Homework 5: Divide & Conquer, Randomized Algorithms, and Fast Fourier Transform

Instructions: **Due 02/18/25 11:59pm**

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Please try to write concise responses.
- You should not use only pseudocode to describe your algorithms.
- Unless otherwise stated, saying log means base 2.
- Comparisons and basic arithmetic (addition, subtraction, bit-shifting) operations take $\mathcal{O}(1)$ time.

Q1 A monkey currently has b rotten bananas and seeks to get rid of as many rotten bananas as possible. There are n baskets laid out in order and each basket has a multiplier that can be applied on the current amount of bananas the monkey has. For example, using a basket with a multiplier of 0.5 will halve the amount of rotten bananas the monkey has. These multipliers are stored in the array $M = [m_1, m_2, \ldots, m_n]$. The monkey is allowed to start at some basket i and end at some basket j, where $1 \leq i \leq j \leq n$, applying multipliers to his bananas *without* skipping any baskets. For example, given the multipliers $M = [3, 0.1, 0.2, 100, 0.75, 4, 0.1]$ and $b = 100$ bananas, the monkey's best course of action would be to start at position 2 and stop at position 3. The number of rotten bananas that the monkey would have remaining is then $100(0.1)(0.2) = 2$.

You will design an efficient *divide-and-conquer* algorithm that performs the following task:

Input: An amount of bananas $b > 0$, an array of multipliers $M = [m_1, m_2, \ldots, m_n]$
Output: The maximum amount of bananas the monkey can get rid of.
(doesn't have to be an integer)

(a) Describe your algorithm.

Note: Feel free to merge (a) and (b) if it's easier to explain that way.

Our solution will divide our space into half and find the min the monkey would be left with in the left and right half array. As it returns i and j we also know whether its connectable. We keep track of the subarrays themselves using i', j' which would be initialized as $i' = 0, j' = n - 1$. So we first calculate the middle index

$$k = (j + i)/2$$

Then we recursively call the left and right subarray as,

$$i_l, j_l, \min_l = T(i', k)$$
$$i_r, j_l, \min_r = T(k + 1, j')$$

Now we have the minimum of both the left and right half. However there can still exist a minimum crossing the two subarrays. To compute this what we'll do is start at the center and expand to the left and right until we get the minimum product.

Now we compuare the three minimums and return the smallest one along with the indices and minimum value.

In the end we will have i, j, min of the smallest sub array. If min is greater than or equal to 1 return 0 else return $b \times (1 - m)$

(b) Show the correctness of your algorithm.

The algorithm works by decomposing the problem into dividing the problem into two different parts - finding the i, j that returns the minimum product in each side of the array as well as the center. Given that we divide the array by half we can have three different cases of the minimum product,

1. Minimum of the left subarray: In this case our left recursive call $T(l', k)$ will compute this.

2. Minimum of the right subarray: Our right recursive call will compute this.
3. Minimum spanning both subarrays: We also need to compute the spanning minimum. For this we start at the center and iteratively expand left and right until we get the minimum (essentially go on on both sides until we reach something greater than or equal to 1). When this loop ends we'll get the minimum that spans both the arrays.

Using this information for the array that we are analyzing we'll identify the smallest product possibly within the array recursively.

(c) Provide a recurrence for your algorithm and use it to analyze the runtime.

At each stage $T(n)$ we decompose the problem by 2 by calling the function for each sub array and we also expand from the center which would be a worst case $O(n)$ as we might have to consider the entire left and right half. So our recurrence would look like,

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

We see that $\log_b(a) = \log_2(2) = 1$ and $d = 1$ so using master theorem we have,

$$T(n) = \mathcal{O}(n \log n)$$

Q2 Suppose you are walking down a long road, whose length you don't know in advance. Along the road are houses, which you would like to photograph with your very old camera. This old camera allows you to take as many pictures as you want, but only has enough memory to store one picture at a time. The street contains n houses, but you don't know n before you reach the end of the street.

Your goal is to end up with one photo in your camera, where that photo is equally likely to show any of the n houses. One algorithm for achieving this goal is to walk all the way down the street, counting houses, so that we can determine n. Then we roll an n-sided die, where X denotes the roll outcome. Then we walk to the house numbered X and take its picture. However, you're a busy person and you don't want to walk down the street again. Can you achieve your goal by walking up the street only once?

(a) Describe a randomized algorithm to solve this problem.

Hint: Your algorithm will involve replacing the current photo stored in memory with some probability as you walk (only once) down the street.

