Segment Trees

COMP 4951: Advanced Problem Solving
Micah Stairs

Outline

- Motivation (Range Queries using Arrays)
- Segment Tree Operations, Structure, and Construction
- Single Value Updates and Range Queries
- **Problem #1: Excellent Engineers**
- Range Updates
- Coordinate Compression
- **Problem #2: Worst Weather Ever**

Motivation (Range Queries using Arrays)

Range Queries

0	1	2	3	4	5	6	7	8	9	10
3	2	5	3	0	1	12	4	8	21	2

- sum(0, 4) = 3 + 2 + 5 + 3 + 0 = <u>13</u>
- sum(3, 5) = 3 + 0 + 1 = <u>4</u>
- sum(7, 7) = <u>4</u>

Range Queries

0	1	2	3	4	5	6	7	8	9	10
3	2	5	3	0	1	12	4	8	21	2

- Naive solution: O(n) per query
- Other solution: O(n) pre-processing, then O(1) per query

Sums:

0	1	2	3	4	5	6	7	8	9	10
3	5	10	13	13	14	26	30	38	59	61

Range Queries

0	1	2	3	4	5	6	7	8	9	10
3	2	5	3	0	1	12	4	8	21	2

Sums:

0	1	2	3	4	5	6	7	8	9	10
3	5	10	13	13	14	26	30	38	59	61

- sum(4,6) = sum(0,6) - sum(0,3) = 26 - 13 = 13
- sum(8,8) = sum(0,8) - sum(0,7) = 38 - 30 = 8
- sum(i,j) = sum(0,j) - sum(0, i-1)

Range Queries

Update
↓

0	1	2	3	4	5	6	7	8	9	10
3	2	5	3	0	1	12	4	8	21	2

Sums:

0	1	2	3	4	5	6	7	8	9	10
3	5	10	?	?	?	?	?	?	?	?

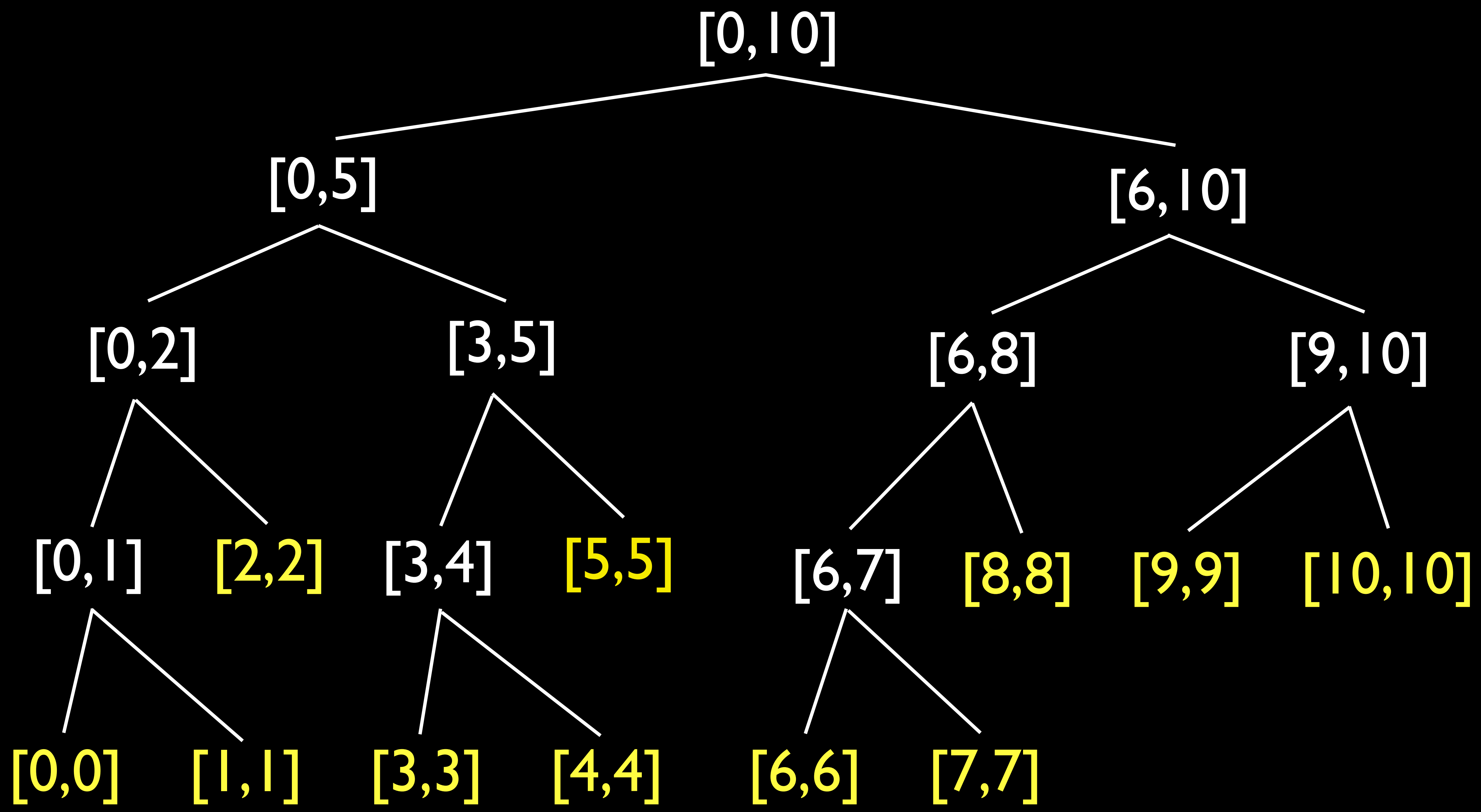
- Updates are O(n)
- Is there a better approach?

Segment Tree Operations, Structure, and Construction

Segment Tree Operations

- update(i, value) is O(log(n))
- sum(i,j) is O(log(n))
- max(i,j) is O(log(n))
- min(i,j) is O(log(n))

Structure (Binary Tree)

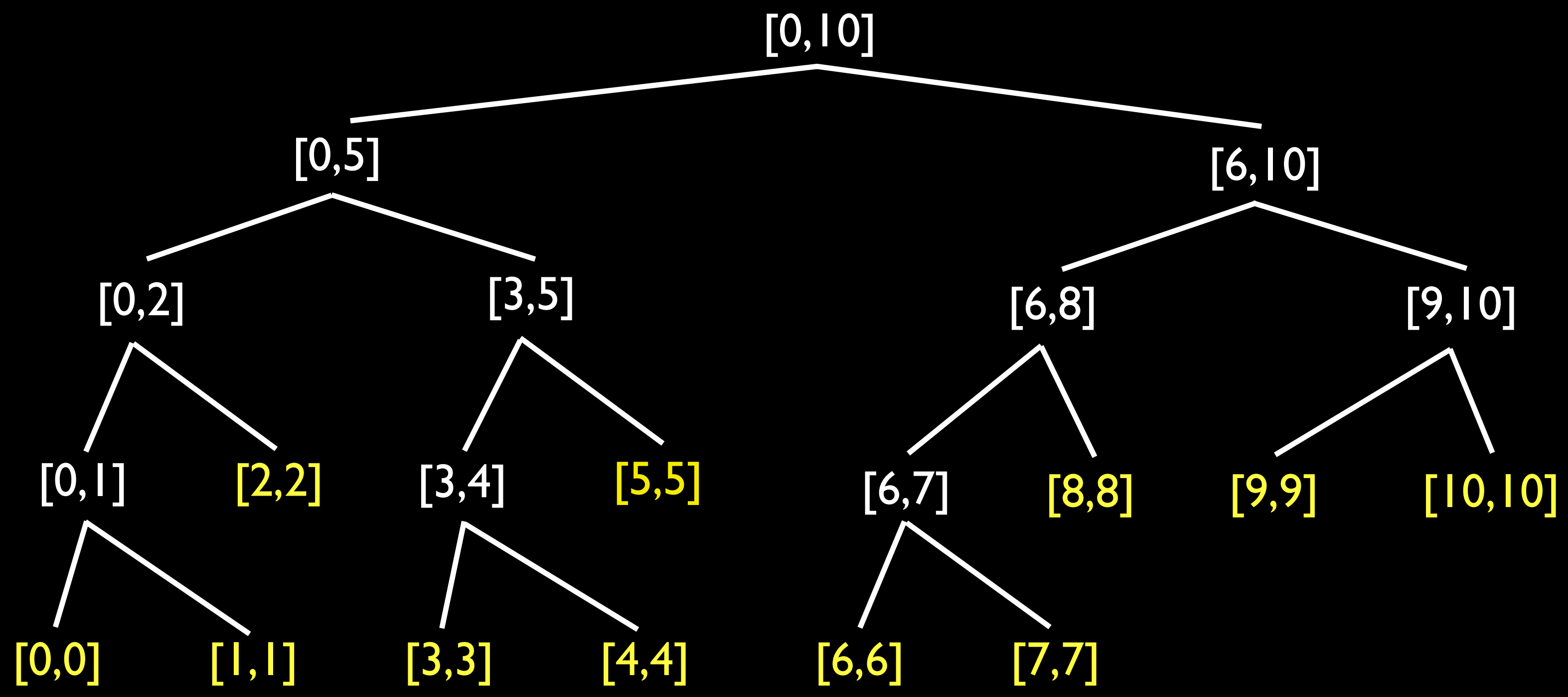


Construction

Time: O(n), Space: O(n)

```
class Node {

  int minPos, maxPos;
  int min = 0, max = 0, sum = 0;
  Node left, right;

  public Node(int l, int r) {

    minPos = l;
    maxPos = r;

    // Reached leaf
    if (l == r) {
      left = right = null;

    // Add children
    } else {
      int mid = (l + r)/2;
      left = new Node(l, mid);
      right = new Node(mid + 1, r);
    }

  }

}
```

```
Node root = new Node(0, 10);
```



