The 6th Problem Set

April 4, 2018

Pre-requisites

You should do the followings

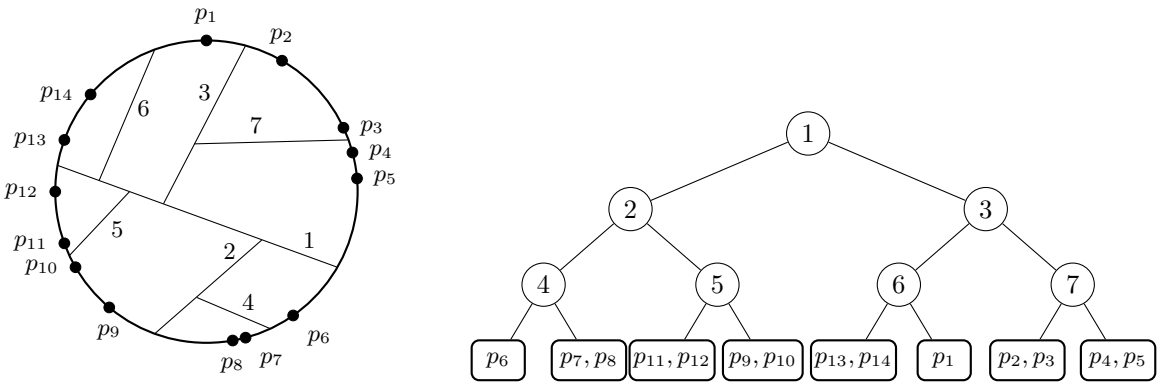
1. On a separate and clean paper, you need to describe your own strategy to solve the problems below, and to justify why your strategy is effective while handling each problem
2. On a new clean paper, transform your strategy into an algorithm, using your own form to express algorithms. Further, you need 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 3 hours
   - Problem #2: Within 1.5 hours
   - Problem #3: Within 3 hours
   - Problem #4: Within 4 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.

Problem #1

Suppose that you are one of programmers in a first-person shooter game[1] development company, and consider a computer game using a disk-type board in a virtual room. In reality, you may see similar types of shooting games in an amusement park. However, note that we run the game in some imaginary space that we call a game room. This game is played in a *circular* game room and many shooting disks are installed in this room but, initially, all target disks are off. In this game, each player stands behind the imaginary boarder of the circular room and shoots at targets in the room. As you can imagine, assume that there will be lots of players and targets, however, for any given player, it only makes sense to display the targets that are close by that player. This assumption can be justified by a combination of a sensor of be able to measure proximity and some simple switch. Thus if the sensor detects a player within 5m, it signals the switch to turn on the light bulb so that the player can recognize such a target.

Therefore, whenever a new target, ϕ, appears, the game should only make it visible to the players who are in the same zone as ϕ. In order to fast process such requests, you come up with constructing a binary room division (BRD) tree, Φ, for use in the game engine. See Figure 1 below. The room in Figure 1 has 8 zones which are disjointly partitioned by 7 lines. For example, line 1, 3, and 7 offer a zone $\phi_{1:3:7}$ in which there are two player p_4, p_5. Then the two players will be given a target $\phi_{1:3:7}$ and they will be able to shoot the target $\phi_{1:3:7}$.



(a) A circular space (or room)

(b) The corresponding BRD tree Φ

Figure 1: A binary room division tree

Now you are given a list of n players $P = (p_1, \ldots, p_n)$ and their coordinates on the disk $C = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$ where (x_i, y_i) is the coordinate of player p_i for each $1 \leq i \leq n$. Thus this problem requires you (as a programmer) to design an efficient algorithm to build such a BRD tree Φ satisfying that

1. height$(\Phi) = O(\log n)$; you should properly prove that your BRD tree has this height,
2. the run-time complexity comes to $O(n \log n)$.

The root r of the BRD tree Φ is associated with a line L_r that divides the set of players into two groups of roughly equal size. For example, in Figure 1, line L_1 is associated with the root node, entitled "1". Similarly, it should be recursively applied to a partitioning process of each group into two groups of roughly equal size. Of course, during the partitioning process, you should consider the coordinates of players.

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.

[1]For example, Doom, Resident Evil, and Far Cry.

- 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 some proper delimeter. The file contains the number of coordinates N and the corresponding N coordinates. Assume that your game engine can only detect players on the border of the circle. Thus although $N \geq n$ coordinates are given, the only n players should appear at the circular boarder of the disk. By the way, you should have $N \geq 10^3$. Moreover, there are some options that you can choose as a coordinate system; for example, Cartesian coordinates and polar coordinates. Therefore according to the coordinate system that you choose, the coordinate representation (x, y) of a player p should be written differently. A simple way to obtain N coordinates is firstly to generate n coordinates on the circle and then to choose $N - n$ random points. Of course, you may have your own way; this is always a better direction towards this problem.

