Theory Assignment-1: ADA Winter-2024

Aditya Sharma - 2022038 Ayan kumar Singh - 2022122

27-01-2024

1 Preprocessing

1. The algorithm assumes **0 indexed arrays** for different calculations.
2. The algorithm assumes the arrays given are already sorted in themselves in **ascending order**, the correctness of the algorithm depends on this assumption.
3. The algorithm assumes that the inputs provided i.e. arr1, arr2, arr3 are non-empty and the pointers for these arrays and the value of k is within the acceptable ranges.

2 Algorithm Description

The algorithm utilizes a recursive binary approach to eliminate one half of an array in each call.

Terminologies used:

1. `arr1`, `arr2`, `arr3`: Three arrays of the same size initially passed to the `kth()` function.
2. `end1`, `end2`, `end3`: Pointers to the end of `arr1`, `arr2`, `arr3` respectively.
3. `mid1`, `mid2`, `mid3`: Middle index of `arr1`, `arr2`, `arr3` respectively, calculated as $\texttt{mid}(i) = \frac{\texttt{end}(i) - \texttt{arr}(i)}{2}$ (Note: Integer division is used).

Functions used:

1. `int kth_for_two(int *arr1, int *arr2, int *end1, int *end2, int k)`: Base case until one array reaches length zero, then calls `kth_for_two()`.
2. `int kth(int *arr1, int *arr2, int *arr3, int *end1, int *end2, int *end3, int k)`: Recursive function considering three arrays, eliminating one array at each step based on the comparison of middle elements.

Now we try to understand how our algorithm works step by step :

1. `int kth_for_two(int *arr1, int *arr2, int *end1, int *end2, int k)`. Three arrays are passed to this function with starting and ending pointers for all three and k (the k-th smallest element that has to be found in the union of all three arrays).
2. Calculation of `mid1`, `mid2`, and `mid3`, followed by the relationship between k and the summation of `mid1 + mid2 + mid3`.

```
if (k < mid1 + mid2 + mid3)
    This means that we can eliminate the second half of an array out of three.
    Three arrays represented using curly brackets { }
    " | " represents the middle index of the array
    Marking each half of the array using .1 for the first half and .2 for the second half
    For example: 1.1 means the first array's first half

    { [1.1] | [1.2] } { [2.1] | [2.2] } { [3.1] | [3.2] }
    let the three arrays be represented using curly brackets { }
    " | " represents the middle index of the array
    And, we have marked each half of the array using .1 for the first half and .2 for the second
    half e.g., 1.1 means the first array's first half
    Now, we find
    Wlog of generality,
    let,  arr(i)[mid(i)] = max(arr1[mid1], arr2[mid2], arr3[mid3])
    in which i belongs to = {1,2,3};

    This means that all the elements in the other two first halves of the other two arrays
    will be less than arr(i)[mid(i)]
    and, we also know that all the elements in [i.1] will be <= arr(i)[mid(i)]
    We also know that all the elements in [i.2] will be >= arr(i)[mid(i)]
    Therefore, we can infer that the kth smallest element will not be in [i.2]
    So, we safely remove [i.2] and recursively call the function with new parameters

    So our new arrays will be:
    {  [.1]  |  [.2]   } {  [.1]    |    [.2]   }   {  [i.1]  }
    Other two arrays will remain as same but i.2 is removed.

else if (k >= mid1 + mid2 + mid3) this means that we can eliminate first half of an array
out of the three arrays
    let this be the arrays we have initially,
    {  [1.1]  |  [1.2]   } {  [2.1]    |    [2.2]   }   {  [3.1]    |    [3.2]    }

    Now, we find
        arr(i)[mid(i)] = min(arr1[mid1], arr2[mid2], arr3[mid3])
        in which i belongs to = {1,2,3};

        For sake of better understanding let us assume that i = 2 i.e second array.
        We know, that all the array are sorted in themselves.
        Therfore, all the elements in [2.1] are <= arr[mid2]
        And as arr[mid2] = min(arr1[mid1], arr2[mid2], arr3[mid3])
        we can safely ignore first half of array 2 that is [2.1]

        So, our new arrays will be:
        {  [1.1]  |  [1.2]   } {   [2.2]   }   {  [3.1]    |    [3.2]    }
        Other two arrays will remain as same but 2.1 is removed.
```

