Data Structure and Algorithm, Spring 2019

Homework 1

Release: Tuesday, March 5, 2019

Due: 23:59:59, Sunday, March 17, 2019

TA E-mail: `dsa1@csie.ntu.edu.tw`

Rules and Instructions

- In homework 1, the problem set contains 5 problems and is divided into two parts, non-programming part (problem 1, 2, 3) and programming part (problem 4, 5).
- Please go to *DSA Judge* (`https://dsa2019.csie.org`) and complete the first problem to familiarize yourself with the usage of the judge system.
- For problems in the non-programming part, you should combine your solutions in ONE PDF file, and then submit it via the judge system. Your file should be as clean as possible, and the use white text on black background is prohibited. Your solution must be as simple as possible. At the TAs' discretion, solutions which are too complicated will be regarded as incorrect. Moreover, if you would like to use any theorem which is not mentioned in the classes, please include its proof in your solution.
- For problems in the programming part, you should write your code in C programming language, and then submit it via the judge system. You can submit up to 5 times per day for each problem. The compilation command of the judge system is `gcc main.c -static -std=c11 -O2 -lm`.
- Discussions with others are encouraged. However, you should write down your solutions in your own words. In addition, for `each problem`, you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution of that problem in your solution.
- The score of the part that is submitted after the deadline will get some penalties according to the following rule:

$$LateScore = (\frac{86400 - DelayTime(sec.)}{86400})OriginalScore$$

Non-Programming Part

Problem 1. Stack/Queue (15 *points*)

1. (9 *points*) Please implement a **min stack** using 2 stacks and O(1) extra space. There are four functions that you should implement, and all the functions should run in O(1)-time. You can assume that the stacks have infinite capacity and all operations are valid. For instance, no pop() will be called on an empty stack. In these functions, please declare all the variables that you need.

    1. push(x) – Push element x into the stack.
    2. pop() – Removes the element from the top of the stack.
    3. getMin() – Get the value of minimal element from the stack. (The minimal element won't be removed)
    4. top() – Get the value of the top element in the stack.

    You can explain how you implement the function in plain text or in C code. However, make sure that your method is clearly described, or you may receive some penalty in score.

    Ans:

```
//S1 stores data and S2 stores min value.
void push(x){
    S1.push(x);
    if(S2.empty())
            S2.push(x);
    else
            S2.push(min(x, S2.top()));
    return;
}
void pop(){
    S1.pop();
    S2.pop();
    return;
}
int getMin(){
    return S2.top();
```

```
}
int top(){
    return S1.top();
}
```

2. (6 *points*) The double-ended queue is a data structure similar to a queue. However, we can push to or pop from both ends. That is, you have four functions: `pushBack`, `pushFront`, `popBack`, `popFront`. You are given a fixed "push order" : (`1,2,3,4,5`), which means that you can only push "2" if you have already pushed "1", and a desired "pop order" : (`3,4,1,2,5`), which means that you can only pop "4" if you have already pop "3". Your job is to decide whether there is a sequence of operations that will satisfy both the "pop order" and the "push order".
   If there are solutions, give any one of them. Otherwise, you have to write down "WA". See the example below to understand more about the question.

   Example: push order :(1,2,3,4,5) -> pop order :(3, 4, 1, 2, 5)
   Solution:

Operation	Queue content
pushFront(1)	[1]
pushFront(2)	[2, 1]
pushFront(3)	[3, 2, 1]
3=popFront()	[2, 1]
pushFront(4)	[4, 2, 1]
4=popFront()	[2, 1]
1=popBack()	[2]
2=popBack()	[ ]
pushFront(5)	[5]
5=popFront()	[ ]

   (a) (2 *points*) push order :(1,2,3,4,5) -> pop order :(3,5,2,1,4)
   Ans:

pushFront(1)	[1]
pushFront(2)	[2, 1]
pushFront(3)	[3, 2, 1]
3=popFront()	[2, 1]
pushBack(4)	[2, 1, 4]
pushFront(5)	[5, 2, 1, 4]

