# CS29003 Algorithms Laboratory

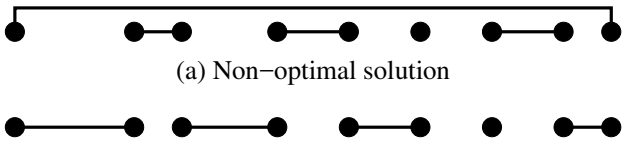
## Assignment No: 5

Last date of submission: 13–February–2018

---

The king of Prioquesia has built $N$ foonerators $F_0, F_1, F_2, \ldots, F_{N-1}$ arranged in a straight line (in fixed positions). The distance between $F_i$ and $F_{i+1}$ is $d_i$ for $i = 0, 1, 2, \ldots, N-2$. The king also has $K$ pairs of jacks. Each pair of jacks is meant for connecting a pair of foonerators. One foonerator can hold only one jack, so each foonerator can be connected to at most another foonerator. The connections are to be made by costly titanium bars. So the king intends to choose those $K$ pairs of foonerators such that the total connection length is as small as possible. Assume that $K < N/2$, and that all distances $d_i$ are positive integers.

A greedy algorithm works well for solving the king's problem. The first insight is that we can choose only consecutive pairs (this is feasible since $K < N/2$). In order to see why, suppose that some solution connects two non-consecutive foonerators $F_i$ and $F_j$ with $j \geqslant i+2$. If there exist two consecutive foonerators $F_l$ and $F_{l+1}$ with $i < l < l+1 < j$ such that neither $F_l$ nor $F_{l+1}$ is connected to a foonerator, then replacing the connection between $F_i$ and $F_j$ by the connection between $F_l$ and $F_{l+1}$ improves the solution. If no such $l$ exists, then we have a situation similar to what is illustrated in the following figure. In this case too, the cost can be improved by choosing consecutive foonerator pairs only.



(a) Non−optimal solution



(b) Cost improvement

An obvious greedy strategy now is to choose consecutive pairs in an increasing order of separation. A heap (more correctly, a priority queue) helps here. There is no need to sort the array $D = (d_0, d_1, d_2, \ldots, d_{N-2})$ of distances. Each node in the heap should store the left endpoint $l$ of a connection, the right endpoint $r$ of the connection, and the cost $c$ of the connection. For a reason to be explained in Part 3, storing $r$ is necessary, and $c$ is not necessarily the actual distance between the two endpoints. The heap ordering is with respect to the costs. Since we extract minimum-cost connections first, the heap should be a min-heap.

### Part 1: Heap functions

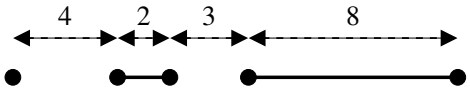
As discussed above, the heap comprises nodes of triples $(l, r, c)$, and the heap ordering is with respect to $c$. Write the min-heap functions **`heapify`**, **`makeheap`**, **`insert`** and **`deletemin`** for this heap. Write your own functions. Do not make any built-in library calls, even if available.

### Part 2: A greedy algorithm

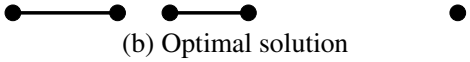
Initialize your heap array with the $N-1$ triples $(i, i+1, d_i)$ for $i = 0, 1, 2, \ldots, N-2$. Build a min-heap from the array. Now, repeat the following steps until $K$ connections are added. Let the entry stored at the root of the heap be $(l, r, c)$. If neither $l$ nor $r$ is so far connected to another foonerator, add the connection $(l, r)$. Irrespective of whether you add the connection or not, make a deletemin from the heap. Write a function **`greedy`** to implement this strategy. The function should print the connections added, and the total cost.

### Part 3: The greedy algorithm with error correction

The algorithm of Part 2 is not guaranteed to give an optimal solution, as demonstrated below ($N = 5$, $K = 2$).

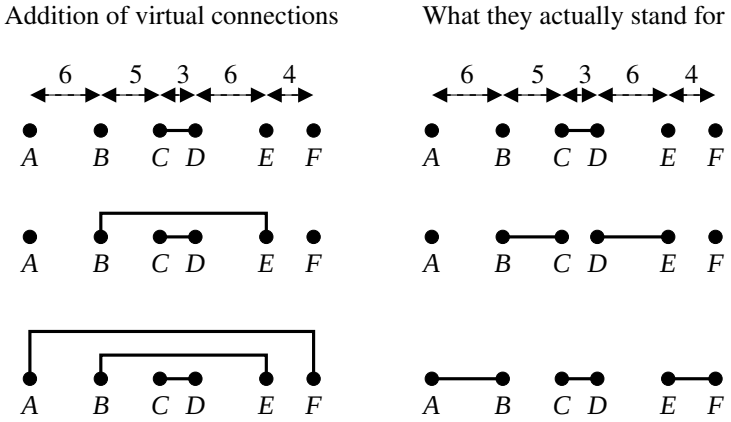



(a) Greedy solution



(b) Optimal solution

This problem can however be handled using a *future error-correction* strategy. The following figure explains the idea. Suppose that $A, B, C, D, E, F$ are consecutive foonerators with their separations as shown.




