

ICPC Review

Announcements

- Tea Time is at Little Italy this week.

The Problems

- | | |
|--|-------------------------|
| ● Stars - First solved at 10 minutes | 42 / 55 submits correct |
| ● Treasure - First Solved at 22 minutes | 17 / 66 submits correct |
| ● Haybales - First Solved at 53 minutes | 11/173 submits correct |
| ● Rainbow - First Solved at 60 minutes | 9 / 35 submits correct |
| ● Ducks - First Solved at 94 minutes | 3 / 13 submits correct |
| ● Long Long - First Solved at 106 minutes | 3 / 25 submits correct |
| ● Security - First Solved at 165 minutes | 5 / 30 submits correct |
| ● Move Away - First Solved at 169 minutes | 3 / 8 submits correct |
| ● Unsatisfy - First Solved at 191 minutes | 2 / 14 submits correct |
| ● Flipping - First Solved at 272 minutes | 1 / 8 submits correct |
| ● Exciting - Not Solved | 0 / 6 submits correct |

Star Arrangements - Statement

Find all ways of arranging stars on a flag satisfying the following conditions:

- 1) There are at least 2 rows of stars.
- 2) All odd numbered rows have the same number of stars.
- 3) All even numbered rows have the same number of stars.
- 4) The difference in the number of stars between any 2 adjacent rows is 0 or 1
- 5) The first row cannot have fewer stars than the second, and must have at least 2 stars.

Describe all possible fields, with # stars in 1st row, then # stars in second row, sorted by increasing 1st row, then increasing 2nd row as a tie-breaker.

Star Arrangements - Solution

There are only 2 cases to consider. Call the number of stars in the flag S and the number of stars in the first row F .

We can try all values of F from 1 to $2F - 1$ (when we have only 2 rows, and the first row is bigger than the second). There are 2 cases to consider:

- 1.) Odd rows have 1 more star than even rows.
 - a.) If we have the same number of odd and even rows, this means that S is divisible by $2F - 1$.
 - b.) If we have one more odd than even row, we can just see if $S + F - 1$ (adding a final even row) is divisible by $2F - 1$.
 - c.) If either of this is true, we can print $F F-1$.
- 2.) All rows have the same number of stars. This is true if and only if S is divisible by F , and we can print $F F$.

Star Arrangements - Code

```
import java.util.*;

public class Stars {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int S = in.nextInt();

        for (int F = 2; F + F - 1 <= S; F++) {
            if (S % (2 * F - 1) == 0 || (S + F - 1) % (2 * F - 1) == 0)
                System.out.println(F + " " + (F-1));
            if (S % F == 0)
                System.out.println(F + " " + F);
        }
    }
}
```

Treasure Map - Statement

You know about a variety of gold mines. Each gold mine produces G gold on day 1 and D less gold for every subsequent day.

You also know how long it takes to travel from gold mine to gold mine.

While you can't stay at a given gold mine multiple days in a row, you can exit it and come back later to collect gold (albeit less of it) any time in the future.

What is the most gold you can collect?

Treasure Map - Solution

To simplify things, we will number the first day as day 0. Then, the amount of gold at a mine with values G and D is $\max(0, G - \text{day} * D)$.

We want to easily calculate the maximum amount of gold we can earn at a given mine, starting on a given day. We have this relation:

$$\begin{aligned} \text{best}(\text{mine}, \text{day}) = & \text{value}(\text{mine}, \text{day}) + \\ & \max \text{ over } e \text{ in edges adjacent to mine of:} \\ & \text{best}(e.\text{otherMine}, \text{day} + e.\text{time}) \end{aligned}$$

Use DP to speed this up!

Treasure Map - Code

```
import java.util.*;
import java.io.*;

public class Treasure {

    static int[] G;
    static int[] D;
    static List<List<Edge>> E;
    static int[][] memo;
    ...
}

class Edge {
    int N;
    int T;

    Edge(int N, int T) {
        this.N = N;
        this.T = T;
    }
}
```

Treasure Map - Code

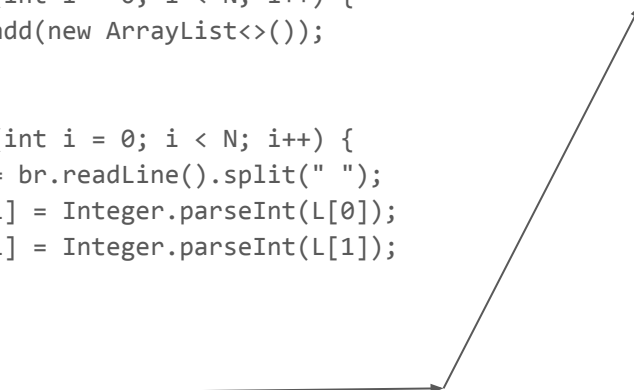
```
static int value(int mine, int day) {  
    return Math.max(0, G[mine] - D[mine] * day);  
}  
  
static int best(int mine, int day) {  
    if (day >= 1000) return 0;  
    if (memo[mine][day] >= 0) return memo[mine][day];  
  
    int solution = 0;  
    for (Edge e : E.get(mine)) {  
        solution = Math.max(solution, best(e.N, day + e.T));  
    }  
    solution += value(mine, day);  
    memo[mine][day] = solution;  
    return solution;  
}
```

Treasure Map - Code

```
public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String[] L = br.readLine().split(" ");
    int N = Integer.parseInt(L[0]);
    int M = Integer.parseInt(L[1]);
    G = new int[N];
    D = new int[N];
    E = new ArrayList<>(N);
    for (int i = 0; i < N; i++) {
        E.add(new ArrayList<>());
    }

    for (int i = 0; i < N; i++) {
        L = br.readLine().split(" ");
        G[i] = Integer.parseInt(L[0]);
        D[i] = Integer.parseInt(L[1]);
    }
    ...
    for (int i = 0; i < M; i++) {
        L = br.readLine().split(" ");
        int a = Integer.parseInt(L[0]) - 1;
        int b = Integer.parseInt(L[1]) - 1;
        int t = Integer.parseInt(L[2]);
        E.get(a).add(new Edge(b, t));
        E.get(b).add(new Edge(a, t));
    }

    memo = new int[N][1001];
    for (int i = 0; i < N; i++) Arrays.fill(memo[i], -1);
    System.out.println(best(0, 0));
}
```

A diagram consisting of two arrows. One arrow starts at the end of the first loop (the one that iterates over N and adds to E) and points to the start of the second loop (the one that iterates over M and adds to E). The second arrow starts at the end of the second loop and points to the line 'System.out.println(best(0, 0));'.

