CS 344: Design and Analysis of Computer Algorithms

Instructor: Kangning Wang

Homework 2: Due on **February 21, 2025**

Acknowledgment: Our team members are Alexander Lin (al1655), Pranav Tikkawar (pt422), and Ivan Zheng (iz72) .

Problem 1

Solve these recurrence relations and represent the answers in the Θ notation. In each of these questions, the given formulas are for $n > 100$, and we know $T(n) = 1$ for all $n \leq 100$. All fractional inputs are rounded down. Other than the first one below, you can write down your answer without providing your proof (which can be tedious), but you should be able to give a proof if asked. Example: $T(n) = 2T(n/2) + \frac{n}{\ln n}$ gives $T(n) = \Theta(n \log \log n)$.

1. $T(n) = T(0.4n) + T(0.3n) + T(0.2n) + n$. (Prove this one formally using strong induction.)
2. $T(n) = T(n/3) + 10$.
3. $T(n) = 7T(n/2) + n^2$.
4. $T(n) = T(n/3) + T(2n/3) + n \ln n$.
5. $T(n) = 2T(n-1) + 1$.
6. $T(n) = T(n-1) + T(n-2) + 1$.
7. $T(n) = T(\sqrt{n}) + 344$.

Solution.

1. $\Theta(n)$ because proof by induction. Base case: for n $\leq$ 100, T(n) = 1. Inductive hypothesis: assume T(k) = $\Theta(k)$ for all k $<$ n. Inductive step: T(n) = $\Theta(0.4n) + \Theta(0.3n) + \Theta(0.2n) + n = \Theta(n)$. Thus, T(n) = $\Theta(n)$.
2. $\Theta(logn)$ because using Master Theorem, a = 1, b = 3, f(n) = 10, c = 0, $\log_b$a = 0. Thus, $\Theta(n^{log_3 1} logn) = \Theta(logn)$.
3. $\Theta(n^{log_2 7})$ because using Master Theorem, a = 7, b = 2, f(n) = n^2, c = 2, $\log_b$a $\approx$ 2.807. Thus, T(n) = $\Theta(n^{log_2 7})$.
4. $\Theta(nlog^2 n)$ because the depth is $\log_3$n and each level's work is nlnn. Thus, $\Theta(nlnn \cdot log_3 n)$ = $\Theta(nlog^2 n)$.

5. $\Theta(2^n)$ because T(n) = 2T(n - 1) + 1 and T(n - 1) = 2T(n -2) + 1. Expanding, 2(2T(n - 2) + 1) + 1 = 4T(n - 2) + 3. T(n - 2) = 2T(n - 3) + 1, expanding again, 4(2T(n - 3) + 1) + 3 = 8T(n - 3) + 7. We can see the formula of T(n) = 2^kT(n - k) + (2^k - 1). Assuming T(0) = 0, T(n) = 2^n - 1. Thus, $\Theta(2^n)$.

6. $\Theta(\phi^n)$ where ϕ = $\frac{1+\sqrt{5}}{2}$ because we can see the ressemblance to the Fibonacci sequence in T(n).

7. $\Theta(loglogn)$. Assume n = 2^{2^k}, then $\sqrt{n} = 2^{2^{k-1}}$. T(n) = T($n^{\frac{1}{2}}$) + 344. Expanding, T($n^{\frac{1}{2}}$) = T($n^{\frac{1}{4}}$) + 344 $\rightarrow$ T($n^{\frac{1}{4}}$) = T($n^{\frac{1}{8}}$) + 344. Since n = 2^{2^k}, $log_2(log_2(n))$ = k. Thus, T(n) = 344 $\cdot$ $log_2(log_2(n))$.

Problem 2

Given a directed graph $G = (V, E)$, we want to count the number of walks of length k from u to v for a given pair of vertices (u, v). We are only interested in this number modulo a constant p, so that each basic arithmetic operation only takes constant time. Let A be the adjacency matrix of G, and let $B = A^k$.

1. Prove using induction that B_{uv} is exactly the number of walks of length k from u to v.
2. It is known that $n \times n$ matrix multiplication can be done in $O(n^{2.372})$ time. Design an algorithm that counts the number of walks (modulo p) of length k from u to v in $O(n^{2.372} \log k)$ time, where $n = |V|$.

Solution.

1. We need to prove that B_{uv} is the number of walks of length k from u to v. Base case: for k = 1, B = A. By definition, A_{uv} = 1 if there is an edge from u to v, which is the number of walks of length 1 from u to v. Inductive hypothesis: assume that for some $k \geq 1$, $(A^k)_{uv}$ is the number of walks of length k from u to v. Inductive step: to show A^{k+1} is the number of walks of length k from u to v, using matrix multiplication, $(A^{k+1})_{uv} = \sum_{w \in v}(A^k)_{uw} \cdot A_{wv}$, where $(A^k)_{uw}$ is the number of walks of length k from u to w and A_{wv} is 1 is there is a direct edge from w to v. Summing over all intermediate vertices w, this gives the total number of walks of length k + 1 from u to v.