Here, $CD$ is the least-cost consecutive pair, and so is selected by the greedy algorithm if $CD$ appears at some time at the root of the min-heap. Suppose that at that point of time neither of the six foonerators of the figure is connected. The algorithm adds the connection $CD$. We have $\text{cost}(CD) = 3$ (this is a real connection). But in future it may so happen that deleting $CD$ and adding $BC$ and $EF$ leads to a better cost. This situation can be captured by the virtual connection $BE$. Adding both $CD$ and $BE$ essentially means adding $BC$ and $DE$. Thus the cost of $BE$ is $\text{cost}(BC) + \text{cost}(DE) - \text{cost}(CD) = 5 + 6 - 3 = 8$.

Now, suppose that $CD$ and $BE$ are already added, and we want to add $AF$. This means deleting $BE$ followed by adding $AB$ and $EF$. We now have $\text{cost}(AF) + \text{cost}(BE) + \text{cost}(CD) = \text{cost}(AB) + \text{cost}(CD) + \text{cost}(EF)$, that is, $\text{cost}(AF) = \text{cost}(AB) + \text{cost}(EF) - \text{cost}(BE) = 6 + 4 - 8 = 2$. These virtual connections rectify any error that the greedy choices make.

Write a function **`greedyec`** to implement the corrected greedy algorithm. The function starts with an initial heap built from the $N-1$ triples $(i, i+1, d_i)$ for $i = 0, 1, 2, \ldots, N-2$. These triples correspond to the real connections. Repeat until $K$ connections (real and/or virtual) are added. Let the root of the heap store $(l, r, c)$. We have $r = l + 1$ if this corresponds to a real connection, or $r > l + 1$ if this corresponds to a virtual connection. Make a deletemin in the heap. Now, check whether $l$ and $r$ are unconnected. If so, add the connection. If the connection is added, look at the virtual connection $(l-1, r+1)$. If $l - 1 \geqslant 0$ and $r + 1 \leqslant N - 1$, this is a valid connection. Moreover, if $l - 1$ and $r + 1$ too are unconnected, it is admissible to add the virtual connection $(l-1, r+1)$. The cost of this connection is $d_{l-1} + d_r - c$. Each insert is preceded by a deletemin, so the size of the heap never exceeds $N - 1$.

When $K$ connections (both the real and the virtual ones) are added, make final adjustments to identify the real connections that the collection of added connections stands for. Print these final connections, and the total cost.

**The *main*() function**

- Read $N$ and $K$ from the user.
- Read the $N-1$ distances $d_0, d_1, d_2, \ldots, d_{N-2}$ in an array $D$.
- Call **`greedy`** on $N, K, D$ to obtain the greedy solution.
- Call **`greedyec`** on $N, K, D$ to obtain the optimal solution achieved by the greedy algorithm with error correction.

---

Submit a single C/C++ source file. Do not use global/static variables.

**Sample output**

```
    N = 100
    K = 40
    12 17  6  6  6 11 16 13 20 15 19 11 12 20 16 12 14  6  9  5  8  9 15 13 13
    18  7  8 19  8 20 10  5  5 12  6 12  7 15 11 17 13 17  8 13 12 15  6 13 19
     6 16  8 16  9 16 14 11 20 12 14 19 17 14 20  8 16 11 10 10 17  6 18 13  9
    10 20 19 11 13 18 12  8  5  8 12 16 17 18 15  8 12 14 20  5 13  8 16 19

*** Part 1: Greedy algorithm
--- Adding connections
    (32,33): 5 (19,20): 5 (83,84): 5 (94,95): 5 ( 3, 4): 6 (17,18): 6
    (35,36): 6 (71,72): 6 (47,48): 6 (50,51): 6 (37,38): 7 (26,27): 7
    (65,66): 8 (43,44): 8 (90,91): 8 (96,97): 8 (52,53): 8 (29,30): 8
    (74,75): 9 (21,22): 9 (54,55): 9 (68,69):10 (78,79):11 (39,40):11
    (11,12):11 ( 5, 6):11 (57,58):11 ( 0, 1):12 (15,16):12 (81,82):12
    (85,86):12 (45,46):12 (59,60):12 ( 7, 8):13 (23,24):13 (41,42):13
    (63,64):14 (92,93):14 ( 9,10):15 (87,88):17
    Total cost = 381

*** Part 2: Greedy algorithm with error correction
--- Adding connections
    ( 0, 1):12 ( 2, 3): 6 ( 4, 5): 6 ( 7, 8):13 (11,12):11 (15,16):12
    (17,18): 6 (19,20): 5 (21,22): 9 (23,24):13 (26,27): 7 (29,30): 8
    (31,32):10 (33,34): 5 (35,36): 6 (37,38): 7 (39,40):11 (41,42):13
    (43,44): 8 (45,46):12 (47,48): 6 (50,51): 6 (52,53): 8 (54,55): 9
    (57,58):11 (59,60):12 (63,64):14 (65,66): 8 (67,68):11 (69,70):10
    (71,72): 6 (74,75): 9 (78,79):11 (81,82):12 (83,84): 5 (85,86):12
    (90,91): 8 (92,93):14 (94,95): 5 (96,97): 8
    Total cost = 365
```