

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

CSE 2046

ANALYSIS OF ALGORITHMS

HOMEWORK 1

Due Date: 11.05.2022

|  |  |  |
| --- | --- | --- |
|  | Student Id | Name Surname |
| 1 | 150119745 | Haydar Taha TUNÇ |
| 2 | 150119739 | Emir Ege EREN |
| 3 | 150119743 | Burak DURSUN |

PURPOSE OF THE EXPERIMENT

In this experiment we use seven different methods to find the kth smallest element in various input files. We determine the performance characteristics of these seven algorithms.

1. DESIGNING THE EXPERIMENT

First, we created 84 input files. Because the best, worst and average cases of each algorithm are unique. For each algorithm we have 4 best, 4 average and 4 worst case inputs.

1. **EXPLANATION ALGORITHM**
   1. **Insertion Sort**

Insertion Sort is a computer science sorting algorithm that produces a sorted array item by item at each step. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

-It is simple to use.

-When used on small datasets, it is quite efficient.

-It is efficient when used on arrays that are mostly already sorted.

-Sorts the array in place, using no additional memory.

-A real-life example of this sorting method is people's order of cards.

Worst Case Time Complexity: Θ(n2)

Best Case Time Complexity: Θ(n)

Average Time Complexity: Θ(n2)

* 1. **Merge Sort**

Merge sort is a general-purpose, comparison-based sorting algorithm used in computer science. Merge sort is a divide and conquer algorithm.

A merge sort works like this in theory:

1-Divide the unordered list into n sublists, each with one item (a list of items is considered sorted).

2-Combine sublists in a recursive manner to build new sorted sublists until only one sublist remains. The sorted list will be here.

Worst Case Time Complexity: Θ(n log n)

Best Case Time Complexity: Θ(n log n)

Average Time Complexity: Θ(n log n)

* 1. **Quick Sort**

Quicksort is a sorting algorithm in place. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort. Quicksort is a divide and conquer algorithm. It divides the remaining components into two subarrays based on whether they are more or less than the pivot element in the array. The substrings are then recursively sorted. This may be done in-place, with only a little amount of extra RAM required for sorting.

Worst Case Time Complexity: Θ(n log n)

Best Case Time Complexity: Θ(n)

Average Time Complexity: Θ(n log n)

* 1. **Selection Sort**

Selection sort is an in-place comparison sorting method in computer science. It has an O(n2) time complexity, making it inefficient on big lists and often outperforming the insertion sort. Selection sort is known for its simplicity, and in some instances, it outperforms more sophisticated algorithms, particularly when auxiliary memory is limited.

The algorithm divides the input list into two parts: a sorted sublist of items at the beginning (left) of the list that is built up from left to right, and a sublist of the remaining unsorted items that take up the rest of the list. The sorted sublist is initially empty, but the unsorted sublist contains the complete input list. The method then finds the smallest (or biggest, depending on sorting order) element in the unsorted sublist, swaps it with the leftmost unsorted element (sorting it), and moves the sublist borders one element to the right.

Worst Case Time Complexity: Θ(n2)

Best Case Time Complexity: Θ(n2)

Average Time Complexity: Θ(n2)

* 1. **Heap Sort**

Heapsort is a comparison-based sorting method in computer science. Heapsort is similar to selection sort in that it separates its input into a sorted and an unsorted region, then successively decreases the unsorted part by taking the biggest element from it and putting it into the sorted region. Unlike selection sort, heapsort does not waste time scanning the unsorted region in linear time; instead, heap sort keeps the unsorted region in a heap data structure to identify the biggest element in each step more rapidly.

Although it is somewhat slower in reality on most computers than a well implemented quicksort, it has a better worst-case Θ(n logn) runtime. Although heapsort is an in-place sort, it is not a stable sort.

Worst Case Time Complexity: O(n log n)

Best Case Time Complexity: O(n log n)

Average Time Complexity: O(n log n)

* 1. **Quick Select**

Quick Select is a search strategy for finding the kth smallest element in an unordered list in computer science. It works well in reality and has decent average-case performance, but it performs poorly in worst-case scenarios. The most common selection algorithms in efficient real-world implementations are Quick Select and its derivatives.

Quick Select has the same concept as quicksort in that it picks one element as a pivot and divides the data into two halves depending on the pivot, using less than and greater than values. Quick Select, unlike quicksort, just recurses into one side - the side having the element it's looking for.

