Sottostruttura ottima
  - Scelta greedy
  - Correttezza nei casi limite e nei valori estremi
- La complessità dell'algoritmo è $O(n)$.

# Batterie (Esercizio 1.3 di 14-greedy.pdf)

## Casi limite

- La prima stazione deve essere raggiungibile con autonomia $r$, quindi partiamo dalla seconda
- Se l'ultima stazione a cui abbiamo fatto rifornimento non ci permette di raggiungere $L$, aggiungiamo la stazione precedente;
- Se esiste una stazione sola, ci sono due casi:
  - se $r \geq L$, la carica ci permette di uscire dall'autostrada; ma allora *deadline* $\geq L$ e l'algoritmo restituisce l'insieme vuoto
  - se $r < L$, l'algoritmo restituisce l'unica stazione presente

# Batterie (Esercizio 1.3 di 14-greedy.pdf)

## Scelta greedy

- Assumiamo che esista una soluzione $S$ che non includa la stazione $k$, l'ultima stazione tale che $D[k] \leq r$, a partire dall'inizio (km 0).
- sia $i$ la prima stazione utilizzata in $S$, con $i < k$;
- se sostituisco $i$ con $k$, ottengo una soluzione $S' = S - \{i\} \cup \{k\}$ che è comunque una soluzione ottima (ha la stessa dimensione) e rispetta tutti i vincoli ($k$ è raggiungibile dall'inizio e può raggiungere la stazione successiva in $S$, perchè questa era raggiungibile da $i$ che era più indietro).

# Batterie (Esercizio 1.8 di 13-pd.pdf)

È possibile definire ricorsivamente il problema come segue. Assumiamo che esistano due stazioni fittizie $0, n+1$, con $D[0] = 0$ e $D[n+1] = L$ e $C[0] = C[n+1] = 0$. Non è quindi necessario passare $L$.

Definiamo una tabella $DP[0 \ldots n+1]$, tale che $DP[i]$ rappresenti il costo minimo da pagare nel caso si acquisti la batteria alla stazione $i$-esima. La nostra soluzione si trova in $DP[0]$. $DP[i]$ può essere definito in maniera ricorsiva nel modo seguente:

$$DP[i] = \begin{cases} 0 & \text{se } i = n+1 \\ \min_{j:j>i \wedge D[j] \leq D[i]+r} \{DP[j]\} + C[i] & \text{altrimenti} \end{cases}$$

# Batterie (Esercizio 1.8 di 13-pd.pdf)

```
int minCostStops(int[] D, int[] C, int n, int r)
int[] DP = new int[0...n+1]
DP[n+1] = 0
for i = n downto 0 do
    int j = i + 1
    int DP[i] = +∞
    while j ≤ n + 1 and D[j] ≤ D[i] + r do
        DP[i] = min(DP[i], DP[j])
        j = j + 1
    DP[i] = DP[i] + C[i]
return DP[0]
```

# Batterie (Esercizio 1.8 di 13-pd.pdf)

Inviduare l'indice $j$ per cui $D[j] \leq D[i] + r$ può richiedere $O(n)$; quindi il costo pessimo di tale algoritmo è $O(n^2)$.

# Supersequenza comune minimale (Compito 10/01/2017)

La lunghezza della più lunga supersequenza comune può essere calcolata tramite la seguente espressione ricorsiva:

$$DP[i][j] = \begin{cases} i & i \geq 0 \wedge j = 0 \\ j & i = 0 \wedge j \geq 0 \\ DP[i-1][j-1] + 1 & i > 0 \wedge j > 0 \wedge p_i = t_j \\ \min\{DP[i-1][j], DP[i][j-1]\} + 1 & i > 0 \wedge j > 0 \wedge p_i \neq t_j \end{cases}$$

- Nel caso una delle stringhe sia vuota, tutti i caratteri dell'altra stringa devono essere presenti nella supersequenza, da cui i due casi base.
- Se gli ultimi caratteri delle due stringhe sono uguali, si considera il sottoproblema in cui viene rimosso l'ultimo carattere di entrambi e si aggiunge +1 per contare questo carattere.
- Nel caso siano diversi, si prenderanno i due sottoproblemi in cui si rimuove l'ultimo carattere di una delle stringhe e si aggiunge +1 per contare questo carattere, prendendo il valore più piccolo fra i due.

