CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

Worksheet 2

Monday, Sep 15, 2025

1. **k-th Smallest.** Given two *sorted* arrays A and B of size m and n respectively, and an integer k, $1 \leq k \leq m+n$, design an algorithm to find the k-th smallest number in A and B. Describe your algorithm and analyze the running time of your algorithm. Your algorithm should run in $O(\log(m+n))$ time.
Answer: We cannot afford merging A and B, as it takes linear time rather than the desired logarithmic time. The idea of the algorithm is to reduce k by half using a single comparison. Specifically, each time we compare $A[mid_1 - 1]$ and $B[mid_2 - 1]$, where $mid_1 = \min(m, k/2)$ and $mid_2 = \min(n, k/2)$. If $A[mid_1 - 1] > B[mid_2 - 1]$, we eliminate all the elements before $B[mid_2]$, because it is impossible for the k-th smallest value to be located before and including $B[mid_2 - 1]$; otherwise, we eliminate all the elements before $A[mid_1]$.

   Define function find-kth-smallest(A, m, B, n, k) return the k-th smallest number of $A[0 \cdots m)$ and $B[0 \cdots n)$. We assume A and B start with index 0. We also assume A and B represent "pointers" to the array, i.e., we will use $A+3$ to represents the array shifted 3 elements. The pseudocode is shown below:

Algorithm 1 Find k-th smallest value of two sorted arrays

```
1:  function FIND-KTH-SMALLEST(A, m, B, n, k)
2:      if m = 0 then
3:          return B[k − 1]
4:      end if
5:      if n = 0 then
6:          return A[k − 1]
7:      end if
8:      if k = 1 then
9:          return min(A[0], B[0])
10:     end if
11:     mid_1 ← min(m, ⌊k/2⌋)
12:     mid_2 ← min(n, ⌊k/2⌋)
13:     if A[mid_1 − 1] > B[mid_2 − 1] then
14:         return FIND-KTH-SMALLEST(A, m, B[mid_2 ... n − 1], n − mid_2, k − mid_2)
15:     else
16:         return FIND-KTH-SMALLEST(A[mid_1 ... m − 1], m − mid_1, B, n, k − mid_1)
17:     end if
18: end function
```

   In above algorithm, at each recursive level, either k is halved, or one array is completely eliminated (and then in the next recursive call the algorithm terminates). The running time is $T(k) = T(k/2) + \Theta(1) \Rightarrow T(k) = \Theta(\log k)$.

2. **Matrices.** A is an $n \times n$ matrix containing integers. For simplicity, you may assume n is a power of 2. Design a divide-and-conquer algorithm to find the maximum element of A. Write the recurrence

relation for the time complexity and report its runtime in Θ notation. What do you observe?

Answer:

We can recursively divide the matrix into 4 quadrants, i.e., subproblems of size $\frac{n}{2}$ (not $\frac{n}{4}$). After getting the maximum elements of the 4 quadrants, determine the maximum of the 4 elements in $O(1)$ time. The recurrence relation is:

$T(n) = 4 \cdot T(\frac{n}{2}) + O(1)$ Using Master Theorem, the time complexity is $\Theta(n^2)$. In this case, using Divide-and-Conquer does not offer an improvement in asymptotic running time compared to a naive iterative search of all elements.

Algorithm 2 Find Max Element

```
1: function FIND-QUADRANT-MAX(A[1...n][1...n])
2:     if n = 1 then return A[1][1]
3:     end if
4:     m ← n/2
5:     m1 ← FIND-QUADRANT-MAX(A[1...m][1...m])          ▷ Top-left quadrant
6:     m2 ← FIND-QUADRANT-MAX(A[1...m][m+1...n])        ▷ Top-right quadrant
7:     m3 ← FIND-QUADRANT-MAX(A[m+1...n][1...m])        ▷ Bottom-left quadrant
8:     m4 ← FIND-QUADRANT-MAX(A[m+1...n][m+1...n])      ▷ Bottom-right quadrant
9: return max(m1, m2, m3, m4)
10: end function
```

3. **Selection.** Consider the selection algorithm in which the input elements are divided into groups of 7 (instead of 5). Write the recurrence to describe the worst-case running time of the algorithm and find the solution of the recurrence.

Answer:

We divide the n elements into groups of 7, take each group median, and recursively find the median of these $\lceil n/7 \rceil$ medians. Call this pivot M. At least half of the group medians are greater than or equal to M, and each such group contributes at least 4 elements greater than or equal to M, so at least $4 \cdot (n/14) = 2n/7$ elements are greater than or equal to M. Similarly, at least $2n/7$ are less than or equal to M, so the larger side has size at most $5n/7$. The recurrence is therefore

$$T(n) \leq T(n/7) + T(5n/7) + O(n).$$

Claim. $T(n) = O(n)$ under certain conditions.

