DESIGN AND ANALYSIS OF ALGORITHMS – CSCI 612

NAME: AADITHYA GOPALAKRISHNA BHARADWAJ

ZID: Z1862641

ASSIGNEMENT NUMBER, SEMESTER: Assignment 7, SPRING 2019

CONTENT

1.) PROBLEM STATEMENT:

In this assignment, you will implement HeapSort and RadixSort algorithms and study the time efficiency of the algorithms based on experiments. You can choose your preferred programming language. You are required to write a report that records and interpret running time of your program under different conditions. You need to submit your source file to Blackboard, and your report on the due day in class. See requirements below. 1. Your program runs on the Linux server (hopper/turing) at the Computer Science Department unless otherwise noted. If the department server does not support your programming language, you can use your own computer. But you need to specify it in your report. 2. Implementation of the sorting algorithms: You cannot use sorting library routines (e.g. qsort of C programming, sort and stable_sort of C++ programming … etc.) Use an array to store your collection of data. 3. About the program: 1) The driver program takes up to three command line arguments in form of “[N=n] [S=R/A/D] [B=Y/N] G=I/S/H/R”. The first argument specifies the size of the sorting task, i.e., the number of integers. You will use sizes of 100, 1000, 10000, 100000, 1000000, … etc. to test your program. The second argument specifies the pattern of data to be sorted: ‘A’ indicates ascending order; ‘D’ indicates descending order; ‘R’ indicates random data. The fourth argument specifies the algorithm invoked when the program executes. “[]” indicates an argument is optional. By default, N is 1000, S is ‘R’. For example, invoking the following command ./your-program N=10000 S=A G=H

will generate 10000 ascending sorted integers to be initial data and use heap sort algorithm to sort the data. And the command ./your-program G=H will generate 1000 random integers and use heap sort algorithm to sort the data. I assumed a C++ program in above examples. If yours is Java, Python, or other programming languages, you need to use proper command. 2) The third argument above is useful when heap sort is used. It specifies whether or not run buildheap only (without sorting step), which is designed to study the time cost of building a heap. 3) Preparation of your initial data: Generate the specified number of random integers if the second command line argument is "S=R" or there is no argument on this. You can use library routine(s) for this step. If the argument is "S=A", use a simple approach to produce the specified number of integers in ascending order, e.g., in sequence of "1, 2, 3, 4, 5, 6, …" Do similarly for the argument "S=D'.

2.) PROGRAMMING LANGUAGE USED AND COMMANDS USED TO COMPILE AND EXECUTE THE CODE:

a.) I used C++ Programming Language to implement the insertion and selection sort algorithm.

b.) Used the Command g++ <program name.cpp> to compile the program.

c.) Used the Command time ./a.out N= <No. of Arrya Elements> S= <Ascending/Descending/Random> G=<Insertion/Selection/Heap/Radix technique> B=Y/N (If buildheap is being used or not) to execute the code.

3.) SYSTEM INFORMATION WHERE CODE WAS EXECUTED:

a.) The code was executed in Ubuntu 18.04. The System Specification is as follows:

1.) **Processor:** Intel core i5 7th Generation, Quad Core Processor

2.) **RAM:** 8 GB Ram

3.) **Hard Disk Size:** 1 TB

4.) FINDINGS:

S.NO	No. OF ELEMENTS IN THE ARRAY	TECHNIQUE	ARRAY ORDER	REAL TIME	USER TIME	SYSTEM TIME
1	1000	INSERTION	ASCENDING	0m 0.0042s	0m 0.0041s	0m 0s
2	10000	INSERTION	ASCENDING	0m 0.0285s	0m 0.0113s	0m 0.002s
3	100000	INSERTION	ASCENDING	0m 0.2018s	0m 0.0395s	0m 0.044s
4	1000000	INSERTION	ASCENDING	0m 1.4813s	0m 1.2221s	0m 0.072s
5	1100000	INSERTION	ASCENDING	0m 1.6247s	0m 1.5015s	0m 0.081s
6	1000	INSERTION	DESCENDING	0m 0.0125s	0m 0.008s	0m 0.004s
7	10000	INSERTION	DESCENDING	0m 0.0174s	0m 0.0166s	0m 0.008s
8	100000	INSERTION	DESCENDING	0m 14.259s	0m 14.098s	0m 0.014s
9	300000	INSERTION	DESCENDING	2m 5.865s	2m 5.483s	0m 0.028s
10	600000	INSERTION	DESCENDING	3m 3.325s	3m 45.25s	0m 1.256s
11	1000	INSERTION	RANDOM	0m 0.0125s	0m 0.0055s	0m 0.004s
12	10000	INSERTION	RANDOM	0m 0.165s	0m 0.1195s	0m 0.008s
13	100000	INSERTION	RANDOM	0m 7.675s	0m 7.395s	0m 0.031s
14	300000	INSERTION	RANDOM	1m 6.265s	1m 5.436s	0m 0.0465s
15	600000	INSERTION	RANDOM	4m 18.375s	4m 17.14s	0m 2.08s
16	1000	SELECTION	ASCENDING	0m 0.0125s	0m 0.008s	0m 0.004s
17	10000	SELECTION	ASCENDING	0m 0.187s	0m 0.176s	0m 0.008s
18	100000	SELECTION	ASCENDING	0m 15.75s	0m 15.58s	0m 0.215s
19	300000	SELECTION	ASCENDING	2m 17.64s	2m 17.13s	0m 0.76s
20	600000	SELECTION	ASCENDING	5m 32.42s	5m 12.46s	0m 1.25s
21	1000	SELECTION	DESCENDING	0m 0.0115s	0m 0.008s	0m 0.003s
22	10000	SELECTION	DESCENDING	0m 0.2155s	0m 0.201s	0m 0.021s
23	100000	SELECTION	DESCENDING	0m 15.321s	0m 15.11s	0m 0.045s
24	300000	SELECTION	DESCENDING	2m 15.544s	2m 13.978s	0m 0.065s
25	600000	SELECTION	DESCENDING	7m 14.771s	7m 13.25s	0m 2.15s
26	1000	SELECTION	RANDOM	0m 0.0135s	0m 0.009s	0m 0.005s
27	10000	SELECTION	RANDOM	0m 0.251s	0m 0.0233s	0m 0.012s
28	100000	SELECTION	RANDOM	0m 15.404s	0m 15.133s	0m 0.026s
29	300000	SELECTION	RANDOM	2m 18.666s	2m 17.765s	0m 0.081s
30	600000	SELECTION	RANDOM	8m 23.429s	8m 21.12s	0m 2.45s
31	1000	RADIX	ASCENDING	0m 0.006s	0M 0.006s	0m 0s

32	10000	RADIX	ASCENDING	0m 0.014s	0m 0.007s	0m 0.002s
33	100000	RADIX	ASCENDING	0m 0.044s	0m 0.026s	0m 0.004s
34	1000000	RADIX	ASCENDING	0m 1.724s	0m 1.378s	0m 0.095s
35	1100000	RADIX	ASCENDING	0m 2.125s	0m 1.985s	0m 0.125s
36	1000	RADIX	DESCENDING	0m 0.006s	0m 0.006s	0m 0s
37	10000	RADIX	DESCENDING	0m 0.016s	0m 0.006s	0m 0.002s
38	100000	RADIX	DESCENDING	0m 0.144s	0m 0.026s	0m 0.004s
39	1000000	RADIX	DESCENDING	0m 1.724s	0m 1.38s	0m 0.110s
40	1100000	RADIX	DESCENDING	0m 2.126s	0m 1.985s	0m 0.125s
41	1000	RADIX	RANDOM	0m 0.011s	0m 0.006s	0m 0s
42	10000	RADIX	RANDOM	0m 0.056s	0m 0.017s	0m 0.004s
43	100000	RADIX	RANDOM	0m 0.214s	0m 0.041s	0m 0.008s
44	1000000	RADIX	RANDOM	0m 2.097s	0m 0.251s	0m 0.119s
45	1100000	RADIX	RANDOM	0m 2.239s	0m 0.265s	0m 0.131s
46	1000	HEAP – No BUILDHEAP	ASCENDING	0m 0.008s	0m 0.008s	0m 0.002s
47	10000	HEAP – No BUILDHEAP	ASCENDING	0m 0.017s	0m 0.015s	0m 0.004s
48	100000	HEAP – No BUILDHEAP	ASCENDING	0m 0.146s	0m 0.114s	0m 0.006s
49	1000000	HEAP – No BUILDHEAP	ASCENDING	0m 2.151s	0m 0.216s	0m 0.051s
50	1100000	HEAP – No BUILDHEAP	ASCENDING	0m 2.243s	0m 0.276s	0m 0.145s
51	1000	HEAP – NO BUILDHEAP	DESCENDING	0m 0.010s	0m 0.010s	0m 0.003s
52	10000	HEAP – NO BUILDHEAP	DESCENDING	0m 0.021s	0m 0.012s	0m 0.008s
53	100000	HEAP – NO BUILDHEAP	DESCENDING	0m 0.167s	0m 0.125s	0m 0.015s
54	1000000	HEAP – NO BUILDHEAP	DESCENDING	0m 2.236s	0m 0.256s	0m 0.078s
55	1100000	HEAP – NO BUILDHEAP	DESCENDING	0m 2.545s	0m 0.315s	0m 0.175s
56	1000	HEAP – NO BUILDHEAP	RANDOM	0m 0.011s	0m 0.006s	0m 0.002s
57	10000	HEAP – NO BUILDHEAP	RANDOM	0m 0.061s	0m 0.010s	0m 0.007s
58	100000	HEAP – NO BUILDHEAP	RANDOM	0m 0.219s	0m 0.28s	0m 0.019s
59	1000000	HEAP – NO BUILDHEAP	RANDOM	0m 2.341s	0m 0.285s	0m 0.079s