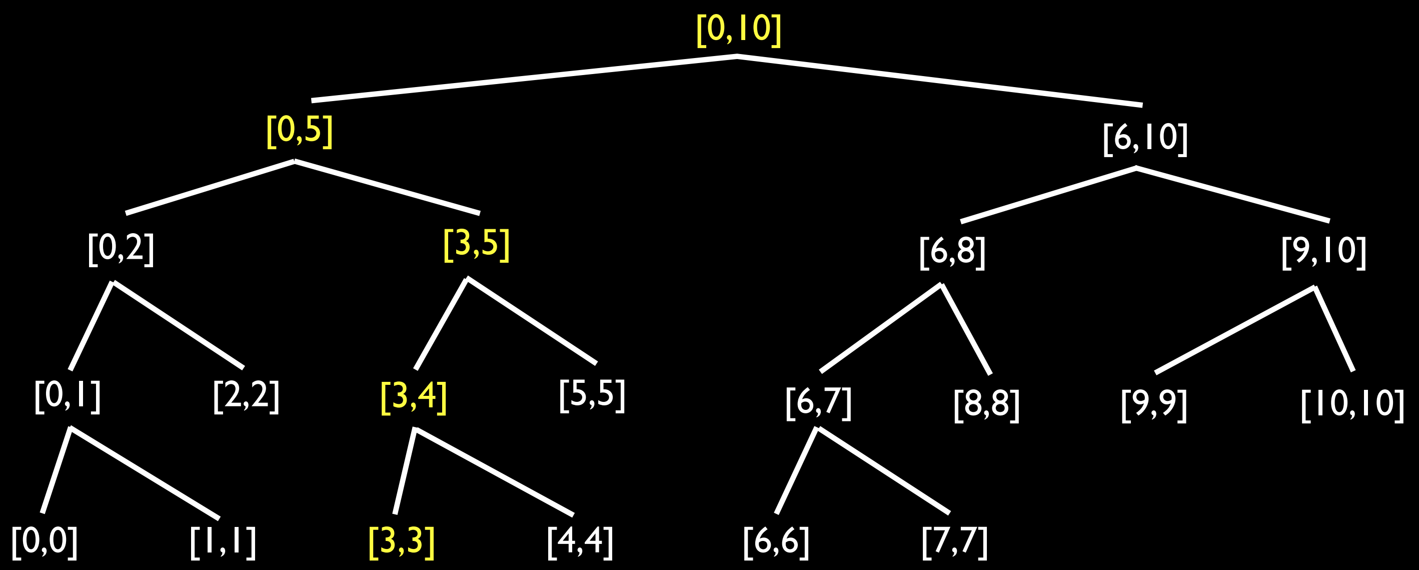
Single Value Updates and Range Queries

Update Value

Time: O(log(n))

```
// Recursively update the value in the specified postion (returns the change in sum)
public int update(int pos, int value) {

  // Reached leaf
  if (minPos == maxPos) {
    int oldValue = min = max;
    min = max = value;
    return value - oldValue;

  // Update child
  } else {

    int changeInSum;

    // Go left
    if (pos <= (minPos + maxPos)/2)
      changeInSum = left.update(pos, value);

    // Go right
    else
      changeInSum = right.update(pos, value);

    // Update properties
    min = Math.min(left.min, right.min);
    max = Math.max(left.max, right.max);
    sum += changeInSum;

    return changeInSum;

  }

}
```

```
update(3, value);
```

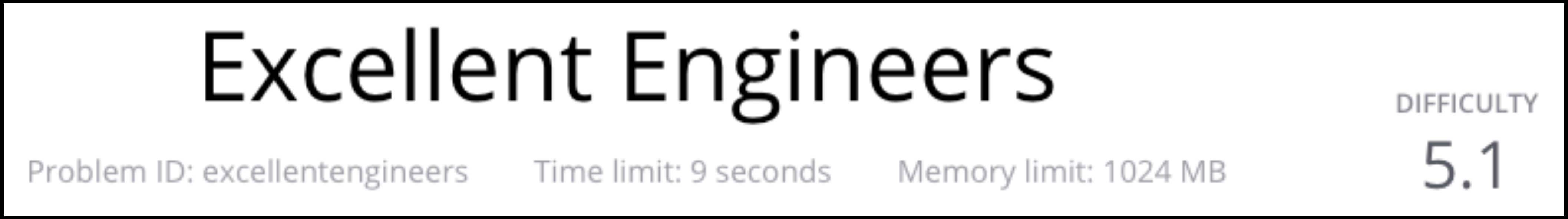


Queries

Time: O(log(n))

```
public int sum(int leftPos, int rightPos) {

  // Node's range fits inside query range
  if (leftPos <= minPos && maxPos <= rightPos) {

    return sum;

  // Ranges do not overlap
  } else if (rightPos < minPos || leftPos > maxPos) {

    return 0;

  // Ranges partially overlap
  } else {

    return left.sum(leftPos, rightPos) + right.sum(leftPos, rightPos);

  }

}
```

Problem #1: Excellent Engineers

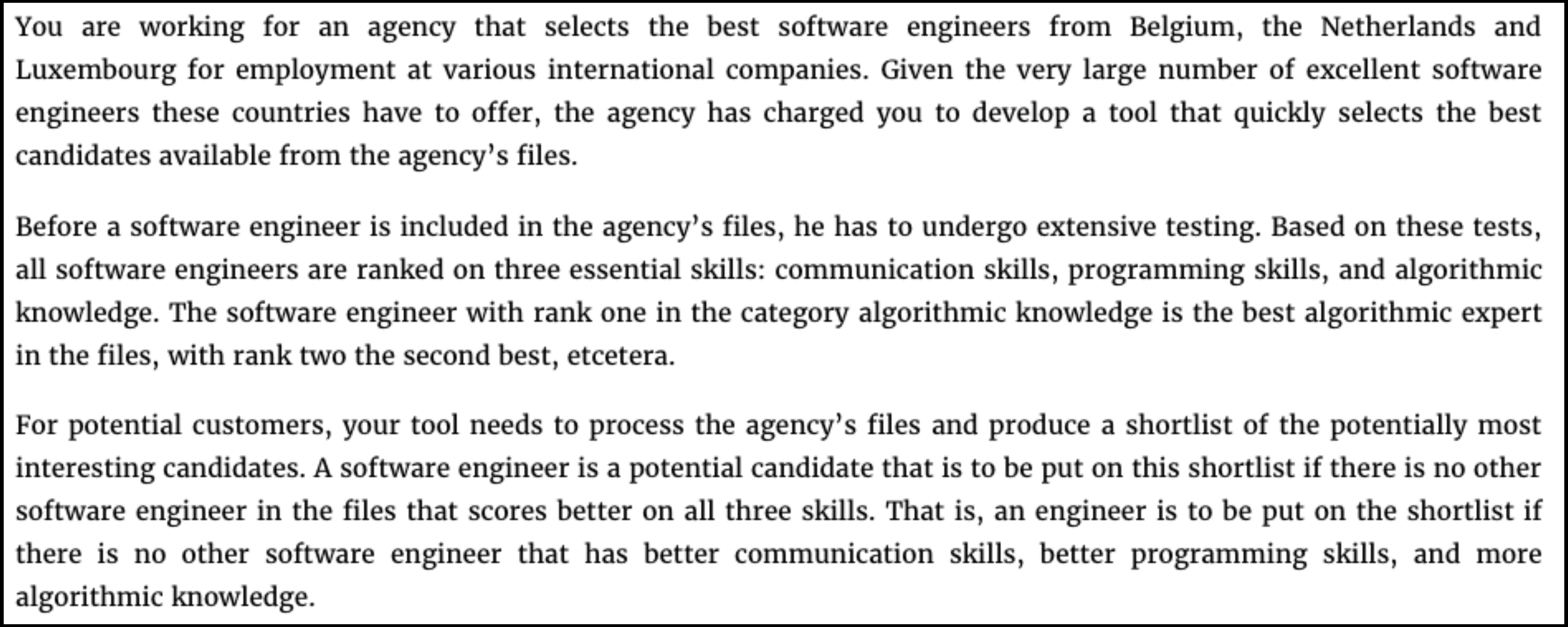


Excellent Engineers

Problem ID: excellentengineers　Time limit: 9 seconds　Memory limit: 1024 MB

DIFFICULTY
5.1

Problem Description

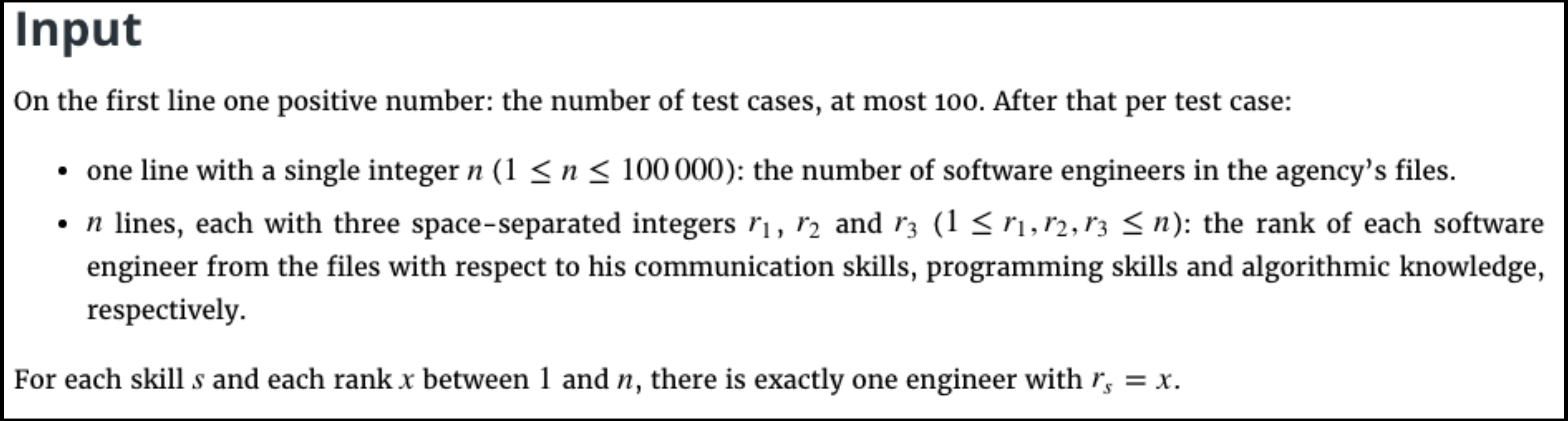


You are working for an agency that selects the best software engineers from Belgium, the Netherlands and Luxembourg for employment at various international companies. Given the very large number of excellent software engineers these countries have to offer, the agency has charged you to develop a tool that quickly selects the best candidates available from the agency's files.

