Homework 3 Solution

Due: Oct 15, in class.

Instruction. There are 7 problems in this homework, worth 100 points in total. For all algorithm design problems, you need to provide proofs on the correctness and running time of the algorithms unless otherwise instructed.

Problem 1 [15 pts]

Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

Solution: We first divide these 2 integers into 4 shorter integers. These integers are $x = (1001)_2$, $y = (1011)_2$, $u = (1011)_2$, and $w = (1010)_2$ satisfying $(10011011)_2 = x \cdot 2^4 + y$ and $(10111010)_2 = u \cdot 2^4 + w$. . . . . (5 pts)

Via a recursive call, we get $p = xu = (1100011)_2$, $q = yw = (1101110)_2$, and $t = (x+y)(u+w) = (110100100)_2$. . . . . (5 pts)

Using these results, we can get final answer which equals to $p \cdot 2^8 + (t - p - q) \cdot 2^4 + q = (111000010011110)_2 = (28830)_{10}$. . . . . (5 pts)

Problem 2 [15 pts]

Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size $n - 1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

Solution: Let $T(n)$ denote the running time of an algorithm with input of size n. Suppose the boundary condition is $T(n) = 1$ for $n \leq 1$.

(a) For this algorithm, $T(n)$ satisfies $T(n) = 5T(n/2) + O(n)$. Solving this recurrence, we get $T(n) = O(n^{\log_2 5})$. . . . . (4 pts)

(b) For this algorithm, $T(n)$ satisfies $T(n) = 2T(n-1) + O(1)$. Solving this recurrence, we get $T(n) = O(2^n)$. .... (4 pts)

(c) For this algorithm, $T(n)$ satisfies $T(n) = 9T(n/3) + O(n^2)$. Solving this recurrence, we get $T(n) = O(n^2 \log n)$. .... (4 pts)

Since $\lim_{n\to\infty} \frac{n^2 \log n}{n^{\log_2 5}} = 0$ and $\lim_{n\to\infty} \frac{n^2 \log n}{2^n} = 0$, $n \log n$ is asymptotically better than other time complexities. Hence, the third algorithm is better than other algorithms and it is better to choose the third one. .... (3 pts)

Note: Master theorem.

Problem 3 [15 pts]

You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values so there are $2n$ values total and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution:

Analysis (Proof of Correctness):

Say A and B are the two databases and $A(i)$, $B(i)$ are i^{th} smallest elements of A, B.

First, let us compare the medians of the two databases. Let k be $\lceil \frac{1}{2}n \rceil$, then $A(k)$ and $B(k)$ are the medians of the two databases. Suppose $A(k) < B(k)$ (the case when $A(k) > B(k)$ would be the same with interchange of the role of A and B). Then one can see that $B(k)$ is greater than the first k elements of A. Also $B(k)$ is always greater then the first $k-1$ elements of B. Therefore $B(k)$ is at least $2k^{\text{th}}$ element in the combined database. Since $2k \geq n$, all elements that are greater than $B(k)$ are greater than the median and we can eliminate the second part of the B database. Let B' be the first half of B (i.e., the first k elements of B).

Similarly, the first $\lfloor \frac{1}{2}n \rfloor$ elements of A are less than $B(k)$, and thus, are less than the last $n-k+1$ elements of B. Also they are less than the last $\lceil \frac{1}{2}n \rceil$ elements of A. So, they are less than at least $n - k + 1 + \lceil \frac{1}{2}n \rceil = n + 1$ elements of the combined database. It means that they are less than the median and we can eliminate them as well. Let A' be the remaining part of A (i.e., the $\left[\lfloor \frac{1}{2}n \rfloor + 1; n\right]$ segment of A).

Now we eliminate $\lfloor \frac{1}{2}n \rfloor$ elements that are less than the median, and the same number of elements that are greater then median. It is clear that the median of the remaining elements is the same as the median of the original set of elements. We can find the median in the remaining set using recursion for A' and B'. Note that we can't delete elements from the databases. However, we can access i^{th} smallest elements of A' and B': the i^{th} smallest elements of A' is $i + \lfloor \frac{1}{2}n \rfloor^{\text{th}}$ smallest elements of A, and the i^{th} smallest elements of B' is i^{th} smallest elements of B.

