# VE 281 Quiz 1

Name: 周骥岩N , ID No.: 518021911039

- **Problem 1.**

  We want to sort the following array in ascending order using quickSort (taught in class):

  7 2 8 5 10 6 9 4 1 3.

  We use the `quickSort(int A[], int left, int right)` function, where our "random" function always chooses **the first element** in array `A` as the pivot. Write down all the calls of `quickSort` until `A` is sorted. Hint: remember quickSort has two recursive calls: left than right. You may use the non-in-place partition for this question. Please annotate the version of the partition algorithm you pick.

  1st call: `quickSort({7, 2, 8, 5, 10, 6, 9, 4, 1, 3}, 0, 9)` not-inplace

  2nd call: quickSort({2, 5, 6, 4, 1, 3, 7, 9, 10, 8}, 0, 5)

  quickSort({1, 2, 3, 4, 6, 5, 7, 9, 10, 8}, 0, 0)

  quickSort({1, 2, 3, 4, 6, 5, 7, 9, 10, 8}, 2, 5)

  quickSort({1, 2, 3, 5, 6, 4, 7, 9, 10, 8}, 2, 1)

  quickSort({1, 2, 3, 5, 6, 4, 7, 9, 10, 8}, 3, 5)

  quickSort({1, 2, 3, 4, 5, 6, 7, 9, 10, 8}, 3, 3)

  quickSort({1, 2, 3, 4, 5, 6, 7, 9, 10, 8}, 5, 5)

  quickSort({1, 2, 3, 4, 5, 6, 7, 9, 10, 8}, 7, 9)

  quickSort({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 7, 7)

  quickSort({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 9, 9)

- **Problem 2.**

  Write the in place partition in pseudo code or plain English. And illustrate the operations of partition on the arrary: 6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3 (with the first element 6 as the pivot) step by step using the algorithm.

  ① swap pivot to the beginning of the array.

  ② let i=1, j=N-1

  ③ Increment i until A[i] ≥ pivot

  ④ decrement j until A[j] < pivot

  ⑤ If i<j, swap A[i] with A[j] and go back ③

  ⑥ Otherwise, swap the first element (pivot) with A[j]

  6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3

  6, 2, 3, 5, 11, 10, 4, 1, 9, 7, 8

  6, 2, 3, 5, 1, 10, 4, 11, 9, 7, 8

  6, 2, 3, 5, 1, 4, 10, 11, 9, 7, 8

  4, 2, 3, 5, 1, 6, 10, 11, 9, 7, 8

- **Problem 3.**

  Is selection sort stable?

  No.

  Assume we have the following elements and we use the integer part of the element as the key in ascending selection sort.

  (2, a) (2, b) (3, a) (1, a)

  Assume we **always swap** the current element with the smallest remaining element in the unsorted part of the array, even if they have equal keys. Please arrange the above elements into an array such that the stability is violated (if there exists any).

  (1,a) (2,a) (2,b) (3,a)

  Assume we **never swap** the current element with the smallest remaining element if they have equal keys. Please arrange the above elements into an array such that the stability is violated (if there exists any).

  (2,a) (2,b) (3,a) (1,a)

  Is bubble sort stable? Explain why (both bubble sort and selection sort swap elements. Please focus on their similarity and differences).

Yes. Both bubble sort and selection sort swap elements. But generally, bubble sort moves smaller to the front and larger to the end. The comparison and swap are between the two nearby elements. If the nearby elements have the equal key, we can choose to not swap them. So stability is kept. For selection sort, generally, it finds the smallest for the current position. The swap isn't between two nearby elements.

