Applied Algorithms

Written Assignment-3

Jaya Sandeep Ketha

October 2023

Q1. Think back to your programming assignment (1), where you implemented randomized and deterministic quicksort. You got wildly different running times for the deterministic version depending on the input list. Now, consider the following deterministic version of the selection algorithm: pick the pivot to be the first element of the list. I would like you to find the median element of a given list using this algorithm. What do you think will be the running time if the list is sorted? Give in the big-Oh notation, and explain why.
Answer:

The deterministic quick sort of assignment 1 is modified to obtain median.
When an input is sorted array, we can observe the quadratic nature of the running time vs array size.

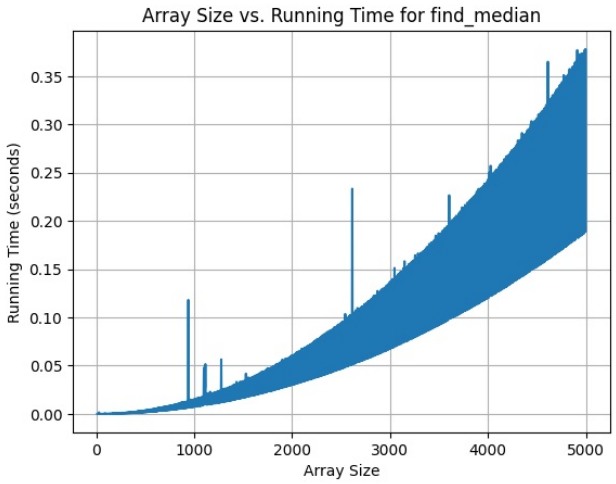

Figure 1: Array Size vs Running time - Quadratic

If we use a deterministic version of the selection algorithm that always picks the first element of the list as the pivot to find the median element of a given list, the running time when the list is sorted will still be $O(n^2)$. Because:

1. In the worst case, when the list is sorted (either in ascending or descending order), selecting the first element as the pivot will always result in unbalanced partitions. This means that the pivot will consistently be either the smallest or largest element in the unprocessed portion of the list.

2. In each recursive step, we will partition the list into two sublists, one with n-1 elements and one with just the pivot element.

3. The pivot element, being the smallest or largest, will not help us in finding the median efficiently because it does not provide a meaningful division of the elements.

4. As a result, we will essentially be making n-1 recursive calls to find the median of the n-1 elements in the sublist. Each of these recursive calls will have a time complexity of O(n-1), and this pattern will continue until we have processed all elements in the list.

5. The recurrence relation for the worst-case time complexity will be T(n) = T(n-1) + O(n-1), where T(n) is the time it takes to find the median of an n-element list.

To solve this recurrence relation, we can see that:

$$\begin{aligned} T(n) &= T(n-1) + O(n-1) \\ &= (T(n-2) + O(n-2)) + O(n-1) \\ &= T(n-2) + O(n-2) + O(n-1) \\ &= \dots \end{aligned}$$

Continue this expansion until we reach the base case, which is T(1) or T(0), and we'll find that we have:

$$T(n) = O(1) + O(2) + O(3) + ... + O(n-1)$$

This is an arithmetic series, and the sum of the first $n-1$ natural numbers is:

$$\frac{(n-1)\cdot(n-1+1)}{2} = \frac{(n-1)\cdot n}{2} = \frac{n(n-1)}{2}$$

Therefore, the time complexity of this deterministic selection algorithm when applied to a sorted list is:

$$\begin{aligned} T(n) &= O(1+2+3+\ldots+(n-1)) \\ &= O\left(\frac{(n-1)\cdot n}{2}\right) \\ &= O\left(\frac{n^2-n}{2}\right) \\ &= O\left(\frac{n^2}{2} - \frac{n}{2}\right) \\ &= O(n^2 - \frac{n}{2}) \\ &= O(n^2) \end{aligned}$$

So, the running time of this deterministic selection algorithm when the list is sorted is $O(n^2)$, which is the same as the worst-case time complexity for the quicksort algorithm when always choosing the first element as the pivot. This occurs because the unbalanced partitions result in many recursive calls and, therefore, a quadratic time complexity.

