CS601 Algorithms and Complexity **2021 Jul-Nov**

Midsem Solution

Total Marks: 40 *Deadline:*

It's quite hard to verify pseudocodes, and they are also prone to errors. You are strongly encouraged to describe you algorithms in words as much as possible, as is done here.

Part 1 Que 2. Recall the divide and conquer approach for finding the square of an integer. Show that if in the divide step, you divide the integer into 4 parts, then you can find the square of an n bit integer in time $O(n^{1.4037})$. You can directly use anything proved in the class. You can assume that the multiplication or division of an n bit integer with a constant can be done in $O(n)$ time.

Ans. We are going to follow the exact same approach that we used to achieve $O(n^{1.46})$ running time by dividing the integer into 3 parts. Let our n bit integer be a. Let us divide into four parts as follows.

$$a = a_3\ 2^{3n/4} + a_2\ 2^{2n/4} + a_1\ 2^{n/4} + a_0,$$

where each of a_0, a_1, a_2, a_3 is an $n/4$ bit integer. Squaring both the sides we get,

$$a^2 = a_3^2\ 2^{6n/4} + 2a_3a_2\ 2^{5n/4} + (a_2^2 + 2a_3a_1)\ 2^{4n/4} + 2(a_2a_1 + a_3a_0)\ 2^{3n/4} + (a_1^2 + 2a_2a_0)\ 2^{2n/4} + 2a_1a_0\ 2^{n/4} + a_0^2. \tag{1}$$

Now, the question is to compute these 7 terms shown in red, how many squarings of $n/4$ bit integers are needed (together with some $O(n)$ operations)? It turns out that 7 squarings will be good enough. For this, we need to see computation of these 7 terms as as a squaring of a polynomial. More precisely let us define a polynomial

$$A(x) = a_3\ x^3 + a_2\ x^2 + a_1\ x + a_0.$$

Its square will be given by

$$(A(x))^2 = a_3^2\ x^6 + 2a_3a_2\ x^5 + (a_2^2 + 2a_3a_1)\ x^4 + 2(a_2a_1 + a_3a_0)\ x^3 + (a_1^2 + 2a_2a_0)\ x^2 + 2a_1a_0\ x + a_0^2.$$

Thus, finding the square of $A(x)$ is equivalent to compute the desired 7 terms. We have a three step plan for computing the square of $A(x)$:

1. Compute the evaluations of the polynomial $A(x)$ at 7 different values of x.
2. Square these 7 numbers to get 7 evaluations of $(A(x))^2$.
3. Since $(A(x))^2$ is a degree 6 polynomial, we can obtain its coefficients from its 7 evaluations (**this is the crucial point from where the magic number of 7 comes**).

Step 1: The 7 values for x can be chosen arbitrarily. We choose $x = -3, -2, -1, 0, 1, 2, 3$. To compute $A(x)$ for any of these values of x, we first need to multiply a_3, a_2, a_1, a_0 with some constants and then add them up. From the assumption in the question, constants can be multiplied in $O(n)$ time. After that there are 3 additions, which can also be done in $O(n)$. Thus, step 1 can be finished in $O(n)$ time.

Step 2: We need to find squares of $A(-3), A(-2), A(-1), A(0), A(1), A(2), A(3)$. From step 1, it can be seen that these numbers have at most $n/4+6$ bits. Thus, each of these numbers can be written as $(\alpha\ 2^6 + \beta)$, where α is $n/4$ bits and β is bounded by 2^6. For squaring $(\alpha\ 2^6 + \beta)$, we need to compute α^2 plus a few additional $O(n)$ time operations. In summary, step 2 involves computing squares of 7 $n/4$ bit numbers and some other operations doable in $O(n)$ time.

Step 3: Given $A(-3)^2, A(-2)^2, A(-1)^2, A(0)^2, A(1)^2, A(2)^2, A(3)^2$, we want to compute coefficients of $(A(x))^2$. Lets us define $B(x) = (A(x))^2$. In other words, we have 7 evaluations for the degree 6 polynomial $B(x)$ and we want to compute its coefficients. Suppose coefficients of $B(x)$ are given by

$$B(x) = b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0.$$

The relation between coefficients and evaluations of $B(x)$ are given as below.

$$\begin{pmatrix} B(-3) \\ B(-2) \\ B(-1) \\ B(0) \\ B(1) \\ B(2) \\ B(3) \end{pmatrix} = \begin{pmatrix} 3^6 & -3^5 & 3^4 & -3^3 & 3^2 & -3 & 1 \\ 2^6 & -2^5 & 2^4 & -2^3 & 2^2 & -2 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2 & 1 \\ 3^6 & 3^5 & 3^4 & 3^3 & 3^2 & 3 & 1 \end{pmatrix} \begin{pmatrix} b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix}$$

