Programming Assignment 4: Binary Search Trees

Revision: May 4, 2018

Introduction

In this programming assignment, you will practice implementing binary search trees including balanced ones and using them to solve algorithmic problems. In some cases you will just implement an algorithm from the lectures, while in others you will need to invent an algorithm to solve the given problem using hashing.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply binary search trees to solve the given algorithmic problems.
2. Implement in-order, pre-order and post-order traversal of a binary tree.
3. Implement a data structure to compute range sums.
4. Implement a data structure that can store strings and quickly cut parts and patch them back.

Passing Criteria: 3 out of 5

Passing this programming assignment requires passing at least 3 out of 5 code problems from this assignment. In turn, passing a code problem requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Problem: Binary tree traversals

Problem Introduction

In this problem you will implement in-order, pre-order and post-order traversals of a binary tree. These traversals were defined in the week 1 lecture on tree traversals, but it is very useful to practice implementing them to understand binary search trees better.

Problem Description

Task. You are given a rooted binary tree. Build and output its in-order, pre-order and post-order traversals.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $1 \le n \le 10^5$; $0 \le key_i \le 10^9$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex.

Output Format. Print three lines. The first line should contain the keys of the vertices in the in-order traversal of the tree. The second line should contain the keys of the vertices in the pre-order traversal of the tree. The third line should contain the keys of the vertices in the post-order traversal of the tree.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	12	6	6	12

Memory Limit. 512MB.

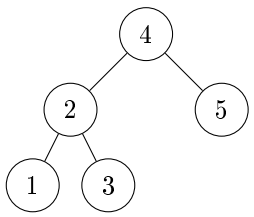
Sample 1.

Input:

```
5
4 1 2
2 3 4
5 -1 -1
1 -1 -1
3 -1 -1
```

Output:

```
1 2 3 4 5
4 2 1 3 5
1 3 2 5 4
```



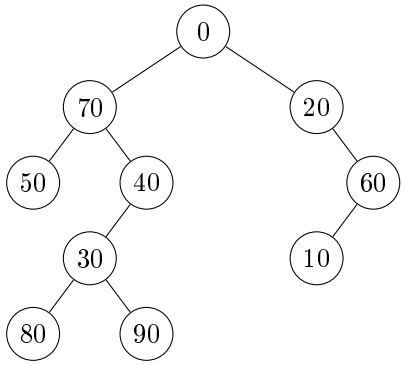
Sample 2.

Input:

```
10
0 7 2
10 -1 -1
20 -1 6
30 8 9
40 3 -1
50 -1 -1
60 1 -1
70 5 4
80 -1 -1
90 -1 -1
```

Output:

```
50 70 80 30 90 40 0 20 10 60
0 70 50 40 30 80 90 20 60 10
50 80 90 30 40 70 10 60 20 0
```



Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the input, define the methods to compute different traversals of the binary tree and write the output. You need to implement the traversal methods.

What to Do

Implement the traversal algorithms from the lectures. Note that the tree can be very deep in this problem, so you should be careful to avoid stack overflow problems if you're using recursion, and definitely test your solution on a tree with the maximum possible height.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Problem: Is it a binary search tree?

Problem Introduction

In this problem you are going to test whether a binary search tree data structure from some programming language library was implemented correctly. There is already a program that plays with this data structure by inserting, removing, searching integers in the data structure and outputs the state of the internal binary tree after each operation. Now you need to test whether the given binary tree is indeed a correct binary search tree. In other words, you want to ensure that you can search for integers in this binary tree using binary search through the tree, and you will always get correct result: if the integer is in the tree, you will find it, otherwise you will not.

Problem Description

Task. You are given a binary tree with integers as its keys. You need to test whether it is a correct binary search tree. The definition of the binary search tree is the following: for any node of the tree, if its key is x, then for any node in its left subtree its key must be strictly less than x, and for any node in its right subtree its key must be strictly greater than x. In other words, smaller elements are to the left, and bigger elements are to the right. You need to check whether the given binary tree structure satisfies this condition. You are guaranteed that the input contains a valid binary tree. That is, it is a tree, and each node has at most two children.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $0 \le n \le 10^5$; $-2^{31} < key_i < 2^{31} - 1$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex. All keys in the input will be different.

Output Format. If the given binary tree is a correct binary search tree (see the definition in the problem description), output one word "CORRECT" (without quotes). Otherwise, output one word "INCORRECT" (without quotes).

