# Homework 5

CS41, Spring 2023 — Alina Palacios and Nina Zhuo

1. We build a graph $G = \{V, E\}$, where the vertices represent a planet-year pair representing all possible planets the Algoids could live on out from year 0 to $Y$. Each edge is weighted by the population growth in remainder of year denoted by $\alpha_{j,y}^{1-t_{p_j,p_i}}$. The goal is to maximize the total growth of the population. Since we want to account for the travel time to the planet the Algoids are traveling to, or in other words run the DAG longest path, we must take the *log* of the edge weights to convert the multiplication into addition. There are two options that the Algoids can choose to maximize their population growth. This can represented by two equations. Let the function $F(P_i, y)$ represent the max growth at year $y$ ending on Planet $P_i$.

   i. If it is optimal for the Algoids to STAY, then the equation is $F(P_i, y) = log(\alpha_{piy} + F(p_i, y-1)$. We take the log of the alpha function and account for the previous year that the Algoids stayed on the first planet.

   ii. If it is optimal for the Algoids to LEAVE, then the equation would be $F(P_i, y) = log(max\{\alpha_{p_i,y}\ ,\ \alpha_{j,y}^{1-t_{p_j,p_i}} + F(P_i, y-1)\})$.

   The first vertex of the graph represents the planet the Algoids are currently living on, $P_0$, paired with the current year 0. The edges will branch out to other vertices which are the planets in the system, $P_1, P_2, ...P_n$, paired by the number of years out from year 0. There would be an edge from each planet from planet-year pair $P_0$ to $P_n$ at year $Y$ that goes out to a dummy end vertex, since it is unclear what planet would be optimal for the Algoids to end up on at year $Y$ to maximize population growth. We will traverse through the graph and we will find the max between the choice of STAY and LEAVE using our $F(p_i, y)$ at each year $y$. Once we reach our dummy end vertex, we find the Algoid's maximum possible population after $Y$ years by calculating $2^{F(p_i,y)}$. This algorithm will take $O(n^2Y)$ time because we are using a nested `for` loop to iterate through the possible $P_n$ planets, and we multiply by $Y$ years since we are incrementing the number of years as we go further into each level of the graph.

2. We build a graph $G = \{V, E\}$ where the vertices represent each successive second from $t = 0$ to $n$ such that $|V| = n$. Each vertex has an outgoing edge to each successive vertex from $j = i$ to $n$. Edges are weighted on the number of Sentinels disarmed when a pulse is released from the EMP at time $j$ after last being released at time $i$. In other words, the EMP has had $i - j$ time to recharge. Thus, each edge weight can be calculated using $min\{f(j-i), G[j]\}$. This process will take $O(n^2)$ time as we are using a nested `for` loop.

   Because we don't know which time $t$ is optimal for the release of the first pulse, we set up a dummy start vertex with edges out to every $t = 0$ to $n$. Similarly, because we don't know which time $t$ is optimal for the release of the final pulse, each vertex from $t = 0$ to $n$ has an edge that goes out to a dummy end vertex.

   We traverse the graph to find the DAG longest path from 0 to $n$ such that the number of Sentinels disarmed is maximized. Let $D(i, j)$ be the longest path up to vertex $i$ such that $D(i, j)$ will always be the maximum number of Sentinels disarmed up to $t = i$.

$$D(i,j) = max\{D(i,j), D(i,j) + min\{f(j-i), G[j])\}\}$$

We update the maximum number of Sentinels disarmed up to time $t = i$ using a nested `for` loop, iterating over each $i = 0$ to $n$ and each $j = i$ to $n$ such that $D(i,j)$ is updated $n^2$ times. We return $D(i,j)$ as the maximum total number of Sentinels that can be disarmed by time $t = n$.

3. We build a graph $G = \{V, E\}$ where each vertex represents all valid substrings within the input string $s$. In other words, $|V|$ = number of substrings in $s$ that return TRUE for DICT($w$). We make all valid strings in $n^2$ time where $n$ is number of characters in $s$. We use a double `for` loop where $i$ is the start index and $j$ is the end index such that all vertices are substrings where $s[i,j]$ is a word. We draw an edge between each word $(i,j)$ and $(j+1,k)$ where $k$ represents an end index $j+1 < k < n$. This process will take $O(n^2)$ time as we are using a nested `for` loop.

   Because we don't know which word starting with the letter $s[0]$ is the optimal choice, we create a dummy start vertex with edges going out to all words that start with $s[0]$. Similarly, because we don't know which word ending with the letter $s[n]$ is the optimal choice, we draw an edge from every word that ends with $s[n]$ out to a dummy end vertex.

   If there exists some path from the dummy start vertex to the dummy end, we return TRUE. Otherwise, we return FALSE.

4. We build a graph $G = \{V, E\}$, where the vertices are pairs that represent all possible combinations of the of $i$, the indices of the array $x$, and $j$, the indices of the array $y$. We draw an edge from each $(i,j)$ pair to every subsequent pair of possibly equivalent $(i,j)$ pairs to check if the $i$-th index of $x$ matches any subsequent $j$ index of $y$ and/or if the $j$-th index of $y$ matches any subsequent $i$-th index of $x$. We will have a dummy start vertex pointing to all the $(i,j)$ pairs of the array, since the subsequence can start anywhere. We will also have a dummy end vertex coming from each $(i,j)$ pair, since we don't know where in the array the subsequence will end. Our algorithm operates as follows: we will have a double `for` loop that will compare the indices of $x[i]$ and $y[j]$. Let $K(i,j)$ be the length of the longest subsequence up to our $(i,j)$ pair. There are two cases the algorithm can run into:

   i. If $x[i] \neq y[j]$, $K(i,j)$ would return 0 and not be incremented.
   ii. Else, we would look at the previous element to determine if there is a valid subsequence shared between the two array, denoted as $K(i-1, j-1) + 1$.

   We return $K(i,j)$ as the largest $k$ for which there are indices $i$, $j$ with $x[i...i+k-1 = y[j...j+k-1]$. This algorithm will take $O(mn)$ time because we are using a nested `for` loop to draw an edge from each possible vertex, in this case from each $(i,j)$ pair to every subsequent pair of possibly equivalent $(i,j)$ pairs.