Melissa Hazlewood
CECS 328
Lab 5

```cpp
//
//  CECS328Lab5.cpp
//  Lab5
//
//  Created by Melissa Hazlewood on 11/25/19.
//  Copyright © 2019 Melissa Hazlewood. All rights reserved.
//

#include <iostream>
#include <random>
#include <iomanip>

void heap_sort(int arr[], int n);
void build_MaxHeap(int arr[], int n);
void max_heapify(int arr[], int i, int n);
void selectionSort(int arr[], int i, int n);
void printArray(int arr[], int i, int n);
int * generateArray(int n);

void partA1to3();
void partA4and5();
void partB();

using namespace std;

int main()
{
    srand(static_cast<int>(time(0))); //only call once; makes everything random every time

    partA1to3();
    partA4and5();
    partB();

}

void partA1to3()
{
    cout << "*******************************************   Part A *******************************************\n" << endl;
    // testing values
//    int arr[7] = {2, 6, -10, 99, -43, 98, 4};
```

```
//    int n = 7;
//    int i = 0;

    // actual values
    int * arr;
    int n;
    int i = 0;

    cout << "1.)" << endl;
    cout << "Enter a positive integer: ";
    cin >> n;

    cout << "\n2.)" << endl;
    cout << "Generating array with length " << n << "..." << endl;
    arr = generateArray(n);
    cout <<  "Done.\n\nArray:" << endl;
    printArray(arr, i, n);

    cout << "\n3.)" << endl;
    cout << "Sorting array using heap sort algorithm..." << endl;
    heap_sort(arr, n);
    cout <<  "Done.\n\nSorted array:" << endl;
    printArray(arr, i, n);

}

void partA4and5()
{
    int * arr;
    int n;
    int i = 0;
    double avgHeapTime = 0; //avg running time of heap sort with n=1000 elems
    double avgSelTime = 0; //avg running time of selection sort with n=1000 elems
    double avgHeapTime100 = 0; //avg running time of heap sort w n=100 elems
    double avgSelTime100 = 0; //avg running time of selection sort w n=100 elems
    clock_t timer;

    for (i = 0; i < 100; i++) //100 iterations
    {
        n = 1000;

```

```cpp
        // HEAP SORT (n = 1000)
        arr = generateArray(n);
        timer = clock(); //start the clock
            heap_sort(arr, n);
        timer = clock() - timer; //clock ticks elapsed; stop the clock
        avgHeapTime += timer; //add time to total (for heap sort, 100 iterations)


        // SELECTION SORT (n = 1000)
        arr = generateArray(n);
        timer = clock(); //restart the clock
            selectionSort(arr, 0, n);
        timer = clock() - timer; //clock ticks elapsed; stop the clock again
        avgSelTime += timer; //add time to total (for selection sort, 100 iterations)


        n = 100;

        // HEAP SORT (n = 100)
        arr = generateArray(n);
        timer = clock(); //start the clock
            heap_sort(arr, n);
        timer = clock() - timer; //clock ticks elapsed; stop the clock
        avgHeapTime100 += timer; //add time to total (for heap sort, 100 iterations)


        // SELECTION SORT (n = 100)
        arr = generateArray(n);
        timer = clock(); //restart the clock
            selectionSort(arr, 0, n);
        timer = clock() - timer; //clock ticks elapsed; stop the clock again
        avgSelTime100 += timer; //add time to total (for selection sort, 100 iterations)
    }


    cout << "\n4.)" << endl;
    cout << "Determining average running time for heap sort..." << endl;
    avgHeapTime = avgHeapTime/100/static_cast<double>(CLOCKS_PER_SEC);
    avgHeapTime100 = avgHeapTime100/100/static_cast<double>(CLOCKS_PER_SEC);
    cout << "Done." << endl;

```

```
    cout << "\n5.)" << endl;
    cout << "Determining average running time for selection sort..." << endl;
    avgSelTime = avgSelTime/100/static_cast<double>(CLOCKS_PER_SEC);
    avgSelTime100 = avgSelTime100/100/static_cast<double>(CLOCKS_PER_SEC);
    cout << "Done." << endl;


    cout << scientific << setprecision(2) << "Results:" << endl;
    cout << "\n|  Algorithm \t\t|   Avg Running Time (n=1000)\t|   Avg Running Time (n=100)   |" << endl;
    cout << "-----------------------------------------------------------------------------------" << endl;
    cout << "|  Heap sort\t\t|\t\t   " << avgHeapTime << " s \t\t\t|  \t\t    " << avgHeapTime100 << " s\t\t   |" << endl;
    cout << "|  Selection sort\t|\t\t   " << avgSelTime << " s \t\t\t|    \t\t" << avgSelTime100 << " s\t\t   |" << endl;
    cout << fixed << setprecision(4);
    cout << "|  Selection/Heap\t|\t\t\t\t\t\t\t\t|\t\t\t\t\t\t\t   |" << endl;
    cout << "|  Ratio  \t\t\t|\t\t\t " << avgSelTime/avgHeapTime << "\t\t\t\t|\t\t\t  " << avgSelTime100/avgHeapTime100 << "\t\t   |" << endl;

    cout << "\nDiscussion of results:" << endl;
    cout << "\tWe can see from the table that at smaller array lengths, selection sort is slightly faster than heap sort, but the algorithms are for the most part comparable. As array size grows, it is clear that heap sort beats out selection sort, completing the task in a quarter of the time." << endl;


}

void partB()
{
    cout << "\n\n********************************************   Part B ******************************************\n" << endl;

    clock_t timer;

    cout << "1.)" << endl;
    cout << "Generating array..." << endl;
    int * arr = generateArray(10);
```

