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".

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.**

Solution:
Algorithm: Construct a polynomial $A(x) = a_1 + a_2x + a_3x^2 + \cdots + a_{m-1}x^{m-1}$ and $P(x) = p_d + p_{d-1}x + p_{d-2}x^2 + \cdots + p_1x^{d-1}$. Use FFT to compute the product $A(x) \cdot P(x)$. Do a linear scan on the coefficients of the resulting polynomial (starting at term x^{d-1} until term x^{m-d+1}) to find the maximum and the corresponding day (which will be $exponent - d + 2$).

Proof of Correctness: Notice that to get the total number of ice-creams that can be sold by PNPenguins over the duration of d consecutive days, we want to multiply the amount that can be sold by one PNPenguin multiplied with the number of PNPenguins that will be selling on that day. After multiplication, the coefficient of the term x^n will be : $\sum_{i=1}^{d-1} a_{n-(d-1)+i}p_i$, which is exactly the total number of ice-creams PNPenguins can sell if they start on day $n - (d-1) + 1 = n - d + 2$ (an extra +1 for the 1-indexing).

Runtime Analysis: Constructing the polynomials take linear time $O(m + d)$. Running FFT to do polynomial multiplication takes $O(m \log m)$ time. The final linear scan takes

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

linear time $O(m)$. Therefore, the overall runtime will be $O(m \log m)$ since it dominates over other linear times.

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: Time for FFT!

Algorithm description

Construct two vectors: vector $\vec{p} = [0, \ldots, 0, x_0, \ldots, x_{n-1}]$ is the "padded" vector which has length $2n$ and consists of every bit in b individually, preceded by n zeros; vector $\vec{f} = [(1 - x_0), \ldots, (1 - x_{n-1})]$ is the "flipped" vector which consists of every bit in b, flipped (replace 0 with 1 and 1 with 0).

Now, we want to compute the cross-correlation of these two vectors. The dot product between f and $p[j : n + j]$ is the number of $(n - j)$-inversions, which we can report for j from 1 to $n - 1$. (Note that the first and last dot products will be ignored, since there are no 0- or n-inversions.)

To compute this cross correlation, we perform the following steps:

1. Reverse $\vec{f}$ to yield $\vec{f_R}$.
2. Generate the polynomials P and F by treating $\vec{p}$ and $\vec{f_R}$ as coefficient arrays, respectively.
3. Compute the "cross-correlation polynomial": $C = P \cdot Q$.
4. The coefficients of C is the cross correlation of $\vec{p}$ and $\vec{f}$.

Proof of correctness

There is a k-inversion starting at index i if and only if $x_i = 1$ and $x_{i+k} = 0$; which in turn is true if and only if $x_i(1 - x_{i+k}) = 1$. Consider our dot product $f \cdot p[j : n + j] = \sum_{i=0}^{n-1} f_i p_{i+j}$; but since $p_{i+j} = 0$ whenever $i + j < n$, this is $\sum_{i=n-j}^{n-1} f_i p_{i+j} = \sum_{i=0}^{j-1} f_{i+n-j} p_{n+i}$ which, by the definition of f and p, is $\sum_{i=0}^{j-1}(1 - x_{i+n-j})x_i$. But by the first sentence, each term of this sum is 1 if there is a $(n - j)$-inversion starting at i, and 0 otherwise, so this sum is the number of $(n - j)$-inversions.

Since we know from lecture that the cross-correlation algorithm computes these products correctly, we can conclude that our whole algorithm is correct.

Runtime

Constructing our two vectors p and f takes a total of $\Theta(n)$ time, and we know from lecture that the cross-correlation algorithm takes $\Theta(n \log n)$ time. Arranging the results correctly

may take $\Theta(n)$ time, or constant time, depending on implementation. So, since we do no other work, our algorithm as a whole takes $\Theta(n \log n)$ time.

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:
Observe that for a vertex v,

$$d[v] = \sum_{c \text{ is a child of } v} d[c] + (\#\text{ descendants of } v \text{ at distance } K)$$

The algorithm proceeds as follows. Initialize all the $d[v]$ values to 0 and start a dfs at the root node. At every step of the algorithm, we will maintain the ancestors of the current node in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it. Since a path can have at most n vertices, the length of this array is at most n.

