Hoare's Quickselect Algorithm

Description

Quickselect, short for Hoare's selection algorithm, is a recursive algorithm, which finds the *k*th smallest element in an unordered list. It is related to the sorting algorithm from the same developer Tony Hoare (Quicksort) and thus uses a similar approach. Usually, this algorithm is coded as an in-place algorithm, so no second list is initialized. The whole procedure takes place in the one given list.

➢ Approach

Given an unordered list and an integer k ($\leq n$), for the desired kth smallest element, the algorithm decides on a pivot. There are different strategies for choosing a pivot. For this project, the pivot is chosen randomly from the (sub)list. Other strategies use medians, or even more complex algorithms to get a pivot (for example the Floyd-Rivest algorithm). The pivot element is then swapped to the right end of the (sub)list.

With the pivot element, the list is grouped into elements smaller and elements greater than the pivot element. This procedure is called partition and happens with two pointers, which traverse the (sub)list from the left and from the right. The left one searches for elements greater than the pivot while the right one searches for those, which are smaller than the pivot. If each pointer has found one, the two elements are swapped, and the search continues until the pointers meet each other. As soon as this happens the pivot element is swapped back to the correct place in the list. Now, the list contains two sublists. One with smaller elements and one with greater elements than the pivot and in between the pivot element Itself.

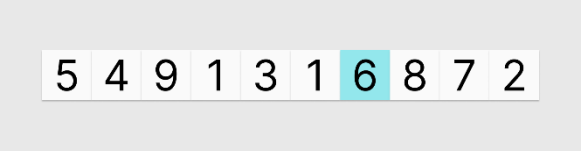
The next step is to call the same method with one of the sublists. To decide which one, the algorithm evaluates the new place of the pivot element. Holds the desired element a lower ranking number, it must be in the left sublist and vice versa with a greater ranking number. Again, a new random pivot inside the sublist is chosen and the two pointers traverse it and rearrange the elements inside. The other list is ignored, unlike in the quicksort algorithm, where both sublists must be ordered.

This loop continues until the correct place of one of the chosen pivots matches k. In that case, the corresponding element is returned.

➢ Example

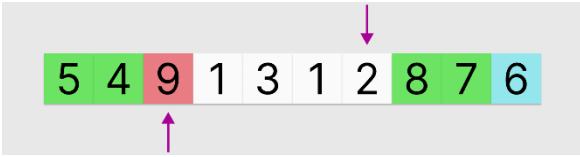
The following example illustrates the procedure with an unordered list. It starts with an unordered list {5,4,9,1,3,1,6,8,7,2} and we want the 3rd smallest element ($k = 3$).

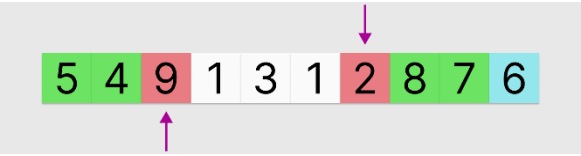
- A random pivot is chosen and moved to the right end of the list (pivot = 6). Then the partition is called with the limit indices 0 and 8 and the pivot element 6.

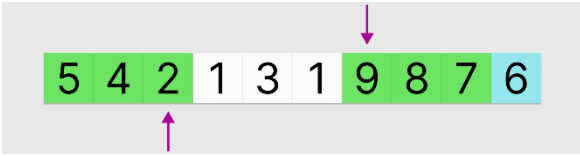


- The left pointer traverses the list until it finds an element greater than the pivot.

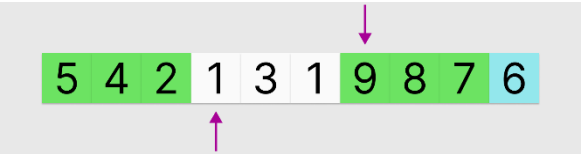
- The right pointer traverses the list until it finds an element smaller than the pivot. The two misplaced elements are swapped.

