- The assignment is due at Gradescope on Friday, Feb. 24 at 6:00p.
- You can either type your homework using LaTex or scan your handwritten work. We will provide a LaTex template for each homework. If you writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- You are permitted to study with up to 2 other students in the class (any section) and discuss the problems; however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites. *Consulting problem solutions on the web is not allowed.*
- *Show your work.* Answers without justification will be given little credit.

PROBLEM 1 (25 POINTS) *Do K-T Chapter 6, Exercise 24 (on gerrymandering).*

Solution:
Input: A list of precincts, $p_1 \cdots p_n$ with a number of A party votes and B party votes for each precinct.
Output: Determine if it is possible to partition the precincts in two districts of the same size such that one party wins both districts.

Family of problems:
In order to solve the question, we will build out the recurrence to consider all possible voter counts we can return for a given amount of precincts, based on the minimum amount. Each possible count will be labeled with 1 and each impossible count will be labeled with a 0. Gradually we will build towards the maximum amount, described in the iteration section. Note that we will also **process linearly** through $p_1 \cdots p_n$ to build out the family of problems, meaning that our possible solution will be with respect our position on the array of precincts. Also we will store complete problems in a three dimensional matrix, to not repeat unneeded work. Finally, once we have a complete matrix we will iterate through the possible answers which could meet our requirements and return true if we have an answer there.

Let v be the number of total party A voters in our the district that we are building, say district 1. Let i be the number of current precincts in our set. Let j be the overall amount of precincts we are selecting from, with relation to $p_1 \cdots p_n$. Note that we plan to iterate through the precincts to find our optimal answer, so we have the given equation (where n is the last precinct in $p_1 \cdots p_n$): $i \leq j \leq n$.

Base case
First we consider base cases to build our recurrence. Note that we are using 1 to represent a susceptible Gerrymandering district and 0 to represent a non susceptible case. Let v_i be the voters in a given district i, so v_1 is the voters in p_1.

$$BASE(v, i, j) = \left\{ \begin{array}{ll} 1 & \text{if } v = 0 \text{ and } i = 0 \\ 1 & \text{if } v = v_1 \text{ and } i = 1 \text{ and } j = 1 \end{array} \right\} \tag{1}$$

We can justify the base cases as follows: The first is that $v = 0$ and $i = 0$. This is trivially true, if any district is susceptible, that district could be formed from the set if we start with 0 districts and 0 votes. Similarly, the second base case is $v = v_1$ and $i = 1$ and $j = 1$. Here we consider if any given district is the only district in the set, and the overall range of that set is 1. Again this is trivially true, because we cannot form an amount of voters different to the amount in first district we pick from. Note that these base cases are not "susceptible" with respect to the entire list, but instead we build a "susceptible" solution from "susceptible" base cases.

Building Recurrence
Next we move to the recurrence below (note that this is assuming base case and $1 < i \leq j$):

$$RECUR(v,i,j) = \left\{ \begin{array}{ll} 1 & \text{if } RECUR(v,i,j-1) = 1 \\ 1 & \text{if } RECUR((v-A_j),i-1,j-1) = 1 \end{array} \right\} \quad (2)$$

Essentially, in this recursion we backtrack one precinct from a valid "susceptible" case and consider the two potential cases: where we place the precinct in the district we are building or we don't place it in the district. First, $RECUR(v,i,j-1) = 1$. In this case we don't place the precinct in the district, so we don't have any difference between the current case (in voter amount or precincts amount in district), other than we are 1 index less in our processing. So we have $j-1$. Next, $RECUR((v-A_j),i-1,j-1) = 1$. Here, we have the count of previous district, but we subtract an index, so $j-1$, and we subtract from the overall count of districts, so we have $i-1$. Finally, we subtract A_j (party A votes in precinct j) from v, because in step j we will add all votes from party A. Therefore, we get $j-1$ from j for both recurrences.

Combining these relations, we have the recurrence.

