Vatsal Jingar (2020CS50449)
Stitiprajna Sahoo (2020CS10394)

Assignment 2
COL351 (Analysis and Design of Algorithms), Semester-1 (2022-23)

1. Note : For proving the correctness of algorithms, we have provided many subclaims and claims in every section (For eg. Algorithm, Idea, or Seperate) which would help in building up the proof of correctness.
Sections:
1. Idea: for getting a general idea how algorithm is intuitively working and explanation of algorithm. including some proofs of sub-claims.
2. Algorithm: Pseudo code and working of algorithm

1 MAXIMUM SUM

Question: Let $D = (d_1, ..., d_n)$ be a disc with n sectors such that sector d_i in disc stores an integer p_i , for $i \in [1, n]$. Design an $O(n)$ time algorithm to compute a contiguous collections of sectors in D such that the sum of integers associated with the chosen sectors is maximum.

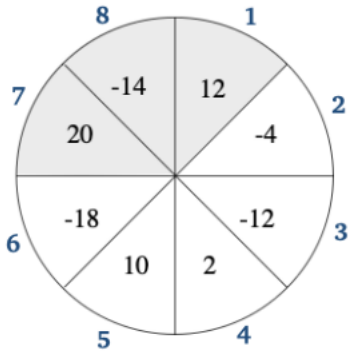

Figure 1: Depiction of a disc with $n = 8$ sectors. The contiguous collection of sectors that has maximum associated sum is $\{1, 7, 8\}$.

SOLUTION:

Idea:

- The question asks us to find a contiguous collections of sectors in D, with maximum sum of the integers present in the chosen sector.
- Let's consider the sectors to be present in form of array. If the array contains all negative elements, then the answer would be the maximum integer out of all, as adding any negative integer to another would only decrease the final sum value. Else, we have to look at all contiguous sub arrays of the array (ending at all indices) having a positive sum, and update the maximum sum to be the maximum of all these sub-arrays.
- We can maintain an array dp_{max} whose i^{th} element contains the maximum positive sum ending at index i. At iteration of i^{th} element, $dp_{max}[i]$ can be updated as

$$dp_{max}[i] = max(0, dp_{max}[i-1] + d_i)$$

- Now, for a circular disk array as given in the question, the above idea will compute the maximum sum of contiguous sectors if the sectors are $d_i, d_{i+1}, ...d_j$, such that $i < i+1 < ... < j$, i.e. the contiguous sectors would be a contiguous sub-array of the array of sectors $[d_1, d_2, ...d_n]$.
- But, if this condition is not satisfied, i.e. $(i > j)$ the contiguous sectors are in the form of $d_i, d_{i+1}...d_n, d_1, d_2 ...d_j$, i.e. the sectors don't form a contiguous sub-array of the array, then the idea can not compute this sum.

1.1 Claim: 1 $\implies$

- This case can be countered by the fact that, if the sum of contiguous sectors: $d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$ is maximum, then the sum of contiguous sectors $d_{j+1}, d_{j+2}, ...d_{i-1}$ will be negative. Because, had it been positive, then the total sum = sum($d_{j+1}, d_{j+2}, ...d_{i-1}$) + sum($d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$) will be greater than $d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$, and thus would contradict that the sum of contiguous sectors: $d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$ is maximum.
- This sum of contiguous sectors $d_{j+1}, d_{j+2}, ...d_{i-1}$ must be the minimum sum of a contiguous sectors of the disk ending at $i-1$, as total sum of all the sectors = sum($d_1, d_2, ...d_n$) = sum($d_{j+1}, d_{j+2}, ...d_{i-1}$) + sum($d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$). This can be proved as:
  Let S_1 = sum($d_i, d_{i+1}, ..d_n, d_1, d_2, ...d_j$) and S = sum($d_1, d_2, ...d_n$). This means sum($d_{j+1}, d_{j+2}, ...d_{i-1}$) $=S - S_1$
  $\implies$ $S_1 \geq S_2 \; \forall S_2$ = sum(all possible contiguous sectors of the disk), as we have stated that S_1 is maximum. $\implies$ $S - S_1 \leq S - S_2 \; \forall$ such S_2. Hence proved, $S - S_1$ is minimum sum.
- This set of contiguous sectors $d_{j+1}, d_{j+2}, ...di - 1$ form a contiguous part of the array $d_1, d_2, ...d_n$, as $i > j$. So, this set will give the minimum negative sum of any contiguous sub-array of D.

