CS3000 Algorithms

Solutions to Recitations 02

Akshar Varma *Date: 13 July 2023*

Problem 1 (Sorting special arrays)

Consider the problem of sorting an array $A[1, \ldots, n]$ of integers. We presented an $O(n \log n)$-time algorithm in class and, also, proved a lower bound of $\Omega(n \log n)$ for any comparison-based algorithm.

1. Give an efficient sorting algorithm for a ***boolean***[1] array $B[1, \ldots, n]$.
2. Give an efficient sorting algorithm for an array $C[1, \ldots, n]$ whose elements are taken from the set $\{1, 2, 3, 4, 5\}$.
3. Give an efficient sorting algorithm for an array $D[1, \ldots, n]$ whose elements are distinct ($D[i] \neq D[j]$, for every $i \neq j \in \{1, \ldots, n\}$) and are taken from the set $\{1, 2, \ldots, 2n\}$.
4. In case you designed linear-time sorting algorithms for the previous subparts, does it mean that the lower bound for sorting of $\Omega(n \log n)$ is wrong? Explain.

Solution:

1. (Boolean Array) Keep two counters c_0 and c_1 and sweep through the array. Whenever you see a 0 or False increment c_0 the counter for 0 and when you see a 1 or True increment c_1 the counter for 1. At the end, in the output array, make the first c_0 entries 0 and the next c_1 entries 1.

   Time complexity: $O(n)$ **Space complexity:** $O(1)$ for the two counters.

2. (Sort when elements are from $\{1, 2, 3, 4, 5\}$) This is very similar to the boolean array algorithm. We keep 5 counters: c_1, c_2, c_3, c_4, c_5 and increment the counter i when you see element i. At the end make the first c_1 positions to be 1, c_2 to be 2 and so on.

   time complexity: $O(n)$ **space complexity:** $O(1)$ for the five counters.

3. (Sorting n elements from $\{1, 2, \ldots, 2n\}$) This is similar to the previous problems, but instead of a constant number of possible elements each potentially occurring n times, we have n out of $2n$ elements each occurring at most once. We slightly modify the earlier algorithm and use a boolean array B of length $2n$ initialized to all False. We then sweep the input and whenever we see element i, we set B_i to be True. At the end, we sweep B from the start and insert whatever element was True into the output array.

   time complexity: $O(n)$ **space complexity:** $O(n)$.

4. We did find linear time solutions to the previous parts. However, note that we didn't use any comparisons in those sorting algorithms. The lower bound of $\Omega(n \log n)$ is true only when the algorithm is a comparison based sorting algorithm. The lower bound theorem crucially relies on that assumption. Since that assumption is not valid for any of the sorts above, the lower bound theorem cannot be applied for those sorts. You can read up on Counting Sort for more details: https://en.wikipedia.org/wiki/Counting_sort

[1] In a boolean array $B[1, \ldots, n]$, each element $B[i]$ (for $i = 1, \ldots, n$) is either 0 or 1.

In computer science, counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the $\Omega(n \log n)$ lower bound for comparison sorting does not apply to it.

■

Problem 2 (Local maximum)

Given an array $A = [a_1, a_2, \ldots, a_n]$ of distinct positive integers which is not necessarily sorted, find any local maximum in the array A. An element at index $1 < i < n$ is a local maximum if a_i is at least as big as elements on both side of it. That is $a_i \geq a_{i-1}$ and $a_i \geq a_{i+1}$. For $i = 1$ or $i = n$, we only compare $a_1 \geq a_2$ and $a_n \geq a_{n-1}$ respectively.

a) Give a $\Theta(n)$ time algorithm.

b) Give an $O(\log n)$ algorithm using divide-and-conquer.

Solution:

a) Give a $\Theta(n)$ time algorithm. A possible solution :

   A global maxima is also a local optima.

b) Give an $O(\log n)$ algorithm using divide-and-conquer.

   We can use a binary search like idea to reduce our array size by a factor of 2 at each recursive call and do a constant work at each step. That gives us a divide and conquer solution which runs in time $O(log\ n)$.

■

Problem 3 (Counting Inversions)

An **inversion** in an array $A[1 \ldots n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. Describe and analyze an algorithm to count the number of inversions in an n-length array in $O(n \log n)$ time. [***Hint****: Remember mergesort.*]

Solution:

Consider Mergesort. In the base case, we can directly count the number of inversions. Let m be the index of the middle element where the array has been split. So at some intermediate point in the algorithm, we have already counted inversions (i, j) where both $i, j \leq m$ and where both $i, j \geq m$. We only need to consider the number of inversions where $i \leq m$ and $j \geq m$.

