1 Comparison Sort (Elementary)

Before you read the section note series, one thing you need to know is: Section notes are only for strengthening the stuffs you have learned in class. Hence, these will not cover all the details, and you need to put the course slides aside to get better understanding.

Also, you can find all codes used in this section notes at VE281/Discussion/Discussion1

1.1 Sorting Basics

- Why **in place**?

  Assume you have a huge amount of data and you want to sort them. If you need to have extra memory to first store some values, you need mountains of extra storage capacity. Instead, if you can sort all the data in space, you only need $O(1)$ memory space, which is far more economical.

- Why **stable**?

  Usually when you need to sort some arrays (or vectors), the data type is not simply integer or float. Instead, it is a class or structure, with compound data. For example, we have:

```
struct Student {
  int score;
  string name;
}

// You need to sort Student Students[10]
```

  Suppose we have an array of *Student*, and we would like to sort the array based on *score*, while not changing the relative order if there is a tie. For example,

  The **original** array: $\{90,'Alice'\}$, $\{85,'Bob'\}$, $\{90,'Roihn'\}$,

  Stable result: $\{85,'Bob'\}$, $\{90,'Alice'\}$, $\{90,'Roihn'\}$,

  Unstable result: $\{85,'Bob'\}$, $\{90,'Roihn'\}$, $\{90,'Alice'\}$,

  Elementary sorts tend to be stable (not all).

  Complex sorts are often **not** stable.

Remark 1.1. Here we want to keep the original relative order while sorting; **Another** type of problems need you to sort with **multiple** keys (such as sorting poker cards, or first sort scores of Chinese, and then Maths...), which uses different type of sorting algorithms.

1.2 Insertion Sort

Insertion sort is widely used for a dataset with already sorted entries. When you need to add a new entry (such as dataset of user info, and a new user registers), insertion sort is quite efficient to take the job.

Here I will show you the sample code for insertion sort in C++. You can first try it by yourself, and then take this as your materials for check and review.

```
// REQUIRES: an array A and its size n
// EFFECTS: sort array A
// MODIFIES: array A
void insertion_sort(int *A, int size) {
```

```
    for (int i = 1; i < size; i++) {
        int j = 0;
        while (j < i && A[i] >= A[j]) {
            j++; // Find the location to insert the value (Think about why we use >= here)
        }
        int tmp = A[i]; // Store the value we need to insert
        for (int k = i; k > j; k--) {
            A[k] = A[k-1]; // Move back all the elements in (j,i]
        }
        A[j] = tmp;
    }
}

int main(){
    int A[5] = {2,1,4,5,3};
    for (auto item: A) {
        cout << item << ',';
    }
    cout << endl;
    insertion_sort(A, 5);
    for (auto item: A) {
        cout << item << ',';
    }
    cout << endl;
    return 0;
}
```

If you fully understand the mechanism of insertion sort, you may understand the slightly smarter implementation of insertion sort.

```
// EFFECTS: sort array A in a smarter way
void smart_insertion_sort(int*A, size_t size) {
    for (size_t i = 1; i < size; i++) {
        for (size_t j = i; j > 0; --j) {
            if (A[j] < A[j-1]) {
                swap(A[j], A[j-1]);
            }
        }
    }
}
```

* Actually swap takes longer time than copy. (swap takes approximately 3 times of work of copy). Try to modify it to use copy instead!

Advantages of Insertion sort:

- Run time depends upon initial order of keys in input
- Algorithm is tunable (We can make further improvement on the code above, this part is not required in VE281)
- **Best sort for small values of** n

Disadvantages of Insertion sort:

- Time complexity is $O(n^2)$

1.3 Selection Sort

Selection sort follows the steps below:

i) Find the smallest element in array, swap with first position

ii) Find the second smallest element in array, swap with second position

iii) ...

```
// REQUIRES: an array A and its size n
// EFFECTS: sort array A
// MODIFIES: array A
void selection_sort(int *A, int size) {
    for (int i = 0; i < size; i++) {
        int max = A[i];
        int index = i;
        for (int j = i; j < size; j++) {
            if (max > A[j]) {
                index = j;
                max = A[j];
            }
        }
        A[index] = A[i];
        A[i] = max;
    }
}
```

Advantages of Selection sort:

- Minimal copying of items

  Good choice when objects are large and copying is expensive.

- Fairly efficient for small n

Disadvantages of Selection sort:

- Time complexity is $O(n^2)$
- Run time only slightly dependent upon pre-order

1.4 Bubble sort

Bubble sort always swap the two adjacent entries, and after each outer iteration, we will have at least one entry (always the largest / smallest one among unordered entries) is swapped to the correct place.

```
// REQUIRES: an array A and its size n
// EFFECTS: sort array A
// MODIFIES: array A
void bubble_sort(int *A, int size) {
    for (int i = size-2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            if (A[j] > A[j+1]) {
                swap(A[j], A[j+1]);
            }
        }
    }
}
```

Advantages of Bubble sort:

- Completes some 'pre-sorting' while searching for largest / smallest key (not largest / smallest keys will also be swapped)
- Adaptive version may finish quickly if the input array is almost sorted (Adaptive: check whether there occurs at least one swap in an outer iteration; if not, then it convinces that the array is already sorted).

Disadvantages of Bubble sort:

- Time complexity is $O(n^2)$
- Many swaps

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!

In next section, we will talk about quicker sorting algorithms: Merge sort and Quick sort.