Worst Case Time Complexity: Θ(n2)

Best Case Time Complexity: Θ(n logn)

Average Time Complexity: Θ(n logn)

1. **ANALYZING RESULTS**
2. **Insertion Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Insertion Sort | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,006667 | 0,006287 | 0,006477 |
| Best | 500 | 0,038625 | 0,042892 | 0,0407585 |
| Best | 1000 | 0,067208 | 0,065355 | 0,0662815 |
| Best | 10000 | 0,592959 | 0,472625 | 0,532792 |
| Worst | 100 | 0,259417 | 0,321643 | 0,290530 |
| Worst | 500 | 3,200708 | 2,783454 | 2,992081 |
| Worst | 1000 | 4,258792 | 4,839734 | 4,549263 |
| Worst | 10000 | 41,308917 | 40,972710 | 41,1408135 |
| Average | 100 | 0,082541 | 0,075639 | 0,07909 |
| Average | 500 | 3,108583 | 2,863648 | 2,9861155 |
| Average | 1000 | 3,394917 | 3,998921 | 3,696919 |
| Average | 10000 | 35,815583 | 36,341217 | 36,0784 |

To find the Best case of Insertion Sort, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

To find the Worst case of Insertion Sort, we created with sizes 100, 500, 1000, and 10000 input files sorted from largest to smallest.

To find the Average case of Insertion Sort, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

1. **Merge Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Merge Sort | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,090791 | 0,083519 | 0,087155 |
| Best | 500 | 1,170667 | 1,276418 | 1,2235425 |
| Best | 1000 | 1,304917 | 1,385427 | 1,345172 |
| Best | 10000 | 2,096291 | 2,289633 | 2,192962 |
| Worst | 100 | 0,103292 | 0,111368 | 0,10733 |
| Worst | 500 | 1,332083 | 1,197362 | 1,264225 |
| Worst | 1000 | 1,246625 | 2,156385 | 1,701505 |
| Worst | 10000 | 2,090292 | 1,875416 | 1,982854 |
| Average | 100 | 0,109292 | 0,189457 | 0,1493745 |
| Average | 500 | 1,114625 | 1,241539 | 1,178082 |
| Average | 1000 | 1,324916 | 1,289942 | 1,307429 |
| Average | 10000 | 2,7505 | 2,803448 | 2,776974 |

To find the Best case of Merge Sort, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

To find the Worst case of Merge Sort, we created with sizes 100, 500, 1000, and 10000 input files sorted from largest to smallest.

To find the Average case of Merge Sort, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

1. **Quick Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,042 | 0,084782 | 0,063391 |
| Best | 500 | 0,336917 | 0,29243 | 0,3146735 |
| Best | 1000 | 1,10325 | 1,473905 | 1,2885775 |
| Best | 10000 | 4,464167 | 5,081328 | 4,7727475 |
| Worst | 100 | 0,080791 | 0,10204 | 0,0914155 |
| Worst | 500 | 1,362917 | 1,713757 | 1,538337 |
| Worst | 1000 | 3,032209 | 2,988784 | 3,0104965 |
| Worst | 10000 | 36,1205 | 35,832517 | 35,9765085 |
| Average | 100 | 0,053542 | 0,167838 | 0,11069 |
| Average | 500 | 0,350333 | 0,429221 | 0,389777 |
| Average | 1000 | 0,643042 | 0,504142 | 0,573592 |
| Average | 10000 | 3,676042 | 4,103738 | 3,88989 |

To find the Best case of Quick Sort, we created with sizes 100, 500, 1000, and 10000 input files sorted from largest to smallest.

To find the Worst case of Quick Sort, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

To find the Average case of Quick Sort, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

1. **Selection Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Selection Sort | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,065125 | 0,083738 | 0,0744315 |
| Best | 500 | 2,181083 | 1,495282 | 1,8381825 |
| Best | 1000 | 2,645125 | 3,100304 | 2,8727145 |
| Best | 10000 | 40,4655 | 40,562012 | 40,513756 |
| Worst | 100 | 0,197417 | 0,102429 | 0,149923 |
| Worst | 500 | 5,1725 | 5,241993 | 5,2072465 |
| Worst | 1000 | 7,700375 | 7,738269 | 7,719322 |
| Worst | 10000 | 67,148667 | 66,97025 | 67,0594585 |
| Average | 100 | 0,148334 | 0,151502 | 0,149918 |
| Average | 500 | 3,829042 | 3,673097 | 3,7510695 |
| Average | 1000 | 4,214292 | 4,398001 | 4,3061465 |
| Average | 10000 | 110,955291 | 110,548224 | 110,7517575 |

To find the Best case of Selection Sort, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

To find the Worst case of Selection Sort, we created with sizes 100, 500, 1000, and 10000 input files sorted from largest to smallest.