2. Initialize B as identity matrix. Using exponentiation by squaring:
   while $k > 0$:
   if k is odd, multiply B = $(B \cdot A)$ mod p
   update A = $(A \cdot A)$ mod p
   divide k by 2
   return B_{uv} mod p
   Each matrix multiplication takes $O(n^{2.372})$ time and the number of multiplications by exponentiation by squaring is $O(logk)$.

Problem 3

We have an array of n distinct integers $(a_1, a_2, \ldots, a_n)$. An inversion is a pair of indices (i, j) with $i < j$ and $a_i > a_j$. The weight of an inversion (i, j) is defined as $(a_i - a_j)^2$. Give an algorithm that computes the sum of weights of all inversions in $O(n \log n)$ time.

Solution. We can do this by using a modified merge sort. The idea is to count the number of inversions while merging two sorted halves of the array. We can also keep track of the sum of weights of inversions during this process. The algorithm works as follows:

1. Split the array into two halves.
2. Recursively count the inversions and their weights in each half.
3. Merge the two halves while counting the inversions that cross the split.
4. During the merge step, for each element in the left half, count how many elements in the right half are smaller and calculate their weights.
5. The total weight of inversions is the sum of the weights from the left half, right half, and the cross-inversions.
6. Return the total weight.

The time complexity of this algorithm is $O(n \log n)$ because the merge sort algorithm runs in $O(n \log n)$ time, and we are only adding a constant amount of work during the merge step. The space complexity is $O(n)$ for the temporary arrays used during the merge step.

Problem 4

Given n points $(p_1, p_2, \ldots, p_n)$ on a Euclidean plane in *general position* (that is, no two points coincide and no three points are on the same line), find three different points p_i, p_j, p_k among them to minimize the perimeter of the triangle with p_i, p_j, p_k as its vertices. Your algorithm should run in $O(n \log n)$ time. You can assume that among the points $(p_1, p_2, \ldots, p_n)$, all x-coordinates are distinct and all y-coordinates are distinct.

Solution.
The solution is to essentially divide and conquer the problem recursively into smaller problems. In the end, the final complexity of the algorithm will be $O(n \log n)$.

We follow a similar idea to the way of finding smallest distance between two points given during lecture. Lets call our set of points P. We sort all points in P by x-coordinate and y-coordinate, which is two operations of order $O(n \log n)$, to be used later. Our algorithm is as follows:

1. If the size of P is less than or equal to some arbitrary limit, lets say ten points, then manually compute the smallest perimeter triangle by enumerating through every combination of three points, returning the triplet and the total perimeter p.

2. Draw a vertical line that divides P into P_1 and P_2. Recursively run this algorithm on P_1 and P_2 to get their triplets and minimum perimeters p_1 and p_2.

3. Let d be the smaller of p_1 and p_2. We now need to see if we can construct a smaller perimeter triangle using points from both P_1 and P_2. Here, we only need to consider points within distance $\frac{d}{2}$ from the line we drew in step 2.

This is because we are considering perimeters of triangles while taking points from both P_1 and P_2. Therefore, if a point is further than $\frac{d}{2}$ from the line, the minimum perimeter triangle we can draw with it has length $p > 2 * \frac{d}{2} = d$, as we must pick a point on the line and another $\frac{d}{2}$ from the line, giving length $\frac{d}{2}$, and the sum of the other two sides drawn by taking any third point must be greater than the first (triangle inequality). We can find these points in $O(n)$ time because we have presorted by x-coordinate.

We can now split this "tape" taken around the line into imaginary squares of side length $\frac{d}{4}$. We can guarantee that there are no more than two points in each of these boxes. This is because if we do, then these three points will construct a triangle with a perimeter smaller than d. To show this is true, we want to construct the largest triangle in the unit square. Consider vectors $\vec{a}, \vec{b}$ as the short sides of the triangle and $\vec{c} = \vec{a} + \vec{b}$ as the long side, where the starting and ending points of the vectors are inside the unit square. Then $\|a+b\|^2 = (a+b) * (a+b) = \|a\|^2 + 2(a * b) + \|b\|^2$. Then we have $\|a\|^2 + \|b\|^2 \leq \|a+b\|^2$. But we want to maximize the sum of all three of these quantities, so clearly we must take $\|a\|^2 + \|b\|^2 = \|a+b\|^2$, which is a right triangle. The largest right triangle in a unit square is clearly when we take our three points at three different corners. If we move any of the two points on the diagonal "inwards" towards the middle point, we will obviously create a smaller right triangle, and if we rotate the right triangle, we shrink all three sides as well. The perimeter of this right triangle is $2 + \sqrt{2} = 3.414...$, so the largest perimeter triangle we can construct in our squares of $\frac{d}{4}$ have perimeter $\frac{3.414...}{4} d < d$. Therefore we cannot have more than 2 points in each box.