60	1100000	HEAP – NO BUILDHEAP	RANDOM	0m 2.651s	0m 0.345s	0m 0.192s
61	1000	HEAP – YES BUILDHEAP	ASCENDING	0m 0.007s	0m 0.004s	0m 0.001s
62	10000	HEAP – YES BUILDHEAP	ASCENDING	0m 0.015s	0m 0.008s	0m 0.003s
63	100000	HEAP – YES BUILDHEAP	ASCENDING	0m 0.117s	0m 0.091s	0m 0.006s
64	1000000	HEAP – YES BUILDHEAP	ASCENDING	0m 1.952s	0m 1.945s	0m 0.049s
65	1100000	HEAP – YES BUILDHEAP	ASCENDING	0m 2.216s	0m 0.219s	0m 0.132s
66	1000	HEAP – YES BUILDHEAP	DESCENDING	0m 0.009s	0m 0.008s	0m 0.003s
67	10000	HEAP – YES BUILDHEAP	DESCENDING	0m 0.019s	0m 0.010s	0m 0.006s
68	100000	HEAP – YES BUILDHEAP	DESCENDING	0m 0.161s	0m 0.108s	0m 0.011s
69	1000000	HEAP – YES BUILDHEAP	DESCENDING	0m 2.125s	0m 0.238s	0m 0.070s
70	1100000	HEAP – YES BUILDHEAP	DESCENDING	0m 2.493s	0m 0.296s	0m 0.158s
71	1000	HEAP – YES BUILDHEAP	RANDOM	0m 0.010s	0m 0.009s	0m 0.002s
72	10000	HEAP – YES BUILDHEAP	RANDOM	0m 0.056s	0m 0.009s	0m 0.005s
73	100000	HEAP – YES BUILDHEAP	RANDOM	0m 0.210s	0m 0.19s	0m 0.014s
74	1000000	HEAP – YES BUILDHEAP	RANDOM	0m 2.256s	0m 0.268s	0m 0.076s
75	1100000	HEAP – YES BUILDHEAP	RANDOM	0m 2.591s	0m 0.325s	0m 0.186s

The time complexity for **insertion sort** is:

- **Worst Case:** $O(n^2)$
- **Best Case:** $\Omega(n)$

We **cannot say insertion time cost is $\Theta(n^2)$** but **we can say worst case insertion sort time cost is $\Theta(n^2)$**

The time complexity for **selection sort is same for best case and worst case** and it is $O(n^2)$, $\Theta(n^2)$ and $\Omega(n^2)$.

The time complexity for **Radix sort is is same for best and worst cast** and it is O(nk), Θ(nk) and Ω(nk). i.e. it depends on the value of the input and based on that the time complexity increases.

The time complexity of **Heap sort is also same for best and worst case** and it is of the order O(nlogn), Θ(nlogn) and Ω(nlogn).

The time complexity for Building heap is $O(n)$. Hence we can see **when we use buildheap the time taken for sorting is less than when no buildheap is used.**

For small array (less that 20-30 elements), both insertion and selection sort are typically faster than the O(n*logn). But usually **insertion sort will perform less comparisons than the selection sort when the array elements increases.** This we can conclude from Table 1 the time taken by insertion sort is less than that of selection sort.

Insertion sort is also more stable than selection sort. Insertion sort is also In-Place as in int only requires a constant amount of O(1) of additional memory space and it can sort a list as it receives it.

Thus we can safely conclude that Insertion sort takes the least time, followed by Radix Sort, then Heap sort and the maximum time is taken by Selection sort.

From Table 1 we also inference that if the array elements are in **Ascending order then the time taken to sort be it Insertion Sort or Selection Sort is faster compared to that of Descending and Random values** and from the array elements, I compared with the Random values took more time to sort

We can also safely conclude that Real Time value is greater than the User Time value for any array elements and that the System takes the least amount of time to sort the elements.

Real Time > User Time > System Time

Note:

I have attached the Source Code along with the PDF file of this report to Blackboard. I couldn't login to hopper.cs.niu.edu as the password I tried to enter said is invalid, hence executed in my personal system.