So, we can use a similar idea to compute the array dp_{min} for all indices as:

$$dp_{min}[i] = min(0, dp_{min}[i-1] + d_i)$$

.
Then compute the minimum of all such contiguous sub-array sums present in dp_{min}.

- In the final step, we can consider both the maximum values obtained from the first step, and the maximum of (sum-minimum sum of contiguous sub-array), which will consider all possible collections of contiguous sectors and find maximum possible sum out of those.

Algorithm:

- Initialise $sum = 0, n = size(D)$
- Initialize arrays dp_{max} and dp_{min} of length n such that $\forall i \in [0, n-1]$, $dp_{max}[i] = dp_{min}[i] = 0$
- For $i = 1$ to n, $sum = sum + d_i$
- For $i = 1$ to n, check if $\exists d_i$ such that $d_i \geq 0$ (d_i is non-negative).
- If there is no such d_i, then:
  - find maximum element of the set D, and this is the answer.
- Else:
  - Do the following for $i = 1$ to $n - 1$:
    1. $dp_{max}[i] = max(dp_{max}[i-1] + d_i, 0)$
    2. $dp_{min}[i] = min(dp_{min}[i-1] + d_i, 0)$.
  - maxpossum = max($dp_{max}[i] : \; \forall i \in [0, n-1]$)
  - minnegsum = min($dp_{min}[i]) : \; \forall i \in [0, n-1]$)
  - ans = max($maxpossum, sum - minnegsum$)

Time Complexity Analysis:

- *Time complexity of the above algorithm:* $O(n)$
- Analysis:
  - Computing sum of all elements of D will take $O(n)$ time, as it would require going through the array D one time.
  - Checking for the presence of non-negative elements in D would similarly take $O(n)$ time.
  - Finding maximum of all the elements would also take $O(n)$ (if no non-negative element present)

- In the next step, running the loop to update dp_{max} and dp_{min}, would require $O(n)$ time, as it would require iterating over the array one time only, and at each point requires access to only the previous element of the array.
- Thus, time complexity of the whole algorithm would be of the order of $O(n)+O(n)+O(n) = O(n)$. **So it takes linear time O(n) for computation**.

Proof of Correctness:

- Claim 1.1: *Let $dp_{max}[k]$ is the maximum positive sum of any contiguous sub-array ending at $d_k \forall k \leq i$, the algorithm computes $dp_{max}[i+1]$ as the maximum positive sum of any contiguous sub-array ending at d_{i+1}.*

  For any i, $dp_{max}[i]$ can be equal to either 0 or any positive value:

  - If $dp_{max}[i] = 0$, there exists no maximum positive sum, for any contiguous sub-array ending at d_i. So, if $d_{i+1} > 0$, $[d_{i+1}]$ can make the contiguous array with a positive sum, or if $d_{i+1} <= 0$, there can not be any contiguous sub-array with positive sum ending at d_{i+1}.
    Our algorithm computes $dp_{max}[i+1] = max(dp_{max}[i] + d_{i+1}, 0) = max(0 + d_{i+1}, 0)$, which would be d_{i+1} if it is positive else 0. So, this condition is handled correctly by the algorithm.
  - If $dp_{max}[i] > 0$, there exists a contiguous sub-array $[d_m, d_{m+1}, ...d_i]$, which has a positive sum and it is the maximum out of all such contiguous sub-arrays ending at d_i.
    For any contiguous sub-array to end at d_{i+1}, it will be either $[d_{i+1}]$ or $[d_m, d_{m+1}, ...d_i, d_{i+1}]$ for all $m < i$. We need the maximum of sum of all such sub-arrays having positive sum. Now, sum of any such sub-array would be of the form:
    * sum of $[d_m, d_{m+1}, ...d_i, d_{i+1}]$, which is equal to sum of $[d_m, d_{m+1}, ...d_i]$ + $d_i + 1$.
    * sum of $[d_{i+1}]$ only, which is equal to d_{i+1}

