ESO207A: Data Structures and Algorithms	*July-Nov 2017*
Final Exam	*20th November, 2017*

Instructions.

a. Please answer all parts of a question together.

b. The exam is closed-book and closed-notes. Collaboration of any kind is ***not*** permitted. Cell phones on person are not allowed. Please use the mounted clock in the rear of the examination hall for time reference.

c. You may cite and use algorithms and their complexity as done in the class.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms or conclusions.

Problem 1. *Short Answers* $(2.5 \times 6 = 15)$

Write down your recommendation for the data structure to use, that in each of the following cases, supports the given set of operations on a dynamic set S. Support your answer with (time complexity) reasons. Every member of S has a numeric (or alphanumeric) *key* attribute, in addition to perhaps other attributes. Here, *ptr* is a pointer to an element of the data structure containing a member of S. Explain if probabilistic or deterministic guarantee requirements will change your choice?

(a) *Insert*(S, x), *DeleteMin*(S), *DecreaseKey*(x, key) and *FindMin*(S).

Solution.. This is the Min Priority Queue and can be implemented as a min-Heap. *Insert*, *DeleteMin* and *DecreaseKey* all take $O(\log n)$ time, where, $n = |S|$.

(b) *Insert*(S, x), *Delete*(S, ptr), *Search*(S, key).

Solution.. Usually, the best choice would be a hash table with a hash function drawn from a universal family of hash functions. Assuming that the hash function is computed in time $O(1)$, the time complexity *Insert* is $O(1)$, *Delete* is $O(1)$ (by maintaining doubly linked chains instead of singly linked ones), and *Search* takes expected time $O(1)$.

(c) *Insert*(S, x), *Delete*(S, ptr), *Search*(S, key), *Successor*(S, ptr).

Solution.. The best choice would be a balanced or nearly-balanced binary search tree structure, like red-black trees, where, all the above operations would be done in time $O(\log n)$, where, $n = |S|$.

For each of the following statements, either prove it or give a counterexample.

(a) Let e be any edge of minimum weight in G (there may be multiple edges with minimum weight). Then e must be part of *some* MST.

Solution.. *Proof 1.* Let $e = \{a, b\}$ and consider the cut $(\{a\}, V - \{a\})$. Then, e is a lightest edge crossing this cut (since globally, it is a lightest edge). By the cut-property (let $X = \phi$) it is part of some MST.

Proof 2. Order the edges in the order of non-decreasing weight so that e is the first one. This is always possible. The first edge is always chosen by Kruskal's algorithm. So there is an MST, the one found by Kruskal, that contains e.

(b) Let e be any edge of minimum weight in G (there may be multiple edges with minimum weight). Then e must be part of *every* MST.

Solution.. This is false. See Figure 1.

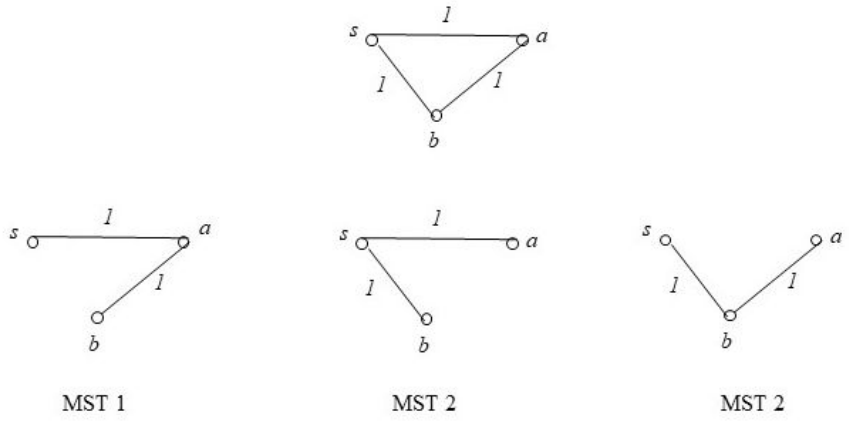


Figure 1: A (jointly) minimum weight edge in a graph may not be part of an MST.

(c) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

Solution.. This is false. See Figure 2.

Problem 2. *Graphs* (10)
Given an undirected tree $T = (V, E)$ in adjacency list format and a designated root vertex $r \in V$. You have to *preprocess* the tree so that queries of the form "given vertices u and v in the rooted tree, is u an ancestor of v?" can be answered in $O(1)$ time. The preprocessing itself should take linear time. *Note:* In a rooted tree with root r, u is an ancestor of v iff the unique path from r to v passes through u.

