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

HW 2 Solution

Due September 22, 10:00 pm

Instructions: You are encouraged to solve the problem sets on your own, or in groups of three to five people, but you must write your solutions strictly by yourself. You must explicitly acknowledge in your write-up all your collaborators, as well as any books, papers, web pages, etc. you got ideas from.

Formatting: Each problem should begin on a new page. Each page should be clearly labeled with the problem number. The pages of your homework submissions must be in order. You risk receiving no credit for it if you do not adhere to these guidelines.

Late homework will not be accepted. Please, do not ask for extensions since we will provide solutions shortly after the due date. Remember that we will drop your lowest three scores.

This homework is due Monday, September 22, at 10:00 pm electronically. You need to submit it via Gradescope. Please ask on Campuswire about any details concerning Gradescope and formatting.

1. (20 pts.) **Lower Bounds.** Consider the decision-tree model of the following problems. Apply the leaf-counting argument to obtain the information theory lower bounds (ITLB) for each of them:

(a). Find the median of an unsorted list of elements.

(b). Given a set of n elements, determine whether they are all distinct.

(c). Given two sorted list of n_1 and n_2 elements respectively, produce a merged list of sorted elements.

Answer: Remember that for all of this problem we are trying to find the *Lower Bound* complexity of the problem.

(a). There are n possible outcomes for the problem. So, the information theoretic lower bound is $\Omega(\log n)$.

(b). We assume each element can be compared with others for equality only. So, each pair of element can be either equal or not. Total outcomes are $\binom{n}{2}+1$ where there is one outcome with the answer "Yes" and $\binom{n}{2}$ "No" outcomes (at least one pair is equal). The "No" outcomes are unique, with each one corresponding to a different certificate that identifies a specific pair of equal elements (e.g., A = B). So, the ITLB lower bound is $\Omega(\log(\binom{n}{2}+1)) = \Omega(\log n)$.

(c). There are in total n_1+n_2 slots and we need to figure out where to place n_1 (or n_2) elements. The others will just fall in place as they are sorted. Consequently, there are $\binom{n_1+n_2}{n_1}$ possible answers. The information theoretic lower bound will be:

$$\begin{aligned}\log\binom{n_1+n_2}{n_1} &\leq \log 2^{n_1+n_2}\\ &= \Omega(n_1+n_2)\end{aligned}$$

2. (20 pts.) **Overlapping Intervals.** You are given a list of n intervals $[x_i, y_i]$, where x_i, y_i are integers with $x_i \leq y_i$. The interval $[x_i, y_i]$ represents the set of integers between x_i and y_i. For instance, the interval [3,6] represents the set $\{3,4,5,6\}$. Define the *overlap* of two intervals I, I' to be $|I \cap I'|$, i.e. the number of integers that are members of both intervals. Design a divide-and-conquer algorithm that, when given n

intervals, finds and outputs the pair of intervals with highest overlap (you may resolve ties arbitrarily). A trivial $\Theta(n^2)$ algorithm can be achieved by comparing all pairs of intervals; look for something better. (*Hint:* Try splitting the list using the left endpoints of the intervals.)

Answer:

First, we sort the list of intervals by their left endpoints (this only happens once). Then we do the following:

Algorithm 1 Max-Overlap

```
1:  function MAX-OVERLAP(I[1,...,n])
2:      Sort the intervals I[1,...,n] by their left endpoints.
3:      return RECURSIVE-MAX-OVERLAP(I[1,...,n])
4:  end function
5:  function RECURSIVE-MAX-OVERLAP(I[1,...,n])
6:      if n = 1 then
7:          return 0
8:      end if
9:      L ← RECURSIVE-MAX-OVERLAP(I[1,...,⌊n/2⌋])          ▷ Largest overlap on left half
10:     R ← RECURSIVE-MAX-OVERLAP(I[⌊n/2⌋+1,...,n])        ▷ Largest overlap on right half
11:     C ← 0          ▷ Largest overlap between an interval in left half and an interval in right half
12:     J ← interval in I[1,...,⌊n/2⌋] with the largest right endpoint
13:     for each interval J' in I[⌊n/2⌋+1,...,n] do
14:         C ← max(C, OVERLAP(J,J'))
15:     end for
16:     return max(L,C,R)
17: end function
```

At each recursive step, we break the list into two halves and recursively find the largest overlap on the left half, and on the right half. Then we search for the largest overlap between an interval of the left half and an interval of the right half. From the left half we only need to consider the interval whose right endpoint is the greatest—no other interval from the left half can produce a greater overlap with one on the right. So we find that interval (in linear time), and then check its overlap with all of the intervals on the right half, which also takes linear time.

