The 5th Problem Set

March 16, 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 1.5 hours
   - Problem #2: Within 5 hours
   - Problem #3: Within 1.5 hours
   - Problem #4: Within 2 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

Let $\mathfrak{M}_{n\times m}(F)$ be the set of $n \times m$ matrices over a set F for positive integers n, m. When we write $A \in \mathfrak{M}_{n\times m}(F)$, we will write an $n \times m$ matrix $A = (a_{ij}) = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}$ for all real numbers $a_{ij} \in F$. For $A \in \mathfrak{M}_{n\times m}(F)$, let A^t be the transpose matrix of A; then $A^t = (a_{ji})$ and thus A^t is an $m \times n$ matrix whose element at (i, j) is a_{ji} of A. A sparse matrix $\mathfrak{S}_{n\times m} \subsetneq \mathfrak{M}_{n\times m}$ is a special case of $n \times m$ matrices, but it has at most 25% non-zero elements. For example, we say that $A \in \mathfrak{S}_{4\times 4}$ when its non-zero elements are two.

To obtain the transpose matrix A^t of $A \in \mathfrak{S}_{n\times m}$, a straightforward method relies on a nested `for` loop in terms of rows and columns. Accordingly, the straightforward method requires $O(nm)$ run-time complexity; More precisely, transposition requires $O(nm)$ integer comparisons and thus one needs to use such a nested `for` loop.

Now the mission in this problem is to devise an efficient algorithm to transpose A into A^t with $O(n+m)$ run-time complexity. To this end, you may need to use additional memory spaces.

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 dimension (n, m) of matrix and an indicator T for a base set F by which the data type of the input matrix is determined. As an example, T can be chosen from a predefined list $(\mathtt{Z}, \mathtt{Q}, \mathtt{R}, \mathtt{C}, \ldots)$ where `Z` means the integers, `Q` means the rational numbers and so on. Of course, you can specify a customized set such as a set `V` whose elements are positive and even numbers. In addition, the file should specify all $n \times m$ elements of a sparse matrix $A \in \mathfrak{S}_{n\times m}$; thus the input file is given in form as follows:

```
(n m)
T
a11 a12 ... a1m
a21 a22 ... a2m
...
an1 an2 ... anm
```

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the transpose matrix $A^t \in \mathfrak{S}_{m\times n}$ of $A \in \mathfrak{S}_{n\times m}$. Moreover, you are required to implement two algorithms; one is a basic algorithm whose complexity is $O(mn)$ and the other is its improved version whose complexity is $O(n + m)$. For this reason, a comparison of execution times between two algorithms should be given beneath the transpose matrix.

```
a11 a21 ... am1
a12 a22 ... am2
...
a1n a2n ... amn
************************************
O(n^2): _____ msec vs. O(n): _____ msec
************************************
```

Problem #2

Let $[1,n] := \{1,2,\ldots,n\}$ for convenience. Let S_n be the set of all permutations σ over where a bijective function from $[1,n]$ to $[1,n]$ is a permutation. That is,

$$S_n = \big\{\sigma \big| \sigma : [1,n] \to [1,n] \text{ is a bijection}\big\}$$

Then we can see that $|S_n| = n!$. For some $\sigma \in S_n$, usually we would write

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{pmatrix},$$

but from now on, we use $\sigma = (i_1, i_2, \ldots, i_k) \in S_n$ for some $k \leq n$ to denote a permutation defined by

$$\begin{cases} \sigma(i_j) = i_{j+1}, & j = 1,2,\ldots,k-1 \\ \sigma(i_k) = i_1, & \\ \sigma(\ell) = \ell, & \ell \notin \{i_1, i_2, \ldots, i_k\} \end{cases}$$

For example, $\sigma = (1,3,4,5) \in S_6$ means

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 2 & 4 & 5 & 1 & 6 \end{pmatrix}.$$

Hereafter we call $(i_1, i_2, \ldots, i_k) \in S_n$, **$k$-cycle** and in particular, when $k = 2$ we call (i_1, i_2) **transposition**. In fact, since $\sigma(i_1) = i_2$ and $\sigma(i_k) = i_1$, it is obvious that these permutations are cyclic. Furthermore, we can easily check that if σ is a transposition, then $\sigma \circ \sigma = id$ and $\sigma^{-1} = \sigma$ where $\circ$ is the composition of functions and id is the identity function. For example $(1,2) \circ (1,2) = id$.

