1 Complexity Analysis (20 points, after Lec2)

(a) Based on the given code, answer the following questions:(4 points)

```cpp
void question_1a(int n) {
   int count = 0;
   for (int i = 0; i < n; i++) {
    for (int j = i; j > 0; j--) {
      count += 1;
    }
   }
  cout << count << endl;
}
```

i) What is the output? Describe the answer with variable n. (1 points)

ii) What is the time complexity of the following function? What do you find when comparing it with your answer of the previous part i)? (3 points)

Solution:

i) the value of *count* is related to the number of iterations, which can be calculated as:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{1} 1 = \sum_{i=0}^{n-1} i = \frac{(0+n-1)*n}{2} = \frac{1}{2}(n^2-n) \quad \text{(1 point)}$$

ii) The time complexity of the given function is $O(n^2)$, (2 points)

and we found that the answer we get in previous part $\frac{1}{2}(n^2-n) = O(n^2)$ (1 point)

(b) What is the time complexity of the following function? (4 points)

```cpp
void question_1b(int N, int M, int K) {
  int count = 0;
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < M / 2; j++) {
      count++;
    }
  }
  for (int i = 0; i < K; i++) {
    count--;
  }
}
```

Solution: For the first part of loops with i and j, the time complexity is:

$$T_1(n) = N/2 * M/2 = NM/4 = O(NM) \quad \text{(1 point)}$$

For the second part of loop, the time complexity is:

$$T_2(n) = K = O(K) \quad \text{(1 point)}$$

Since we do not know the relationship between NM and K, we can get the overall time complexity as:

$$T(n) = O(NM) + O(K) \text{ or } O(NM + K) \quad \text{(2 points)}$$

(c) What is the time complexity of the following function? Select **All** the answers that are correct, and state your reason. (4 points)

```cpp
void question_1c(int n) {
  int count = 0;
  int m = static_cast<int>(floor(sqrt(n)));
  for (int i = n/2; i < n; i++) {
    for (int j = 1; j < n; j = 2*j) {
      for (int k = 0; k < n; k += m) {
        count++;
      }
    }
  }
}
```

i) $\Theta(n^{1/2} \log n)$
ii) $\Theta(n \log n)$
iii) $O(n \log n)$
iv) $\Theta(n^{3/2} \log n)$
v) $\Theta(n^2 \log n)$
vi) $O(n^{5/2} \log n)$
vii) $\Theta(n^{5/2} \log n)$

Solution: From line 3, we know that $m = \lfloor \sqrt{n} \rfloor \approx= \sqrt{n}$.
For the inner most loop (with iterator k), we have:

$$T_k(n) = \Theta(n/m) = \Theta(\sqrt{n}) \quad \text{(1 point)}$$

For the second loop (with iterator j), we have:

$$T_j(n) = T_k(n) \cdot \log_2(n) = \Theta(\log n \cdot \sqrt{n}) \quad \text{(1 point)}$$

For the outer most loop (with iterator i), we have:

$$T(n) = T_i(n) = (n/2 - 1) \cdot T_j(n) = \Theta(n^{3/2} \log n) \quad \text{(1 point)}$$

Therefore, *iv)* and *vi)* are correct. (1 point)

(d) What is the time complexity of the following function? Show your steps. (4 points)

```
int unknown_function(int n) {
  if (n <= 1) return 1;
  return n * (unknown_function(n-1));
}
```

Solution: If you simulate the recursion, you can find that: For $n = n_0$, where n_0 is the initial input, after $O(1)$ operation, it will go to $n = n_0 - 1$; Then for $n = n_0 - 1$, it will go to $n = n_0 - 2$ after $O(1)$ operation...(2 points)

Finally it will run n_0 times of $O(1)$ operation and meet the end. Hence, the time complexity is $O(n)$. (2 points)

(e) Consider the following four statements regarding algorithm complexities:

i) an algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n \log n)$ time complexity.

ii) an algorithm with a $\Theta(n \log n)$ time complexity will always run faster than an algorithm with a $\Theta(n^2)$ time complexity.

iii) an algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n!)$ time complexity.

iv) an algorithm with a $\Theta(n!)$ time complexity will always run faster than an algorithm with a $\Theta(n^2)$ time complexity.