Jumping Haybales - Statement

Given a square grid of either empty spaces or haybales, and the longest distance we can jump (K), determine the shortest number of jumps it takes to get from the top left to bottom right of the grid, knowing that we can't land on a haybale.

Jump distance of 2: 4 moves

Start			
1		2	3
			4

Jump distance of 1: impossible

Start		
		Finish

Jumping Haybales - Solution

Iterate over all rows and columns. In each cell, the best we can do equals the minimum of the smallest value within K cells above us, and the smallest value within K cells to the left of us, plus 1.

We can use a priority queue for each row / column to keep track of the best cells, by sorting first by distance from the start, then by trying to go as far down/right as possible.

We can use the priority queue to get the cheapest value possible. If it is in range, we will keep it in the priority queue and use it, otherwise we will remove it from the priority queue and look for the next element.

Jumping Haybales - Code

```
class Best implements Comparable<Best> {
    int jumps;
    int idx;

    Best(int jumps, int idx) {
        this.jumps = jumps;
        this.idx = idx;
    }

    public int compareTo(Best other) {
        if (jumps != other.jumps)
            return jumps - other.jumps;
        return other.idx - idx;
    }
}

static int findBest(PriorityQueue<Best> pq, int min) {
    while(pq.size() > 0) {
        Best prev = pq.peek();
        if (prev.idx >= min) return prev.jumps + 1;
        pq.poll();
    }
    return INF;
}
```

Jumping Haybales - Code

```
import java.util.*;
import java.io.*;

public class Haybales {

    static int INF = 1000000;

    public static void main(String[] args) throws Exception {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] L = br.readLine().split(" ");
        int N = Integer.parseInt(L[0]);
        int K = Integer.parseInt(L[1]);

        char[][] grid = new char[N][0];

        @SuppressWarnings("unchecked")
        PriorityQueue<Best>[] cols = new PriorityQueue[N];
        for (int i = 0; i < N; i++) {
            grid[i] = br.readLine().toCharArray();
            cols[i] = new PriorityQueue<Best>(K);
        } ...
    }
}
```

Jumping Haybales - Code

```
...
int best = 0;

cols[0].add(new Best(0, 0));
for (int r = 0; r < N; r++) {

    PriorityQueue<Best> row = new PriorityQueue<>(K);
    if (r == 0) row.add(new Best(0, 0));

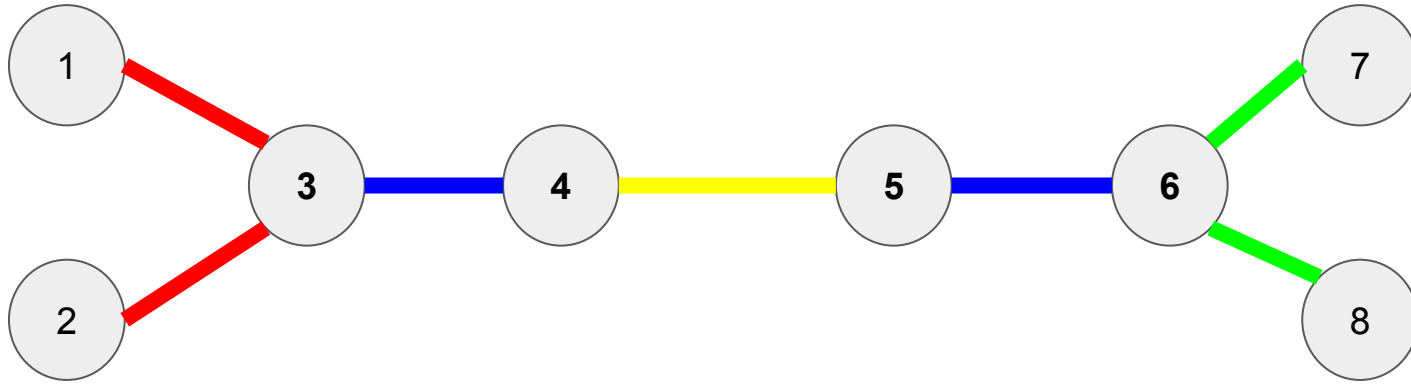
    for (int c = 0; c < N; c++) {
        if (grid[r][c] == '#' || (r == 0 && c == 0)) continue;
        best = Math.min(findBest(row, c-K), findBest(cols[c], r-K));
        if (best != INF) {
            row.add(new Best(best, c));
            cols[c].add(new Best(best, r));
        }
    }
}

System.out.println(best == INF ? -1 : best);
}
```


