Data Structure and Algorithm, Spring 2019

Homework 2

Release: Saturday, March 30, 2019

Due: 23:59:59, Thursday, April 11, 2019

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

Rules and Instructions

- In homework 2, 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:

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

Problem 1. Handsome and his Mysterious Jets (18 *points*)

Handsome, an ambitious student, always seeks opportunities to learn every kind of knowledge and improve his professional skill. One day, he invented a jet, intending to fly high to the sky, even to the space. The jet is elaborately designed, so the altitude of the place it lands on must be no less than that of the place it took off. Such restriction refines its flying path.

Handsome would like to fly through a sequence of mountains from west to east, on top of which airports and runways are constructed. The jet can fly as far as Handsome wishes, but he still sometimes wants to land on some mountains to see the scenery. Moreover, Handsome loves efficiency, so he never travels westward.

In the following questions, all the counts start from 0, and all the mountains have unique altitude.

1. (3 *points*) Consider 5 mountains with heights $[3, 2, 4, 5, 1]$, from west to east. Please list all the feasible flying hops. For example, $(0, 3)$ is a feasible hop since mountain 3 is taller than mountain 0, so the plane can take off at mountain 0 and land on mountain 3. On the other hand, $(0, 4)$ is not a feasible flying hop.

   There are 5 feasible hops, $(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)$ respectively.

2. (4 *points*) Consider N mountains on the west side and M mountains on the east side. Both the west and east sides of mountains are coincidentally sorted respectively from the shortest to the tallest, from west to east. Handsome want to take off from one of the west side mountain and land on one of the east side mountain without stopover. Given the mountain altitudes of the N west mountains and the M east mountains sequentially, please design an algorithm to find the number of feasible flying hops in $O(N + M)$ time.

   Denote x_i to be the $i-th$ mountain on the west side, and y_i be the $i-th$ mountain on the east side. If x_1 is shorter than y_1, than hops from x_1 to y_i for all $0 \leq i < M$ will all be feasible since the array is sorted. Therefore, we can generalize the algorithm to the following pseudo code. The algorithm linearly scan through both sides of mountains once, and for scanning a single element cost constant time. Thus, the overall time complexity is $O(N + M)$.

```
ptr = 0, ans = 0
for i in 0..(N-1):
    while ptr < M:
        if west[i] < east[ptr]:
            ans += (M-ptr)
            break
        else ptr++
```

```
return ans
```

3. (6 *points*) Consider N mountains. Given the heights of the N mountains from west to east, please design an algorithm to calculate the number of feasible flying hops in $O(N \lg N)$ time.

   This algorithm is quite similar to *Merge Sort*. In the merging part, we can use the procedure in (2) to calculate the number of feasible hops between the two segments, which causes only constant time overhead. Therefore, the complexity is $O(N \lg N)$, same as merge sort.

4. (5 *points*) After hundreds of years, earthquake occurs much more often than before, and each time a segment of mountains will be reversed in order. In each earthquake record, there are two numbers L and R indicating that all the mountains between mountain L and mountain R (both inclusive) are reversed in order. Given Q earthquake records in chronological order, please design an algorithm based on (3) to determine after EACH earthquake, whether the number of feasible flying hops are odd number or not. Your algorithm should run in $O(N \lg N + Q)$ time.

   If there are X feasible hops between mountain L and R, then after the earthquake, the number of feasible hops in this segment will become $\frac{(R-L)\times(R-L+1)}{2} - X$. The difference in the total feasible hops will thus be $|\frac{(R-L)\times(R-L+1)}{2} - 2X|$, where $2X$ is always an even number. Therefore, if $\frac{(R-L)\times(R-L+1)}{2}$ is an even number, then the parity won't change, and if it is an odd number, then the parity will change. The calculation costs $O(1)$ time, so after calculation of the initial number of feasible hops in $O(N \lg N)$ time as mentioned in (3), the time cost for the Q queries will be $O(Q)$, which implies the total time complexity $O(N \lg N + Q)$.