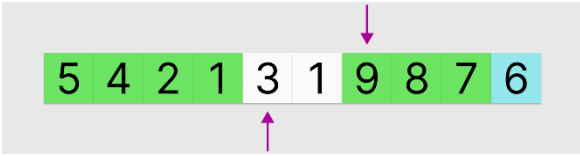


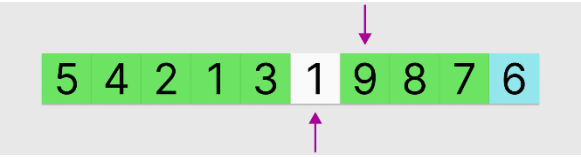


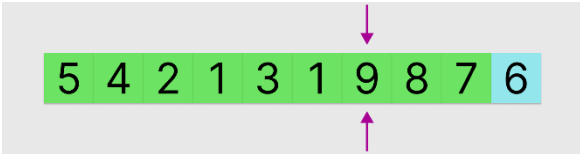


- The left pointer continues its search until a new greater element is found or the two pointers meet each other.

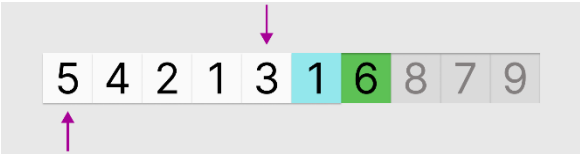








- The pivot is then moved to the correct place (where the left pointer has stopped). The sublist is decided on (in this case the left one, because (pivot index + 1 > *k).* A new random pivot is chosen (1) and the sublist is partitioned with new pointers.



- This recursive procedure lasts until the replaced pivot index equals the desired *k*.

- When the pivot index equals *k* after partitioning, that element is returned (in this case, the index of element 2 + 1 equals our k = 3). So, the third-smallest element in this list is the 2.

Pseudocode

For this chapter, the algorithm is written in pseudocode and the individual methods are explained.

```
*** pseudocode for Quickselect algorithm; author Jorma Steiner ***
*********************************************************************


Algorithm select(unsortedlist, k)
      list <- unsortedlist

      return recSelect(0, list.size - 1, k)


Algorithm recSelect(leftIndex, rightIndex, k)


      if leftIndex = RightIndex
            return list[leftIndex]

      pivot <- random int between leftIndex and rightIndex
      pivot <- partition(leftIndex, rightIndex, pivot)

      if pivot + 1 = k
            return list[pivot]
      else if pivot + 1 < k
            return recSelect(pivot + 1, rightIndex, k)
      else
            return recSelect(leftIndex, pivot -1, k)


Algorithm partition(lowEnd, highEnd, pivotIndex)

      pivotvalue <- list[pivot]
      swap(pivotIndex, highEnd)
      i <- lowEnd - 1
      j <- highEnd

      loop always
            do i++ while list[i] < pivotvalue

            do j-- while list[j] > pivotvalue

            if j <= i
                  swap(i, highEnd)
                  return i

            swap(i,j)
```

➢ select(unsortedlist, k)

This method is the starting point for the Quickselect algorithm. It is called with a list and an Integer k as parameters. The method then calls the recursive method *recSelect* with both ends of the list and k as parameters. The outcome will then be returned.

➢ recSelect(leftIndex, rightIndex, k)

This method chooses as pivot index of a sublist and calls the method *partition* with both ends of the sublist and the pivot index. The sublist is defined by the given indices, which represent the left and right edges. If the sublist has only one element, the method returns that one element.

After partitioning a list with more than one element, the returned pivot index is compared to *k*. Either *recSelect* is called again with a new sublist or the element at this index is returned. The element is returned only if pivot index + 1 and *k* are equal.

➢ partition(lowEnd, highEnd, pivotIndex)

In this method, two pointers traverse the list, each from one side, and search for greater/smaller elements than the element at the pivot index. But first, the pivot element is moved to the end of the (sub)list.