How many of these statements are true? Show your reasons. (4 points)

Solution: The speed for an algorithm differs with input size and constant time consumptions. For example, for $f(n) = 10n^2$, $g(n) = n \log n$, and take $n = 1$, then $g(n)$ will run faster than $f(n)$; When $n = 100$, then $g(n)$ runs slower than $f(n)$.

Hence, none of these statements are true.

For grading, each statement counts 1 point.

2 Master Theorem (15 points, after Lec3)

2.1 Recurrence Relation (9 points)

What is the complexity of the following recurrence relation? (if not mentioned, please state it with big-theta notation.)

(a) $T(n) = \begin{cases} c_0, & n = 1 \\ 4T\left(\frac{n}{2}\right) + 16n + n^2 + c, & n > 1 \end{cases}$

Solution: The other operations are in $O(n^2)$ for $n > 1$.

Hence, We have

$$a = 4,\ b = d = 2\ \Rightarrow a = b^d = 4 \quad \text{(2 points)}$$

Also, base station satisfies the requirements for the master theorem, hence by master theorem, we have:

$$T(n) = O(n^2 \log n) \quad \text{(1 point)}$$

(b) $T(n) = \begin{cases} c_0, & n = 1 \\ 5T\left(\frac{n}{25}\right) + \sqrt{n} + c, & n > 1 \end{cases}$

Solution: The other operations are in $O(\sqrt{n})$ for $n > 1$.

Hence, We have

$$a = 5,\ b = 25,\ d = 0.5\ \Rightarrow a = b^d = 5 \quad \text{(2 points)}$$

Also, base station satisfies the requirements for the master theorem, hence by master theorem, we have:

$$T(n) = O(\sqrt{n} \log n) \quad \text{(1 point)}$$

(c) $T(n) = \begin{cases} c_0, & n = 1 \\ 3T(n-1) + c, & n > 1 \end{cases}$ (Hint: Can you still use master theorem here?)

Solution: Here we cannot use master theorem, since we cannot find the input size shrinkage factor b in the given recurrence relation. (1 point)

Therefore, we need to apply normal approach to solve out the time complexity:

$$T(n) = 3T(n-1) + c_1 = 3^2T(n-2) + c_2 = 3^3T(n-3) + c_3 = \cdots = 3^{n-1}c_0 + c_n$$

Hence, we have $T(n) = O(3^n)$ (2 points)

2.2 Master Theorem on code (6 points)

Based on the function below, answer the following question. **Assume that** $cake(n)$ **runs in** $\log n$ **time.**

```
void pie(int n) {
  if (n == 1) {
    return;
  }
  pie(n / 7);
  int cookie = n * n;
  for (int i = 0; i < cookie; ++i) {
    for (int j = 0; j < n; ++j) {
      cake(n);
    }
  }
  for (int k = 0; k < n; ++k) {
    pie(n / 3);
  }
  cake(cookie * cookie);
}
```

Calculate the recurrence relation of this function.

Solution: We can divide the given function line by line and calculate their partial time complexity:

- Line 2-4: $T_1(n) = O(1)$
- Line 5: $T_2(n) = T(\frac{n}{7})$ (1 point)
- Line 6-11: $T_3(n) = O(n^2 \cdot n\log(n)) = O(n^3 \log n)$ (1 point)
- Line 12-14: $T_4(n) = nT(\frac{n}{3})$ (1 point)
- Line 15: $T_5(n) = O(\log(n^2 \cdots n^2)) = O(4\log(n)) = O(\log(n))$ (1 point)

Then, we need to sum up all the partial time complexity to get the overall time complexity:

$$T(n) = T(\frac{n}{7}) + O(n^3 \log n) + nT(\frac{n}{3}) + O(4\log(n)) = T(\frac{n}{7}) + O(n^3 \log n) + nT(\frac{n}{3})$$

Finally, we have:

$$T(n) = \begin{cases} O(1) & (n = 1) \quad \text{(1 point)} \\ T(\frac{n}{7}) + O(n^3 \log n) + nT(\frac{n}{3}) & (n > 1) \quad \text{(1 point)} \end{cases}$$

3 Sorting Algorithms (45 points, after Lec4)

3.1 Sorting Basics (12 points)

3.1.1 Sorting algorithms' working scenarios (6 points)

What is the most efficient sorting algorithm for each of the following situations?