In sub-problem 2-4, you should briefly justify the correctness and time complexity of your solution to receive full credits of this problem. Both pseudo-code and C code are accepted.

Problem 2. Algorithmic Complexity Attack (1) (14 *points*)

We've known that in some algorithms, the average-case time complexity is different from the worst-case time complexity. If a server applies such kind of algorithms and the attackers are able to intentionally put a bad input sequence to make the worst-case occur, then the computation resources of the server can be exhausted. Once the resource is exhausted, or the server is overloaded, the server can hardly provide services to other benign users. We call such attack "*Algorithmic Complexity Attack*," which is a kind of "*Denial of Service (DoS)*" attack. We say an algorithm might be vulnerable to such attack if a malicious input can result in a worse time complexity than the average case. More specifically, if $f(x) = o(g(x))$, we say $g(x)$ is a worse time complexity than $f(x)$.

1. (2 *points*) Which of the following algorithms might be vulnerable to *Algorithmic Complexity Attack* if all the inputs are valid? Explain your reason by providing their worst-case and average-case time complexity.

   - (0.5 *points*) Quick Sort $O(N \lg N)/O(N^2)$
   - (0.5 *points*) Merge Sort $O(N \lg N)/O(N \lg N)$
   - (0.5 *points*) Selection Sort $O(N^2)/O(N^2)$
   - (0.5 *points*) Insertion Sort $O(N^2)/O(N^2)$

   Please refer to the in-class slides. Quick sort is vulnerable since its average-case complexity is $O(NlgN)$, while worst-case is $O(N^2)$. Others won't be vulnerable to such attack since all of them has no difference between their average-case and worst-case complexity.

2. (4 *points*) Consider there is a server providing quick sort service which takes a sequence of 10^6 unique positive integers as input from user, performs Quick Sort algorithm in ascending order with rightmost element always being the pivots, and then output the sorted number sequence. You, as an attacker, please provide a valid input sequence to most effectively enforce *Algorithmic Complexity Attack* to the server, and briefly explain why your input sequence can achieve such attack.

   $[1, 2, 3, ..., 999999, 1000000]$ serves as an example of the attack since the pivot is always on the rightmost position, and the pivot is always the largest element of the unsorted sequence.

3. (8 *points*) Consider the following C functions and pseudo codes. The pseudo codes come from Textbook Chapter 7.

```
unsigned long int next = 1;
```

```
int rand(void){ // RAND_MAX assumed to be 32767
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
void srand(unsigned int seed) {
    next = seed;
}

PARTITION(A, p, r):
    x = A[r]
    i = p - 1
    for j = p to r-1:
        if A[j] <= x:
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i + 1

RANDOMIZED-PARTITION(A, p, r):
    i = p + rand() % (r - p + 1)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r):
    srand(1126)
    if p < r:
        q = RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q-1)
        RANDOMIZED-QUICKSORT(A, q+1, r)
```

You, as an attacker again, would like to generate malicious input to attack the quick-sort server. However, this time the server modified the algorithm and used `RANDOMIZED-QUICKSORT` instead. How would you generate such "*Algorithmic Complexity Attack*" input sequence? In this problem, only methods and thoughts are required. You don't need to really generate such sequence or provide elaborate code. but be sure to convey your methods clearly. Note that your attacking sequence generation procedure should be within polynomial time.

Since the random seed is fixed, it is possible for us to predict the random number sequence. Therefore, we can simulate the quick sort procedure, and fill the pivot position with the largest element. In this way, the selected pivot on the server side will still be the largest ones, and the pivot will not be able to equally separate the sequence.