Solution.. Perform a *DFS* from the root vertex r and store $u.d$ and $u.f$ for every vertex $u \in V$. Now to check if u is an ancestor of v, check if the interval $[v.d, v.f]$ is fully contained in the interval $[u.d, u.f]$.

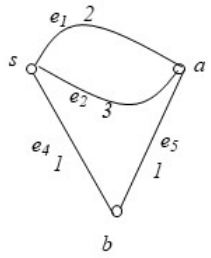
Proof: Given that T is a rooted tree with root r, it is the DFS tree obtained by doing DFS on T from r. The correctness follows from the parenthesis property.

Problem 3. *Graphs* Give a linear time algorithm to find an odd-length cycle in a strongly connected directed graph, or say that no such odd-length cycle exists. (10)

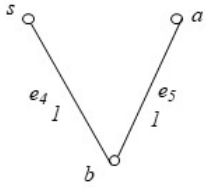
Problem 4. *Modular Arithmetic* Given a positive number $n > 1$ and numbers a, b and c, design an efficient algorithm that finds a solution x to the equation $ax + b = c \mod n$, if one exists and returns *NO SOLN* if no such x exists. Analyze its complexity. (10)

Solution.. *Notation* We use the notation $x|y$ to denote that x divides y. Note that there is a solution to $ax = e \mod n$ iff $\gcd(a, n)|e$. *Proof:* Suppose $\gcd(a, n) = g$ and $g|e$. Then, there exist integers x, y

(a)

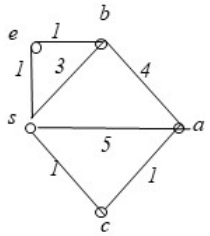


Graph G

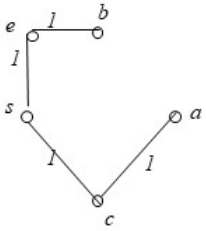


MST of weight 2

The cycle e_1 e_2 has unique smallest edge e_1. But e_1 is not part of any MST.



Graph G



MST of weight 4

The cycle s-a-b has the unique lightest edge s-b. But this graph has a unique MST of weight 4 that does not include this edge s-b.

Figure 2: A unique lightest edge of a cycle may not be part of *any* MST. Two examples are given. In Figure (a), the graph is not a simple graph, that is there are multiple edges between edges. This allows for a simpler example. In Figure (b), the graph is a simple graph.

such that $ax' + ny' = g$ and $e = gd$. Multiplying $ax' + ny' = g$ both sides by d, we get the equation

$$adx' + ndy' = e$$

Taking $\mod n$ both sides, we get

$$ax = e \mod n$$

where, $x = dx'$. This also gives the algorithm for finding x.

Conversely, suppose $g \not| e$. Let $A = \{ax' + ny' \mid x', y' \in \mathbb{Z}\}$. As proved in the class, every member of A is a multiple of $\gcd(a, n)$ (the proof is repeated below). Since $g \not| e$, $e \notin A$. Hence, there is no solution of the form $ax' + ny' = e$. On the contrary, if there was a solution to the equation $ax = e \mod n$, then there is a y such that $ax + ny = e$ (i.e., $n|ax - e$), which would be a contradiction. Hence there is no solution to $ax = e \mod n$.

So, we can now design a simple algorithm for solving $ax + b = c \mod n$. Let $e = c - b$. Use Extended Euclid's algorithm to find x', y' and $g = \gcd(a, n)$ such that $ax' + ny' = g$. Now check if $g|e$. If so, let $e = dg$ and a solution for $ax = e \mod n$ is given by $x = d \cdot x' \mod n$. If $g \not| e$, then there is no solution, as proved above and the algorithm reports *NO SOLN*.

The complexity is one call to Extended-Euclid algorithm followed by a constant number of checks of the form whether $g|e$ etc., which can be done in linear time (i.e., $O(\log |g| + \log |e|)$, etc.. The beauty of Extended Euclid's algorithm running on a, n is that in two steps, both arguments reduce in absolute value by at least 1/2 and remain an integer. Hence, the Extended Euclid algorithm runs for at most $O(2 \log(\min, |a|, |n|)$ iterations. In each iteration, some multiplications etc., are done, which can again be done in at most linear time. Overall, the complexity is $O(\log^2 \max(|a|, |n|)$, that is it is efficient.