Now, while processing the node v, we first index into the array equal to the index of its kth ancestor, call this ancestor a. Then we increment $d[a]$ to account for v. Once we finish processing a child c of v, we increment $d[v]$ by $d[c]$.

We provide pseudocode for this algorithm below:

```
def find_distant_descendants(G=(V, E), r, K):
    d = [0 for v in V]
    visited = [False for v in V]
    ancestors = []

    def explore(v):
        visited[v] = True

        # if possible, increment the d-value of the Kth ancestor by 1
        if len(ancestors) >= K:
            d[ancestors[-K]] += 1

        # add v to the list of ancestors and recurse on children
        ancestors.append(v)
        for each edge(v, u) in E:
            if not visited[u]:
                explore(u)
        ancestors.poplast()
```

```
        # if possible, increment the d-value of the parent
        if len(ancestors) >= 1:
            d[ancestors[-1]] += d[v]

    explore(r)
    return d
```

Runtime Analysis: Since we perform a constant number of extra operations at each step of DFS, the algorithm is still $O(|V|)$.

Correctness For a vertex v and a child c, every node counted in $d[c]$ should be included in $d[v]$ because their distance to v can only increase. Furthermore, nodes that are exactly K away from v will not be counted for any of its children, since they will be closer than K to the corresponding child. So, these get accounted for whenever we visit a kth descendant of v and increment $d[v]$. Notice that when we finish processing a node v, its $d[v]$ value will be correct and so it can be used by its parent.

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:

Main Idea: Construct a directed graph $G = (V, E)$ with vertices representing the igloos and edges representing the one-way roads. Then, a reliable igloo is equivalent to a vertex that belongs to a cycle. Thus, our job is to determine the number of vertices in G that belong to some cycle.

First, we run Kosaraju's algorithm on G to compute all of its strongly connected components and slightly modify the algorithm so that it also computes the number of vertices in each SCC. Then, the desired set of vertices can be obtained by including all vertices $v \in V$ that lie in an SCC with size at least 2.

Proof of Correctness: Any SCC $S \subset G$ is a subgraph in which all its vertices $v \in S$ can reach one another. Thus, the vertices of any SCC with size at least two by definition have a cycle because any vertex v can reach any other vertex $u \in v \in S$, which in turn can return to v.

Runtime Analysis: Running Kosaraju's takes $O(n+m)$ time, and traversing through all of the vertices to determine the desired set of vertices takes $O(n)$ time. This is assuming that during Kosaraju's, we construct a hashmap with vertices as keys and SCC size as values. Thus, the overall runtime is

$$O(n+m) + O(n) \in O(n+m)$$

(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: This is a directed graph, with each node representing some species and an edge from a to b indicating that b descended *directly* from a. You can think of the graph as a directed tree rooted at Mew.

We run DFS on this graph, storing the post-numbers and pre-numbers for each node. When the program is queried, it checks whether a is descended from b by checking if $pre[b] < pre[a] < post[a]$, or b is descended from a ($pre[a] < pre[b] < post[a]$), otherwise concludes a and b share a common ancestor but are not descended from each other.

The DFS takes $O(x)$ time, but it is part of the precomputation. The query involves indexing into two arrays and comparing them, all of which takes $O(1)$ time.

(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: We can view each box as a node in a directed graph, with an edge from a to b indicating that a fits in b. Since Falexa only gives the edges in the opposite direction (in that it tells Bob the boxes that can fit into b), we have to construct a graph and then reverse the edges, which is doable in linear time. This graph is a DAG. If box a can fit in box b and box c can fit in box a, box b cannot fit into box c. We set incoming edge weights to a node a to be the weight w of the corresponding box. We can linearize this DAG by running DFS starting at the smallest box x, and then find the shortest path from the x to any other box in linear time by iterating through the DAG's vertices in

topological order to compute each new shortest path by building on paths to previous nodes in the DAG.

6 [Coding] DFS & SCCs

For this week's homework, you'll implement implement DFS and the SCC-finding algorithm covered in lecture. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-sp24-coding`

   and navigate to the `hw03` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw03` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 3 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public edstem threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.

- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.