**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

**CSE2046 – Homework**

**Students:**

***150119910 - Samet Köser***

***150119651 – Şafak Gençay***

***150119908 – Mertcan Çiy***

**DECIDING ON METRICS**

Inserting counters into the program and measuring the difference between start and end time of the program are two alternative metrics to analyze algorithms. We decided to use the second option and also while using the second option, we used “seconds” for the experiment. We used Python for the experiment and in Python, there is “time” library that provides time-related functions. Firstly, we got the start time and stored it into a variable and called the algorithm function to sort the input list. Then, using time function, we calculated the difference between start and end time for our algorithms. To be clearly sure of the time results, we repeated the process for each algorithms five times and got the average of the results.

**DECIDING ON INPUTS**

1. *INSERTION SORT:*

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

* ***Best Case:***Sorted with ascending array*. Time complexity is Ω(*n*).*
* ***Average Case:*** *Random inputs. Time complexity is Θ(*n^2*).*
* ***Worst Case:***Sorted with descending array*. Time complexity O(*n^2*).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,07444206873575846333 | 0,2067049344380696633 |
| **1.** | 0,07383084297180176 | 0,20048952102661133 |
| **2.** | 0,0796506404876709 | 0,2091207504272461 |
| **3.** | 0,06984472274780273 | 0,21050453186035156 |
| **5000 (av.)** | 1,9475070635477702 | 5,349310602705269 |
| **1.** | 1,8565351963043213 | 5,380309820175171 |
| **2.** | 1,8909111022949219 | 5,542132564792315 |
| **3.** | 2,0950748920440674 | 5,125489423148321 |
| **10000 (av.)** | 7,334764771639575 | 22,9957778829904366 |
| **1.** | 7,590308427810669 | 23,11411738395691 |
| **2.** | 7,420532423654603 | 22,89875432179456 |
| **3.** | 6,993453463453453 | 22,97446194321984 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,13184356689453125 |
| **1.** | 0,14042234420776367 |
| **2.** | 0,1249847412109375 |
| **3.** | 0,12589049339294434 |
| **2499 (av.)** | 0,85632691383361816 |
| **1.** | 0,8202948570251465 |
| **2.** | 0,9495718479156494 |
| **3.** | 0,8108668327331543 |
| **5000 (av.)** | 3,44513092041015622 |
| **1.** | 3,404022455215454 |
| **2.** | 3,6381874084472656 |
| **3.** | 3,327219247817993 |
| **7499 (av.)** | 7,5562227249145508 |
| **1.** | 7,310159683227539 |
| **2.** | 8,205726146697998 |
| **3.** | 7,282616138458252 |
| **10000 (av.)** | 13,4757683753967284 |
| **1.** | 13,820802688598633 |
| **2.** | 13,350232124328613 |
| **3.** | 13,32207465171814 |

*2- MERGE SORT:*

Merge sort is a divide and conquer algorithm. Algorithm divides 2 part until elements will be single. Then, conquer part is start. It will comparison two elements and smaller one will be on the left.

* ***Best Case:***Sorted with ascending array *. Time complexity is Ω(*nlogn*).*
* ***Average Case:*** *Random inputs. Time complexity is Θ(*nlogn*).*
* ***Worst Case:***Permutation parses the array into n parts and reorganizes according to the algorithm. Worst case inputs should have 2n elements. *Time complexity O(*nlogn*).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,009946727574013437 | 0,00025098451325 |
| **1.** | 0,009979963302612305 | 0,0002531564894743 |
| **2.** | 0,009965658187866211 | 0,0002498432165496 |
| **3.** | 0,009894561231561795 | 0,0002499561654945 |
| **5000 (av.)** | 0,06018724871258166 | 0,001626648857623833 |
| **1.** | 0,06073284149169922 | 0,0016456512164942 |
| **2.** | 0,05987048149108887 | 0,0016358497431617 |
| **3.** | 0,05995842315495689 | 0,0015984456132156 |
| **10000 (av.)** | 0,1269311307578885866 | 0,035885546644238 |
| **1.** | 0,12554430961608887 | 0,0356489431215642 |
| **2.** | 0,13068413734436035 | 0,0358845313216546 |
| **3.** | 0,12456494531321654 | 0,0361231654894952 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,0096065521240234374 |
| **1.** | 0,01016092300415039 |
| **2.** | 0,009957075119018555 |
| **3.** | 0,010003328323364258 |
| **2499 (av.)** | 0,02853074073791504 |
| **1.** | 0,02692699432373047 |
| **2.** | 0,02596139907836914 |
| **3.** | 0,03789782524108887 |
| **5000 (av.)** | 0,060057401657104492 |
| **1.** | 0,05489015579223633 |
| **2.** | 0,05385422706604004 |
| **3.** | 0,0538945198059082 |
| **7499 (av.)** | 0,10054597854614258 |
| **1.** | 0,13264131546020508 |
| **2.** | 0,08381009101867676 |
| **3.** | 0,11970806121826172 |
| **10000 (av.)** | 0,123671865463256834 |
| **1.** | 0,11572670936584473 |
| **2.** | 0,14065933227539062 |
| **3.** | 0,11971807479858398 |