Formally, the algorithm is the following. We write recursive function median(n, a, b) that takes integers n, a and b and find the median of the union of the two segments $A[a+1; a+n]$ and $B[b+1; b+n]$. To find the median in the while set of elements we evaluate median$(n, 0, 0)$.

....(5 pts)

Algorithm:

Algorithm 1 median

```
1: Input: $n, a, b$.
2: if $n = 1$ then Output: $\min(A(a+1), B(b+1))$
3: $k = \lceil \frac{1}{2}n \rceil$
4: if $A(a+k) < B(b+k)$ then
5:     Output: median$(k, a + \lfloor \frac{1}{2}n \rfloor, b)$
6: else
7:     Output: median$(k, a, b + \lfloor \frac{1}{2}n \rfloor)$
```

....(5 pts)

Time Complexity:

Let $Q(n)$ be the number of queries asked by our algorithm to evaluate median$(n, 0, 0)$. Then it is clear that $Q(n) = Q(\lceil \frac{1}{2}n \rceil) + 2$. Therefore $Q(n) = 2\lfloor \log n \rfloor + 2 = O(\log n)$. .... (5 pts)

Problem 4 [15 pts]

Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Lets call a pair a significant inversion if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

Solution: We will define a recursive divide and conquer algorithm `ALG` which takes a sequence of distinct numbers $a_1, \cdots, a_n$ and returns N and $a'_1, \cdots, a'_n$ where

- N is the number of significant inversions
- $a'_1, \cdots, a'_n$ is same sequence sorted in the increasing order.

`ALG` is similar to the algorithm from the chapter that computes the number of inversions. The difference is that in the 'conquer' step we merge twice. First we merge $b_1, \cdots, b_k$ with $b_{k+1}, \cdots, b_n$ just for sorting, and then we merge $b_1, \cdots, b_k$ with $2b_{k+1}, \cdots, 2b_n$ for counting significant inversions. Now we define `ALG` formally.

For $n = 1$, `ALG` just returns $N = 0$ and $\{a_1\}$ for the sequence. For $n > 1$, we have the following.

- let $k = \lfloor n/2 \rfloor$
- call `ALG`$(a'_1, \cdots, a'_k)$. Say it returns N_1 and $b_1, \cdots, b_k$
- call `ALG`$(a'_{k+1}, \cdots, a'_n)$. Say it returns N_2 and $b_{k+1}, \cdots, b_n$
- compute the number of significant inversions (a_i, a_j) where $i \le k \le j$.

- return $N = N_1 + N_2 + N_3$ and $a'_1, \cdots, a'_n = \texttt{MERGE}(b_1, \cdots, b_k; b_{k+1}, \cdots, b_n)$

`MERGE` can be implemented in $O(n)$ time.

- Initialize counters $i \leftarrow k$, $j \leftarrow n$, $N_3 \leftarrow 0$.
- If $b \leq 2b_j$
  - If $j > k + 1$ decrease j by 1.
  - If $j = k + 1$ return N_3
- If $b > 2b_j$, then increase N_3 by $j - k$. Then,
  - If $i > 1$, decrease i by 1.
  - If $i = 1$ return N_3.

Explanation: For every i we count the number of significant inversions between b_i and all b'_js. If $b_i \leq 2b_j$ then there is no significant inversions between b_i and any b_m, such that $m \geq j$, we decrease j. If $b_i > 2b_j$ then $b_i > 2b_m$ for all m such that $k < m \leq j$. In other words, we have deleted $j - k$ significant inversions involving b_i. So we increase N_3 by $j - k$. Finally, when are down to $i = 1$ and have counted significant inversions involving b_1, there are no more significant inversions to be detected.

Problem 5 [15 pts]

Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, . . . well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Heres a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0 : $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermostintuitively, some portion of it can be seen if you look down from "$y = \infty$."

Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all of the ones that are visible. Figure 1 gives an example.

