Algorithms Assignment

Sorting Algorithms

Due Date: October 23, 2024

Max Thursby | 010967047 | mathursb@uark.edu

0 Late Days

Implementation:

[1] Quick Sort:

The quick sort implementation consists of 3 functions, the main recursive function, sort(), and the two helper functions, partition() and setPivotID(). The main function is very short and only contains the base case, if (l>=r), a call to the partition helper function setting the return value equal to the pivot index and two calls to itself on the subarrays l -> pivotInd-1, and pivotInd+1 -> r. Inside the previously mentioned partition helper function, it starts by setting an int value ‘pID’ equal to the return of the other helper function setPivotID. It then swaps the positions of array[pID] and array[r] to put the pivot at the end of the vector and sets the ArrayValue ‘pivot’ equal to array[r] where the pivot value is. A value ‘i’ is then set to the location of where the next smaller value should go and we check the key of ‘j’ which starts at j and goes to r. If the value at ‘j’ is smaller than the pivot it gets swapped with ‘i’ and ‘i’ is incremented. Lastly in the partition function after the for loop we swap ‘i+1’ and ‘r’ to put the pivot back in place and we return the pivot index with i + 1. The Last function used in the quick sort is the setPivotID() function, This uses the previous ‘l’ and ‘r’ values to find an ‘m’ value which is the middle point between ‘l’ and ‘r’. The keys for each of these values are taken and

compared with each other to find the median value, and this median value is returned as the pivot index.

[2] Merge Sort:

The merge sort algorithm uses the same base case as the quick sort and creates the same ‘m’ value as the setPivotID() function. However after this the merge sort algorithm makes recursive calls to itself on the subarrays ‘l->m’ and ‘m+1->r’ these inturn continue to split the array until the base case then these arrays begin the merge function. The merge function takes in four parameters the array, and the three positions l, m, and r. It creates a tmp vector for storing the new sorted array and sets variables x and y to l and m+1 respectively. Then while x <= m and y <= r it checks if the key at x is less than the key at y, if so it pushes array x into the tmp vector and increments x by one and if not it does the same for y. After this loops finishes it checks for the remainder of x and y values to push them into the back of the array. Lastly it puts the new sorted array into the original and this repeats for every iteration the original function took.

[3] Heap Sort:

Heap sort is again split into two different function calls, both of which are recursive, sort() and heapify(). Sort starts by initializing an integer value ‘n’ set equal to the size of the array, r-l+1, this value is then used to initialize a new vector named heap which we copy the original array into. The function then creates the max heap by looping through the non leaf nodes in the heap using a for loop from (n/2)-1 to 0 calling the heapify() function. Then in a loop from n-1 to 0 it starts extracting the largest value in the heap at heap[0] and swaps it with the end of the vector heap[i]. This goes until the heap is completely sorted in the heap vector. The last call sort() makes is copying the new heap vector into the old array vector. The heapify() function is pretty straight forward, it starts by initializing a largest value that is set equal to the current

value passed into the function, and left and right children for that value which are located at 2*current, +1 for the left child and +2 for the right child. Next it checks if left is less than the value N passed into heapify() and if it is it sets largest equal to left and this same logic is applied to the right child as well. Lastly heapify() checks if the largest value has changed since it was called by performing if(largest!=current) if this checks then it swaps the current and largest positions in the heap and recursively calls itself on the new heap it created.

[4] Intertwined Sort:

The intertwined sort consists of the heap sort in order to create a sorted list and then a separate sort() function that starts with the previous base case of (l>=r) and then continues by setting variables equal to l and r and a size variable equal to (r-l+1). It then creates a tmp vector to store the new sorted vector in and calls the heap sort function to sort. It then proceeds with the intertwined sort logic which is a for loop through the entire list that checks if the value of 'i' is even or odd and depending will push the value at array[left] into the tmp vector and increment left or array[right] and decrement right respectively. It then copies this new sorted vector into the original vector to finish.

Time Complexity Comparisons:

Quick sort: 7.37 sec

Merge sort: 11.17 sec | ^ 51.5%

Heap sort: 15.27 sec | ^ 107%

Intertwined sort: 16.32 sec | ^121%

The fastest algorithm was the quick sort with the longest being intertwined sort.

Outputs:

[1]

```
max@max-envy-linux:~/Desktop/Homework/Algorithms/HW_3/HW3$ make qsort
g++ -I./include/ -std=c++11 src/algorithms/mst.cpp src/graph.cpp src//main.cpp src/sort/qsort.cpp  -o bin/qsort
./bin/qsort
Perform unit test on the sorting algorithm
You are using Quick Sort algorithm
[Test Case 1]. Performing Testing Quick Sort on Array of 10 elements.
[Pass Test Case 1]. Average Running Time of Your Quick Sort Implementation Is 0.000002122620000 Seconds

[Test Case 2]. Performing Testing Quick Sort on Array of 100 elements.
[Pass Test Case 2]. Average Running Time of Your Quick Sort Implementation Is 0.000030222320000 Seconds

[Test Case 3]. Performing Testing Quick Sort on Array of 1000 elements.
[Pass Test Case 3]. Average Running Time of Your Quick Sort Implementation Is 0.000422235030000 Seconds

[Test Case 4]. Performing Testing Quick Sort on Array of 10000 elements.
[Pass Test Case 4]. Average Running Time of Your Quick Sort Implementation Is 0.005234732490000 Seconds

[Test Case 5]. Performing Testing Quick Sort on Array of 100000 elements.
[Pass Test Case 5]. Average Running Time of Your Quick Sort Implementation Is 0.062834247190000 Seconds

[Test Case 6]. Performing Testing Quick Sort on Array of 1000000 elements.
[Pass Test Case 6]. Average Running Time of Your Quick Sort Implementation Is 7.373595197300003 Seconds

Your sorting implementation is correct


Perform unit test on your implementation with graph
Minimum Spanning Tree. Total Cost: 16



Perform graph testing  on your implementation with graph
Minimum Spanning Tree. Total Cost: 9290
You have to use OpenCV to visualize your map
```

