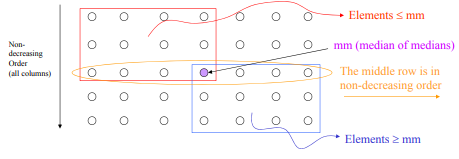
# Introduction

Given a list of n numbers, the Selection Problem is to find the kth smallest element in the list. The first approach, Select 1, simply uses a Merge Sort operation to sort the list of numbers. From there, Select 1 returns the kth element in the list. Select 2, on the other hand, utilizes the partitioning procedure from Quick Sort. With some pivot, it partitions the elements less than the pivot to the left of the pivot position and the elements that are greater than the pivot to the right of the pivot position. The process is repeated until the pivot position is in the kth element’s position after partitioning. The kth element, that is now the same as the pivot position, is returned. Select 2 repeats the partitioning procedure iteratively until it reaches the correct position. Select 3 works the same way, however it instead calls the partitioning procedure recursively. Select 4 follows a very similar pattern to Select 2 and Select 3, however it instead attempts to choose a better pivot. By creating subsets of the original array and then finding the medians of those arrays, we can finally get the median of those medians (MM) and use it as our pivot. Theoretically, this should perform the partitioning more optimally.



# Data Sets, Test Strategies, and Explanation of Results

To test the different implementations of the Selection Problem, I separated them into their own classes and ran through test cases with each. I had set arrays of integer elements ranging from 1 to n for the sizes 10, 50, 100, 250, 500, and 1000. Each array did not contain duplicate elements and was not altered between tests to maintain accurate results. This allowed for easy timing and allowed me to achieve quick confirmation that the algorithms were performing correctly. For the trials, I tested each value of n twice with the kth element being equal to 1, n/4, n/2, 3n/4, and n. Each value of n resulted in 10 timings which were then averaged to give a final result. The goal with the testing was to capture accurate results and to cover different cases within each array since the quick sort partitioning is not consistent with every case.

Select 1 used the Merge Sort O(nlogn) implementation, and so I was not expecting there to be much variance within each size of n as the algorithm would perform the same regardless of what k was equal to. The results followed expectations and gave me a good base-line to compare the other implementations to.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_1 Timings (nanoseconds)** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
| k=1 | 337352 | 348160 | 361245 | 460800 | 641138 | 1008072 |
|  | 329956 | 336782 | 357262 | 464214 | 700302 | 966543 |
| n/4 | 314026 | 339058 | 366365 | 464214 | 622933 | 972801 |
|  | 340196 | 327112 | 356693 | 451698 | 639432 | 905671 |
| n/2 | 316303 | 332801 | 354418 | 447716 | 630898 | 1093404 |
|  | 331094 | 333369 | 362382 | 459662 | 621796 | 888036 |
| 3n/4 | 316303 | 337351 | 373760 | 460232 | 711680 | 940374 |
|  | 327111 | 335076 | 377742 | 455680 | 647965 | 827733 |
| n | 311752 | 331663 | 362951 | 452836 | 633174 | 942650 |
|  | 342471 | 329387 | 365227 | 455111 | 620089 | 936392 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_1 averaged results** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
|  | 326656.4 | 335075.9 | 363804.5 | 457216.3 | 646940.7 | 948167.6 |

Select 2 was the first implementation that utilized the partitioning procedure from quick sort. I was expecting the timing to be faster than Select 1 in some cases and slower in others, depending on the pivot. The best case should be O(n) when the pivot is the kth smallest. The worst case would call the partition n times, meaning that it would instead be O(n2).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_2 timings (nanoseconds)** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
| k=1 | 329956 | 328249 | 332800 | 335076 | 342471 | 367503 |
|  | 319715 | 329387 | 338489 | 337920 | 335644 | 373191 |
| n/4 | 315733 | 328817 | 341334 | 339057 | 345884 | 397653 |
|  | 317439 | 323698 | 331094 | 340764 | 376605 | 391395 |
| n/2 | 320854 | 336214 | 324835 | 340765 | 386276 | 403911 |
|  | 351573 | 329387 | 326542 | 333369 | 389120 | 440320 |
| 3n/4 | 315733 | 333369 | 324267 | 335076 | 374330 | 417564 |
|  | 324835 | 321991 | 324836 | 335644 | 367502 | 420409 |
| n | 327112 | 320853 | 328818 | 339626 | 350436 | 392533 |
|  | 320285 | 320285 | 325973 | 335645 | 355556 | 394809 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_2 averaged results** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
|  | 324323.5 | 327225 | 329898.8 | 337294.2 | 362382.4 | 399928.8 |

As you can see, Select 2 performed similarly to Select 1 at the lower values of n but then greatly outperformed Select 1 at higher n values.

Select 3 was expected to act the same as Select 2 with the exception that it solved the problem recursively. My expectation was that the recursive call would potentially slow down the timing as it would need extra time to handle the stack frame.

