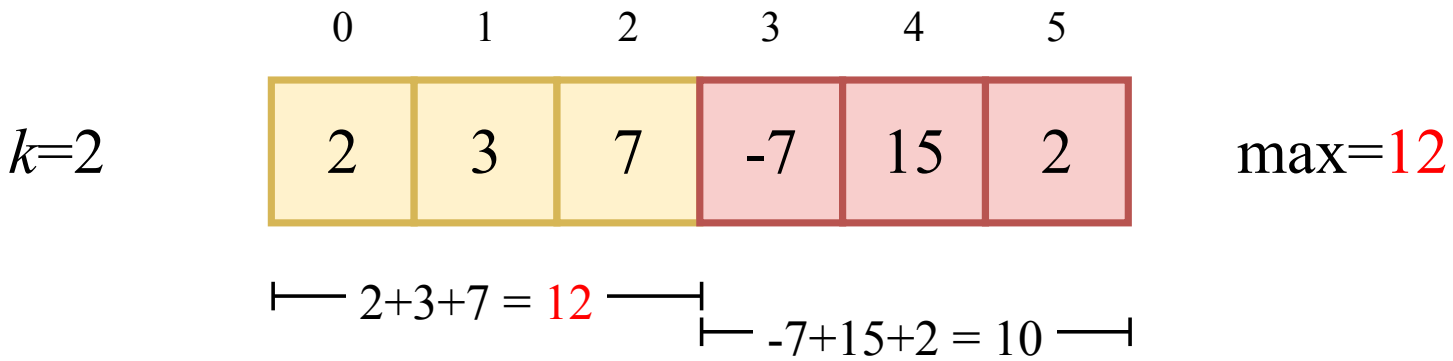
Supersequenza comune minimale

- 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 e T, cioè della più corta stringa che abbia sia P sia T come sottosequenze
- Discutere correttezza e complessità dell'algoritmo proposto
- Esempio 1: L'unica supersequenza comune minimale di `AB` e `BC` è `ABC`, quindi la sua lunghezza è 3
- Esempio 2: Le supersequenze comuni minimali di `DAB` e `DCB` sono `DACB` e `DCAB`, quindi la loro lunghezza è 4

OBIETTIVO: warm-up e riutilizzo di idee già viste in precedenza

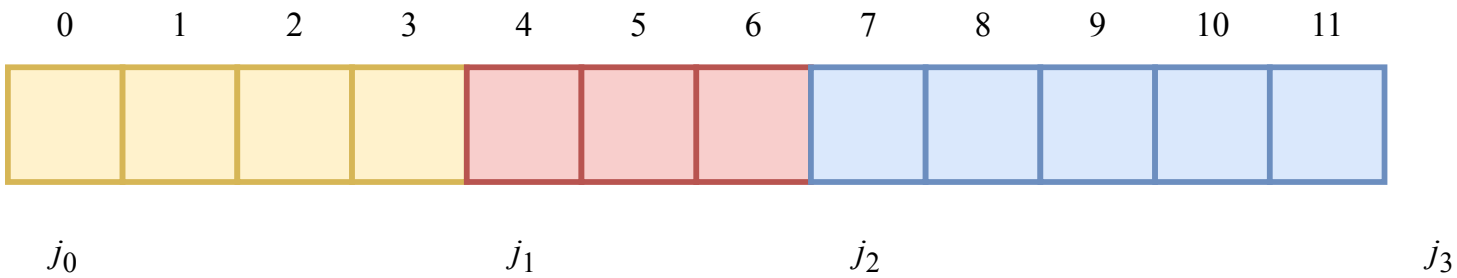
Costo partizione di un vettore

- Il costo $C(i, j)$ di un sottovettore $V[i \dots j]$ di V è pari alla somma dei suoi elementi
- Una k-partizione di V è una divisione di V in k sottovettori contigui e non vuoti che coprono totalmente il vettore e non si sovrappongono.
- Il costo della k-partizione è il costo massimo dei suoi sottovettori.



