

TUTORIAL SHEET 5

1. You are given a positive integer N . You want to reach N by starting from 1, and performing one of the following two operations at each step: (i) increment the current number by 1, or (ii) double the current number. For example, if $N = 10$, you can start with 1, and then go in the sequence 1, 2, 4, 8, 9, 10, or 1, 2, 4, 5, 10. Give an efficient algorithm, which given the number N finds the minimum number of such operations to go from 1 to N .

Solution: We use a recursive algorithm. If N is even, we half it and recursively solve for $N/2$, and at the last step we use doubling. If N is odd, we first subtract 1, recursively solve the problem and then add 1 at the last step.

To prove correctness we use induction on N . If N is odd, the last step must be increase by 1, and by induction, the algorithm is optimal for $N - 1$. So the algorithm is optimal for N as well. Now suppose N is even and consider an optimal algorithm for N . First assume that the last step done by this algorithm is doubling $N/2$ to N . Since the algorithm also has this step as the last step, and by induction hypothesis, the greedy algorithm takes optimal number of steps to reach $N/2$, we see that the algorithm is optimal for N as well.

So the more interesting case is when the optimal algorithm goes from $N - 1$ to N as the last step. Now, the optimal way of reaching N_1 , by induction hypothesis, is given by the algorithm. But the algorithm for $N - 1$ has as its last two steps (note that $N - 1$ is odd), doubling of $(N - 2)/2$ followed by increment by 1. Thus, we see that the optimal algorithm for N has as its last three steps going from $(N - 2)/2$ to $N - 1$ and then incrementing twice. But we can replace these three steps by two steps: first double $(N - 2)/2$ and then increment once. This is a contradiction, and so this case cannot happen. This proves that the algorithm is optimal.

2. Suppose you are given two sets of n points, one set $\{p_1, \dots, p_n\}$ on the $y = 0$ line and the other set $\{q_1, \dots, q_n\}$ on the $y = 1$ line. Now we draw n line segments given by $[p_i, q_i], i = 1, \dots, n$. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time.

Solution: This is like counting number of inversions. First arrange the points by increasing order of p_1, \dots, p_n . Suppose $p_1 < p_2 < \dots < p_n$. Now construct an array storing q_1, \dots, q_n . Then the number of intersections is equal to the number of pairs (i, j) such that $i < j$ but $q_i > q_j$. So use the last exercise below to solve it.

3. You are given a rooted binary tree T where each vertex v has an integer $val(v)$ stored in it (you can assume that all the integers involved are distinct). A vertex v is said to be a *local minimum* if $val(v) \leq val(w)$ for all the neighbours w of v (i.e., its value is at

most the value at its children and at its parent). Show how to find a local minimum in $O(H)$ time, where H is the height of T .

Solution: Start at the root. If it is a local minimum, we can stop. Otherwise there is a child node with value less than that at the root. Now solve recursively at the child node. Since we spend constant time at each node, the running time is $O(n)$.

Solve the same problem when the graph is an $n \times n$ grid graph. An $n \times n$ grid graph has vertices labelled (i, j) , where $1 \leq i, j \leq n$ and (i, j) is adjacent to $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$, $(i, j + 1)$ (with appropriate restrictions at the boundary). The time taken by the algorithm should be $O(n)$.

Solution: We will use divide and conquer and the recurrence $T(n) = T(n/2) + O(n)$. This is a tricky exercise and the recursive problem needs to be defined carefully.

So when we call $\text{FindLocalMinimum}(G)$ for a grid G , we shall ensure two things : (i) there exists a local minimum in the interior of G , (ii) the minimum value among the boundary squares of G is not a local minimum (interior of a grid is the set of squares which are not on the boundary). Further the algorithm is supposed to output a non-boundary local minimum. We can satisfy this condition initially by adding an extra layer of ∞ on the boundary around the input grid.

Now in the recursive case, we divide the grid G into 4 $n/2 \times n/2$ grids. Let X be the set of squares on the boundary of any of these grids – i.e., the squares on the boundary of the original grid and the squares on the middle horizontal and vertical lines in G .

Let p be the square in X with the minimum value. If it is not on the boundary of G and is a local minimum, we output it (and stop). If p is on the boundary of G , we know that it is not a local minimum (by property (ii) above). Now let G_1 be a grid which contains a square adjacent to p and has smaller value. We recurse on the grid G_1 (including the nodes in X which lie on the boundary of G_1). Check that both the conditions (i) and (ii) hold for G_1 .

4. (a) Let $n = 2^l - 1$ for some positive integer l . Suppose someone claims to hold an unsorted array $A[1 \dots n]$ of distinct l -bit strings; thus, exactly one l -bit string does not appear in A . Suppose further that the only way we can access A is by calling the function $FB(i, j)$, which returns the j^{th} bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to FB .