Q2. Assume that I give you a deterministic algorithm A for finding the median in a list (assume that you will be given an unsorted list of length n, and the median is the value of the element of rank n/2). The running time of this algorithm is $O(n)$. Describe in pseudocode a deterministic algorithm that works by making calls to A to find the kth smallest element in the list. Explain in words why this should work, and analyze the asymptotic running time. You will get full score for an $O(nlogn)$ time algorithm, 3 bonus points for an $O(n)$ one.
Answer:

Code:

Algorithm 1 Finding the kth Smallest Element in an Array

function kthSmallest($arr[0..n-1]$, k)

Step 1: Divide $arr[]$ into $\lceil n/5 \rceil$ groups where the size of each group is 5, except possibly the last group, which may have fewer than 5 elements.

Step 2: Find the median of all $\lceil n/5 \rceil$ groups using the function A given. Create an auxiliary array $median[]$ and store the medians of all $\lceil n/5 \rceil$ groups in this $median$ array.

Recursively call this method to find the median of $median[0..\lceil n/5 \rceil - 1]$.

Step 3: $medOfMed \leftarrow$ kthSmallest($median[0..\lceil n/5 \rceil - 1]$, $\lceil n/10 \rceil$)

Step 4: Partition $arr[]$ around $medOfMed$ and obtain its position.

$pos \leftarrow$ partition($arr, n, medOfMed$)

if $pos == k$ **then**

Step 5: return $medOfMed$

else if $pos > k$ **then**

Step 6: return kthSmallest($arr[low.....pos-1], k$)

else

Step 7: return kthSmallest($arr[pos+1.....high], k-pos+low-1$)

end if

end function

```
# Given function to find median of arr[] from index low to high
def Algorithm_A(arr, low, high):
    ....................

def kthSmallest(arr, low, high, k):
    # If k is smaller than number of elements in array
    if (k > 0 and k <= len(arr)):
        # Number of elements in arr[low,...high]
        n = len(arr)

        # Divide arr[] in groups of size 5, calculate median of every group
        #and store it in median[] array.
        median = []

        i = 0
        while (i < n // 5):
            median.append(Algorithm_A(arr, low + i * 5, 5))
                i += 1

        # For last group with less than 5 elements
        if (i * 5 < n):
                median.append(Algorithm_A(arr, low + i * 5, n % 5))
                i += 1

        # Find median of all medians using recursive call.
        # If median[] has only one element, then no need of recursive call
        if i == 1:
            medOfMed = median[i - 1]
        else:
                medOfMed = kthSmallest(median, 0, i - 1, i // 2)

        # Partition the array around a medOfMed element and get position of
        # pivot element in sorted array
        pos = partition(arr, low, high, medOfMed)
```

```
        # If position is same as k
        if (pos - low == k - 1):
            return arr[pos]
        if (pos - low > k - 1): # If position is more, recur for left subarray
                return kthSmallest(arr, low, pos - 1, k)

        # Else recur for right subarray
        return kthSmallest(arr, pos + 1, high, k - pos + low - 1)

        # If k is more than the number of elements in the array
        return -1

def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp

# It searches for x in arr[low..high], and partitions the array around x.
def partition(arr, low, high, x):
    for i in range(low, high):
        if arr[i] == x:
            swap(arr, high, i)
            break

    x = arr[high]
    i = low
    for j in range(low, high):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, high)
    return i
```

Time Complexity Analysis:
The worst-case time complexity of the above algorithm is $O(n)$.
Let us analyze all steps.

Steps (1) and (2) take $O(n)$ time as finding the median of an array of size 5 takes $O(1)$ time and there are $n/5$ arrays of size 5.

Step (3) takes $T(n/5)$ time.

Step (4) is a standard partition and takes $O(n)$ time.

The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. The worst case size is the maximum number of elements greater than medOfMed (obtained in step 3) or the maximum number of elements smaller than medOfMed.

Reference: (https://en.wikipedia.org/wiki/Median_of_medians)

The median-calculating recursive call does not exceed worst-case linear behavior because the list of medians has size $n/5$, while the other recursive call recurses on at most 70% of the list.

Let T(n) be the time it takes to run a median-of-medians Quickselect algorithm on an array of size n. Then we know this time is:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + c \cdot n$$

where the $T\left(\frac{n}{5}\right)$ part is for finding the true median of the $\frac{n}{5}$ medians, by running an (independent) Quickselect on them (since finding the median is just a special case of selecting a k-smallest element).

