ASSIGNMENT 6

Name: DONGWOOK LEE

Problem 6.1

(a)

```
procedure bubbleSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length(A) - 2 inclusive do:
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

If there wasn't any swap among elements, it means that the array has already been sorted in right way. Thus, there is no need of more comparison iteration.
➔ End the 'do-while' loop

(If there was any swap, bool swapped is going to be true, which makes iteration keep going)

If two elements are in wrong order, swap;

(b)

```
Best Case:
int i;
for(i = 0 ; i < n ; i++){
    A[i] = i;
}
// We don't have to swap any pair of elements if they are all sorted from the beginning
```

➔O(n)

```
Worst Case:
int i;
for(i = 0 ; i < n ; i++){
    A[i] = n-i-1;
}
// We need to swap every consecutive pair of elements at least once
```

➔$O(n^2)$

```
Random Case:
int i;
for(i = 0 ; i < n ; i++){
    A[i] = rand() % n;
}
// If we simulate several cases with randomly generated arrays, we can measure the average (or
// common) execution time of our algorithm
```

➔$O(n^2)$

(c)

INSERTION SORT IS STABLE

In Insertion Sort, an element x_i is being compared to all elements at the left of it (from x_{i-1} to x_0). Then, x_i stops its comparison when it meets an element smaller or equal to itself in key value size.

➔ (Fix its position at the right side of an element with key value smaller than or equal to its).

e.g. Suppose two elements A and B have same key value (A on the left of B).

A is getting sorted first, and then B. Conclusively, B will still place on the right side of A.

MERGE SORT IS STABLE

Dividing an array into two subarrays does not change any order of the elements.

In merging (putting subarrays together int one) process, the elements are being compared and allocated into new temporary array from left to right.

➔ Does not change the original relative order of two elements with identical key value

HEAP SORT IS UNSTABLE

In the process of swapping a maximum valued element with the last element of the heap (and the size of heap being manipulated decreases by one per step), the original relative order of two elements can possibly be changed.

➔ Relative order of the elements with same key value can be changed during swap.

BUBBLE SORT IS STABLE

In Bubble Sort, only a pair of elements is being compared (two elements at once).

And the thing is, if two elements are equal in their values, they do not swap each other, which means they keep their original position. Thus, when two elements with same key value meet for comparison, their relative position before and after comparison will not differ.

➔ Relative order remains even after sorting, because two elements do not swap if they equal.

(d)

INSERTION SORT IS ADAPTIVE

If the array has already been sorted from beginning, of course the elements will be compared each other as usual, but do not really have to move their position. If there is an element which has to move 5 indices frontward, all five elements in front of that element must move to right by one (which requires five repetition – or loop).

➔ If the array has already been sorted, it will obviously decrease the execution time of algorithm.

MERGE SORT IS NOT ADAPTIVE

No matter the order of given array, this algorithm recursively divides a given array into two subarrays (until the size of subarray becomes 1). Thus, the only parameter that effects its execution time is SIZE OF ARRAY. Even if the array has been already sorted, all elements will be divided into individual subarrays and then be merged & relocated into new larger arrays.

➔ No matter the original order of the elements, time complexity remains same.

HEAP SORT IS NOT ADAPTIVE

No matter the order of a given array, the number of calls of Max-Heapify depends on the number of elements we have in our set.

BUBBLE SORT IS ADAPTIVE

If we have our array already been sorted, the elements do not have to be swapped each other. And as shown above, the time complexity of best-case (originally ordered array) and others obviously have difference ($O(n)$ and $O(n^2)$).

➔ Bubble sort's time complexity is effected by the previous order of a given array.

Problem 6.2

(a)

```cpp
#include <iostream>
#include <random>
#include <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

int n = 10;
int printsize = n;