Before a software engineer is included in the agency's files, he has to undergo extensive testing. Based on these tests, all software engineers are ranked on three essential skills: communication skills, programming skills, and algorithmic knowledge. The software engineer with rank one in the category algorithmic knowledge is the best algorithmic expert in the files, with rank two the second best, etcetera.

For potential customers, your tool needs to process the agency's files and produce a shortlist of the potentially most interesting candidates. A software engineer is a potential candidate that is to be put on this shortlist if there is no other software engineer in the files that scores better on all three skills. That is, an engineer is to be put on the shortlist if there is no other software engineer that has better communication skills, better programming skills, and more algorithmic knowledge.

Input and Output

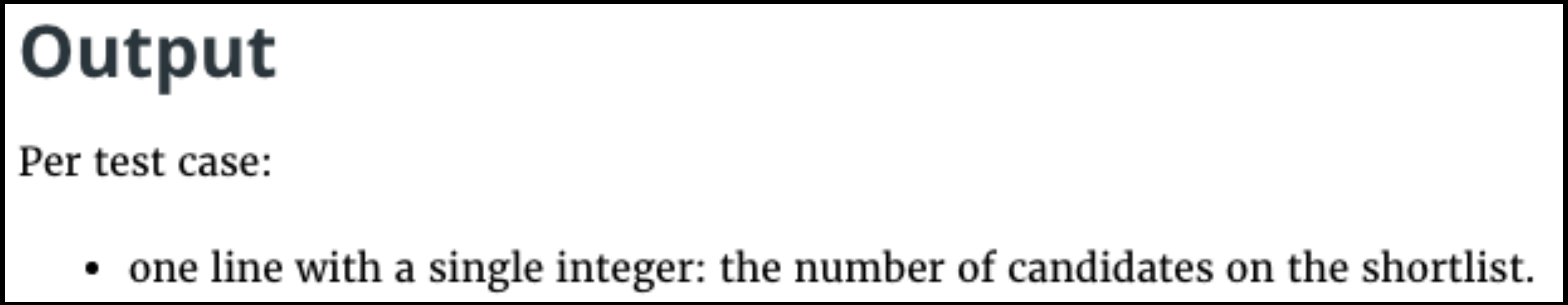


Input

On the first line one positive number: the number of test cases, at most 100. After that per test case:

- one line with a single integer n ($1 \le n \le 100\,000$): the number of software engineers in the agency's files.
- n lines, each with three space-separated integers r_1, r_2 and r_3 ($1 \le r_1, r_2, r_3 \le n$): the rank of each software engineer from the files with respect to his communication skills, programming skills and algorithmic knowledge, respectively.

For each skill s and each rank x between 1 and n, there is exactly one engineer with $r_s = x$.

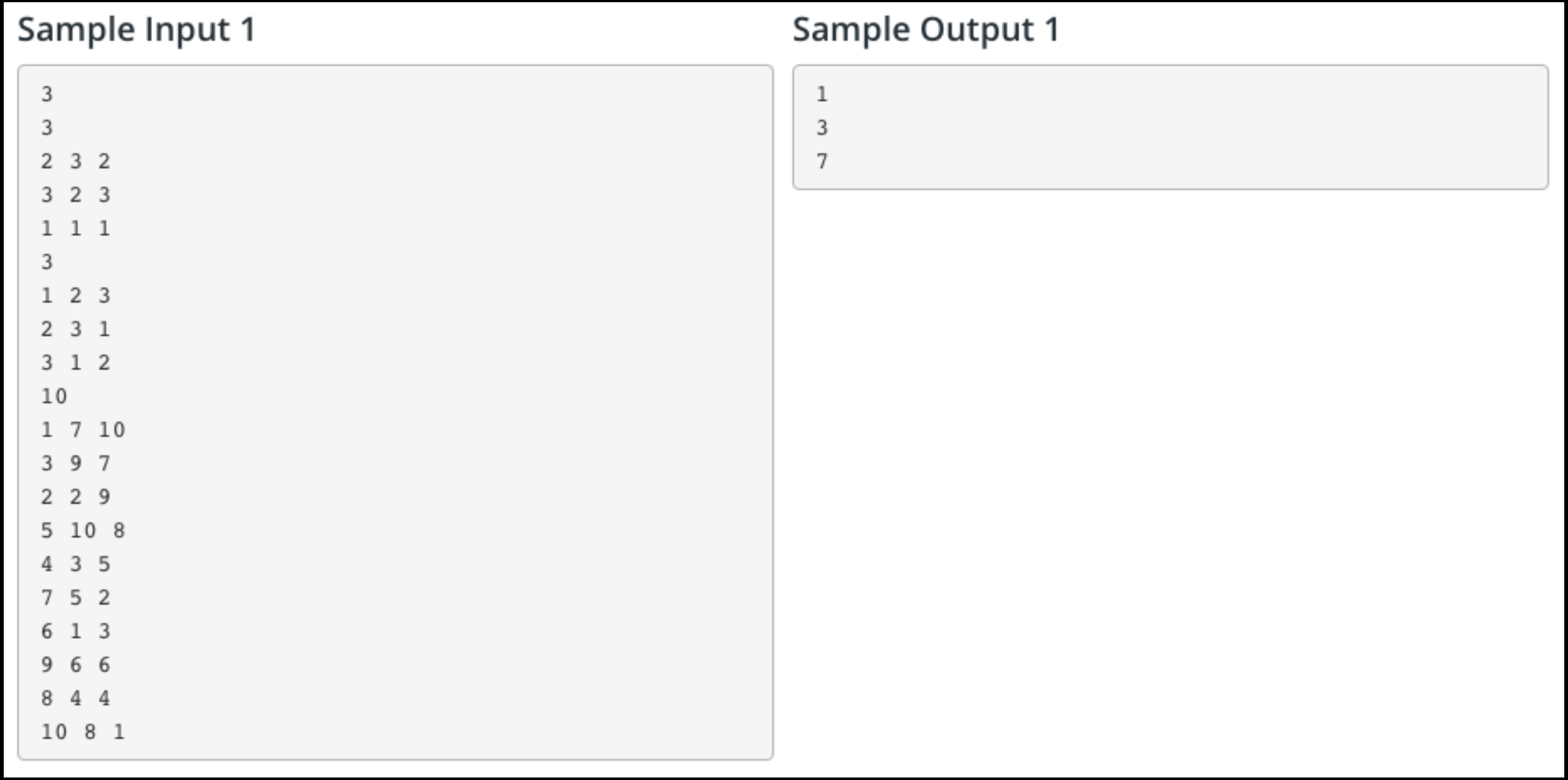


Output

Per test case:

- one line with a single integer: the number of candidates on the shortlist.

Test Data



Sample Input 1

```
3
3
2 3 2
3 2 3
1 1 1
3
1 2 3
2 3 1
3 1 2
10
1 7 10
3 9 7
2 2 9
5 10 8
4 3 5
7 5 2
6 1 3
9 6 6
8 4 4
10 8 1
```

Sample Output 1

```
1
3
7
```

Solution

```java
// Get the number of software engineers
int n = Integer.valueOf(br.readLine());
int[] secondSkill = new int[n];
int[] thirdSkill = new int[n];

// Read in software engineers in agency files
for (int i = 0; i < n; i++) {
  String[] line = br.readLine().split(" ");
  int skill1 = Integer.valueOf(line[0]) - 1;
  int skill2 = Integer.valueOf(line[1]) - 1;
  int skill3 = Integer.valueOf(line[2]) - 1;
  secondSkill[skill1] = skill2;
  thirdSkill[skill1] = skill3;
}
```

Solution

```java
// Set up segment tree
Node root = new Node(0, n - 1);

// Check software engineers in order of first skill
// NOTE: This means that software engineers we check later will be worse in
//       at least one skill, so they can be ignored
int count = 0;
for (int i = 0; i < n; i++) {

  // Find the best third skill ranking out of all of the engineers who's
  // second skills are better
  int previousBest = root.min(0, secondSkill[i]);

  // If this engineer's third skill ranking is worse, then there is at least
  // one engineer who is better in all three skill categories, but if this
  // skill ranking is better, then we increment the number of candidates
  // on the short-list
  if (thirdSkill[i] < previousBest)
    count++;

  // Update segment tree
  root.update(secondSkill[i], thirdSkill[i]);

}

System.out.println(count);
```

Solution

```
class Node {

  // There are only 100,000 engineers, so this value is sufficiently large
  static final int INF = 1_000_000;

  int minPos, maxPos;
  int min = INF;
  Node left, right;

  public Node(int l, int r) {

    minPos = l;
    maxPos = r;

    // Reached leaf
    if (l == r) {
      left = right = null;

    // Add children
    } else {
      int mid = (l + r)/2;
      left = new Node(l, mid);
      right = new Node(mid + 1, r);
    }

  }
```

