Solution for NOI 2017

March 14, 2017

Introduction

In this year, the scientific committee includes 7 members:

- Bernard Teo *btzy1996@hotmail.com*
- Ranald Lam *ranaldmiao@gmail.com*
- Wei-Liang Gan *ganweiliang@gmail.com*
- Mark Theng *krawthekrow@gmail.com*
- Frank Stephan *fstephan@comp.nus.edu.sg*
- Steven Halim *stevenha@comp.nus.edu.sg*
- Wing-Kin Sung *ksung@comp.nus.edu.sg*

There are 5 tasks in NOI 2017.

- Task 1: Best Place
- Task 2: Roadside Advertisements
- Task 3: Hotspot
- Task 4: RMQ
- Taks 5: I want to be the very best too!

Below, we will discuss the solutions for the 5 tasks.

Task 1: Best Place

Author: Ranald Lam
Singapore IOI Team 2012 - 2014, Raffles Institution Alumni
ranaldmiao@gmail.com

The abridged problem: Given a list of N co-ordinates on a 2-dimensional plane, find the co-ordinate (Ex, Ey) where the sum of Manhattan distances from (Ex, Ey) to all the N co-ordinates is minimized. If there are multiple co-ordinates that minimizes the sum of Manhattan distances, output any one of these co-ordinates.

Subtask 1

In Subtask 1, there are a total of $N = 2$ points. One possible co-ordinate that minimized the sum of Manhattan distances is to output the co-ordinates of either one of these 2 points.

Subtask 2

In Subtask 2, there are now $N = 1000$ points. However, all $Y_i = 0$ and $0 \leq X_i \leq 1\,000$. Ignoring the Y co-ordinates, this reduces the problem to a list of N co-ordinates on a line (1-dimension). As the X values are limited to between 0 and 1 000, the co-ordinate $(Ex, 0)$ will also be bounded to between 0 and 1 000.
For each possible value of $(Ex, 0)$ we compute the sum of Manhattan distances in $O(N)$ and then output the co-ordinate with the minimum sum of Manhattan distances.

A sample C++ code for Subtask 2 is provided as below:

```
int N, X[1000], Y[1000];
cin >> N;
for (int i = 0; i < N; ++i)
    cin >> X[i] >> Y[i];

int best = INT_MAX, ans = 0;
for (int i = 0; i <= 1000; ++i) {
    int sum = 0;
    for (int k = 0; k < N; ++k) {
        if (X[k] > i) sum += X[k] - i;
        if (X[k] < i) sum += i - X[k];z
    }
    if (sum < best) {
        best = sum;
        ans = i;
    }
}
cout << ans << " " << 0 << endl;
```

Subtask 3

In Subtask 3, $N \leq 100\,000$ and $0 \leq X_i \leq 10^9$. However, $Y_i = 0$ still holds and the problem is also reduced to a list of N co-ordinates on a line (1-dimension). We make an important observation for the case on the line. For a candidate X co-ordinate Ex, if there are more co-ordinates on the left (i.e. $X_i < Ex$) than co-ordinates on the right (i.e. $X_i > x$), then shifting X to $X - 1$ will get a lower sum of Manhattan distances.
To illustrate this, observe the following distribution. Our candidate co-ordinate for Ex is currently at position 5.

```
Position     :   0 -- 1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8 -- 9
Points       :   x -- x -- x -- - -- - -- C -- x -- x -- - -- -
                                          ^
```

At $Ex = 5$, the sum of Manhattan distances for all N co-ordinates to $(Ex, 0)$ is 15. As there are 3 co-ordinates to the left of C, and 2 points to the right of C, the sum of Manhattan distances will decrease by $1(3 - 2)$ by moving

one step to the left. At $Ex = 4$, the sum of Manhattan distances for all N co-ordinates to $(Ex, 0)$ is 14. This can be repeated until the number of co-ordinates on the left equals to the number of co-ordinates on the right.
In order to implement this, we can make another important observation. The observation is that the point where the number of co-ordinates on the left equals to the number of co-ordinates on the right, is at the median point. Hence, we can take the X co-ordinates of all the N co-ordinates and compute the median X co-ordinate. To do so, we can place the X co-ordinates in an array, sort it in ascending order and find the X co-ordinates at index $N/2$.