N
n
(x_1, y_1)
(x_2, y_2)
$\cdots$
(x_N, y_N)

Output format. The output should be given as a text-format file, named `output.txt`. The BRP tree Φ should be given in form of the followings:

1. Each internal node v_i of Φ consists of $(v_i, v_{i,\text{left}}, v_{i,\text{right}})$ where v_i is the name of an internal node in Φ and its two children nodes are represented by $v_{i,\text{left}}, v_{i,\text{right}}$.
2. When writing each node of Φ at the output file, use the **level order traversal**[2]. In general, the level order traversal requires use of queue; however, its implementation completely depends on your own choice, as always mentioned.
3. The terminal nodes are carefully handled, and by using the symbols $\{, \}$ in addition, write each leaf node with its parent v_ℓ and its children $\{v_{\text{left}}\}$ and $\{v_{\text{right}}\}$; thus $(v_\ell, \{v_{\text{left}}\}, \{v_{\text{right}}\})$. For example, the BRD tree in Figure 1 should be written in file $(1, 2, 3), (2, 4, 5), (3, 6, 7), (4, \{p_6\}, \{p_7, p_8\}), \ldots, (7, \{p_2, p_3\}, \{p_4, p_5\})$.
4. $\{v_{\text{left}}\} \cap \{v_{\text{right}}\} = \varnothing$ and $\{v_{\text{left}}\} \cup \{v_{\text{right}}\} = \{p_1, \ldots, p_n\}$ for all terminal nodes $v \in \Phi$.
5. When an internal node has an empty terminal node (also known as the null node), write it by $*$.

$(v_1, v_{1,\texttt{left}}, v_{1,\texttt{right}})$
$(v_2, v_{2,\texttt{left}}, v_{2,\texttt{right}})$
$\cdots$
$(v, \{v_{\texttt{left}}\}, \{v_{\texttt{right}}\})$

[2]The level-order traversal is read down from top to bottom and from left to right. For the details, refer to [HSA08, §5.3.6] and [Sed98, §5.6].

Problem #2

In this problem, we revisit Problem #3 in the 3rd Problem set–The *Josephus problem*. As a simple variant of the Josephus problem, let us consider the numbers 1 to n arranged in a circle and repeatedly removing every ℓ-th number around the circle, outputting the resulting sequence of the numbers. For example, with $n = 10$ and $\ell = 4$, the sequence must be

$$4, 8, 2, 7, 3, 10, 9, 1, 6, 5.$$

Now what you do is that given values for n and m, design an algorithm for outputting the sequence resulting from this variant of the Josephus problem in $O(n \log n)$ time, or better.

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 blanks. The file contains the total number of soldiers n and a prefixed value ℓ.

n ℓ

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the sequence of numbers of removed soldiers. Let $\pi(i)$ be a permutation over the set $\{1, 2, \ldots, n\}$. As you can see, the sequence is uniquely determined by n and ℓ; thus the resulting sequence can be represented by a permutation π.

$p_{\pi(1)}, p_{\pi(2)}, \cdots, p_{\pi(n)}$

Problem #3

Like Problem #1 suppose that you are now working in a different shooting game development team, in which tons of deadly noxious ladybugs[3] are climbing up a wall to attack a city, while a game player is moving left and right in front of the wall. Fortunately, the player can kill them using various weapons like sword, shield, and bombs. See Figure 2 below.

The position of each bug is expressed with a pair of (x, y), where x is the horizontal position of the bug and y is its height on the wall. Because the player is allowed to move in horizontal line, the player's position is specified by with just a horizontal value x_p. A bomb that is the strongest weapon of the player is to kill the bug that is highest on the wall from all the bugs within a given horizontal distance, r, of x_p.

Let assume that the bugs are stored in a binary search tree (BST), T, of height h, ordered in terms of their horizontal positions. Design an algorithm for augmenting the BST T so that it answers maximum-bug queries in $O(h)$ time where such a query is given by a closed range $R = [x_p - r, x_p + r]$ and you need to return the coordinates of the bug's with maximum y-value whose horizontal position, $x \in R$. Furthermore, you should implement the operations for adding and deleting bugs to and from T.