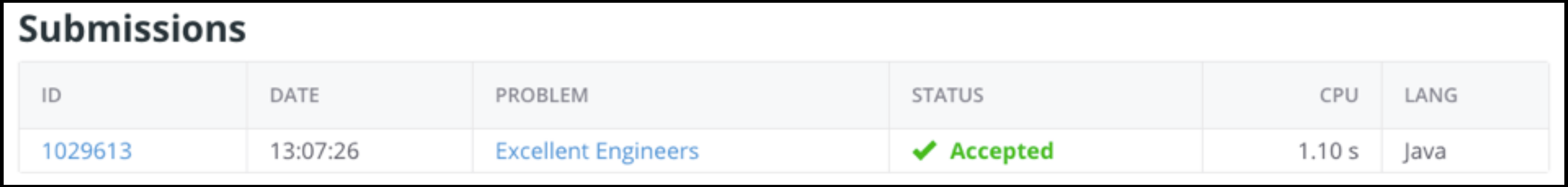
Solution

```
public void update(int pos, int value) {

  // Reached leaf
  if (minPos == maxPos) {
    min = Math.min(min, value);

  // Update child
  } else {

    // Go left
    if (pos <= (minPos + maxPos)/2)
      left.update(pos, value);

    // Go right
    else
      right.update(pos, value);

    // Update minimum
    min = Math.min(left.min, right.min);

  }

}
```

Solution

```java
// Return minimum value in specified range
public int min(int leftPos, int rightPos) {

  // Node's range fits inside query range
  if (leftPos <= minPos && maxPos <= rightPos) {

    return min;

  // Ranges do not overlap
  } else if (rightPos < minPos || leftPos > maxPos) {

    return INF;

  // Ranges partially overlap
  } else {

    return Math.min(left.min(leftPos, rightPos), right.min(leftPos, rightPos));

  }

}
```

Solution



Submissions

ID	DATE	PROBLEM	STATUS	CPU	LANG
1029613	13:07:26	Excellent Engineers	✓ Accepted	1.10 s	Java

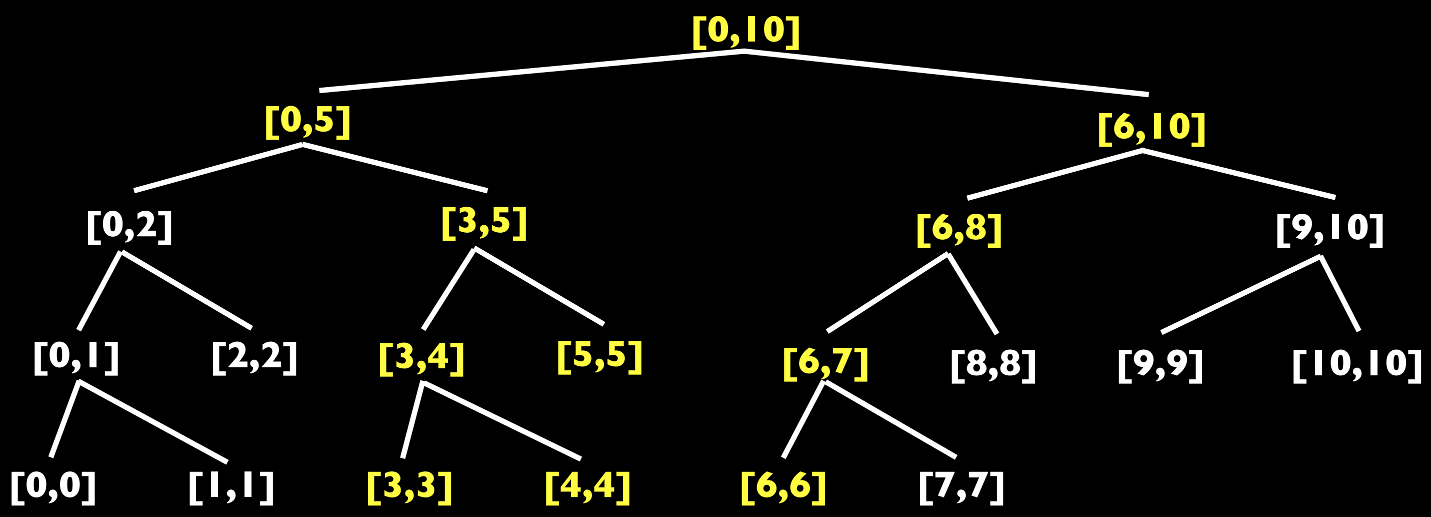
Range Updates

Update Range

Time: O(n)

```
// Recursively change all values in the specified range by a particular amount (returns the change in sum)
public int update(int leftPos, int rightPos, int valueChange) {

  // Reached leaf
  if (minPos == maxPos) {
    min += valueChange;
    max += valueChange;
    sum += valueChange;
    return valueChange;

  // Update children
  } else {

    int changeInSum = 0;

    // Go left
    if (leftPos <= (minPos + maxPos)/2)
      changeInSum += left.update(leftPos, rightPos, valueChange);

    // Go right
    if (rightPos >= 1 + (minPos + maxPos)/2)
      changeInSum += right.update(leftPos, rightPos, valueChange);

    // Update properties
    min = Math.min(left.min, right.min);
    max = Math.max(left.max, right.max);
    sum += changeInSum;

    return changeInSum;

  }
}
```

```
update(3, 6, valueChange);
```



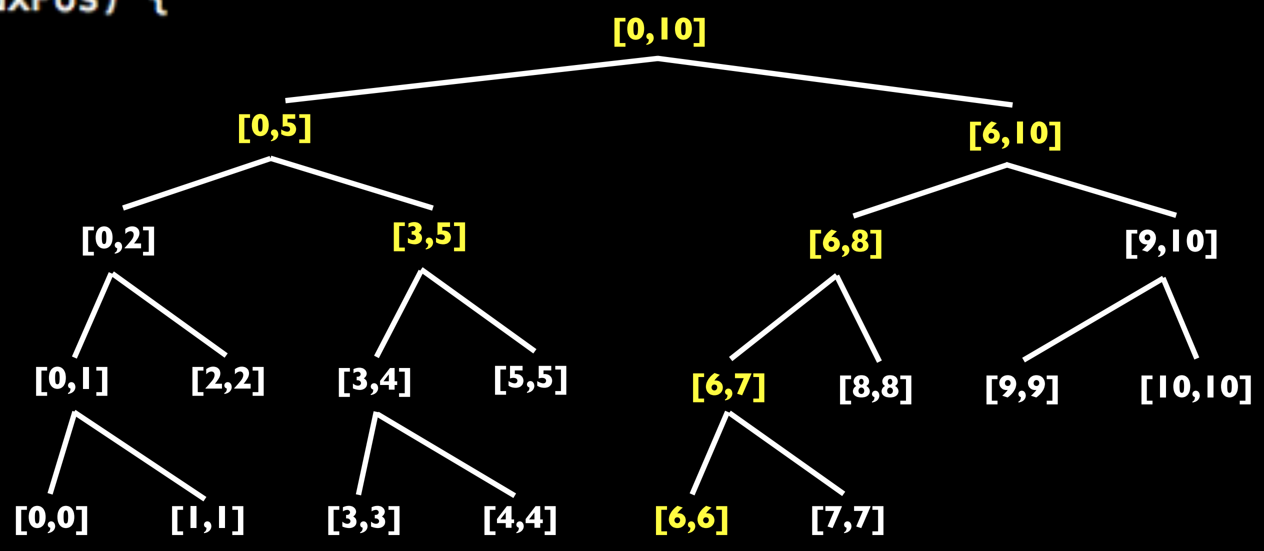
Using Lazy Propagation

Time: O(log(n))

- Add instance variable: `int lazyPropagation = 0;`

```
// Change all values in the specified range by a certain amount (returns the change in sum)
public int update(int leftPos, int rightPos, int valueChange) {

  // Do lazy updates to children
  doLazyUpdates();

  int changeInSum = 0;

  // Node's range fits inside query range
  if (leftPos <= minPos && maxPos <= rightPos) {
    . . .


  // Ranges do not overlap
  } else if (rightPos < minPos || leftPos > maxPos) {

    // Do nothing

  // Ranges partially overlap
  } else {
    . . .

  }

  return changeInSum;
}
```