- **Problem 4.**

  With regard to all the algorithms we talked about in class (the class versions), please fill in the following table (use the big-O with the tightest bounds covered in class):

| | Worst-Case Time (Big-O) | Average-Case Time (Big-O) | In Place (Yes/No) | Stable (Yes/No) |
|---|---|---|---|---|
| Insertion Sort | $O(N^2)$ | $O(N^2)$ | Yes | Yes |
| Selection Sort | $O(N^2)$ | $O(N^2)$ | Yes | No |
| Bubble Sort | $O(N^2)$ | $O(N^2)$ | Yes | Yes |
| Merge Sort | $O(N \log N)$ | $O(N \log N)$ | No | Yes |
| Quick Sort | $O(N^2)$ | $O(N \log N)$ | Yes (Weakly) | No |

- **Problem 5.**

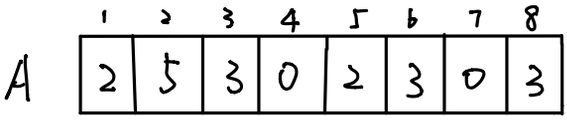
  In class, in the last step of the counting sort algorithm, we iterate the original unsorted array `A` backward as we fill in the sorted array. Explain all the necessary updates needed for the algorithm in order to traverse the original unsorted array `A` in a forward fashion.

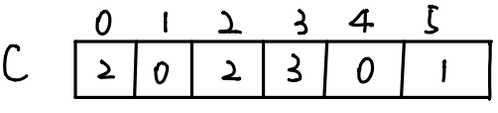
  Current steps in counting sort:

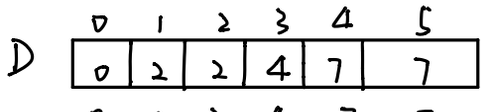
  1. Allocate an array `C[k+1]`.
  2. Scan `A`, `For i=1 to N: increment C[A[i]]`.
  3. `For i=1 to k: C[i] = C[i-1] + C[i]`.
  4. `For i=N to 1: put A[i] in new position C[A[i]], then decrement C[A[i]]`.

1. Allocate an array C[k+1]
2. Scan A, For i=1 to N: increment C[A[i]]
3. Allocate D[k+1] and D[0]=0

   For i=1 to N: D[i]=D[i-1]+C[i-1]
4. For i=1 to N: put A[i] in new position D[A[i]] then increment D[A[i]]




0 2 3 4 7 7
0 2 3 4 7 8
0 2 3 5 7 8
1 2 3 5 7 8
1 2 4 5 7 8
1 2 4 6 7 8
2 3 4 6 7 8
2 2 4 7 7 8

0 1 2 3 4 5 6 7

2
2 5
2 3 5
0 2 3 5
0 2 2 3 5
0 2 2 3 3 5
0 0 2 2 3 3 5
0 2 2 2 3 3 3 5

# VE 281 Homework 1

Name: 周维骁 , ID No.: 5180219110 39

- **Exercise 1.**

  **Definition 1** ($o$-Notation)**.** *Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$, written as $f(n) = o(g(n))$, if*

  $$\forall c > 0. \exists n_0. \forall n \geq n_0. f(n) < cg(n).$$

  An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows:

  $$f \mathcal{R} g \text{ if and only if } f(n) = \Theta(g(n)).$$

  A complexity class is an equivalence class of $\mathcal{R}$.

  The equivalence classes can be ordered by $\prec$ defined as: $f \prec g$ iff $f(n) = o(g(n))$.

  Example: $1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$.

  Please order the following functions by $\prec$ and give your (verbal or math) explanation:

  $$(\sqrt{2})^{\log n}, (n+1)!, ne^n, (\log n)!, n^3, n^{1/\log n}.$$

- **Exercise 2**

  Prove that $\log(n!) = \Omega(n \log n)$. Notice that after you prove this, you will understand the lower bound of comparison sort is $\Theta(n \log n)$.

  $\log n! = \log 1 + \log 2 + \cdots + \log n$

  $\geq \log \frac{n}{2} + \log\left(\frac{n}{2}+1\right) + \cdots + \log n$

  $\geq \frac{n}{2} \log \frac{n}{2}$

  $= \frac{1}{2} n (\log n - \log 2)$

  We want to prove $\log n - \log 2 \geq \frac{1}{2} \log n$

  for certain $n \geq n_0$.