    As $dp_{max}[i] > 0$, it has a sub-array $[d_m, d_{m+1}, ...d_i]$ which has the maximum positive sum. So, let's say sum of sub-array $[d_m, d_{m+1}, ...d_i]$ = s, then as $s > 0$, $s + d_{i+1} > d_{i+1}$, so $s + d_{i+1}$ is the maximum possible sum of any contiguous sub-array ending at d_{i+1}, as s is the maximum of all contiguous sub-array sums ending at d_i. If $s + d_{i+1}$ is positive, it should be the answer for $dp_{max}[i+1]$, else $dp_{max}[i+1] = 0$. And our algorithm computes $dp_{max}[i+1] = max(dp_{max}[i-1] + d_i, 0)$, which is correct as if $dp_{max}[i-1] + d_i > 0$, then $dp_{max}[i] = dp_{max}[i-1] + d_i$, else 0.

  Hence, the algorithm computes $dp_{max}[i+1]$ as the maximum positive sum of any contiguous sub-array ending at d_{i+1}

- Claim 1.2: *Let $dp_{min}[k]$ is the minimum negative sum of any contiguous sub-array ending at $d_k \forall k \leq i$, the algorithm computes $dp_{min}[i+1]$ as the minimum negative sum of any contiguous sub-array ending at d_{i+1}*
  This can be proved in a similar way as Claim 1.1

- Claim 1.3: *If the contiguous sectors that give the maximum sum do not form a contiguous sub-array of the array D, then the compliment sectors will form a contiguous sub-array of D, and will have the minimum negative sum.*
  This has been proved in Claim 1.

- By the proofs of claim 1.1, 1.2 and 1.3, we can proof the correctness of our algorithm:

- If there is no positive numbered sector, then the answer should be the maximum numbered sector out of those, as adding any other sector's value or taking any other sector will decrease the sum further. This case has been handled correctly by the algorithm initially.

- Else, the algorithm computes both the maximum positive contiguous sub-array sums ending at every index of D as well as the minimum negative contiguous sub-array sums ending at every index of D.

- As we know, the maximum positive sum of the contiguous sectors has to end at one of the indices of D (if it is a sub-array of D), else it's compliment sectors have to end at one of the indices of D, and have a negative minimum sum (from Claim 1.3)

- All the maximum positive sums of conitguous sub-arrays of D and minimum negative sums of conitguous sub-arrays of D are stored correctly in the array dp_{max} and dp_{min} respectively. *(As proved in Claim 1.1)*

- The required answer should be maximum of all such positive sums and values of (total sum - negative sums). Our algorithm also takes all these values and computes their maximum. If there would have been any other sum of contiguous sectors whose sums would exceed the computed value, then it would have been a part of either $dp_{max}[i]$ or $sum - dp_{min}[i]$ for some index i, hence the algorithm would compute that value*hence contradicted*.

- Thus, the correctness of our algorithm is PROVED.

2 FOREX TRADING

Question: Suppose you are a trader aiming to make money by taking advantage of price differences between different currencies. You model the currency exchange rates as a weighted network, wherein, the nodes correspond to n currencies - $c1, ..., cn$, and the edge weights correspond to exchange rates between these currencies. In particular, for a pair (i, j), the weight of edge (i, j), say $R(i, j)$, corresponds to total units of currency c_j received on selling 1 unit of currency c_i.

1. Design an algorithm to verify whether or not there exists a cycle $(ci1, ..., cik, cik + 1 = ci1)$ such that exchanging money over this cycle results in positive gain, or equivalently, the product $R[i1, i2] \times R[i2, i3] \times ....R[ik1, ik] \times R[ik, i1]$ is larger than 1.
2. Present a cubic time algorithm to print out such a cyclic sequence if it exists.

SOLUTION 2.1:

IMPORTANT LEMMA: G has a 'negative weight cycle' reachable from source if and only if we can make improvement in the n^{th} iteration of the bellman ford algorithm.

- If we suppose $v_0 -> v_1 ---> v_k$ as a negative cycle. It will be reachable as graph is strongly connected. v_0 and v_k are same.
- If this is a negative cycle, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$
- We can prove this lemma by contradiction.
- If we assume, that we have $d_n(v_i) \leq d_n(v_{i-1})$ + w(v_{i-1},v_i) for all i = 1,...,k. (no improvement)
- Then doing this for all v_i, $1 \leq i \leq k$, we get
- $\sum_{i=1}^{k} d_n(v_i) \leq \sum_{i=1}^{k} d_n(v_{i-1})$ + $\sum_{i=1}^{k} w(v_{i-1}, v_i)$
- Now, we can see that $\sum_{i=1}^{k} d_n(v_{i-1})$ = $\sum_{i=0}^{k-1} d_n(v_i)$ = $d_n(v_0)$ + $\sum_{i=1}^{k-1} d_n(v_i)$ = $\sum_{i=1}^{k} d_n(v_i)$
- if we cut these two terms we get :
- $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$
- This contradicts our assumption. Hence proved.