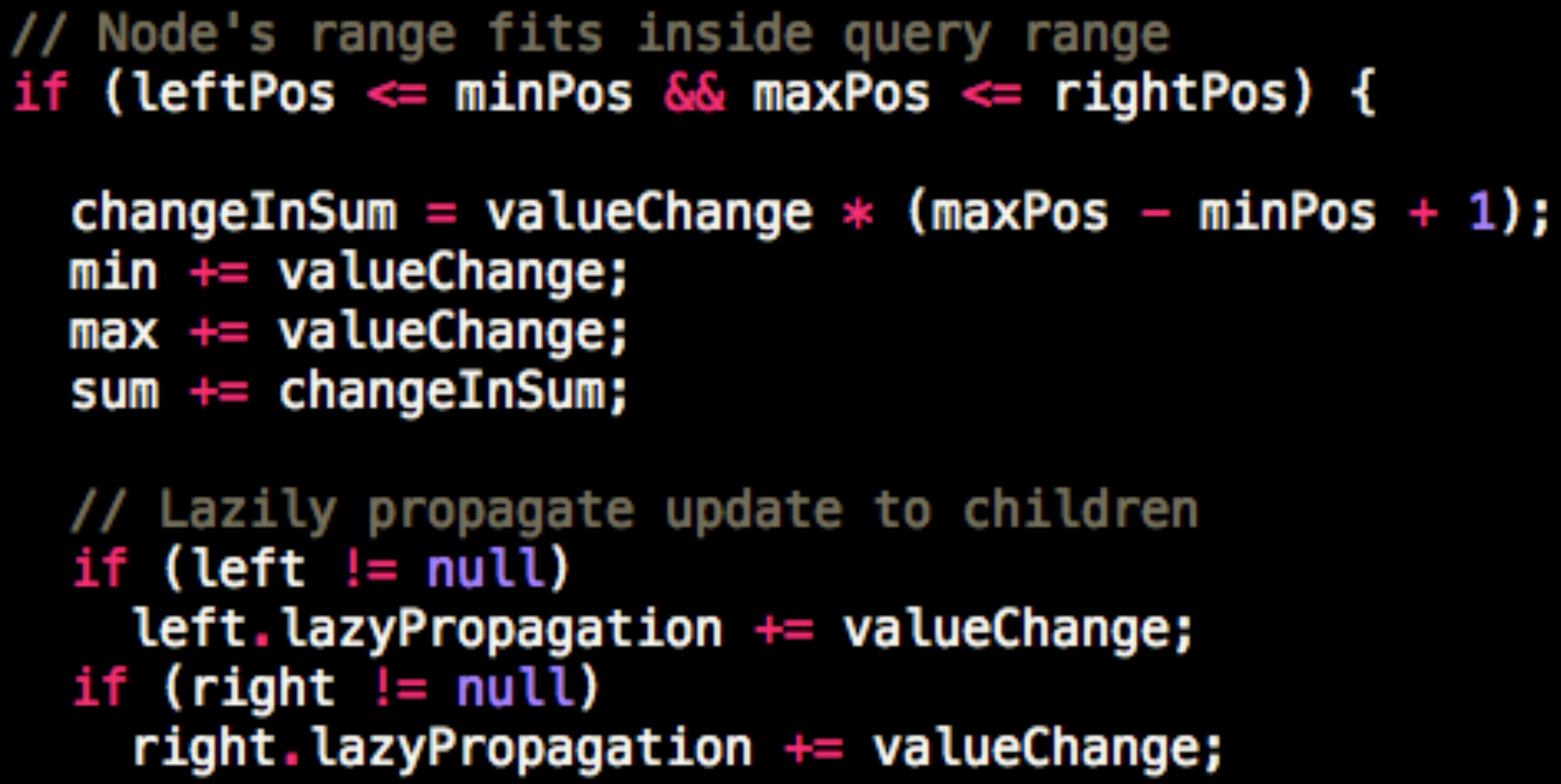
```
update(3, 6, valueChange);
```



Using Lazy Propagation

Time: O(log(n))

...



```
// Node's range fits inside query range
if (leftPos <= minPos && maxPos <= rightPos) {

  changeInSum = valueChange * (maxPos - minPos + 1);
  min += valueChange;
  max += valueChange;
  sum += changeInSum;

  // Lazily propagate update to children
  if (left != null)
    left.lazyPropagation += valueChange;
  if (right != null)
    right.lazyPropagation += valueChange;
```

...

Using Lazy Propagation

Time: O(log(n))

```
...

  // Ranges do not overlap
  } else if (rightPos < minPos || leftPos > maxPos) {

    // Do nothing

  // Ranges partially overlap
  } else {

    changeInSum += left.update(leftPos, rightPos, valueChange);
    changeInSum += right.update(leftPos, rightPos, valueChange);

    // Update properties
    min = Math.min(left.min, right.min);
    max = Math.max(left.max, right.max);
    sum += changeInSum;

  }

  return changeInSum;

}
```

Coordinate Compression

Coordinate Compression

```java
long minPos, maxPos;
long max = -1;
Node left, right;

public Node(int leftIndex, int rightIndex, List<Long> coordinates) {

  // Get compressed coordinates
  minPos = coordinates.get(leftIndex);
  maxPos = coordinates.get(rightIndex);

  // Reached leaf
  if (leftIndex == rightIndex) {
    left = right = null;

  // Add children
  } else {
    int midIndex = (leftIndex + rightIndex)/2;
    left = new Node(leftIndex, midIndex, coordinates);
    right = new Node(midIndex + 1, rightIndex, coordinates);
  }

}
```

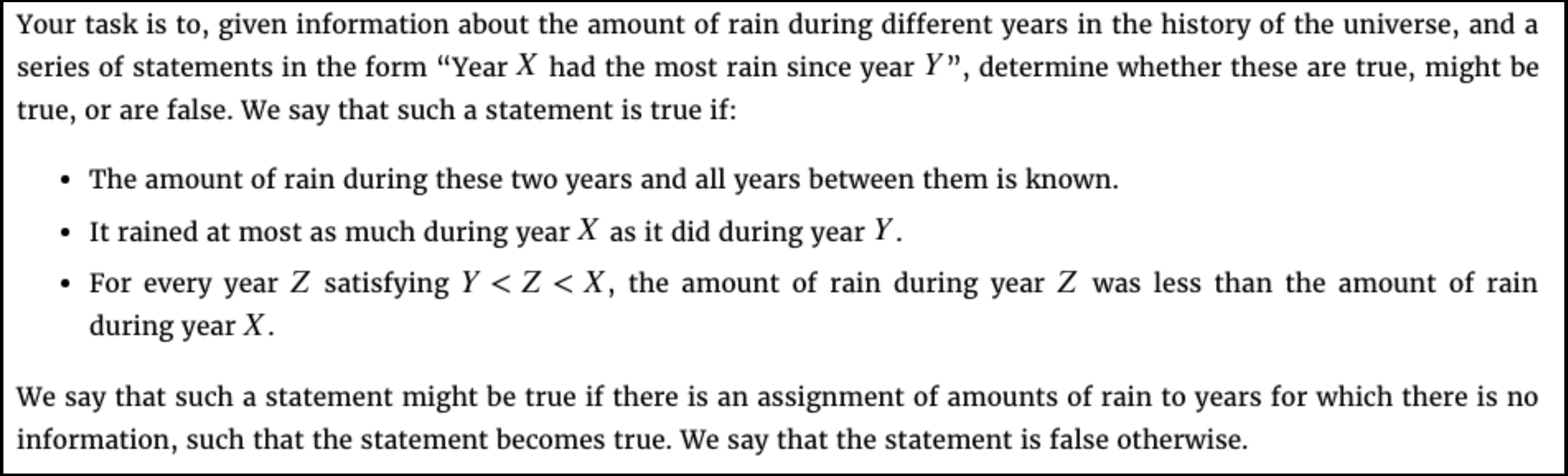
Problem #2: Worst Weather Ever

Worst Weather Ever

Problem ID: worstweather Time limit: 2 seconds Memory limit: 1024 MB

DIFFICULTY 7.7

Problem Description

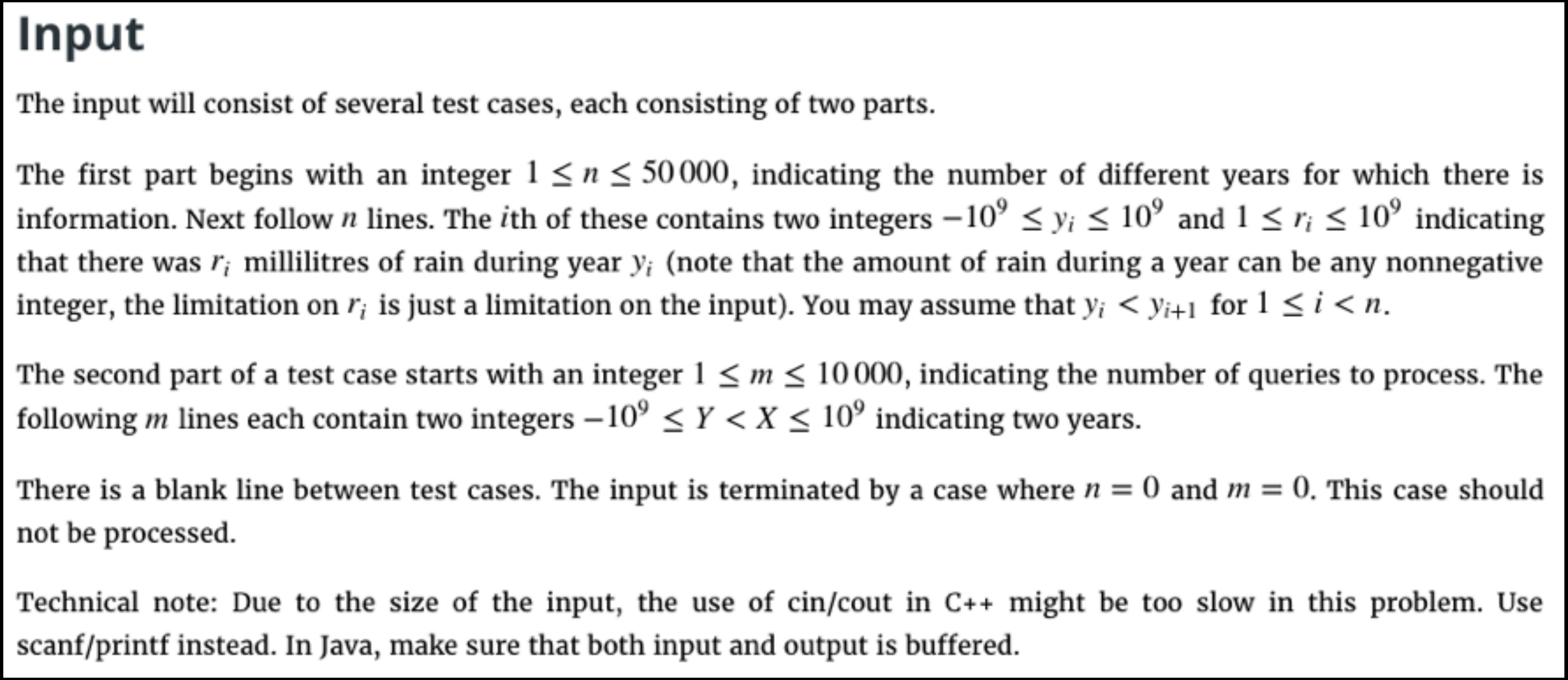


Your task is to, given information about the amount of rain during different years in the history of the universe, and a series of statements in the form “Year X had the most rain since year Y”, determine whether these are true, might be true, or are false. We say that such a statement is true if:

- The amount of rain during these two years and all years between them is known.
- It rained at most as much during year X as it did during year Y.
- For every year Z satisfying $Y < Z < X$, the amount of rain during year Z was less than the amount of rain during year X.

