Before you start:

Homework Files

You can download the starter files for coding as well as this *tex* file (you only need to modify *HW1.tex*) on canvas and do your homework with latex. Or you can scan your handwriting, convert to pdf file, and upload it to canvas before the due date. If you choose to write down your answers by hand, you can directly download the pdf file on canvas which provides more blank space for solution box.

Submission Form

A pdf file as your solution named as VE281_HW1_[Your Student ID]_[Your name].pdf uploaded to canvas
Estimated time used for this homework: **4-5 hours.**

0 Student Info (0 point)

Your name and student id:

Solution: Yinchen Ni 520370910026

1 Complexity Analysis (28 points)

(a) What is the time complexity of the following function? (6 points)

```
void question_1a(unsigned int n) {
   int result = 0;
   for (int i = 1; i < n; i *= 2) {
    for (int j = 1; j * j < n; j++) {
      result += j;
    }
   }
  cout << result << endl;
}
```

Solution: $O\left(\sqrt{n}\log n\right)$

(b) What is the time complexity of the following function? (6 points)

```
void question_1b(unsigned int m, unsigned int n, unsigned int p) {
   int count = 0;
   for (int i = m/2; i < m; i++) {
    for (int j = 1; j < n * n; j++) {
      count++;
    }
    for (int j = 114514; j > p; j--) {
      count--;
    }
   }
  cout << count << endl;
}
```

Solution: $O\left(mn^2\right)$

(c) Mr. Blue Tiger wants to create his own version of fibonacci sequence. Since 3 is his favorite number, he decides that any element should be the sum of its previous three elements. Can you help him figure out the time complexity of his recursive function? Select **All** the answers that are correct, and state your reason. (6 points)

```
int TigerNacci(unsigned int n) {
  if (n <= 3) return 1;
  return TigerNacci(n - 1) + TigerNacci(n - 2) + TigerNacci(n - 3);
}
```

i) $\Theta(n^3 \log n)$
ii) $\Theta(3^n \log n)$
iii) $O(3^n \log n)$
iv) $\Theta(3^n)$
v) $\Theta(n^2 \log n)$
vi) $\Theta(2^n \log n)$
vii) $O(2^n \log n)$
viii) $\Theta(2^n)$

Solution: iii) or vii).

Suppose the time complexity is $T(n)$, then

$$T(n) = \begin{cases} T(n-1) + T(n-2) + T(n-3) + C, & n > 3 \\ C & n \leq 3 \end{cases}$$

Let $T(n) = AT(n-1)$, then

$$A^3 = A^2 + A + 1$$

We get

$$A_1 = 1.84, \quad A_2 = -0.42 + 0.61\text{i}, \quad A_3 = -0.42 - 0.61\text{i},$$

and

$$T(n) = c_1 \cdot A_1^n + c_2 \cdot A_2^n + c_3 \cdot A_3^n + c_4.$$

Notice that $|A_2| = |A_3| < 1$, so when n is really large, the term $c_2 A_2^n$ and $c_3 A_3^n \to 0$. Therefore,

$$T(n) \approx c_1 \cdot A_1^n = \Theta(1.84^n) = O(2^n \log n) = O(3^n \log n)$$

(d) Sort the following functions with the $\prec$ notation. (10 points)

Definition 1.1 ($\prec$ 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

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

Funcitons can be ordered by $\prec$ with its definition: $f \prec g$ iff $f(n) = o(g(n))$. For example,

$$1 \prec \log n \prec n$$

Sort $(\sqrt{2})^{\log n}, (n+2)!, n^n, 0.2n^{\sqrt{2}}, n^{2/\log n}$ with $\prec$, briefly explain your answer.

Solution: $(n^{2/\log n}) \prec (\sqrt{2})^{\log n} \prec 0.2n^{\sqrt{2}} \prec (n+2)! \prec n^n$

- $(\sqrt{2})^{\log n} = o\left((n+2)!\right)$, $c = 1, n_0 = 1$

- $(n+2)! = o(n^n)$, $c = 1, n_0 = 6$
- $(\sqrt{2})^{\log n} = o(0.2n^{\sqrt{2}})$, since $n^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\cdot\log_{\sqrt{2}} n} = (\sqrt{2}^C)^{\log n}$, $C > 1$
- $0.2n^{\sqrt{2}} = o\left((n+2)!\right)$, $c = 1, n_0 = 1$
- $n^{2/\log n} = o((\sqrt{2})^{\log n})$, $c = 1, n_0 = 10^{16}$.