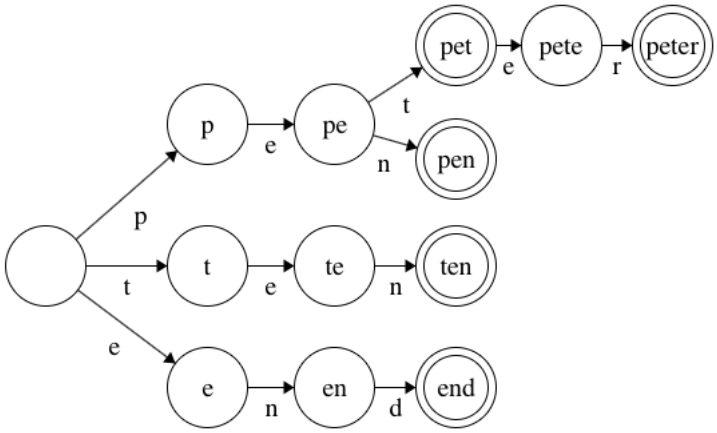
Problem 3. Aho–Corasick Automaton (28 *points*)

During the class, you should have learned the KMP string matching algorithm, which can efficiently find all occurrences of a given string P in another given string T. Aho-Corasick automaton (or Aho-Corasick algorithm) can be considered as an extended KMP algorithm. Given a string set S, it can find all the occurrences of any string $s \in S$ in another given string T. Before we introduce Aho-Corasick automaton, we need to introduce a data structure named `trie` first.

- ## Trie

  Trie (usually pronounced as "try" to be distinguished from "tree") is a data structure for storing a set of strings. Then we can efficiently find out whether a string is stored in a trie or not.

  Each edge on trie has a character and, for each node, it is associated to the string formed by concatenating all the characters on the path from root to the node. The root node is associated to an empty string. For example, let "vocabulary set" be the set of strings stored in the trie and "vocabularies" are the strings in vocabulary set, then a trie with the vocabulary set {"pen", "pet", "peter", "ten", "end"} looks like the figure below.



  The double-circled nodes are the end of the vocabularies in the vocabulary set. With the trie structure, you can find a string s on the trie in $O(|s|)$ time by "walking" on the trie nodes. You can also use trie to know whether a given string is in the vocabulary set or not.

- ## Aho-Corasick Automaton

  In the KMP algorithm, we construct prefix functions (i.e., π function in the lecture slides) and use them for string matching. The concept of Aho-Corasick automaton is applying

KMP algorithm to a trie, while the prefix functions becomes "fail links" that points to another node on trie.

The fail link of each node u points to the node that is associated to the u's longest suffix other than u itself. That is, for any node u, let $\pi[u]$ be the node pointed by the fail link of u, $str[u]$ be the string associated with node u; then, for any node u,

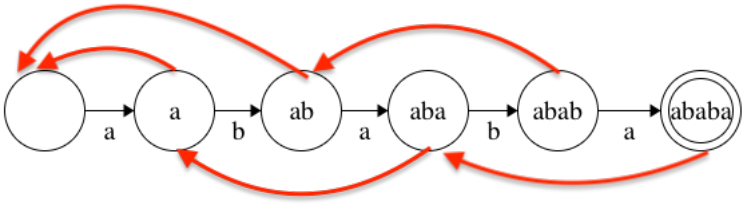
$$\pi[u] = \arg\max_{v}\{|str[v]|\} \text{ where } |str[v]| < |str[u]| \text{ and } str[v] \sqsupset str[u].$$

When performing string matching, you can use the fail link to find the longest matched prefix in the vocabulary set. Explicitly, you start from the root node of the trie. Let u be the current node and c be the encountered character. If $str[u] + c$ (i.e., the concatencation of $str[u]$ and c) exists on the trie, then move to the node associated with $str[u] + c$. Otherwise, let $u = \pi[u]$ and repeat the process until u becomes the root node.