First let us begin by having n be the variable that keeps track of how many houses we've seen until how. First we start at house 1 and take a picture. For the next house at $n = 2$ we'll roll a dice with $n = 2$ sides. If it lands at 1 then we'll take a picture of house 2 else we'll leave the picture unchanged. So for any $n'th$ house we'll roll a dice with n sides and if it lands on 1 then we'll take the picture of the $n'th$ house else we'll leave the image unchanged. We'll continue this until we reach the last house.

(b) Prove that, for all i, $\mathbb{P}(i\text{-th item is output}) = 1/n$.

For any i the probability that the $i'th$ house is the one that is chosen would be $1/i$ when we're at the i'th house and for every house after than we need the image to be unchanged. Using our algorithm we know that that would be $\frac{i'-1}{i'}$ for an arbitrary $i' > i$ house. So we need this to be ture for every $i' > i$. So we have,

$$\mathbb{P}(i) = \frac{1}{i}\prod_{k=1}^{n-i}\frac{(i+k)-1}{i+k}$$

We can expand this to get,

$$\mathbb{P}(i) = \frac{1}{i}\frac{i}{i+1}\frac{i+1}{i+2}\cdots\frac{n-1}{n}$$

We see that the denominiator of the $k'th$ term cancels out with the numerator of the $k+1$'th term. Hence we will only be left with the denominator of the last term which is,

$$\mathbb{P}(i) = \frac{1}{n}$$

So we show that for any i that the probability that the $i'th$ house is chosen is $1/n$

Q3 You're a chronic sports gambler and *DraftDuels* releases a promotion regarding the Atlanta Falcons where you think you might win some money. If, throughout the duration of the season, every player on the Falcons roster wins "Player of the Game" for at least one game, you will win a \$100 cash prize. Otherwise, you win nothing. Let's say there's n players on the football team, and each one of them has a $1/n$ chance of earning the "Player of the Game" title for any specific game. Prove that the expected number of games you would need in a season to win your \$100 bet is $\Theta(n\log(n))$.

Hint: Define G_k = number of games to get kth unique player to win "Player of the Game" after $k-1$ different players have already won "Player of the Game". Then, let $G = \sum_k G_k$, and use linearity of expectation.

We need to find the expected number of games in a season to win \$100 with the bet. Let G be the expected number of games where,

$$G = G_1 + \cdots + G_n$$

Where G_i is the number of games to get the $k'th$ unique player after $k-1$ players have already won.

The linearity of expectation tells us that,

$$\mathbb{E}(G) = \mathbb{E}(G_1 + \cdots + G_n) = \mathbb{E}(G_1) + \cdots + \mathbb{E}(G_n)$$

Now for any G_i we have $i-1$ non unique palyers and $n-i+1$ unique players so,

$$\mathbb{P}(G_i) = \frac{n-i+1}{n}$$

If $\mathbb{E}$ is the expected number of games for a given G_i then we have two cases, either with probability $\mathbb{P}(G_i)$ we get it in the first try else with probability $1-\mathbb{P}(G_i)$ we have $1+\mathbb{E}$ steps so,

$$\begin{aligned} E &= p + (1-p)(1+E) \\ E &= p + 1 + E - p - Ep \\ Ep &= 1 \\ E &= \frac{1}{p} \end{aligned}$$

So we have for a given i,

$$\mathbb{E}(G_i) = \frac{1}{\mathbb{P}(G_i)} = \frac{n}{n-i+1}$$

So,

$$\begin{aligned} \mathbb{E}(G) &= \sum_{i=1}^{n} \frac{n}{n-i+1} \\ &= \frac{n}{n} + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + n \\ &= n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + 1\right) \\ &= n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) \end{aligned}$$

We know that $\sum_{n=1}^{n} \frac{1}{n} = \Theta(\log n)$

So we have,

$$\mathbb{E}(G) = \Theta(n \log(n))$$

Q4 We are given n coins which each have their own probability of landing heads c_1,c_2, ..., c_n. Now, we flip all the coins and record how many of them landed on heads. We are interested in analyzing the probability that exactly k of the flips landed on heads. You will design an efficient *divide-and-conquer* algorithm that performs the following task:

Input: Coins with probabilities $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ of landing heads, and some integer $0 \leq k \leq n$
Output: The probability exactly k coins landed heads

(a) Lets first try a simple case by hand. If there are 3 coins, with probabilities c_1, c_2, and c_3 of landing on heads, what is the probability that when we flip all of them exactly 2 of them will be heads?