Formal Recurrence Relationship

$$OPT(v,i,j) = \left\{ \begin{array}{ll} \text{return 1} & \text{if } OPT(0,0,j) \\ \text{return 1} & \text{if } OPT(v_1,1,1) \\ \text{return 1} & \text{if } OPT(v,i,j-1) = 1 \\ \text{return 1} & \text{if } OPT((v-A_j),i-1,j-1) = 1 \\ \text{return 0} & \text{else} \end{array} \right\} \quad (3)$$

Matrix Dimensions
Now we must consider what intervals to run our algorithm over when we iterate. We know our minimums from the base case, so we consider upper "legal" bounds for the solution. To solve the entire set, we consider upper bounds $j = n$, $i = \frac{n}{2}$, which are intuitive: we can have exactly half of the precincts to be in the district we build, and we want to consider n precincts for j. However to find v, we must determine the bound on voters where we can win both precincts. Therefore, we will run a loop on the set of $p_1 \cdots p_n$ and sum the overall amount of votes (A and B party) over all precincts as integer V, and A party votes as V_p. This is computed in $O(N)$ time. Therefore, we need to have at least $\frac{V}{4} + 1$ votes returned by v to be susceptible, however, our bound for iterations will be the minimum votes not in v to win the other district, which is $V_p - (\frac{V}{4} + 1)$. Note if we have more than $V_p - (\frac{V}{4} + 1)$ votes in v, we would not have more than a quarter of overall votes in the other, meaning it is impossible to win in. Therefore, we will perform this linear computation to find V first and have $v = V_p - (\frac{V}{4} + 1)$ as an upper bound for v. This gives us the dimensions of our matrix as follows:

Start (lower bound): $M[0][0][0]$
End (upper bound): $M[V_p - (\frac{V}{4} + 1)][\frac{n}{2}][n]$

First we run a linear loop and initialize all starting values from $k = 0 \cdots n$, $[0, 0, k]$, in the bottom of the matrix and gradually build on previous cases, through iterative calls to $OPT(v, i, j)$. Using the three bounds describe above, we fill out our matrix with 3 nested loops.

Proof of Correctness: To prove correctness, we will show that $LHS = RHS =$ all recurrence cases in RHS, or $LHS =$ all recurrence cases in RHS. First we will show that $RHS =$ the two base cases. These are proved in the **Base Case** section, and can be referred to there, which show that for $LHS =$ if $OPT(0, 0, j)$ and $LHS =$ if $OPT(v_1, 1, 1)$. Therefore for these cases we have proved $LHS = RHS$.
Now we prove $LHS = RHS$ for the two recurrence cases. As described in the **Building Recurrence** section, each of these recurrence cases considers if the previous precinct was added or not added to the set we are building, and if that amount of voters is possible for the current group of precincts, built from adding voters to the base case of exactly the amount of votes in a single precinct in the district. Therefore, both recurrence cases show the two options for j from $j - 1$, which proves $LHS = RHS$ for both recurrences, meaning $LHS =$ all recurrence cases in RHS, proving correctness.
Constraints for Solution As we have shown, we begin at the lowest base cases possible and increment the loops outwards until we have reached the upper bound of a possible solution. Therefore, once we have a full matrix, we can determine if there is any possible way to Gerrymander based on the size of the district constraint in the problem. To do this, we query through every instances of the matrix, where $j = n$, $i = \frac{n}{2}$ (these are required values in our solution by question constraints). Note that we have the lower bound as $\frac{V}{4} + 1$ and the upper bound as $v = V_p - (\frac{V}{4} + 1)$, in terms of the limits of possible vote range to have a majority in both. Therefore, we will check every entry in between this range of bounds, and if any entry equals 1, then we know a susceptible Gerrymandering is possible and if no entry equals 1 we know that it is impossible, because we have considered every variation of problems in the matrix and proved the recurrence relation.

Run time
We will use a three dimensional matrix to store sub problem solutions, so we only need to execute each problem once, and afterwards we can reference the answer in constant time. Note that a given sub-problem can be solved based on constant operations, so it is $O(1)$ for a given problem. However, we have a total number of $n \cdot \frac{n}{2} \cdot (V_p - (\frac{V}{4} + 1))$ sub problems. Note that in worst case run time $V_p - (\frac{V}{4} + 1) = O(nm)$, as we calculate it from the overall amount of voters in each precinct, so it is some constant operations with respect to

nm. Therefore, this simplifies to $O(n^3m)$ in terms of constructing the matrix. In our traversal where we check for a correct solution, we have worst case $V_p - (\frac{V}{4} + 1)$ iterations, giving us $O(n^3m) + O(nm)$ considering lookup time, which simplifies to $O(n^3m)$

PROBLEM 2 (25 POINTS) *Do K-T Chapter 6, Exercise 14 (on mobile networks). Note: this problem can be solved by applying the techniques of K-T Section 6.3 on multi-way choices.*

Solution:
A:
Input: A list of graphs, $G_0 \cdots G_b$.
Output: Determine if it is possible to have a single $s - t$ path along all graphs in $G_0 \cdots G_b$. If so return the shortest path possible. If not return no path possible.