The $O(n)$ term $c \cdot n$ is for the partitioning work to create the two sides, one of which our Quickselect will recurse (we visited each element a constant number of times in order to form them into $\frac{n}{5}$ groups and take each median in $O(1)$ time).

The $T\left(\frac{7n}{10}\right)$ part is for the actual Quickselect recursion (for the worst case, in which the k-th element is in the bigger partition that can be of size $\frac{7n}{10}$ maximally).

$$T(n) \leq \frac{cn}{5} + c\left(\frac{7n}{10}\right) + O(n)$$

$$T(n) \leq \frac{9cn}{10} + O(n)$$

$$T(n) \leq cn$$

The worst-case running time is therefore linear.

Q3. I give you n elements in the following format: you have $\sqrt{n}$ small arrays of length $\sqrt{n}$ each. Each small array is unsorted, but you know that all elements in Array i are greater than all elements in Array i1. For instance, for n = 25, Array 1 could consist of 3, 2, 4, 1, 8, and Array 2 could have 21, 12, 13, 18, 20. Array 2 could not have 7 as 7 is smaller than 8, which is a member of Array 1. How would you find the kth smallest element out of all n elements? Give a short explanation and pseudocode; analyze asymptotic running time. Your running time should be less than linear.

Answer:
Approach:
The key idea is to follow median of medians approach to search on the range of possible values for the kth smallest element in a 1D array.

Algorithm 2 Function KTH_SMALL(arr, low, $high$, k)

```
1: if low = high then
2:     return arr[low]
3: end if
4: pivot_index ← PARTITION(arr, low, high, low)          ▷ Find the pivot index
5: if k = pivot_index then
6:     return arr[k]
7: else if k < pivot_index then
8:     return KTH_SMALL(arr, low, pivot_index − 1, k)    ▷ Recursively search the left partition
9: else
10:    return KTH_SMALL(arr, pivot_index + 1, high, k)   ▷ Recursively search the right partition
11: end if
```

Algorithm 3 Function FIND_KTH_SMALLEST_IN_2D(a, k)

$n \leftarrow \text{length}(a) \times \text{length}(a[0])$ ▷ Calculate the total number of elements in the 2D array
if $k \leq 0$ or $k > n$ **then**
    return None ▷ Invalid k value
end if
$array\_idx \leftarrow \lfloor (k-1)/\text{length}(a[0]) \rfloor$ ▷ Calculate the row index of the selected element
if $array\_idx \geq \text{length}(a)$ **then**
    return None ▷ k is too large
end if
$sel\_arr \leftarrow a[array\_idx]$ ▷ Get the selected row from the 2D array
$inner\_idx \leftarrow k - (array\_idx \times \text{length}(a[0])) - 1$ ▷ Calculate the index within the selected row
return KTH_SMALL(sel_arr, 0, $\text{length}(sel\_arr) - 1$, $inner\_idx$) ▷ Find the kth smallest element within the selected row

1. Partition Function:
- This function is responsible for partitioning the array around a pivot element.
- A pivot element is selected, and its value is stored.
- The pivot element is moved to the end of the subarray.
- Two pointers, 'i' and 'j', are used to iterate through the subarray.
- The 'i' pointer keeps track of elements less than the pivot, and when a smaller element is found at 'j', it is swapped with the element at 'i'.
- This process effectively places all elements less than the pivot to the left of 'i'.
- Finally, the pivot element is placed in its correct position by swapping it with the element at 'i'.
- The pivot's final index, 'i', is returned.

2. Median of Medians Function:
- This function finds the median of medians, which is an estimate of the overall median of the array.
- It first checks for the base case where there's only one element in the array and returns that element.
- The array is divided into subarrays of a fixed size (e.g., subarray size of 5).
- The medians of these subarrays are calculated recursively by selecting a middle element.
- The median of medians is calculated by recursively calling 'kth_smallest_element' to find the median of the medians.
- The median of medians is used as the pivot element for the partitioning process.
- The partitioned index 'pivot_index' is determined.
- If 'k' is the same as the pivot index, the function returns the element at that index, indicating that the kth smallest element has been found.
- If 'k' is less than the pivot index, the function recurses on the left subarray, otherwise on the right subarray.

