1 Comparison Sort (High-level)

Elementary sort(Bubble sort, Insertion sort, Selection sort) can only have $O(n^2)$ complexity in average case. Only when we have the input with special properties such as: small array size / nearly sorted / completely sorted, can we use elementary sort for convenience.
However, in most of the cases, we will encounter arrays with huge size. And an $O(n^2)$ can no longer satisfy our requirements on speed. That's why we introduce two high-level sorting algorithm with $O(n \log n)$ average time complexity: quick sort and merge sort.

1.1 Master Theorem

Master Theorem in VE281 is mainly used for calculating the time complexity of recursive function. You must have learned master theorem in VE203 before. Hence I will not put too much emphasis on this part. However, calculation of master theorem is still very important in VE281. And you need to fully understand how to apply master theorem to calculate the time complexity of given function.
If you want to apply master theorem, it must satisfy the following format:

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$$

- Base case: $T(n) \leq$ *constant* for all sufficiently small n.
- $a =$ number of recursive calls (**integer** ≥ 1).
- $b =$ input size shrinkage factor (**integer** > 1).
- $O(n^d) =$ the runtime of other operations in one recursion. d is a **real** value.
- a, b, d are independent of n.

Only when your recurrence relation satisfies the requirements above can you use the following formula:

$$T(n) = \begin{cases} O(n^d \log n) & if\ a = b^d \\ O(n^d) & if\ a < b^d \\ O(n^{\log_b a}) & if\ a > b^d \end{cases}$$

For example, the following function implements the binary search on a ascending array.

```
int binary_search(int * A, size_t left, size_t right, int target) {
  mid = (left+right) / 2;
  if (A[mid] == target) return mid;
  if (A[mid] < target) return binary_search(A, mid+1, right, target);
  return binary_search(A, left, mid, target);
}
```

To compute the time complexity of the given function,

> **Solution:** You can find that for each iteration, we first calculate the *mid*, which takes $O(1)$ time complexity. And then we choose either to return a value, or go into **one** new recursion loop. And every time we do the recursion, the length of array that we are looking at shrinks by the factor of 2.

Hence, we have: $a = 1,\ b = 2\ d = 0$.
And also

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(n^0) = T\left(\frac{n}{2}\right) + O(1)$$

since $a = b^d$, we finally have $T(n) = O(\log n)$.

1.2 Merge Sort

Merge sort first divides the given input into two sub arrays with nearly the same length, and then recursively does the division until there is only one or no elements left. Next we merge those sub arrays, with the merged array sorted. Finally we can get the sorted array.
There are many different versions of implementation of merge sort. Hence please stick to the algorithm we learned in this course! All the assignments and exams are based on the algorithm that we provide in this course.

```
1  void merge_helper(int *A, size_t sizeA, int *B, size_t sizeB, int *C) {
2      for (size_t i = 0, j = 0, k = 0; k < sizeA + sizeB; ++k) {
3          if (i == sizeA) {
4              C[k] = B[j++];
5          }else if (j == sizeB) {
6              C[k] = A[i++];
7          }else{
8              C[k] = (A[i] <= B[j]) ? A[i++] : B[j++];
9          }
10     }
11 }
12
13 // EFFECTS: merge two sorted arrays
14 void merge(int *A, size_t left, size_t mid, size_t right) {
15     size_t sizeA = mid - left + 1;
16     size_t sizeB = right - mid;
17     int *tmpA = new int[sizeA], *tmpB = new int[sizeB];
18     for (size_t i = left; i <= mid; i++) {
19         tmpA[i-left] = A[i];
20     }
21     for (size_t i = mid+1; i <= right; i++) {
22         tmpB[i-mid-1] = A[i];
23     }
24     int *tmpC = new int[sizeA + sizeB];
25     merge_helper(tmpA, sizeA, tmpB, sizeB, tmpC);
26     for (size_t i = left; i <= right; i++) {
27         A[i] = tmpC[i-left];
28     }
29     delete tmpA, tmpB, tmpC;
30 }
31
32 // REQUIRES: an array A, and two ints left and right
33 // EFFECTS: sort array A
34 // MODIFIES: array A
35 void merge_sort(int *A, size_t left, size_t right) {
36     if (left >= right) return;
37     size_t mid = (left + right) / 2;
38     merge_sort(A, left, mid);
39     merge_sort(A, mid+1, right);
```

```
    merge(A, left, mid, right);
}
```

The time complexity of merge sort in both worst and average case is $O(n \log n)$. As we learned in *discussion0.pdf*, $O(n^2)$ runs much slower when encountering large input compared with $O(n \log n)$ algorithms. Hence merge sort is widely used in many scenarios.
However, it is also memory-consuming, since it needs extra memory for **not-in-space** sorting. The overall space complexity is $O(n)$ (Different parts of codes contributes different amount of space usage)

1.3 Quick sort

Quick sort first chooses a pivot among the given array. The pivot can be randomly selected, or be selected based on some solid rules (such as always regarding the first entry as pivot). The best pivot is the median of the array.
Then it divides the array into two parts: putting all the elements smaller than the pivot on the left of the pivot, and putting all the elements greater than it on the right. This step is called **partition**.
Then it will divide the two sub arrays and repeat the steps above until all the sub arrays have been sorted.
One version of quick sort code is shown below, which is **slightly different** from what we learned in class. You need to stick to the version taught in class!!

```
// EFFECTS: partition the array with first element as pivot
int partition(int *A, size_t left, size_t right) {
    size_t i = left - 1;
    int x = A[right];
    for (size_t j = left; j < right; j++) {
        if (A[j] <= x) {
            i++;
            swap(A[i], A[j]);
        }
    }
    swap(A[i+1], A[right]);
    return i+1;
}

// REQUIRES: an array A, and two ints left and right
// EFFECTS: sort array A
// MODIFIES: array A
void quick_sort(int *A, size_t left, size_t right) {
    if (left >= right) return;
    size_t pivot_index = partition(A, left, right);
    if (pivot_index != 0) quick_sort(A, left, pivot_index-1);
    quick_sort(A, pivot_index+1, right);
}
```

The time complexity of quick sort in average case is different from that in worst case. The former one is $O(n \log n)$, while the latter one is $O(n^2)$.
The quick sort is not naturally stable, but it can be modified to be stable, while it is unnecessary to do that since you will need extra $O(n)$ memory space. This part is not required in this course. You can simply regard quick sort as **not stable** in VE281.
Although quick sort has $O(n^2)$ worst case time complexity, it is still widely used, and works well on smaller array(although slower than insertion sort).
You can find more details of difference between quick sort and merge sort in https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/

TRY TO WRITE THE CODE ABOVE ON YOUR OWN!
The implementations can be quite various, but you may meet thousands of problems (especially edge cases) while training. Only when you overcome all the bugs can you partially understand the beauty of each sorting algorithm. Good Luck!

2 Non-comparison sort

Non-comparison sort is only used under some special cases: such as all the elements are guaranteed to fall in a solid range. For this section, you need to understand the mechanism of three different type of non-comparison sort, and read the slides carefully.

2.1 Counting sort

Counting sort has $O(n + k)$ time complexity. The best way of understanding counting sort is to simulate the process by your own according to the steps given in the slides. We will not talk much about counting sort here.

2.2 Bucket Sort

Bucket sort actually contains a step of using comparison sort. It first puts each element in different buckets based on its key. And then sort those buckets by a comparison sort.
Bucket Sort is actually a **stable** sort. You can try by yourself to test how it can be stable.

2.3 Radix Sort

Radix sort can be applied to sort keys that are built on positional notation or that contains multiple keys. You can find more details on slides.