The left pointer will begin looking for a greater element. It does so by traversing from left to right and by comparing the current element with the pivot element. The right pointer does the same but from right to left and while searching for smaller elements. Two found elements are swapped unless the pointers have already met each other. In this case, the element at the left pointer is swapped with the pivot element itself, and its new index is then returned. The end state after the *partition* is a (sub)list with a pivot element, which is at the exact place inside the whole list if it is ordered. Left from the pivot element are only elements smaller and right of it are the greater ones.

Running Time Analysis

➢ Average case

For the average case, we look at the primitive operations inside the pseudocode for a given list with size *n*. We can say that for the average case, each iteration of the recursive method divides the list into two sublists with the same size or similar sizes (in case *n* is an even number and the pivot element is in between)

The partitioning method visits every single element in the given (sub)list either with the left pointer or the right pointer. Therefore, the running time for the partitioning algorithm is $T(n) = nc_1 + c_2$ with c_1 as the number of operations in one iteration of the loop and c_2 as the number of operations outside the loop. Both *c*'s are constant.

Because each *recSelect* call divides the list in the middle, we get a running time looking like $T(n) = T(n/2) + nc_1 + c_2 + c_3$ with $nc_1 + c_2$ from the partitioning method and c_3 as the number of operations inside the recursive method. All three c_i are constant.

Using the table from the right we get a "Divide" algorithm, *q = 1, r = 2* and *f(n) = O(n).* Therefore, the average case should have a complexity of *O(n).*

			$f(n)$			
			$O(1)$	$O(\log n)$	$O(n)$	$O(n^2)$
R1	$q = 1$	$k \geq 1$	n	$n \cdot \log n$	n^2	n^3
R2	$q \geq 2$	$k \geq 1$	q^n	$\log n \cdot q^n$	$n \cdot q^n$	$n^2 \cdot q^n$
D1	$q = 1$	$r \geq 2$	$\log_r n$	$\log n$	n	n^2
D2	$q \geq 2$	$r = q$	n	n	$n \cdot \log_r n$	n^2
D3	$q \geq 2$	$r > q$	$n^{\log_r q}$	$n^{\log_r q}$	n	n^2
D4	$q \geq 2$	$r < q$	$n^{\log_r q}$	$n^{\log_r q}$	$n^{\log_r q}$	n^2

➢ Worst case

For the worst case, we also look at the primitive operations inside the pseudocode for a given list with size *n*. The worst case occurs when the algorithm must find the smallest (greatest) element in an ordered list and the chosen pivots are always the biggest (smallest) one of the sublists. So, each partitioning reduces the list size by only one element (the pivot itself).

The partitioning method has still the same running time as for the average case. $T(n) = nc_1 + c_2$

If each iteration reduces the list only by one element, the method *recSelect* will be called *n* times. The running time for the recursive method is $T(n) = T(n-1) + nc_1 + c_2 + c_3$ with $nc_1 + c_2$ from the partitioning method and c_3 as the number of operations inside the recursive method. All three c_i are constant.

Using the table from above we get a "Decrease" algorithm, q = *1, k = 1* and *f(n) = O(n).* Therefore the worst case has a complexity of $O(n^2)$.

To minimize the risk of the worst-case scenario, for each iteration we use a random pivot index for partitioning.