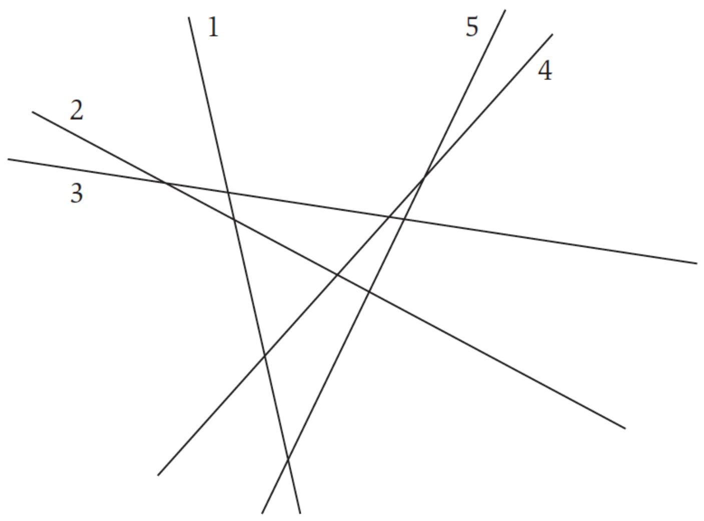

Figure 1: An instance of hidden surface removal with five lines (labeled 1 – 5 in the figure). All the lines except for 2 are visible.

Solution: First label the lines in the order of increasing slope, and then use divide and conquer approach. If $n \leq 3$, the base case of divide and conquer approach - we can easily find the visible lines in constant time. (The first and third lines will always be visible; the second will be visible if and only if it meets the first line to the left of where the third line meets the first line).

Let $m = \lceil n/2 \rceil$. We first recursively compute the sequence of visible lines among $L_1, \cdots, L_m$ - say they are $\mathcal{L} = \{L_{i_1}, \cdots, L_{i_p}\}$ in order of increasing slope. We also compute, in this recursive call, the sequence of points $a_1, \cdots, a_{p-1}$ where a_k is the intersection of line L_{i_k} with line $L_{i_{k+1}}$. Notice that $a_1, \cdots, a_{p-1}$ will have increasing x-coordinates; for if two lines are both visible, the region in which the line of smaller slope is uppermost lies to the left of the region in which the line of larger slope is uppermost. Similarly we recursively compute $\mathcal{L}' = \{L_{j_1}, \cdots, L_{j_q}\}$ of visible lines among $L_{m+1}, \cdots, L_n$, together with the sequence of intersection points $b_k = L_{j_k} \cap L_{j_{k+1}}$ for $k = 1, \cdots, q-1$.

To complete the algorithm, we must show how to determine the visible lines in $\mathcal{L} \cup \mathcal{L}'$ together with the corresponding intersection points, in $O(n)$ time. (Note that $p + q \leq n$, so it is enough to run in time $O(p+q)$). We know that L_{i_1} will be visible, because it has the minimum slope among all the lines in this list; similarly, we know that L_{j_q} will be visible, because it has the maximum slope.

We merge the sorted lists $a_1, \cdots, a_{p-1}$ and $b_1, \cdots, b_{q-1}$ into a single list of points $c_1, c_2, \cdots, c_{p+q-2}$ ordered by increasing x-coordinate. This takes $O(n)$ time. Now, for each k, we consider the line that is uppermost $\mathcal{L}$ at x-coordinate c_k and the line that is uppermost $\mathcal{L}'$ at x-coordinate in c_k, and the line that is uppermost in $\mathcal{L}'$ at x coordinate c_k. Let l be the smallest index for which the uppermost line in $\mathcal{L}'$ lies above the uppermost line in $\mathcal{L}$ at x-coordinate c_l. Let the two lines at this point be $L_{i_s} \in \mathcal{L}$ and $L_{j_t} \in \mathcal{L}'$. Let (x^*, y^*) denote the point in the plane at which L_{i_s} and L_{j_t} intersect. We have thus established that x^* lies between the x-coordinate of c_{l-1} and c_l. This means that L_{i_s} is the uppermost in $\mathcal{L} \cup \mathcal{L}'$ immediately to the left of x^*, and L_{j_t} is uppermost is $\mathcal{L} \cup \mathcal{L}'$ immediately to the right of x^*. Consequently, the sequence of visible lines among $\mathcal{L} \cup \mathcal{L}'$ is $L_{i_1}, \cdots, L_{i_s}, L_{j_t}, \cdots, L_{j_q}$; and the sequence of intersection points in $a_{i_1}, \cdots, a_{i_{s-1}}, (x^*, y^*), b_{j_t}, \cdots, b_{j_{q-1}}$. Since this is what we want to return in the next level of recursion, the algorithm is complete.

