Selection Sort

There are two parts of data: (1) unsorted array (2) sorted array

- Start by let the input be the unsorted array
- Let sorted array be an empty array

Algorithm

```
While the unsorted array is not empty
    Get the maximum one
    Put it at the front of the sorted array
    Delete it from the unsorted array
```

Code (C++)

```cpp
template<typename T>
void selection_sort(vector<T> &v) {
    size_t pos = v.size() - 1;
    while (pos > 0) {
        int max_idx = 0;
        for (size_t i=0; i<=pos; i++) {
            if (v[i] > v[max_idx]) {
                max_idx = i;
            }
        }
        swap(v[pos], v[max_idx]);
        pos--;
    }
}
```

Time Complexity: $O(n^2)$

Heap Sort

Treat unsorted portion of the array as a binary heap.

- At start, we `build_heap()` $O(n)$ on the unsorted array
    - `build_heap()` = `fix_down()` on $\frac{n}{2}$ to 1

Algorithm

```
For each iteration, assume that the heap is at A[1...pos]
    The maximum element in A[1...pos] is at A[1]
    Do binary heap pop(), which is swapping A[1] with A[pos] and fix_down
```

Code (C++)

```
build_heap(v);

for (int i=v.size()-1; i>=0; i--) {
    swap(v[i], v[0]);
    fix_down(0, v, i);
}
```

`build_heap(v)` - to build binary heap.

```
void build_heap(vector<int> &v) {
    for (int i=v.size() / 2; i>=0; i--) {
        fix_down(i, v, v.size());
    }
}
```

`fix_down(index, v, sz)` - to fix heap property from top to bottom.

```
void fix_down(int index, vector<int> &v, int sz) {
    int temp = v[index];
    while (index*2+1 < sz && v[index*2+1] > temp) {
        if (index*2+2 < sz && v[index*2+2] > v[index*2+1]) {
            v[index] = v[index*2+2];
            index = index*2+2;
        } else {
            v[index] = v[index*2+1];
            index = index*2+1;
        }
    }
    v[index] = temp;
}
```

Time Complexity: $O(n\ log\ n)$

Insertion Sort

Try to insert the last element of the unsorted part to the sorted part.

Code (C++)

```
template<typename T>
void insertion_sort(vector<T> &v) {
    for (int pos=v.size()-2; pos>=0; pos--) {
        T tmp = v[pos];
        size_t i = pos+1;
        while (i < v.size() && v[i] < tmp) {
            v[i-1] = v[i];
            i++;
        }
```

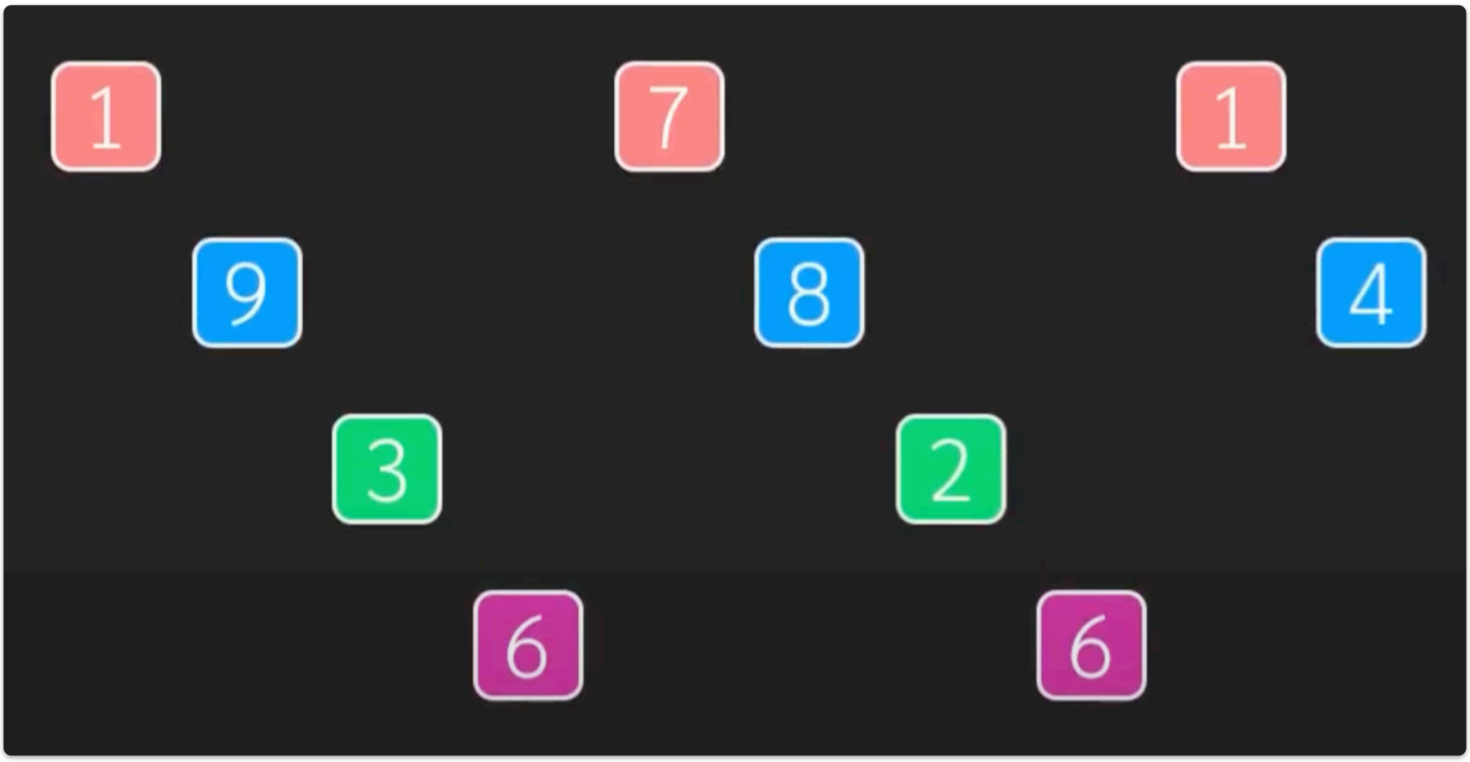
```
            v[i-1] = tmp;
        }
    }
```

Time Complexity: $O(n^2)$

Shell Sort

Start by letting G = some large value less than N and doing these process

- (a) Divide A into G smaller arrays, each consist of element in A that is G elements apart
- (b) Sort each sub-array by insertion sort
  Repeat (a) and (b) with smaller value of G
- Keep doing (a) and (b) until G is 1
- When $G = 1$ it is basically the insertion sort but the array should be almost sorted.



Example $G = 4$, Split into 4 groups (the distance between each element in the same group can be divided by G), and sort each sub-array.

Code (C++)

```
void shell_sort(vector<int> &v) {
    vector<int> gaps = {701, 301, 132, 57, 23, 10, 4, 1};
    int n = v.size();
    for (int G: gaps) {
        for (int round=n; round>=n-G+1; round--) {
            int pos = round-G;
            while (pos >= 0) {
                // Insertion sort
                int tmp = v[pos];
                int i = pos+G;
                while (i<n && tmp>v[i]) {
                    v[i-G] = v[i];
                    i = i + G;
                }
                v[i-G] = tmp;
                pos = pos - 1;
            }
        }
    }
}
```

Time Complexity: Depends on sequence of G (Current best case is $O(N^{\frac{4}{3}})$)