We say that such a statement might be true if there is an assignment of amounts of rain to years for which there is no information, such that the statement becomes true. We say that the statement is false otherwise.

Input and Output



Input

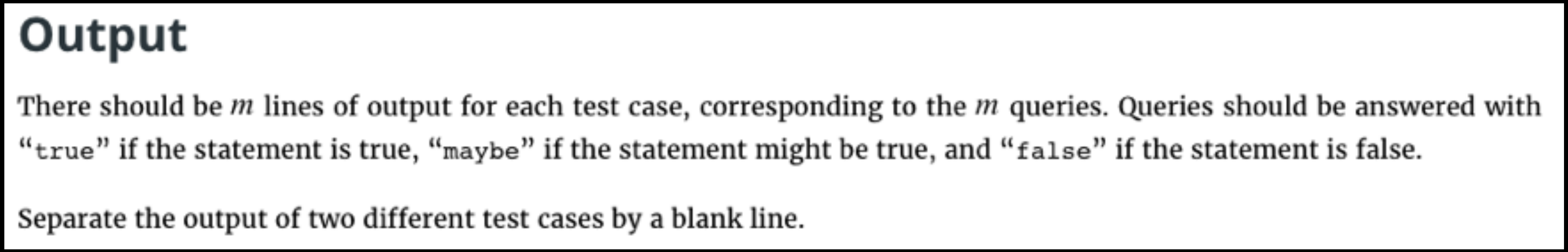
The input will consist of several test cases, each consisting of two parts.

The first part begins with an integer $1 \leq n \leq 50\,000$, indicating the number of different years for which there is information. Next follow n lines. The ith of these contains two integers $-10^9 \leq y_i \leq 10^9$ and $1 \leq r_i \leq 10^9$ indicating that there was r_i millilitres of rain during year y_i (note that the amount of rain during a year can be any nonnegative integer, the limitation on r_i is just a limitation on the input). You may assume that $y_i < y_{i+1}$ for $1 \leq i < n$.

The second part of a test case starts with an integer $1 \leq m \leq 10\,000$, indicating the number of queries to process. The following m lines each contain two integers $-10^9 \leq Y < X \leq 10^9$ indicating two years.

There is a blank line between test cases. The input is terminated by a case where $n = 0$ and $m = 0$. This case should not be processed.

Technical note: Due to the size of the input, the use of cin/cout in `C++` might be too slow in this problem. Use scanf/printf instead. In Java, make sure that both input and output is buffered.

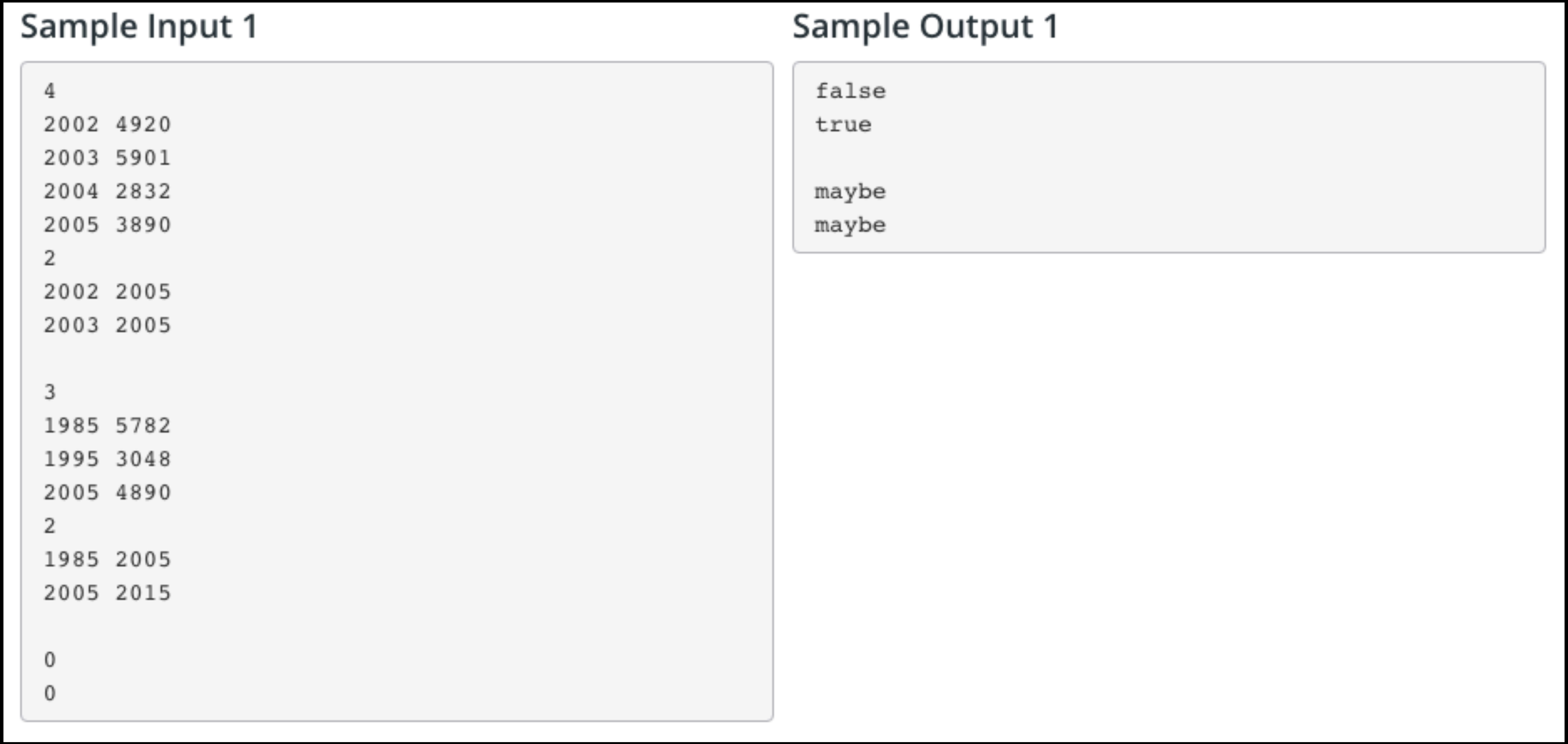


Output

There should be m lines of output for each test case, corresponding to the m queries. Queries should be answered with "`true`" if the statement is true, "`maybe`" if the statement might be true, and "`false`" if the statement is false.

Separate the output of two different test cases by a blank line.

Test Data



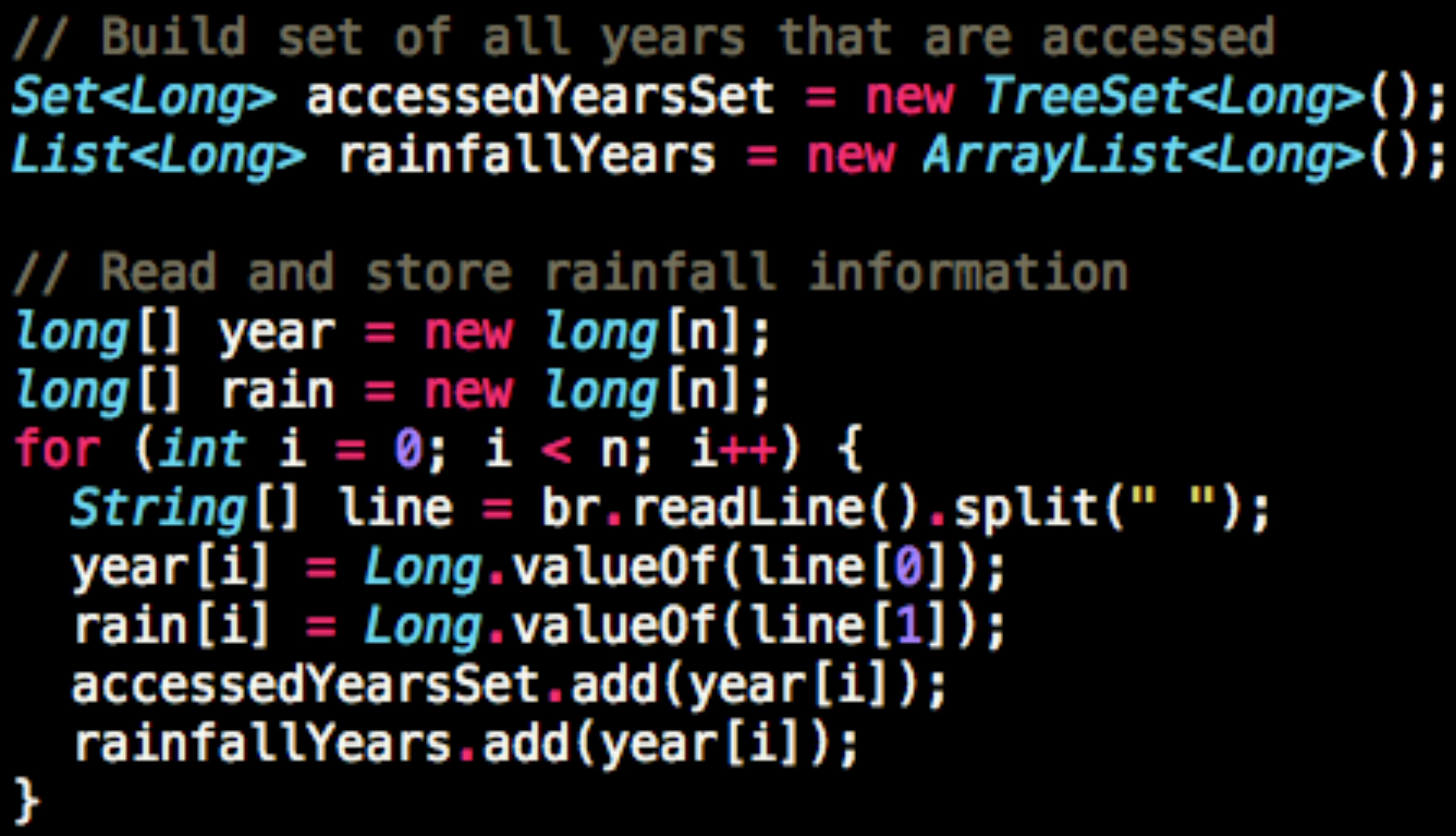
Sample Input 1

```
4
2002 4920
2003 5901
2004 2832
2005 3890
2
2002 2005
2003 2005

3
1985 5782
1995 3048
2005 4890
2
1985 2005
2005 2015

0
0
```