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	2	2	3	10	10	6

Memory Limit. 512MB.

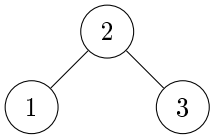
Sample 1.

Input:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

Output:

```
CORRECT
```



Left child of the root has key 1, right child of the root has key 3, root has key 2, so everything to the left is smaller, everything to the right is bigger.

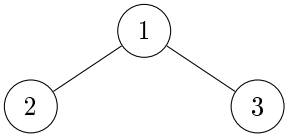
Sample 2.

Input:

```
3
1 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```



The left child of the root must have smaller key than the root.

Sample 3.

Input:

```
0
```

Output:

```
CORRECT
```

Empty tree is considered correct.

Sample 4.

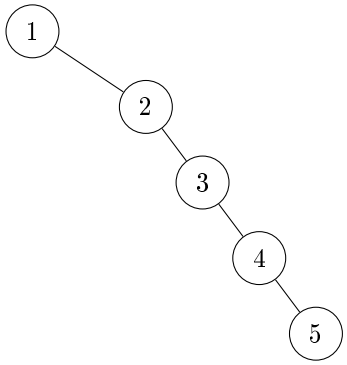
Input:

```
5
1 -1 1
2 -1 2
3 -1 3
4 -1 4
5 -1 -1
```

Output:

```
CORRECT
```

Explanation:



The tree doesn't have to be balanced. We only need to test whether it is a correct binary search tree, which the tree in this example is.

Sample 5.

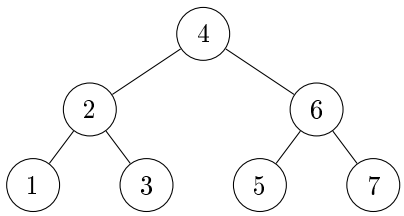
Input:

```
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
```

Output:

```
CORRECT
```

Explanation:



This is a full binary tree, and the property of the binary search tree is satisfied in every node.

Sample 6.

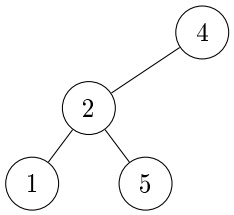
Input:

```
4
4 1 -1
2 2 3
1 -1 -1
5 -1 -1
```

Output:

```
INCORRECT
```

Explanation:



Node 5 is in the left subtree of the root, but it is bigger than the key 4 in the root.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch. Filename: `is_bst`

What to Do

Testing the binary search tree condition for each node and every other node in its subtree will be too slow. You should come up with a faster algorithm.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Problem: Is it a binary search tree? Hard version.

Problem Introduction

In this problem you are going to solve the same problem as the previous one, but for a more general case, when binary search tree may contain equal keys.

Problem Description

Task. You are given a binary tree with integers as its keys. You need to test whether it is a correct binary search tree. Note that there can be duplicate integers in the tree, and this is allowed. The definition of the binary search tree in such case is the following: for any node of the tree, if its key is x, then for any node in its left subtree its key must be strictly less than x, and for any node in its right subtree its key must be greater than **or equal** to x. In other words, smaller elements are to the left, bigger elements are to the right, and duplicates are always to the right. You need to check whether the given binary tree structure satisfies this condition. You are guaranteed that the input contains a valid binary tree. That is, it is a tree, and each node has at most two children.

Input Format. The first line contains the number of vertices n. The vertices of the tree are numbered from 0 to $n - 1$. Vertex 0 is the root.

The next n lines contain information about vertices 0, 1, ..., $n-1$ in order. Each of these lines contains three integers key_i, $left_i$ and $right_i$ — key_i is the key of the i-th vertex, $left_i$ is the index of the left child of the i-th vertex, and $right_i$ is the index of the right child of the i-th vertex. If i doesn't have left or right child (or both), the corresponding $left_i$ or $right_i$ (or both) will be equal to -1.

Constraints. $0 \le n \le 10^5$; $-2^{31} \le key_i \le 2^{31} - 1$; $-1 \le left_i, right_i \le n - 1$. It is guaranteed that the input represents a valid binary tree. In particular, if $left_i \ne -1$ and $right_i \ne -1$, then $left_i \ne right_i$. Also, a vertex cannot be a child of two different vertices. Also, each vertex is a descendant of the root vertex. Note that the minimum and the maximum possible values of the 32-bit integer type are allowed to be keys in the tree — beware of integer overflow!

Output Format. If the given binary tree is a correct binary search tree (see the definition in the problem description), output one word "CORRECT" (without quotes). Otherwise, output one word "INCORRECT" (without quotes).

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	2	2	3	10	10	6

Memory Limit. 512MB.

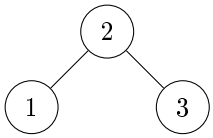
Sample 1.

Input:

```
3
2 1 2
1 -1 -1
3 -1 -1
```

Output:

```
CORRECT
```



Left child of the root has key 1, right child of the root has key 3, root has key 2, so everything to the left is smaller, everything to the right is bigger.

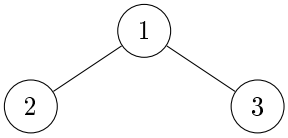
Sample 2.

Input:

```
3
1 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```



The left child of the root must have smaller key than the root.

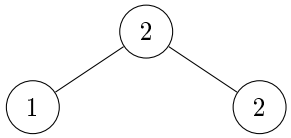
Sample 3.

Input:

```
3
2 1 2
1 -1 -1
2 -1 -1
```

Output:

```
CORRECT
```



Duplicate keys are allowed, and they should always be in the right subtree of the first duplicated element.

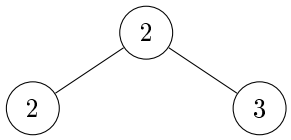
Sample 4.

Input:

```
3
2 1 2
2 -1 -1
3 -1 -1
```

Output:

```
INCORRECT
```



The key of the left child of the root must be strictly smaller than the key of the root.

Sample 5.

Input:

```
0
```

Output:

```
CORRECT
```

Empty tree is considered correct.

Sample 6.

Input:

```
1
2147483647 -1 -1
```

Output:

```
CORRECT
```

Explanation:

The maximum possible value of the 32-bit integer type is allowed as key in the tree.

Sample 7.

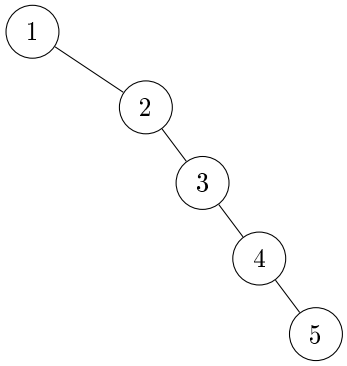
Input:

```
5
1 -1 1
2 -1 2
3 -1 3
4 -1 4
5 -1 -1
```

Output:

```
CORRECT
```

Explanation:



The tree doesn't have to be balanced. We only need to test whether it is a correct binary search tree, which the tree in this example is.

Sample 8.

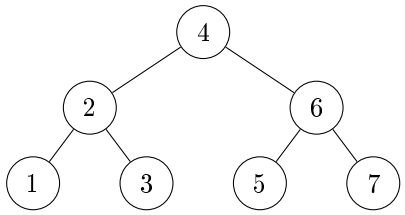
Input:

```
7
4 1 2
2 3 4
6 5 6
1 -1 -1
3 -1 -1
5 -1 -1
7 -1 -1
```

Output:

```
CORRECT
```

Explanation:



This is a full binary tree, and the property of the binary search tree is satisfied in every node.

Starter Files

The starter solutions for this problem read the input data from the standard input, pass it to a blank procedure, and then write the result to the standard output. You are supposed to implement your algorithm in this blank procedure if you are using `C++`, `Java`, or `Python3`. For other programming languages, you need to implement a solution from scratch. Filename: `is_bst_hard`

What to Do

Try to adapt the algorithm from the previous problem to the case when duplicate keys are allowed, and beware of integer overflow!

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

4 Advanced Problem: Set with range sums

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

In this problem, your goal is to implement a data structure to store a set of integers and quickly compute range sums.

Problem Description

Task. Implement a data structure that stores a set S of integers with the following allowed operations:

- **add**(i) — add integer i into the set S (if it was there already, the set doesn't change).
- **del**(i) — remove integer i from the set S (if there was no such element, nothing happens).
- **find**(i) — check whether i is in the set S or not.
- **sum**(l, r) — output the sum of all elements v in S such that $l \le v \le r$.

Input Format. Initially the set S is empty. The first line contains n — the number of operations. The next n lines contain operations. Each operation is one of the following:

- "+ i" — which means **add** some integer (not i, see below) to S,
- "- i" — which means **del** some integer (not i, see below)from S,
- "? i" — which means **find** some integer (not i, see below)in S,
- "s l r" — which means compute the **sum** of all elements of S within some range of values (not from l to r, see below).