Let $P(n) : T(n) \leq T(n/7) + T(5n/7) + O(n)$. Assume that for all $n < k$ that $P(n)$ is true i.e., $T(n) \leq cn$ where $c > 0$. Now, for the inductive step, we have the following based on our recursive definition ($n = k$):

$$\begin{aligned} T(k) &\leq T(k/7) + T(5k/7) + O(k) \\ &\leq c(k/7) + c(5k/7) + O(k) \text{ (Induction Hypothesis)} \\ &\leq c(\frac{6k}{7}) + Ck \text{ (new constant C for } O(k)) \\ &= k(\frac{6c}{7} + C) \end{aligned}$$

Thus, as long as $C \leq c/7$, $T(n) = O(n)$

4. **Selection.** Suppose you have a black box algorithm $A1$ that finds the $\lfloor n/10 \rfloor$-th smallest of n elements (for any given n). Show that you can find the median of n elements by making $O(1)$ calls to $A1$ and using no other pairwise element-comparisons.
Answer:
Append $4n+5$ "inf"/∞ values to the original set, yielding a total of $5n+5$ elements. Now, call A1 on this new set to obtain the median value $\lfloor \frac{5n+5}{10} \rfloor = \lfloor \frac{n+1}{2} \rfloor$.

5. **Merge.** *A k-way merge operation.* Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

   a) Here's one strategy: Using the `merge` procedure, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

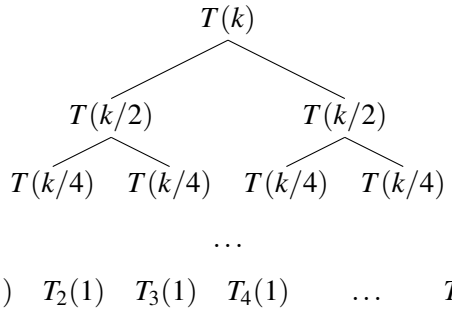
   b) Give a more efficient solution to this problem, using divide-and-conquer.

   Answer:

   a) Each merge step requires $O(x+y)$ time for x elements in first array and y elements in the second array. Extending this, we can see that initially we start with $2n$ being the time required to merge the first two arrays. Then to merge with the third array it would take $3n$, and then $4n$ for the fourth and so on. Thus it can be represented as:

$$T(n) = 2n+3n+4n\ldots(k-1)n+kn = n(2+3+4\ldots(k-1)+k) = n\left(\frac{k(k+1)}{2}-1\right) = O(nk^2)$$

   b) Recursively divide the arrays into two sets, each of $k/2$ arrays. Then merge the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is $k = 1$, when no merging needs to take place.



   For each level, one can see that it will be $O(nk)$. Thus, the running time is given by $T(k) = 2T(k/2)+O(nk)$. By the Master theorem, $T(k) = O(nk\log k)$.

   Another possible approach to solve this problem without using divide-and-conquer is to merge all k arrays at the same time. To do this, you need an efficient way to decide which of the arrays has the next smallest element. This can be accomplished with a min-heap (topic of week 4). Simply store the smallest elements (which are also the first elements) from each array in a min-heap, along with the array they came from and which index they occupied in that array. To fill the output array, delete the minimum element in the heap in $O(\log k)$ time, go to that element's

original array, and insert the next value into the heap in $O(\log k)$ time. This process must be repeated n times, so the total running time is $O(nk\log k)$, the same as the divide-and-conquer approach.

6. **Array Rotations.** Consider a rotation operation that takes an array and moves its last element to the beginning. After n rotations, an array $[a_0, a_1, a_2 \ldots a_{m-1}]$ of size m where $0 < n \leq m$, will become:

$$[a_{m-n}, a_{m-n+1}, \ldots, a_{m-2}, a_{m-1}, a_0, a_1, \ldots a_{m-n-1}]$$

Notice how a_{m-1} is adjacent to a_0 in the middle of the new array. For example, two rotations on the array $[1,2,3,4,5]$ will yield $[4,5,1,2,3]$.

You are given a list of unique integers `nums`, which was previously sorted in ascending order, but has now been rotated an unknown number of times. Find the number of rotations in $O(\log n)$ time. *(Hint: consider Binary Search.)*

Answer:

Given an array like $arr = [10,1,2,3,4,5]$. We can use a modified version of binary search where in after we get the middle, we have to choose the side containing the pivot point. This side can be determined by the following:

1. When pivot is on the left m, then we know that the start of the array will be greater than middle i.e. $arr[start] > arr[mid-1]$
2. Else if pivot is on the right m then the middle will be of a greater value than the end $arr[mid+1] > arr[end]$

Pseudo code:

while $start < end$ **do**
  $mid \Leftarrow \lfloor (start + end)/2 \rfloor$
  if $arr[mid] > arr[mid+1]$ **then**
    return mid
  end if
  if $arr[mid-1] > arr[mid]$ **then**
    return $mid - 1$
  end if
  if $arr[start] > arr[mid-1]$ **then**
    $end = mid - 1$
  else
    $start = mid + 1$
  end if
end while