A sample C++ code for Subtask 3 is provided as below:

```
const int MAXN = 100000;
int N, X[MAXN], Y[MAXN];
cin >> N;
for (int i = 0; i < N; ++i) {
    cin >> X[i] >> Y[i];
}
sort(X, X+N);
cout << X[N/2] << " 0" << endl;
```

This will allow us to solve Subtask 3 in time compleixty $O(N \log N)$.

Subtask 4

In Subtask 4, we can use the same idea as Subtask 2. As $0 \leq X_i \leq 100$ and $0 \leq Y_i \leq 100$, there are only 10 000 possible points for (Ex, Ey). For each candidate point, we run a $O(N)$ loop to obtain the sum of Manhattan distances.
As with Subtask 2, we will output the candidate point with the lowest sum of Manhattan distance to the N co-ordinates. This solution runs in $O(NXY)$ where X is the range of values for X_i and Y is the range of values for Y_i.

Subtask 5

In Subtask 5, we observe that sum of Manhattan distances in 2 dimensions can be reduced to sum of 2 distances in 1. This is because the difference in X co-ordinates can be calculated independently of the difference in Y co-ordinates. As such, we can find Ex and Ey independently.

In order to find Ex and Ey, Subtask 5 allows for an $O(N^2)$ computation. We can make another observation that the Ex that minimises the sum of Manhattan distances can be one of the N X_i co-ordinates. Hence, we can narrow down our candidate point from 10^9 to the N X_i co-ordinates. We then compute the sum of Manhattan distances in X co-ordinate using an $O(N)$ loop and keep track of the point with the minimum sum of Manhattan distances.
As we have established we can compute X co-ordinates and Y co-ordinates independently, we will repeat this procedure to find Ey using the N Y_i co-ordinates.

Sample C++ code for Subtask 5:

```
const int MAXN = 100000;
int N, X[MAXN], Y[MAXN];
cin >> N;
for (int i = 0; i < N; ++i) {
    cin >> X[i] >> Y[i];
}
long long best = LLONG_MAX;
int Xans = 0;
for (int i = 0; i < N; ++i) {
    long long sum = 0;
    for (int j = 0; j < N; ++j) {
        sum += abs(X[i] - X[j]);
    }
    if (sum < best) {
        best = sum;
        Xans = X[i];
    }
}
best = LLONG_MAX;
int Yans = 0;
for (int i = 0; i < N; ++i) {
    long long sum = 0;
    for (int j = 0; j < N; ++j) {
        sum += abs(Y[i] - Y[j]);
    }
    if (sum < best) {
        best = sum;
        Yans = Y[i];
    }
}
cout << Xans << " " << Yans << endl;
```

Subtask 6

In Subtask 5, we have observed that we can find Ex and Ey independently. In Subtask 3, we have illustrated a method to find Ex in $O(N \log N)$ time complexity.
For Subtask 6, we will need to combine Subtask 5's observation and Subtask 3's implementation to find the median in X co-ordinates and the median in Y co-ordinates. These medians will correspond to the (Ex, Ey) that minimizes the sum of Manhattan distances which is required by the problem.

Sample C++ code for Subtask 6:

```
const int MAXN = 100000;
int N, X[MAXN], Y[MAXN];
cin >> N;
for (int i = 0; i < N; ++i) {
    cin >> X[i] >> Y[i];
}
sort(X, X+N);
sort(Y, Y+N);
cout << X[N/2] << " " << Y[N/2] << endl;
```

Task 2: Roadside Advertisements

Author: Steven Halim
Department of Computer Science, National University of Singanpore,
stevenha@comp.nus.edu.sg

The abridged problem: Given a weighted tree T of up to $V = 50\,000$ vertices ($E = V - 1$ in a tree) and up to $Q = 10\,000$ queries containing 5 distinguished vertices of the tree, find a subtree of T that connects these 5 distinguished vertices with minimum total weight and report this answer.

The actual name of this problem is General-Steiner-Tree (for technical details, you can read `http://www.comp.nus.edu.sg/~stevenha/cs4234/lectures/03b.SteinerTree.pdf`). It is an NP-hard optimization problem (read: a very hard problem where nobody on earth knows how to solve it in polynomial time unless $P = NP$), but fortunately this problem is asked on its special case: a tree (a tree is a connected graph that only have one unique path between any pair of vertices) and it does have fast/polynomial solution(s).