(a) A small array of integers.

- A) insertion sort
- B) selection sort
- C) quick sort
- D) bucket sort

Solution: A
Insertion sort has low overhead, compared to quick sort and merge sort, and has limited steps of swaps and no recursions, which enables its high speed for small array of integers.

(b) A large array of integers that is already almost sorted.

- A) insertion sort
- B) selection sort
- C) quick sort
- D) bucket sort

Solution: A
Insertion sort is adaptive for almost sorted arrays.

(c) A large collection of integers that are drawn from a very small range.

- A) insertion sort
- B) selection sort
- C) quick sort
- D) bucket sort

Solution: D
For such a special case, it is not efficient to use comparison sort here. Instead, a non-comparison sort can provide a sorting algorithm with O(n) time complexity. Also, bucket sort behaves better than counting sort.

3.1.2 Sorting snapshots (6 points)

(a) Suppose you had the following unsorted array:

$$\{22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47\}$$

A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

$$\{9, 13, 22, 52, 66, 74, 28, 59, 71, 11, 35, 47\}$$

which of the following sorts is currently being run on this array?

A) bubble sort
B) insertion sort
C) selection sort
D) quick sort
E) merge sort
F) none of above

Solution: E

(b) Suppose you had the following unsorted array:

$$\{22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47\}$$

A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

$$\{9, 11, 13, 22, 28, 74, 66, 59, 71, 35, 52, 47\}$$

which of the following sorts is currently being run on this array?

A) bubble sort
B) insertion sort
C) selection sort
D) quick sort
E) merge sort
F) none of above

Solution: C/D

(c) Suppose you had the following unsorted array:

$$\{22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47\}$$

A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

$$\{22, 9, 13, 11, 35, 28, 47, 59, 71, 66, 52, 74\}$$

which of the following sorts is currently being run on this array?

A) bubble sort
B) insertion sort
C) selection sort
D) quick sort
E) merge sort
F) none of above

Solution: F

3.2 Squares of a Sorted Array (17 points)

Sxt is now doing his homework0 for VE281. He is given an integer array A sorted in **non-decreasing order (non-positive numbers included)**, and is required to return an array of the **squares** of each number sorted in non-decreasing order. It is guaranteed that there is **always one zero**(0) in the given array.
For example, given an array: $\{-4, -2, 0, 1, 3, 7\}$, Sxt needs to return $\{0, 1, 4, 9, 16, 49\}$.
Here is Sxt's code:

```cpp
#include <iostream>
using namespace std;

// REQUIRES: an array A and its size n
// EFFECTS: sort array A
// MODIFIES: array A
void insertion_sort(int *A, size_t size) {
    for (size_t i = 1; i < size; i++) {
        size_t j = 0;
        while (j < i && A[i] >= A[j]) {
            j++; // Find the location to insert the value
        }
        int tmp = A[i]; // Store the value we need to insert
        for (size_t k = i; k > j; k--) {
            A[k] = A[k-1];
        }
        A[j] = tmp;
    }
}

int main(){
    int A[5] = {-4, -1, 0, 3, 10};
    size_t sizeA = 5;
    for (size_t i = 0; i < sizeA; i++) { //Calculate the square of input array
        A[i] = A[i] * A[i];
    }
    insertion_sort(A, sizeA);
    for (auto item: A) {
        cout << item << ' ';
    }
    cout << endl;
    return 0;
}
```

However, he finds his code runs slowly when it encounters array with great length. Also, he guesses some operations maybe useless in his code because of the **special property of the input array**. Hence, he hopes that you can help him find out where he can improve his code to have $O(n)$ **time complexity**. You can modify this code however you like and briefly state your reason for why you modify it in that way and can satisfy the required time complexity.
You can find the code above in the given starter file *square.cpp*. Please modify the code above in *square.cpp* file, and upload the file together to canvas(DO NOT CHANGE THE FILE NAME!!). You can find more details for homework submission at the beginning of this homework.