- Proving other-side (converse) statement:
- We will prove contra-positive of this statement.
- So, the actual statement is Ïf improvement then it imples a negative weighted edge cycle."
- So, the contra-positive will be Ïf no negative weighted cycle in graph then it imples no improvement in n^{th} iteration."
- If we assume that Bellman-Ford correctly reports the shortest distance between source and other nodes if no negative cycle in $n - 1$ iteration.
- Then this statement can be reframed into Bellman-Ford Algorithm statement.

Idea :

- We know that Bellman Ford Algorithm can help in checking whether a negative weighted cycle exist or not.
- This problem can be solved using Bellman Ford algorithm with slight modification.
- We know that if product of n numbers is greater than 1, then the product of log of these numbers will be greater than zero, i.e. for any set of positive integers $a_1, a_2, ...a_n$:

  $$a_1 \times a_2 \times ....a_n > 1$$

  Also if similar condition holds, we know that log of reciprocal of these n numbers will be lesser than zero.

- So, if all the weighted edges are replaced by log of reciprocal of these n numbers, the problem will be modified to finding a cycle with sum less than 0. This problem can now be solved using bellman ford algorithm.

Algorithm:

- Assumption : The weight edges are always positive as they are exchange rates.
- E = edge set containing R(i,j), where i,j represents two countries as nodes and R(i,j) is exchange rate that is weight of edge.
- J = []
- For all edges e in E :
  Replace as new edge e_{new} := log(1/e)
  J := J + e_{new}
- G_{new} = (V,J)
- Bellman-Ford(G_{new}) := (Update edges $|V|$ times) // let N = $|V|$
- If $\exists$ (v_i,v_j) ϵ V, s.t, dis(v_i,v_j)$_{N-1}$! = dis(v_i,v_j)$_N$:
  return Yes
- return No

Time Complexity

- Creating new graph will take O(m), m = number of edges as we are updating all the edges.
- Time to run Bellman-Ford on new Graph will take O(mn), m = number of edges and n = number of vertices. *Shown in Lecture Slides*
- For checking whether the distance is updated, we have to at max check all the edges so, it will be O(m).
- Total Time = O(mn) + O(m) + O(m) = O(mn)

Proof of Correctness

- Claim 1 : Bellman ford can detect the negative edge cycles in n iterations, where n is number of vertices. This is proved in the class, Lecture on Bellman Ford. (proved in IMPORTANT LEMMA)
- Claim 2 : if $\prod_{i=1}^{n} e_i > 1$ then $\sum_{i=1}^{n} a_i < 0$, where $a_i = log(\frac{1}{e_i})$, and $e_i > 0$.
  This can be proved as we know that $\sum log(edge_i) = log(\prod edge_i)$
- So, if the edges are replaced from e_i to $log(\frac{1}{a_i})$, It can be observed that we would have to detect negative cycle, for the new graph, which can be done using Bellman-Ford Algorithm.

SOLUTION 2.2:

Idea :

- The idea from 2.1 can be extended to find the cycle.
- According to the idea of bellman ford algorithm, it can detect the negative cycle, in Nth iteration. So if any edge is updated in Nth iteration, pick a vertex of such edge and a DFS on it can provide the desired negative cycle.
- If vertex v is updated in Nth iteration, this will meant that it is present in negative weighted cycle.
- Creating a DFS tree, as it is present is cycle, there must be a back edge from any of descendants of vertex v to the ancestors of v. So DFS on v, and finding the descendant having highest point less than v, and so the path till that this descendant and path from ancestor to v, these all nodes will be part of negative weighted cycle.

Algorithm :

- T = DFS(G), G = Graph of currencies
- Bellman-Ford(G_{new}), G_{new} is the graph by replacing edges with log(1/e), generated in Question 2.1.