Interestingly, all permutations can be written as a composition of disjoint cycles. For example, $\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 9 & 6 & 5 & 7 & 1 & 4 & 8 & 2 \end{smallmatrix}\right)$. Then because it has $1 \mapsto 3 \mapsto 6 \mapsto 1$, there is a 3-cycle $(1,3,6)$, similarly because it has $4 \mapsto 5 \mapsto 7 \mapsto 4$ we can find another 3-cycle $(4,5,7)$. Further, we can find a 2-cycle $(2,9)$ and that the permutation fixes 8. Hence, $\sigma = (1,3,6) \circ (4,5,7) \circ (2,9)$. Generally speaking, since any permutation in S_n is a composition of cycles and any cycle is a composition of transpositions, any permutation in S_n is a composition of transpositions. For example, consider $\sigma^* = (1,5,2,4,3)$. Then two expressions for σ^* as a composition of transpositions are

$$\sigma^* = (1,5) \circ (5,2) \circ (2,4) \circ (4,3)$$

and

$$\sigma^* = (1,2) \circ (3,4) \circ (2,3) \circ (1,2) \circ (2,3) \circ (3,4) \circ (4,5) \circ (3,4) \circ (2,3) \circ (1,2).$$

For an arbitrary permutation $\sigma \in S_n$, we can write it as a composition of r transpositions,

$$\sigma = (i_1, i_2, \ldots, i_k) = \tau_1 \circ \tau_2 \circ \cdots \circ \tau_r$$

where the τ_i's are transpositions and r is the number of transpositions. Although the τ_i's are not determined uniquely, $r \mod 2$ is determined uniquely (if $r \mod 2 = 0$, then r is even; otherwise r is odd). For instance, the two expressions for $\sigma^* = (1,5,2,4,3)$ as above involve 4 and 10 transpositions, which are both even because $4 \mod 2 = 10 \mod 2 = 0$. This fact makes it possible to define the sign function for a permutation as follows.

Definition 0.1 (sign) *Let S_n be defined as above. Given $\sigma \in S_n$, we define the function* $\operatorname{sgn} : S_n \to \{\pm 1\}$ *by*

$$\operatorname{sgn}(\sigma) = (-1)^r$$

when the permutation σ is the composition of r transpositions. We call the function sgn *signum or* ***sign*** *of the permutation σ.*

Now let us turn back to the main point. This problem interests in computing the determinants of arbitrary square matrices in $\mathfrak{M}_{n\times n}(F)$. Given a square matrix $A = (a_{ij}) \in \mathfrak{M}_{n\times n}(F)$ with each $a_{ij} \in F$ for a base set F, the determinant of A is defined as a function $\det(A) : \mathfrak{M}_{n\times n}(F) \to F$ and denoted by $\det(A)$. More concretely, for a given matrix $A \in \mathfrak{M}_{n\times n}(F)$ we define the determinant of A by

$$\det(A) = \sum_{\sigma\in S_n} \operatorname{sgn}(\sigma) \cdot a_{\sigma(1),1} \cdot a_{\sigma(2),2} \cdots a_{\sigma(n),n}$$

For example, consider a matrix $B = \left(\begin{smallmatrix} 2 & 3 \\ 1 & 2 \end{smallmatrix}\right) \in \mathfrak{M}_{2\times 2}(\mathbb{Z})$ where $F = \mathbb{Z}$. Then since $\sigma_1 = (1, 2), \sigma_2 = (2, 1)$, we see that

$$\begin{aligned} \det(B) &= \sum_{\sigma\in S_2} \operatorname{sgn}(\sigma) a_{\sigma(1),1} a_{\sigma(2),2} \\ &= \operatorname{sgn}(\sigma_1) a_{1,1} a_{2,2} + \operatorname{sgn}(\sigma_2) a_{2,1} a_{1,2} \\ &= (-1)^0 \cdot 2 \cdot 2 + (-1)^1 \cdot 3 \cdot 1 \\ &= 1. \end{aligned}$$