Solution: (State your reason here) The official solution should be: As you see that we can always separate the given array into two parts: negative part and nonnegative part. After taking the square of the given array, we can see that the original negative part keeps a decreasing order, while the non-negative part keeps an ascending order. Hence, we need to first reverse the former one, and get two ascending subarrays. Next we need to merge the two sorted array with merge function which we use in merge sort. Since we only need to do the merge once, the time complexity is O(n).

*Some students choose to use counting sort for this question. For this question, since we only give restriction on time complexity, after long discussion, we choose to give them full points. However, it is really inefficient to use non-comparison sort in such situation, and we will never recommend you to use counting sort in this case.

The following is one of the TA's code

```
#include <iostream>
#include <vector>
#include <sstream>
#include <string>
using namespace std;

// REQUIRES: two vectors v1, v2
// EFFECTS: merge them into one
// MODIFIES: None
vector<int> merge(vector<int> &v1, vector<int> &v2)
{
    vector<int> res(v1.size() + v2.size());
    size_t i, j, k;
    i = j = k = 0;
    while (i < v1.size() && j < v2.size())
    {
        if (v1[i] <= v2[j])
            res[k++] = v1[i++];
        else
            res[k++] = v2[j++];
    }
    res.erase(res.begin() + k, res.end());
    if (i == v1.size())
        res.insert(res.end(), v2.begin() + j, v2.end());
    else
        res.insert(res.end(), v1.begin() + i, v1.end());
    return res;
}

// EFFECTS: swap the value of a and b
// MODIFIES: a, b
void swap(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
}

// EFFECTS: reverse the vector v
// MODIFIES: v
void reverse(vector<int> &v)
{
    size_t len = v.size();
    for (size_t i = 0; i < len; i++)
    {
        if (i >= len - 1 - i)
            return;
        else
            swap(v[i], v[len - 1 - i]);
    }
}

// EFFECTS: implement the function fp to all the elements in vector v
// MODIFIES: v
void map(vector<int> &v, int (*fp)(int a))
{
    size_t len = v.size();
    for (size_t i = 0; i < len; i++)
        v[i] = fp(v[i]);
}

// REQUIRES: vector A
// EFFECTS: do the square sort asked by the question
vector<int> square_sort(vector<int> &A)
{
    // the main process
    size_t zero_pos = 0;
    while (zero_pos < A.size())
        if (A[zero_pos++] == 0)
            break;
    vector<int> v1(A.begin() + zero_pos, end(A));
    vector<int> v2(A.begin(), A.begin() + zero_pos);

    reverse(v2);
    map(v2, [](int x) -> int
        { return -x; });

    // Merge the two vectors together into a sorted one (idea comes from merge sort)
    // Then, we square them all
    vector<int> res = merge(v1, v2);
```

```cpp
    map(res, [](int x) -> int
        { return x * x; });
    return res;
}

// MAIN
int main()
{
    vector<int> A;
    string s;
    getline(cin, s);

    stringstream ss(s);
    int temp;
    while (ss >> temp)
        A.emplace_back(temp);

    vector<int> res = square_sort(A);

    // print res
    for (auto item : res)
        cout << item << " ";
    cout << "\n";
    return 0;
}
```

3.3 Tim Sort (12 points)

We want to find the 6th largest element, which is 6, in the following array, Insertion sort is a simple and fast 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.

