Eric Watson

ECE 331 Homework 5

Implementation

This assignment required us to implement the merge sort and heap sort algorithms as c functions, and then compare the execution times upon testing the algorithms with integer arrays of size 10, 100, and 1000. To ensure a valid comparison, the same array of randomly generated integers was passed to both functions. For both the merge sort and heap sort functions I followed the pseudo code provided in the textbook very closely. However, I did initially encounter some problems with my heap sort algorithm that required me to make adjustments.

For the heap sort algorithm I started by creating a 'struct' with attributes size and a pointer to the array. This allowed me to simulate the 'heap', and make simpler references to array indices. I found the layout of the *if* and *else* statements in the pseudo-code of book to be slightly confusing for the function 'Max-Heapify', so I took the premise and eliminated the initial *else* statement by initializing the variable *largest* as equal to my *index* from the start, and then changed the value of *largest* based on the corresponding conditions presented in the pseudo code.

Within my main function I populate an array with random integers, make a copy of the array for the second sorting algorithm, initiate and stop the execution timers when each sorting function is called, and then print the resulting sorted arrays to ensure each algorithm was sorting properly. In order to populate the array with random integers I utilized the c library functions *rand()* and *srand()*. Since our maximum test array size was only 1000, I made this the seed value so that the randomly generated numbers would be ranging from 0 to 1000. In order to obtain the execution times I used the C library timing function *clock()*.

Analysis

Like merge-sort, heap-sort's running time is O(n lg n). This is because the call to Build-Max-Heap takes time O(n) and each of the *n-1* calls to Max-Heapify takes time O(lg n). However, unlike merge sort, heap-sort sorts in place (similar to insertion sort), thus only a constant number of array elements are stored outside the input array at any time. This means that you will not unexpectedly run out of heap or stack space on very large inputs (it uses O(1) auxillary space). On the other hand, merge sort is a very stable O(n lg n) sorting method, but it has a O(n) auxillary space and therefore uses a larger amount of stack space for greater number of inputs.

The analysis performed for this homework assignment revealed that heap sort had a longer execution time for each of the array sizes that were tested. I believe that this is partly due to the fact that there is a higher level of complexity in the c programming implementation of heap sort, than merge sort. Therefore, depending on how the code is written, this can impact code performance and execution time.

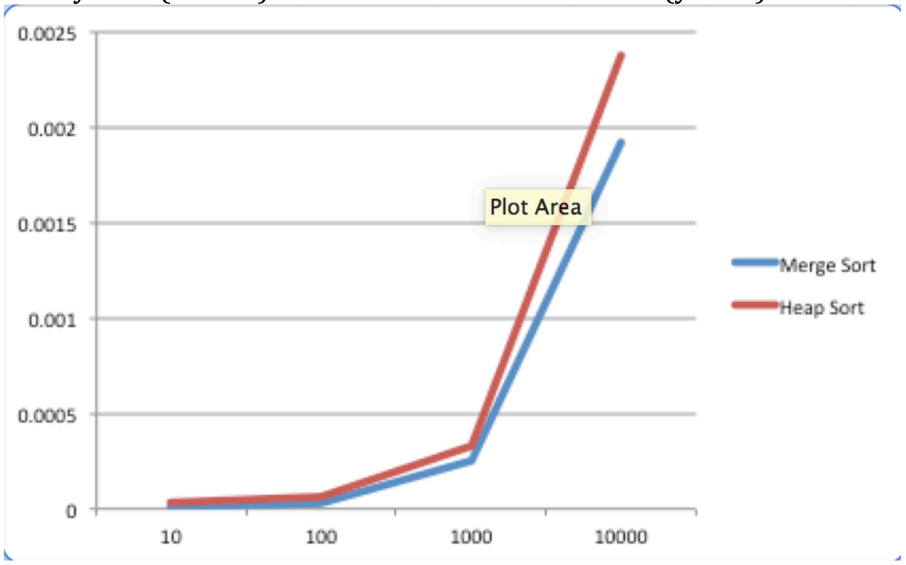
Based on research, and the analysis performed for this homework assignment, I would recommend using the heap sort algorithm in situations where you do not need a stable sort and there are a very large number of inputs. On the other hand, I would use the merge-sort algorithm when you need a stable, O(n lg n) sorting method and when you have a smaller number of inputs, such that you are not concerned about the amount stack space used.