Just for completeness' sake, here is a proof (Euler?) that for any two numbers a and b, the set $A = \{ax + by \mid x, y \in \mathbb{Z}\}$ is the set of all multiples of $g = \gcd(a, b)$. First note that if d is a common divisor of a and b, then $d|A$, that is d divides all the elements in A. Hence, $g|A$, since g is the gcd. Now consider the smallest positive number in A and call it h. Suppose we divide a by h, to get say, $a = qh + r$. Now, r is either 0 or $r > 0$. If $r > 0$, then, $r < h$. But since, $r = a - qh$, and h is an integer linear combination of a and b, so is r. So $r \in A$, and $r > 0$ being less than h contradicts the fact that h was the smallest positive integer of A. Hence, r must be 0 and $a = qh$ or $h|a$. By a similar argument, $h|b$. Hence, h is a common divisor. We know that $g|A$ and therefore, $g|h$. Hence, $h = g = gcd(a, b)$.

We have shown that every element of A is some multiple of g. If $w = dg$ is some multiple of g, then we just note that $g = ax' + by'$ (since it belongs to A) and therefore, $w = dg = a(dx') + b(dy')$, which means $w \in A$.

Problem 5. A *feedback edge set* of an undirected graph $G = (V, E)$ is a subset of edges $E' \subset E$ that intersects every cycle of the graph (i.e., for every cycle $C \subset E$, $C \cap E' \neq \phi$). Thus removing the edges E' will render the graph acyclic. Give an efficient algorithm for the following problem. (10)

Input: Undirected graph $G = (V, E)$ with positive edge weights w_e.

Output: A feedback edge set $E' \subset E$ of minimum total weight $\sum_{e \in E'} w_e$.

Solution.. Let $G = (V, E)$ be the given undirected connected graph. Let F be a feedback edge set and $T = E - F$ be the remaining set of edges. Then, T is a spanning tree. Hence, one way to find the minimum weight feedback edge set of a connected undirected graph is to find the maximum weight spanning tree T and return the complement of its edges $E - T$.

Finding the max. weight spanning tree is simple, reverse signs on all edge weights and compute the minimum spanning tree.

One simple algorithm would be to use DFS to find the connected components of G and then use the above algorithm to find the maximum spanning tree of each connected component (using Prim or Kruskal) and take the union of the edges in the complement of the maximum spanning tree.

Alternately, Kruskal's algorithm can be used first, since Kruskal identifies all the connected components and gives a minimum spanning tree (or with signs of edge weights reversed, this would be the maximum spanning tree) for each component. The complement of what is returned by this version of Kruskal is the desired minimum weight feedback edge set.

Problem 6. *Divide and Conquer.* (15)
Let $A[1, \ldots, n]$ be an array of n distinct integers. The number of inversions in A is the number of pairs (i, j) such that $1 \leq i < j \leq n$ and $A[i] > A[j]$. For example, the number of inversions in the array $\boxed{4\ 3\ 2\ 5}$ is 3 corresponding to the "out-of-order" numbers $(4, 3)$, $(4, 2)$ and $(3, 2)$. Given an $O(n \log n)$ time algorithm for counting the number of inversions of a given array A. (*Note*: A $\Theta(n^2)$ algorithm will get only 3 marks.)

Solution.. The key to an efficient solution is to additionally also sort A using the classical divide-and-conquer merge-sort kind of method and to count the number of inversions in the process. Consider the array $\boxed{4\ 3\ 5\ 2\ 1\ 7\ 6\ 0}$ as a running example for the divide and conquer method. The division part is simple: We split the array across the middle. Now we do two things: We (recursively) sort each half **and** count the number of inversions in each half.