Melissa Hazlewood
CECS 328
Lab 5

```
    int * arr = generateArray(10);
    cout << "Done. Unsorted 10-element array generated:" << endl;
    printArray(arr, 0, 10);

    cout << "\n2.)" << endl;
    cout << "Sorting array using heap sort..." << endl;
    timer = clock();
    heap_sort(arr, 10);
    timer = clock() - timer;
    cout << "Done in " << scientific << setprecision(1) << timer/static_cast<double>(CLOCKS_PER_SEC) << " seconds." << endl;

    cout << "\n3.)" << endl;
    cout << "Sorted array:" << endl;
    printArray(arr, 0, 10);

    cout << endl;
}



void heap_sort(int arr[], int n)
{
    int temp;

    build_MaxHeap(arr, n);

    // "deleting" the root
    for (int i = n; i > 0; i--)
    {
        temp = arr[0]; //swaps root (always first elem) with the right-most leaf (the last element of the array before the previously deleted roots
        arr[0] = arr[i-1];
        arr[i-1] = temp;
        max_heapify(arr, 0, i-1); //max-heapifies the new "root" up to the length of the unsorted "subarray" i-1
    }
}

void build_MaxHeap(int arr[], int n)
{
```

```
    for (int i = n; i >= 0; i--)
    {
        max_heapify(arr, i, n);
    }
}

void max_heapify(int arr[], int i, int n)
{
    int left = 2*i + 1;
    int right = 2*i + 2;
    int maxn = arr[i];
    int maxi = i;
    int temp;

    if (left < n) //if there's a left child, check if there's also a right child, then see if left is bigger than node i (after checking right)
    {
        if (right < n) //if there's a right child, check if it's bigger than node i
        {
            maxn = max(arr[right], maxn);
            if (maxn == arr[right])
                maxi = right; //update maxi
        }

        maxn = max(arr[left], arr[maxi]);
        if (maxn == arr[left])
            maxi = left; //update maxi

        if (maxi != i) //if node is not bigger than than both children, swap it with the larger node and continue max heapify on the original node (now at maxi)
        {
            temp = arr[i];
            arr[i] = arr[maxi];
            arr[maxi] = temp;
            max_heapify(arr, maxi, n);
        }
    }
}
```

Melissa Hazlewood
CECS 328
Lab 5

```
void selectionSort(int arr[], int i, int n)
{
    int temp;
    int min;
    int minInd = 0;

    while (i < n)
    {
        min = 101; //since our elems will never be larger than 100, everything in arr will be smaller, so this value will immediately get overriden in finding the actual min

        for (int j = i; j < n; j++)
        {
            if (arr[j] < min)
            {
                min = arr[j];
                minInd = j;
            }
        }

        temp = arr[i];
        arr[i] = min;
        arr[minInd] = temp;

        i++; //find next smallest number
    }
}


void printArray(int arr[], int i, int n)
{
    cout << "\t[ ";
    for (int i = 0; i < n-1; i++)
    {
        cout << arr[i] << ", ";
    }
    cout << arr[n-1] << " ]\n";
}


int * generateArray(int n) //diff from labs 1-4 for future reference; this one has repeat elems
{
    int * arr = new int[n]; //create new pointer to the array that will be returned
//    srand(static_cast<int>(time(0))); //commented out bc it is called already in main()

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 201 - 100;
    }
    return arr;
}

```

Melissa Hazlewood
CECS 328
Lab 5

```
*******************************************   Part A   *******************************************

1.)
Enter a positive integer: 15

2.)
Generating array with length 15...
Done.

Array:
    [ 12, -74, -78, -91, 53, 48, 81, -28, -31, 58, -91, -90, 56, -90, 27 ]

3.)
Sorting array using heap sort algorithm...
Done.

Sorted array:
    [ -91, -91, -90, -90, -78, -74, -31, -28, 12, 27, 48, 53, 56, 58, 81 ]

4.)
Determining average running time for heap sort...
Done.

5.)
Determining average running time for selection sort...
Done.
Results:

|  Algorithm        |   Avg Running Time (n=1000)   |   Avg Running Time (n=100)   |
-----------------------------------------------------------------------------------
|  Heap sort        |          2.75e-04 s           |           1.89e-05 s         |
|  Selection sort   |          1.15e-03 s           |           1.62e-05 s         |
|  Selection/Heap   |                               |                              |
|  Ratio            |            4.1765             |             0.8589           |

Discussion of results:
    We can see from the table that at smaller array lengths, selection sort is slightly faster
        than heap sort, but the algorithms are for the most part comparable. As array size grows,
        it is clear that heap sort beats out selection sort, completing the task in a quarter of
        the time.
```

```
*******************************************   Part B   *******************************************

1.)
Generating array...
Done. Unsorted 10-element array generated:
    [ 78, 85, -73, 50, 15, -94, -92, -85, 21, 61 ]

2.)
Sorting array using heap sort...
Done in 3.0e-06 seconds.

3.)
Sorted array:
    [ -94, -92, -85, -73, 15, 21, 50, 61, 78, 85 ]

Program ended with exit code: 0
```