The 4th Problem Set

March 5, 2018

Pre-requisites

You should do the followings

1. On a separate and clean paper, you need to describe your own strategy to solve the problems below, and to justify why your strategy is effective while handling each problem
2. On a new clean paper, transform your strategy into an algorithm, using your own form to express algorithms. Further, you need to analyze the total running steps of your algorithm and the required memory amount to finish your algorithm. Then express the total costs using Big-O notation.
3. Use the `Code ocean` (`https://codeocean.com`) platform; if necessary, you may invite me using my email address `lightsun.kim@gmail.com`.
4. Time limits for each problem
   - Problem #1: Within 4 hours
   - Problem #2: Within 3 hours
   - Problem #3: Within 3 hours

   Then you need to prepare two answer codes; one is a C code that you have made within each time limit, and the other is a C code augmented and fixed from the original code later.

1 Problem #1

Let n be a positive integer and a set $X = \{1, 2, \ldots, 5n\}$. Assume that you are given an integer α and an integer array $A = \langle a_1, a_2, \ldots, a_n \rangle$ whose element $a_i \in X$ for each $1 \leq i \leq n$. Your goal in this problem is to implement an efficient algorithm for determining whether the set A has two elements $a_i \in A$ and $a_j \in A$ such that $\alpha = a_i + a_j$, for $1 \leq i \neq j \leq n$. A simple solution for this problem is the algorithm `Exhaustive`(A, n, α) below. This naive algorithm is trying to exhaustively look into all possible pairs from A.

Algorithm `Exhaustive`(A, n, α)

INPUT: An n-element integer array $A = \langle a_1, \ldots, a_n \rangle$ whose elements are in X where $X = \{1, 2, \ldots, 5n\}$ for some positive integer n.
OUTPUT: A pair of indices (i, j) such that $\alpha = a_i + a_j$ if exists; otherwise $\perp$

```
1.  for i ← 1 to n
2.      for j ← 1 ∧ j ≠ i to n
3.          if a_i + a_j = α then
4.              return (i, j)
5.  return ⊥
```

This algorithm will work correctly; however it requires $O(n^2)$ run-time complexity and thus when n is a huge-sized integer (e.g., $n = 10^{20}$) it shows a very slow execution time. Thus your technical mission is to improve this algorithm in a stepwise manner. To do this, firstly you need to find a somewhat improved algorithm `Somewhat`(A, n, α) whose run-time complexity is $O(n \log n)$. Next, you should devise a significantly improved algorithm `Significant`(A, n, α) whose run-time complexity is merely $O(n)$. Moreover, in order to argue that your idea is effective so as to improve the run-time complexity, you should measure running times of all three programs in milliseconds over 10^6 randomly generated samples. All sampled elements are positive integers at most of 22 bits.

Language requirements. During tackling this problem, you should follow the programming rules:

- You should use an ANSI C programming language whose source code can run on `Code ocean` platform.
- Function naming: Begin with the lower character, and every parameters are strong-typed variables (i.e., do not use `void` typed variables). All functions should have a single return value; thus even if a function will return no values; you should provide `return` keyword.
- Variable naming: Begin with a type-discriminating prefix. For example, if a variable name is for an age and is with an integer type, you need to declare the variable as `int iAge;` Especially for string-type variables you are strongly recommended to use the prefix `sz`. For example, if a variable name is for a name, then `szName` is a preferable choice.

Input format. The input is given a text-format file, named `input.txt` and all strings are separated by commas. The file contains 10^6 integers, $A = (a_1, a_2, \ldots, a_{10^6})$, which have been randomly generated prior to execution, and each a_i is in the range of 1 and 2^{22}. You see that $n = 10^6$ and because $10^6 \approx 2^{20}$ you have $5n \approx 2^{22}$. Note that the input file does not include the size of the array A; thus the input file is given in form as follows:

$a_1, a_2, \ldots, a_{1000}$
$a_{1001}, a_{1002}, \ldots, a_{2000}$
$\ldots$
$a_{999001}, a_{999002}, \ldots, a_{10^6}$

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes (1) the pair of indices (i, j) if exists; otherwise $(0, 0)$ indicating there are no matching indices (2) if exists, the solution $\alpha = a_i + a_j$, else skip this line and (3) the execution times of your each algorithm.

```
**************************************
(1) By my exhaustive algorithm
**************************************
(i,j)=(i,j)
α=a_i+a_j
The total execution time: _______ sec

**************************************
(2) By my somewhat improved algorithm
**************************************
(i,j)=(i,j)
α=a_i+a_j
The total execution time: _______ sec

**************************************
By my significantly improved algorithm
**************************************
(i,j)=(i,j)
α=a_i+a_j
The total execution time: _______ sec
```

2 Problem #2

Regarding a binary tree T, we follow the usual terminologies such as binary trees, root, child/parent nodes, height and so on [HSAf08, §5.1,§5.2]. Especially we define ancestor and descendent nodes in order to avoid any type of ambiguity as follows.