Costo partizione di un vettore (Versione formale)

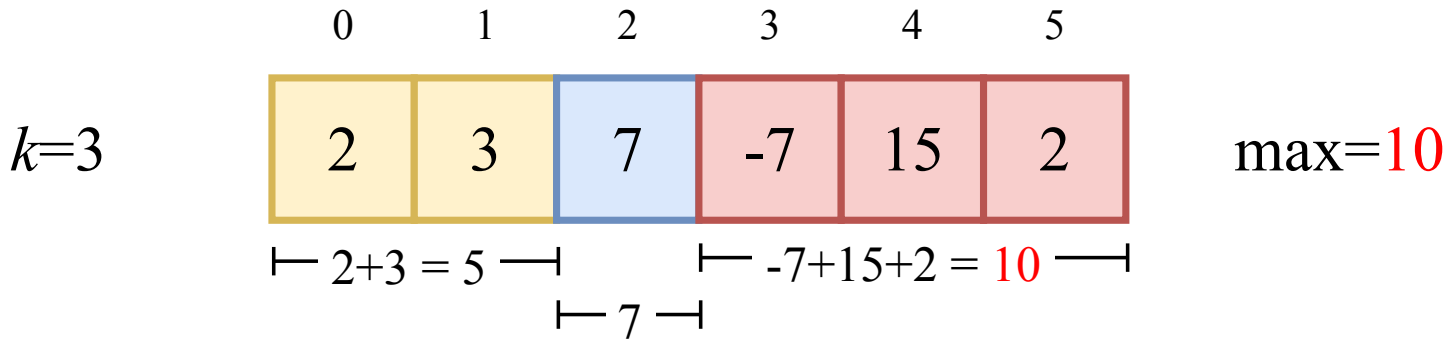
- Il costo di un sottovettore $V[i \dots j]$ è $C(i,j) = \sum_{t=i}^{j} V[t]$
- Una k-partizione di V è una divisione di V in k sottovettori contigui $V[j_0 \dots j_1 - 1], V[j_1 \dots j_2 - 1], \dots, V[j_{k-1} \dots j_k - 1]$ con $j_0 = 0, j_k = n$ e $j_t < j_{t+1}, \forall t, 0 \leq t < k$.
- Il costo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



SUGGERIMENTO: Verificate se la vostra comprensione del problema è in linea con la formulazione matematica dello stesso

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.



Strategia per approcciare il problema:

1. Soluzione naif per $k = 2$: $O(n^2)$; ottimizzata: $O(n)$
2. Soluzione naif per $k = 3$: $O(n^3)$; ottimizzata: $O(n^2)$
3. Soluzione naif per k: $O(n^k)$; ottimizzata $O(n^{k-1})$; tramite programmazione dinamica: $O(kn^2)$

Batterie (numero di fermate)

- 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.
- Una batteria carica permette di percorrere R chilometri; prima che si esaurisca, bisogna fermarsi in un'area di servizio e sostituirla con una nuova batteria carica.
- Sia $dist$ un vettore strettamente crescente contenente n interi, dove $dist[i]$ è la distanza dell'area di servizio i-esima dal km 0.
- Scrivere un algoritmo che, dati $dist$, n, L e R, restituisca il numero minimo di fermate necessarie per arrivare al km L.
- Assumiamo che il viaggio sia possibile: la distanza fra 0 e $dist[0]$, ogni distanza fra due stazioni successive e la distanza fra $dist[n-1]$ e L sono minori o uguali a R; oppure, $L \leq R$.

Batterie (costo di sostituzione)

Consideriamo una variante del problema precedente, dove la sostituzione ha un costo. Per semplificare la formulazione, aggiungiamo una posizione fittizia finale che rappresenta l'uscita dall'autostrada.

- Sia *dist* un vettore strettamente crescente contenente $n+1$ interi positivi: per $0 \leq i < n$, $dist[i]$ è la distanza dell'area di servizio i-esima dal km 0, mentre $dist[n] = L$ è l'uscita dell'autostrada.
- Sia *cost* un vettore contenente $n+1$ interi non negativi: per $0 \leq i < n$, $cost[i]$ è il costo di una nuova batteria nell'area i-esima, mentre $cost[n] = 0$ indica che non è necessario pagare per arrivare a L.
- Il costo totale del viaggio è dato dalla somma dei costi delle batterie sostituite per arrivare al km L.
- Scrivere un algoritmo che, dati *dist*, *cost*, n e R, restituisca il costo totale minimo. Discutere correttezza e complessità.