2 Master Theorem (20 points)

2.1 Recurrence Relation (10 points)

What is the complexity of the following recurrence relation? (5 points each)

(a) $T(n) = \begin{cases} c_0, & n = 1 \\ 8T\left(\frac{n}{2}\right) + (n+1)(n+2) + c, & n > 1 \end{cases}$

Solution: $\Theta(n^3)$

(b) $T(n) = \begin{cases} c_0, & n = 1 \\ 2T\left(\frac{n}{4}\right) + 2\log n + \sqrt{n} + c, & n > 1 \end{cases}$

Solution: $\Theta(\sqrt{n}\log n)$

2.2 Master Theorem on code (10 points)

(a) Derive the recurrence relation of the TigerNacci function in Problem 1c, then find out its time complexity. (Hint: Can you use master theorem here?) (5 points)

Solution: $\Theta(1.84^n)$, see the procedure above.

(b) Based on the function below, answer the following question. **Assume that** $helper(n)$ **runs in** $O(n^6)$ **time.** (5 points)

```
void problem_2_2_b(int n) {
  if (n < 3) {
    return;
  }
  int m = static_cast<int>(floor(sqrt(n)))
  for (int i = 0; i < 281; ++i) {
    helper(m);
    for (int j = m; j < i; j++) {
      helper(m);
    }
```

```
11      problem_2_2_b(n / 3);
12    }
13  }
```

Calculate the recurrence relation of this function, then find out its time complexity.

Solution: $\Theta()$
Suppose the time complexity is $T(n)$, then due to the recurrence relation

$$T(n) = 281 \cdot T\left(\frac{n}{3}\right) + 281n^6 + C.$$

We find that

$$n^{\log_3 281} = n^{5.13} \quad \Rightarrow f(n) = 281n^6 = \Omega(n^{\log_3 281+\varepsilon}),$$

Therefore,

$$T(n) = \Theta(n^6).$$

3 Sorting Basics (22 points)

3.1 Sorting algorithms' working scenarios (16 points)

Select the most efficient sorting algorithm for each of the following situations. (4 points each)

(a) You are sorting a set of electronic files. However, they are huge and you found that copying and moving these files are too expensive. So you want to copy and move them as less as possible.

A) insertion sort
B) selection sort
C) quick sort
D) merge sort

Solution: B)

(b) Your friend is transmitting a sorted array of integers to you. However, due to some interference, a few integers in the array have changed their position. This causes the sorted array to become a nearly sorted one.

A) insertion sort
B) selection sort
C) quick sort
D) merge sort

Solution: A)

(c) You are dealing with a sequence of integers that are stored in a linked list. This means that it is expensive for you to access integer in a specific position.

A) insertion sort
B) selection sort
C) quick sort
D) merge sort

Solution: A)

(d) You are a game programmer in the 1980s. You need to display a relatively small set of the names of defeated enemies in a sorted order as quickly as possible. Since it is old time, the players are used to occasional long time waiting before the display.

A) insertion sort
B) selection sort
C) quick sort
D) merge sort

Solution: A)

3.2 Brute force or sorting? (6 points)

You need to find 100 richest persons from an unsorted list containing wealth of 10000 different persons. Two potential solution are as follows.

i) repeat linear search for 100 times. Each time you need to find the richest person.

ii) convert the list into an array, and then sort the array(complexity O(n logn)) and fetch data from the sorted array.

Which solution would you suggest? Assume the time of linear search for 10 items in an unsorted list takes 10 ns and the time for sorting 10 items takes 10 ns. You can neglect the time to convert the list into an array and the time to fetch data from the sorted array.

Solution: I would suggest to perform a sorting algorithm first.

- Linear search for 100 times would take $100 \times 10000 = 0.01[s]$.
- For sorting algorithm, since $C \cdot 10 \times \log 10 = 10[ns]$, for 10000 terms the estimated time is $C \cdot 10000 \times \log 10000 = 40000[ns]$.