The running time is $\Theta(n \log n)$. Lines 8 through 11 take $\Theta(n)$ time, and we recursively call the function on two instances of half the size. This gives the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

which solves to $T(n) = \Theta(n \log n)$ by the Master Theorem. The sorting we do once at the very beginning also takes time $\Theta(n \log n)$, so the total running time is again $\Theta(n \log n)$.

3. (20 pts.) **Peak Elements**. Given an array with n distinct elements, a peak element is one that is strictly larger than both of its neighbors (or larger than its only neighbor, if it is on the edge of the array). For example, in [4,9,3,10], there are two peak elements: the 9 and the 10. Design an algorithm with $\Theta(\log n)$ running time to find the index of a peak element in an array. If an array has multiple peak elements, you may return any of them.

Answer:

Our algorithm first picks the center element $A[\lfloor n/2 \rfloor]$, and checks if it is a peak element. If it is, output it, and stop. If it is not, there must exist a larger neighbor. If $A[\lfloor n/2 \rfloor - 1] > A[\lfloor n/2 \rfloor]$, recursively search for a local maximum in $A[0, \ldots, \lfloor n/2 \rfloor - 1]$. Otherwise, search for a peak element in $A[\lfloor n/2 \rfloor + 1, n-1]$. We only

look at one half, even when both neighbors are larger. At the base case, we have a list of just one element, which will be a peak element.

The algorithm is correct in all cases. When $A[\lfloor n/2 \rfloor]$ is a peak element, the algorithm outputs it. If $A[\lfloor n/2 \rfloor]$ is not, suppose we are in the case that the algorithm chooses to search $A[0, \ldots, \lfloor n/2 \rfloor - 1]$ (symmetric argument for the other case). Now, imagine if we start at $A[\lfloor n/2 \rfloor]$, and continue searching to the left until we either reach $A[0]$ or an element that has a smaller element to its left. Either way, this element must be a peak element. Thus, there must exist a peak element in this half of the list. Furthermore, exactly one end of this half list is not an end element of the original list, that is $A[\lfloor n/2 \rfloor - 1]$, but we know $A[\lfloor n/2 \rfloor - 1] > A[\lfloor n/2 \rfloor]$, so if $A[\lfloor n/2 \rfloor - 1]$ is a peak element in the half list, it is a peak element in the original list.

The running time is $\Theta(\log n)$, since at each level we solve one problem of size $\frac{n}{2}$, plus constant work (comparing the middle elements to neighbors). Thus, $T(n) = T(\frac{n}{2}) + \Theta(1)$, and so $T(n) = \Theta(\log n)$ by the Master Theorem.

4. (20 pts.) **Median of Medians of Medians** Recall that in the median-of-medians selection algorithm, if we use group size 3, the time complexity satisfies $T(n) \leq T(n/3) + T(2n/3) + O(n)$. We cannot conclude $T(n) = O(n)$ based on this recurrence relation. The following algorithm aims to solve this issue by grouping twice and use the median of medians of medians as the pivot.

MoMoMSelect(A, k):

1. Partition A into $n/3$ groups, each of size 3.
2. Let M be the list of medians of these $n/3$ groups.
3. Partition M into $n/9$ groups, each of size 3.
4. Let M' be the list of medians of these $n/9$ groups.
5. Find the median p of M' by a recursive call MoMoMSelect(M', $|M'|/2$).
6. Use p as the pivot to partition A and recurs on the appropriate subarray as in Select.

What guarantee can you derive about the pivot p? Find the recurrence relation describing the worst-case running time of this algorithm. What is the solution to the recurrence relation?