Donald Trump

La Route 66 è una strada che collega Chicago a Los Angeles e lungo la quale si trovano n città. Nel suo tour elettorale Donald Trump (DJT) ha deciso di percorrerla in una sola direzione, senza mai tornare indietro sui propri passi. DJT non può tenere un comizio in ogni città, ma deve sceglierne un sottoinsieme.

Date due città i, j in cui terrà un comizio, esse devono trovarsi a distanza maggiore o uguale a D. La città i-esima si trova al miglio $miles[i]$; quindi la distanza fra i e j è pari a $|miles[j] - miles[i]|$.

Seguendo queste regole, The Donald vorrebbe parlare al maggior numero possibile di potenziali elettori. Si stima che al comizio nella città i-esima saranno presenti $people[i]$ persone.

Donald Trump

Sorprendentemente, The Donald non ha grandi conoscenze informatiche e ha chiesto a voi di risolvere il problema; in particolare, vorrebbe un algoritmo

int serveDJT(**int**[ ] *miles*, **int**[ ] *people*, **int** n, **int** D)

che prenda in input il vettore *miles* delle posizioni delle città (strettamente crescente), il vettore *people* del numero stimato di partecipanti, la dimensione n dei vettori e la distanza minima D tra due città in cui tenere un comizio, e restituisca il maggior numero complessivo di persone che possono essere presenti ai suoi comizi.

Thank you for your attention to this matter! DJT

Spoiler alert!

Supersequenza comune minimale – Ricorrenza

$DP[i][j]$ rappresenta la lunghezza della più corta supersequenza comune dei prefissi $P[0 \ldots i-1]$ e $T[0 \ldots j-1]$.

$$DP[i][j] = \begin{cases} \end{cases}$$

Supersequenza comune minimale – Ricorrenza

$DP[i][j]$ rappresenta la lunghezza della più corta supersequenza comune dei prefissi $P[0 \ldots i-1]$ e $T[0 \ldots j-1]$.

$$DP[i][j] = \begin{cases} i & j = 0 \\ j & i = 0 \\ DP[i-1][j-1] + 1 & i > 0, j > 0, P[i-1] = T[j-1] \\ \min\{DP[i-1][j], DP[i][j-1]\} + 1 & i > 0, j > 0, P[i-1] \neq T[j-1] \end{cases}$$

- Se uno dei due prefissi è vuoto, la supersequenza deve contenere tutti i caratteri dell'altro prefisso, da cui i casi base
- Se gli ultimi caratteri coincidono, li inseriamo una sola volta nella supersequenza e ci riduciamo al sottoproblema sui due prefissi più corti
- Se gli ultimi caratteri sono diversi, dobbiamo includerne uno dei due: confrontiamo quindi i due sottoproblemi ottenuti rimuovendo l'ultimo carattere di una delle due stringhe, e scegliamo il migliore