Rainbow Roads - Statement

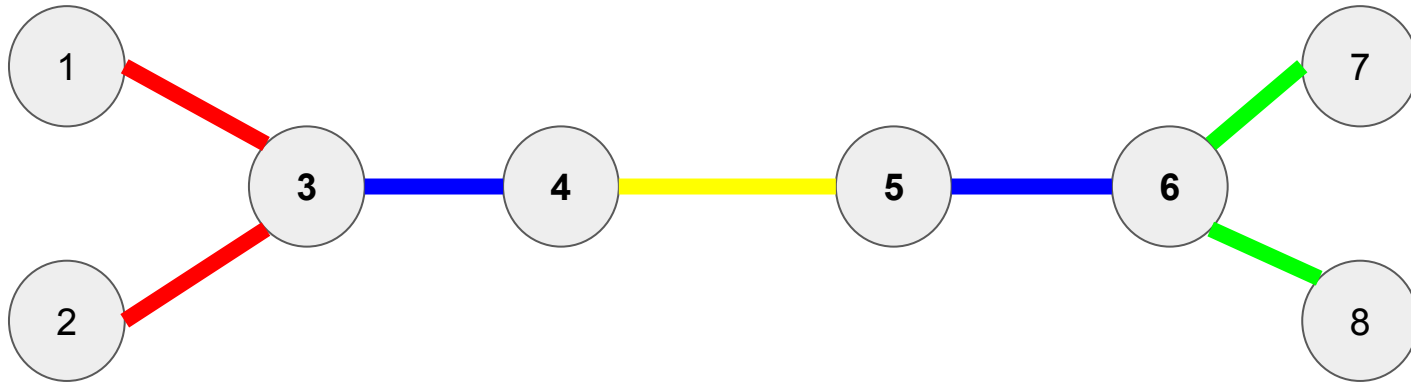
Given a tree with colored edges, identify all vertices such that no paths starting at that vertex have 2 adjacent edges with the same color.

Rainbow Roads - Example



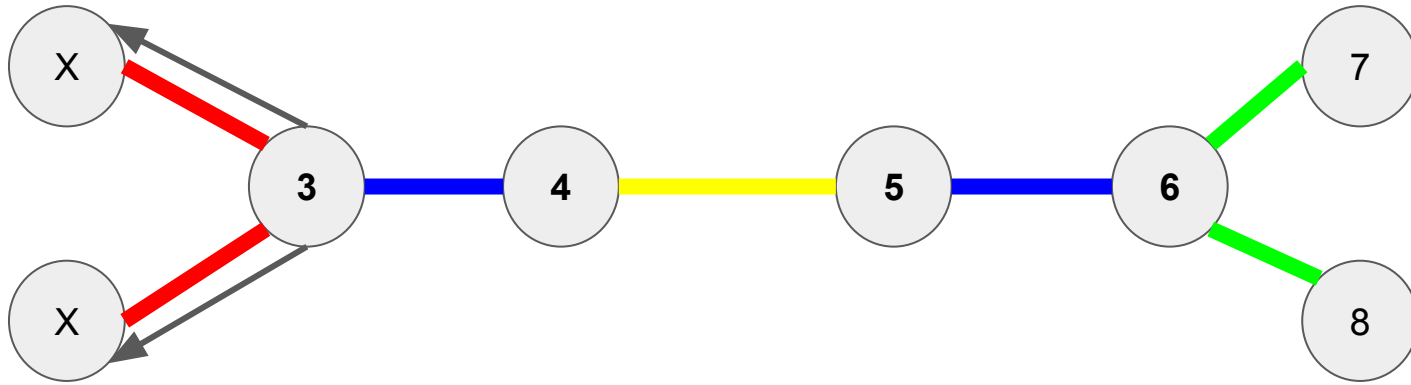
1 -> 2 and 2 -> 1 have red-red, and 7 -> 8 and 8 -> 7 have green-green

Rainbow Roads - Solution



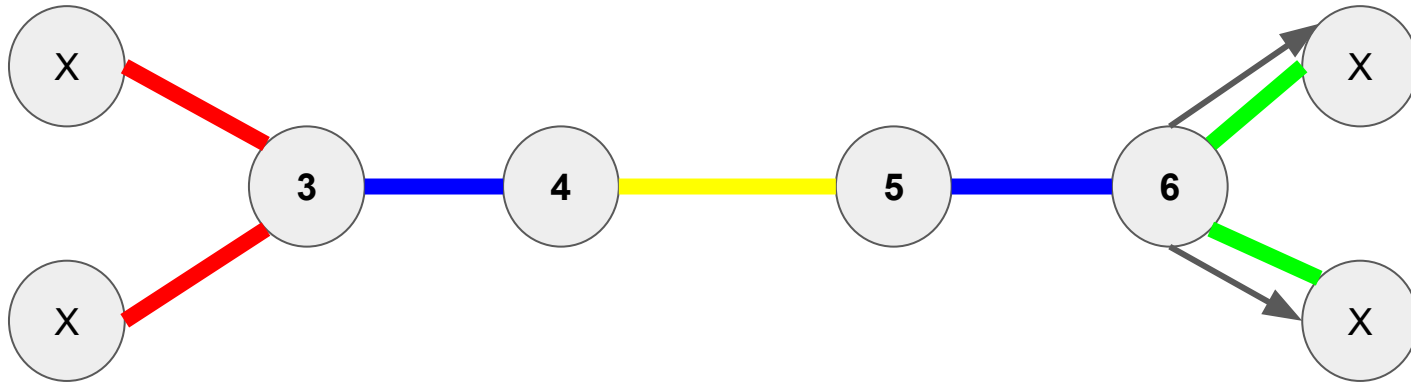
Visit all vertices. Each time we are at a vertex with multiple edges of the same color, mark all subtrees those edges lead to as not “special”.

Rainbow Roads - Solution



Example - when visiting vertex 3, we can remove the subtrees containing 1 and 2.

Rainbow Roads - Solution



Example - when visiting vertex 6, we can remove the subtrees containing 7 and 8.

Rainbow Roads - Solution

The input size may be large enough that a recursive solution would cause a stack overflow. To address this issue, we will instead perform this iteratively, using a stack.

Rainbow Roads - Solution

Another issue to deal with is that we don't want to eliminate the same sets of vertices. Therefore, we will only visit each edge once. When we eliminate a subtree, we will recursively (ish) mark all vertices as not special, then delete all edges except the one leading back to our parent.

We need separate lists of edges for both traversing and looking for duplicate colors.

Another option is to make sure each vertex is visited at most twice, without repeatedly visiting any edges.