As usual, if we do not know the full solution, we can start from easier subtasks first.

Subtask 1

In Subtask 1, we see that $V = 5$ and there is only $Q = 1$ query (that must include all those 5 test vertices). The answer is nothing but the sum of the 4 edge weights that are present in the input data. This is an $O(1)$ giveaway solution worth 7 points (at least non zero) that you can use to boost your motivation during this NOI 2017.

Subtask 2

In Subtask 2, the fact that each vertex can only be connected to at most 2 other vertices implies that the input graph is not just a tree, but an even simpler form of the tree: a line graph. We can thus find one vertex with degree 1 (one of the endpoint, let's say it is vertex A), and do a DFS traversal from vertex A via series of vertices of degree 2 to reach another vertex with degree 1 again (the other endpoint, let's say it is vertex B). This traversal can give us an array A of size $V-1$ that contains edge weights. For example, if we have the following weighted tree where each vertex is connected to at most 2 other vertices:

```
  2
1/ \1
0   1
     \2
      3
     /3
    5
     \2
      4
```

We can flatten it into a line graph that can then be represented with an array A of size $V-1$:

```
     0-2-1-3-5-4
A    = 1 1 2 3 2
```

Then, this problem degenerates into a classic Dynamic Programming (DP) 1D Range Sum Query. First, we transform array A into array pre also of size $V-1$:

```
     0-2-1-3-5-4
A    = 1 1 2 3 2
pre = 1 2 4 7 9
```

Now, if the query mentions 5 vertices {2,4,3,1,5} with the leftmost vertex is vertex 2 (index 1 in array A) and the rightmost vertex is vertex 4 (index 4 in array A), then the answer is immediately $pre[4]-pre[1-1] = pre[4]-pre[0] = 9-1 = 8$. The time complexity of this solution is $O(V+Q)$, i.e. we can answer each query in $O(1)$. Minor details are omitted. This solution worth 23 points.

Subtask 3

In Subtask 3, we notice that Q is small (not more than 100) but the input graph is really a (big) tree. So what if we run brute force? It turns out that we can. Using similar tree as above (but notice that now vertex 5 is connected to 3 other vertices so this test case is not valid for Subtask 2) and a query {6, 4, 3, 1, 2}:

```
  2
1/ \1
0   1
     \2
      3
     /3
    5
  7/ \2
  6   4
```

We can do the following strategy, for each of the 5 special vertices, go up the tree until we hit the root but ensuring that we do not count an edge twice, for example, after starting from vertex 6 upwards to the root, we have computed and marked $6 \to 5 \to 3 \to 1 \to 2$ with cost 7+3+2+1 = 13 so far:

```
 [2]
1/ \1<
0  [1]
     \2<
      [3]
     /3<
   [5]
 >7/ \2
 [6]  4
```

Then for vertex 4, we can use the same strategy of going via path $4 \to 5 \to 3 \to 1 \to 2$ but realizing that sub-path $5 \to 3 \to 1 \to 2$ is a duplicate and not recounted, so we only add one more edge weight $4 \to 5$ so the total cost is $13+2 = 15$ and we have this situation:

```
 [2]
1/ \1<
0  [1]
     \2<
      [3]
     /3<
   [5]
 >7/ \2<<
 [6] [4]
```

Is that all? Unfortunately you will get wrong answer if you stop here. For the same weighted tree, if the query is $\{6, 4, 3, 1, 5\}$ instead, then we HAVE to exclude edge $1 \to 2$ as vertex 2 (root) is NOT part of the 5 special vertices. There is a simple way to do this exclusion and we will leave it to the reader to figure it out. The time complexity of this solution is $O(V + Q \times V)$ as we may need to traverse long paths back to the root. But since $Q \leq 100$ and $V \leq 50\,000$, then $Q \times V \leq 5\,000\,000$ and this should run fast enough. This solution worth 40 points with some contestants probably missing out some of the exclusion cases.

Subtask 4

In Subtask 4, we need the full solution. In fact, there are at least 3 different possible solutions that can score full 100 points. Each of them requires somewhat advanced data structure/algorithm knowledge involving 'LCA'.