However, to make sure that your solution can work in an online fashion, each request will actually depend on the result of the last **sum** request. Denote $M = 1\ 000\ 000\ 001$. At any moment, let x be the result of the last **sum** operation, or just 0 if there were no **sum** operations before. Then

- "+ i" means **add**$((i + x) \bmod M)$,
- "- i" means **del**$((i + x) \bmod M)$,
- "? i" means **find**$((i + x) \bmod M)$,
- "s l r" means **sum**$((l + x) \bmod M, (r + x) \bmod M)$.

Constraints. $1 \le n \le 100\ 000$; $0 \le i \le 10^9$.

Output Format. For each find request, just output "Found" or "Not found" (without quotes; note that the first letter is capital) depending on whether $(i+x) \bmod M$ is in S or not. For each **sum** query, output the sum of all the values v in S such that $((l+x) \bmod M) \le v \le ((r+x) \bmod M)$ (it is guaranteed that in all the tests $((l + x) \bmod M) \le ((r + x) \bmod M)$), where x is the result of the last **sum** operation or 0 if there was no previous **sum** operation.

Time Limits.

language	`C`	`C++`	`Java`	`Python`	`JavaScript`	`Scala`
time (sec)	1	1	4	120	120	4

Memory Limit. 512MB.

Sample 1.

Input:

```
15
? 1
+ 1
? 1
+ 2
s 1 2
+ 1000000000
? 1000000000
- 1000000000
? 1000000000
s 999999999 1000000000
- 2
? 2
- 0
+ 9
s 0 9
```

Output:

```
Not found
Found
3
Found
Not found
1
Not found
10
```

Explanation:
For the first 5 queries, $x = 0$. For the next 5 queries, $x = 3$. For the next 5 queries, $x = 1$. The actual list of operations is:
find(1)
add(1)
find(1)
add(2)
sum$(1, 2) \to 3$
add(2)
find(2) $\to$ Found
del(2)
find(2) $\to$ Not found
sum$(1, 2) \to 1$
del(3)
find(3) $\to$ Not found
del(1)
add(10)
sum$(1, 10) \to 10$

Adding the same element twice doesn't change the set. Attempts to remove an element which is not in the set are ignored.

Sample 2.

Input:

```
5
? 0
+ 0
? 0
- 0
? 0
```

Output:

```
Not found
Found
Not found
```

First, 0 is not in the set. Then it is added to the set. Then it is removed from the set.

Sample 3.

Input:

```
5
+ 491572259
? 491572259
? 899375874
s 310971296 877523306
+ 352411209
```

Output:

```
Found
Not found
491572259
```

Explanation:
First, 491572259 is added to the set, then it is found there. Number 899375874 is not in the set. The only number in the set is now 491572259, and it is in the range between 310971296 and 877523306, so the sum of all numbers in this range is equal to 491572259.

Starter Files

The starter solutions in C++, Java and Python3 read the input, write the output, fully implement splay tree and show how to use its methods to solve this problem, but don't solve the whole problem. You need to finish the implementation. If you use other languages, you need to implement a solution from scratch.

Note that we strongly encourage you to use stress testing, max tests, testing for min and max values of each parameter according to the restrictions section and other testing techniques and advanced advice from this reading. If you're still stuck, you can read the hints in the next What to Do section mentioning some of the common problems and how to test for them, resolve some of them.

What to Do

Use splay tree to efficiently store the set, add, delete and find elements. For each node in the tree, store additionally the sum of all the elements in the subtree of this node. Don't forget to update this sum each time the tree changes. Use split operation to cut ranges from the tree. To get the sum of all the needed elements after split, just look at the sum stored in the root of the splitted tree. Don't forget to merge the trees back after the **sum** operation.