Rainbow Roads - Code

```
class Visit {
    Node Node;
    Node Parent;

    Visit(Node Node, Node Parent) {
        this.Node = Node;
        this.Parent = Parent;
    }
}

class Node {
    List<Edge> Edges = new ArrayList<>();
    boolean Special = true;
    int Visits;

    void addEdge(Node neighbor, int color) {
        Edges.add(new Edge(neighbor, color));
        Visits = 0;
    }
}
```

```
class Edge {
    Node Node;
    int Color;
    boolean Visited;

    Edge(Node Node, int Color) {
        this.Node = Node;
        this.Color = Color;
        this.Visited = false;
    }
}
```


Rainbow Roads - Code

```
static void kill(Node node, Node parent) {

    if (node.Visits >= 2) return;
    node.Visits++;

    Stack<Visit> S = new Stack<>();
    S.push(new Visit(node, parent));

    while(S.size() > 0) {
        Edge edgeToParent = null;
        Visit V = S.pop();

        V.Node.Special = false;

        for (Edge edge : V.Node.Edges) {
            if (edge.Node != V.Parent && !edge.Visited && edge.Node.Visits < 2) {
                edge.Visited = true;
                edge.Node.Visits++;
                S.push(new Visit(edge.Node, V.Node));
            }
        }
    }
}
```

Rainbow Roads - Code

```
import java.util.*;
import java.io.*;

public class Rainbow {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)));

        int N = Integer.parseInt(br.readLine());

        Node[] G = new Node[N];
        for (int i = 0; i < N; i++) G[i] = new Node();

        for (int i = 0; i < N-1; i++) {
            String[] L = br.readLine().split(" ");
            int U = Integer.parseInt(L[0]) - 1;
            int V = Integer.parseInt(L[1]) - 1;
            int C = Integer.parseInt(L[2]) - 1;

            G[U].addEdge(G[V], C);
            G[V].addEdge(G[U], C);
        } ...
    }
}
```

Rainbow Roads - Code

```
for (Node node : G) {
    Node[] prev = new Node[N];
    for (Edge edge : node.Edges) {
        if (prev[edge.Color] != null) {
            kill(prev[edge.Color], node);
            kill(edge.Node, node);
        }
        prev[edge.Color] = edge.Node;
    }
}

int count = 0;
for (Node node : G) {
    if (node.Special) count++;
}

out.println(count);
for (int i = 0; i < N; i++) {
    if (G[i].Special) out.println(i+1);
}
out.flush();
}
```

Ducks in a Row - Statement

Given a sequence of Ducks and Geese, and integers N and K , find the smallest number of times you can select a subsequence of Ducks and Geese, and change all Ducks to Geese, and Geese to Ducks, such that there are at least K groups of N consecutive ducks.

Ducks in a Row - Example

We need 2 (or more) runs of length 2 (or more).

We have: DDDGD

We can flip the 3rd and 4th birds: DDGD

To arrive at: DDGD

Which has 2 runs of length 2: DDGD

Ducks in a Row - Example 2

We need 3 (or more) runs of length 2 (or more).

We have:

GGDGGDGG

We can flip every bird:

GGDGGDGG

To arrive at:

DDGDDGDD

Which has 3 runs of length 2:

DDGDDGDD

Ducks in a Row - Solution

As a simple shortcut, we know that K sequences of length N require $K * N$ ducks, plus $K - 1$ geese separating the runs. Therefore, if $K * N + K - 1 > |B|$, we can immediately return false. Otherwise, it will be possible to do, as we can at least just iterate over the list and flip any ducks that are out of order.

Ducks in a Row - Solution

Another thing to realize is that any overlapping “spells” can be re-written as the same or fewer number of non-overlapping spells.

For example:

SSSSSS S equals SS
SSSSSSSSSS SSSS

Ducks in a Row - Solution

We can develop a recurrence relation:

If I am at position **I**, and still have to satisfy **S** sequences, and I have **T** more ducks to satisfy this sequence, and **C** is true iff I am casting:

Let **D** be true if either (**B**[**I**] == 'D' and !**C**) or (**B**[**I**] == 'G' and **C**)

Let **T'** equal **T** - 1, **S'** equal **S** + 1 if **T'** equals 0, **S** otherwise.

Let **R** equal 1 if **C** is false, 0 otherwise.

Case 1: - I wish to remain casting/not casting.

I can do it in $\text{solve}(\mathbf{C}, \mathbf{S}', \mathbf{T}', \mathbf{I}+1)$ if **D**, $\text{solve}(\mathbf{C}, \mathbf{S}, \mathbf{N}, \mathbf{I}+1)$ otherwise.

Case 2: I wish to start or end a spell.

I can do it in $\mathbf{R} + \text{solve}(!\mathbf{C}, \mathbf{S}', \mathbf{T}', \mathbf{I}+1)$ if **NOT D**, $\mathbf{R} + \text{solve}(!\mathbf{C}, \mathbf{S}, \mathbf{N}, \mathbf{I}+1)$ otherwise.

We can use a memoization array of size $2 \times N + 1 \times K + 1 \times |B|$ to drastically speed this up.

Ducks in a Row - Code

```
import java.util.*;
import java.io.*;

public class Ducks {

    static final int INF = 1000000000;
    static final int NOT_IN_CACHE = -1;

    static char[] L;
    static int N;
    static int K;
    static int M;
    static int[][][][] memo;

    ...
}
```

Ducks in a Row - Code

```
static int solve(boolean casting, int seqsLeft, int ducksLeft, int pos) {

    if (seqsLeft == 0) return 0;
    if (pos == M) return INF;

    int dLeft = Math.max(0, ducksLeft), castInt = casting ? 1 : 0;
    if (memo[castInt][seqsLeft][dLeft][pos] != NOT_IN_CACHE) return memo[castInt][seqsLeft][dLeft][pos];

    boolean isDuck = ((L[pos] == 'D') != casting);

    int best = INF, newDucksLeft = ducksLeft - 1; newSeqsLeft = (ducksLeft == 1 ? seqsLeft - 1 : seqsLeft);

    if (isDuck) best = Math.min(best, solve(casting, newSeqsLeft, newDucksLeft, pos + 1));
    else best = Math.min(best, solve(casting, seqsLeft, N, pos + 1));

    if (!isDuck) best = Math.min(best, 1-castInt + solve(!casting, newSeqsLeft, newDucksLeft, pos + 1));
    else best = Math.min(best, 1-castInt + solve(!casting, seqsLeft, N, pos + 1));

    memo[castInt][seqsLeft][dLeft][pos] = best;
    return best;
}
```