Performance Test

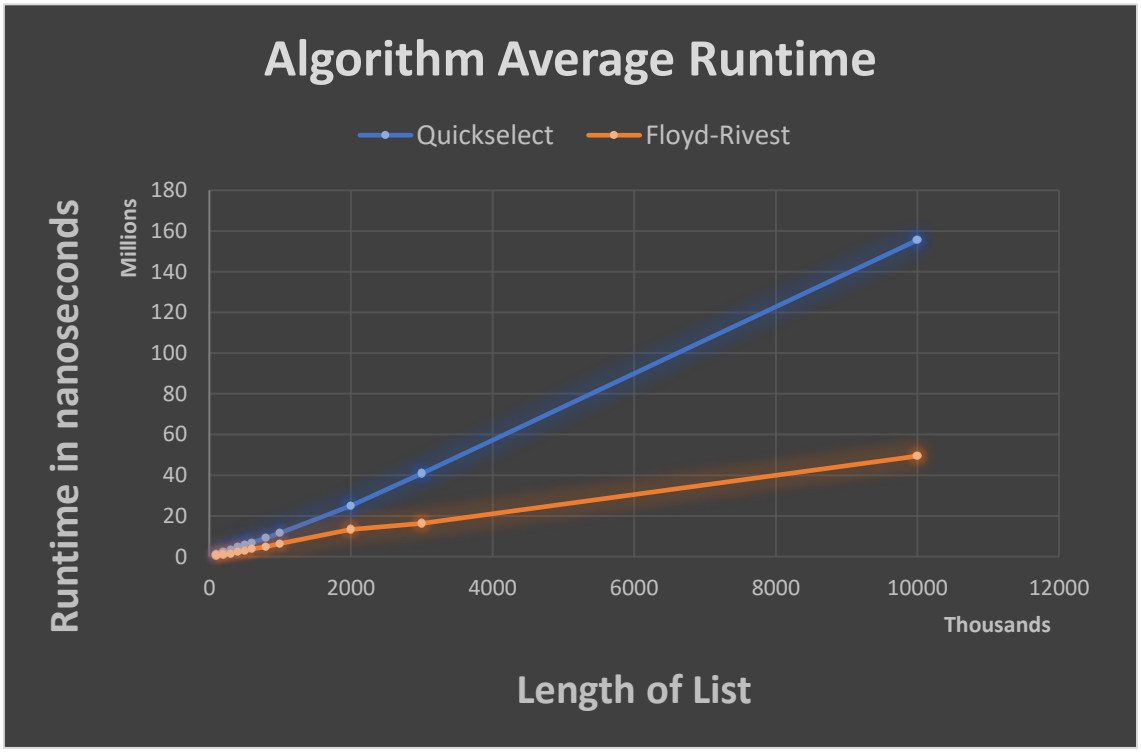
After implementing the Quickselect algorithm a performance test was conducted with large random inputs. The tests used different lengths of the list but each one was executed a thousand times by calling the method *select(list, k)*. The algorithm always searched for a number with a random integer $0 < k \leq list.size()$. For each test scenario the average running time in nanoseconds was taken and combined into a chart.

Length of List	Average time in nanoseconds (Floyd-Rivest)	Average time in nanoseconds (Quickselect)
100'000	573'786	1'117'165
200'000	1'006'531	2'273'478
300'000	1'448'670	3'320'099
400'000	2'506'870	4'894'131
500'000	2'753'065	5'971'132
600'000	3'813'217	6'932'565
800'000	4'836'398	9'113'753
1'000'000	6'429'235	11'591'132
2'000'000	13'351'171	25'034'377
3'000'000	16'403'723	40'941'604
10'000'000	49'410'088	155'500'351

The chart shows us that the running time for the Quickselect algorithm is approximately linear to the size of the list. It corresponds with the theoretical running time from above.

The same scenario was also run with an improved algorithm. It's called Floyd-Rivest algorithm and it uses the same principles like Quickselect but utilizes two pivot elements at the same time. It partitions the list not only into two but three sublists. For the project, an implementation from the geeksforgeeks.org website was adapted and used. The complexity of the average case is the same as Quickselect *O(n)*[1].

Although both Select algorithms have the same complexity we can see that Floyd-Rivest runs much faster. For example, Quickselect needs 0.15 seconds on average for a list with ten million random numbers. Floyd-Rivest instead only needs 0.049 seconds on average. Over all the conducted performance tests Floyd-Rivest is around 2.15 times faster than Quickselect. For more information on how Floyd-Rivest functions please refer to geeksforgeeks.org.



[1] https://www.geeksforgeeks.org/floyd-rivest-algorithm/ Visited: 18.01.2023