Sample Output 1

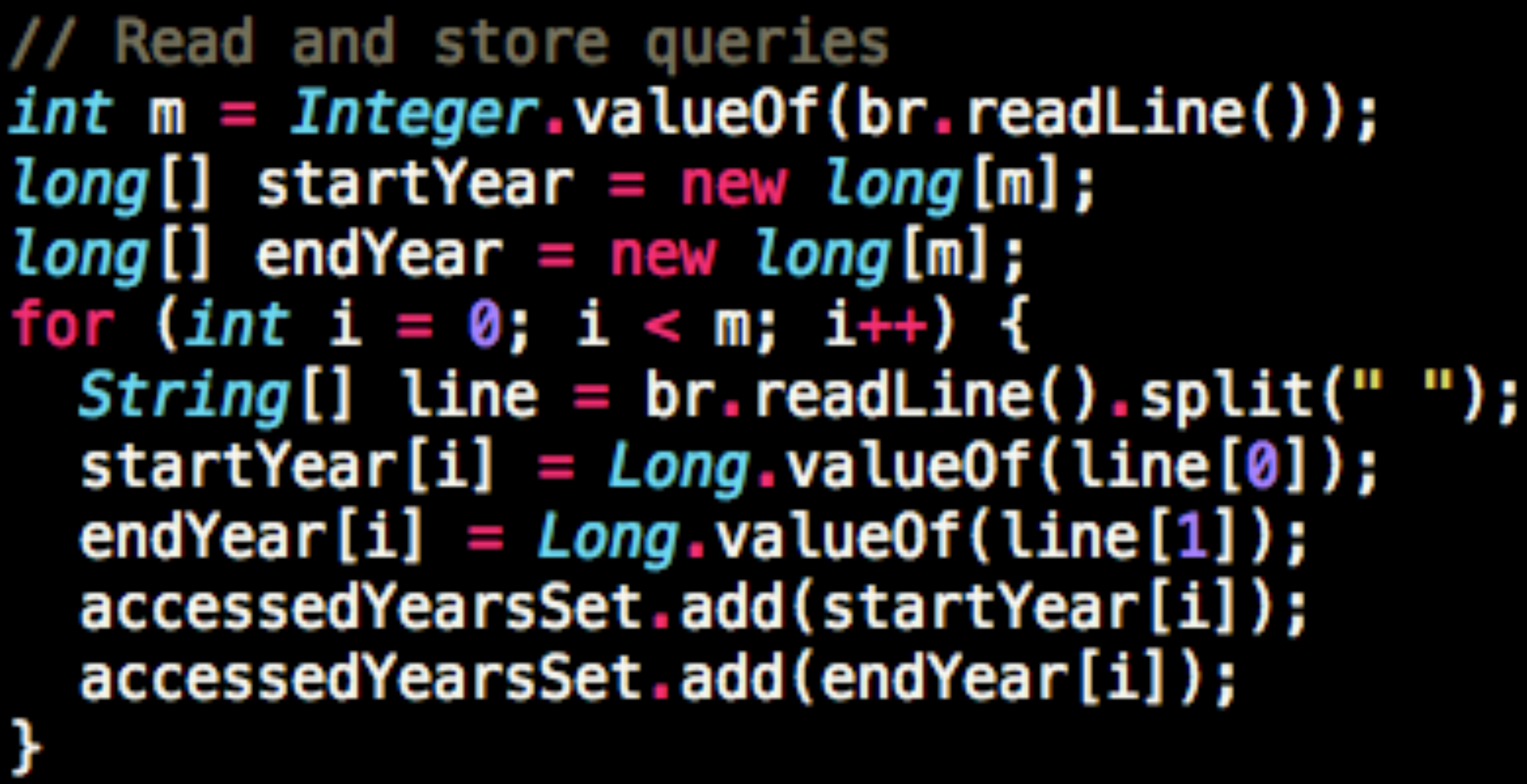
```
false
true

maybe
maybe
```

Solution



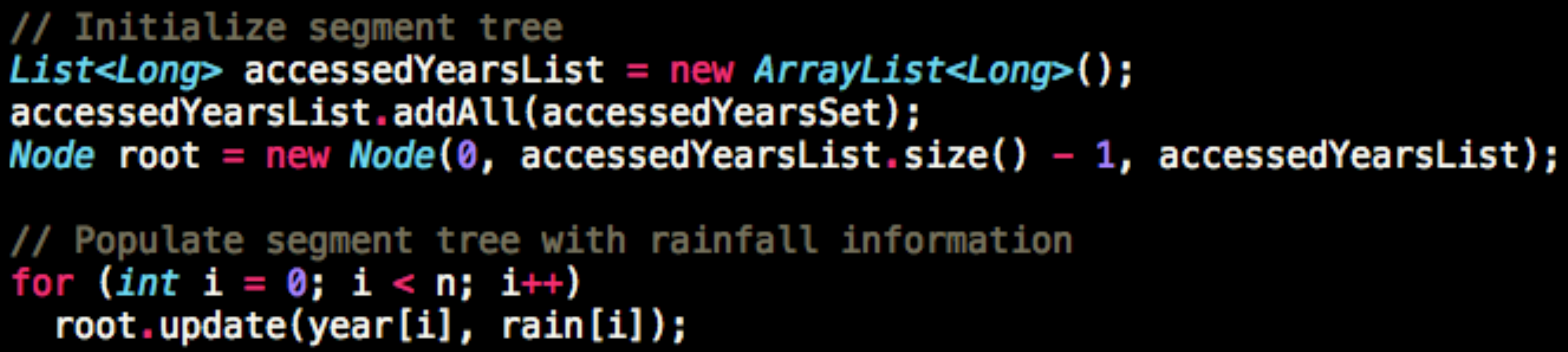
```
// Build set of all years that are accessed
Set<Long> accessedYearsSet = new TreeSet<Long>();
List<Long> rainfallYears = new ArrayList<Long>();

// Read and store rainfall information
long[] year = new long[n];
long[] rain = new long[n];
for (int i = 0; i < n; i++) {
  String[] line = br.readLine().split(" ");
  year[i] = Long.valueOf(line[0]);
  rain[i] = Long.valueOf(line[1]);
  accessedYearsSet.add(year[i]);
  rainfallYears.add(year[i]);
}
```

Solution



```java
// Read and store queries
int m = Integer.valueOf(br.readLine());
long[] startYear = new long[m];
long[] endYear = new long[m];
for (int i = 0; i < m; i++) {
  String[] line = br.readLine().split(" ");
  startYear[i] = Long.valueOf(line[0]);
  endYear[i] = Long.valueOf(line[1]);
  accessedYearsSet.add(startYear[i]);
  accessedYearsSet.add(endYear[i]);
}
```

Solution



```java
// Initialize segment tree
List<Long> accessedYearsList = new ArrayList<Long>();
accessedYearsList.addAll(accessedYearsSet);
Node root = new Node(0, accessedYearsList.size() - 1, accessedYearsList);

// Populate segment tree with rainfall information
for (int i = 0; i < n; i++)
  root.update(year[i], rain[i]);
```

Solution

```
// Process and respond to queries
for (int i = 0; i < m; i++) {

  long startYearRain = root.max(startYear[i], startYear[i]);
  long endYearRain = root.max(endYear[i], endYear[i]);

  // If unknown, pick values of rain which allow for an answer of 'maybe', if possible
  if (startYearRain == -1 && endYearRain == -1)
    startYearRain = endYearRain = 1_000_000_000L;
  else if (startYearRain == -1)
    startYearRain = endYearRain;
  else if (endYearRain == -1)
    endYearRain = startYearRain;

  . . .

}
```

Solution

```java
// Process and respond to queries
for (int i = 0; i < m; i++) {

  . . .

  // Determine if there are unknown years
  boolean hasAllYears = hasAllYears(rainfallYears, startYear[i], endYear[i]);
  boolean hasStartYear = hasAllYears(rainfallYears, startYear[i], startYear[i]);
  boolean hasEndYear = hasAllYears(rainfallYears, endYear[i], endYear[i]);

  // Pre-compute values
  boolean endYearDidNotRainMore = startYearRain >= endYearRain;
  long maxRainInBetween = root.max(startYear[i] + 1, endYear[i] - 1);

  // Evaluate conditions and print answer
  if (endYearDidNotRainMore && maxRainInBetween < endYearRain) {
    if (hasAllYears)
      System.out.println("true");
    else
      System.out.println("maybe");
  } else
    System.out.println("false");

}
```