Ducks in a Row - Code

```
public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String[] LL = br.readLine().split(" ");
    N = Integer.parseInt(LL[0]);
    K = Integer.parseInt(LL[1]);
    L = br.readLine().toCharArray();
    M = L.length;

    if (K * N + K - 1 > M) {
        System.out.println(-1);
    } else {
        memo = new int[2][K+1][N+1][M];
        for (int i = 0; i < 2; i++)
            for (int j = 0; j <= K; j++)
                for (int k = 0; k <= N; k++)
                    Arrays.fill(memo[i][j][k], NOT_IN_CACHE);

        System.out.println(solve(false, K, N, 0));
    }
}
```

Long Long Strings - Statement

Given a two lists of instructions - either to insert a character at a given position or delete the character at a given position, determine if the two programs are equivalent.

Long Long Strings - Example

Program 1

Visualized

Delete 1

ABCDE -> BCDE

Delete 2

B**C**DE -> BDE

Program 2

Visualized

Delete 3

AB**C**DE -> ABDE

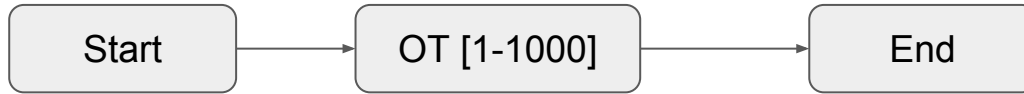
Delete 1

ABDE -> BDE

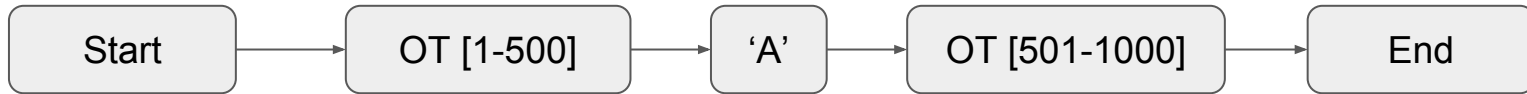
Since BDE equals BDE, and the two strings would be equal regardless of the Initial value, Program 1 and Program 2 are equivalent.

Long Long Strings - Solution

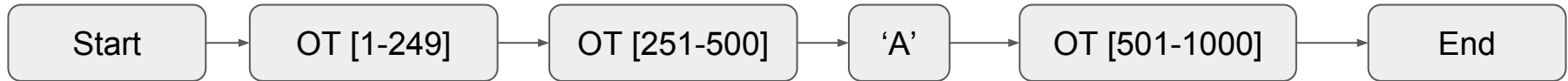
We can represent our long string as a linked - list style structure, allowing for efficient manipulation, storing the original text in blocks.



Insert(500, A)



Delete (250)

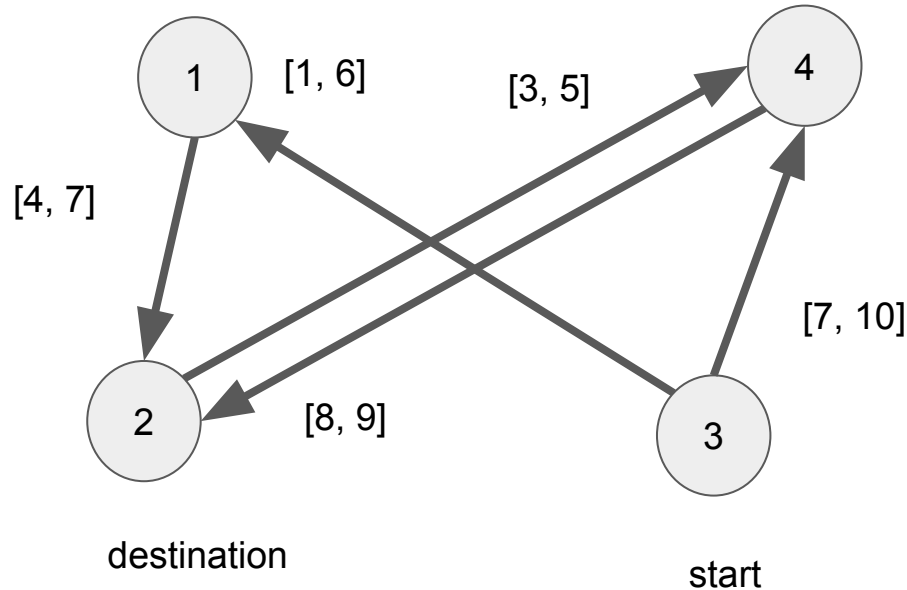


By storing the length of each block of text, we can traverse the list to find which node to operate on.

Security Badges - Statement

Given a directed graph, where each edge indicates a range of valid IDs that can traverse the edge, determine how many different IDs can traverse the entire graph.

Security Badges - Example



In all, 5 ids can get from room 3 to room 2.

4, 5, and 6 can go from 3 to 1, and then to 2.

8 and 9 can go from 3 to 4, and then to 2.

Security Badges - Solution

We can use a dfs or bfs to determine if a given ID can make it from the source to the destination.