3. Kth Smallest Element Function:
- This function is a variation of the quickselect algorithm, which is used to find the kth smallest element in a subarray.
- It starts with a pivot element chosen (in this case, the first element of the subarray).
- The partitioning process places the pivot element in its correct position in the sorted order.
- Depending on whether 'k' is less than, equal to, or greater than the pivot index, the function recursively calls itself on the left subarray, returns the pivot element if 'k' matches the pivot index, or recurses on the right subarray.
- This process repeats until 'k' matches the pivot index, which indicates that the kth smallest element has been found.

4. Find Kth Smallest in 2D Function:
- This function is the entry point for finding the kth smallest element in a 2D array.
- It calculates the total number of elements 'n' by multiplying the number of rows by the number of columns.
- It checks for invalid input where 'k' is less than or equal to 0 or greater than 'n'.

- It determines the array index and inner index corresponding to the ‘k‘ value.
- The selected 1D array containing the kth smallest element is determined, and the inner index is calculated within that array.
- Finally, it calls ‘kth_smallest_element‘ on the selected 1D array to find the kth smallest element.

The Median of Medians algorithm is used to guarantee worst-case linear time complexity, and the entire process ensures that the kth smallest element is efficiently found in a 2D array.

Python Code:

```
import math
import random

def partition(arr, low, high, pivot_index):
    pivot_value = arr[pivot_index]
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    i = low
    for j in range(low, high):
        if arr[j] < pivot_value:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]
    return i

def med_of_med(arr, low, high, k):
    if low == high:
        return arr[low]

    sub_size = 5
    num_subarrays = (high - low + 1) // sub_size

    medians = []
    for i in range(num_subarrays):
        sub_low = low + i * sub_size
        subarray_high = sub_low + sub_size - 1
        if subarray_high > high:
            subarray_high = high
        median_index = low + i * sub_size + (subarray_high - sub_low) // 2
        medians.append(med_of_med(arr, sub_low, subarray_high, median_index))

    median_of_medians_index = kth_small(medians, 0, len(medians) - 1, len(medians) // 2)

    pivot_index = arr.index(median_of_medians_index)
    pivot_index = partition(arr, low, high, pivot_index)

    if k == pivot_index:
        return arr[k]
    elif k < pivot_index:
        return kth_small(arr, low, pivot_index - 1, k)
    else:
        return kth_small(arr, pivot_index + 1, high, k)

def kth_small(arr, low, high, k):
    if low == high:
        return arr[low]
```

```
    pivot_index = partition(arr, low, high, low)

    if k == pivot_index:
        return arr[k]
    elif k < pivot_index:
        return kth_small(arr, low, pivot_index - 1, k)
    else:
        return kth_small(arr, pivot_index + 1, high, k)

def find_kth_smallest_in_2d(a, k):
    n = len(a) * len(a[0])
    if k <= 0 or k > n:
        return None

    array_idx = int((k - 1) // len(a[0]))

    if array_idx >= len(a):
        return None  # k is too large

    sel_arr = a[array_idx]

    inner_idx = k - array_idx * len(a[0])
    inner_idx -= 1

    return kth_small(sel_arr, 0, len(sel_arr) - 1, inner_idx)
```

Asymptotic Running Time Analysis:
The find_kth_smallest_in_2d function selects particular array which can contain kth smallest element so it has O(1) time complexity. The kth smallest element from an array can be found from given array using same approach as in Q2. As median of medians approach is employed which has generally O(n log n) in worst case scenario or O(n) in most cases, where n is number of elements in array. But in this case we have $\sqrt(n)$ elements in each array, so the time complexity is $(O(\sqrt(n)\log(\sqrt(n)))$ in worst case, or $O(\sqrt(n))$ in most cases.
$(O(\sqrt(n)\log(\sqrt(n))) \leq$O(n)