Note: Your answer should be written in terms of c_1, c_2, and c_3.

The ways we can choose 2 of the coins out of the three are $\binom{3}{2}$ which is 3 for each of them we compute the probability that those coins are on heads and the other coin lands on tails. So we get,

$$p = (c_1c_2(1 - c_3)) + (c_2c_3(1 - c_1)) + (c_1c_3(1 - c_2))$$

(b) For each coin c_i, what is a polynomial function $p_i(x)$ such that when you compute

$$p(x) = \prod_{i=1}^{n} p_i(x)$$

the coefficient of the x^kth term of $p(x)$ is the probability that exactly k flips landed on heads?

If we define $p_i(x) = c_ix + (1 - c_i)$ then the coefficient of x^k'th term will be the probability that exactly k flips landed on heads. This works because if we consider multiplying arbitrary polynomials defined by $p_i = a_ix + b_i$ then if we consider the product of $p_1, \ldots, p_n$ the x^k'th term will have the coefficient $(a_i \ldots a_{n-1}b_n) + (a_i \ldots a_kb_{k+1}b_n) + \ldots$ (we have a every of k choises in n) so if we replace a_i with c_i and b_i with $1 - c_i$ then each term in our sum with the k choices of $a_i = c_i$ would be the probability that we get heads on those k choices and the $n - k$ number of b_i would be the probability that the rest would be tails. So we have each term representing the probability of getting a specific choice of k coins as heads from n and we sum up these probabilities to compute the overall probability of getting k heads - this would be represented in our coefficient.

(c) Use this to describe an algorithm to solve the problem.

Our problem now comes down to finding a way to compute the coefficient of the x^k'th term of our polynomial, $p(x) = \prod_{i=1}^{n} p_i(x)$ To do this we can use a divide an conquer approach. Instead of multiplying out the n polynomial we'll divide our problem into two polynomials of half the size and multiply them together. The algorithm is as follows,

We have our algo $f((p_1, \ldots, p_n))$ that takes in a list polynomials $p_1, \ldots, p_n$ and computes the product. First our base case is when the list is of length 1 where we return the same polynomial. Now for any function call we have the following,

$$\begin{aligned} p_l &= f((p_1, \ldots, p_{\lfloor \frac{n}{2} \rfloor})) \\ p_r &= f((p_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, p_n)) \\ \text{prod} &= FFT(p_l, p_r) \end{aligned}$$

Here FFT is our polynomial multiplier which we use as a black box for now with runtime of $O(n \log n)$. prod would be the product of p_l and p_r which we get recursively. We then return prod.

(d) Provide a recurrence for your algorithm and use it to analyze the runtime.

Hint: To solve the recurrence, It's easier to divide both sides by n and to try analyzing $\frac{T(n)}{n}$.

Our recurrence would be,

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

Let us divide both sides by n to get,

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + O(\log n)$$

Let $\frac{T(n)}{n} = S(n)$ so we have,

$$S(n) = S(n/2) + O(\log n)$$

From here as we unroll our recursive we have the depth as $\log(n)$ so we'll have,

$$\begin{aligned} S(n) &= O(\log n) + O(\log n/2) + \ldots (\log n \text{ times}) \\ &= \log^2(n) \end{aligned}$$

So our total time complexity is,

$$\begin{aligned} T(n) &= nS(n) \\ &= O(n \log^2 n) \end{aligned}$$

Q5 The following is the Fast Fourier Transform algorithm as presented in the textbook.

```
function FFT($a, \omega$):
    Input: An array $a = (a_0, a_1, \ldots, a_{n-1})$, for $n$ a power of 2
    Input: A primitive $n$th root of unity, $\omega$
    Output: $M_n(\omega)a$
    if $\omega = 1$ then
        return $a$
    $(s_0, s_1, \ldots, s_{n/2-1}) \leftarrow \mathrm{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
    $(s'_0, s'_1, \ldots, s'_{n/2-1}) \leftarrow \mathrm{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
    for $j := 0$ to $n/2 - 1$ do
        $r_j \leftarrow s_j + \omega^j s'_j$
        $r_{j+n/2} \leftarrow s_j - \omega^j s'_j$
    return $(r_0, r_1, \ldots, r_{n-1})$
```

(a) Provide a modification of the algorithm where n is a power of 3, and explain why it works.