*3- QUICK SORT:*

*Algorithm chose a pivot and compare all element of array. If pivot bigger than other, it will be on right position, vice versa it will be on left position. In the end pivot find the it’s position. There are 4 different ways to choose pivot:*

1. *Always pick first element as pivot. (Our way)*
2. *Always pick last element as pivot*
3. *Pick a random element as pivot.*
4. *Pick median as pivot.*

* ***Best Case:***when the pivot element is the middle element*. Time complexity is Ω* (n\*log n)*.*
* ***Average Case:*** *Random inputs. Time complexity is Θ (n\*log n).*
* ***Worst Case:***when the pivot element is either greatest or smallest element*. Time complexity O(n^2).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,010942043360021666 | 0,12460853499760 |
| **1.** | 0,010131648945312 | 0,1245612313553455 |
| **2.** | 0,011152165489432 | 0,1251234534231548 |
| **3.** | 0,011542315645321 | 0,1241409202143215 |
| **5000 (av.)** | 0,020785722386583 | 0,1594896791392759 |
| **1.** | 0,020212356432153 | 0,1564894513215647 |
| **2.** | 0,021023156165465 | 0,1621564645313165 |
| **3.** | 0,021121654562131 | 0,1598231215649465 |
| **10000 (av.)** | 0,10176368107341 | 0,2983516984644771 |
| **1.** | 0,101654213456462 | 0,2945664543123165 |
| **2.** | 0,098452316544317 | 0,2989654841321651 |
| **3.** | 0,105184513219451 | 0,3015231569489497 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,0154875115144460658 |
| **1.** | 0,015491724014282227 |
| **2.** | 0,015471935272216797 |
| **3.** | 0,015485564213456218 |
| **2499 (av.)** | 0,0154901981353759762 |
| **1.** | 0,01547861099243164 |
| **2.** | 0,015452146530151367 |
| **3.** | 0,01550149917602539 |
| **5000 (av.)** | 0,043611431121826171 |
| **1.** | 0,04671216011047363 |
| **2.** | 0,07796049118041992 |
| **3.** | 0,031139373779296875 |
| **7499 (av.)** | 0,084489965438842774 |
| **1.** | 0,07800054550170898 |
| **2.** | 0,09474897384643555 |
| **3.** | 0,0779714584350586 |
| **10000 (av.)** | 0,146987438201904298 |
| **1.** | 0,15707945823669434 |
| **2.** | 0,14059114456176758 |
| **3.** | 0,124908447265625 |

*4- SELECTION SORT:*

*Firstly, first index will be pivot and algorithm compare it with all elements. The smallest one will change position with pivot. Then, pivot shifts one index.*

* ***Best Case:*** *There is no best case. Time complexity is Ω(n^2).*
* ***Average Case:*** *Random inputs. Time complexity is Θ(n^2).*
* ***Worst Case:*** *There is no worst case. Time complexity O(n^2).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,0732099706803 | 0,14875245931978 |
| **1.** | 0,0745621315465 | 0,14654978561513 |
| **2.** | 0,0725465489468 | 0,14946212354576 |
| **3.** | 0,0725212315476 | 0,15024546879845 |
| **5000 (av.)** | 1,808447590605 | 5,1476725082142033 |
| **1.** | 1,8156453123153 | 5,05845623154531 |
| **2.** | 1,8054658946532 | 5,10224564654865 |
| **3.** | 1,8042315648465 | 5,28231564654865 |
| **10000 (av.)** | 5,8083523078139 | 28,1650986601585 |
| **1.** | 5,7545231321564 | 26,1564978456123 |
| **2.** | 5,5499848435231 | 30,2156416549876 |
| **3.** | 6,1205489477622 | 28,1231564798756 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,128231573104858396 |
| **1.** | 0,959226131439209 |
| **2.** | 0,11061477661132812 |
| **3.** | 0,14048266410827637 |
| **2499 (av.)** | 0,87278318405151366 |
| **1.** | 0,9540534019470215 |
| **2.** | 0,8758912086486816 |
| **3.** | 0,8602631092071533 |
| **5000 (av.)** | 3,24810619354248048 |
| **1.** | 3,498399257659912 |
| **2.** | 3,1246910095214844 |
| **3.** | 3,268197536468506 |
| **7499 (av.)** | 7,1382178783416746 |
| **1.** | 6,959360599517822 |
| **2.** | 7,153883695602417 |
| **3.** | 6,793817043304443 |
| **10000 (av.)** | 12,3103307553648306 |
| **1.** | 12,265552043914795 |
| **2.** | 12,436438233426862 |
| **3.** | 12,531253579743726 |

*5- HEAP SORT:*

Heap sort is a transform and conquer algorithm. Firstly this algorithm builds a max heap from the input array. Then, it removes the root element from the top and adds to the last index of the sorted array one by one.