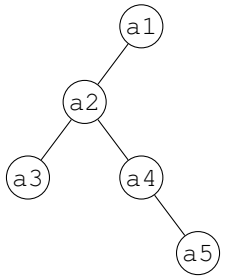
Definition 2.1 (Ancestors, Descendants) *Let α, β be nodes of a binary tree T. α is an ancestor of β if α is a parent of β, or α is the ancestor of the parent of β. Further, α is descendant of β if α is a child of β, or α is a descendant of a child of β.*

Let T be a binary node with N nodes. Keeping in mind these terms, we define a **roman node** of order d by a node $\alpha \in T$ such that the number of descendants in α's left subtree differ from the number of descendants of α's right subtree by at most d. More formally, for a node $\alpha \in T$ let α_L, α_R be the left and right nodes of α, respectively. Then let $T_L(\alpha), T_R(\alpha)$ be binary trees whose roots are α_L and α_R, respectively. Writing $T_L(\alpha) = \{u_1, \ldots, u_n\}$ and $T_R(\alpha) = \{v_1, \ldots, v_m\}$ for some integers $n, m \leq N$, if α is a roman node of order d, then $|n - m| \leq d$. This problem requires that you should implement a C program so that it finds each node α of T such that α is not a roman node of order d, but all of α's descendants are roman nodes of order d. Furthermore your algorithm for this program should satisfy $O(N)$ run-time complexity. You may get some idea from Problem #3 in the next pape.

Input format. The input is given a text-format file, named `input.txt` and all strings are separated by blanks.

`rn=`α`,d=`d
ℓ_1
ℓ_2 ℓ_3
$\cdots$
$\ell_{2^{t-1}}, \ell_{2^{t-1}+1}, \ldots, \ell_{2^t-1}$

Here the first line means that α is the label name of roman node and d is the order of α. A binary tree T of height t is described from the second line to the end line. As you expected, the second line shows the label name of the root node of T and its below line consists of its left child node and right child node. Every label name consists of a single lowercase alphabet followed by a non-zero integer, for example, `u9`, `v12`, or `a09`. If its left/right children do not exits, the symbol '$*$' is placed at its corresponding position. Thus each ℓ_i means either a label name or the empty child node ($*$). For example, consider the following binary tree.



The following shows a part of `input.txt` for the tree as above

```
...
a1
a2 *
a3 a4 * *
* * * a5 * * * *
```

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the resulting roman nodes of order d.

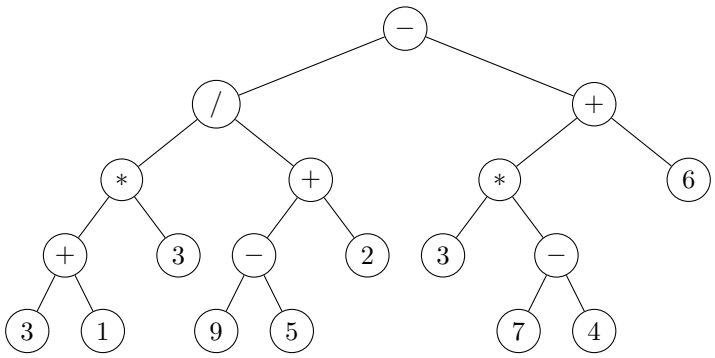
$\alpha_1, \cdots, \alpha_w$

3 Problem #3

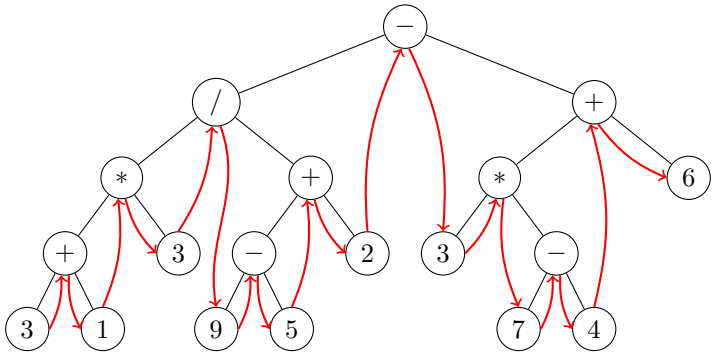
Roughly speaking, there are dozens of traversal methods of a binary tree. Some of well-known tree traversals include pre-order, post-order, in-order. An important application of tree traversal is to evaluate an arithmetic express E without considering parentheses included in E. For example, consider an arithmetic expression

$$E = ((((3 + 1) * 3)/((9 - 5) + 2)) - ((3 * (7 - 4)) + 6)).$$

This expression can be represented as a proper binary tree as follows.