Let us call the 7×7 matrix above as M. Then coefficients of $B(x)$ can be computed as

$$M^{-1} \begin{pmatrix} B(-3) \\ B(-2) \\ B(-1) \\ B(0) \\ B(1) \\ B(2) \\ B(3) \end{pmatrix}$$

Note that all the entries in M^{-1} are constants. The evaluations of $B(x)$ have $n/2 + 12$ bits at most. Thus, the above matrix vector product can be done in $O(n)$ time. Coefficients of $B(x)$ are indeed the desired 7 terms in Equation (1).

Once we compute the 7 terms in Equation (1), we just need to do some shifts and additions. All this can be done in $O(n)$. To summarize, to find a^2, we did squares of 7 $n/4$ bit integers together with some $O(n)$ operations. Thus, for the time complexity we can write

$$T(n) = 7T(n/4) + O(n).$$

Solving it, we get

$$T(n) = O(n^{\log_4 7}) = O(n^{(\log_2 7)/2}) \approx O(n^{1.4037}).$$

Part 1 Que 3. Suppose you have a movable shop that you can take from one place to another. You usually take your shop to one of the two cities, say A and B, depending on whichever place has more demand. Suppose you have quite accurate projections for the earnings per day in both the cities for the next n days. However, you cannot simply go to the higher earning city each day because it takes one whole day and costs c to move from one city to the other. For example, if you are in city A on day 5 and want to move to city B, then on day 6 you will have no earnings, you will pay a cost of c and on day 7 you will have earnings of city B. Design a polynomial time algorithm that takes as input the moving cost ,the earnings per day in the two cities say, $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$, and outputs a schedule for the n days that maximizes the total earnings. Assume that you can start with any of the two cities on day 1, without costing anything.

Ans. 7 marks for the optimal earning + 3 marks for the optimal schedule

A bit of thinking tells you that greedy ideas won't work. For example, you can try the following greedy strategy: if you are in city A on day i, then compare $a_{i+1} + a_{i+2}$ (earnings for next two days if you don't move to B) with $-c + b_{i+2}$ (earnings for next two days if you move to B). Depending on which is greater, you decide about moving. Such a strategy is bound to fail because we are not considering the values for future days. For example, suppose $-c + b_{i+2} > a_{i+1} + a_{i+2}$ and thus, you decided to move to B on day $i+1$. It is possible that a_{i+3} is a huge number and you will miss on it because you are in B and cannot be in A on day $i+3$. Similarly, any short-sighted strategy will fail.

Now, we will apply the dynamic programming approach. The set of possible solutions can be divided into two categories: those which start with city A and those which start with city B. Let us try to find the optimal schedule in each of these categories. Suppose we start with city A on day 1. Now, either we can continue to be in A on day 2 or we can move to B by paying cost c and earn there on day 3. We would like to take the maximum of the two scenarios. However note that the two scenarios don't seem to be representable by a subproblem. It would be **wrong** to represent it as $a_1 + \max(OPT(2), -c + OPT(3))$, because we don't know from which city the optimal solutions for $OPT(2)$ and $OPT(3)$ will start with.

What we need to do is to keep computing **two** different optimal functions defined as follows:

OptA(j) = the optimal earning from day j to day n assuming that we earn in city A on day j.

OptB(j) = the optimal earning from day j to day n assuming that we earn in city B on day j.

Now, we can write the recursive formulas for these two quantities with the above logic. If you are in A on day j, either you can continue in A or you can move to B by paying cost c and skipping day $j+1$. We need to take maximum of the two scenarios. Thus,

$$\text{OptA}(j) = a_j + \max(\ \text{OptA}(j+1),\ -c + \text{OptB}(j+2)\ ).$$

Similarly,

$$\text{OptB}(j) = b_j + \max(\ \text{OptB}(j+1),\ -c + \text{OptA}(j+2)\ ).$$

These numbers can be computed from $j = n$ to 1. Initialization should be $\text{OptA}(n) = a_n$ and $\text{OptB}(n) = b_n$. Once we have computed all these values, we need to take max(OptA(1), OptB(1)) to get the actual optimal value.

To compute the optimal solution, we do the following. We will compute $s_1, s_2, \ldots, s_n \in \{A, B, -\}$.