- Descendant = {}, Ancestor = {}
- if ∃ v ϵ V, s.t, $dis(v_i, v_j)_{N-1} != dis(v_i, v_j)_N$,
  DFS(v_i, T), while checking for all such decendant v, for which Level(Backedge(v)) < Level(v_i), Store Descendantv ∪ v, and Ancestor ∪ node(backedge(v)).
  (*We know, DFS(v), return descendants of v, By Lecture Slides*)
- AllcycleSets = {}
- For all v in Descendant :
  Path1 = Path from v_i to v, which will stored with DFS(v_u).
  Path2 = DFS(Ancestor.pop(),T) for reaching v_i.
  CycleSet = [], Sum = 0
  For ∀ v ϵ Path1, Path2
  CycleSet := CycleSet ∪ {v}, Sum += wtNew(v,previousV)
  If Sum < 0,
  AllcycleSets ∪ CycleSet
- return AllcycleSets.pop()

Assumptions :

- .pop() : refers to taking out last element from list.
- DFS(v,T) : Searching in tree T using DFS starting from node v.
- previousV : for reaching to v, the node which is used.
- Node(backedge(v)) : The node to which there is a backedge from v.
- wtNew : replacement of edge weights by there log(1/e).

Time Complexity of Algorithm:

- DFS(G), will take O(m+n), *Shown in Class*, where m = number of edges, and n = number of vertices.
- Bellman Ford will take O(mn), *Shown in Class*
- DFS(v1) and other two DFS are all O(m+n) in total O(3(m+n)) = O(m+n)
- Path is calculated during DFS.
- Creating CycleSet will take O($n(m+n)$), n = vertices as cycle can have at max all the vertices, as there can be n-1 vertices at most as the descendants of node v. And there can be back edge from all vertices at worst. And we will have to perform DFS search for all.
- In total it will take O(m+n) + O(mn) + O($n(m+n)$) = O(mn+n^2), as m is number of edges, it can be at max, O(n^2). So the total time complexity is O(n^3).

Proof of Correctness

- Claim 1 : Assuming that the negative cycle is detected correctly by previous step. If any edge is updated in N_{th} iteration, will meant that it is involved in negative cycle.
  By the claim of Bellman Ford.
- Claim 2 : DFS of v in T (DGS(G)) will correctly provide all the descendants of the node v.
  This is proved in class.
- Claim 3 : Back-Edge to any of the ancestor will mean cycle, because there is direct path from main node to descendant and if descendant can join to ancestor of node v, which will create a cycle, ancestor to main node, main node to descendant, descendant to ancestor. This meant that all the cycles involving main node will computed correctly.
- Claim 4 : Loop over all such descendants of v, where v is the node which is one of the end of edge involved in negative cycle will correctly provide all such cycles, which have sum less 0.
- This loop will correctly take all the descendants as we are considering every node from descendant list.
- Sum will be correctly computed, can be proved by induction over the Sum which will be 0 initially and will add each node in path after each iteration.

- CycleSet is correctly computed, this can be proved by above four claims.
- As CycleSet will contain the set of edges in cycle having total wight less than 0, the algorithm correctly print the sequence if such cycle exist. As the improvement is checked over all the edges, it will always provide the negative cycle if it exist.

3 DISJOINT COLLECTION OF PATHS

Question: Let $T = (V, E)$ be a rooted binary tree on n vertices, and S be a set of paths in the tree T. Two paths $P, Q \in S$ are said to be disjoint if they do not have a common edge. Design a polynomial in n time algorithm which finds a subset X of S of maximum size such that each pair of paths in X is disjoint.
The size of set S is at most ${}^nC_2$ since T is acyclic. Note that each path in S can be identified by just its endpoints.

Idea:

- We can solve this problem using the sub-problem of finding maximam cardinality of disjoint paths for a given nodes upto a maximum depth. Then for the next depth we can calculate the set using this sub-problem.
  For efficiency, we do the following pre-processing:
- Let the set of paths be $S = \{p_1, p_2, p_3, ...p_k\}$
- Initially we run a DFS of depth and label all nodes with its depth.
- For every edge in the tree, we store all the paths that pass through it in form of a list (we can name this list as *path-traverse list*)
- After having data about all the paths passing through all edges, we can store the node with minimum depth for each path. This can be in form of a list of nodes $N = n_1, n_2, ...n_k$, where n_i has the minimum depth among all nodes of p_i
- We can then create a $k \times k$ matrix M whose entries $M[i][j]$ will be equal to 1 if paths p_i and p_j are not edge disjoint, else 0. This can be done by the path traverse list for each edge, where each pair of paths present in the list of any edge will not be disjoint, hence it can be updated in the matrix.
- We have with us matrix M, path traverse list $\forall$ edges and minimum depth nodes list N.
- We will first sort the paths according to the depths of n_i, in non-increasing order. We can assume this sorted list to be $\{p_1, p_2, p_3...p_k\}$ where $depth(n_1) \geq depth(n_2) \geq depth(n_3), ... \geq depth(n_k)$
- We can claim (sub-claim 3.1) that if for 2 paths, the value of min-depth is same but the node is different, the paths will be disjoint. This can be proved that as n_i and n_j are the nodes with minimum depth, for the 2 paths, there can be no common edges in the paths that are in the parent tree of n_i or n_j. So, if a common edge exists between p_i and p_j, it will create a cycle pertaining to existence of 2 paths from n_i to n_j (one through the common edge and one through the the root of the tree)
- From here we will store maximum cardinal disjoint paths for every node rooted sub-tree, depth wise. Let the list be *DisjointList* This will create the solution to our sub-problem and help us find the maximum cardinal disjoint paths for the whole tree. For every node we can find the maximum disjoint paths set for the sub-tree rooted at it, by taking the maximum disjoint paths of sub-trees rooted in its child recursively.
- The working of the algorithm will be:

  BASE CASE:
- Let the depth of n_1 be d. d will be the maximum of all minimum depths of any path of the set S. (As the min-depth nodes sorted according to depth in decreasing order)
- $\forall$ nodes having depth d, we take the paths in its path traversal list and add it to the *DisjointList* for all nodes with same depth d. If any node as more than 1 paths with the node as their min-depth, then the node will take those paths which are disjoint (by the help of M). Atmost 2 paths are checked for all paths present.

  RECURSIVE STEP:
- For a node if the left child and right child have *DisjointList* $L_1 = \{p_1, p_2, ...p_a\}, L_2 = \{q_1, q_2, ...q_b\}$. For the node, we will check its path traverse list of the 2 edges that connect it to its children. Let all the paths be $D = \{d_1, d_2, ...d_c\}$.
- We can select atmost 2 paths out of D, as taking one more will result in common edge. (As all the paths are passing from either its left child edge or right child edge, only atmost 2 paths can be disjoint). $\forall$ paths d_i, d_j, we take maximum of disjoint paths from L_1, L_2 and D

- *Sub-claim 3.2*: $L_1 \cup L_2$ can never have overlapping edges. As, min-depth of all paths in L_1 must $\leq d_0$(d_0 is the minimum of minimum depth of all paths in subtree at L_1, as only those paths have been added to the sets yet), then all the nodes and edges will lie below it in the subtree rooted at node n_1, similarly all paths nodes and edges of L_2 will lie below n_2 in the subtree rooted at it. As it is a tree, there can be no cross edge among 2 sibling subtrees of a node, so there can be no common edge in any 2 paths from $L_1 \cup L_2$
- Recurrence relation: *DisjointList(parent)* = maximum of all three below:
  - disjoint paths from $(L_1 + L_2), \{d_i, d_j\} \forall (i, j)$
  - disjoint paths from $\{(L_1 + L_2), d_i\} \forall d_i$
  - $|L_1| + |L_2|$
- *Sub-claim 3.3:* This relation takes into consideration all possible cases of paths collection that can be added to the set, with all paths having depth $\geq depth(parent)$.
  This can be proved as all the paths having depth greater than $depth(parent)$ will lie either completely in its left subtree (would $_1$ it adding it will retain more disjoint paths of the subtree), or lie completely in right sub-tree, or will have either edge of the parent with its left or right child or both. We are checking disjoint property while adding every edge to the set, also checking maximality. So this will correctly return the maximum disjoint paths having minimum depth $\geq depth(parent)$. (Correctness of recurrence relation)
- This recurrence relation can be then applied at every node, for all nodes at a same depth, and this is continued depth wise, starting from bottom of the tree to its root. Finally we get max disjoint list of the subtree rooted at root, which will give the max disjoint list of the tree itself.