Algorithm
$G_F = G_0$
for i from 1 to b
    remove every edge from G_F that is not in G_i

$P =$ find shortest $s - t$ path on G_F **if a path exists**
else
    return no path possible
return P

Correctness: We start with the first graph, G_0 as G_F and with every iteration we remove all edges that are not in the i graph from G_F. Therefore, by the end of iterating through the list, G_F will contain edges in every $G_0 \cdots G_b$, meaning any path on G_F by the end of the for loop could be taken on each of these paths. Next, we will use Dijkstra to find the shortest path on G_F, which we know is the shortest shared by all $G_0 \cdots G_b$. This is because any shorter path on all of the set of graphs must contain some edge not in a graph in the sequence (meaning the path would be invalid), or else all the edges would be on G_F, and then the path would be returned as P. Therefore if we can find a path over every $G_0 \cdots G_b$, then we know P will return the shortest common path.

Runtime: We have a loop of b iterations, and in each loop we potentially compare every edge, E between the two graphs. Note that we will define E to be the maximum of all $E_1, \cdots E_b$ edges, because in the worst case run time we compare this amount of edges every iteration. Comparing two graphs can be done in $O(E)$ time for each iteration in the loop simply by checking each edge in G_F to see if it occurs in G_i and removes those which don't. Therefore, the loop will be $O(b \cdot E)$ run time. Finding the shortest path through Dijkstra will take $O(E \log V)$. Therefore, we have $O(b \cdot E) + O(E \log V)$, which is a polynomial run time.

B:

Input: A list of graphs, $G_0 \cdots G_b$, and a given "switch cost" constant K.
Output: Return a list of paths $p_0 \cdots p_b$ for $G_0 \cdots G_b$ such that the overall cost of the path sequence is minimized.

Sufficiency of this family of problems: Before applying our recurrence, we will repeatedly use the $A[G_i, G_j]$ function to create a two dimensional matrix of the shortest path between every interval on the set of graphs, or all i, j of $0 \leq i \leq j \leq b$. From here, we will use a minimization function across these values and the switching cost to minimize cost of paths over $0 \cdots b$, tying our sub problems to the solution.

Base Case: For $b = 0$, we have $OPT(0)$ and know that the optimal solution will be the shortest $s - t$ path for G_0. $A[G_0, G_0]$ will be the shortest path on only G_0. Therefore we compute this and return it, which is optimal.

Next, assume that the graph switches paths at least once. Then, we make the following observation: the optimal cost path p_b for G_b either is the shortest path on G_b or it contains the shortest path on an earlier graph G_i. This is because if it is optimal to switch paths at b, as G_b is the final graph we are considering, we will switch to the shortest path with respect to only G_b, as this is the only condition we need to optimize for. If not, we know that the optimal path p_b must be the optimal path along $p_i \cdots p_b$, where p_i is the last time that the optimal cost algorithm switches paths. By assumption this occurs, because the graph switches paths at least once.

Finding the shortest path for a given graph is straightforward. Therefore, we will focus on finding the shortest path between a range of indexes, with respect to the ordering within $G_0 \cdots G_b$. Note that this is exactly the problem solved in A, so we will apply A with every set of indexes within $G_0 \cdots G_b$ to solve our sub problems. This gives the following solution for minimizing the cost at b, where i is the last index where the switch occurs at (we can have $i = b$, which addresses the last switch at b), and $A[G_i, G_b]$ is the minimum cost path equation solved by A:

$$OPT(b) = OPT(i-1) + (b - (i-1)) \cdot A[G_i, G_b] + K$$

As proved previously, we know $A[G_i, G_b]$ is the shortest path over the interval of two graphs, given one exists. While we know there must be a minimum path for i to b, as assumed that it is the last change in the optimal solution, however we do not know this applies for every two indexes. Therefore in our implementation we will require a switch whenever a path is is impossible to be found. As we will use a minimizing function and consider the size of the path given by $A[G_i, G_b]$, we can return ∞ whenever no path is possible, so the minimize function will never select a path impossible for a graph. We extend the minimize function over the interval as so:

$$OPT(b) = \min_{1 \le i \le b} OPT(i-1) + (b - (i-1)) \cdot A[G_i, G_b] + K$$

We will run this recurrence linearly through $G_0 \cdots G_b$, which will eventually given the optimal solution. However, we have assumed that this function does change paths at least once. Therefore we must consider the case where the function never changes paths. This edge case can be incorporated by checking if $A[G_0, G_b] \cdot b + 1$ (the value of the cost function if path never changes) is smaller than the algorithm we have built based on the assumption the optimal solution switches path at least once. We will also include a base case to build our recursion on, which is the shortest path on G_0. This completes our recurrence for the minimum cost of graphs for a given n processing $G_0 \cdots G_b$ linearly:

Claim: Recurrence

$$OPT(n) = \left\{ \begin{array}{ll} A[G_0, G_0], & \text{if } n = 0 \\ \min(n+1 \cdot A[G_0, G_n], (\min_{1 \le i \le n} OPT(i-1) + (n - (i-1)) \cdot A[G_i, G_n] + K)), & \text{if } n \neq 0 \end{array} \right\} \tag{4}$$

Proof of Correctness:
We want $LHS = RHS =$ if $n = 0$, if $n < b$, if $n = b$ for $OPT(n)$. As we want the solution between $0 \cdots b$, we know this will cover all cases, so we have $LHS =$ if $n = 0$, if $n \neq 0$ for $OPT(n)$. We will assume that all for all $0 \le i < n$, $OPT(n)$ is true, and prove by induction.
$n = 0$ is straight-forward, and our base case. We run Dijkstra's on a single graph, G_0 to find the shortest path with respect to the n. This is our base case, and as there is no cost of switching for the initial graph, the shortest path on G_0 is always the optimal path. This proves $LHS =$ if $n = 0$.
For $n \neq 0$, we consider the minimum of the last time it was optimal to switch and if the path never switched. We consider these cases below:
Path never changed: Then we have $n + 1 \cdot A[G_0, G_n]$, or the minimum path possible along all graphs $G_0 \cdots G_n$ given by answer A multiplied by $n + 1$. If it is optimal to never switch, then we know that this is the optimal solution.
Path changes at least once: Then let i be the last time that the path changed before n. Here we add the $OPT(i-1)$ solution, which we know to be true, because at it's most $i - 1 < n$, meaning we assume it is true by inductive assumption. Therefore, from the path changing to n, we have $(n - (i-1)) \cdot A[G_i, G_n] + K$, or the the sum of the optimal path along $G_i \cdots G_n$, multiplied by the amount of times this path will be in the $OPT(n)$ solution, and the switching cost k. As we know the path switches at i the last time, this is an optimal solution. Furthermore, we can identify each i through the minimizing function, because we will consider all sub-intervals with $1 \le i \le n$ and take the minimum overall. Therefore, we have $LHS =$ if $n \neq 0$ if the path

changes once or if it never changes, meaning we have $LHS =$ if $n \neq 0$ and subsequently $LHS = RHS$, which proves the correctness of our algorithm.

Format of Answer:
This recurrence minimizes overall cost, but it does not find the sequence of paths themselves. To do this, we could easily add another conditional statement to check when changes in the path occur and add them from $A[G_i, G_n]$ when needed, which would return a sequence of paths rather than the minimum cost amount.

Run time analysis:
In order to decrease run time, we will run $A[G_i, G_n]$ on every pair $0 \leq i \leq n \leq b$. Therefore, we have b^2 pairs, and $O(b \cdot E) + O(E\log V)$ for each $A[G_i, G_n]$ as proved previously. So, we have $O(b^3 \cdot E) + O(b^2 \cdot E\log V)$ run time to generate all our sub problems. As we generate sub problems through recursive calls within each sub problem, $A[G_i, G_n]$, once we have generated all sub problems we will have $A[G_0, G_b]$, which is our desired solution, which we can access in constant time. Therefore $O(b^3 \cdot E) + O(b^2 \cdot E\log V)$ is overall run-time.