To find the Average case of Selection Sort, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

1. **Heap Sort**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Heap Sort | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,027333 | 0,02257 | 0,0249515 |
| Best | 500 | 0,065833 | 0,059422 | 0,0626275 |
| Best | 1000 | 0,174083 | 0,217396 | 0,1957395 |
| Best | 10000 | 1,015708 | 1,152118 | 1,083913 |
| Worst | 100 | 0,132791 | 0,175209 | 0,154 |
| Worst | 500 | 0,384125 | 0,316878 | 0,3505015 |
| Worst | 1000 | 0,595416 | 0,634593 | 0,6150045 |
| Worst | 10000 | 2,200292 | 2,233079 | 2,2166855 |
| Average | 100 | 0,149916 | 0,152853 | 0,1513845 |
| Average | 500 | 0,355958 | 0,342171 | 0,3490645 |
| Average | 1000 | 0,529709 | 0,505033 | 0,517371 |
| Average | 10000 | 1,832916 | 1,890327 | 1,8616215 |

To find the Best case of Heap Sort, we created 100, 500, 1000 and 10000 input files using the same number.

To find the Worst case of Heap Sort, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

To find the Average case of Heap Sort, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

1. **Quick Select**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Select | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,020667 | 0,029412 | 0,0250395 |
| Best | 500 | 0,088375 | 0,079034 | 0,0837045 |
| Best | 1000 | 0,240458 | 0,239349 | 0,2399035 |
| Best | 10000 | 1,393666 | 1,458202 | 1,425934 |
| Worst | 100 | 0,191917 | 0,110992 | 0,1514545 |
| Worst | 500 | 3,3165 | 3,83275 | 3,574625 |
| Worst | 1000 | 5,648625 | 6,009516 | 5,8290705 |
| Worst | 10000 | 93,621291 | 93,790315 | 93,705803 |
| Average | 100 | 0,103209 | 0,163278 | 0,1332435 |
| Average | 500 | 2,657833 | 2,212639 | 2,435236 |
| Average | 1000 | 5,549209 | 5,599245 | 5,574227 |
| Average | 10000 | 89,300458 | 88,99327 | 89,146864 |

To find the Best case of Quick Select, we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

To find the Worst case of Quick Select, we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

In order to find the Average case in Quick Select, we reduced the numbers decreasing from the size of the file to 1 according to the file size, then we wrote the unwritten numbers from 1 to the input file, respectively, from smallest to largest. The sizes of these files are 100, 500, 1000 and 10000 respectively.

1. **Quick Select(Median of Three)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Select Median Of Three | size | ms-1 | ms-2 | Average |
| Best | 100 | 0,310084 | 0,304254 | 0,307169 |
| Best | 500 | 0,89575 | 0,936085 | 0,9159175 |
| Best | 1000 | 0,329583 | 0,353568 | 0,3415755 |
| Best | 10000 | 0,763667 | 0,853346 | 0,8085065 |
| Worst | 100 | 0,288875 | 0,257304 | 0,2730895 |
| Worst | 500 | 0,316958 | 0,280487 | 0,2987225 |
| Worst | 1000 | 0,690083 | 0,720517 | 0,7053 |
| Worst | 10000 | 1,057584 | 1,530416 | 1,294 |
| Average | 100 | 0,309083 | 0,396381 | 0,352732 |
| Average | 500 | 0,932292 | 1,285493 | 1,1088925 |
| Average | 1000 | 0,414708 | 0,474951 | 0,4448295 |
| Average | 10000 | 1,309084 | 1,279462 | 1,294273 |

To find the Best case of Quick Select(median of three), we created input files with sizes 100, 500, 1000, and 10000, sorted from smallest to largest.

In order to find the Worst case in Quick Select(median of three), we reduced the numbers decreasing from the size of the file to 1 according to the file size, then we wrote the unwritten numbers from 1 to the input file, respectively, from smallest to largest. The sizes of these files are 100, 500, 1000 and 10000 respectively.

To find the Average case of Quick Select(median of three), we created input files of 100, 500, 1000 and 10000 sizes, which are randomly generated between 1 and the size of the file, respectively.

The fastest working sorting algorithm for large entries is the heap sort algorithm. For small inputs, the fastest algorithm is the quick sort algorithm.

**We took the "k" value as half of the input size for all algorithms.**

**REFERENCES**

https://stackoverflow.com/

https://www.geeksforgeeks.org/

https://en.wikipedia.org/

Course materials

**Division of Labour:**

While preparing the codes of the project and the report, we connected via the online platform and we completed all the steps together by making joint decisions.