DUA:

Konstantin Krasser

October 23, 2024

Aufgabe 1 (9 points). In the lecture, you learned about the partition function. This function can not only be used to sort an array A (as in quicksort), but also to find the k-th smallest value in A, i.e., the k-th element in the ascending sorted order of the elements in A. Example: In the array $A = [6, 7, 2, 9, 3, 1, 0]$, the 4th smallest element is the number 3 (only 0, 1, and 2 are smaller). We assume that every element in A is unique.

1. Answer to question 1:

   What would a naive approach, using comparison-based search algorithms, look like to find the k-th smallest element in A? What is the lower asymptotic bound on the runtime of this approach?

   The naive approach would be to sort the array and then return the k-th element. The time complexity for comparison-based sorting algorithms is:

   $$O(n \log n) \tag{1}$$

2. Answer to question 2:

   (2 points) Provide a modified partition function in pseudocode that rearranges the array A such that the pivot element is at position i_p, and all elements to the left of i_p are smaller than the pivot element, and all elements to the right of i_p are greater than the pivot element. The manipulation of A is done in-place, so all changes are made directly in A, and partition does not need to explicitly return the array A, only i_p.

```
function partition(A, low, high)
    pivot = A[high]
    i_p = low - 1
    for j = low to high - 1 do
        if A[j] <= pivot then
            i_p = i_p + 1
            swap(A[i_p], A[j])
    swap(A[i_p + 1], A[high])
    return i_p + 1
```

3. Answer to question 3:

   (4 points) Describe in words and in pseudocode how the modified partition function can be used to efficiently find the k-th smallest value in A.

The modified partition function can be used to find the k-th smallest element in an array A by using a quickselect algorithm. We choose a pivot and partition the array around it. If the pivot index i_p equals k, we are done. If $i_p > k$, recursively search in the left subarray, and if $i_p < k$, recursively search in the right subarray.

```
function quickselect(A, low, high, k)
    if low == high:
        return A[low]
    i_p = partition(A, low, high)
    if i_p == k:
        return A[i_p]
    else if i_p > k:
        return quickselect(A, low, i_p - 1, k)
    else:
        return quickselect(A, i_p + 1, high, k)
```

4. Answer to question 4:

   (2 points) What is the runtime of your algorithm in the best case and in the worst case? Provide an example call for both cases. The best/worst case should apply to general k, not a specific k.

   The best-case runtime occurs when the pivot always splits the array evenly, giving a time complexity of $O(n)$, since each partition step reduces the problem size by half. The worst-case runtime occurs when the pivot is always the smallest or largest element, leading to a time complexity of $O(n^2)$ because each partition only reduces the problem size by one element.

   Example best-case: $quickselect([1, 2, 3, 4, 5], 0, 4, 2)$.

   Example worst-case: $quickselect([5, 4, 3, 2, 1], 0, 4, 2)$.