Problem 6 [10 pts]

Consider an n-node complete binary tree T, where $n = 2^{d-1}$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

Solution: For simplicity, we will say u *is smaller than* v, or $u \prec v$ if $x_u < x_v$. We will extend this to sets: If S is a set of nodes, we say $u \prec S$ if u has a smaller value than any node in S.

The algorithm is as follows. We begin at the root r of the tree, and see if r is smaller than its two children. If so, the root is the local minimum. Otherwise, we move to any smaller child and iterate.

The algorithm terminates when either (1) we either reach a node v which is smaller than both its children, or (2) we reach leaf w. In the former case, we return w.

The algorithm performs $O(d) = O(\log n)$ probes of the tree; we must now argue that the returned value is a local minimum. If the root r is returned, then it is a local minimum as explained above. If we terminate in case (1), v is a local minimum because v is smaller than its parent (since it was chosen in the previous iteration) and its two children (since we terminated). If we terminate in case (2), w is a local minimum because w is smaller than its parent (again since it was chosen in the previous iteration).

Problem 7 [15 pts]

Suppose now that youre given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution: Let B denote the set of nodes on the *border* of the grid G, i.e. the outermost rows and columns. Say that G has *Property* (*) if it contains a node $v \notin B$ that is adjacent to a node in B and satisfies $v \prec B$. Note that in a grid G with Property (*), the global minimum does not occur on the border B (since the global minimum is no larger than v, which is smaller than B) - hence G has at least one local minimum that does not occur on the border. We call such a local minimum an *internal local minimum.*

We now describe a recursive algorithm that takes a grid satisfying Property (*) and returns an internal local minimum, using $O(n)$ probes. At the end, we will describe how this can be converted into a solution for the overall problem.

Thus let G satisfy the property (*) and let $v \notin B$ be adjacent to a node in B and smaller than all nodes in B. Let C denote the union of the nodes in the middle row and the middle column of G, not counting the nodes in the border. Let $S = B \cap C$; deleting S from G divides up G into four sub-grids. Finally let T be all nodes adjacent to S.

Using $O(n)$ probes, we find the node $u \in S \cup T$ of minimum value. We know that $u \notin B$, since $v \in S \cup T$ and $v \prec B$. Thus, we have two case. If $u \in C$, then u is an internal local minimum, since all of the neighbours of u are in $S \cup T$, and u is smaller than all of them. Otherwise, $u \in T$. Let G' be the sub-grid containing u, together with the portions of S that border it. Now G' satisfies Property (*), since u is adjacent to the border of G' and is smaller than all nodes on the border of G'. Thus, G' has an internal local minimum, which is also an internal local minimum of G. We call our algorithm recursively on G' to find such an internal local minimum.

If $T(n)$ denotes the number of probes needed by the algorithm to find an internal local minimum in an $n \times n$ grid, we have the recurrence $T(n) = O(n) + T(n/2)$, which solves to $T(n) = O(n)$.

Finally, we convert this into an algorithm to find a local minimum (not necessarily internal) of a grid G. Using $O(n)$ probes, we find the node v on the border B of minimum value. If v is a corner node, it is a local minimum and we are done. Otherwise, v has a unique neighbour u not on B. If $v \prec u$, then v is a local minimum and we are done. Otherwise, G satisfies Property (*) (since u is smaller than every node on B), and we call the above algorithm.