CS 170 Homework 3

Due **2/12/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

Solution:

- OH and HW Party

2 Ice-Cream Loving PNPenguins

A group of PNPenguins are planning to run an ice-cream stand together for d consecutive days over the summer, which lasts $m > d$ days. Due to varying commitments, the group has agreed to staff a certain number of PNPenguins to sell ice-cream on each day of their operation period. Their goal is to find a starting date that can maximize their profit. To do this, they asked chatPenguin[1] to help them generate a list of predictions that contains the number of ice-creams each staffed PNPenguin can sell on that summer day. Their goal is to find a starting date that can maximize their profit.

Formally, we define the following:

- p_i represents the number of PNPenguins on duty to sell ice-cream on day i after their starting date ($1 \leq i \leq d$).
- a_j represents the amount of ice-cream that chatPenguin predicts each PNPenguin (on duty) to sell on day j ($1 \leq j \leq m$).

Design an $O(m \log m)$ algorithm to find the maximum number of ice-creams the group of PNPenguins can sell, as well as the starting date that allows them to sell that maximum amount. **Please provide a 3-part solution.**

[1] State-of-the-art LLM designed specifically for making ice-cream selling predictions.

Solution:
Example: $\mathbf{d = 3, \ m = 6}$

$$p = [2, 3, 5], a = [11, 12, 15, 8, 9, 7]$$

Start	Total Ice-Cream Sold
Day -1	$0 + 0 + 55 = 55$
Day 0	$0 + 33 + 60 = 93$
Day 1	$22 + 36 + 75 = 133$
Day 2	$24 + 45 + 40 = 109$
Day 3	$30 + 24 + 45 = 99$
Day 4	$16 + 27 + 35 = 78$
Day 5	$18 + 21 + 0 = 39$
Day 6	$14 + 0 + 0 = 14$

Flip $p \to [5, 3, 2]$ and represent both p and a as polynomials and multiply them.

$$p = [5, 3, 2], a = [11, 12, 15, 8, 9, 7]$$
$$P(x) = 5 + 3x + 2x^2,\ A(x) = 11 + 12x + 15x^2 + 8x^3 + 9x^4 + 7x^5$$

$$\begin{aligned} P(x) \cdot A(x) &= \left(5 + 3x + 2x^2\right)\left(11 + 12x + 15x^2 + 8x^3 + 9x^4 + 7x^5\right) \\ &= 55 + 93x + 133x^2 + 109x^3 + 99x^4 + 78x^5 + 39x^6 + 14x^7 \\ &\Rightarrow [\underset{0}{55}, \underset{1}{93}, \underset{2}{133}, \underset{3}{109}, \underset{4}{99}, \underset{5}{78}, \underset{6}{39}, \underset{7}{14}] \end{aligned}$$

We can see that the max coefficient is 133 at index 2, since we are counting the days we start before summer up day $2 - d$ then we can find the day by calculating $2 - d + index$ which in this case, $d = 3$ givig day $2 - 3 + 2 = 1$, which from the table above is the correct answer.

Algorithm:

Working under the assumption that we are given p and a as arrays we can reverse array p or ensure that we iterate over it with index 0 being length$(p) - 1$. We want to do this so that our result shows the total ice-cream sold consecutively from day k to $k + d$ where k is the day we start. Otherwise, our result would yield the total amount from day $k + d$ to k(no time machine). If we think of our arrays as representing polynomials P and A we can use FFT to find value representations and find $P \cdot A$. After that we use FFT^{-1} to find resulting polynomial $R = PA$. R holds the asnwer we are looking for, so all that is left is to find the max coefficient and start day, which is done by using formula $2 - d + index(max)$.

Proof of Correctness:

Since the proof of correctness for FFT and FFT^{-1} are given in the textbook and given in lecture, I will focus on showing why multiplying the reverse polynomial $P(x)$ by $A(x)$ gives us the max amount of ice cream possibly sold.

$$\text{Let } p = [b_0, b_1, \ldots, b_{d-1}] \text{ and } a = [a_0, a_1, \ldots, a_{m-1}]$$

$$\begin{aligned} P(x) &= b_{d-1} + b_{d-2}x + b_{d-3}x^2 + \cdots + b_0 x^{d-1} \\ A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1}x^{m-1} \end{aligned}$$

To find the coefficients of $(P \cdot A)(x) = R(x)$ we have...

$$R_k = \sum_{j=1}^{k} b_{d-1-j} a_{k-j}$$

$$R_0 = b_{d-1} a_0 \Rightarrow \text{ day } 1 - d$$

$$p = [b_0, b_1, \dots, b_{d-1}]$$

$$a = [\ a_0\ , a_1, \dots, a_{m-1}]$$

$$R_1 = b_{d-1} a_1 + b_{d-2} a_0 \Rightarrow \text{ day } 2 - d$$

$$p = [b_0, b_1, \dots, b_{d-2}, b_{d-1}]$$

$$a = [a_0, \quad a_1, \dots, a_{m-1}]$$

$$R_2 = b_{d-1} a_2 + b_{d-2} a_1 + b_{d-3} a_0 \Rightarrow \text{ day } 3 - d$$

$$p = [b_0, b_1, \dots, b_{d-3},\ b_{d-2},\ b_{d-1}]$$

$$a = [a_0, \quad a_1 x, \quad a_2, \dots, a_{m-1}]$$

$$\vdots$$

Above can see that p will keep shifting to the right until b_0 is multiplied with a_{m-1}, and because of the fact that we reversed p, the correct like terms will be grouped and give us the correct sum of total ice-cream sold across at most d days in m.