Supersequenza comune minimale – Algoritmo

```
int scs(Item[] P, Item[] T, int n, int m)
int[][] DP = new int[0...n][0...m]
for i = 0 to n do
    DP[i][0] = i                                   % Caso base j = 0
for j = 0 to m do
    DP[0][j] = j                                   % Caso base i = 0
for i = 1 to n do
    for j = 1 to m do
        if P[i-1] == T[j-1] 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** k)

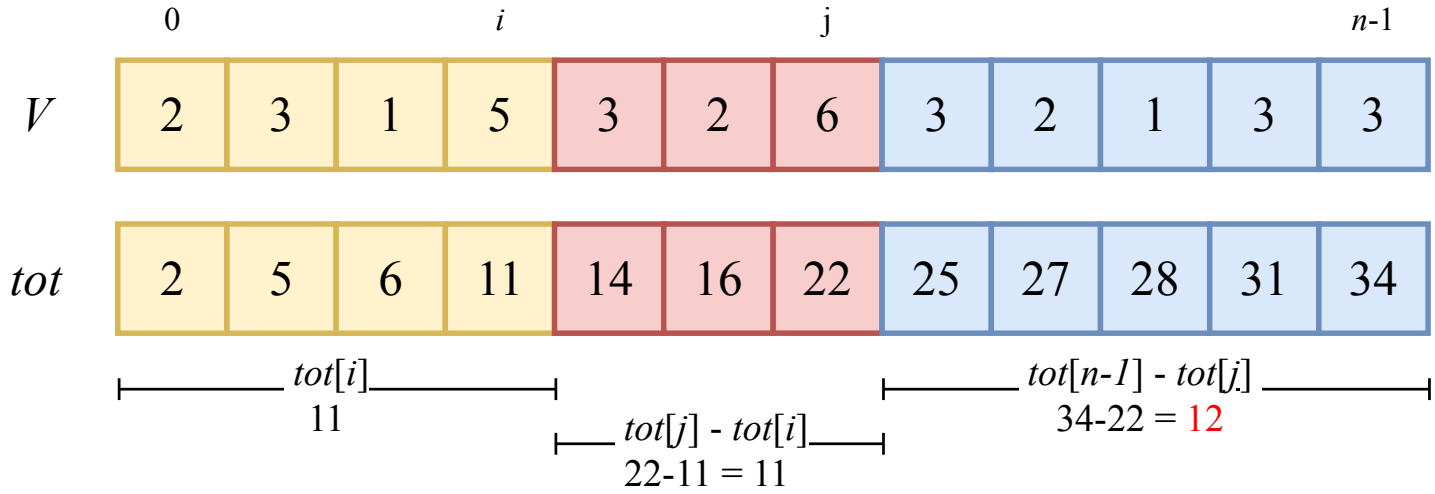
```
int minSoFar = +∞
for i_1 = 0 to n − k do
    tot_1 = sum(V, 0, i_1)
    for i_2 = i_1 + 1 to n − k + 1 do
        tot_2 = sum(V, i_1 + 1, i_2)
        for i_3 = i_2 + 1 to n − k + 2 do
            tot_3 = sum(V, i_2 + 1, i_3)
            for ... = ... + 1 to n − k + ... do
                ...
                for i_{k−1} = i_{k−2} + 1 to n − 1 do
                    tot_k = sum(V, i_{k−1} + 1, n − 1)
                    minSoFar = min(minSoFar, max(tot_1, ..., tot_k))
```

2-partizione

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

int $tot = 0$
for $i = 0$ **to** $n - 1$ **do**
  $tot = tot + V[i]$
int $sumSoFar = 0$
int $minSoFar = +\infty$
for $i = 0$ **to** $n - 2$ **do**
  $sumSoFar = sumSoFar + V[i]$
  $minSoFar = \min(minSoFar, \max(sumSoFar, tot - sumSoFar))$
return $minSoFar$

3-partizione



---

int[ ] computeTot(**int**[ ] V, **int** n)

---

int[ ] tot = **new int**$[0 \ldots n-1]$
$tot[0] = V[0]$
for $i = 1$ **to** $n-1$ **do**
  $tot[i] = tot[i-1] + V[i]$
return tot

3-partizione

```
int partition3(int[] V, int n)
int[] tot = computeTot(V, n)
int minSoFar = +∞
for i = 0 to n − 3 do
    for j = i + 1 to n − 2 do
        int temp = max(tot[i], tot[j] − tot[i], tot[n − 1] − tot[j])
        minSoFar = min(minSoFar, temp)
return minSoFar
```

Nota: il trucco di usare tot può essere utilizzato per qualunque k, facendo scorrere $k-1$ indici e riducendo di un fattore $O(n)$ la complessità.

Ma... si può fare meglio di così, tramite programmazione dinamica

k-partizione

Sia $DP[t][i]$ il costo minimo di una t-partizione di $V[0 \dots i]$.
Problema originale: $DP[k][n-1]$. Utilizziamo il vettore di supporto tot

$$DP[t][i] = \begin{cases} +\infty & i+1 < t \\ tot[i] & t = 1 \\ \min\limits_{0 \le j < i} \left\{ \max\big(DP[t-1][j],\ tot[i] - tot[j]\big) \right\} & t > 1,\ i+1 \ge t \end{cases}$$

- Il numero di elementi in $V[0 \dots i]$ è pari a $i+1$; se $i+1 < t$, non ho abbastanza elementi per ottenere t partizioni
- Se $t = 1$, c'è un'unica partizione la cui somma è pari a $tot[i]$
- Altrimenti, scegliamo l'ultimo taglio: per ogni $j < i$, $V[j+1 \dots i]$ è l'ultimo sottovettore e ci riduciamo al problema $DP[t-1][j]$

k-partizione