Solution 4A

One of the intended solution is to first root the (static) tree using DFS and build a Sparse Table in $O(V \log V)$ so that we can answer Lowest Common Ancestor (LCA) queries as Range Minimum Queries (RMQs) in $O(1)$ per query (details of this are listed in the LCA and Sparse Table section of Competitive Programming 3 or 3.17 textbook). For each query, we do this:

```
// sort by increasing DFS number, somewhat O(1)
sort(l, l+5, [](int a, int b) { return (H[a] < H[b]); });
subroot = LCA5(); // O(1)
// printf("deepest ancestor that connects {%d,%d,%d,%d,%d} = %d\n",
//        l[0], l[1], l[2], l[3], l[4], subroot);
// first connection, connect location l[0] with subroot, O(1)
ans = SP(subroot, l[0]);
// printf("initial ans = %d\n", ans);
for (i = 1; i < 5; i++) { // somewhat O(1)
  ans += SP(l[i], LCA2(l[i], l[i-1])); // subsequent connections :)
  // printf("after connecting %d with %d, ans = %d\n",
  //        l[i], LCA2(l[i], l[i-1]), ans);
}
printf("%d\n", ans); // this is the final answer
```

The time complexity is $O(V \log V + Q)$ with the heaviest part for Sparse Table initialization, then we can answer each query in $O(1)$.

Solution 4B

Similar as solution 4A above, but uses 2^k parent decomposition technique to compute the LCA part. The time complexity is $O(V \log V + Q \log V)$.

Solution 4C

The key insight is to realize that the special vertices are **exactly** 5, and that is ‘small’. We can select 10 more vertices which are the pairwise LCA of the given 5 vertices, e.g. if vertices {A, B, C, D, E} are given in the query, the vertices that should be selected for further processing will be {A, B, C, D, E, lca(A, B), lca(A, C), lca(A, D), lca(A, E), lca(B, C), lca(B, D), lca(B, E), lca(C, D), lca(C, E), lca(D, E)}. Then, we remove duplicated vertices (if any) and we draw an edge between every two vertices that represent the distance between them. We can calculated this via similar method as All-Pairs Shortest Paths (APSP) on tree. Notice that the resulting pre-processed subgraph is very small, up to 15 vertices and up to 15*14/2 = 105 edges. We can run Kruskal's or Prim's to obtain the Minimum Spanning Tree (MST). This sum of edges in the MST will be the answer to the query. The time complexity of this solution is $O(V \log V + Q * MST - cost)$ where the the MST-cost is rather small.

Task 3: Hotspot

Author: Wing-Kin Sung
Department of Computer Science, National University of Singapore,
ksung@comp.nus.edu.sg

Below is the solutions for various subtasks.

Subtask 1

In Subtask 1, the graph is a line and $k = 1$. We just identify all nodes between A_1 and B_1; then, we report them.

Subtask 2

In Subtask 2, the graph is a tree and $k = 1$. Using Dijkstra's algorithm, we find the shortest path from A_1 to B_1. Then, we report all nodes on the shortest path.

Subtask 3

In Subtask 3, the graph is a line and $k < 200$. We implement a counter for every node. Initially, every counter is initialized as zero. Then, we iterate i from 1 to k. For the i iteration, we increment the counter of all nodes between A_i and B_i by one. Finally, we report all nodes with the maximum count.

Subtask 4

In Subtask 3, the graph is a tree and $k < 200$. We implement a counter for every node. Initially, every counter is initialized as zero. Then, we iterate i from 1 to k. For the i iteration, using Dijkstra's algorithm, we find the shortest path from A_i to B_i. Then, we increment the counter of all nodes on the shortest path by one. Finally, we report all nodes with the maximum count.

Subtask 5

In Subtask 5, it is a general graph and $k < 20$. For every node u, we implement a variable sum_u. Initially, we set $sum_u = 0$.

Let $d(u, v)$ be the shortest distance between u and v. Using FloydWarshall algorithm, we compute $d(u, v)$ for all u, v using $O(n^3)$ time.

For every pair (A_i, B_i), we can verify if a node u is on the shortest path from A_i to B_i by checking if

$$d(A_i, u) + d(u, B_i) = d(A_i, B_i).$$

Let S_i be the set of all nodes on the shortest path between A_i and B_i. S_i, for $i = 1, \ldots, k$, can be found using $O(kn + n^3)$ time.