5=popFront() [2, 1, 4]
2=popFront() [1, 4]
1=popFront() [4]
4=popFront() [ ]

(b) (2 *points*) push order :(1,2,3,4,5) -> pop order :(5,2,3,4,1)
Ans: WA

(c) (2 *points*) push order :(1,2,3,4,5) -> pop order :(2,4,3,5,1)
Ans:
pushFront(1) [1]
pushFront(2) [2, 1]
2=popFront() [1]
pushFront(3) [3, 1]
pushFront(4) [4, 3, 1]
4=popFront() [3, 1]
3=popFront() [1]
pushFront(5) [5, 1]
5=popFront() [1]
1=popFront() [ ]

Problem 2. Complexity (20 *points*)

$$(\lg(n) = \log_2(n),\ \ln(n) = \log_e(n),\ \log(n) = \log_{10}(n))$$

1. (6 *points*) Read the following C-code. For each function, find out the tightest bound of its running time using Θ notation. Brief explanation is needed. Your answer should be as simple as possible. For instance, $\Theta(n^2 + n)$ should be written as $\Theta(n^2)$.

   a. (1 *points*)

```
void func(int n) {
    int sum = 0;
    for (int i = 1; sum < n; i++) {
        sum = sum + i;
    }
}
```

   Ans: $n^{\frac{1}{2}}$

   b. (1 *points*)

```
void func(int n) {
    int i = 1, j = 1;
    while (i < n || j*j*j < n) {
        i = i * 2;
        j = j + 1;
    }
}
```

   Ans: $n^{\frac{1}{3}}$

   c. (2 *points*)

```
void func(int n) {
    int i = 2;
    while (i < n) {
        i = i * i;
    }
}
```

   Ans: $lg(lg(n))$

   d. (2 *points*)

```
void func(int n) {
    if (n == 0) {
        return;
    }
    if (n <= 1000) {
        func(n-1);
        func(n-1);
    } else {
        func(n-1);
        func(n-1);
        func(n-1);
    }
}
```

Ans: 3^n

2. (9 *points*) Prove or disprove the following statements. You should provide a formal proof or a counterexample for each statement. Please note that in the following statements, $f(n), g(n), i(n), j(n)$ are *non-negative*, *monotonically increasing* functions.
non-negative: The ranges of these functions are all non-negative.
monotonically increasing: If $n_1 > n_2$, then $f(n_1) \geq f(n_2)$.

   a. (1 *points*) If $f(n) = O(i(n))$, $g(n) = O(j(n))$, then $f(n) - g(n) = O(i(n) - j(n))$.
   Disprove.
   Let $f(n) = 2n,\ g(n) = i(n) = j(n) = n$
   $\Rightarrow f(n) - g(n) = n \neq O(n - n) = O(0)$

   b. (2 *points*) If $f(n) = O(i(n))$, $g(n) = O(j(n))$, then $f(n) * g(n) = O(i(n) * j(n))$.
   Prove.
   $f(n) = O(i(n))$
   $\Rightarrow \exists c_1,\ n_1 \in \mathbb{N}\ s.t.\ \forall n \geq n_1,\ f(n) \leq c_1 \times i(n)$
   $g(n) = O(j(n))$
   $\Rightarrow \exists c_2,\ n_2 \in \mathbb{N}\ s.t.\ \forall n \geq n_2,\ g(n) \leq c_2 \times j(n)$
   Take $c = c_1 \times c_2, n^* \geq max(n_1, n_2)$
   $\Rightarrow \forall n \geq n^*, f(n) * g(n) \leq c_1 c_2 i(n) j(n) = c * i(n) j(n)$
   $\Rightarrow f(n) * g(n) = O(i(n) * j(n))$

   c. (2 *points*) If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$
   Disprove.