Sorting Algorithm Execution Times

Array Size:	**10**	**100**	**1000**
Merge Sort	11 µ seconds	30 µ seconds	250 µ seconds
Heap Sort	33 µ seconds	59 µ seconds	332 µ seconds

Execution Time Comparison

Array Size (x- axis) vs Execution time in second (y-axis)



Upon plotting the test results side by side, we can see that the rate of growth is indeed roughly the same, as explained by the text to be O(n lg n).

The terminal outputs below demonstrate the full functionality of the program. For debugging purposes I print the original randomly generated array (which was copied for use with both algorithms), the execution times for each sorting algorithm, and also the resulting sorted arrays for both methods. I wrote my code so that the array debugging statements print only the first 10 array elements. This demonstrates that the algorithms are working properly, without creating a lengthy terminal output for large sized arrays.

Testing Array Size of 10

```
Erics-MacBook-Pro-2:HW05 Watson$ gcc hw05.c
Erics-MacBook-Pro-2:HW05 Watson$ ./a.out
Input Array Size: 10
The randomly generated values are:
949
217
257
893
214
746
424
255
625
697
```

```
Heap Sort Execution Time: 0.000033 seconds
The first 10 heap-sorted values are:
214
217
255
257
424
625
697
746
893
949
Merge Sort Execution Time: 0.000011 seconds
The first 10 merge-sorted values are:
214
217
255
257
424
625
697
746
893
949
```

Testing Array Size of 100

```
Erics-MacBook-Pro-2:HW05 Watson$ ./a.out
Input Array Size: 100
Heap Sort Execution Time: 0.000059 seconds
The first 10 heap-sorted values are:
31
33
36
47
49
74
82
86
88
92
Merge Sort Execution Time: 0.000030 seconds
The first 10 merge-sorted values are:
31
33
36
47
49
74
82
86
88
```

Testing Array Size of 1000

```
Erics-MacBook-Pro-2:HW05 Watson$ ./a.out
Input Array Size: 1000
Heap Sort Execution Time: 0.000332 seconds
The first 10 heap-sorted values are:
1
1
3
6
6
6
6
7
10
10
Merge Sort Execution Time: 0.000250 seconds
The first 10 merge-sorted values are:
1
1
3
6
6
6
6
7
10
10
```

Eric Watson

HW05 Source Code (hw05.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// Using a struct for the 'heap'
struct MaxHeap
{
        int size;
        int* array;
};

// In order to maintain the max-heap property, we call the procedure max-heapify
void max_heapify(struct MaxHeap* max_heap, int index){
        //root is at index 0
        // leftchild(i) = 2*i + 1
        // rightchild(i) = 2*i +2
        // parent(i) = (i-l)/2
        int largest = index;       // initialize largest = index
        int left = (index <<1) +1; //left child = 2*index +1
        int right = (index+1) <<1; //right child = 2*index +2

        // Determine if left child of root exists and is > root
        if(left< max_heap->size && max_heap->array[left]> max_heap->array[index]){
                largest =left;
        }
        // Determine if right child of root exists and is > the largest so far
        if( right < max_heap->size && max_heap->array[right] > max_heap->array[largest]){
                largest =right;
        }
        if(largest != index){
                // Exchange A[i] with A[largest]
                int q = max_heap->array[largest];
                max_heap->array[largest] = max_heap->array[index];
                max_heap->array[index] = q;

        max_heapify(max_heap, largest);
        }
}
// The procedure buildmaxheap goes through the remaining
// nodes of the tree and runs max_heapify on each one.
// Make sure to return struct!
struct MaxHeap* buildmaxheap(int* array, int size){

        int i;
        struct MaxHeap* max_heap = (struct MaxHeap*) malloc (sizeof(struct MaxHeap));
        max_heap->size = size;  //size of heap
        max_heap->array = array; // assign address to first element of array