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 dimension (n, n) of matrix and an indicator T for a base set F by which the data type of the input matrix is determined. As mentioned before, you can specify a customized set such as a set `V` whose elements are positive and even numbers. In addition, the file should specify all n^2 elements of $A \in \mathfrak{M}_{n\times m}(F)$; thus the input file is given in form as follows:

```
(n n)
T
a11  a12  ...  a1n
a21  a22  ...  a2n
...
an1  an2  ...  ann
```

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the determinant of $A \in$ along with all elements.

$$\begin{vmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \dots & & & \\ \dots & & & \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{vmatrix} = \det(A)$$

Problem #3

In this problem we deal with a quite simple problem that is almost never related with mathematics. Let L be a sorted list of n distinct integers; that is, $L = (a_1, a_2, \ldots, a_n)$ where each $a_i \in [1, n+1]$. Then it is clear that in the list L, there is exactly one element $a \in [1, n+1]$ but $a \notin L$. Now design an efficient algorithm to find the integer a in this range that is not in L while having $O(\log n)$ run-time complexity. However, as before in Problem #1 you are required to present a naive algorithm with $O(n)$ run-time complexity.

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 first describes the number of elements $n > 1000$ and an unsorted list $L^* = (a_1, \ldots, a_n)$. If $n \leq 10^3$, then abort the program. The order of each element a_i is randomly selected during writing it at the file and a missing element also should be randomly picked up.

```
n
L*={a_1  a_2  ...  a_n}
```

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes the sorted list L and the finding result a in different lines. In addition, you need to report on a comparison result between your $O(n)$ algorithm and $O(\log n)$ algorithm in milli-seconds. During sorting the unsorted list L^*, you need to implement the `quicksort` algorithm but your implementation should be an **iterative** version of the `quicksort` algorithm. Notice that do not copy any kinds of `quicksort` algorithms publicly open in the Internet; instead refer to a proper textbook that describes the `quicksort` algorithm. Of course, sorting is not the main part of this problem.

```
L=(a_1  a_2  ...  a_n)

**************************************
O(n): _____ msec vs. O(log n): _____ msec
**************************************
```

Problem #4

This problem has a similar setting as in problem #3. However, this problem considers a concrete scenario as follows. Your mission is a part of automating a medicine prescription management system for an imaginary company, named `Abc`. More specifically, When a prescription order comes into the automating system, it is given as a sequence of ℓ medication requests,

$$a_1 \text{ oz of drug } X_1, a_2 \text{ oz of drug } X_2, \ldots, a_\ell \text{ oz of drug } X_\ell$$

where $a_i \in \mathbb{R}$ (the reals) and X_i is the name of medication. Examples of X_i's include Acetaminophen, Chrolpheniramine maleate, or Pseudoephedrine hydrochloride. We assume that $a_1 < a_2 < \cdots < a_\ell$.

In reality, `Abc` has an unlimited supply of n distinctly sized empty bottles for each medication. Each bottle has a tag specifying its capacity in ounces (for example, 4.23 oz and 134 oz). To process a medication order, you need to match each medication **request**–"a_i oz of drug X_i" with the **size** of the smallest bottle in the inventory than can contain a_i ounces. Therefore you need to design an algorithm to process such a medication order of ℓ requests while requiring at most $O(\ell \log(\frac{n}{\ell}))$ run-time complexity. Assume that the bottle sizes are stored in an array B, sorted by their capacities in ounces.

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 describes the sequence of ℓ pairs of requests, $(a_i, X_i)_{i\in[1,\ell]}$ and an sorted list $B = (b_1, b_2, \ldots, b_n)$. As described above, all a_i's are the real numbers.

$(a_1, X_1)\ \ (a_2, X_2)\ \ \ldots\ \ (a_\ell, X_\ell)$
$(b_1\ \ b_2\ \ \ldots\ \ b_n)$

Output format. The output should be given as a text-format file, named `output.txt`. The output file writes a list of triples of processing results, (b_j, a_i, X_i) for some $j \in [1, n]$ and $i \in [1, \ell]$.

$(b_{j_1}, a_1, X_1)\ \ (b_{j_2}, a_2, X_2)\ \ \ldots\ \ (b_{j_\ell}, a_\ell, X_\ell)$