Answer: Let p be the median of array M'. Consider the numbers in M. In half of the $n/9$ subarrays, i.e., those with median that is less than p, there are two numbers (i.e., the median of this subarray and another number) that are guaranteed less than p. This amounts to $2 \cdot (n/9)/2 = n/9$ elements in M that are less than p. Now consider the numbers in A. Recall that each element in M is a median of a subarray of size 3 in A. As we already show that there are $n/9$ elements in M that are less than p, in the corresponding subarrays, there are 2 numbers (the median and another number) that are guaranteed less than p. This amounts to $2 \cdot n/9 = 2n/9$ elements in A that are less than or equal to p. Thus, there are at most $7n/9$ elements are larger than p in A. Symmetrically, there are at most $7n/9$ elements are smaller than p in A.

The recursive call of selection function in above algorithm takes $T(n/9)$ time as $|M'| = n/9$. The recurrence relation for the entire algorithm is: $T(n) \leq \Theta(n) + T(n/9) + T(7n/9)$, which gives $T(n) = \Theta(n)$.

5. (20 pts.) **Transpose of a Matrix.** The transpose of a matrix A, denoted A^T, is a new matrix whose rows are the columns of A. The standard way to compute the transpose of an $n \times n$ matrix takes $O(n^2)$ time.

(a). Consider an $n \times n$ matrix A where n is a power of 2. Partition it into four $n/2 \times n/2$ sub-matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Show how to compute the transpose A^T by recursively computing the transposes of the sub-matrices.

(b). Write a recurrence relation for the time complexity $T(n)$ of this recursive algorithm. Solve the recurrence relation and determine the asymptotic time complexity. Does this recursive approach offer a performance improvement over the standard algorithm?

Answer:

(a). To find the transpose of a matrix A, we swap its rows and columns. When A is partitioned into four sub-matrices, its transpose A^T can be expressed in terms of the transposes of these sub-matrices.

Let $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where each A_{ij} is an $\frac{n}{2} \times \frac{n}{2}$ matrix.

The transpose of A, denoted A^T, is obtained by reflecting the elements across the main diagonal. This operation also applies to the sub-matrices. The transpose of a block matrix is the matrix of transposed blocks, with the off-diagonal blocks swapped.

$$A^T = \begin{bmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{bmatrix}$$

This decomposition suggests a recursive algorithm:

- **Base Case:** If the matrix size $n = 1$, the transpose is the matrix itself, which takes $O(1)$ time.
- **Recursive Step:** For an $n \times n$ matrix with $n > 1$, partition it into four sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$. Recursively compute the transposes of the four sub-matrices: A_{11}^T, A_{12}^T, A_{21}^T, and A_{22}^T. Construct the final transposed matrix by placing the results as shown above.

The crucial step is that we do not need to transpose A_{12} and A_{21} and then swap their positions. Instead, we can simply compute A_{12}^T and A_{21}^T recursively and place them in the correct off-diagonal positions in the final result. The diagonal sub-matrices A_{11} and A_{22} are placed in the diagonal positions after their transposes are computed.

(b). Let $T(n)$ be the time complexity to compute the transpose of an $n \times n$ matrix using the recursive algorithm described above.

The algorithm performs four recursive calls on matrices of size $\frac{n}{2} \times \frac{n}{2}$. Each of these calls contributes $T(n/2)$ to the total time.

The work done outside of the recursive calls consists of partitioning the matrix and re-assembling the results. These operations involve rearranging pointers or creating a new matrix and copying data. For a standard implementation, this overhead is proportional to the size of the matrix, $O(n^2)$. However, for this specific problem, we are simply arranging the four already-transposed sub-matrices. This takes constant time, $O(1)$, assuming the base matrices can be treated as single entities after the recursive calls return. The partitioning step also takes $O(1)$ time.

Therefore, the recurrence relation for the time complexity is:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(1)$$

The base case is $T(1) = 1$, representing the constant time to transpose a 1×1 matrix.

According to the Master Theorem, if Case 1 applies, the solution is $T(n) = O(n^{\log_b a})$.

$$T(n) = O(n^{\log_2 4}) = O(n^2)$$

The asymptotic time complexity of the recursive algorithm is $O(n^2)$. The standard, non-recursive algorithm for transposing an $n \times n$ matrix also has a time complexity of $O(n^2)$, as it requires two nested loops to iterate through all n^2 elements and perform the necessary swaps or assignments.

Rubric:

Problem 1, ? pts

?

Problem 2, ? pts

?

Problem 3, ? pts

?

Problem 4, ? pts

?

Problem 5, ? pts

?