* ***Best Case:*** *All elements of array are same. Time complexity is Ω (n).*
* ***Average Case:*** *Random inputs. Time complexity is Θ (nlogn).*
* ***Worst Case:*** *Ascending order. Time complexity O(nlogn).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,07444206873575846333 | 1,0455020192176533 |
| **1.** | 0,6826596260070801 | 0,9926549458453157 |
| **2.** | 0,6841647624969482 | 0,9915354633331299 |
| **3.** | 0,6798456123154945 | 1,1523156484745143 |
| **5000 (av.)** | 18,61039911056265 | 35,733482272152130666 |
| **1.** | 18,75428080558777 | 35,967015981674194 |
| **2.** | 18,54845199488472 | 35,684949513217456 |
| **3.** | 18,52846453121547 | 35,548481321564742 |
| **10000 (av.)** | 70,7196994193787366 | 96,7402344057634226666 |
| **1.** | 70,74910640716553 | 95,513216547984531 |
| **2.** | 70,68452315684753 | 95,455612317984153 |
| **3.** | 70,72546869412315 | 99,251874351321584 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,97512168884277346 |
| **1.** | 0,12479925155639648 |
| **2.** | 0,9872536659240723 |
| **3.** | 0,8929119110107422 |
| **2499 (av.)** | 6,0155548222608608 |
| **1.** | 6,108682155609131 |
| **2.** | 5,981645894321591 |
| **3.** | 6,025489321947923 |
| **5000 (av.)** | 23,1866331559615906 |
| **1.** | 23,056489423198456 |
| **2.** | 22,984651321654868 |
| **3.** | 22,998223197512547 |
| **7499 (av.)** | 52,3437415571862062 |
| **1.** | 52,879432167456321 |
| **2.** | 52,264547236954327 |
| **3.** | 52,125879526138306 |
| **10000 (av.)** | 90,180624432528434 |
| **1.** | 90,15649486432132 |
| **2.** | 90,25894531324945 |
| **3.** | 90,44762992858887 |

*6- Quick Select Algorithm (with first input pivot):*

The partition part of the algorithm is same as that of quick sort. After the partition function arranges the elements in list according to the pivot and returns the pivot\_index, instead of recursing both sides of the pivot index, we recurse only for the part that contains our desired element. We define pivot as first element of list.

* ***Best Case:***when the pivot element is the middle element*. Time complexity is Ω* (n\*log n)*.*
* ***Average Case:*** *Random inputs. Time complexity is Θ (n\*log n).*
* ***Worst Case:***when the pivot element is either greatest or smallest element*. Time complexity O(n^2).*

***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,00065958484535 | 0,00776862117351507 |
| **1.** | 0,00065845321562132 | 0,00765123132156564 |
| **2.** | 0,00066184651864231 | 0,00784231654948471 |
| **3.** | 0,00065848453167431 | 0,00781231564949486 |
| **5000 (av.)** | 0,002497926743907516 | 0,0111461765303631866 |
| **1.** | 0,00251564948795317 | 0,01231231561948465 |
| **2.** | 0,00245647894531325 | 0,01023165184845313 |
| **3.** | 0,00252165179845613 | 0,01089456212315178 |
| **10000 (av.)** | 0,0075573282254258303 | 0,03500072617032505 |
| **1.** | 0,07564231231564483 | 0,03512313215617986 |
| **2.** | 0,07651321321561351 | 0,03498448512321576 |
| **3.** | 0,07456432123151657 | 0,03489456123157953 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,00104422569274902344 |
| **1.** | 0,0010330677032470703 |
| **2.** | 0,0010001659393310547 |
| **3.** | 0,0011928081512451172 |
| **2499 (av.)** | 0,0018997669219970703 |
| **1.** | 0,0009987354278564453 |
| **2.** | 0,003232717514038086 |
| **3.** | 0,0010170936584472656 |
| **5000 (av.)** | 0,00388083457946777344 |
| **1.** | 0,0038754940032958984 |
| **2.** | 0,004023551940917969 |
| **3.** | 0,0032110214233398438 |
| **7499 (av.)** | 0,006346082687377928 |
| **1.** | 0,004317760467529297 |
| **2.** | 0,005017995834350586 |
| **3.** | 0,008219480514526367 |
| **10000 (av.)** | 0,0152561227915538038 |
| **1.** | 0,015345612389431567 |
| **2.** | 0,014984321659746513 |
| **3.** | 0,015215648979845312 |

*7- Quick Select Algorithm (with average of first, middle and last elements):*

The partition part of the algorithm is same as that of quick sort. After the partition function arranges the elements in list according to the pivot and returns the pivot\_index, instead of recursing both sides of the pivot index, we recurse only for the part that contains our desired element. We define pivot as average of first, middle and last elements.