//for i =  floor( A.length/2) down to 1
// max-heapify (array, i)

        for (i = (max_heap->size -2)/2; i>=0; --i){
                max_heapify(max_heap, i);
        }
        return max_heap;
}


void heap_sort(int* array, int size){
        struct MaxHeap* max_heap = buildmaxheap(array, size); //build
//for i = A.length downto 2
        while( max_heap->size >1){
// Exchange A[1] with A[i]
```

```
                int q = max_heap->array[0];
                max_heap->array[0] = max_heap->array[max_heap->size -1];
                max_heap->array[max_heap->size-1] = q;

// A.heap-size = A.heap-size - 1
                --max_heap->size;

                max_heapify(max_heap, 0);
        }
}

void merge(int B[], int p, int q, int r){
        int i, j, k;
        int n1 = q - p + 1; //length of subarray A[p..q]
        int n2 = r - q;                 //length of subarray A[q+1..r]
// let L[1..n1+1] and R[1..n2+1] be new arrays
        int L[n1+1], R[n2+1];

        for (i=0; i< n1; i++){
                L[i] = B[p+i-1];
        }

        for (j=0; j< n2; j++){
                R[j] = B[q+j];
        }
//      Issue sentinel cards
        L[n1] = 9999999;
        R[n2] = 9999999;

        i = 0;
        j = 0;

        for (k = (p-1); k < r; k++){
                if (L[i] <= R[j]){
                        B[k] = L[i];
                        i = i + 1;
                }

                else{
                        B[k] = R[j];
                        j=j+1;
                }
        }
}

void merge_sort(int B[], int p, int r){

        if (p < r){

                int q = floor((p+r)/2);
                merge_sort(B, p, q);
                merge_sort(B, q+1, r);
                merge (B,p,q,r);
        }
}
```

```
int main(){

        clock_t begin_heapsort, end_heapsort, begin_merge, end_merge;
        double heapsort_exec_time;
        double merge_exec_time;
        int size,i,j,k,l,m,x,w,z;

        time_t seconds;
        time(&seconds);
        srand((unsigned int) seconds);

        printf("Input Array Size: ");
        scanf("%d", &size);
        int A[size];
        int B[size];    //copy for second sort method

        for (i=0; i< size; i++){                //populate with random integers
                A[i] = rand() % 1000;
        }

//DEBUGGING - Make sure the integers are randomly generated
        printf("The randomly generated values are:\n");
        for (j =0; j < size; j++){
                printf("%d\n", A[j]);
        }

// Copy the Random valued integer array
        for (z=0; z < size; z++){
                B[z] = A[z];
        }

//DEBUGGING - Make sure the vaues are copied correctly
//      printf("The copied values are:\n");
//      for (x =0; x < size; x++){
//              printf("%d\n", B[x]);
//      }

        begin_heapsort = clock();       // start clock
        heap_sort(A, size);
        end_heapsort = clock();         // end clock
        // calculate time (sec)
        heapsort_exec_time = (double)(end_heapsort-begin_heapsort)/ CLOCKS_PER_SEC;
        printf("Heap Sort Execution Time: %f seconds\n", heapsort_exec_time);

//DEBUGGING - Make sure the values are sorted properly
        printf("The first 10 heap-sorted values are:\n");
        for (w =0; w < 10; w++){
                printf("%d\n", A[w]);
        }

        begin_merge = clock();                          //start clock
        merge_sort(B,1,size);                                   // merge sort
        end_merge = clock();                                    // end clock
        merge_exec_time = (double)(end_merge-begin_merge)/ CLOCKS_PER_SEC;

        printf("Merge Sort Execution Time: %f seconds\n", merge_exec_time);

//DEBUGGING - Make sure the values are sorted properly
        printf("The first 10 merge-sorted values are:\n");
        for (m =0; m < 10; m++){
                printf("%d\n", B[m]);
        }

        return(0);
}
```