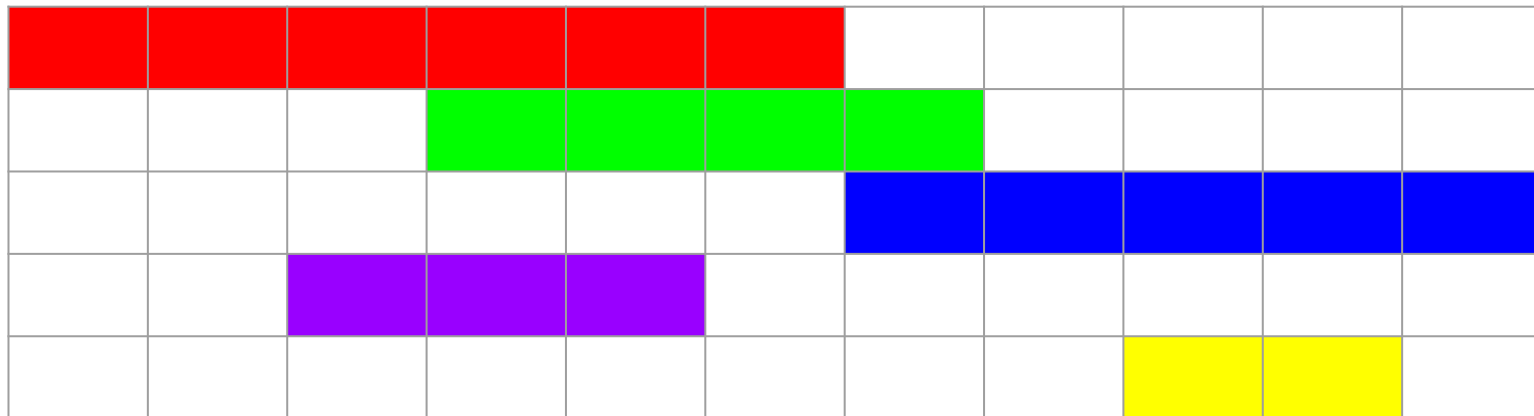
Key observation:

If ID i can make it from the source to the destination, and every edge that allows i to pass through allows $i + 1$ to pass through, then $i + 1$ will also pass through.

Security Badges - Solution

So, if our edges have the intervals:

[4, 7], [1, 6], [7, 10], [3, 5], and [8, 9]:



Security Badges - Solution

Interval $[a, b]$ is the same as interval $[a, b+1)$, so for every interval $[i, j]$, we can put i and $j + 1$ into a list, L , and then sort L in ascending order.

After that, we can iterate over the list. If the ID $L[k]$ passes, then every ID from $L[k]$ to $L[k+1]$, not inclusive, will pass as well, so we can add $L[k+1] - L[k]$ to our running total.

Security Badges - Code

```
class Edge {  
    int lo;  
    int hi;  
    Node to;  
  
    Edge(Node to, int lo, int hi) {  
        this.to = to;  
        this.lo = lo;  
        this.hi = hi;  
    }  
  
    public boolean fits(int id) {  
        return id >= lo && id <= hi;  
    }  
}
```

Security Badges - Code

```
class Node {
    int lastVisited = 0;
    List<Edge> E = new ArrayList<>();

    boolean visit(int id, int visitNum, Node dst) {
        lastVisited = visitNum;
        if (dst == this) return true;
        for (Edge e : E) {
            if (e.to.lastVisited != visitNum && e.fits(id)) {
                e.to.lastVisited = visitNum;
                if (e.to.visit(id, visitNum, dst)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Security Badges - Code

```
import java.util.*;
import java.io.*;

public class Security {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        String[] L = br.readLine().split(" ");
        int N = Integer.parseInt(L[0]);
        int M = Integer.parseInt(L[1]);

        L = br.readLine().split(" ");
        int S = Integer.parseInt(L[0])-1;
        int D = Integer.parseInt(L[1])-1;
        Node[] G = new Node[N];

        for (int i = 0; i < N; i++) G[i] = new Node();
        ...
    }
}
```

Security Badges - Code

...

```
int[] ids = new int[2*M];
```

```
for (int i = 0; i < M; i++) {  
    L = br.readLine().split(" ");  
    int U = Integer.parseInt(L[0]) - 1;  
    int V = Integer.parseInt(L[1]) - 1;  
    int lo = Integer.parseInt(L[2]);  
    int hi = Integer.parseInt(L[3]);  
    ids[2*i] = lo;  
    ids[2*i+1] = hi+1;  
    G[U].E.add(new Edge(G[V], lo, hi));  
}
```

```
Arrays.sort(ids);
```

...

Security Badges - Code

```
...
Arrays.sort(ids);

int tot = 0;

for (int i = 0; i < ids.length - 1; i++) {
    if (ids[i] == ids[i+1]) continue;
    if (G[S].visit(ids[i], i+1, G[D])) {
        tot += ids[i+1] - ids[i];
    }
}