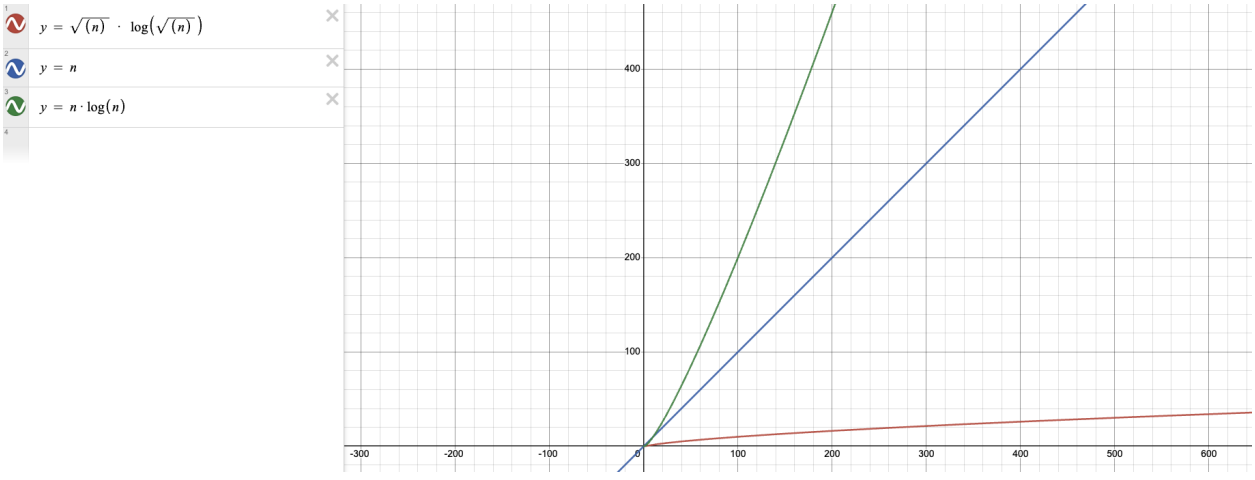

Figure 2: Comparision of running times.

Q4. In the above question, assume that I'd like to sort all of the n elements into one big array and I do it as follows: I sort each small array using randomized quicksort, and combine the resulting lists into one sorted array of length n. Asymptotically speaking, does this give you a better running time than com- bining them first into a big array of length n, then using randomized quicksort on the big array? Argue by analyzing the running times of both algorithms.

Answer:
To analyze the asymptotic running times of the two approaches, let's denote the following variables:

- 'n': the total number of elements

Approach 1:
Sorting each small array using randomized quicksort and then combining them into one sorted array:

1. Sorting each small array of size $\sqrt(n)$ using randomized quicksort:
The time complexity of quicksort is typically $O(nlogn)$, but in the worst case, it can be $O(n^2)$. However, since it is mentioned as using randomized quicksort, the expected time complexity for each small array can be approximated as $O((\sqrt(n)) * log(\sqrt(n)))$.

2. And $\sqrt(n)$ subarrays needs to be sorted giving raise to $O(\sqrt(n) * \sqrt(n) \log(\sqrt(n)))$. So its $O(nlog\sqrt(n))$.

3. Combining the sorted arrays:
Merging $\sqrt(n)$ sorted arrays of sizes $\sqrt(n)$ takes $O(n)$ time.

So, the overall time complexity for Approach 1 is approximately $O(n * log(\sqrt(n))) + O(n)$.

Approach 2:
Combining all elements into a big array of length n and then using randomized quicksort:

1. Combining all elements into a big array of length n takes $O(n)$ time.
2. Sorting the big array of length n using randomized quicksort: The expected time complexity for quicksort on an array of size n is $O(n * log(n))$.

So, the overall time complexity for Approach 2 is $O(n) + O(n * log(n))$.

Now, let's compare the two approaches in terms of their asymptotic behavior:
Approach 1:
- Sorting small arrays: $O(n) * log(\sqrt(n))$
- Combining small arrays: $O(n)$

Total time complexity for Approach 1: $O(n * log(\sqrt(n)) + O(\sqrt(n))$

Approach 2:
- Combining all elements: $O(n)$
- Sorting the big array: $O(n * log(n))$

Total time complexity for Approach 2: $O(n) + O(n * log(n))$

Comparing the two approaches, we can see that the time complexity of Approach 2 is dominated by the sorting step ($O(n * log(n))$), whereas the time complexity of Approach 1 is dominated by the sorting of small arrays ($O(n * log(\sqrt(n))$.

From intuition it is clear that approach 1 has better performance than approach 2 considering values of n in time complexities.