```
int partition(int[] V, int n, int k)
int[][] DP = new int[1...k][0...n-1]
int[] tot = computeTot(V, n)
for i = 0 to n-1 do
    DP[1][i] = tot[i]
for t = 2 to k do
    for i = 0 to n-1 do
        DP[t][i] = +∞
        if t ≤ i+1 then
            for j = 0 to i-1 do
                int cost = max(DP[t-1][j], tot[i] - tot[j])
                DP[t][i] = min(DP[t][i], cost)
return DP[k][n-1]
```

k-partizione

Costo computazionale:

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

Batterie (numero di fermate)

- Assumiamo che il viaggio sia possibile; questa condizione può essere verificata facilmente prima di eseguire l'algoritmo.
- Idea greedy:
  - con la batteria corrente possiamo raggiungere tutte le stazioni fino a una certa distanza limite;
  - cerchiamo la prima destinazione non raggiungibile: può essere una stazione oppure l'uscita al km L;
  - a quel punto dobbiamo necessariamente fermarci in una stazione precedente;
  - conviene allora scegliere la stazione più avanzata tra quelle raggiungibili, cioè l'ultima stazione raggiungibile;
  - fermarsi prima non può aiutare a ridurre il numero totale di fermate.
- Ripetiamo lo stesso ragionamento fino a raggiungere L.
- L'algoritmo richiede tempo $O(n)$.

Batterie (numero di fermate)

```
int minStops(int[] dist, int n, int L, int R)
int deadline = R          % Massima distanza con la batteria corrente
int stops = 0             % Numero di fermate
for i = 1 to n − 1 do
    if dist[i] > deadline then    % La stazione i non è raggiungibile
        stops = stops + 1
        deadline = dist[i − 1] + R    % Riparto dall'ultima fermata
if L > deadline then      % L'uscita non è ancora raggiungibile
    stops = stops + 1
return stops
```

Batterie (numero di fermate)

Scelta greedy

- Sia j l'ultima stazione tale che $dist[j] \leq R$, cioè l'ultima stazione raggiungibile direttamente dall'inizio.
- Assumiamo esista una soluzione ottima S che non includa j
- Sia i la prima stazione utilizzata in S; allora $i < j$.
- Se sostituiamo i con j, otteniamo una soluzione $S' = S - \{i\} \cup \{j\}$ con lo stesso numero di fermate.
- La soluzione S' è ancora ammissibile: la stazione j è raggiungibile dall'inizio e la stazione successiva in S resta raggiungibile anche da j, perché j si trova più avanti di i.
- Esiste quindi una soluzione ottima che utilizza la scelta greedy.

Batterie (numero di fermate)

Sottostruttura ottima

Dopo la fermata scelta greedy, ripartiamo con una batteria carica da una nuova stazione. Il problema residuo ha quindi la stessa struttura del problema iniziale, e possiamo ripetere lo stesso ragionamento.

Un caso limite

- Se $L \leq R$, l'uscita è raggiungibile con la batteria iniziale.
- Inoltre, poiché tutte le stazioni si trovano prima del km L, vale anche $dist[i] < L \leq R$ per ogni i, quindi il ciclo non effettua alcuna fermata.
- Anche l'ultimo controllo fallisce, e l'algoritmo restituisce correttamente 0; questo è vero anche nel caso $n = 0$.

Batterie (costo di sostituzione)

Sia $DP[i]$ il costo minimo per raggiungere la posizione i-esima del vettore esteso e pagare l'eventuale costo associato a tale posizione.

Per raggiungere la posizione i abbiamo due possibilità:

- arrivarci direttamente dall'inizio, la scelta migliore se $dist[i] \leq R$;
- arrivarci da una stazione precedente $j < i$ tale che $dist[i] - dist[j] \leq R$.

Otteniamo quindi la ricorrenza:

$$DP[i] = \begin{cases} cost[i] & \text{se } dist[i] \leq R \\ cost[i] + \min\limits_{\substack{0 \leq j < i \\ dist[i]-dist[j] \leq R}} DP[j] & \text{altrimenti} \end{cases}$$

La soluzione finale si trova in $DP[n]$

Batterie (costo di sostituzione)