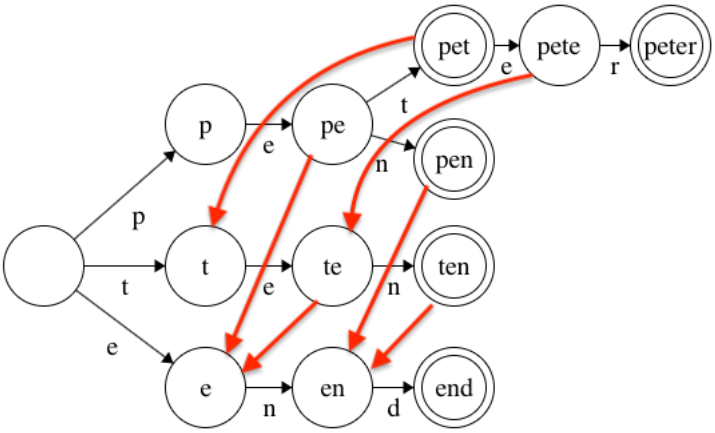
The C-like pseudo code below shows the string-matching procedure:

```
u = (root of trie);
for(int i=0; i<strlen(T); i++) {
     char c = T[i];
     while( !(str[u]+c appears in trie) && (u != (root of trie)) ) {
          u = fail_link[u];
     }
     if( (str[u]+c appears in trie) )
          u = (the node associates with str[u]+c);
}
```

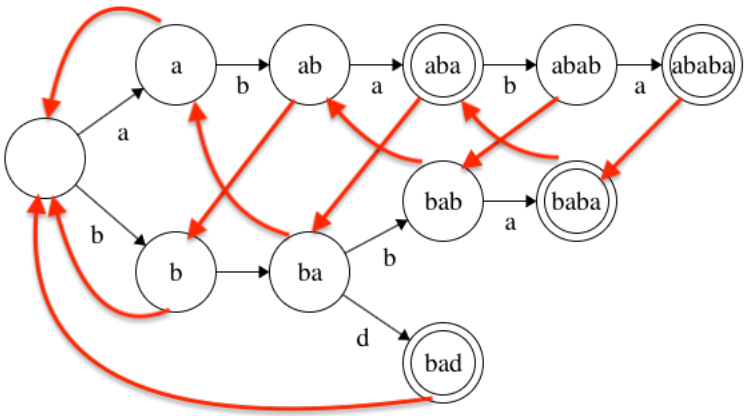
When there's only one vocabulary in trie, the fail links acts exactly same as prefix function. The figure below is a one-vocabulary trie with fail links (red arrows), i.e., an Aho-Corasick automaton.



The figure below is another Aho-Corasick automaton example. For simplicity, the fail links that point to the root are omitted in the figure.



1. (10 *points*) Please draw a Aho-Corasick automaton with the vocabulary set {"aba", "ababa", "bad", "baba"}. You will get at least 5 points if your trie structure is correct but the fail links are drown incorrectly.



2. (2 *points*) Show that every fail link points to a node with smaller depth.

   By the definition of the fail link, it should point to a node associated to one of its suffix other than itself. For any string s, The length of any s's suffix other than s itself is less than $|s|$. Because the depth of a node is the same as the length of the string associated with it, the fail link should point to a node with lower depth.

3. (8 *points*) Briefly explain that you can construct an Aho-Corasick automaton in $O(\sum_{i=1}^{n} |s_i|)$, where s_i are the strings in the vocabulary set and n is the size of the vocabulary set. You don't need to give a rigorous proof of correctness. However, please explain your procedure and how you achieve the desired time complexity.

   Let "non-trivial suffix" be the suffix of a string other than the string itself. For any node u, let $\pi[u]$ be the node pointed by the fail link of u, $str[u]$ be the string associated with

node u, $par[u]$ be the parent of u, c is the last character of $str[u]$.
Like KMP introduced in the slide, $\pi[u]$ should be the first node v on sequence

$$\pi[par[u]], \pi[\pi[par[u]]] \dots$$

such that $str[v] + c$ exists on the trie.
For any string s in the vocabulary set, we consider the nodes associates to s's prefixes. Because when we walk from root to the nodes associate with s, the depth of the node pointed by the fail links would either increment by 1 or decrease, the total time spent on these nodes is $O(|s|)$. Thus the time complexity for constructing fail links is $O(\sum_{i=1}^{n} |s_i|)$

4. (4 *points*) Given a string T and the vocabulary set $\{s_i | i = 1, \dots, n\}$, briefly explain that for each position i of T, you can find the longest s such that $s \sqsupset T[1..i]$ and s is a prefix of one of the vocabularies, in $O(|T| + \sum_{i=1}^{n} |s_i|)$.