Runtime Analysis:

FFT$(P(X)) \Rightarrow O(d \log d)$

FFT$(A(x)) \Rightarrow O(m \log m)$

$(P \cdot A)(x) \Rightarrow O(m)$

FFT$^{-1}(R(x)) \Rightarrow O(m \log m)$

$\underline{\text{Finding } max \ \Rightarrow O(m)}$

Total: $O(m \log m)$

3 Counting k-inversions

A *k-inversion* in a bitstring x is when a 1 in the bitstring appears k indices before a 0; that is, when $x_i = 1$ and $x_{i+k} = 0$, for some i. For example, the string 010010 has two 1-inversions (starting at the second and fifth bits), one 2-inversion (starting at the second bit), and one 4-inversion (starting at the second bit).

Devise a $O(n \log n)$ algorithm which, given a bitstring x of length n, counts all the k-inversions, for each k from 1 to $n-1$. You can assume arithmetic on real numbers can be done in constant time. **Please give a 3-part solution; for the proof of correctness, only a brief justification is needed.**

Solution:

Algorithm:

To solve this problem, we begin by taking our given bitstring x and creating another bitstring, denoted as x', which is identical to x but reversed with its bits flipped. We then represent both x and x' as polynomials and compute their product $x \cdot x'$ using FFT and its inverse FFT^{-1}. The term x^{n-k-1} in the resulting polynomial corresponds to the number of inversions with value k. The coefficient of this term represents the count of k-inversions. By extracting the coefficient of x^{n-k-1}, we can efficiently compute the number of k-inversions for all values of k from 1 to $n-1$.

Proof of Correctness:

Given a bitstring x of length n, we construct another bitstring x' by reversing x and flipping its bits. Let $x[i]$ denote the bit at index i in the bitstring x. Each term in the resulting polynomial corresponds to a pair of indices (i, j) such that $x[i]$ and $x'[j]$ are both 1. For any term in the resulting polynomial, if the index difference between the corresponding indices i and j is k, then this indicates that there is a 1 at index i in x and a 0 at index j in x', giving us the k-inversion.

Runtime:

Since the algorithm relies heavy on on FFT, the runtime will be that of runnig FFT which is is reliant on the size of the string given, n. Giving a runtime of $O(n \log n)$

4 Distant Descendants

You are given a tree $T = (V, E)$ with a designated root node r and a positive integer K. For each vertex v, let $d[v]$ be the number of descendants of v that are a distance of at least K from v. Describe an $O(|V|)$ algorithm to output $d[v]$ for every v. **Please give a 3-part solution; for the proof of correctness, only a brief justification is needed.**

Hint 1: write an equation to compute $d[v]$ given the d-values of v's children (and potentially another value).

Hint 2: to implement what you derived in hint 1 for all vertices v, we recommend using a graph traversal algorithm from lecture and keeping track of a running list of ancestors.

Solution:

Algorithm Distant Descendants

```
procedure EXPLORE($v, d$)
    visited[$v$] $\leftarrow$ true
    $d[v] \leftarrow 0$
    for each neighbor $u$ of $v$ do
        if not visited[$u$] then
            children[$v$] $\leftarrow$ children[$v$] $+ 1$
            $d[u] \leftarrow$ explore($u$, distance $+ 1$)
            if distance $\geq K$ then
                $d[v] \leftarrow d[v] + d[u] + 1$
    return $d[v]$ + children[$v$]
```

Proof of Correctness:

Since this is a subroutine of DFS, the proof of correctness is similar and can be found in the textbook. On line 7, we recursively explore the subtree rooted at the child vertex u, updating $d[u]$ with the count of distant descendants for u.

Runtime Analysis:

Runtime of explore is $O(|V + |E|)$ and to output every $d[v]$ is equivalent to linear scan with respects to the $|V| = O(|V|)$, so we have $O(|V| + |E| + |V|) = O(2|V|) = O(|V|)$

5 Where's the Graph?

Each of the following problems can be solved with techniques taught in lecture. Construct a simple directed graph, write an algorithm for each problem by black-boxing algorithms taught in lecture, and analyze its runtime. You do not need to provide proofs of correctness.

(a) The CS 170 course staff is helping build a roadway system for PNPenguin's hometown in Antarctica. Since we have skill issues, we are only able to build the system using one-way roads between igloo homes. Before we begin construction, PNPenguin wants to evaluate the reliability of our design. To do this, they want to determine the number of *reliable* igloos; an igloo is reliable if you are able to leave the igloo along some road and then return to it along a path of other roads. However, PNPenguin isn't very good at algorithms, and they need your help.