Algorithm 1 timSort(a[.],x)

Input: an array a of n elements, and 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: end for
 4: for step = x; step < n; step *= 2 do
 5:     for left = 0; left < n; left += 2 * step do
 6:         mid = left + step - 1;
 7:         right = min(left + 2*step - 1, n - 1);
 8:         merge(a, left, mid, right);
 9:     end for
10: end for
```

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

(a) Suppose n = 1000 and x = 32. How many times will insertionSort be performed?

(b) Suppose x = 32. Express in terms of n how many comparisons in the worst case will be performed in insetionSort.

(c) Express the worst case running time of the whole algorithm in terms of big-Oh notation.

(d) Is this algorithm in-place? If not, express the additional space needed in terms of the big-Oh notation.

Solution. 1. We need

$$\left\lceil \frac{1000}{32} \right\rceil = 32$$

times of `insertionSort` in total.

2. In the worst case, we need $n(n-1)/2$ comparisons for an array of length n. In this case we have $\lfloor n/32 \rfloor$ arrays of length 5 and an array of length $n \bmod 32$. So in total we will need

$$496 \times \lfloor n/32 \rfloor + \frac{(n \bmod 32)(n \bmod 32 - 1)}{2} = O(n)$$

1

total times of comparisons.

3. The first `for` loop has time complexity $O(n)$. In the second outer `for` loop (line 3) will execute $\lceil \log_2 n \rceil = O(\log n)$ times while the inner `for` loop (line 4) will execute $\lceil n/(2 \times \text{step}) \rceil = O(n)$ times each time we reach the inner loop. So in total the running time is of

$$O(n) + O(\log n) \times O(n) = O(n \log n).$$

4. This algorithm is not in-place because the `merge` function needs $O(n)$ additional space.

3.4 Quicker sort simulation (16 points)

To fully understand the mechanism of sorting algorithm, please simulate the given array for each iteration of required algorithm.

3.4.1 Quick sort (8 points)

Assume that we always choose the **first entry** as the pivot to do the partition, and we want to sort the array in **ascending order**. Then, for the following array:

$$A = \{6, 2, 8, 10, 3, 1, 9\}$$

Sxt shares his answer for this array:

Solution:

Iter	Current Subarray	Pivot	Swapped Subarray	Current Array
1	$\{6, 2, 8, 10, 3, 1, 9\}$	6	$\{3, 2, 1, 6, 10, 8, 9\}$	$\{3, 2, 1, 6, 10, 8, 9\}$
2	$\{3, 2, 1\}$	3	$\{1, 2, 3\}$	$\{1, 2, 3, 6, 10, 8, 9\}$
3	$\{1, 2\}$	1	$\{1, 2\}$	$\{1, 2, 3, 6, 10, 8, 9\}$
4	$\{\}$	None	$\{\}$	$\{1, 2, 3, 6, 10, 8, 9\}$
5	$\{2\}$	None	$\{2\}$	$\{1, 2, 3, 6, 10, 8, 9\}$
6	$\{\}$	None	$\{\}$	$\{1, 2, 3, 6, 10, 8, 9\}$
7	$\{10, 8, 9\}$	10	$\{9, 8, 10\}$	$\{1, 2, 3, 6, 9, 8, 10\}$
8	$\{9, 8\}$	9	$\{8, 9\}$	$\{1, 2, 3, 6, 8, 9, 10\}$
9	$\{8\}$	None	$\{8\}$	$\{1, 2, 3, 6, 8, 9, 10\}$
10	$\{\}$	None	$\{\}$	$\{1, 2, 3, 6, 8, 9, 10\}$
11	$\{\}$	None	$\{\}$	$\{1, 2, 3, 6, 8, 9, 10\}$

*Brief explanation: You need to strictly follow the algorithm we learned in class as the following (since there are so many different kinds of quick sort with slight changes).