   First construct an Aho-Corasick automaton in $O(\sum_{i=1}^{n} |s_i|)$ time like problem 3. Then run the string-matching algorithm mentioned in the problem description. For each position, the desired string is the string associated with the current node. The total time spent on switching between the nodes is $O(|T| + \sum_{i=1}^{n} |s|)$. The reason is similar to that of KMP.

5. (4 *points*) Given a string T and the vocabulary set $\{s_i | i = 1, \dots, n\}$, briefly explain that for each position i of T, you can find the longest s, if it exists, such that $s \sqsupset T[1..i]$ and s is one of the vocabularies, in $O(|T| + \sum_{i=1}^{n} |s_i|)$.
   (Hint: Not only the nodes you traversed on the trie, but also the nodes that can be reached from the traversed nodes by the fail links should be considered occurrence of the vocabulary. For example, when you find an occurrence of "ababa", there's also a 'baba' occurrence.)

   First, for each node, construct another fail link that points to the first-encountered end of one of the vocabularies. Let $F[u]$ be this special fail link. We can achieve this in $O(\sum_{i=1}^{n} |s|)$ time by finding them in the depth order. For any node u, if $\pi[u]$ is an end of one of the vocabularies, then $F[u] = \pi[u]$. Otherwise, $F[u] = F[\pi[u]]$.
   The remaining part is almost exactly the same as the previous problem. Though the desired string is $str[F[u]]$ rather than $str[u]$.

   In sub problem 3-4. and 3-5., you can assume that you only need to output an interval of T rather than print out the whole desired string. For example, if you find a vocabulary "lie" appears in T "believe", then you only need to output "[2, 4]" (O-indexed). This avoids a $O(|s_i|)$ time complexity on giving out an answer.

Programming Part

Problem 4. Pancake Party (20 *points*)

Arvin has finally solved the pancake tower problem, but now he has something else on his mind. To throw a successful party, great party games are necessary, so Arvin created a new game called Kono My Pancakes! (KMP! in short).

KMP! works like this :

Arvin would make many pancakes, each with a different amount of sugar in it. For each pancake, he would also brew a cup of tea with exactly the same amount of sugar added. Pancakes and tea would be labeled with sequence numbers. Now, Arvin challenges his guests to sort the treats by the amount of sugar in it (from least to most). To make the game a little easier, anyone is allowed bring up a pancake and a cup of tea to Arvin, and he would tell them which one has more sugar.

Other guests complain that KMP! is too hard and the party always ends before anyone solves it. Can you prove to them that the puzzle created by Arvin is actually solvable?

Input Format

The first line contains an integer N, indicating how many pancakes Arvin has made.

The second and third line each has N space-separated integers, representing the sequence number of pancake and tea.

Output Format

Print N lines, each consisting of two space-separated integers.

For each line, the first integer is the sequence number of the pancake, and the second is the sequence number of the corresponding cup of tea.

The N pairs should be printed in ascending order according to the amount of sugar inside.

Input Constraint

Sequence number for each pancake would be unique.

Sequence number for each cup of tea would also be unique.

Testgroup0 (20%), $0 < N \leq 10^4$

Testgroup1 (80%), $0 < N \leq 5 \times 10^5$

Notes

Note that this problem is an *Interactive* problem. That is, you need to ask the server for the secrets by calling the `query` function, and you can use those information to find the answer. Therefore, please include "*Arvin.h*" in your header so that your code can use the `query` function on the server side.