Counting inversions that cross the middle boundary is done during the merge process. If during merge, we find that $A[i] > A[j]$ where i and j are the indices to the left and right half respectively, then all elements to the right of $A[i]$ are inversions with respect to $A[j]$. This adds $m-i+1$ inversions so we can keep a count of crossing inversions during the merge step without incurring any additional run time penalty. Thus this algorithm to count inversions runs in $\Theta(n \log n)$ time.

■

Problem 4 (Ternary Tree Track Totals)

A ternary tree is a rooted tree where each node (except the leaves) have three children each. We are given a ternary tree T with a positive integer label on each node of the tree. Further, you are given that the tree has k levels such that at level $i \in \{1, \ldots, k\}$, there are 3^{i-1} nodes since every node at level $i-1$ has 3 children each.

You want to find the maximum path sum starting at the root of the tree and following any path on the tree from root to a leaf. From every node in your path, except the terminal leaf node, you have three options for which child to use for your path.

Solution:

Note that this problem has optimal substructure since the Maximum Path Sum from the a node to any leaf in that subtree is equal to the value at that node added to the maximum path sum from the three children. Further, note that there is no overlapping subproblems since every path is independent of the other paths. Thus, this problem allows a divide-and-conquer solution but does not require dynamic programming.

Given the tree input, let $A[i]$ denote the value of the integer label at a node i. Further, let $i.left, i.middle, i.right$ denote the left, middle and right children of the node i. To solve the maximum path sum problem, we use the following recurrence where $S[i]$ returns the maximum path sum starting at node i and ending at any leaf node:

$$S[i] = \begin{cases} A[i] & \text{if } i \text{ is a leaf node} \\ A[i] + \max(S[i.left], S[i.middle], S[i.right]) & \text{otherwise} \end{cases}$$

The output of the algorithm is the value of $S[root]$ where $root$ is the root node of the tree.

The running time can be expressed in terms of the number of nodes n in the tree using the recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 3T(n/3) + O(1) & \text{otherwise} \end{cases}$$

This recurrence has the solution: $T(n) = O(n)$ which look like a linear time solution, however note that for a tree that has k levels, the number of nodes is exponential in k. Precisely, it is $3/2 \cdot (3^k - 1) = O(3^{k+1})$. ■

Problem 5 (Tiling checkerboards)

Suppose you are given a $2^n \times 2^n$ checkerboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by L-shaped

tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time.

[***Hint**: First prove that such a tiling always exists.*]

Solution:

Proof of existence/correctness: We first look at the simplest case where $n = 1$ and one square is removed. There is exactly one way to tile it and this forms our base case. Since we know how to solve a small instance of this problem, given a larger instance we need to split this into smaller instances of the same problem and recurse until we reach our base case. This recursion needs to maintain two key properties: that the sides are powers of two and that there is exactly one square missing.

In the general case, suppose we are given an $2^n \times 2^n$ board with the (i, j) square missing. Without loss of generality, (i, j) is in the top left quadrant.[2] Since we want the smaller instances to be powers of two, we need to split this into four equal sized quadrants. We need to further ensure that each has a missing square. Only the top left quadrant has a missing square, so we need to create missing squares in the rest before recursing. To do this, we place a tile such that the indices $(2^{n/2}, 2^{n/2}) + 1, (2^{n/2} + 1, 2^{n/2}), (2^{n/2} + 1, 2^{n/2} + 1)$ are used up. The top right quadrant now has its bottom left square missing, the bottom left quadrant has its top right square missing and the bottom right quadrant has its top left square missing. This ensures that all four quadrants have sizes $2^{n-1} \times 2^{n-1}$ and that each has a single missing square. This proves, by induction, that there exists a tiling.

Algorithm: If board is 2×2, tile it. Otherwise split into four quadrants, and add a tile so that the three previously whole quadrants now have a whole in their corners. Recurse on each quadrant.

Analysis: Let $N = 4^n = 2^n \times 2^n$ be the size of the input, then we have a straightforward recurrence equation for the running time of the above algorithm. $T(N) = 4T(N/4) + O(1) = \Theta(N) = \Theta(4^n)$ which proves (by the Master Theorem) that the running time of our algorithm is as required.

■

[2] otherwise rotate the board to make this true.