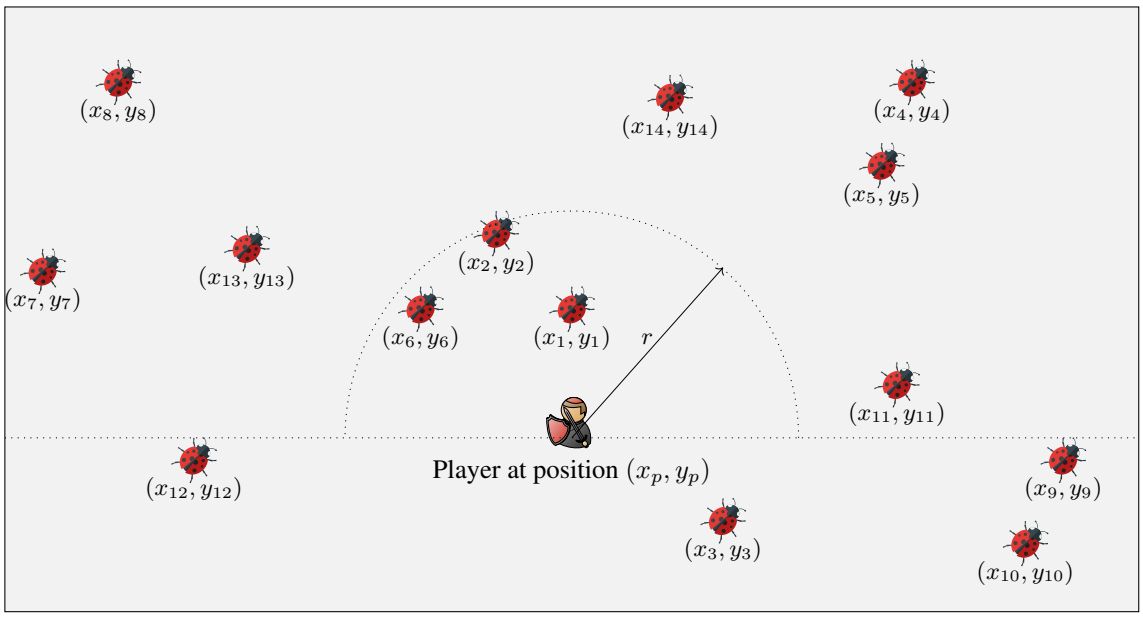


Figure 2: A snapshot of the game

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.

[3] In fact, the bugs are never harmful to human-beings as far as I know.

Input format. The input is given a text-format file, named `input.txt` and all strings are separated by blanks. The file first describes the number of bugs $n > 10^4$, the height and width of the wall, (ℓ, w), and an **unsorted** list of points $(v_1, \ldots, v_n)$ where each point $v_i = (x_i, y_i)$. Then the input file specifies the vertical position of the player, y_p, and the horizontal radius r.

n
$\ell \;\; w$
$x_1 \;\; x_2 \;\; \cdots \;\; x_{10}$
$x_{11} \;\; x_{12} \;\; \cdots \;\; x_{20}$
$\cdots$
$x_{9901} \;\; x_{9902} \;\; \cdots \;\; x_{10000}$
$y_1 \;\; y_2 \;\; \cdots \;\; y_{10}$
$y_{11} \;\; y_{12} \;\; \cdots \;\; y_{20}$
$\cdots$
$y_{9901} \;\; y_{9902} \;\; \cdots \;\; y_{10000}$
$y_p \;\; r$

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the list of resulting points from maximum-bug queries. Thus with the list of size t $(0 \leq t \leq n)$, the file shows a point in each line; thus it has t lines of points.

(x_{i_1}, y_{i_1})
(x_{i_2}, y_{i_2})
$\cdots$
(x_{i_t}, y_{i_t})

Problem #4

Consider an integer $x = \texttt{12345678912345678912345678}$ in decimal form. Like this number, it is hardly to easily be handled by a general desktop computer, since even in a 64-bit machine, the number x cannot be directly stored in an integral type such as `int` or `unsigned long`. Therefore we need to use a special way to deal with those huge integers. Hereafter we call an integer of greater than 64 bits a *big* integer. Of course, your algorithm should properly care any integer whose bit length is less than or equal to 64 bits. This problem focuses on such an efficient way to perform arithmetics over big integers.