PROBLEM 3 (25 POINTS) *(Pokémon battle; K-T Sec. 6.3 techniques applicable here too) You are an up-and-coming Pokémon trainer supervising a training battle. Your n Pokémon $p_1, \ldots, p_n$ are arranged in a line, in that order. At each time step, a single pair of adjacent Pokémon fight. The winner stays in battle at the same position on the line; the loser "faints" and is removed. The battle ends when there is only a single Pokémon left. Example: in the first fight, perhaps p_4, p_5 fight and p_5 wins. Now p_4 is removed and p_5 can fight next with p_3 or p_6. (Or the next fight may not involve p_5 at all.) As the trainer, you know for each pair p_i, p_j $(i < j)$ which of the two Pokémon would win in a fight. This is recorded in the (i, j)-entry of a matrix M which you can access. Still, there are many possible ways the battle could unfold, depending on which adjacent pair of Pokémon fights next at each step. Please give an algorithm running in time polynomial in n, that outputs a list of all possible winners of the tournament. For full credit, prove its correctness and polynomial runtime.*

Solution:
Input: A list of Pokemon $p_0 \cdots p_n$, along with a matrix which gives a winner for any fight between two Pokemon.
Output: A list of all possible winners of the tournament.

Sufficiency of this family of problems: We will consider every interval from every possible pair in $0 \cdots n$ and gradually merge sets from there, and we will build sets from smaller sub problems. If the correctness of merging and the recurrence is proved, $OPT[0, n]$ will return the set of all possible winners for $p_0 \cdots p_n$, as it covers the entire interval.

Helper "Merge" Function:

This function will merge two lists of "winners" from two "sub-tournaments". A winner is any Pokemon that can win a "sub-tournament". A "sub-tournament" is a partition of the original set $p_1 \cdots p_n$, with the same order but $p_i \cdots p_j$, where $i \leq j$. With these definitions in mind, we will define our "merge" function as follows:

$$\textbf{merge}(a, b) = \{\forall x \in A | \exists y \in B, \textbf{y beats x in battle}\} \cup \{\forall y \in B | \exists x \in A, \textbf{x beats y in battle}\}$$

We take two subsets of "winning" Pokemon and compare each of them against each other. In our implementation, we will use merge on sets which are directly neighboring each other, where the winners have already been selected. As any winners of an overall set must be a winner from one side of the partition and able to beat at least one winner from the other side of the partition,

this merge function will merge the winner between two partitions.

Proposed Recurrence:
In order to solve the overall problem, we will create the following recurrence, for all pairs that satisfy $1 \leq i \leq j \leq n$ to build an $n \cdot n$ sized matrix of winner lists from every possible interval with respect to $p_0 \cdots p_n$:

$$OPT[i,j] = \left\{ \begin{array}{ll} p_i, & \text{if } i = j \\ \bigcup_{k=i}^{j-1} \text{merge } OPT[i,k], OPT[k+1,j], & \text{if } i \neq j \end{array} \right\} \tag{5}$$

Base case: for $n = 1$, $OPT[1,1]$ will return p_1 because of the first condition, which is optimal. For $n = 2$, $OPT[1,2]$ will merge $OPT[1,1]$ and $OPT[2,2]$, which will return the winner of the fight, p_1, p_2, which is the only solution possible in a tournament with 2 Pokemon. Therefore, for $n < 3$, we have proved optimally.

Proof of recurrence: We want

$LHS = RHS =$ (if $i = j$ in RHS and if $i \neq j$ in RHS)

First, the we have (if $i = j$ and if $i \neq j$) covers all cases of i, j in the RHS. Therefore, we can now prove :
$LHS =$ (if $i = j$ in RHS and if $i \neq j$ in RHS)

Proving $LHS =$ (if $i = j$ in RHS) is straightforward. If $i = j$, we have a sub tournament with one Pokemon. Therefore, only that Pokemon, p_i can win. This is returned by the recurrence, so $LHS =$ (if $i = j$ in RHS) is true.

Now we will prove $LHS =$ if $i \neq j$ in RHS
I propose that this recurrence will return the list of all possible winners for a given range i, j, where $i \neq j$. We want to prove that $OPT[i,j]$ correctly returns all Pokemon that could win in the sub tournament $i \cdots j$, within $1 \cdots n$. Therefore, let the range of the interval, or $j - i = x$ and by inductive hypothesis we will assume that all sets returned by OPT where $j - i < x$ are true, and we will show that the interval $j - i = x$ is true. Note that we have also already proved that $j - i < 3$ is true in our base case. Therefore, in order to prove correctness, we just need to prove induction. We will prove two lemma:

First Lemma: No Pokemon that can't win the tournament are returned by $OPT[i,j]$, when $i \neq j$.
Proof by contradiction: Assume that we have some Pokemon at original position p_e such that $i \leq e \leq j$ and p_e is incorrectly returned by $OPT[i,j]$ when it couldn't win the sub tournament $i \cdots j$. If $i = e = j$, then we only have p_e, so p_e can win. Therefore assume that we have $i \neq j$. As p_e is returned

by $OPT[i,j]$, then it must be returned by the merge of some partition of two subsets $OPT[i,k], OPT[k+1,j]$, where $i \leq k \leq j$, as $OPT[i,j]$ takes the union of all possible merged subsets. Therefore, if p_e is returned, we know that we have some k such that $OPT[i,j]$ is partitioned in a way that p_e is a winner in $OPT[i,k]$, (which we have to be true by inductive assumption, as $k-i < j-i$), and p_e must be able to beat some winner in $OPT[k+1,j]$(which we also have to be true by inductive assumption, as $j-(k+1) < j-i$). Therefore let us take the partition k to represent a way for p_e to win. This proves that there can be some partition in $OPT[i,j]$ where p_e is a winner in $OPT[i,k]$ and where some Pokemon that p_e can beat, say p'_e is a winner in $OPT[k+1,j]$. Therefore, this partition proves that p_e could win the tournament $i \cdots j$. Therefore, no Pokemon that can't win the tournament are returned when $i \neq j$.

Second Lemma: Every Pokemon that can win the tournament is returned by $OPT[i,j]$, when $i \neq j$. Proof by contradiction: Assume that we have some Pokemon at original position p_e such that $i \leq e \leq j$ and p_e is **NOT** returned by $OPT[i,j]$ when it could win the sub tournament $i \leq j$. If $i = e = j$, then we return p_e, and we would be done. Therefore assume that we have $i \neq j$. As p_e is not returned by $OPT[i,j]$, then it must have been removed in the merge of $OPT[i,k], OPT[k+1,j]$. As we know both $OPT[i,k]$ and $OPT[k+1,j]$ are correct by inductive assumption, we will consider some cases for p_e:

p_e is in neither set for all $OPT[i,k], OPT[k+1,j]$: If p_e isn't in a winner in either $OPT[i,k], OPT[k+1,j]$, because we check every partition of $k = i$ to $k = j-1$, then it must cannot be a possible winner for $OPT[i,j]$. Note that we know that $OPT[i,k], OPT[k+1,j]$ are optimal by inductive assumption, and if p_e never beats a Pokemon on either side of every partition in every possible partition between the two sets then we know p_e can never win, so $OPT[i,j]$ is a correct solution in this case.

p_e is in a set in some $OPT[i,k], OPT[k+1,j]$ As we have p_e is not in the final solution but it is in at least one set, by assumption, then for every k in $\bigcup_{k=i+1}^{j-1}$, p_e must have lost to some Pokemon on the other side of the partition for every partition possible, otherwise it would have been added to the winning set through the union function. Therefore, it is impossible for p_e to win, so $OPT[i,j]$ is a correct solution in this case.

As no Pokemon that can't win the tournament are returned and every Pokemon that can win the tournament is returned, our recurrence is correct for $i \neq j$ in RHS. Therefore, we have $LHS =$ if $i \neq j$ in RHS, which proves $LHS = RHS$ and correctness.

Implementation and run time: First we have merge function, which is $O(n^2)$, as in it's worst case it runs on $(\frac{n}{2}, \frac{n}{2})$ sized groups, performing $O(\frac{n^2}{4})$ com-

putations which is $O(n^2)$. We also iterate union over all dividers $k = i + 1$ to $k = j - 1$, which will be $O(n)$ run time for every instance, getting to $O(n^3)$ run time for OPT. This assumes that taking the union between two sets is constant time, and if not, we have another $O(n)$ for taking the union for a given linear run through, though we will assume union is constant in this analysis. We will run the OPT problem for every set of indexes i, j for $p_1 \cdots p_n$. There are $O(n^2)$ of these indexes, meaning we have $O(n^5)$ run time to generate all sub problems. Because we will save sub problems in a memorization table, we don't have to repeat work on subsequent recursion calls, and once we have computed the solution to a problem for the first time we can access it instantly through dynamic programming in the future. Finally, we access , for $OPT[0, n]$ in constant time for the solution, which gives $O(n^5)$ runtime.