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void printall(int* A) {
    int i;
    for (i = 1; i <= printsize; i++) {
        cout << A[i] << " ";
    }cout << endl;
}

void MAX_HEAPIFY(int* A, int i) {
    int largest;
    bool again = false;
    // l = LEFT(i) & r = RIGHT(i)
    int l = 2 * i, r = 2 * i + 1;
    //if l <= A.heapsize and A[l] > A[i] then largest = l;
    if (l <= n && A[l] > A[i]) {
        largest = l;
    }
    else {
        largest = i;
    }
    //if r <= A.heapsize and A[r] > A[largest] then largest = r
    if (r <= n && A[r] > A[largest]) {
        largest = r;
    }
    if (largest != i) {
        swap(A[i], A[largest]);
        again = true;
    }

    if (again) {
        MAX_HEAPIFY(A, largest);
    }
}
```

```cpp
void BUILD_MAX_HEAP(int* A) {
    int i;
    for (i = n / 2; i > 0; i--) {
        MAX_HEAPIFY(A, i);
    }
}

void Heapsort(int* A) {

    BUILD_MAX_HEAP(A);

    while (n > 1) {
        //exchange A[1] with A[n]
        swap(A[1], A[n]);

        //A.heapsize = A.heapsize - 1
        n--;

        //MAX_HEAPIFY(A,1)
        MAX_HEAPIFY(A, 1);
    }

}

int main() {

    //Generator
    srand(time(NULL));
    int i;
    int* A = new int[n+1];
    for (i = 1; i <= n; i++) { // do not use 0th index
        A[i] = rand() % n;
    }

    printall(A);
    Heapsort(A);
    printall(A);

    return 0;
}
```

(b)

```cpp
#include <iostream>
#include <random>
#include <ctime>
#include <chrono>
using namespace std;

int n = 0;
int printsize = 0;

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void printall(int* A) {
    int i;
    for (i = 1; i <= printsize; i++) {
        cout << A[i] << " ";
    }cout << endl;
}

void REVERSED_MAX_HEAPIFY(int* A, int i) {
    bool again = false;
    // parent & sibling
    int parent = i / 2;
    //if parent >= 1 and A[parent] > A[i] then largest = parent;
    if (parent >= 1 && A[parent] > A[i]) {
        swap(A[i], A[parent]);
        again = true;
    }

    if (again) {
        REVERSED_MAX_HEAPIFY(A, parent);
    }
}
```

```cpp
void MAX_HEAPIFY(int* A, int i) {
    int largest;
    bool again = false;
    // l = LEFT(i) & r = RIGHT(i)
    int l = 2 * i, r = 2 * i + 1;
    //if l <= A.heapsize and A[l] > A[i] then largest = l;
    if (l <= n && A[l] > A[i]) {
        largest = l;
    }
    else {
        largest = i;
    }
    //if r <= A.heapsize and A[r] > A[largest] then largest = r
    if (r <= n && A[r] > A[largest]) {
        largest = r;
    }
    if (largest != i) {
        swap(A[i], A[largest]);
        again = true;
    }

    if (again) {
        MAX_HEAPIFY(A, largest);
    }
}

void BUILD_MAX_HEAP(int* A) {
    int i;
    for (i = n / 2; i > 0; i--) {
        MAX_HEAPIFY(A, i);
    }
}

int FlowRootTillLeave(int* A, int i) {
    int l = 2 * i, r = 2 * i + 1;
    int larger;
    if (r <= n) {
        if (A[l] >= A[r]) {
            swap(A[l], A[i]);
            larger = l;
        }
        else {
            swap(A[r], A[i]);
            larger = r;
        }
        FlowRootTillLeave(A, larger);
    }
    else if (l <= n) {
        swap(A[l], A[i]);
        return l;
    }
    return i;
}
```

```cpp
void Heapsort(int* A) {

    BUILD_MAX_HEAP(A);

    while (n > 1) {
        //exchange A[1] with A[n]
        swap(A[1], A[n]);

        //A.heapsize = A.heapsize - 1
        n--;

        //Flow till the end
        int currentLV = FlowRootTillLeave(A, 1);

        //Reversed_MAX_HEAPIFY
        REVERSED_MAX_HEAPIFY(A, currentLV);
    }

}

int main() {

    cin >> n;
    printsize = n;

    //Generator
    srand(time(NULL));
    int i;
    double sum = 0.0;
    int *A = new int[n + 1];

    for (int h = 0; h < 100; h++) {
        //Generator
        n = printsize;
        for (i = 1; i <= n; i++) { // do not use 0th index
            A[i] = rand() % 10;
        }

        //printall(A);
        chrono::system_clock::time_point start = chrono::system_clock::now();
        Heapsort(A);
        chrono::duration<double> ET = chrono::system_clock::now() - start;
        sum += ET.count()*1000000;
    }cout << sum/100.0 << endl;

    return 0;
}
```