3. Calculation of k:
   (a) When we remove the second half of an array, there is no need to modify k; i.e., k remains unchanged.
   (b) When we remove the first half of an array, we have to modify k as now we are not looking for the k-th smallest element in the reduced problem, but rather for $\text{arr}[(k - \text{mid}(i) - 1)]$ in the final array. Therefore, $k \to (k - \text{mid}(i) - 1)$ (where i corresponds to the removed array).
4. Therefore, in each recursive call, we are removing either the first or second array based on the value of k. We will reach a base case when an array size is reduced to zero. This means that now,
   - When one of the arrays has reached size zero, we will be left with only two arrays to work with. For this, we have a helper function `int kth_for_two` that follows the same principles and logic but is designed to work for two arrays.

   Base Case for `kth()`:

   - When one of the arrays has reached size zero, we call `kth_for_two()` for the other two arrays as parameters.

Base Case for `kth_for_two()`:

- When one of the arrays has reached size zero, we return the k-th element of the other array.

Main Function:
This is a demo main function that will be used to call the function `kth()` to find the k-th smallest element in equi-sized arrays (5 in this case).

```
int main()
{
    int arr1[5] = {1, 2, 3, 4, 5};
    int arr2[5] = {6, 7, 8, 9, 10};
    int arr3[5] = {11, 12, 13, 14, 15};

    int k = 10;

    // Assuming 0th based indexing, therefore passing k-1 to the parameter to get kth element
    cout << "Kth smallest element is:" << kth(arr1, arr2, arr3,arr1 + 5, arr2 + 5, arr3 + 5, k - 1);

    return 0;
}
```

3 Recurrence Relation

The relation $T(n)$ is for kth_for_two

$$T(n) = T\left(\frac{3n}{4}\right) + c$$

The relation $T'(n)$ is for kth

$$T'(n) = T\left(\frac{5n}{6}\right) + d$$

4 Complexity Analysis

There are two functions in our algorithm kth_for_two and kth which both have different recurrence relations. The recurrence relation relation for kth_for_two is:

$$T(n) = T\left(\frac{3n}{4}\right) + c$$

The recurrence relation relation for kth is:

$$T'(n) = T\left(\frac{5n}{6}\right) + d$$

Now, solving for kth_for_two using the master theorem comparing to

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

here now $a = 1$, $b = \frac{4}{3}$, $f(n) = c$ and the general solution is

$$T(n) = n^{\log_b a} \cdot U(n)$$

Solving this, our equation becomes

$$T(n) = n^{\log_b a} \cdot U(n) = n^0 \cdot U(n) = U(n)$$

here the value of $U(n)$ depends on $h(n)$ and $h(n) = \frac{f(n)}{n^{\log_b a}}$

Calculating $h(n)$ by substituting the value of $f(n) = c$ and $n^{\log_b a} = n^{\log_{\frac{4}{3}} 1} = n^0 = 1$
Now, $h(n) = \frac{c}{1} = c$. But looking at the cases of the master theorem $h(n) = \log_2 n \cdot c$
Using the 3rd case of the master theorem, $U(n)$ becomes $\frac{\log_2 n^{0+1}}{0+1}$, $\implies$

$$U(n) = \log_2 n \cdot c$$

Now, $T(n) = U(n) = \log_2 n \cdot c$
ignoring constants, $T(n) = \log_2 n$, and therefore, the time complexity of the algorithm becomes

$$T(n) = O(\log_2(n))$$

Similarly, again using master theorem for solving $T'(n)$ yields:

$$T'(n) = O(\log_2(n))$$

Now our algorithm works in a way that kth() reduces the problem from 3 arrays to 2 arrays i.e size of one arrays becomes zero in recursive cycles and then kth_for_two() is called on the remaining two arrays

$$T_p(n) = T(n) + T'(n)$$
$$= O(\log_2(n)) + O(\log_2(n))$$
$$= 2 \cdot O(\log_2(n))$$
$$T_p(n) = O(\log_2(n))$$