Given our proposed roadway layout, design an efficient algorithm that determines the number of reliable igloos.

Solution: We could create a graph G, where every igloo is represented by a node and each directed edge would be a road from one igloo to the next. We could then find all the connected igloos by running DFS on G, finding the post numbers. Then we run DFS on the reverse graph G' starting at igloos with the highest post numbers found during the initial DFS on G. We can keep a counter or list containing all the connected iglos and return the counter or length of the list as the number of reliable igloos. This algorithm, Kosaraju's Algorithm, is guaranted to find all the strongly connnected components in $O(|E| + |V|)$

(b) There are x different species of Pokemon, all descended from the original Mew. Also, all species have at most 1 parent. For any species of Pokemon, Professor Juniper knows all of the species *directly* descended from it. She wants to write a program that answers queries about these Pokemon. The program would have two inputs: a and b, which represent two different species of Pokemon. Her program would then output one of three options in constant time (the time complexity cannot rely on x):

(1) a is descended from b.

(2) b is descended from a.

(3) a and b share a common ancestor, but neither are descended from each other.

Unfortunately, Professor Juniper's laptop is very old and its SSD drive only has enough space to store up to $O(x)$ pieces of data for the program. Give an algorithm that Professor Juniper's program could use to solve the problem above given the constraints.

Hint: Professor Juniper can run some algorithm on her data before all of her queries and store the outputs of the algorithm for her program; time taken for this precomputation is not considered in the query run time.

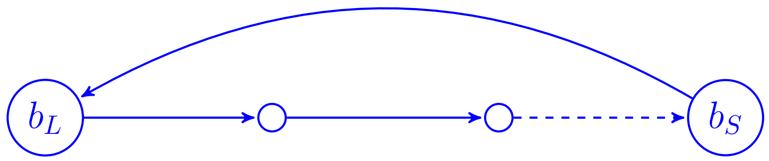
Solution: Before running the queries, we can represent the relationships between the Pokemon as directed tree structure T and run dfs on T keeping track of the pre-number and post-number (pre , post) for every vertex $v \in V$, saving the result in array, pre_post, where the i^{th} index represent node i and pre_post$[i]$ is a tuple containing the pre and post numbers. The size of pre_post is proportional to the number of species, x, making the space complexity $O(x)$. In order to answer any of the three queries all we need is access to a and b, which is constant time, we can index into pre_post[a or b] (if a and b are not integers we can represent them as integers and store their true value in a map). We can tell the sort of relationship between a and b by analyzing their post and pre numbers. **(1)** can be assesed by checking post and pre numbers. Specifically, if a is descendant of b then pre(b) $<$ pre(a) $<$ post(a) $<$ post(b). **(2)** is similar to (1), if b is a descendant of a, then pre(a) $<$ pre(b) $<$ post(b) $<$ post(a). For (1) $a \subset b$ and (2) $b \subset a$. **(3)** for this, since it we are representing the relationship as tree structure and because we are told that that all x are descendants of Mew, then we all we know to do is ensure that $a \not\subset b$ and $b \not\subset a$.

(c) Bob has n different boxes. He wants to send the famous "Blue Roses' Unicorn" figurine from his glass menagerie to his crush. To protect it, he will put it in a sequence of boxes. Each box has a weight w and size s; with advances in technology, some boxes have negative weight. A box a inside a box b cannot be more than 15% smaller than the size of box b; otherwise, the box will move, and the precious figurine will shatter. The figurine needs to be placed in the smallest box x of Bob's box collection.

Bob (and Bob's computer) can ask his digital home assistant Falexa to give him a list of all boxes less than 15% smaller than a given box c (i.e. all boxes that have size between 0.85 to 1 times that of c). Bob will need to pay postage for each unit of weight (for negative weights, the post office will pay Bob!). Find an algorithm that will find the lightest sequence of boxes that can fit in each other in linear time (with respect to the number of edges in the graph).

Hint: How can we create a graph knowing that no larger box can fit into a smaller box, and what property does this graph have?

Solution: For every box b, we ask Falexa for the list of boxes that are smaller than b. We create a graph with an edge from b to the each box. The resulting graph will be a directed acyclic graph since every box is different and a smaller box b_S cannot have an outgoing edge to a larger box b_L. For the sake of the argument lets say we have the following graph:


$$\begin{tikzcd} b_L \arrow[r] & \circ \arrow[r] & \circ \arrow[r, dashed] & b_S \arrow[lll, bend right] \end{tikzcd}$$

Since Falexa only gives boxes that are less than 15% smaller than b_L, this is contraction as b_S has an outgoing edge to b_L (creating a cycle) but b_L is larger, showing that we cannot have cycles and therefore the resulting graph must be a DAG. We can store the information about the weight of the boxes (from intial scan) and run DFS to find the lightest path. The initial scan of the $n = |V|$ boxes is $O(|V|)$ and the cost to run DFS is $O(|V| + |E|)$, $|E|$ being the number of egdes. Giving the total runtime of $O(2|V| + |E|) = O(|V| + |E|)$.