1. *Divide step*: $\boxed{4\ 3\ 5\ 2}\ \boxed{1\ 7\ 6\ 0}$
2. *Conquer Step:*
   (a) Sorting $\boxed{4\ 3\ 5\ 2}$ gives $\boxed{2\ 3\ 4\ 5}$.
   The inversions in $\boxed{4\ 3\ 5\ 2}$ are $(4, 3)(4, 2)(3, 2)(5, 2)$ for a total of 4.
   (b) Sorting the second half $\boxed{1\ 7\ 6\ 0}$ gives $\boxed{0\ 1\ 6\ 7}$.
   The inversions in $\boxed{1\ 7\ 6\ 0}$ are $(1, 0)(7, 6), (7, 0), (6, 0)$ for a total of 4.
   (c) So the recursive steps have given us two halves sorted and an inversion count of $4 + 4 = 8$. Output (from recursive step) is

$$\boxed{4\ 3\ 5\ 2}\boxed{1\ 7\ 6\ 0}, \qquad \text{inversion count} = 8$$

3. *Combine Step:* We have to combine the two partial solutions of the halves into a solution to the full instance. We will *merge* the sorted halves to get the sorted list (this is easy!) and fix the inversion count. The problem with the inversion counts so far is that we have not counted the contribution from *cross* halves. Namely an inversion that contributes to the cross count is of the form (a, b), where, $a > b$ and a is in the first half and b is in the second half.

   For example, 1 in the second half should contribute 4 to the inversion cross-count, as there are 4 items in the first half all smaller than it. the 7 and 6 in the second half contribute 0 to the inversion cross-count. Similarly, 0 contributes 4 to the inversion cross count.

   Let us decide to count the number of cross-inversions of the form (a, b) by counting the number

of cross-inversions in which b participates, for each b in the second half. That is,

$$\begin{aligned}\text{Number of cross inversions} &= \sum_{b \text{ in second half}} |\{(a,b) \mid a \in \text{ first half and } a > b\}| \\ &= \sum_{b \text{ in second half}} \text{inv}(b)\end{aligned}$$

So our job is simply to count inv(b), for each b in the second half. Let us design this into the combine step in more detail. Say we are merging the two halves. Consider the first instance when b from the second half when being compared with some a from the first half is found to be $< a$. Since the left half is sorted, all the elements to the right of a are also larger than b. So the inversion count inv(b) is the number of elements in the first list. We keep adding this to a running count, that would give the cross-inversion count. The algorithm is given below. We will use the book's idea of using sentinels (∞ values inserted) to reduce pseudo-code length. The top-level call is CNT_INVERSIONS$(A, 1, n)$.

```
CNT_INVERSIONS(A, p, r)
1.  if p == r return 0
2.  else
3.      q = ⌊(p + r)/2⌋
4.      i1 = CNT_INVERSIONS(A, p, q)
5.      i2 = CNT_INVERSIONS(A, q + 1, r)
6.      ic = MERGE_CNT_CROSS_INV(A, p, q, r)
7.      return i1 + i2 + ic
```

```
MERGE_CNT_CROSS_INV(A, p, q, r)
1.  Copy A[p...q] to L[1...q − p + 1]
2.  Copy A[q + 1...r] to R[1...r − q]
3.  n1 = q − p + 1, n2 = r − q
4.  L[q − p + 2] = ∞, R[r − q + 1] = ∞
5.  //Now merge and count inversions
6.  cross_inv = 0, i = 1, j = 1
7.  for  k = p to r
8.      if L[i] ≤ R[j]
9.          A[k] = L[i]
10.         i = i + 1
11.     else // L[i] > R[j]
12.         A[k] = R[j]
13.         j = j + 1
14.         cross_inv = cross_inv + n1 − i + 1
15. return cross_inv
```

Problem 7. (15)

A server has n customers labeled 1 through n to be served. The service time required to serve the ith customer is t_i minutes and is known in advance. The server serves only one customer at a time (no

parallelism). If the customers are served in the order $i_1, i_2, \ldots, i_n$, then, the waiting time for customer i_1 is t_{i_1}, the waiting time for i_2 is $t_{i_1} + t_{i_2}$, and so on, the waiting time for i_n is $t_{i_1} + \ldots + t_{i_n}$. We wish to minimize the *total waiting time*

$$T = \sum_{i=1}^{n} (\text{ time spent waiting by customer } i \text{ }) \ .$$

Give an efficient algorithm for computing the optimal order in which to process the customers. *Marks Distribution:* Desired algorithm has time $O(n \log n)$. Design and Argument (11), Algorithm, complexity (4). Other solutions will be marked according to correctness arguments, pseudo-code and complexity.

Solution.. The optimal order is obtained by the *shortest service time first* rule. Sort the customers in non-decreasing order of service times and process the customers from the head of this list in this order. This is a greedy paradigm: Of all the customers left to service, choose the one with the smallest service time. Time taken is the time to sort: $O(n \log n)$.