Next, for every S_i, we need to find the number of shortest paths from A_i to B_i that passes through u. We define $p_{A_i}(u)$ be the number of shortest paths from A_i to u. $p_{A_i}(u)$ satisfies the following recursive formula. Let the set of neighbors of u be $neig(u)$.

$$p_{A_i}(A_i) = 1 \tag{1}$$

$$p_{A_i}(u) = \sum_{v \in neig(u)} \{p_{A_i}(v) \mid d_{A_i}(v) + 1 = d_{A_i}(u)\} \text{ for every } u \in S_i \tag{2}$$

By dynamic programming, we can compute $p_{A_i}(u)$ for all nodes u in $O(m)$ time.

Similarly, let $p_{B_i}(u)$ be the number of shortest paths from B_i to u. By dynamic programming, we can compute $p_{B_i}(u)$ for all nodes u in $O(m)$ time.

After that, we set $sum_u = \{\frac{p_{A_i}(u) * p_{B_i}(u)}{p_{A_i}(B_i)} \mid u \in S_i\}$.

Finally, we report all nodes u that maximize sum_u.

Subtask 6

In Subtask 6, it is a general graph and $k < 2000$. For every node u, we implement a variable sum_u. Initially, we set $sum_u = 0$.

For every pair (A_i, B_i), we first compute the shortest distance from A_i to all nodes u by Dijkstra's algorithm. Such a distance is denoted as $d_{A_i}(u)$. Similarly, we compute the shortest distance from B_i to all nodes u by Dijkstra's algorithm. Such a distance is denoted as $d_{B_i}(u)$.

Observe that a node u is on the shortest path from A_i to B_i if

$$d_{A_i}(u) + d_{B_i}(u) = d_{A_i}(B_i).$$

Let S_i be the set of all nodes on the shortest path between A_i and B_i. As Dijkstra's algorithm runs in $O(m + n \log n)$ time, S_i, for $i = 1, \dots, k$, can be found using $O(km + kn \log n)$ time.

The remaining steps are the same as Subtask 5.

Task 4: RMQ

Author: Mark Theng
SG IOI Team 2014,
krawthekrow@gmail.com

Subtask 1

To solve the first subtask, one could iterate over all possible permutations of cows. For each permutation, the RMQs can be verified manually in $O(QN)$ time, resulting in an overall time complexity of $O(N!\,QN)$.

Subtask 2

Each RMQ $\min(L_i, R_i) = C_i$ poses the following constraints:

- All tag numbers in the range (L_i, R_i) must be at least C_i. Call this the class 1 constraint.
- C_i must exist somewhere within (L_i, R_i). Equivalently, C_i cannot be outside that range. Call this the class 2 constraint.

One can construct a Boolean matrix M_{ij} denoting, for each (i, j), whether tag i can be placed in position j.

If (i, j) satisfies all class 1 constraints, then i must be more than or equal to the *maximum* of all C_k for the queries where $j \in (L_k, R_k)$. The maximums can be precomputed in $O(QN)$ time, then all (i, j) can be checked against the maximums in $O(N^2)$ time.

If (i, j) satisfies all class 2 constraints, then $j \in (L_k, R_k)$ for all queries where $C_k = i$. One can precompute the maximum L_k and minimum R_k for all i in $O(QN)$ time, then check all (i, j) against them in $O(N^2)$ time.

Once the matrix M_{ij} has been generated, a bipartite graph can be constructed. This bipartite graph will contain N vertices enumerating the tag numbers and N vertices enumerating the positions. An edge would exist between the i-th vertex of the first set and the j-th vertex of the second set if M_{ij} is true. Then, a possible permutation exists only if a bipartite matching of cardinality N exists, and if so, the matching would yield a viable permutation. Since there are at most N^2 edges, the Hopcroft-Karp algorithm will compute the solution with a complexity of $O\left(N^{2.5}\right)$.

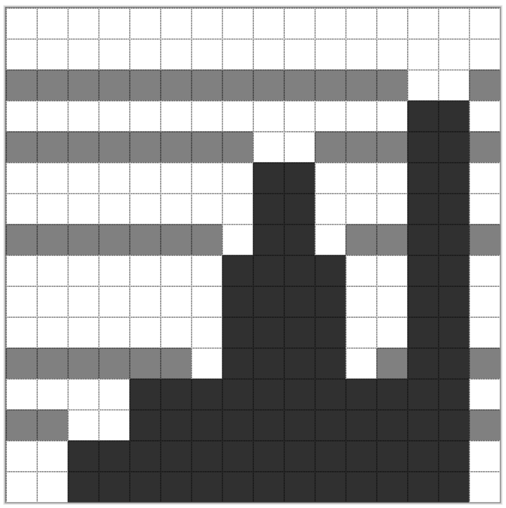


Figure 1: Sample M_{ij} for $N = 16$. Cells invalid due to class 1 constraints are shaded dark grey; remaining cells invalid due to class 2 constraints are shaded a lighter grey.