Since our algorithm uses no extra space, it doesn't create a duplicate array for each recursive call because of the use of pointers/pass by reference for arrays. Therefore, the space complexity of our algorithm is

$$O(1)$$

5 Pseudocode

Algorithm 1 `Algorithm to find kth smallest element from given 3 sorted arrays`

```
function KTH_FOR_TWO($arr1, arr2, end1, end2, k$)
    if arr1 is empty then
        return arr2[$k$]
    end if
    if arr2 is empty then
        return arr1[$k$]
    end if
    $mid1 \leftarrow \frac{\text{end1}-\text{arr1}}{2}$
    $mid2 \leftarrow \frac{\text{end2}-\text{arr2}}{2}$
    if $mid1 + mid2 < k$ then
        if arr1[mid1] > arr2[mid2] then
            return KTH_FOR_TWO($arr1, arr2 + mid2 + 1, end1, end2, k - mid2 - 1$)
        else
            return KTH_FOR_TWO($arr1 + mid1 + 1, arr2, end1, end2, k - mid1 - 1$)
        end if
    else
        if arr1[mid1] > arr2[mid2] then
            return KTH_FOR_TWO($arr1, arr2, arr1 + mid1, end2, k$)
        else
            return KTH_FOR_TWO($arr1, arr2, end1, arr2 + mid2, k$)
        end if
    end if
end function
function KTH($arr1, arr2, arr3, end1, end2, end3, k$)
    if $k > (end1 - arr1) + (end2 - arr2) + (end3 - arr3)$ or $k < 0$ then
        return $-1$
    end if
    if arr1 is eliminated then
        return KTH_FOR_TWO($arr2, arr3, end2, end3, k$)
    end if
    if arr2 is eliminated then
        return KTH_FOR_TWO($arr1, arr3, end1, end3, k$)
    end if
    if arr3 is eliminated then
        return KTH_FOR_TWO($arr1, arr2, end1, end2, k$)
    end if
    $mid1 \leftarrow \frac{\text{end1}-\text{arr1}}{2}$
    $mid2 \leftarrow \frac{\text{end2}-\text{arr2}}{2}$
    $mid3 \leftarrow \frac{\text{end3}-\text{arr3}}{2}$
    if $mid1 + mid2 + mid3 > k$ then
        $expression \leftarrow \max(\text{arr1[mid1]}, \text{arr2[mid2]}, \text{arr3[mid3]})$
        if arr1[mid1] = expression then
            return KTH($arr1, arr2, arr3, arr1 + mid1 + 1, end2, end3, k$)
        else if arr2[mid2] = expression then
            return KTH($arr1, arr2, arr3, end1, arr2 + mid2 + 1, end3, k$)
        else if arr3[mid3] = expression then
            return KTH($arr1, arr2, arr3, end1, end2, arr3 + mid3 + 1, k$)
        end if
    else if $mid1 + mid2 + mid3 \leq k$ then
        $expression \leftarrow \min(\text{arr1[mid1]}, \text{arr2[mid2]}, \text{arr3[mid3]})$
        if arr1[mid1] = expression then
            return KTH($arr1 + mid1 + 1, arr2, arr3, end1, end2, end3, k - mid1 - 1$)
        else if arr2[mid2] = expression then
            return KTH($arr1, arr2 + mid2 + 1, arr3, end1, end2, end3, k - mid2 - 1$)
        else if arr3[mid3] = expression then
            return KTH($arr1, arr2, arr3 + mid3 + 1, end1, end2, end3, k - mid3 - 1$)
        end if
    end if
end function
```

6 Proof of Correctness

Problem Definition

Design an algorithm that outputs the k-th smallest element of three sorted arrays of same size. The running time of your algorithm must be faster than O(n).

Termination

- **Trivial Inputs:**
  1. $k >$ size of arrays combined : Handled in first if condition of kth()
  2. $k < 0$: Handled in first if condition of kth()
  3. Size of arrays is zero : Equivalent to case of k greater than size of arrays combined
  4. Size of arrays is one : One array gets eliminated in first call of kth() then kth_for_two() is called second array gets eliminated in this call. Correct answer is return in next call.
- **General Inputs:**
  1. One of the array (out of three) gets reduce to half in each iteration according to the conditions met.This happens till one array get reduced to size zero and thus eliminated.Then, kth_for_two() is called same happens in this and then the kth smallest element is returned.

Assumptions

- Arrays are sorted in ascending order in themselves.
- Original arrays are passed are of same size.
- 0-indexing followed.