Proof of Correctness of Greedy strategy. Let $c_1, c_2, \ldots, c_n$ be a permutation of $1, 2, \ldots, n$ that gives an optimal order that minimizes total waiting time. If $t_{c_1} \leq t_{c_2} \leq \ldots \leq t_{c_n}$, then we are done. Otherwise, there is some i such that $t_{c_i} > t_{c_{i+1}}$. Consider the permutation c' that orders all customers identically as c except that the positions of c_i and c_{i+1} are interchanged. That is,

$$c'_j = \begin{cases} c_j & \text{if } j \in \{1, \ldots, n\} - \{i, i+1\} \\ c_{i+1} & \text{if } j = i \\ c_i & \text{if } j = i+1 \end{cases}$$

Let $T(c)$ and $T(c')$ denote the total waiting times for the permutations c and c' respectively. How are $T(c)$ and $T(c')$ different? Note that the waiting time for any customer $c_1, \ldots, c_{i-1}$ remains the same under c' (which is $c_1 + \ldots + c_j$, for $1 \leq j \leq i-1$. Now for $j > i+1$, once again the waiting time of any customer $c_{i+2}, \ldots, c_n$ is the same in c as under c', and equals $c_1 + \ldots + c_j$, for $j = i+2, \ldots, n$. So the only two waiting times that are altered are the customers in the ith and $i+1$th position in c and whose positions are reversed in c'. Therefore

$$\begin{aligned} T(c) - T(c') = & \text{ (Waiting time for customer } c_i \text{ under } c) + \text{ (Waiting time for customer } c_{i+1} \text{ under } c) \\ & - \big(\text{ (Waiting time for customer } c_i \text{ under } c') + \text{ (Waiting time for customer } c_{i+1} \text{ under } c') \big) \end{aligned}$$

Consider c. What is the waiting time for customer c_i under ordering c? It is $\sum_{k=1}^{i} t_{c_k}$. And what is the waiting time for the customer c_{i+1} under ordering c? It is $\sum_{k=1}^{i+1} t_{c_k}$. Total waiting time for customers c_i and c_{i+1} under c is

$$\sum_{k=1}^{i} t_{c_k} + \sum_{k=1}^{i+1} t_{c_k} = 2 \sum_{i=1}^{i-1} t_{c_k} + 2t_{c_i} + t_{c_{i+1}}$$

Now consider c'. The waiting time for customer c_{i+1} under c' is $\sum_{k=1}^{i-1} t_{c_k} + c_{i+1}$. The waiting time for customer c_i under c' is $\sum_{k=1}^{i-1} t_{c_k} + c_{i+1} + c_i$. Adding, the sum of waiting time for customers c_i and c_{i+1} under the ordering c' is

$$2 \sum_{k=1}^{i-1} t_{c_k} + 2c_{i+1} + c_i$$

Therefore,

$$\begin{aligned} T(c) - T(c') &= 2t_{c_i} + t_{c_{i+1}} - \left(2t_{c_{i+1}} + t_{c_i}\right) \\ &= t_{c_i} - t_{c_{i+1}} \\ &> 0 \end{aligned}$$

which is a contradiction, since we assumed that $T(c)$ is the optimal total waiting time over all permutations. The contradiction arose because we assume that there was an out of order pair c_i, c_{i+1} such that $t_{c_i} > t_{c_{i+1}}$. This proves that in the optimal ordering, for every i, $t_{c_i} \leq t_{c_{i+1}}$, that is, the customers should be ordered in increasing (non-decreasing) order of their service times. In other words, the greedy paradigm gives an optimal sequence.

Problem 8. *Dynamic Programming* (15)
Funfone is considering opening a series of base stations along a certain national highway, which is modeled as a straight line. There are n towns on this highway at kilometer numbers $k_1, k_2, \ldots, k_n$. The constraints are as follows.

- In each town, Funfone may open at most one base station. The profit from opening a base station at location i is p_i, where, $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two base stations should be at least r kms apart.

If $i_1, i_2, \ldots, i_l$ are the set of designated locations for opening base stations and this set of locations is feasible (i.e., any two are at least r kms. apart), then the total profit is the sum of individual profits, that is, *total profit* $= p_{i_1} + p_{i_2} + \ldots + p_{i_l}$.
Give an efficient algorithm to compute the maximum total profit subject to the given constraints. *Marks Distribution*: Modeling the problem (2), Explaining and setting up the recurrence (7), Pseudo-code and correctness (4), Complexity Analysis (2).