- If OptA(1) > OptB(1) then set $s_1 \leftarrow$ A, otherwise set $s_1 \leftarrow$ B.
- for ($j = 2$ to n):
  - if $s_{j-1} =$ A then $s_j \leftarrow$ A or –, depending on whether OptA(j) is larger or $-c$+OptB($j+1$),
  - if $s_{j-1} =$ B then $s_j \leftarrow$ B or –, depending on whether OptB(j) is larger or $-c$+OptA($j+1$),
  - if $s_{j-1} = -$ then $s_j \leftarrow$ opposite of s_{j-2}.

Part 2 Que 2. Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, we want to check whether there exists an undirected graph G on n vertices whose vertex degrees are precisely $d_1, d_2, \ldots, d_n$ (that is the ith vertex has degere d_i) and construct such a graph if one exists. G should not have multiple edges between the same pair of vertices and should not have self-loop edges having same vertex as the two endpoints.

Example 1: (2, 1, 3, 2) . The graph with the set of edges $(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)$ has this degree sequence.

Example 2: (3, 3, 1, 1) . There is no graph on four vertices having these degrees.

Let us propose the following **greedy** algorithm: Start with the vertex v_1, we need to assign d_1 neighbors to it. Among the remaining vertices, choose top d_1 vertices with highest degrees (break ties arbitrarily, if any) and assign them as neighbors for v_1. Now, delete v_1, reduce the degrees of the neighbors of v_1 by 1 each and find the remaining graph on last $n-1$ vertices recursively.

Boundary cases: If all degrees are zero, then return empty set of edges. If $d_1 > n - 1$ or any degree is negative, then say that no such graph exists.

Greedy on example 1 will have the following recursion chain:

→ $(d_1 = 2, d_2 = 1, d_3 = 3, d_4 = 2)$: Add edges $(v_1, v_3), (v_1, v_4)$.

→ $(d_2 = 1, d_3 = 2, d_4 = 1)$: Add edges (v_2, v_3).

→ $(d_3 = 1, d_4 = 1)$: Add edges (v_3, v_4).

→ $(d_4 = 0)$: No more edges. Graph construction done.

Prove that the greedy algorithm will always work correctly. You just need to prove that if there exists a graph G for the given list of degrees, then there also exists a graph G' which has the same list of degrees and where v_1 is connected to the top d_1 vertices with highest degrees. The rest of the argument follows by induction.

Ans. Suppose there exists a graph G with the given list of degrees. If neighbors of v_1 in G are already the top d_1 with highest degree then we are done. Otherwise there is a vertex from the top d_1, say u, which is not a neighbor of vertex v_1. Since the degree of v_1 is d_1, it must have another neighbor w which is not among the top d_1 vertices. Now, the idea is to remove w from the neighborhood of v_1 and add u. However, this process will change the degrees of w and u, which we do not want. We will fix it by swapping another pair of edges.

We know that $deg(u) \geq deg(w)$ (u is in top d_1, while w is not). Moreover, w is connected with v_1, while u is not. Hence, there must be a neighbor of u which is not a neighbor of w. Let this vertex be s. Finally, we add/remove following four edges.

- Add (v_1, u)
- Remove (v_1, w)
- Add (s, w)
- Remove (s, u)

Clearly, after this process the degrees of four vertices remain same as before. In conclusion, we have created a new graph, where v_1 has one more neighbor from the top d_1. We can keep repeating this process till all the neighbors of v_1 are from the top d_1. In the end, we will have the desired graph G'. This finishes the main claim.

As we said the correctness of the greedy algorithm now follows from induction. Assume that the algorithm is correct for $n-1$ or smaller number of vertices. Base case of 1 vertex is easy to verify.

From the above arguments, if there is a graph G with the given degree sequence D, then we can safely assume that in G, v_1 is connected with the top d_1 vertices among the remaining. Consider the subgraph H of G obtained by deleting v_1. The degree sequence of H, say $\tilde{D}$, can be obtained by first removing d_1 and then reducing the top d_1 degrees by one each. We conclude that a graph with degree sequence D exists if and only if a graph with degree sequence $\tilde{D}$ exists. By induction hypothesis we can say that our algorithm can correctly work on $\tilde{D}$. And, thus, the algorithm is also correct on D.

Part 2 Que 3. For a pair of 2-dimensional points (a, b) and (c,d), we say that the first one is better than the second one if $a \geq c$ and $b \geq d$. Given a set of n points in 2-dimensions, our goal is to find the number of pairs of points where the first one is better than the second one.

Example: $p_1 = (1, 4), p_2 = (2, 5), p_3 = (4, 1), p_4 = (5, 4)$. Here there are 3 pairs: $(p_4, p_3), (p_4, p_1), (p_2, p_1)$.