Then evaluating the arithmetic express E by the inorder traversal is performed as follows:



To the contrary, this problem considers an operation in this opposite direction. This is to say, given a binary tree T representing an arithmetic expression you are required to print the expression E by reading the tree T. One popular way to do this task is to use a tree traversal known as the **Euler tour traversal**.

Roughly, the Euler tour traversal of a binary tree T can be considered as a "walk" around the tree T, where start by going from the root r towards its left child $T_L(r)$, thinking of the edges of T as being "walks" that you always keep to your left. Refer to Figure 1, in the next page. Yo can see that you stop by each node $v \in T$ three times by the Euler tour which consists of three basic operations:

1. Left visit. You will see v before the Euler tour of v's left subtree $T_L(v)$. Denote it by $\texttt{left}(v)$.
2. Below visit. You will also see v immediately after the Euler tour of v's two subtrees). Denote it by $\texttt{below}(v)$.
3. Right visit. You will see again v after the Euler tour of v's right subtree $T_R(v)$. Denote it by $\texttt{right}(v)$.

You may observe that if v is an external node, then those three visiting operations all happen at the same time. Thus the Euler tour traversal is a generalization of pre-order, in-order, and post-order traversals. In other words, the pre-order traversal of T is equivalent to an Euler tour traversal such that each node has an associated "visit" operation occur only when it is encountered on the left. Similarly, the in-order and post-order traversals of T are equivalent to

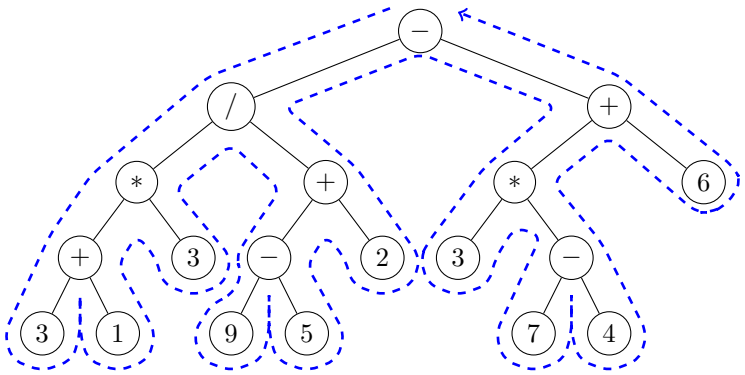


Figure 1: Euler tour traversal of a binary tree

an Euler tour traversal such that each node has an associated "visit" operation occur only when it is encountered from below or on the right, respectively.

The goal in this problem is simply to implement a C program to print a fully parenthesized arithmetic expression E from its representing binary tree T. Your program should implement and invoke the three Euler tour operations `left`(), `below`(), and `right`(), properly. Assume that you have only six arithmetic operations $+, -, *, /, \hat{}$, and $\%$.

Input format. The input is given a text-format file, named `input.txt`. This file describes a binary tree T of N nodes in the level-order manner where each node v is written by $v = [u_{i1}, u_{i2}, u_{i3}]$ for the label name of v and its left/right children u_{i2}, u_{i3}, respectively. Indicating a node without children is by #.

$[u_{11}, u_{12}, u_{13}]$
$[u_{21}, u_{22}, u_{23}]$
...
$[u_{N1}, u_{N2}, u_{N3}]$

For example, the binary tree in Figure 1 is given by

```
[-,/,+]
[/,*,+]
[+,*,6]
[*+,3]
[+,-,2]
[*,3,-]
[6,#,#]
[+,3,1]
[3,#,#]
[-,9,3]
[2,#,#]
[3,#,#]
[-,7,4]
[3,#,#]
[1,#,#]
[9,#,#]
[5,#,#]
[7,#,#]
[4,#,#]
```

Output format. The output should be given as a text-format file, named `outputc.txt`, `outputq.txt`, or `outputl.txt`. The output file writes a fully parenthesized arithmetic expression.

```
** The arithmetic expression **
E =
```

References

[HSAf08] E. Horowitz, S. Sahni, and S. Anderson-Free, *Fundamentals of Data Structures in C*, Second edition, Silicon Press, 2008.