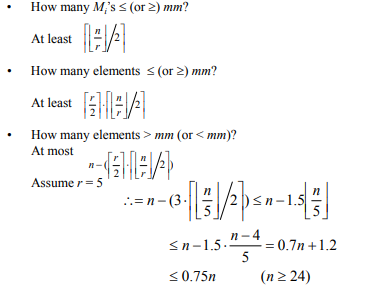
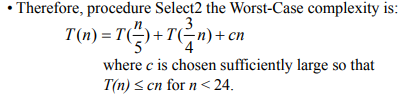
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_3 timings**  **(nanoseconds)** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
| k=1 | 288995 | 300373 | 295823 | 306062 | 303217 | 309476 |
|  | 286151 | 293547 | 294116 | 299805 | 306631 | 310613 |
| n/4 | 287858 | 288427 | 298098 | 302649 | 306631 | 341334 |
|  | 290702 | 294685 | 297529 | 295253 | 307769 | 348729 |
| n/2 | 294115 | 304925 | 291271 | 301512 | 319715 | 340195 |
|  | 290133 | 304925 | 293547 | 302080 | 319716 | 343040 |
| 3n/4 | 287858 | 296391 | 310614 | 304924 | 336213 | 338489 |
|  | 291271 | 294116 | 304356 | 302080 | 332232 | 332231 |
| n | 290703 | 286720 | 290133 | 298666 | 321422 | 343609 |
|  | 294116 | 288995 | 296391 | 316871 | 322560 | 346453 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Select\_3 averaged results** | **n=10** | | **50** | **100** | **250** | **500** | **1000** |
|  | 290190.2 | 295310.4 | | 297187.8 | 302990.2 | 317610.6 | 335416.9 |

Surprisingly, Select 3 performed better than both the Select 2 and Select 1 implementations. It should be noted though, that Select 3 was still very close in timings to Select 2. At higher values, the timings may differ and the recursive part could potentially slow it down much more.

Select 4 was the final implementation of the Selection problem. This implementation would include a segment that would first find the median of medians, and then proceed to use that index as the pivot. The goal with this final selection method was to optimize the pivot selection which would theoretically allow the partition to perform much better.

Analysis:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_4 timings**  **(nanoseconds)** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
| k=1 | 516551 | 558649 | 563769 | 2045156 | 2143574 | 2418347 |
|  | 515982 | 548978 | 564907 | 2046863 | 2169743 | 2417778 |
| n/4 | 512569 | 538169 | 585956 | 2045156 | 2092943 | 2773904 |
|  | 505174 | 538738 | 565475 | 2028089 | 2084410 | 2412089 |
| n/2 | 518827 | 530205 | 601315 | 2042880 | 2171449 | 2424605 |
|  | 519965 | 547271 | 566044 | 2033209 | 2168605 | 2486045 |
| 3n/4 | 582542 | 538737 | 694614 | 1989405 | 2170881 | 2405263 |
|  | 521671 | 554098 | 558649 | 1985423 | 2125939 | 2392748 |
| n | 512000 | 539307 | 647396 | 2022400 | 2141868 | 2381938 |
|  | 534756 | 547271 | 561494 | 2042881 | 2123663 | 2379663 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Select\_4 averaged timings** | **n=10** | **50** | **100** | **250** | **500** | **1000** |
|  | 524003.7 | 544142.3 | 590961.9 | 2028146 | 2139308 | 2449238 |

As you can see, the Select 4 method performed the worst out of all the select methods. At no point was this method faster than the others and it did not appear to be closing the gap very quickly, if at all.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Averaged Times (nanoseconds)** | **10** | **50** | **100** | **250** | **500** | **1000** |
| Select\_kth\_1 | 326656.4 | 335075.9 | 363804.5 | 457216.3 | 646940.7 | 948167.6 |
| Select\_kth\_2 | 324323.5 | 327225 | 329898.8 | 337294.2 | 362382.4 | 399928.8 |
| Select\_kth\_3 | 290190.2 | 295310.4 | 297187.8 | 302990.2 | 317610.6 | 335416.9 |
| Select\_kth\_4 | 524003.7 | 544142.3 | 590961.9 | 2028146 | 2139308 | 2449238 |

# Select 2 Vs Select 3

As stated previously, the biggest difference between these two was the way they handled calling the partition method. Select 2 used an iterative approach, partition calls inside a loop, to continue to call partition until it reached the kth element. Select 3, on the other hand, recursively called partition until it hit the kth element. What surprised me is that the recursive version not only looked more elegant but also seemed to run faster.

The results of the two were very close however it looks like Select 3 would always be less. I cannot fully explain why that would be and so I would need to be able to test this at higher values to be sure.

# Select 4 Vs Select 1

Looking at the difference between Select 4 and Select 1 results left me quite astonished. I was expecting to maybe see some similarity in results, however the Select 4 results were far inferior.

Going into the project, I had an idea that the Select 4 would not be better since the theoretical comparisons do not really account for the overhead of finding the median of medians. Although asymptotically it may appear to be the better method, in practice it failed miserably.

# Strengths and Constraints

My implementation of this project was done in Java. I used simple arrays to hold the values, rather than needlessly adding the functionality of lists or other methods. I did my best to follow the pseudocode given to us, however I’m sure there are some areas of my code that could be optimized. For instance, when reaching a base case in Select 4, I sort the array (array length < r) using an insertion sort. Perhaps it would be faster to use a different sort, although I did not do any testing for that small part.

My testing could have been more extensive to get clearer results. Although I was able to reach test cases of n=1000, there may be surprising results at higher test cases. I limited myself to the size of 1000 to better handle the arrays that would be used. If I were to use random values inside the arrays and to not worry about the arrays being consistent, I could easily test at higher values of n.

Overall, I am satisfied with the results of this project and have learned a lot in the process. Going further, I hope to understand the effect that recursion has better and to gain more knowledge as to why theoretical complexities do not always match up with real world results.