Hint: First figure out how to write a polynomial of degree d, where d is a power of 3, as a linear combination of polynomials that are a function of x^3. Next, find a way to group *triplets* of each root of unity instead of the *pairs* that the algorithm from the book follows. Finally, provide a divide and conquer algorithm where each subproblem is inputted with the $(n/3)$rd roots of unity.

Let p be a polynomial of degree d where d is a power of 3. Let it be,

$$p = a_0 + a_1 x + \cdots + a_d x^d$$

We can write this as a combination of three different polynomials defined as,

$$\begin{aligned} p_1 &= a_0 + a_3 x + a_6 x^2 + \cdots + a_{d-3} x^{d/3-1} + a_d x^{d/3} \\ p_2 &= a_1 + a_4 x + a_7 x^2 + \cdots + a_{d-2} x^{d/3-1} \\ p_3 &= a_2 + a_5 x + a_8 x^2 + \cdots + a_{d-1} x^{d/3-1} \end{aligned}$$

So we have,

$$p = p_1(x^3) + x p_2(x^3) + x^2 p_3(x^3)$$

Now let us take the n'th root of unity and we get,

$$\begin{aligned} z^n &= 1 \\ z &= e^{(2\pi i k\theta)/n} \qquad \text{where } 1 \leq k \leq n \end{aligned}$$

Now as n is a power of 3 any of our n root if we cube we get,

$$z^3 = e^{3(2\pi k\theta)/n} = e^{(2\pi k\theta)/(n/3)}$$

Or in other words the cube of our n'th root would be a $\frac{n}{3}$ root of unity. So for each $\frac{n}{3}$ root of unity we have 3 distinct roots for it ($k = 0, 1, 2$ would give us this) - for now let them be w_1, w_2, w_3 such that for any of those three $w_i^3 = w^3$. So we can do the following,

$$\begin{aligned} A(w_1) &= p_1(w^3) + w_1 p_2(w^3) + w_1^2 p_3(w^3) \\ A(w_2) &= p_1(w^3) + w_2 p_2(w^3) + w_2^2 p_3(w^3) \\ A(w_3) &= p_1(w^3) + w_3 p_2(w^3) + w_3^2 p_3(w^3) \end{aligned}$$

So now essentially by evaluating three smaller poylnomials p_1, p_2, p_3 at one point w^3 we are able to evaluate our larger polynomial p at three different places w_1, w_2, w_3.

Let us define our function as $\text{FFT}_3(a, \omega)$. Our input is the same as the question $a = (a_0, a_1, \ldots, a_{n-1}$ but for n a power of 3 and an n'th root of unity. First our base case is if $w = 1$ where we return a. Else we do the following,

$$\begin{aligned} (x_0, x_1, \ldots, x_{n/3-1}) &= \text{FFT}_3((a_0, a_3, \ldots, a_{n-3}), w^3) \\ (y_0, y_1, \ldots, y_{n/3-1}) &= \text{FFT}_3((a_1, a_4, \ldots, a_{n-2}), w^3) \\ (z_0, z_1, \ldots, z_{n/3-1}) &= \text{FFT}_3((a_2, a_5, \ldots, a_{n-1}), w^3) \end{aligned}$$

Now our x_i, y_i, z_i are the evaluations returned from the three sub-polynomials. We'll now put these sub problems back together. Firstly if w is an n'th root of unity then three roots of w^{3j} will be $we^{(2\pi ij)/3}, we^{(4\pi ij)/3}$ which we'll call as w_1, w_2, w_3 respectively.

Now we'll iterate from $j = 0$ to $n/3 - 1$ (essentially we're dividing our matrix in three parts both in terms of columns and rows) and we do the following,

$$\begin{aligned} r_j &= x_j + w_1 y_j + w_1^2 z_j \\ r_{j+n/3} &= x_j + w_2 y_j + w_2^2 z_j \\ r_{j+2n/3} &= x_j + w_3^2 y_j + w_3^2 z_j \end{aligned}$$

And then we return $(r_0, r_1, \ldots, r_{n-1})$

(b) Provide a recurrence for your algorithm and use it to analyze the runtime.

We divide our problem into three sub problem for three different polynomials each with $n/3$ terms. We iterate from 0 to n for a given input n for that function call which would be $O(n)$ so our recurrence would be,

$$T(n) = 3T(n/3) + O(n)$$

Using master theorem we have $n^d = n \implies d = 1$ and $a = 3, b = 3$ so our depth is $n^{\log_b(a)} = n^1$ because we have $d = log_b(a)$ we have our run time as,

$$O(n \log n)$$