Let $f(n) = 2n,\ g(n) = n$
$2^{f(n)} = 2^{2n} = 4^n \neq O(2^n)$

d. (2 *points*) $f(n) + g(n) = \Theta(\max(f(n), g(n)))$

Prove.
$max(f(n), g(n)) \leq f(n) + g(n) \leq 2 * max(f(n) + g(n)) \Rightarrow$
Take $c_1 = 1, c_2 = 2$, and then the statement is therefore proved

e. (2 *points*) $\log(n!) = \Theta(nlogn)$

Prove.
(1) $log(n!) = \sum_{i=1}^{n} log(i) \leq nlog(n)$
$\Rightarrow log(n!) = O(nlogn)$
(2) $log(n!) = \sum_{i=1}^{\frac{n}{2}} log(i) + \sum_{i=\frac{n}{2}+1}^{n} log(i) \geq \sum_{i=\frac{n}{2}+1}^{n} log(i)$
$\geq \frac{n}{2} \times log(\frac{n}{2}) = \frac{n}{2} \times (log(n) - log2) = \frac{1}{2}nlogn - \frac{log2}{2}n$
$\Rightarrow log(n!) = \Omega(nlogn - n) = \Omega(nlogn)$
By (1), (2)$\Rightarrow log(n!) = \Theta(nlogn)$

Hint : Use the definitions of the asymtotic notation as starting points.

3. (5 *points*) Define a function $f(n)$ as follows:

$$f(n) = \begin{cases} 1 & \text{, if } n = 1 \\ 2f(\lfloor \frac{n}{2} \rfloor) + n & \text{, otherwise} \end{cases}$$

Please find out the tightest bound of $f(n)$ with Θ notation. You should provide a proof to get credits. Answers without any proof will NOT be given any credits.

$f(n) = O(nlogn)$
$\forall k \in \mathbb{N},\ f(2^k) = 2f(2^{k-1}) + 2^k = 4f(2^{k-2}) + 2^k + 2 * 2^{k-1} = ...$
$= 2^k f(1) + k(2^k) = (k+1)2^k$
$\forall n \in \mathbb{N},\ \exists k\ s.t. 2^k \leq n < 2^{k+1} \Rightarrow k \leq lg(n) \leq k+1$
$(k+1)2^k = f(2^k) \leq f(n) < f(2^{k+1}) = (k+2)(2^{k+1})$

$f(n) < (k+2)2^{k+1} \leq (lg(n)+2)2^{lg(n)+1} = (lg(n)+2)2^{lg(2n)} = 2n(lg(n)+2)$
$\Rightarrow f(n) = O(nlg(n))$
$f(n) \geq (k+1)2^k > (lg(n) - 1 + 1)2^{lg(n)-1} = lg(n) * 2^{lg(\frac{n}{2})} = \frac{n}{2}lg(n)$
$\Rightarrow f(n) = \Omega(nlg(n))$
$\Rightarrow f(n) = \Theta(nlg(n))$

Or you can prove it by recursive tree method.

Problem 3. Linked List (25 *points*)

You learned in class that both *arrays* and *linked lists* can be used to store data. Arrays need only $O(1)$-time to index an item, while linked lists need $O(n)$-time. Linked lists need only $O(1)$-time to insert/delete an item, while arrays need $O(n)$-time. At first glance, you may think that we can choose which of these two data structures we want to use depending on the operations we need to support. However, in practice, we often need both insert/delete and index operations. For example, the following seemingly simple problem:

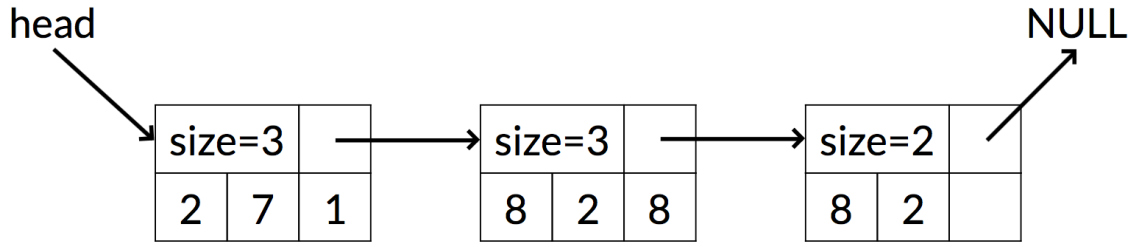
Given a list of integers, you need to support three kinds of operations.