System.out.println(tot);
}
```

Move Away - Statement

Given up to 50 circles, what is the furthest you can be from the origin while still being inside every circle?

Move Away - Solution

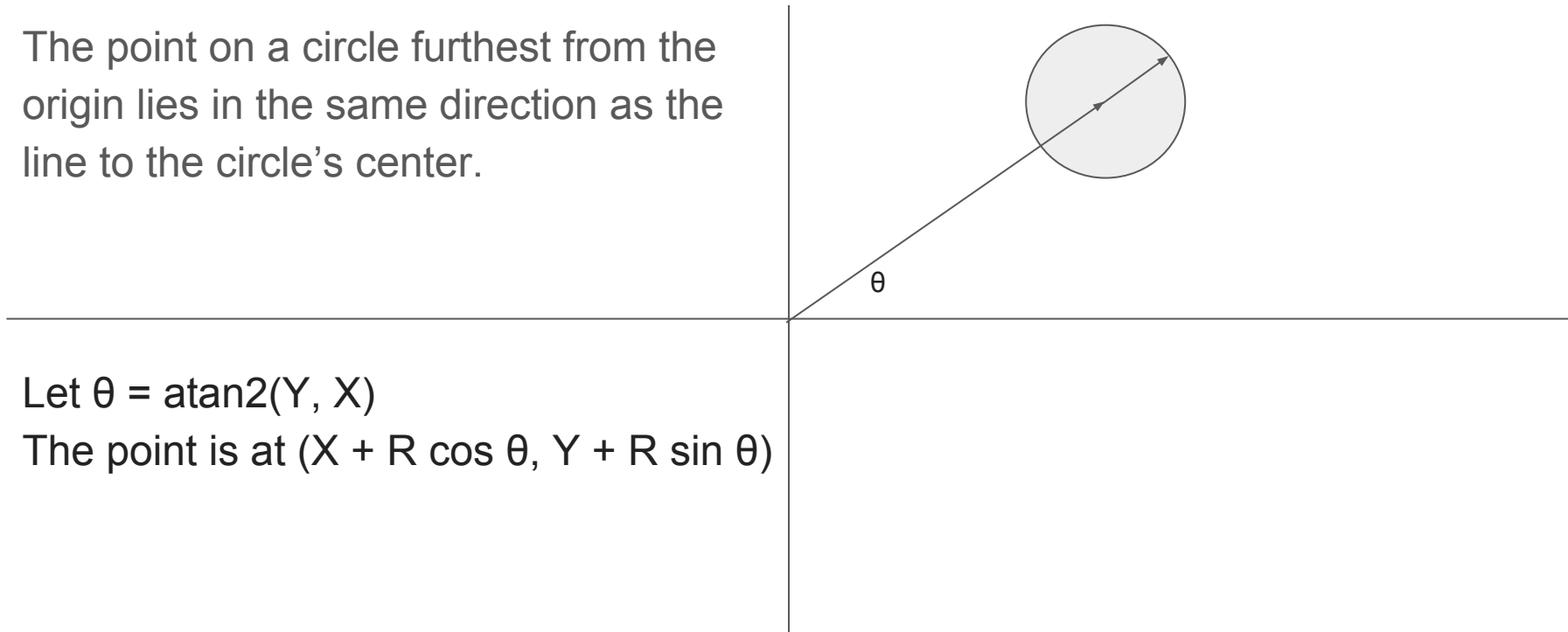
The critical point will be one of the following:

- 1) The point on a circle furthest from the origin.
- 2) The intersection of two circles.

We just need to find which of these both lies inside every circle and is furthest from the origin.

Move Away - Solution

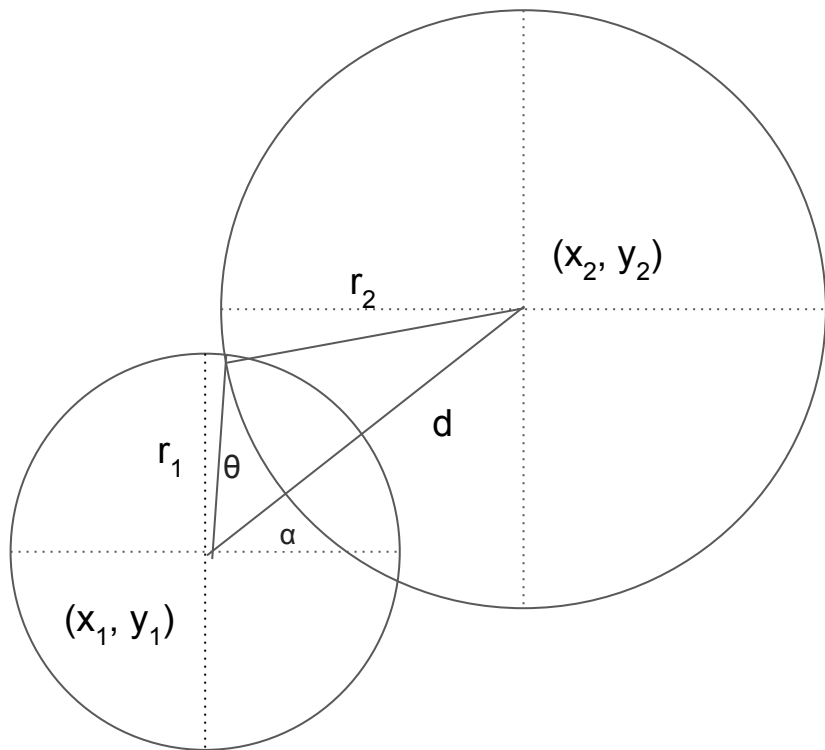
The point on a circle furthest from the origin lies in the same direction as the line to the circle's center.



Let $\theta = \text{atan2}(Y, X)$

The point is at $(X + R \cos \theta, Y + R \sin \theta)$

Move Away - Solution



To find the intersections of 2 circles:

Let $\alpha = \text{atan2}(y_2 - y_1, x_2 - x_1)$

Let $d = \|(x_2 - x_1, y_2 - y_1)\|$

By the law of cosines,

Let $\theta = \arccos((r_1^2 + d^2 - r_2^2) / (2r_1d))$

Then we have two points:

$$(x_1 + r_1 \cos(\alpha + \theta), y_1 + r_1 \sin(\alpha + \theta))$$

and

$$(x_1 + r_1 \cos(\alpha - \theta), y_1 + r_1 \sin(\alpha - \theta))$$

Move Away - Code

```
class Point {  
    double X;  
    double Y;  
  
    Point(double X, double Y) {  
        this.X = X;  
        this.Y = Y;  
    }  
  
    double dist(Point o) {  
        double dX = X - o.X;  
        double dY = Y - o.Y;  
        return Math.sqrt(dX * dX + dY * dY);  
    }  
}
```

Move Away - Code

```
class Circle {
    static double EPSILON = 1e-9;

    Point Center;
    double R;

    Circle(double X, double Y, double R) {
        this.Center = new Point(X, Y);
        this.R = R;
    }

    Point[] intersections(Circle o) {
        double dist = Center.dist(o.Center);
        if (dist + o.R + EPSILON < R) return new Point[]{};
        double theta = Math.acos((R*R + dist * dist - o.R * o.R) / (2 * R * dist));
        double angle = Math.atan2(o.Center.Y - Center.Y, o.Center.X - Center.X);

        Point p1 = new Point(Center.X + Math.cos(theta + angle) * R, Center.Y + Math.sin(theta + angle) * R);
        Point p2 = new Point(Center.X + Math.cos(angle - theta) * R, Center.Y + Math.sin(angle - theta) * R);

        return new Point[]{p1, p2};
    }
}
```

Move Away - Code

```
...
Point furthest() {
    double theta = Math.atan2(Center.Y, Center.X);
    return new Point(Center.X + R * Math.cos(theta), Center.Y + R * Math.sin(theta));
}

boolean contains(Point p) {
    double dist = Center.dist(p);
    return dist < R + EPSILON;
}
}
```


Move Away - Code