Algorithm:

Preprocessing:

- Create DFS(T), and store the depth for every node
- Initialise *path traverse list(edge)* $\forall$ edge $\in E$ = []
- For all path in S:
  If only end points of the path is given, then all its edges can be found out by DFS(T)
  - checkpath = path
  - For all edge in checkedpath:
    - checkedge = edge
    - *path traverse list* of checkedge = *path traverse list*(checkedge) $\cup$ checkedge

    for node in checkpath: *(We can initialise min depth node of Checkpath as None with depth infinity)*
    - min depth node(checkpath) = node with minimum (depth(node), depth(min depth node (checkpath)))
    - min depth (checkpath) = depth(min depth node(checkpath))
  - Initialize matrix M of size $k \times k$
  - For edge in E:
    For any 2 paths $\{p_i, p_j\} \forall \{i, j\}$
    M[i][j]=1 if overlapping edge present (not disjoint), else 0
- Sort S, with decresing order of min-depth.
- Assuming f(node, list) returns max disjoint list for subtree rooted at node:
  Let P = $p_1, p_2, ...p_t$ be the paths with min-depth = depth(node)
  $f(node, maxdisjointlist) = max$
  $(max(|f(leftchild, maxdisjointlist)| + |f(rightchild, maxdisjointlist)| + \{p_i, p_j\} \forall (p_i, p_j) \in P)$,
  $max(|f(leftchild, maxdisjointlist)| + |f(rightchild, maxdisjointlist)| + p_i \forall p_i \in P)$,
  $max(|f(leftchild, maxdisjointlist)| + |f(rightchild, maxdisjointlist)|))$
- *While doing this step, considering any p_i from set P will have to check disjointness with paths from maxdisjointlists of its subtree from M , and take only retain only those paths in disjointlist whose matrix returns 0*

- This step will recurse till leaves i.e. f(leaf,maxdisjoinlist)=null
- return f(rootnode,maxdisjointlist)

Time complexity :

$=>$ nk for creating path traverse list.
$=> nk^2$ for creating overlapping list.
Tree Step:

- T( ) : Time for computing
- T(depthParent) = T(depth/2) + T(depth/2) + T(checking all paths having min depth at that node)
- T(checking all paths) : for any level and same node, we can have at worst k paths.
- T(depthParent) = 2T(d/2) + O(k^2)
- T(d) = 2^d + 2^dO(k^2)
- T(d) = 2^dO(k^2)
- T(d) = nk^2, as 2^d = n
  As k = O(n^2), as we have at max n^2 paths. (path between any two nodes)
  T(d) = O(n^5)
  Total Time = O(n^5) + O(nk^2) (Creating Overlapping matrix) + O(nk) (Creating Traverse List)
  T(d) = O(n^5) = O(n^c) (Polynomial)

Proof of Correctness:

- Some claims needed to prove the correctness algorithm are written in idea section. *(sub-claims 3.1, 3.2 and 3.3)*
  - *SUB-CLAIM:3.1: If for 2 paths, the value of min-depth is same but the node is different, the paths will be disjoint*
  - *SUB-CLAIM:3.2: If maxdisjoint list of left child subtree of a node is L_1 and maxdisjoint list of right child subtree of a node is L_2, then $L_1 \cup L_2$ can never have a any pair of paths that is not disjoint*
  - *SUB-CLAIM 3.3: The recurrence relation takes into consideration all possible cases of paths collection that can be added to the set, with all paths having depth $\geq depth(parent)$, and takes the maximum out of them*

  Proofs of the above sub-claims have been provided in the Idea section above (Page 7,8)
- For creation of overlapping matrix, we can use induction over loop to prove that it is formed correctly.
- Recursive Step Correctness:
  - Base : For all nodes at leaves, no path below it can exist, so trivially, it will be empty.
  - Hypothesis : Suppose left child and right child correctly give max disjoint lists.
  - Induction Step : For any node till that node, the max disjoint paths till that node will be max over all possible combinations. As at any node we can have only two paths at most which can have passing through the given node, we will be covering all the possible disjoint combinations while adding the max disjoint list of left and right. So it will give correctly give the max disjoint list till that node. *Claim 3.3 proved in idea*
  - This induction step will prove that max disjoint list would be correctly computed for root node. Till root node as all the nodes are covered we will get max disjoint list over all paths in S.
  - Hence Proved.