Some hints based on the problems some learners encountered with their solutions:

- Use the sum attribute to keep updated the sum in the subtree, don't compute the sum from scratch each time, otherwise it will work too slow.
- Don't forget to do splay after each operation with a splay tree.
- Don't forget to splay the node which was accessed last during the find operation.
- Don't forget to update the root variable after each operation with the tree, because splay operation changes root, but it doesn't change where your root variable is pointing in some of the starters.
- Don't forget to merge back after splitting the tree.
- When you detach a node from its parent, don't forget to detach pointers from both ends.
- Don't forget to update all the pointers correctly when merging the trees together.
- Test sum operation when there are no elements within the range.
- Test sum operation when all the elements are within the range.
- Beware of integer overflow.
- Don't forget to check for null when erasing.
- Test: Try adding nodes in the tree in such an order that the tree becomes very unbalanced. Play with this visualization to find out how to do it. Create a very big unbalanced tree. Then try searching for an element that is not in the tree many times.
- Test: add some elements and then remove all the elements from the tree.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

5 Advanced Problem: Rope

We strongly recommend you start solving advanced problems only when you are done with the basic problems (for some advanced problems, algorithms are not covered in the video lectures and require additional ideas to be solved; for some other advanced problems, algorithms are covered in the lectures, but implementing them is a more challenging task than for other problems).

Problem Introduction

In this problem you will implement Rope — data structure that can store a string and efficiently cut a part (a substring) of this string and insert it in a different position. This data structure can be enhanced to become persistent — that is, to allow access to the previous versions of the string. These properties make it a suitable choice for storing the text in text editors.

This is a very advanced problem, harder than all the previous advanced problems in this course. Don't be upset if it doesn't crack. Congratulations to all the learners who are able to successfully pass this problem!

Problem Description

Task. You are given a string S and you have to process n queries. Each query is described by three integers i, j, k and means to cut substring $S[i..j]$ (i and j are 0-based) from the string and then insert it after the k-th symbol of the remaining string (if the symbols are numbered from 1). If $k = 0$, $S[i..j]$ is inserted in the beginning. See the examples for further clarification.

Input Format. The first line contains the initial string S.
The second line contains the number of queries q.
Next q lines contain triples of integers i, j, k.

Constraints. S contains only lowercase english letters. $1 \le |S| \le 300\,000$; $1 \le q \le 100\,000$; $0 \le i \le j \le n - 1$; $0 \le k \le n - (j - i + 1)$.

Output Format. Output the string after all q queries.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	3	3	6	120	120	12

Memory Limit. 512MB.

Sample 1.

Input:

```
hlelowrold
2
1 1 2
6 6 7
```

Output:

```
helloworld
```

Explanation:
hlelowrold → *hellowrold* → *helloworld*
When $i = j = 1$, $S[i..j] = l$, and it is inserted after the 2-nd symbol of the remaining string *helowrold*, which gives *hellowrold*. Then $i = j = 6$, so $S[i..j] = r$, and it is inserted after the 7-th symbol of the remaining string *hellowold*, which gives *helloworld*.

Sample 2.

Input:

```
abcdef
2
0 1 1
4 5 0
```

Output:

```
efcabd
```

$abcdef \to cabdef \to efcabd$

Starter Files

The starter solutions for C++ and Java in this problem read the input, implement a naive algorithm to cut and paste substrings and write the output. The starter solution for Python3 just reads the input and writes the output. You need to implement a data structure to make the operations with string very fast. If you use other languages, you need to implement the solution from scratch.

What to Do

Use splay tree to store the string. Use the split and merge methods of the splay tree to cut and paste substrings. Think what should be stored as the key in the splay tree. Try to find analogies with the ideas from this lecture.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

6 Solving a Programming Challenge in Five Easy Steps

6.1 Reading Problem Statement

Start by reading the problem statement that contains the description of a computational task, time and memory limits, and a few sample tests. Make sure you understand how an output matches an input in each sample case.

If time and memory limits are not specified explicitly in the problem statement, the following default values are used.

Time Limits.

language	C	C++	Java	Python	JavaScript	Scala
time (sec)	1	1	1.5	5	5	3

Memory limit: 512 Mb.

6.2 Designing an Algorithm

After designing an algorithm, prove that it is correct and try to estimate its expected running time on the most complex inputs specified in the constraints section. If you laptop performs roughly 10^8–10^9 operations per second, and the maximum size of a dataset in the problem description is $n = 10^5$, then an algorithm with quadratic running time is unlikely to fit into the time limit (since $n^2 = 10^{10}$), while a solution with running time $O(n \log n)$ will. However, an $O(n^2)$ solution will fit if $n = 1\,000$, and if $n = 100$, even an $O(n^3)$

solutions will fit. Although polynomial algorithms remain unknown for some hard problems in this book, a solution with $O(2^n n^2)$ running time will probably fit into the time limit as long as n is smaller than 20.