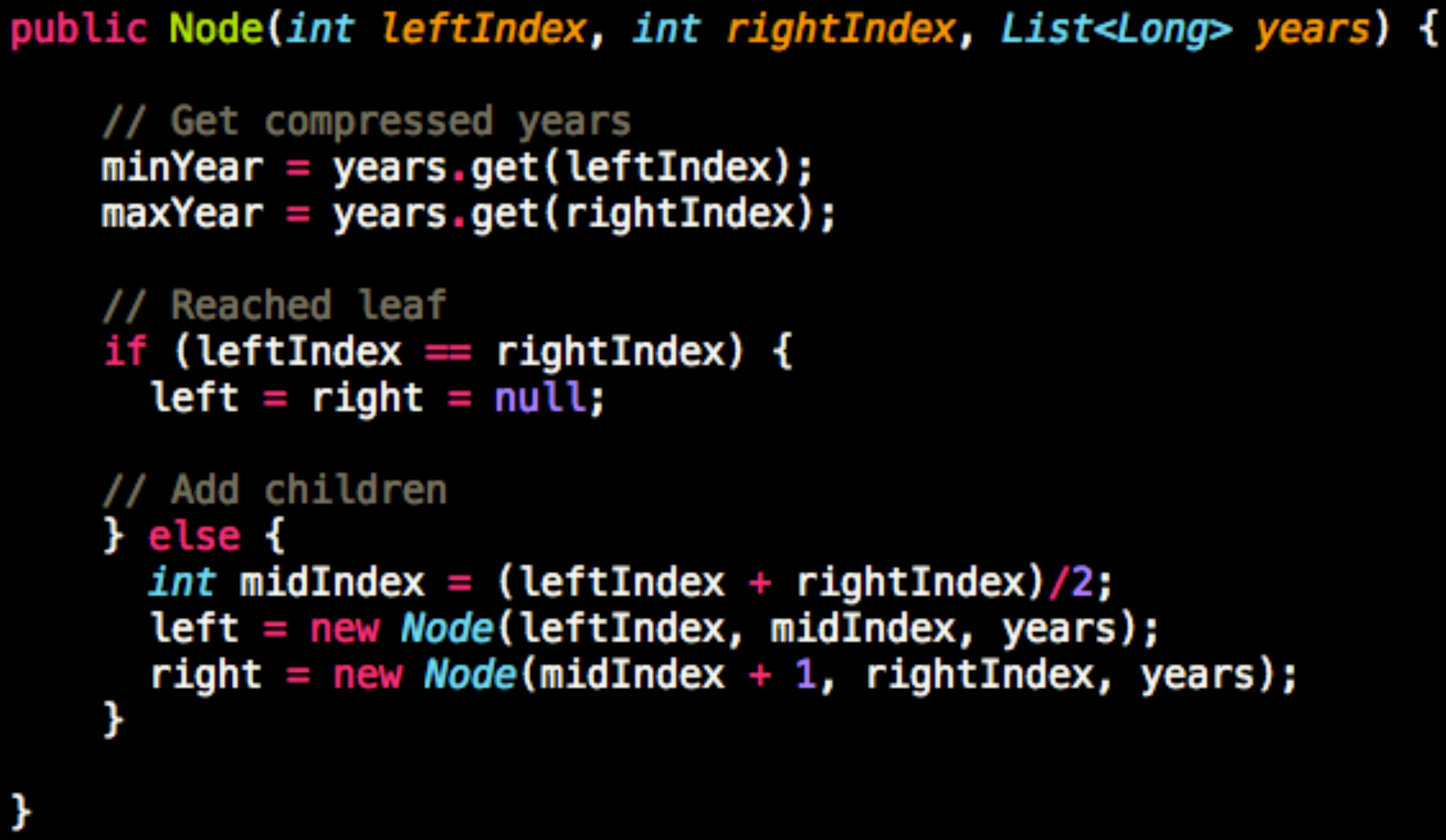
Solution

```java
// Check to see if we have all of the years in the specified range
static boolean hasAllYears(List<Long> years, long startYear, long endYear) {

  int startIndex = Collections.binarySearch(years, startYear);
  int endIndex = Collections.binarySearch(years, endYear);

  if (startIndex < 0 || endIndex < 0)
    return false;

  return endYear - startYear == endIndex - startIndex;

}
```

Solution

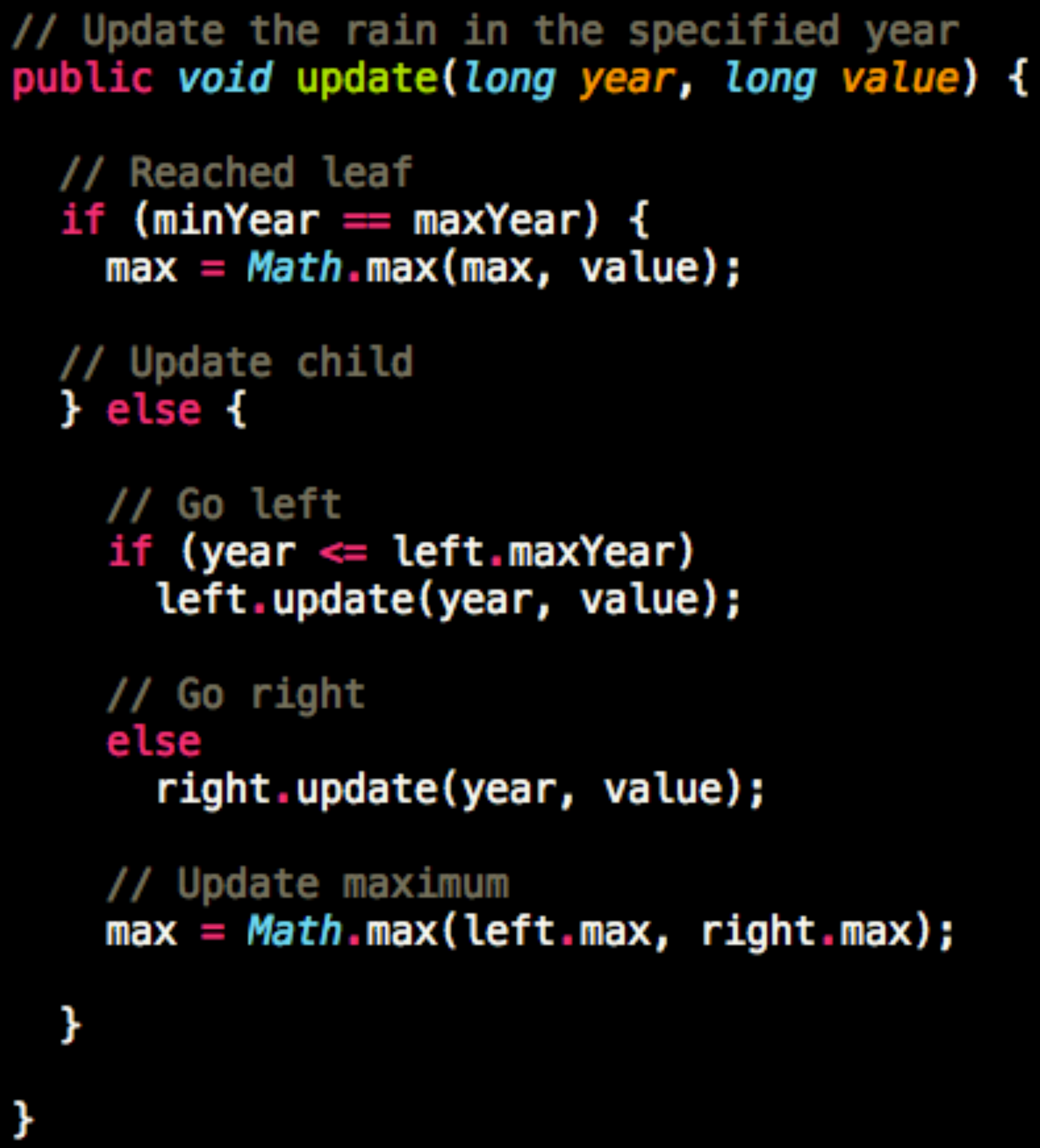
```
class Node {

  long minYear, maxYear;
  long max = -1;
  Node left, right;

  public Node(int leftIndex, int rightIndex, List<Long> years) {

    . . .

  }

  // Update the rain in the specified year
  public void update(long year, long value) {

    . . .

  }

  // Return maximum value amount of rain in specified range
  public long max(long startYear, long endYear) {

    . . .

  }

}
```

Solution



```java
public Node(int leftIndex, int rightIndex, List<Long> years) {

    // Get compressed years
    minYear = years.get(leftIndex);
    maxYear = years.get(rightIndex);

    // Reached leaf
    if (leftIndex == rightIndex) {
      left = right = null;

    // Add children
    } else {
      int midIndex = (leftIndex + rightIndex)/2;
      left = new Node(leftIndex, midIndex, years);
      right = new Node(midIndex + 1, rightIndex, years);
    }

}
```

Solution



```java
// Update the rain in the specified year
public void update(long year, long value) {

  // Reached leaf
  if (minYear == maxYear) {
    max = Math.max(max, value);

  // Update child
  } else {

    // Go left
    if (year <= left.maxYear)
      left.update(year, value);

    // Go right
    else
      right.update(year, value);

    // Update maximum
    max = Math.max(left.max, right.max);

  }

}
```

Solution

```
// Return maximum value amount of rain in specified range
public long max(long startYear, long endYear) {

  // Node's range fits inside query range
  if (startYear <= minYear && maxYear <= endYear) {

    return max;

  // Ranges do not overlap
  } else if (endYear < minYear || startYear > maxYear) {

    return -1;

  // Ranges partially overlap
  } else {

    return Math.max(left.max(startYear, endYear), right.max(startYear, endYear));

  }

}
```

Sources

- https://algo.is/
- https://github.com/TimonKnigge/Competitive-Programming/blob/master/Kattis/excellentengineers.cpp
- https://open.kattis.com/problems/excellentengineers
- https://open.kattis.com/problems/