**Name: Yicheng Jia GWid: G24072171 GWmail:** [**jiayicheng111@gwmail.gwu.edu**](mailto:jiayicheng111@gwmail.gwu.edu)

**Option A:** Quick select, deterministic (Median of Medians) median finding

**Problem Description:**

Median finding is given an **unsorted** array of numbers and find the median number(or the *k*th smallest number). The simple approach is to sort the array and find the middle or the ***k***th element which requires *O(nlogn*) time. Now we are going to find a more efficient algorithm based on divide and conquer algorithms.

**Analysis:**

As mentioned above, given an array A=A[1,…,n] of n numbers and an index i(1≤i≤n), we can simply sort the entire array and find the *k*th smallest number. However, this approach seems to be overkill because it is not necessary to know all the order statistics. In order to prove the plausibility of a more efficient algorithm, it is instructive to consider the ideas like pivot strategy presented in **quicksort** algorithm to design divide and conquer algorithms to find the middle element in linear *O(n)* time.

According to quicksort algorithm, it chooses a random pivot to partition the list into elements less than and greater than the pivot, and calls itself recursively in one of the two sublists. This algorithm has a worst case performance of O() because of the big difference between two sublists. In our median finding algorithm, we can do something similar but chooses the pivot in a more complicated way to ensure sublists’ size relatively comparable by searching the median of medians(pivot), which can be done in linear (*O(n)*) time.(Note: Explanation about time complexity is mentioned in the post. )

Then we can arrive at the median-of-medians algorithm in the following clever way.

1. Divide the array A into sublists of length five. There are n/5 sublists. (Note : the last sublist may have length less than five.)
2. Sort each sublist using insertion sort and determine its median directly.
3. Collect all the n/5 medians from n/5 groups.
4. Use the median of medians algorithm recursively to determine median of the set of all medians from previous step.
5. Use the median of the medians from step 4 as the pivot to partition the array.
6. Invoke the algorithm recursively on the left or the right partition depending on the value of **k** and the size of partition.

The key to this algorithm is that it guarantees the pivot is not very far from the true value. Now we are going to find the upper bound of numbers which is smaller or larger than our pivot. There are about half of the groups(n/10 groups) have at least 3 elements smaller or larger than the median of medians and as a result, thus we can remove at least 3n/10 elements from array. That is to say, we can just remain at most 70% of the array. Besides that there are about 40% of the elements which relationship with pivot is not sure. In conclusion, median of medians algorithm guarantee partitions cannot be too lopsided and leads to linear run time.

**Program Introduction:**

In order to find the ***kt***h smallest element recursively, we have to design a function to deal with all kinds of array and implement it in the main function. Thus I have designed a *find* function and several assistant functions to realize our goal.

The design of *find* function is mainly composed of a pivot-finding algorithm, an element-counting algorithm and a partition algorithm. In the pivot-finding algorithm, we divide the array into n/5 sublists and sort every sublist by using selection sort. After that the algorithm swap every sublists’ middle element with the front element in the whole array in order to both form an updated array and retain the full data. This process will be executed until there is only one element left in the updated array. The last remaining element is the median of medians(pivot).

Then *count* function is executed to count the number which is smaller and larger than the pivot. Finally, according to the counting number, the algorithm determines where the median number of array locates and remove useless elements by swapping them to the back of the original array.

The whole function will execute recursively until the value of k is equal to pivot which we have found by breaking the while loop.

**Program results:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N value** | **Input** | **K value** | **Output(correctness)** | **Average time cost** |
| **8** | {4,2,1,7,3,8,6,9} | **5** | **6(√)** | **62** |
| **16** | {4,2,1,7,5,3,8,6,9,4,4,4,9,9,9,9} | **8** | **5(√)** | **137** |
| **32** | {4,2,1,7,5,3,8,6,9,4,4,4,9,9,9,9,  10,11,15,16,17,18,19,20,28,27,  26,25,23,22,24,21}; | **26** | **22(√)** | **290** |
| **64** | **Not shown** | **40** | **16(√)** | **576** |
| **128** | **Not shown** | **92** | **19(√)** | **1138** |

Note: The average time cost does not have units since every element’s time cost is counted by a counter in the For loop or While loop.

**Time complexity:**

We have observed that steps 1, 2, 3 and 5 require linear amount of time. Step 4 is a recursive call to execute the median finding function. Step 6 is another recursive call after updating the array and removing at least 3n/10 of elements in the last array. Thus, the recurrence relation can be written as:



According to the substitution method, we hypothesis that  for all values of . Therefore, , ,.

Finally, in the previous chart and following graph we can also substantiate that time cost is in proportion to the length of array A.