```
int minCostStops(int[] dist, int[] cost, int n, int R)
int[] DP = new int[0...n]
for i = 0 to n do
    if dist[i] ≤ R then                          % Raggiungibile direttamente
        DP[i] = cost[i]
    else                                         % Bisogna passare per stazione intermedia
        DP[i] = +∞
        int j = i − 1                            % Inizia dalla precedente
        % Arretra finché la distanza resta entro R
        while j ≥ 0 and dist[i] − dist[j] ≤ R do
            DP[i] = min(DP[i], DP[j] + cost[i])
            j = j − 1
return DP[n]
```

Batterie (costo di sostituzione)

La tabella DP contiene $n+1$ valori.

Per calcolare $DP[i]$, nel caso pessimo dobbiamo esaminare tutte le stazioni precedenti $j < i$ per verificare quali consentono di raggiungere i.

Il costo totale è quindi

$$\sum_{i=0}^{n} O(i) = O(n^2)$$

Nota: utilizzando strutture dati speciali, è possibile scendere a $O(n \log n)$

Donald Trump (Compito 16/7/2016)

Per ogni città i, con $1 \leq i \leq n$, sia

$$p(i) = \max\{\, j < i : miles[i-1] - miles[j-1] \geq D \,\}$$

cioè l'indice dell'ultima città precedente compatibile con un comizio nella città i. Se tale città non esiste, poniamo $p(i) = 0$.

Sia $DP[i]$ il numero massimo di persone che posso incontrare nelle prime i città. $DP[n]$ corrisponde alla soluzione del problema originale.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max\{DP[i-1],\ people[i-1] + DP[p(i)]\} & 1 \leq i \leq n \end{cases}$$

Infatti, per la città i-esima (posizione $i-1$), o non tengo un comizio, e allora ottengo $DP[i-1]$, oppure lo tengo, e allora posso aggiungere $people[i-1]$ alla soluzione ottima sulle prime $p(i)$ città.

Donald Trump – Costo $\Theta(n^2)$

int serveDJT(**int**[ ] *miles*, **int**[ ] *people*, **int** n, **int** D)

```
int[ ] DP = new int[0 . . . n]
DP[0] = 0
for i = 1 to n do
    int maxSoFar = people[i − 1]
    int j = 1
    while j < i and miles[i − 1] − miles[j − 1] ≥ D do
        maxSoFar = max(maxSoFar, people[i − 1] + DP[j])
        j = j + 1
    DP[i] = max(maxSoFar, DP[i − 1])
return DP[n]
```

Donald Trump – Costo $O(n \log n)$

È possibile ridurre questo problema al problema dell'*insieme indipendente di intervalli pesati*, associando alla città i-esima

- l'intervallo $[miles[i] - D, miles[i])$
- il peso $people[i]$

Due intervalli sono compatibili se e solo se le corrispondenti città distano almeno D; il problema diventa quindi quello di selezionare un insieme di intervalli compatibili di peso totale massimo.
Utilizzando l'algoritmo visto a lezione, è possibile risolvere il problema in tempo $O(n \log n)$. Se le distanze sono già ordinate, i predecessori compatibili possono essere calcolati in tempo $O(n)$, e quindi anche il problema si risolve in tempo $O(n)$.