1. `void insert(int pos, int x)`: Insert an integer x after the pos^{th} item. If $k = 0$, it is inserted at the front of the list.
2. `void delete(int pos)`: Delete the pos^{th} item.
3. `int query(int pos)`: Return the value of the pos^{th} item.

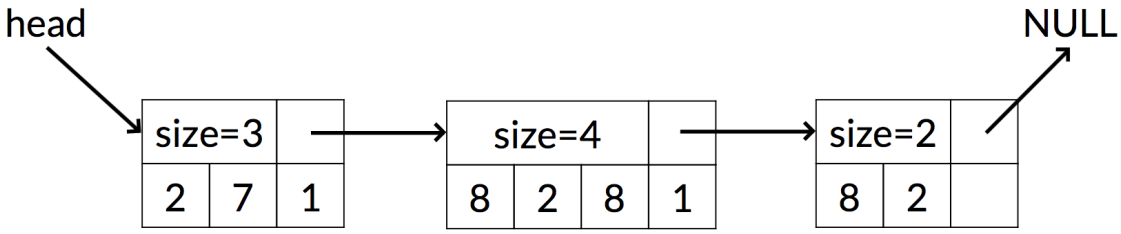
The list has n items originally, and n operations will be executed.

Since indexing and inserting/deleting items will all be required in this problem, no matter we use *array* or *linked list*, the worst-case time complexity of an arbitrary sequence of operation is $O(n^2)$, which is not satisfactory.

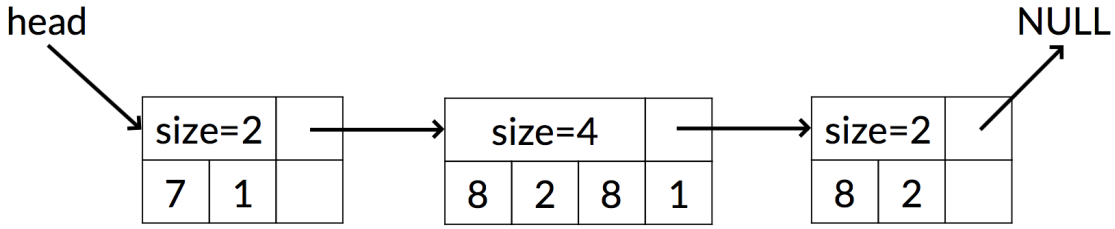
Therefore, here we introduce a new data structure that combines the benefits of both *array* and *linked list*, named *unrolled linked list*. Unrolled linked list is similar to linked list, but instead of storing one item in each node, it stores an array of items in each node. If we combine the arrays in each node sequentially, we get the original list. The unrolled linked list will be maintained that each node stores approximately K items, and there will be approximately $L = \frac{n}{K}$ nodes,where n is the number of items in the list. If the original list is $[2, 7, 1, 8, 2, 8, 8, 2]$, $n = 8$, and we choose $K = 3$, the constructed unrolled linked list looks like this:



After the operation `insert(6, 1)`, the unrolled linked list may look like this:



Then, after the operation `delete(1)`, the unrolled linked list may look like this:



Now, the list is $[7, 1, 8, 2, 8, 1, 8, 2]$, and `query(1)` should return 7. Below is a code segment that defines the structure `Node` of unrolled linked list, and the functions `init` and `query`.

```
typedef struct Node{
    int size;
    int array[MAXK];
    struct Node *next;
} node;
node *head;
int K;
void init(int n, int A[]){ //constructs the unrolled linked list by the input list.
    K = ?;
    head = (node*)malloc(sizeof(node));
    head->size = 0;
    head->next = NULL;
    node *now = head;
    for(int i = 0; i < n; i ++){
        if(now->size == K){
            node *tmp = (node*)malloc(sizeof(node));
            tmp->size = 0;
            tmp->next = NULL;
            now->next = tmp;
            now = tmp;
        }
        now->array[now->size ++] = A[i];
    }
}
```

```
void locate(int *pos, node **now){ //locates which node pos is at and its array index.
    while(*pos > (*now)->size){
        *pos -= (*now)->size;
        *now = (*now)->next;
    }
}
int query(int pos){
    node *now = head;
    locate(&pos, &now);
    return now->array[pos - 1];
}
```

1. (18 *points*) Write the code of the functions `insert` and `delete`, and decide the value K(it may be expressed by n) in the function `init`, so that the main function works correctly and efficiently. The main function reads the original list, calls `init`, and calls the three functions n times in total. Both C code and C-style pseudo code are acceptable.

   Grading Policy:

   - If you get either the function `insert` or `delete` correct, you can get 6 points.
   - If you get both the functions `insert` and `delete` correct, you can get 12 points.
   - If you get both the functions `insert` and `delete` correct, and the main function runs in $o(n^2)$ time (note that this is the little-O notation), you can get 18 points.

   Hint: To achieve correct worst case time complexity, when you `insert` an item, you may need to slice a node into two nodes sometimes.

   Solution: $K = sqrt(n)$

```
void insert(int pos, int x){
    node *now = head;
    locate(&pos, &now);
    for(int i = now->size - 1; i >= pos; i --){
        now->array[i + 1] = now->array[i];
    }
    now->array[pos] = x;
    now->size ++;
    if(now->size == 2 * K){
        node *tmp = (node*)malloc(sizeof(node));
        tmp->size = K;
        tmp->next = now->next;
        now->next = tmp;
        for(int i = 0; i < K; i ++){
```

```
                tmp->array[i] = now->array[i + K];
            }
            now->size = K;
        }
    }
    void delete(int pos){
        node *now = head;
        locate(&pos, &now);
        for(int i = pos - 1; i < now->size - 1; i ++){
            now->array[i] = now->array[i + 1];
        }
        now->size --;
    }
```

2. (7 *points*) Please prove the worst-case time complexity of the main function in the previous problem. If the main function of the previous problem doesn't run in $o(n^2)$-time, but you prove its complexity correctly, you can still get 4 points.

   Solution:

   According to the code, except the last node, each node has an array size between K and $2K$ throughout the whole process. Also, the max number of items in the list is $2n$ throughout the whole process. Therefore, the max number of nodes is $\frac{2N}{K}$. So both the functions `insert` and `delete`'s running time are $O(K + \frac{n}{K})$, and the function `query`'s running time is $O(\frac{n}{K})$, so the total running time is $O(n(K+\frac{n}{K}))$. Since we choose $K = \sqrt{n}$, the total running time is $O(n\sqrt{n}) = o(n^2)$.

3. (1 *point*, hard) If instead of n operations, m operations are executed, where m is another variable unrelated to n, and we don't know the value of m beforehand, please modify the program so that the main function runs in $o((n+m)^2)$, and derive its time complexity. You can simply describe your method without writing codes.

   Formal definition of $o(g(n,m))$: $o(g(n,m)) = \{f(n,m) \mid$ for any constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n,m) \leq cg(n,m)$ for all n, m where $max(n,m) \geq n_0$

   Solution:

   The key point is to routinely reconstruct the unrolled linked list using a new appropriate K value. We can do it by reforming the list using the unrolled linked list, and calling `init` again. So when should we reconstruct it? Here three algorithms are provided.