```
import java.util.*;
import java.text.*;

public class Move {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        int N = in.nextInt();
        Circle[] C = new Circle[N];

        for (int i = 0; i < N; i++) C[i] = new Circle(in.nextInt(), in.nextInt(), in.nextInt());

        List<Point> candidates = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            candidates.add(C[i].furthest());

            for (int j = i + 1; j < N; j++) {
                for (Point p : C[i].intersections(C[j])) {
                    candidates.add(p);
                }
            }
        }
        ...
    }
}
```

Move Away - Code

```
...
double best = 0.0;
Point O = new Point(0, 0);
for (Point p : candidates) {
    boolean valid = true;
    for (Circle c : C) {
        if (!c.contains(p)) {
            valid = false;
            break;
        }
    }

    if (valid) {
        best = Math.max(best, O.dist(p));
    }
}

DecimalFormat fmt = new DecimalFormat("0.000");
System.out.println(fmt.format(best));
}
```

Unsatisfying - Statement

Given a list of disjunctions (two variables, which may or may not be negated, ORed together), determine the minimum number of negation-less disjunctions to add that would make the list unsatisfiable.

Unsatisfying - Example

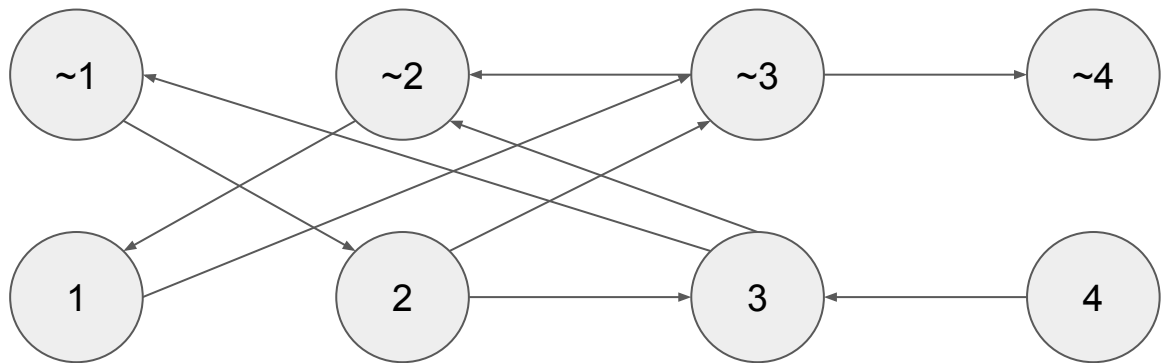
1		2
~1		~3
~2		3
3		~4
~2		~3

If we add $2 \mid 3$ to the list, then no assignment of true or false to the given variables will allow for every statement to be true, so the answer is 1.

Unsatisfying - Solution

Build a graph containing every variable and its complement as vertices. If we have a disjunction $u \vee v$ that means that if $\sim u$ is true, then v must also be true and if $\sim v$ is true, then u must be true, so we will add both of these directed edges.

1		2
~ 1		~ 3
~ 2		3
3		~ 4
~ 2		~ 3



Unsatisfying - Solution

See which nodes are reachable from each starting node.

src\dst	~1	1	~2	2	~3	3	~4	4
~1								
1								
~2								
2								
~3								
3								
~4								
4								

Unsatisfying - Solution

If we can reach $\sim u$ from u : (u must be false)

If we can also reach u from $\sim u$, we don't need to add any disjunctions.

Otherwise, we can add one disjunction. (u, u) .

If we can reach $\sim v$ from u : ($u \rightarrow \sim v$)

If we can also reach v from u ($u \rightarrow v$), we can add one disjunction. (u, u)

Otherwise, we can add 2 disjunctions $(u, u), (v, v)$.

Find whichever case we satisfy with minimal disjunctions added. If we don't find any, it is impossible.

Flipping Out - Statement

We are given a set of patterns of coin flips, and a list of coin flip results.

If the total number of times the patterns appear is even, we add tails to the results.

If the total number of times the patterns appear is odd, we add heads to the results.

Let's say a pattern is missing from the list of patterns. How many different patterns could we add to satisfy the given results?

Flipping Out - Example

Suppose our patterns are:

H

TTH

HHTHT

TH

And that they, along with one other pattern generate the sequence:

TTTHTT

This can only happen if the other pattern is TTT

Flipping Out - Solution

Flipping out relies on an efficient algorithm for counting the matches of the incoming strings.

The best for this is probably the Aho-Corasick algorithm, which constructs a DFA to detect when each pattern is matched.

For every time we have a mismatch, we can examine all suffixes of the string leading to each mismatch that don't appear anywhere previously, and then remove any that already appear in the original list of patterns.

|

Exciting Finish - Statement

Going into the final round of a contest, we know the point values of every participant. The judges will choose to allot X total points in the final round, and they will announce the scores in non-decreasing order (in the event of ties, they will start with the contestant with the smallest initial point value). Additionally, they don't want for there to be any ties in the final results.

How many different final rankings are possible such that every point announcement will result in the 1st place position changing?

Exciting Finish - Example

A has 3 points

B has 1 point

C has 4 points

And we have 12 points to give away. The following final rankings will be exciting:

A gets 2 points, B gets 5 points, and C gets 5 points. (CBA)

A gets 2 points, C gets 2 points, and B gets 8 points

OR

A gets 2 points, C gets 3 points, and B gets 7 points (BCA)

B gets 4 points, A gets 4 points, C gets 4 points (CAB).

Since CBA, BCA, and CAB are all possible valid final orderings, the solution is 3.

Exciting Finish - Solution

Exciting Finish's solution is similar to the $2^n n^2$ approach to travelling salesman. We can determine the number of ways to arrive at a given subset of contestants having been announced, with contestant i being the last contestant announced, and then determine if contestant j , not in that subset, can have their score announced after that.

Instead of keeping track of the minimum distance, we will track the total number of possible arrangements for each total amount of points awarded.