Now that we have this established, we only have to check points in these "boxes" up to three rows away. At the fourth row, the smallest distance we can get is $\frac{3}{4}d$, so the smallest triangle we can make has perimeter $p \geq 2 * \frac{3}{4}d > d$. We therefore have to check the closest 16 boxes, or the closest 32 points, as decided by y-coordinate. Fortunately, we have already sorted the y-coordinates, so we can find these in $O(n)$ time. We must check $\binom{32}{2} = 496$ combinations of points and calculate the perimeters of the triangles formed, but this is also $O(n)$ time.

4. Check the closest 32 points and update d if a smaller perimeter triangle is found. Return d.

The overall time complexity of the algorithm is the same as the one presented in lecture: $T(n) = 2T(\frac{n}{2}) + O(n)$ for the algorithm, which by Master theorem is $O(n \log n)$, and $O(n \log n)$ for the two sorts. This gives us an overall time complexity of $O(n \log n)$

Assumptions for Problems 5 and 6

All basic arithmetic operations can be done in constant time, and the fast Fourier transform runs in $O(n \log n)$ time.

Problem 5

The RUCS stock price has an interesting behavior. Every day independently, exactly one of the following events will happen.

- The price doubles ($\times 2$); this happens with probability 0.5.
- The price quadruples ($\times 4$); this happens with probability 0.2.
- The price halves ($\times 0.5$); this happens with probability 0.3.

The price of the RUCS stock is 1 today, and we need to calculate the probability that its price becomes at least p after n days. Design an algorithm for this that runs in $O(n \log n)$ time. $O(n^2)$ running time is also good if $O(n \log n)$ seems out of reach. (Hint: This problem is an application of polynomial multiplication and the fast Fourier transform.)

Solution.
We can describe the probability distribution of the next day given by the current price x as a polynomial $P(x) = 0.5x^2 + 0.2x^4 + 0.3x^{\frac{1}{2}}$.
To calculate the probability that the price of the nth day we can use $P(x)^n$ which is the polynomial multiplication of $P(x)$ with itself n times. The exponents of the polynomial $P(x)^n$ will the price multiplier and the coefficient is the probability of that multiplier.
Directly multiplying will lead to $O(n^2)$ time complexity. However, we can use the fast Fourier transform to do the polynomial multiplication in $O(n \log n)$ time.

Problem 6

There are n stars in a different universe, and all of them are negligibly small in terms of volume. They happen to be in the same straight line, and their locations are $1, 2, \ldots, n$. (Thus the distance between each consecutive pair is 1.) The star k has mass $m_k > 0$.

You are interested in knowing the net gravity F_k that each star k receives. It is defined as

$$F_k = -\sum_{j<k} \frac{Gm_j m_k}{(k-j)^2} + \sum_{j>k} \frac{Gm_j m_k}{(k-j)^2},$$

where G is the gravitational constant.

Design an algorithm to compute all of $F_1, F_2, \dots, F_k$ in $O(n \log n)$ time. (Hint: This problem is an application of polynomial multiplication and the fast Fourier transform.)

Solution.

We can represent the net gravity F_k as a polynomial. The idea is to represent the mass of each star as a polynomial and then use polynomial multiplication to compute the net gravity for all stars.
When we consider this equation we can rewrite it as: $F_k = Gm_k \left(\sum_{d=1}^{n-k} m_{k+d} w_d - \sum_{d=1}^{k-1} m_{k-d} w_d \right)$ where $w_d = \frac{1}{d^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2}$.
We can represent the mass of the stars as a polynomial $M = m_1 + m_2 x + m_3 x^2 + \ldots + m_n x^{n-1}$ and the weights as a polynomial $W = 1 + \frac{1}{2^2} x + \frac{1}{3^2} x^3 + \ldots + \frac{1}{n^2} x^n$. We can then use polynomial multiplication to compute the net gravity for all stars since

$$S_{forward} = m_1 w_1 + m_2 w_2 + \ldots + m_n w_n = M * W$$
$$S_{backward} = m_n w_1 + m_{n-1} w_2 + \ldots + m_1 w_n = M_{reversed} * W$$

Here, we have left out the powers of x involved in each term, and $M_{reversed}$ is simply reversing the elements of M. We can then use our S to compute the net gravity for all stars.

$$F_k = Gm_k(S_{forward}[k] - S_{backward}[n-k-1])$$

Here, $S[n]$ indicates that we are taking the elements of S starting from term n. We are obviously taking only coefficients and no longer care about the powers of x.
We can use the fast Fourier transform to compute the polynomial multiplication needed to calculate $S_{forward}$ and $S_{backward}$ in $O(n \log n)$ time. The construction of the polynomials M and W will also be $O(n)$, and the final process of putting together the terms to find our F_k is also $O(n)$ since we have computed all the necessary elements already, so the arithmetic is constant time. The final time complexity of the algorithm is therefore $O(n \log n)$.