Solution.. The towns are named 1 through n. For $1 \leq i \leq j \leq n$, let $P[i, j]$ denote the maximum total profit from opening base stations in the segment of the towns i through j (end points inclusive). The base case arises when $i = j$. In this case, a base station must be set up at town i and profit is p_i. That is,

$$\begin{aligned} P[i, i] &= p_i, \quad i = 1, 2, \ldots, n \\ P[j, i] &= 0, \quad \text{if } j > i \end{aligned}$$

Also, the range $P[i, j]$, for $i > j$ is not meaningful and has profit 0. We will extend the domains 1 to n to include artificial towns 0 and $n + 1$.

We now consider the recurrence equation. Suppose we wish to find $P[i, j]$. Let t be a town such that $i \leq t \leq j$. Suppose we open a base station at t. This gives immediately a profit of p_t. Further, no base station is feasible within 50 km on either side of town l. Let $l \geq i$ be the rightmost town that is at least 50 km left from t, denote $l =$ *ClosestLeft*$(t, 50)$ and let $r \leq j$ be the leftmost town that is at least 50 km right from t, denote $r =$ *ClosestRight*$(t, 50)$. However, if the distance between towns i and t is less than or equal to 50km, then, $l = i - 1$, and if the distance between the towns t and j is less than or equal to 50km, then r is set to $j + 1$. That is, change the definition of *ClosestLeft*$(t, 50)$ to include i as a parameter and define it instead as *ClosestLeft*$(t, 50, i)$. Similarly, for *ClosestRight*$(t, 50, j)$.

With this choice of t, the problem $P(i, j)$ divides itself into (a) base-station at t, and the subproblems $P(i, l)$ and $P(r, j)$, which must be solved optimally. Now t can vary from i to j inclusive. So we get the

recurrence relation for the case $1 \leq i < j \leq n$.

$$P[i,j] = \max_{i \leq t \leq j} p_t + P[i,l] + P[r,j], \text{ where, } l = \textit{ClosestLeft}(t, 50, i) \text{ and } r = \textit{ClosestRight}(t, 50, i)$$

since we are looking for the maximum total profit.

Suppose the optimal value of the recurrence relation for $P[i,j]$ occurs at t_0 and l_0 and r_0 are the corresponding closest left and closest right town indices. Then, $B[i,j] = (t_0, l_0, r_0)$ and is used to reconstruct the optimal solution.

Let $s = j - i$. The recurrence has to be computed in increasing order of $s = j - i$, from 0 to $n - 1$.

```
FUNFONE(p, k, P, B) // p is the input profit array p[0...n+1]. k is the kilometer marker array k[0...,n+1].
// Assume p[0] = p[n+1] = 0, k[0] = -∞ and k[n+1] = ∞.
// P[1...n, 1...n] is the optimal cost for recurrence equation as explained.
// B[1...n, 1...n] reconstructs an optimal solution.
1.  for   j = 0 to n + 1
2.      for   i = 0 to j
3.          if  i == j
4.              P[i, j] = p_i
5.              B[i, j] = (i, i − 1, i + 1)
6.          else P[i, j] = 0
7.  for  s = 1 to n − 1
8.      for  i = 1 to n − s
9.          j = i + s
10.         P[i, j] = 0
11.         for   t = i to j
12.             l = ClosestLeft(t, 50, i)
13.             r = ClosestRight(t, 50, j)
14.             cost = p[t] + P[i, l] + P[i, r]
15.             if  cost > P[i, j]
16.                 P[i, j] = cost
17.                 B[i, j] = (t, l, r)
```

Time complexity of loops is $O(n^3)\times$ complexity of finding *ClosestLeft* and *ClosestRight*. By using an augmented RB-tree for the k array, each such query can be solved in time $O(\log n)$ after pre-processing it in time $O(n \log n)$. Thus the overall time complexity is dominated by $O(n^3 \log n)$.

The final solution can be reconstructed from the B array as follows. The top level call is $B(1, n)$.

```
Reconstruct_Opt_Soln(B, i, j)
1.  if i ≤ j
2.      print "Open base-station at B[i, j].t"
3.      l = B[i, j].l
4.      r = B[i, j].r
5.      if l ≥ i
6.          Reconstruct_Opt_Soln(B, i, l)
7.      if r ≤ t
```

8. $Reconstruct\_Opt\_Soln(B, r, j)$