# Supersequenza comune minimale (Compito 10/01/2017)

```
int scs(Item[] P, Item[] T, int n, int m)
int[][] D = new int[0...n][0...m]
for i = 0 to n do
    DP[i][0] = i
for j = 0 to m do
    DP[0][j] = j
for i = 1 to n do
    for j = 1 to m do
        if P[i] == T[j] then
            DP[i][j] = DP[i − 1][j − 1] + 1
        else
            DP[i][j] = min(DP[i − 1][j], DP[i][j − 1]) + 1
return DP[n][m]
```

# Costo partizione di un vettore (Compito 05/06/14)

- Vediamo innanzitutto un approccio generico al problema, estremamente inefficiente per valori grandi di $k$.
- Proponiamo poi soluzioni efficienti "ad-hoc" per $k = 2$ e $k = 3$, "propedeutiche" alla costruzione di una soluzione generale.
- Infine, vediamo una soluzione generica che può essere utilizzata per qualunque valore di $k$, basata su programmazione dinamica

# Costo partizione di un vettore (Versione $\Theta(n^k)$)

**int** partitionK(**int**[ ] $V$, **int** $n$)

**int** $minSoFar = +\infty$
**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** $\ldots = \ldots + 1$ **to** $n$ **do**
        **for** $i_{k-1} = i_{k-2} + 1$ **to** $n$ **do**
          $tot_1 = \mathsf{sum}(V, 1, i_1)$
          $tot_2 = \mathsf{sum}(V, i_1 + 1, i_2)$
          $tot_3 = \mathsf{sum}(V, i_2 + 1, i_3)$
          $\ldots$
          $tot_k = \mathsf{sum}(V, i_{k-1} + 1, n)$
          $minSoFar = \mathsf{min}(minSoFar, \mathsf{max}(tot_1, \ldots, tot_k))$

# 2-partizione

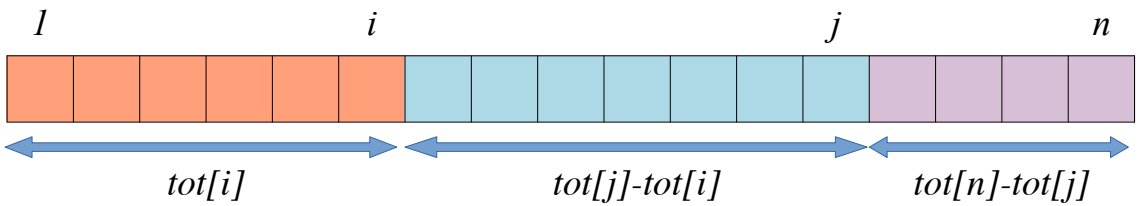
**int** partition2(**int**[ ] $V$, **int** $n$)

**int** $tot = 0$
**for** $i = 1$ **to** $n$ **do**
  $tot = tot + V[i]$
**int** $sumSoFar = 0$
**int** $minSoFar = +\infty$
**for** $i = 1$ **to** $n - 1$ **do**
  $sumSoFar = sumSoFar + V[i]$
  $minSoFar = \min(minSoFar, \max(sumSoFar, tot - sumSoFar))$
**return** $minSoFar$

# 3-partizione

**int** partition3(**int**[ ] $V$, **int** $n$)

**int**[ ] $tot$ = **new int**$[1 \ldots n]$
$tot[1] = V[1]$
**for** $i = 2$ **to** $n$ **do**
  $tot[i] = tot[i-1] + V[i]$
**int** $minSoFar = +\infty$
**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** $minSoFar$



# $k$-partizione

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

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

# $k$-partizione

**int** partition(**int**[ ] $V$, **int** $n$, **int** $k$)

**int**[ ][ ] $DP$ = **new int**$[1 \ldots n][1 \ldots k] = \{-1\}$ % Initizialized to $-1$
**int**[ ] $tot$ = **new int**$[1 \ldots n]$
$tot[1] = V[1]$
**for** $i = 2$ **to** $n$ **do**
  $tot[i] = tot[i-1] + V[i]$
**return** partitionRec$(V, tot, DP, n, k)$

# $k$-partizione

```
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]
```

# $k$-partizione

Costo computazionale:

- La tabella ha dimensione $n \times k$
- Ogni cella richiede tempo $O(n)$ per essere riempita
- Il costo computazionale è quindi $O(kn^2)$