```
void quicksort(int *a, int left, int right) {
  int pivotat; // index of the pivot
  if(left >= right) return;
  pivotat = partition(a, left, right);
  quicksort(a, left, pivotat-1);
  quicksort(a, pivotat+1, right);
}
```

The steps above strictly follows the recursion order. For example, for iter 4, this is the left half part of iter 3, while iter 5 is the right half part of iter 3.

Now please simulate quick sort for the following array:

$$A = \{6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3\}$$

You should follow the format that Sxt has shared.

Solution:

Iter	Current Subarray	Pivot	Swapped Subarray	Current Array
1	$\{6,2,8,5,11,10,4,1,9,7,3\}$	6	$\{4,2,3,5,1,6,10,11,9,7,8\}$	$\{4,2,3,5,1,6,10,11,9,7,8\}$
2	$\{4,2,3,5,1\}$	4	$\{1,2,3,4,5\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
3	$\{1,2,3\}$	1	$\{1,2,3\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
4	$\{\}$	None	$\{\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
5	$\{2,3\}$	2	$\{2,3\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
6	$\{\}$	None	$\{\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
7	$\{3\}$	None	$\{3\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
8	$\{5\}$	None	$\{5\}$	$\{1,2,3,4,5,6,10,11,9,7,8\}$
9	$\{10,11,9,7,8\}$	10	$\{7,8,9,10,11\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
10	$\{7,8,9\}$	7	$\{7,8,9\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
11	$\{\}$	None	$\{\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
12	$\{8,9\}$	8	$\{8,9\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
13	$\{\}$	None	$\{\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
14	$\{9\}$	None	$\{9\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$
15	$\{11\}$	None	$\{11\}$	$\{1,2,3,4,5,6,7,8,9,10,11\}$

Grading: Mistake(s) in one iter counts for 1 point.

3.4.2 Merge Sort (8 points)

For the following array:

$$A = \{6,2,8,10,3,1,7\}$$

Sxt shares part of his answer for applying merge sort to the given array:

Solution:
1. Division: $\{6,2,8,10\}$ $\{3,1,7\}$
2. Division: $\{6,2\}$ $\{8,10\}$ $\{3,1,7\}$
3. Division: $\{6\}$ $\{2\}$ $\{8,10\}$ $\{3,1,7\}$
4. Merge: $\{2,6\}$ $\{8,10\}$ $\{3,1,7\}$
5. Division / Merge: ...

...

Last. Merge: $\{1,2,3,6,7,8,10\}$

Now please simulate merge sort for the following array:

$$A = \{6,2,8,5,11,10,4,1,9,7,3\}$$

Please show all the details of each division or merge.

Solution:

1. Division: $\{6,2,8,5,11,10\}$ $\{4,1,9,7,3\}$

2. Division: $\{6,2,8\}$ $\{5,11,10\}$ $\{4,1,9,7,3\}$

3. Division: $\{6, 2\}$ $\{8\}$ $\{5, 11, 10\}$ $\{4, 1, 9, 7, 3\}$
4. Division: $\{6\}$ $\{2\}$ $\{8\}$ $\{5, 11, 10\}$ $\{4, 1, 9, 7, 3\}$
5. Merge: $\{2, 6\}$ $\{8\}$ $\{5, 11, 10\}$ $\{4, 1, 9, 7, 3\}$
6. Merge: $\{2, 6, 8\}$ $\{5, 11, 10\}$ $\{4, 1, 9, 7, 3\}$
7. Division: $\{2, 6, 8\}$ $\{5, 11\}$ $\{10\}$ $\{4, 1, 9, 7, 3\}$
8. Division: $\{2, 6, 8\}$ $\{5\}$ $\{11\}$ $\{10\}$ $\{4, 1, 9, 7, 3\}$
9. Merge: $\{2, 6, 8\}$ $\{5, 11\}$ $\{10\}$ $\{4, 1, 9, 7, 3\}$
10. Merge: $\{2, 6, 8\}$ $\{5, 10, 11\}$ $\{4, 1, 9, 7, 3\}$
11. Merge: $\{2, 5, 6, 8, 10, 11\}$ $\{4, 1, 9, 7, 3\}$
12. Division: $\{2, 5, 6, 8, 10, 11\}$ $\{4, 1, 9\}$ $\{7, 3\}$
13. Division: $\{2, 5, 6, 8, 10, 11\}$ $\{4, 1\}$ $\{9\}$ $\{7, 3\}$
14. Division: $\{2, 5, 6, 8, 10, 11\}$ $\{4\}$ $\{1\}$ $\{9\}$ $\{7, 3\}$
15. Merge: $\{2, 5, 6, 8, 10, 11\}$ $\{1, 4\}$ $\{9\}$ $\{7, 3\}$
16. Merge: $\{2, 5, 6, 8, 10, 11\}$ $\{1, 4, 9\}$ $\{7, 3\}$
17. Division: $\{2, 5, 6, 8, 10, 11\}$ $\{1, 4, 9\}$ $\{7\}$ $\{3\}$
18. Merge: $\{2, 5, 6, 8, 10, 11\}$ $\{1, 4, 9\}$ $\{3, 7\}$