We want something better than the trivial $O(n^2)$ time algorithm. We propose the following divide and conquer approach.

First sort the points in the decreasing order of the x-coordinate. Idea is to recursively find the desired number of pairs among the first $n/2$ points and among the last $n/2$ points. What remains is to count the desired number of pairs where one point comes from first $n/2$ and the other comes from last $n/2$.

Fill in the remaining details of this divide and conquer algorithm. Do you need something more from the recursive calls? Give a running time analysis for the complete algorithm. Full marks for $O(n \log n)$, only six marks for $O(n \log^2 n)$.

Ans. For a point p, let us denote its x and y coordinates as $x(p)$ and $y(p)$, respectively. As mentioned in the question, the first two steps in the algorithm are as follows.

- Sort the points in the decreasing order of the x-coordinate. If x-coordinates of two points are equal, the sort in decreasing order of y-coordinate.
- Let S_1 be the set of first $n/2$ points in this order and S_2 be the set of last $n/2$ points. Recursively compute the desired number of pairs in S_1 and S_2 separately.

Note that for any $p \in S_2, q \in S_1$, we cannot have p better than q, because of the way we sorted S (we are assuming points are distinct). Thus, we only need to count pairs (p, q) such that $p \in S_1, q \in S_2$. Clearly, going over all pairs is not an option. First note that for any such pair, the deciding factor will be the y

coordinates of p and q, because we already know $x(p) \geq x(q)$. It would be helpful if S_1 and S_2 are sorted with respect to the y coordinates.

- Sort both S_1 and S_2 in increasing order of y coordinates. Let $p_1, p_2, \ldots, p_{n/2}$ and $q_1, q_2, \ldots, q_{n/2}$ be the points in S_1 and S_2 in increasing order of y coordinates.

We are going to traverse over S_1 and S_2 in this order. Maintain two pointers: j for S_1 and i for S_2.

Count(S_1, S_2)**:**

1. Initially, $i = 0$ and $j = 1$.
2. For the current value of j, keep increasing i till the first point that $y(q_i) > y(p_j)$.
3. Note that p_j is better than exactly $i - 1$ points in S_2. So, we add $i - 1$ to the count.
4. Increase j by 1, and go to step 2.

Note that the pointer i is always increasing (or stays unchanged) because both S_1 and S_2 are in increasing order of y coordinates. The process scans both S_1 and S_2 from left to right and thus takes $O(n)$ time.

Complexity: Initial sorting with respect to the x-coordinates was a one time process. We will add it separately. Let $T(n)$ be the time complexity of the rest of the algorithm on input of size n. We first spend time $2T(n/2)$ for recursing on S_1 and S_2. Sorting S_1 and S_2 w.r.t. y-coordinates takes $O(n \log n)$ time. Finally, the Count(S_1, S_2) subroutine takes $O(n)$ times. We get the following recurrence.

$$T(n) = 2T(n/2) + O(n \log n).$$

Solving this, we get $T(n) = O(n \log^2 n)$.

To improve the running time to $O(n \log n)$, we need a small trick. We are going to shift the sorting w.r.t. y-coordinates to the recursive call. The idea is to spend only $O(n \log n)$ time overall on all the sorting. Below are more details. We define two procedures: MergeY and Count-and-Sort. Count-and-Sort is our main algorithm and it will use MergeY and Count as subroutines.

MergeY:

Input: takes two lists of points sorted with respect to y-coordinates.

Output: Union of the two lists sorted with respect to y-coordinates. This is done via the standard $O(n)$ merge procedure in merge sort.

Count-and-Sort:

Input: a list S of 2D points sorted in decreasing order of x-coordinates.

Output: returns the desired number of pairs of points from S, and also sorts S in increasing order of y-coordinates.

1. Divide S into two halves S_1 and S_2.
2. $c_1 \leftarrow$ Count-and-Sort(S_1) (c_1 is the count and S_1 is sorted w.r.t. y-coordinates).
3. $c_2 \leftarrow$ Count-and-Sort(S_2) (c_2 is the count and S_2 is sorted w.r.t. y-coordinates).
4. $c \leftarrow c_1 + c_2+$ Count(S_1, S_2).
5. $S \leftarrow$ MergeY(S_1, S_2).
6. return c.

Complexity: Now, we can write our recurrence as

$$T(n) = 2T(n/2) + O(n).$$

This is because besides the two recursive calls, we are only using MergeY and Count subroutines, both of which are $O(n)$. We get $T(n) = O(n \log n)$.