A sample M_{ij} is included in Figure 1, corresponding to the following set of RMQs:

```
RMQ(2, 14)  = 2
RMQ(4, 14)  = 4
RMQ(6, 11)  = 4
RMQ(7, 10)  = 8
RMQ(8, 9)   = 11
RMQ(13, 14) = 13
```

Note that M_{ij} has a special structure to it – this will be important for the **important observation**.

For 30% of the subtask's score, the class 2 constraints are relaxed: C_i has to exist somewhere within (L_i, R_i), but can also exist outside that range as well.

To solve this, compute M_{ij} taking into account only class 1 constraints. Take, for each j, the minimum i such that M_{ij} is true. Then, check that the array satisfies all the RMQs. We claim that, if this array does not satisfy all the RMQs, then no array satisfies all the RMQs.

Make the following **important observation**: If any i is subject to a class 2 constraint (L_k, R_k), then all (i', j') for $i' < i$ and $j' \in (L_k, R_k)$ will be invalid due to the class 1 constraint.

If the array fails any of the RMQs, it must be due to a class 2 constraint. Let this constraint be that C_k must exist somewhere within (L_k, R_k). Then the **important observation** applies and all M_{ij} for $i < C_k$ and $j \in (L_k, R_k)$ is false. Thus the minimum i such that M_{ij} is true must be at least C_k for all $j \in (L_k, R_k)$. Since we did not set any of these to C_k, it must be that all $M_{C_k j}$ for $j \in (L_k, R_k)$ is false. Hence no array can satisfy that constraint, proving the claim.

Subtask 3 (30%)

Use a range-update, point-query segment tree (using lazy propagation) to find, for each position j, the maximum B_j such that there exists an RMQ (L_k, R_k) containing j such that $C_k = B_j$. Then, the array B_j satisfies all class 1 constraints.

Now, use a point-update, range-query segment tree to check if the array B_j satisfies all the RMQs. If it does, output the array. If not, then for the same reason as discussed earlier, there must be no possible assignment that satisfies all the RMQs.

The first step requires Q updates and N queries on the segment tree, while the second step requires N updates and Q queries. This results in an $O((N + Q) \log N)$ complexity overall.

Subtask 3

Consider the following greedy algorithm: In ascending order of i, assign each tag i to any available j such that all constraints are satisfied. If, for any i, there is no available j, then report that no assignment is possible.

If this algorithm produces a permutation, it will definitely satisfy all the RMQs. All that is left is to show that, if the algorithm reports that no assignment is possible, then there is indeed no possible permutation that satisfies all the RMQs.

Assume that there exists a permutation B_i that satisfies all the RMQs, but the greedy algorithm reports that there is no possible assignment. Let K be the i for which the greedy algorithm failed to assign a position.

Consider the case where there does not exist any RMQ with answer $C_k = K$. Then all M_{Kj} that is false can only be so due to class 1 constraints. There can be at most $N-K-1$ positions j for which M_{Kj} is false, since $B_j > K$ for all such j. On the other hand, only K positions in the array have already been assigned a tag number, so there should be at least $N-(N-K-1)-K = 1$ cell remaining.

On the other hand, if there does exist an RMQ (L_k, R_k) with answer $C_k = K$, then all M_{Kj} for j not within (L_k, R_k) are false due to the class 2 constraint. However, due to the **important observation**, all $j \in (L_k, R_k)$ have M_{ij} false for $i < K$. Thus, all tags previously assigned must have been assigned outside (L_k, R_k). Then, since the greedy algorithm failed to find any admissible j, it must be that all M_{Kj} for $j \in (L_k, R_l)$, and hence all j, is false. But then, there is indeed no permutation B_i that satisfies all the RMQs.

Having proven the correctness of the greedy algorithm, all that remains is to find a fast way to find, for each i, any j that satisfies all the RMQs that has not been previously chosen. This can be achieved using balanced binary search trees (conveniently implemented in the C++ standard library).