* ***Best Case:***when the pivot element is the middle element*. Time complexity is Ω* (n\*log n)*.*
* ***Average Case:*** *Random inputs. Time complexity is Θ (n\*log n).*
* ***Worst Case:***when the pivot element is either greatest or smallest element*. Time complexity O(n^2).*

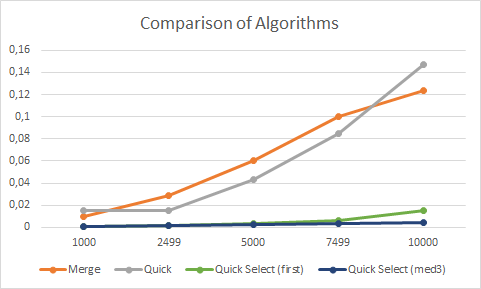
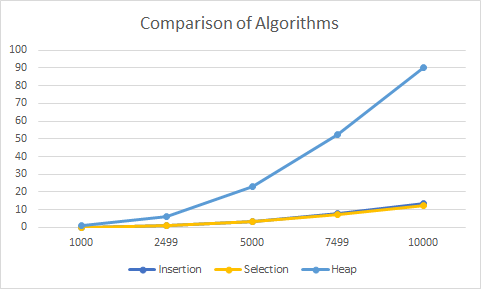
***Best and Worst Cases Output:***

|  |  |  |
| --- | --- | --- |
| **N / Case** | **Best** | **Worst** |
| **1000 (av.)** | 0,000897901072 | 0,00776862117351507 |
| **1.** | 0,0009021517945357 | 0,0035156494848435 |
| **2.** | 0,0008915649484651 | 0,0032547984151328 |
| **3.** | 0,0008981564748531 | 0,0035423135456865 |
| **5000 (av.)** | 0,0019560067319644 | 0,0111461765303631866 |
| **1.** | 0,0020202315649453 | 0,010612315648465 |
| **2.** | 0,0020021654654656 | 0,011564568795432 |
| **3.** | 0,0018456231654823 | 0,011645487984531 |
| **10000 (av.)** | 0,004619649450730133333 | 0,0221241923554753333 |
| **1.** | 0,0045231561945623 | 0,022156498453123 |
| **2.** | 0,0046868468453127 | 0,022231546489756 |
| **3.** | 0,0046489453123154 | 0,021984532123547 |

***Average Case Output:***

|  |  |
| --- | --- |
| **N / Case** | **Average** |
| **1000 (av.)** | 0,00102868178021393538 |
| **1.** | 0,0009975433349609375 |
| **2.** | 0,0010215632195138146 |
| **3.** | 0,0011564891235321569 |
| **2499 (av.)** | 0,00140252113342285152 |
| **1.** | 0,0009980201721191406 |
| **2.** | 0,002024412155151367 |
| **3.** | 0,0019936561584472656 |
| **5000 (av.)** | 0,0026003360748290998 |
| **1.** | 0,0030050277709960938 |
| **2.** | 0,002992391586303711 |
| **3.** | 0,00198793411254882 |
| **7499 (av.)** | 0,00380764007568359372 |
| **1.** | 0,003018617630004883 |
| **2.** | 0,004017829895019531 |
| **3.** | 0,003972291946411133 |
| **10000 (av.)** | 0,00479402542114257808 |
| **1.** | 0,005010843276977539 |
| **2.** | 0,0040204524993896484 |
| **3.** | 0,004987001419067383 |

*Comparison of Algorithms:*



***Conclusion***

*In this experiment, our aim is to compare the speed and efficiency of the 7 algorithms given to us in different conditions. First of all, we researched the working principles and cases of all algorithms. We have added the explanations in detail with the big-oh notations of the algorithms to the report. Then, for our purpose, we tested the worst, best and average cases of all algorithms 3 times with the necessary inputs. The input sizes we use are 1000, 2499, 5000, 7499 and 10000 for the average case, and 1000, 5000 and 10000 for the best and worst case. In the average case, we used 2499 and 7499 as odd input sizes because there was a exception warning in the book, “For example, if all the sizes in a sample are even and your algorithm runs much more slowly on oddsize inputs, the empirical results will be quite misleading.”, This is why we used them. Also, we added the results to the report and got their averages. and we plotted these averages to compare under different conditions. As you can see in the chart, when we compare the algorithms, we can see that they are in the following order from fast to slow: Quick select with median of 3, Quick select with first elements, Quick sort, Merge Sort, Selection Sort, Insertion Sort and heap sort .*