**Course 20407**

**This project was made by Max Freeman Vadim Volkov According**

**to the assumptions for submitting this project in Section 3 the work environment we used was Intellji, running on JDK 18.0.1.1, and as java for the Windows environment We used version: JAVA 8 333. Our study of a comparison of two different sorting algorithms Summary:** **In this project we will compare the performance of k smallest numbers and sorted numbers in the array by implementing a stack-building algorithm and minimum extract against the implementing of random selection and quick sorting:**

Solution to Question 1: We will first analyze the runtime complexity for the first algorithm which is actually BuildMinHeap and ExtractMinHeap. At first the algorithm is actually the algorithm for building the pile whose running time complexity is O (n) (according to book of the course which is named introduction to algorithms by authors Thomas H. cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, we will add link to the book below). Then we taking out all the k(smallest) numbers of the pile sitting at the root and for each individual expense of the root, immediately afterwards we perform ExtractMinHeap algorithm which running time complexity is O (logn) (according to page 109 in the book course mention above of MinHeapify's algorithm), so we can To conclude that the running time complexity for removing the smallest numbers from the pile and performing MinHeapify algorithm is . We will also note that the MinHeapify algorithm, is not a good case or a bad case because it is blocked up by O (logn) and also question 6.2-6 from the book in page 110 we learn that it is Ω (logn). If we look at the routine analysis for stack sorting, we first see that a minimum stack building algorithm is executed, like ours task asked us to do and then there is a loop for running from the original array size n to 2 and each time switching between the last and first index, reduces the array and then it calls the MinHeapify algorithm to root , But this is very similar to performing ExtractMinHeap n times because lines 4-6 in the algorithm on page 116 are the same as lines 3-5 on page 113, so in our task conditions when we find using ExtractMinHeap k times, it springs from a particular case of partially sorting the stack. Summary We have seen that the implementation of the BuildMinHeap algorithms followed by ExtractMinHeap k times is equivalent to performing a stack sort, whose running time complexity is Ο (nlogn) and even if we perform the stack sort partially, we can still block the complexity by Ο (klogn) and by question 6.4-4 in the book page 115 we conclude that sort The stack in the worst case is also Ω (nlogn) and for sorting the stack partially, we can still block the complexity by Ω (klogn) in total because BuildMinHeap followed by ExtractMinHeap k times equal to performing a stack sort we will conclude that the runtime complexity of the first algorithm implementation is: Θ (klogn).

We will now analyze the second algorithm which is actually RandomizedSelect and QuickSort. At the beginning the algorithm is actually RandomizedSelect whose runtime complexity for the worst case and the good case is Ο (n) and at worst case it is at Ο (n ^ 2), according to the book. We then run the fast sorting algorithm from the beginning of the array to the smallest k numbers and the running time complexity of the fast sorting at best and average is Ο (klogk) and at worst is Ο (k ^ 2), passing the array A [0,… .., k]. Summary We can register the running time complexity of the general algorithm RandomizedSelect and QuickSort, for finding the smallest k numbers for the best case and average case as: O (n) + Ο (klogk) = Ο (max (n, klogk)). Also the running time complexity of the general algorithm the RandomizedSelect and QuickSort, for finding the smallest k numbers for the worst case as: Ο (n ^ 2) + Ο (k ^ 2) = Ο (max (n ^ 2, k ^ 2) ).

Solution to question C2 in the project: We have made 2 drivers for testing our algorithms and their performance in separate file, the two drivers for each of the algorithms, in which we run a thousand different tests on random inputs with arrays of different sizes and with a different amount of k numbers, and as can be seen the first algorithm BUILD-MIN-HEAP is more efficient in terms of complexity and running time stability And in terms of the amount of comparisons it makes compared to the second RANDOMIZED-SELECT algorithm depending on n and k, it can be seen that as k approaches size of n which is the size of the array so the difference between them increases in terms of running time efficiency and the amount of comparisons each makes, much more significant and significant To the difference between the amount of comparisons each makes because the second algorithm after its sorting still calls for a QuickSort algorithm which we know has the worst running time Complexity of O (n ^ 2) One of the reasons we can assume that the closer k increases to n the smaller the subset of the minimum numbers increases and thus the chance that a fast sort function will encounter more "bad cases" for it which increases the amount of comparisons, so we think if we want to use an algorithm we can predict the amount of comparisons Consistently then we would probably like to choose the BUILD-MIN-HEAP algorithm, however as we learned the course number on pages 155-157 it is possible to arrive at running time complexity at best and at average case in the RANDOMIZED-SELECT O (n) algorithm, as the number of k numbers we have is significantly smaller relative to n of the empire results of running the algorithms we have interviewed that more to us rather use the second algorithm, for example (can be run and tested using the driver files we have created and will be attached in a separate file to the project):

The print was:  
  
The BuildMinHeap and ExtractMin average comparison count for a 100 array and 50 smallest integers is: 843

The BuildMinHeap and ExtractMin average comparison count for a 200 array and 50 smallest integers is: 1213

The BuildMinHeap and ExtractMin average comparison count for a 500 array and 50 smallest integers is: 2092

The BuildMinHeap and ExtractMin average comparison count for a 1000 array and 50 smallest integers is: 3438

The BuildMinHeap and ExtractMin average comparison count for a 2000 array and 50 smallest integers is: 6027

The BuildMinHeap and ExtractMin average comparison count for a 4000 array and 50 smallest integers is: 11102

The BuildMinHeap and ExtractMin average comparison count for a 6000 array and 50 smallest integers is: 16139

The BuildMinHeap and ExtractMin average comparison count for a 8000 array and 50 smallest integers is: 21146

The BuildMinHeap and ExtractMin average comparison count for a 10000 array and 50 smallest integers is: 26162

The Random Select and QuickSort average comparison count for a 100 array and 50 smallest integers is: 832

The Random Select and QuickSort average comparison count for a 200 array and 50 smallest integers is: 1108

The Random Select and QuickSort average comparison count for a 500 array and 50 smallest integers is: 1783

The Random Select and QuickSort avarage comparison count for a 1000 array and 50 smallest integers is: 2867

The Random Select and QuickSort avarage comparison count for a 2000 array and 50 smallest integers is: 4963

The Random Select and QuickSort avarage comparison count for a 4000 array and 50 smallest integers is: 4998

The Random Select and QuickSort avarage comparison count for a 6000 array and 50 smallest integers is: 13449

The Random Select and QuickSort avarage comparison count for a 8000 array and 50 smallest integers is : 17336

The Random Select and QuickSort avarage comparison count for a 10000 array and 50 smallest integers is: 21451

As we can be seen when the array size was 100 for sorting the 50 smallest numbers, the difference between the two algorithms The difference is not significant at all and is very minor, but when the array size started to increase relative to those 50 small k numbers then we had to sort there was a significant advantage to RANDOMIZED-SELECT, as can be seen above. To summarize our conclusion from our study we will prefer RANDOMIZED-SELECT algorithm when we want to look for a small amount of k minimums relative to n and we will prefer BUILD-MIN-HEAP algorithm when we want predetermined performance regardless of the minimum n and k array size as opposed to RANDOMIZED- The SELECT they performed decreases significantly when k is close to n(in his size) as we showed above.