The function below will be present in the header file.

```
int query(int pancake_seq, int tea_seq){
    //return 1 if sugar in pancake > sugar in tea
    //return 0 if sugar in pancake == sugar in tea
    //return -1 if sugar in pancake < sugar in tea
}
```

For example, the following code segment is the query function for sample testdata. Note that for each test case, the content of the query function may be different.

```
int query(int pancake_seq, int tea_seq){
    int p_sugar = pancake_seq;
    int t_sugar = tea_seq+1;
    if(p_sugar>t_sugar)
        return 1;
    if(p_sugar==t_sugar)
        return 0;
    if(p_sugar<t_sugar)
        return -1;
}
```

Sample Input

```
5
6 2 3 5 4
2 3 1 4 5
```

Sample Output

```
2 1
3 2
4 3
5 4
6 5
```

Problem 5. Spell Runner (20 *points*)

In the Undying Lands around TA 1000, Manwe, king of Valar became aware that Sauron was gradually regaining his powers. In order to stop Sauron, five Maiars were summoned and sent to the Middle-earth, cloaked in the form of old men, to aid the Free People in the coming War of the Rings. Those five Maiers were later known as Istari, or more commonly refered to as the wizards. Fighting against Sauron was no easy task, and demanded strong spirit and skills. Two of the five wizards, namely Alatar and Pallando spent great time trying to interpret the Black Speech crafted by Sauron, and reached a surprising solution. The Black Speech itself is just normal spells, nothing to sweat about. What makes it so dangerous is that if any two substrings of spells with at most one mismatching character is chanted together, the power of the spell would increase drastically.

Knowing this fact, the two Blue Wizards decided to utilize this newly learned knowledge in the battle against Sauron. However, to do so, they must have the ability to identify whether two strings have at most one mismatching character. (The formal definition of having at most one mismatching character is that for two strings $\{S_1, S_2\}$, the strings must be of same length and there exist at most one i such $S_1[i] \neq S_2[i]$. For instance, $\{abc, aac\}$ satisfies the rules, while $\{aa, aac\}$ and $\{abc, acb\}$ does not)

Now, given a vocabulary of the Black Speech, can you possibly devise a fast method that can help the two wizards to save the world?

Input Format

The first line contains an integer N, and the following N lines are the spells we currently know. These N spells are separated by newline characters and indexed from 0 to $N-1$.

After the $N+1$ lines, there's an integer Q, indicating the number of queries.

Each query contains six integers, i_1, l_1 , r_1 , i_2, l_2 ,r_2, specifying two substrings. The first one is the substring of the $(i_1+1)^{th}$ spell starting at the $(l_1+1)^{th}$ and ending at the $(r_1+1)^{th}$ character, and the second one is the substring of the $(i_2+1)^{th}$ spell starting at the $(l_2+1)^{th}$ and ending at the $(r_2+1)^{th}$ character.

Output Format

For each query, if at most one character is different in the two substrings, output a single line of "Y"; otherwise, output "N".

Input Constraint

S_i means the i^{th} spell.

For 20% Testcase:

- $0 \leq Q \leq 10^4$, $\sum_{i=0}^{N-1} |S_i| \leq 10^4$

For 100% Testcase:

- The character set = { uppercase letters, lowercase letters, space, '!', '.', ',' }
- $0 < N \leq 10^4$, $0 \leq Q \leq 5 \times 10^5$, $\sum_{i=0}^{N-1} |S_i| \leq 2 \times 10^5$
- $0 \leq i_1, i_2 < N$
- $0 \leq l_1 \leq r_1 <$ |corresponding S_{i1}|
- $0 \leq l_2 \leq r_2 <$ |corresponding S_{i2}|
- $r_2 - l_2 = r_1 - l_1$

Sample Input

```
5
Sic vos non vobis...
Ari Ari Ari Ari Ari Arrivederci!
MUDAMUDAMUDAMUDAMUDAMUDA!
Ash nazg durbatuluk, ash nazg gimbatul
Ash nazg thrakatuluk, agh burzum-ishi krimpatul
3
0 0 0 1 0 0
0 0 2 1 0 2
3 0 9 4 0 9
```

Sample Output

```
Y
N
Y
```