First, as before, use a segment tree to find, for each j, the minimum i that satisfies all class 1 constraints, which we will call T_j. Then, initialise an empty balanced binary search tree S. Now, in ascending order of i, use a dictionary to find all j such that $T_j = i$, and store these j in S. Then, compute the intersection (L_i, R_i) of all RMQs (L_k, R_k) for which $C_k = i$. Binary search S to find any $j \in (L_i, R_i)$. If no such j exists, then no permutation is possible. Otherwise, assign i to j and remove j from S, and proceed with the next i.

Accounting for at most $O(N)$ insertions, removals and searches on S, each requiring $O(\log N)$ time, and including the time required for the segment tree, the time complexity of this algorithm is $O((N+Q)\log N)$.

Task 5: I want to be the very best too!

Author: Wei-Liang Gan
SG IOI Team 2013,
ganweiliang@gmail.com

Introduction

The abridged problem: Given an $R \times C$ grid of size at most 50000 where each grid square has a number and a colour represented by a number between 1 and 50000, perform 2 types of queries:

1. Change the colour of a grid square
2. Given a number L and a starting square, find the number of unique colours reachable from the starting square by only using squares with number at most L and only moving between squares that are horizontally or vertically adjacent

For all of the solutions presented below, we will use an $R \times C$ array to keep track of the colours and update it when they change. One of the edge cases happens when the number in the starting square is more than L, in this case the answer is simply 0. In the discussions below, we assume that this case has been handled and thus we only consider the case where the number in the starting square is at most L.

Subtask 1

In Subtask 1, we see that $R = 1$ and there are at most $Q \leq 1000$ queries. Also the number at $(X, 1)$ is equal to X. This means that all the numbers at most L are in the grid squares $(1, 1)$ to $(\min(L, C), 1)$. Since Q is small enough, for each query we can just loop through these grid squares and count the number of distinct colours. We can perform this counting using an array of size 50000 that counts whether each colour has appeared or not. This gives a simple $O(QC)$ solution with $O(1)$ per update and $O(C)$ per query.

Subtask 2

In Subtask 2, there are at most $Q \leq 10$ queries. However, the grid is not a single row anymore. Hence, for each query, we have to perform a BFS to find all the squares reachable from the starting square by only using squares with number at most L. Then, we can use a similar method as Subtask 1 to count the number of unique colours in this square. This gives a $O(QRC)$ solution with $O(1)$ per update and $O(RC)$ per query.

Subtask 3

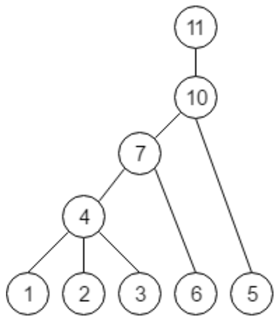
In Subtask 3, we notice that there are only 2 colours. Ignoring the edge case presented earlier, the answer is either 1 or 2. We simply need to check whether all the reachable squares are of the same colour or not. The main observation we need to make is that the numbers in the grid do not change, so given some L the set of reachable squares do not change. Thus, we should find a way to precompute this.

Given some starting square (X_1, Y_1) and any L, if some other square (X_2, Y_2) is reachable from the starting square using numbers at most L, then each of the 4 adjacent neighbours of (X_2, Y_2) with numbers smaller than it are also reachable from (X_1, Y_1) regardless of L.

Therefore, we can generate a tree by iterating through the squares in increasing L. For each square, create a new vertex V. For each of the 4 adjacent neighbours V' that are smaller than it, make V the parent of the root of the tree containing V'. This can be done quickly by using a Union-Find Disjoint Set data structure to track the root of each vertex. For example, the following grid

```
1 4 3
11 2 7
5 10 6
```

will generate the following tree



In this tree, V_1 is an ancestor of V_2 if and only if V_2 is reachable from V_1 using numbers at most V_1. Therefore, given a starting vertex V and a number L, we want to first find the set of vertices that are reachable from V using vertices with number at most L. We notice that a vertex V can reach all the vertices within it's subtree using numbers at most V. Therefore, this set of reachable vertices is simply the subtree of the highest ancestor of V that is at most L.

Thus we need to solve two problems now which we shall discuss separately.