  $\frac{1}{2} \log n \geq \log 2 \Rightarrow n \geq 4$

  Then $\log n! \geq \frac{1}{2} n \cdot \frac{1}{2} \log n = \frac{1}{4} n \log n$.

  Therefore, $\log(n!) = \Omega(n \log n)$

- **Exercise 3**

  Show how QUICKSORT can be made to run in $O(n \log n)$ time in the worst case.

  We can always find the median of the array in linear time.
  If we always choose the median as the pivot, the worst case will need $O(n \log n)$ time.
  So, we can find the median first, then choose it as the pivot and then do as the normal quicksort.

- **Exercise 4**

  Let $A$ be a list of $n$ (not necessarily distinct) integers. Describe an $O(n)$ -algorithm to test whether any item occurs more than $\lceil n/2 \rceil$ times in $A$.

  If an item occurs more than $\lceil n/2 \rceil$ times in A, it will be median of A. We can use randomized selection to find the median of A, and go through the array to count the occurrence. If it's more than $\lceil \frac{n}{2} \rceil$, then this value is true and others are false.

- **Exercise 5**

  Modify the $Merge(int * \ a, int\ left, int\ mid, int\ right)$ function taught in class and make it in-place (provide pseudo code). Explain why the new mergeSort might still run slower than quickSort in practice. Hint: Piazza.

```
Merge ( int * a, int left, int mid, int right) {
    int b = mid+1;
    if (a[mid]<= a[b]) return;
    while ( left <= mid && b <= right) {
        if (a[ left]<= a [ b]) left++;
        else {
            int v= a[b];
            int i = b;
            while ( i != left) {
                a[i] = a[i-1];
                i--;
            }
            a[ left]= v;
            left ++;
            mid ++;
            b++;
        }
    }
}
```

The time compexity of the new approach is $O(n^2)$. While for quicksort, it is $O(n \log n)$. Therefore, the new approach is slower.

(Use the source provided in piazza).

- **Exercise 6**

  Let A be a sorted array of integers and S a target integer. Design algorithms for determining if there exist a pair of indices i, j(not necessarily distinct) such that A[i]+A[j]=S.

  1. Design an $O(n^2)$ algorithm (stating it in plain English is OK).
  2. Design an $O(nlogn)$ algorithm (stating it in plain English is OK).

1. For i from 0 to n-1, we can go through the whole array to find A[i]+A[j]=S .
2. Since it is sorted, for i from 0 to n-1, we can use binary search to find S-A[i] in the array. If it can be found, then we can find j. otherwise, there is no pair.

- **Exercise 7**

  What is the most efficient sorting algorithm for each of the following situations and briefly explain:

  1. A small array of integers.
  2. A large array of integers that is already almost sorted.
  3. A large collection of htegers that are drawn from a very small range.
  4. Stability is reqired, i.e., the relative order of two records that have the same sorthg key should not be changed.

- **Exercise 8**

  Suppose you are given a set of mames and your job is to produce a set of unique first names. If you just remove the last name from all the names, you may have some duplicate first names. How would you create a set of first names that has each name occurring only once? Specifically, design an efficient algorithm for removing all the duplicates from an array.

First, we can use quicksort to sort the array so that all the duplicates are close to each other. Then we can create an array temp[] to store the unique elements. We put array[0] in temp[0], j=1.

For i=1 to N-1, if array[i-1] != array[i], we put array[i] in temp[j++]; Else if array[i-1] == array[i], we do nothing. Finally, temp[] is without duplicates.

- **Exercise 9**

  Suppose $\lim_{n\to\infty} f_1(n)/g_1(n)$ and $\lim_{n\to\infty} f_2(n)/g_2(n)$ exist, and we assume that all the functions are larger than 0 when $n > 0$. Judge whether the following statement is correct or not when $n \to \infty$. Justify your answers.