4 Improved Insertion Sort (17 points)

4.1 Binary Search (9 points)

After Ssy finishes studying sorting algorithm in VE281, he finds himself quite interested about the insertion sort. He quickly implements this sorting algorithm and runs several cases to test his code. However, he finds the performance of this algorithm is not satisfying. Then he decides to improve this insertion sorting algorithm by using binary search instead of linear search. The pseudo code of this algorithm is shown below.

Algorithm 1 ImprovedInsertionSort(a[.])

Input: an integer array a with size n
Output: the sorted array of a

```
1: for i = 1; i < n; i += 1 do
2:     temp = a[i];
3:     p = BinarySearch(a, i-1, temp);
4:     for j = i-1; j ≥ p; j -= 1 do
5:         a[j+1] = a[j];
6:     end for
7:     a[j] = temp;
8: end for
```

Based on the algorithm above, answer the questions. (3 points each)

a) How many data moves needed to be done to sort an array of size n in average? What about the worst case?

b) How many comparisons needed to be performed to sort an array of size n in average? What about the worst case?

c) What's time complexity of the binary search part and data moving part, respectively? Is this algorithm better than the one with linear search?

Solution:

a) average case: $n(n-1)/2$, worst case: $n(n+1)$.

b) average case: $\log n!$, worst case: $n \log n$.

c) search part : $O(n \log n)$, moving part: $O(n^2)$. Therefore this algorithm is a little bit better than the original one, as the time complexity in the searching part decreases. But, the overall time complexity is still $O(n^2)$.

4.2 Improve by Splitting (8 points)

After discussing with Wcy, Ssy discovers that insertion sort works fast for small size array. However, when the size n goes large, insertion sort may not be a good choice since its worst case time complexity is $O(n^2)$. He comes up with a new idea that he can split a large size array into small ones, sort them and merge them. The proposed way to improve Insertion sorting is

i) Split the unsorted array into k subarrays, each with size $\frac{n}{k}$. The last one may have smaller size to maintain the total size n.

ii) Sort each subarrays by insertion sort.

iii) Merge the subarrays into one entire sorted array. This procedure will work as: select the largest integer from k subarrays, then select the second largest integer from k subarrays, etc.

Analyze the algorithm described above, find the optimal k for given n and time complexity for this algorithm.

Solution: We calculate the time complexity for each part:

i) Spilt: constant C

ii) Sort each part: $k \cdot O(n^2/k^2) = O(n^2/k)$.

iii) Merge part: $n \cdot k$

Therefore $T(n) = C_1 + C_2 \cdot n^2/k + n \cdot k$, when $k = \lfloor \sqrt{C_2 \cdot n} \rfloor$ it reaches minimum. Namely, $T(n) = O(n\sqrt{n})$.

5 Sorting Algorithm Design (13 points)

Suppose you are designing a aggregator, you will be given two unsorted arrays of lengths m and n. You need to design an algorithm to find the intersection of these two arrays. Assume there is no duplicates within each array.

a) Describe an algorithm with time complexity $O(mlogm + nlogn)$

b) Describe an algorithm with time complexity $O(min(mLogm + nLogm, mLogn + nLogn))$

c) For the two algorithms above, do you think one of them is always better than the other? If not, when would you use the algorithm in b)?

Solution:

a) First apply quick sort to the two arrays respectively, the time complexity is $O(m \log m)$ and $O(n \log n)$. Then run a linear scan to merge them and the time complexity is $O(m + n)$. So the overall time complexity is $O(m \log m + n \log n)$.

b) First run quick sort for the first array, and the time complexity is $O(m \log m)$. Then for every element in the second array, perform a binary search in the first sorted array, and the total time complexity is $O(n \log m)$. In total, the time complexity is $O((m + n) \log m)$.

We can also choose to perform quick sort for the second array and binary search for the first array, then the time complexity would be $O((m + n) \log n)$. We can choose a smaller one, namely reach a complexity $O(\min(m \log m + n \log m, m \log n + n \log n))$.

c) It seems that the time complexity of the second algorithm is always smaller than the first one. So use b) in any case.