   - Reconstruct the list whenever the number of nodes becomes $2K$.
   - Reconstruct the list whenever the square root of the list size exceeds K.

- Reconstruct the list whenever a node has an array size $2K$.

All three algorithms run in $O((n+m)^{1.5})$ time. The proof of the complexity of the first two algorithms are trivial, and the third one requires some mathematical analysis.

Programming Part

Problem 4. Arvin and the Pancake Tower (20 *points*)

Arvin is a pancake lover.
At the same time, he is also a stack lover.
Everyday, he makes pancakes of different sizes to share with his friends.

He makes these pancakes with a frying pan and two plates. The two plates act as two stacks, respectively used as temporary pancake storage and the final pancake tower holder. Every time a new pancake is made, it is PUSHed onto the first plate, and at any time desired, pancakes can be POPed from the first plate and PUSHed onto the second plate. Pancakes that are PUSHed onto the second plate stays there and wait to be served.

Life was enjoyable when making those pancakes.
But Arvin soon notices a serious problem : How should he stack up the pancakes so that they form a nice looking tower?

Having learned of his problem, his friend Piepie comes up with a nice way to evaluate how good a tower looks. He regard a tower as a number with base $K + 1$ (K being the largest possible size of a pancake), and the size of each pancake in the tower represents a digit in the number, where lower pancake in the tower stands for more significant digit. Piepie thinks a tower that represents a larger number looks better. Arvin agrees with Piepie, but still has trouble figuring out what the tower with the largest value would look like. Now, can you help him arrange his pancakes to form the best-looking pancake tower?

BONUS:
Another problem setting: Arvin gets hungry while making those pancakes, and would like to eat some pancakes to soothe his hunger. Can you come up with the best pancake tower possible if Arvin eats exactly X pancakes?

Input Format

In the first line, there are two integers N and X (seperated by a space), indicating the number of pancakes made and the number of pancakes Arvin will eat.

In the second line there are N space-separated integers that represent the sizes of the pancakes.

Output Format

A single line of $N - X$ space-separated numbers, showing the sizes of the pancakes from the bottom to the top of the tower.

Input Constraint

Testgroup0 (40%), $0 < N, K \leq 2 \times 10^6$, $X = 0$, and the size of each pancake is unique.
Testgroup1 (40%), $0 < N \leq 10^4$, $0 < K \leq 2 \times 10^6$, $X = 0$.
Testgroup2 (20%), $0 < N, K \leq 2 \times 10^6$, $X = 0$.
Testgroup3 (0%, BONUS), $0 < N, K \leq 2 \times 10^6$, $X \leq N$.

Sample Input1

```
5 0
1 4 3 5 2
```

Sample Output1

```
5 3 4 2 1
```

Sample Input2

```
7 0
1 2 1 2 5 3 1
```

Sample Output2

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

Sample Input3

```
7 2
1 2 1 2 5 3 1
```

Sample Output3

```
5 3 2 2 1
```

Problem 5. Life Editing (20 *points*)

Human's life is like a string, and each word shows what you have experienced.

For Q-mao, to decide a person's life string is too difficult. So please write a program to help him modifying the life string more easily.

This program must support the following operations.

Input Format

The first line is an integer N, indicating the number of editing history.

Each of the next N lines contain a command of one of the forms as follows:

`1 i c` : insert a character c after the i^{th} character in the life string.

`2 i` : delete the $(i+1)^{th}$ character in the life string.

`3 l r` : print the $[l+1, r]^{th}$ characters in the life string.

Input Constraint

Testgroup0 (40%), $0 < N \leq 3 \times 10^3$

Testgroup1 (60%), $0 < N \leq 10^5$

For all testdata, $0 < |characters\ you\ should\ print| \leq 10^8$, and there is no illegal command.

Output Format

For each command "`3 l r`", print a line with the life string at that moment.

Sample Input1

```
5
1 0 A
1 0 B
1 2 C
2 1
3 0 2
```

Sample Output1

```
BC
```