[2]

```
max@max-envy-linux:~/Desktop/Homework/Algorithms/HW_3/HW3$ make msort
g++ -I./include/ -std=c++11 src/algorithms/mst.cpp src/graph.cpp src//main.cpp src/sort/msort.cpp  -o bin/msort
./bin/msort
Perform unit test on the sorting algorithm
You are using Merge Sort algorithm
[Test Case 1]. Performing Testing Merge Sort on Array of 10 elements.
[Pass Test Case 1]. Average Running Time of Your Merge Sort Implementation Is 0.000003513720000 Seconds

[Test Case 2]. Performing Testing Merge Sort on Array of 100 elements.
[Pass Test Case 2]. Average Running Time of Your Merge Sort Implementation Is 0.000047146750000 Seconds

[Test Case 3]. Performing Testing Merge Sort on Array of 1000 elements.
[Pass Test Case 3]. Average Running Time of Your Merge Sort Implementation Is 0.000544578610000 Seconds

[Test Case 4]. Performing Testing Merge Sort on Array of 10000 elements.
[Pass Test Case 4]. Average Running Time of Your Merge Sort Implementation Is 0.007413371230000 Seconds

[Test Case 5]. Performing Testing Merge Sort on Array of 100000 elements.
[Pass Test Case 5]. Average Running Time of Your Merge Sort Implementation Is 0.086336922810000 Seconds

[Test Case 6]. Performing Testing Merge Sort on Array of 1000000 elements.
[Pass Test Case 6]. Average Running Time of Your Merge Sort Implementation Is 11.171974479099998 Seconds

Your sorting implementation is correct


Perform unit test on your implementation with graph
Minimum Spanning Tree. Total Cost: 16



Perform graph testing  on your implementation with graph
Minimum Spanning Tree. Total Cost: 9290
You have to use OpenCV to visualize your map
```

[3]

```
max@max-envy-linux:~/Desktop/Homework/Algorithms/HW_3/HW3$ make hsort
g++ -I./include/ -std=c++11 src/algorithms/mst.cpp src/graph.cpp src//main.cpp src/sort/hsort.cpp  -o bin/hsort
./bin/hsort
Perform unit test on the sorting algorithm
You are using Heap Sort algorithm
[Test Case 1]. Performing Testing Heap Sort on Array of 10 elements.
[Pass Test Case 1]. Average Running Time of Your Heap Sort Implementation Is 0.000002774860000 Seconds

[Test Case 2]. Performing Testing Heap Sort on Array of 100 elements.
[Pass Test Case 2]. Average Running Time of Your Heap Sort Implementation Is 0.000044115050000 Seconds

[Test Case 3]. Performing Testing Heap Sort on Array of 1000 elements.
[Pass Test Case 3]. Average Running Time of Your Heap Sort Implementation Is 0.000644062280000 Seconds

[Test Case 4]. Performing Testing Heap Sort on Array of 10000 elements.
[Pass Test Case 4]. Average Running Time of Your Heap Sort Implementation Is 0.008161100840000 Seconds

[Test Case 5]. Performing Testing Heap Sort on Array of 100000 elements.
[Pass Test Case 5]. Average Running Time of Your Heap Sort Implementation Is 0.104383925870000 Seconds

[Test Case 6]. Performing Testing Heap Sort on Array of 1000000 elements.
[Pass Test Case 6]. Average Running Time of Your Heap Sort Implementation Is 15.266830600500001 Seconds

Your sorting implementation is correct


Perform unit test on your implementation with graph
Minimum Spanning Tree. Total Cost: 16



Perform graph testing  on your implementation with graph
Minimum Spanning Tree. Total Cost: 9290
You have to use OpenCV to visualize your map
```

[4]

```
max@max-envy-linux:~/Desktop/Homework/Algorithms/HW_3/HW3$ make isort
g++ -I./include/ -std=c++11 src/algorithms/mst.cpp src/graph.cpp src//interwined_sort.cpp src/sort/isort.cpp  -o bin/
isort
./bin/isort
Perform unit test on the sorting algorithm
You are using Intertwined Sort algorithm
[Test Case 1]. Performing Testing Intertwined Sort on Array of 10 elements.
[Pass Test Case 1]. Average Running Time of Your Intertwined Sort Implementation Is 0.000003539610000 Seconds

[Test Case 2]. Performing Testing Intertwined Sort on Array of 100 elements.
[Pass Test Case 2]. Average Running Time of Your Intertwined Sort Implementation Is 0.000048205780000 Seconds

[Test Case 3]. Performing Testing Intertwined Sort on Array of 1000 elements.
[Pass Test Case 3]. Average Running Time of Your Intertwined Sort Implementation Is 0.000666043770000 Seconds

[Test Case 4]. Performing Testing Intertwined Sort on Array of 10000 elements.
[Pass Test Case 4]. Average Running Time of Your Intertwined Sort Implementation Is 0.008385914430000 Seconds

[Test Case 5]. Performing Testing Intertwined Sort on Array of 100000 elements.
[Pass Test Case 5]. Average Running Time of Your Intertwined Sort Implementation Is 0.112312364750000 Seconds

[Test Case 6]. Performing Testing Intertwined Sort on Array of 1000000 elements.
[Pass Test Case 6]. Average Running Time of Your Intertwined Sort Implementation Is 16.314873336999998 Seconds

Your sorting implementation is correct
```