6.3 Implementing an Algorithm

Start implementing your algorithm in one of the following programming languages supported by our automated grading system: `C`, `C++`, `Haskell`, `Java`, `JavaScript`, `Python2`, `Python3`, or `Scala`. For all problems, we provide starter solutions for `C++`, `Java`, and `Python3`. For other programming languages, you need to implement a solution from scratch. The grading system detects the programming language of your submission automatically, based on the extension of the submission file.

We have reference solutions in `C++`, `Java`, and `Python3` (that we don't share with you) which solve the problem correctly under the given constraints, and spend at most 1/3 of the time limit and at most 1/2 of the memory limit. You can also use other languages, and we've estimated the time limit multipliers for them. However, we have no guarantee that a correct solution for a particular problem running under the given time and memory constraints exists in any of those other languages.

In the Appendix, we list compiler versions and flags used by the grading system. We recommend using the same compiler flags when you test your solution locally. This will increase the chances that your program behaves in the same way on your machine and on the testing machine (note that a buggy program may behave differently when compiled by different compilers, or even by the same compiler with different flags).

6.4 Testing and Debugging

Submitting your implementation to the grading system without testing it first is a bad idea! Start with small datasets and make sure that your program produces correct results on all sample datasets. Then proceed to checking how long it takes to process a large dataset. To estimate the running time, it makes sense to implement your algorithm as a function like `solve(dataset)` and then implement an additional procedure `generate()` that produces a large dataset. For example, if an input to a problem is a sequence of integers of length $1 \le n \le 10^5$, then generate a sequence of length 10^5, pass it to your `solve()` function, and ensure that the program outputs the result quickly.

Check the boundary values to ensure that your program processes correctly both short sequences (e.g., with 2 elements) and long sequences (e.g., with 10^5 elements). If a sequence of integers from 0 to, let's say, 10^6 is given as an input, check how your program behaves when it is given a sequence $0, 0, \ldots, 0$ or a sequence $10^6, 10^6, \ldots, 10^6$. Afterwards, check it also on randomly generated data. Check degenerate cases like an empty set, three points on a single line, a tree which consists of a single path of nodes, etc.

After it appears that your program works on all these tests, proceed to stress testing. Implement a slow, but simple and correct algorithm and check that two programs produce the same result (note however that this is not applicable to problems where the output is not unique). Generate random test cases as well as biased tests cases such as those with only small numbers or a small range of large numbers, strings containing a single letter "a" or only two different letters (as opposed to strings composed of all possible Latin letters), and so on. Think about other possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, complete graphs, bipartite graphs, etc. If you generate trees, try generating paths, binary trees, stars, etc. If you are generating integers, try generating both prime and composite numbers.

6.5 Submitting to the Grading System

When you are done with testing, submit your program to the grading system! Go to the submission page, create a new submission, and upload a file with your program (make sure to upload a source file rather than an executable). The grading system then compiles your program and runs it on a set of carefully constructed tests to check that it outputs a correct result for all tests and that it fits into the time and memory limits. The grading usually takes less than a minute, but in rare cases, when the servers are overloaded, it might take longer. Please be patient. You can safely leave the page when your solution is uploaded.

As a result, you get a feedback message from the grading system. You want to see the "`Good job!`" message indicating that your program passed all the tests. The messages "`Wrong answer`", "`Time limit exceeded`", "`Memory limit exceeded`" notify you that your program failed due to one of these reasons. If you program fails on one of the first two test cases, the grader will report this to you and will show you the test case and the output of your program. This is done to help you to get the input/output format right. In all other cases, the grader will *not* show you the test case where your program fails.

7 Appendix: Compiler Flags

C (`gcc 5.2.1`). File extensions: `.c`. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

C++ (`g++ 5.2.1`). File extensions: `.cc`, `.cpp`. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your `C/C++` compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and MacOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., `cygwin`.

Java (`Open JDK 8`). File extensions: `.java`. Flags:

```
javac -encoding UTF-8
java -Xmx1024m
```

JavaScript (`Node v6.3.0`). File extensions: `.js`. Flags:

```
nodejs
```

Python 2 (`CPython 2.7`). File extensions: `.py2` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python2"). No flags:

```
python2
```

Python 3 (`CPython 3.4`). File extensions: `.py3` or `.py` (a file ending in `.py` needs to have a first line which is a comment containing "python3"). No flags:

```
python3
```

Scala (`Scala 2.11.6`). File extensions: `.scala`.

```
scalac
```