Solution: We use the recurrence $T(n) = T(n/2) + O(n)$. Find the first bit of all the numbers. One of the following two cases will happen : (i) There are 2^{l-1} numbers with first bit 1, and $2^{l-1} - 1$ numbers with first bit 0: in this case recurse on the latter set of numbers (from now on ignore the first bit, and so these 2^{l-1} numbers will be distinct and exactly one $l - 1$ bit number will be missing, (ii) this case is symmetric and handled in the same manner.

- (b) (**Hard**) Now suppose $n = 2^l - k$ for some positive integers k and l , and again we are given an array $A[1 \dots n]$ of distinct l -bit strings. Describe an algorithm to find the k strings that are missing from A using only $O(n \log k)$ calls to FB .

Solution: We use the same idea, but details are more non-trivial. Our recursive procedure $Find(S, h, m)$ will have three arguments – S will be a set of distinct h bit numbers, with m numbers missing (i.e., $|S| = 2^h - m$) and we want to output these m numbers). Note that initially we call $Find(A, l, k)$, where A is the set of all numbers in the array. Let us describe the procedure $Find(S, h, m)$. As above, we look at the first bit of all the numbers in S . Let S_1 be the ones which have first bit as 1, and S_0 be the ones with first bit being 0. Say $|S_1| = 2^{h-1} - m_1$, $|S_0| = 2^{h-1} - m_0$. Then $m_0 + m_1 = m$. Now we ignore the first bits from S_0 and S_1 and think of these as $h - 1$ bit numbers. If $m_0 \neq 0$, we recursively call $Find(S_0, h - 1, m_0)$. Similarly, if $m_1 \neq 0$, we recursively call $Find(S_1, h - 1, m_1)$. We now analyze the running time. First observe that the running time is at most $O(nl)$. Indeed, the recursion tree has depth l and we pay $O(n)$ at each level. So if $k \geq 2^{l-1}$, we are done. Since $n + k = 2^l$, we can assume from now on that $n \geq 2^{l-1}$. So we will now show that the running time is $O(2^l \log k)$. Let $T(l, k)$ be the running time when we have l bit strings out of which k strings are missing. The recurrence is

$$T(l, k) = T(l - 1, k_1) + T(l - 1, k_2) + 2^l,$$

where $k_1 + k_2 = k$. We assume $k_1, k_2 > 0$, otherwise we don't have a recursive term for this part. We show by induction that the solution is $T(l, k) \leq 2^{l+1} \log k$, where all logarithms are to the base 2. By induction assume $T(l - 1, k_1), T(l - 1, k_2)$ are of this form. Also assume that $k_1 \leq k_2$. In this case, $k_1 \leq k/2$ and so $\log k_1 \leq \log k - 1$. So we get (by induction hypothesis)

$$T(l, k) \leq 2^l \log k_1 + 2^l \log k_2 + 2^l \leq 2^l (\log k - 1) + 2^l \log k + 2^l = 2^{l+1} \log k.$$

5. You are given n charged particles located at points with coordinates $\{1, 2, \dots, n\}$ on the real line. The point at coordinate i has charge q_i . Note that the total Coulomb force on a particle at location i is given by

$$-\sum_{j < i} \frac{q_i q_j}{(i - j)^2} + \sum_{j > i} \frac{q_i q_j}{(i - j)^2}.$$

Give an $O(n \log n)$ time algorithm to compute the total force on each of the particles.

Solution: Construct two polynomials: the first one is $q_1 + q_2x + \dots + q_n x^{n-1}$ and the second one is $x + x^2/2^2 + x^3/3^2 + \dots + x^{n-1}/(n-1)^2$. What is the coefficient of x^k when we multiply the two polynomials? Similarly, use the polynomial $q_n + q_{n-1}x + q_{n-2}x^2 + \dots + q_1 x^{n-1}$.

6. We are given a sequence of n distinct integers a_1, \dots, a_n in an array. An inversion is defined as a pair (i, j) such that $i < j$ but $a_i > a_j$. Give an $O(n \log n)$ time to count the number of inversions in the array.

A strong inversion is defined as a pair (i, j) such that $i < j$ but $a_i > 2a_j$. Give an $O(n \log n)$ time to count the number of strong inversions in the array.

Solution: For the first one, use divide and conquer. Divide the arrays into two halves A_L and A_R and recursively count the number of inversions in each. We shall also sort the arrays A_L and A_R in the recursive calls. Now, we scan A_L and A_R from left to right and count the number of inversions involving an index from A_L and an index from A_R . In fact this can be done while we are merging the two arrays. Suppose when we insert an element $A_L[i]$ in the merged array and we already have j elements from A_R in the merged array. Then we know that there are j inversions involving $A_L[i]$.

For the second part, we first recursively solve for A_L and A_R . Now we will merge them into one sorted array as before. But before this merge, we will count the number of strong inversions. Now create a new array A'_L obtained by halving each element of A_L . Now merge A'_L and A_R while counting inversions as before.