One solution for this problem is to store a big integer x of n bits at one dimensional integer array. For example, assuming that $\texttt{sizeof(short int)} = 2$, an integer x of 320 bits can be stored an integer array `x` by declaring `short int x`$[20] = 0$;. To do this, we first encode an integer x as the binary representation

$$x = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2 + a_0$$

and letting $1 < n$ be a multiple of 16 and $\ell = \frac{n}{16}$, we have

$$\texttt{x}[i] = 2^{(\ell-i)\cdot 16} \cdot \left(b_{15} \cdot 2^{15} + b_{14} \cdot 2^{14} + \cdots + b_1 \cdot 2 + b_0\right),$$

so in practice, we may think of $\texttt{x}[i]$ as $(b_{15}b_{14}\cdots b_0)_2$. You should note that your choice of integer array is not limited to `short int` type; rather you may use `unsigned int`-, `unsigned long`-, or `unsigned long long`-typed array at your discretion.

However, this way raises a small problem regardless of underlying integer type. Consider two n-bit big integers x, y that have been store at `x`, `y`, respectively, and $\texttt{z} = \texttt{x} + \texttt{y}$. In order to add two big integers, for some proper t you will do

for $i \leftarrow 0$ **to** t **do**
  $\texttt{z}[i] \leftarrow \texttt{x}[i] + \texttt{y}[i]$.

While doing integer addition between $\texttt{x}[i]$ and $\texttt{y}[i]$, as you know well, the carry problem occurs. That is to say, you should always take care of handling carry at each index i correctly. This is the case of multiplying two big integers.

This problem requires to implement four basic arithmetics: addition, subtraction, multiplication, and division over big integers. Here division between two big integers outputs the unique quotient and remainder satisfying the division algorithm theorem; that is, given two big integers x, y, your algorithm $(q, r) \leftarrow \texttt{divide}(x, y)$ where q, r are also big integers and $x = yq + r$ with $0 \le r < y$.

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`. The file describes the sequence of t big integer arithmetics as follows

$\texttt{add}(x_1, y_1)$
$\texttt{multiply}(x_2, y_2)$
$\texttt{divide}(x_3, y_3)$

add(x_4, y_4)
...
substract(x_t, y_t)

where all x_i's and y_i's are big integers.

In order to read big integer from character strings, you need to implement an algorithm to encode a character string into a big integer. For a character-typed string s, this algorithm may have the following prototype in C:

```
void encodeStr2Big(char* s, const bigint x);
```

Today a general desktop computer works on a 64-bit platform. Therefore, even though all explanations are given assuming 16-bit processors, you should again interpret these descriptions that make sense for your target machine. For convenience, use the following predefined literals and make them included in your code.

```
#define   MAXBITS          1024              /* but not fixed */
#define   SINTSIZE         16                /* bit length of short integer in C */
#define   UINTSIZE         32                /* bit length of integer in C */
#define   ULINTSIZE        64                /* bit length of unsigned long integer in C */
#define   WORDSIZE         sizeof(int) * 8   /* your word size */

#define   NUMWORD          (MAXBITS/WORDSIZE)
#define   BIGMAX           (NUMWORD + 1)
#define   MAXSTRLEN        (BIGMAX * WORDSIZE/3)
                                  ⋮
typedef   unsigned int     _Element;
typedef struct {
            _Element       val[4 * BIGMAX];
} bigint;
```

You can see that WORDSIZE $=$ UINTSIZE because sizeof(int) $= 4$. The literal BIGMAX is useful in for-loop because it is the number of words used in an array that holds MAXBITS. NUMWORD is the max index into an array of words we need to express MAXBITS. And MAXBITS is the total number of bit we expect to be working with; the exact value can be anything but it is desirable to take as input parameter from command line. When computing MAXSTRLEN, because $\log 2 \approx 0.3$ we multiply by $\frac{1}{3}$. Lastly, because we account for both multiplication (2 times) and an additional wasted space (1 times) we set up as 4 times larger than our exact size to hold a big integer. **You should check whether your machine works on 64-bit platform and adjust all literals so that are consistent with your system.**

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes a list of processing results, $z_i = \texttt{op}(x_i, y_i)$, as character strings where op $\in \{$add, subtract, multiply, divide$\}$.

z_1
z_2
...
z_t

As mentioned above, you need to implement an algorithm to covert big integers into character strings to write them at an ASCII text file. Otherwise, the file writes big integers as unreadable characters. One option is

```
void encodeBig2Str(char* s, bigint* x);
```

References

[GT14] M. Goodrich and R. Tamassia, *Algorithm Design and Applications*, Wiley, 2014.

[HSA08] E. Horowitz, S. Sahni, and S. Anderson-Free, *Fundamentals of Data Structures in C*, Second edition, Silicon Press, 2008.

[Sed98] R. Sedgewick, *Algorithms in C++*, Third edition, Addison-Wesley Company, 1998.