19. Merge: $\{2, 5, 6, 8, 10, 11\}$ $\{1, 3, 4, 7, 9\}$
20. Merge: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$

Grading: Mistake(s) in one line counts for 1 point.

4 Selection Algorithm (20 points, after Lec5)

(a) Coned is an undergraduate student taking VE281. He is interested in **random** selection algorithm and wondering which scenario will be the **worst-case** one.

- For the following input sequence, can you give out the pivot sequence for the worst-case scenario? Suppose that we are finding the 3-rd biggest element. (6 points)
  Input sequence:
  $$37, 7, 31, 27, 22, 13, 46$$
  Solution format for the pivot sequence:
  (Certainly the pivot selection is not correct for the requirement :) It is only for showing the format.)

  Solution:
  Pick 22: 7,13,22,37,31,27,46
  Pick 46: 7,13,22,37,31,27,46
  Pick 37: 7,13,22,31,27,37,46
  ...

- What is the time complexity of the worst-case scenario? (2 points)

(b) William is a master of algorithm. He thinks that Coned's selection algorithm is too naive and prefers **deterministic** selection algorithm. However, since his favorite number is 7, he **partitions the numbers by groups of 7 rather than 5**. He is confident that such a change will make no difference to the time complexity of the algorithm. Is him right? Namely, will the time complexity of William's new selection algorithm **still be** $O(n)$, where n is the amount of input numbers? Give your proof. (8 points)

(c) When we consider the time complexity of an algorithm, we usually ignore the constant coefficients since we care about large n. Similarly, if we only consider the large cases, which algorithm do you think may work better, deterministic or random? State your reason in detail. (4 points) (Hint: think about William's change and what professor said in the lecture before you have your answer.)

Solution:

(a)
- Pick 7: 7,37,31,27,22,13,46
  Pick 13: 7,13,37,31,27,22,46
  Pick 22: 7,13,22,37,31,27,46
  Pick 27: 7,13,22,27,37,31,46
  Pick 46: 7,13,22,27,37,31,46
  Pick 31(or 37): 7,13,22,27,31,37,46
  Answer not exclusive. For each correct step, 1 point is given. If pivot selection is right but the number sequence is wrong, 0.5 point is deducted. Last two steps worth 1 point altogether. If 7 steps, 0.5 point is deducted.
- $O(n^2)$

(b) He is right. We have $k = n/7$ and at least $\frac{4}{7} * \frac{1}{2} = \frac{2}{7}$ elements larger/smaller than the pivot. (2 points)

$$T(n) \leq cn + T(\frac{n}{7}) + T(\frac{5}{7}n) \quad \text{(2 points)}$$

Check whether there is an a such that $T(n) \leq an$. Let $a = 7c$.
proof by induction:

$$T(1) \leq 7c$$

Suppose that for $n < k, T(n) \leq 7cn$ and then for $n = k$,

$$T(k) \leq ck + T(\frac{k}{7}) + T(\frac{5}{7}k) \leq ck + ck + 5ck = 7ck \quad \text{(3 points)}$$

Therefore we have $T(n) \leq 7cn = O(n)$. (1 point)

(c) Random side:
If we only consider the large cases, the problem of the deterministic one is that it is not in-place and the extra space complexity will be $O(n)$ for the medians. So the random selection algorithm may work better if we only consider those super large cases.
Also, the deterministic one is easier to be attacked.
Deterministic side:
The characteristics of deterministic selection is that for the worst case, while the time complexity of random selection is $O(n^2)$, deterministic selection also remains $O(n)$. It will be more significant if we only consider super large cases.
Notes: Reasonable explanation, then no deduction. No explanation, then no points.