Exercise 1

The worst case complexity can be seen in the following code and is O(N^2). This is also the best case since we always have to check the rest of the array to find the smallest value.

Exercise1/sort.h – Line 14 - 33

```
template< typename T>
void selectSort(vector<T> &a)
{
    int min; // O(1)
    T temp; // O(1)
    for (int i = 0; i < a.size() - 1; i++) // O(N)
    {
        min = i; // O(1)
        for (int j = i + 1; j < a.size(); j++) // (N-1)/2 = O(N)
        {
            if(a[j] < a[min]) // O(1)
            {
                min = j; // O(1)
            }
        }
        temp = a[i]; // O(1)
        a[i] = a[min]; // O(1)
        a[min] = temp; // O(1)
    }
}
```

Exercise 2

First the counting array is updated where the amount of each number in A are counted at their index. (This has a complexity of O(N) since we must read alle the values from A)

Then the index array is updated with the sums of numbers less than x. (This also has a complexity of O(N) since we are calculating the sum of all previous values, thus only needing to go through once)

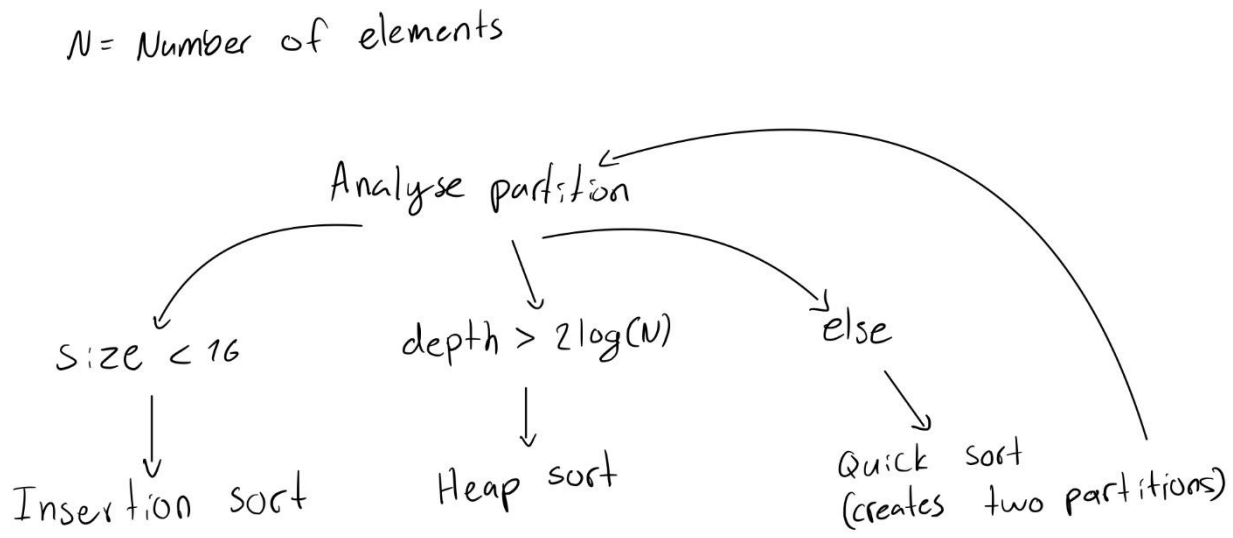
Finally, the output array is filled using the information from A and the counting array. (Here we also only go through A once and thus only using O(N))

Combined we get 3O(N), which is the same as O(N).

Exercise 3

We know that for each element in the heap we remove we must run heapify. Since heapify has a complexity of O(logN) and we have N elements, we get O(NlogN). This means that it does not matter if A is already sorted in increasing or decreasing order.

Exercise 4



For test cases we suggest making cases such that each arrow on the drawing are executed.
E.g. a case where quicksort has run enough times to make the depth larger than 2log(n) such that we enter heap sort. And when the element size is lower than 16 such that we enter insertion sort. To access all these we would implement an array of size much larger than 16 e.g. 1000 and then run the sorting sequence.

Exercise 5

Modify the heap implementation we saw on class to implement a max-heap in which the maximum element is stored in the root. Implement the priority queue interface discussed in class using the max-heap, and provide testing code.

All the code in binary_heap.tpp and binary_heap.h are relevant for the solution to this exercise. The code can be seen below:

Exercise5/binary_heap.tpp

```
/**
 * Insert item x, allowing duplicates.
 */
template<typename Comparable>
void BinaryHeap<Comparable>::insert(const Comparable& x) {
    if (currentSize == array.size() - 1)
        array.resize(array.size() * 2);

    // Move up
    int node = ++currentSize;
    //currentSize++;
    Comparable copy = x;

    array[0] = std::move(copy);
```

```
    for (; x > array[node / 2]; node /= 2)
        array[node] = std::move(array[node / 2]);
    array[node] = std::move(array[0]);
}

/**
 * Find the largest item in the priority queue.
 * Return the smallest item, or throw Underflow if empty.
 */
template<typename Comparable>
const Comparable& BinaryHeap<Comparable>::findMax() const {
    if (isEmpty()) throw underflow_error{"heap is empty."};
    return array[1];
}

/**
 * Remove the maximum item.
 * Throws UnderflowException if empty.
 */
template<typename Comparable>
void BinaryHeap<Comparable>::deleteMax() {
    if (isEmpty()) throw underflow_error("heap is empty.");
    array[1] = std::move(array[currentSize--]);
    maxHeapify(1);
}

/**
 * Remove the maximum item and place it in maxItem.
 * Throws Underflow if empty.
 */
template<typename Comparable>
void BinaryHeap<Comparable>::deleteMax(Comparable & maxItem) {
    if (isEmpty()) throw underflow_error("heap is empty.");
    maxItem = std::move(array[1]);
    array[1] = std::move(array[currentSize--]);
    maxHeapify(1);
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
template<typename Comparable>
void BinaryHeap<Comparable>::buildHeap() {
    for (int i = currentSize / 2; i > 0; --i) {
        maxHeapify(i);
    }
```

```
}

/**
 * Internal method to percolate down in the heap.
 * node is the index at which the percolate begins.
 */
template<typename Comparable>
void BinaryHeap<Comparable>::maxHeapify(int node) {
    int child;
    Comparable tmp = std::move(array[node]);
    //cout << "heapify ";

    for (; node * 2 <= currentSize; node = child) {
        child = node * 2;
        if (child != currentSize && array[child + 1] > array[child])
            ++child;
        if (array[child] > tmp)
            array[node] = std::move(array[child]);
        else
            break;
    }
    array[node] = std::move(tmp);
}

//Prints the heap
template<typename Comparable>
void BinaryHeap<Comparable>::print() {

    cout << endl;
    for (int i = 1; i <= currentSize; i++)
    {
        cout << array[i] << " ";
    }
    cout << endl;
}
```

In binary_heap.h we see the interface implementation for the priority queue. This just calls the appropriate function from the binary heap class, which does the desired function.

Exercise5/ binary_heap.h

```
template <typename Comparable>
class p_queue {
    public:
    BinaryHeap<Comparable> b;

    void pop()
    {
```

```
        b.deleteMax();
    }
    void push(const Comparable& x)
    {
        b.insert(x);
    }
    const Comparable& top() const
    {
        return b.findMax();
    }
    bool empty()
    {
        return b.isEmpty();
    }
    void clear()
    {
        b.clear();
    }
};
```