  1. $n \log n = O(n)$.
  2. $2^n = O(n!)$.
  3. If $f_1(n) = \Omega(g_1(n))$, $f_2(n) = \Omega(g_2(n))$, then $f_1(n)f_2(n) = \Omega(g_1(n)g_2(n))$.
  4. If $f_1(n) = \Theta(g_1(n))$, $f_2(n) = \Theta(g_2(n))$, then $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$.

1. $\lim_{n\to\infty} \frac{n\log n}{n} = \lim_{n\to\infty} \log n$, which diverges.

Therefore, it is wrong.

2. $\lim_{n\to\infty} \frac{2^n}{n!} = \lim_{n\to\infty} \frac{2}{1}\cdot\frac{2}{2}\cdot\frac{2}{3}\cdot\frac{2}{4}\cdot\frac{2}{5}\cdot\frac{2}{6}\cdot\frac{2}{7}\cdot\frac{2}{8}\cdots = 0$

Therefore, $2^n = O(n!)$

3. $f_1(n) = \Omega(g_1(n)) \Rightarrow$

$\exists C_1, n_1, \forall n$, $f_1(n) \geq c_1 g_1(n)$ for $n \geq n_1$

Similarily, $\exists C_2, n_2, \forall n$, $f_2(n) \geq C_2 g_2(n)$ for $n \geq n_2$

Then we can have

$f_1(n) f_2(n) \geq C_1 C_2 [g_1(n) g_2(n)]$ for $n \geq \max\{n_1, n_2\}$

Therefore $f_1(n) f_2(n) = \Omega(g_1(n) g_2(n))$

4. $f_1(n) = \Theta(g_1(n)) \Rightarrow$

$\exists C_1, C_2, n_1, \forall n$, $C_1 g_1(n) \leq f_1(n) \leq C_2 g_1(n)$

Similarily, $\exists C_3, C_4, n_2, \forall n$, $C_3 g_2(n) \leq f_2(n) \leq C_4 g_2(n)$

Then we have

$C_1 g_1(n) + C_3 g_2(n) \leq f_1(n) + f_2(n) \leq C_2 g_1(n) + C_4 g_2(n)$

$\min\{C_1, C_2\} \cdot \max\{g_1(n), g_2(n)\} \leq f_1(n) + f_2(n) \leq (C_2 + C_4) \cdot \max\{g_1(n), g_2(n)\}$

Therefore, $f_1(n) + f_2(n) = \Theta(\max\{g_1(n), g_2(n)\})$

- **Exercise 10**

  We want to find the $6^{\text{th}}$ largest element, which is 6, in the following array, Insertion sort is a simple and fast sorting algorithm when the length of array $n$ is short. However, when $n$ goes large, insertion sort may not be the best choice, as the worst case time complexity is $O(n^2)$. We can speed up insertion sort by combining it with `merge` in `mergeSort` we learnt in the lectures,

**Alg. 1:** `timSort`$(a[\cdot], x)$

**Input :** an array $a$ of $n$ elements, an integer $x > 0$ (you can assume that $x \ll n$)
**Output:** the sorted array of $a$

```
1 for i ← 0; i < n; i += x do
2     insertionSort(a, i, min(i + x − 1, n − 1));
3 for step ← x; step < n; step *= 2 do
4     for left ← 0; left < n; left += 2 × step do
5         mid ← left + step − 1;
6         right ← min(left + 2 × step − 1, n − 1);
7         merge(a, left, mid, right);
```

This algorithm is used as the default sorting algorithm in Java and Python. Here, we assume that $x \ll n$ is a known constant.

1. Suppose $n = 1000$ and $x = 32$. How many times will `insertionSort` be performed?
2. Suppose $x = 32$. Express in terms of $n$ how many comparisons in the worst case will be performed in `insertionSort`.
3. Express the worst case running time of the whole algorithm in terms of the big-Oh notation.
4. Is this algorithm in-place? If not, express the additional space needed in terms of the big-Oh notation.