1. Given the above tree and a vertex V, we want to quickly find the highest ancestor of V that is at most L.

   Notice that the arent of V always has a value larger than V. Therefore, the path from V to the root has values in increasing order. We can thus perform a binary search on k to find the largest k such that the k-th ancestor has number at most L. To do this, we first create an array P where $P[v][i]$ stores the 2^i-th ancestor of v for all vertices v. We can precompute this array in $O(RC \log RC)$ time by processing the vertices from root to leaf and using the recursive formula $P[v][i] = P[P[v][i-1]][i-1]$ since $2^{i-1} + 2^{i-1} = 2^i$. Then, we can perform a binary search as follows:

   ```
   initialise current = V
   for i from log(R*C) to 0 do
       if P[current][i] <= L
           current = P[current][i]
   return current
   ```

2. Given the above tree and a vertex V, we want to be able to update the colour of V to one of two colours, and find out how many distinct

colours are there in the subtree of vertex V.

We can reduce the problem from a subtree to a subarray by running a depth-first search on the tree and replacing the index of each vertex by its position in the pre-order traversal of the tree. This way, the vertices in each subtree form a contiguous sequence of indices.

We then create an array C where $C[x]$ represents the colour of the x-th vertex in the pre-order traversal. Since this subtask only has 2 colours, let's use 0 and 1 to denote the 2 colours. When we update the colour of a vertex, we update the corresponding value in the array. Now we just need to check whether all the vertices in a subtree, and thus the corresponding contiguous subsequence of values in the array all have the same colour. To do this, we only need to check whether the values are all 0 or 1 which can be done by checking the sum of those values. We can thus use any data structure that supports range sum queries with updates such as a Segment tree or a Fenwick tree.

Combining the 2 parts above will give us the required solution.

Subtask 4

In subtask 4, we have a grid of 1 row, the same as that of subtask 1. This means that for each query, we need to count the number of distinct colours in the squares $(X, 1)$ to $(X, \min(L, C))$. However, Q is too large so the $O(QC)$ solution will exceed the time limit.

Suppose we want to count the number of distinct colours in squares $(1, 1)$ to $(P, 1)$. Consider an array A of size C. For each square $(X, 1)$ between $(1, 1)$ and $(P, 1)$, let $A[X]$ be the smallest number larger than X such that $(A[X], 1)$ is of the same colour, or $A[X] = \infty$ if such a square does not exist. If $A[X] \leq P$, that means there is another square of the same colour within that range. Otherwise, it is the rightmost square of that colour within that range since the next square of such a colour either does not exist or is outside the range.

Each colour in the range will only have exactly one rightmost square with that colour, so the number of distinct colours is the number of rightmost squares, which is the number of values from $A[1]$ to $A[P]$ which are greater than P. We also need to be able to update this array quickly with the correct values. We shall discuss these two parts separately.

1. We want to be able to quickly update the values of $A[X]$ as defined above

   We can create a binary search tree for each possible colour since there are at most 50000 colours. The binary search tree for a colour K would contain all the values of X such that $(X, 1)$ has colour K. When we update the colour of a square, we would update the binary search tree of the original colour and the new colour. We can then binary search on the appropriate trees to update the values in A that are affected. The implementation details are left as an exercise for the reader.

2. Given the above array, we want to quickly get the number of values greater than some value P in any contiguous subarray

   This is a data structure problem and the requirement of being able to support updates makes it much harder. The author's solution involves the use of a segment tree where each node stores all the values in the range represented by that node of the Segment tree. This will only need to store $O(C \log C)$ values since each of the $\log C$ layers of the Segment tree only stores each value exactly once. Furthermore, when we update a value we only need to update $O(\log C)$ nodes in the segment tree.

   Now, the data structure in each node needs to support 3 operations in logarithmic time: adding an element, removing an element, and querying how many values there are in the data structure that are greater than P. This can be done with a binary search tree that supports order statistics, which can be constructed from the GNU extension library for policy-based data structures. Alternatively, one can use an augmented Segment tree or Fenwick tree that uses lazy node creation to ensure that the memory use is limited by the number of values in the data structure.

Subtask 5

In subtask 5, we need a full solution that combines the solution from the previous two subtasks. We can easily combine the two answers by running the solution in Subtask 4 on the pre-order traversal array generated in Subtask 3 since every query becomes the number of distinct colours in a contiguous subarray. From this we see how solving subtasks can help lead us to the final intended solution.