Homework 4

Aidan Goodfellow

Saturday 18th November, 2023

1 Problem 1

Algorithm 1 Calculate Minimum Cost to Make Cake's Tree True

```
1: function MinCost(node)
2:     if node is null then
3:         return 0
4:     end if
5:     if node is a leaf then
6:         return node.value == false ? 1 : 0
7:     end if
8:     leftCost ← MinCost(node.left)
9:     rightCost ← MinCost(node.right)
10:    if node is an and node then
11:        return leftCost + rightCost
12:    else if node is an or node then
13:        return min(leftCost, rightCost)
14:    end if
15: end function
```

1.1 Proof of Correctness by induction

Base Case: Leaves

- A leaf node is either true or false. If it's true, no action is required, hence the cost is 0. If it's false, it needs to be flipped to true, costing 1 dollar.
- This is straightforward and clearly correct as per the problem's rule of changing a leaf's value for 1 dollar.

Inductive Step:

Handling 'and' Nodes:

- An 'and' node is true if and only if both of its children are true.

- The cost to make an 'and' node true is the sum of the costs to make both of its children true.
- Inductively, we assume that the costs computed for each child (subtree) are correct (minimal).
- Therefore, the total cost for the 'and' node, being the sum of these minimal costs, is also minimal and correct.

Handling 'or' Nodes:

- An 'or' node is true if at least one of its children is true.
- The algorithm computes the cost to make each child true and then chooses the minimum of these costs.
- By inductive hypothesis, the cost to make each child true is minimal.
- Choosing the minimum of two minimal costs ensures that the cost to make the 'or' node true is also minimal.
- This approach guarantees the 'or' node becomes true with the least possible cost.

- By correctly and minimally setting each node, the algorithm ensures that the whole tree evaluates to true with the minimal cost.

1.2 Time Complexity Analysis

1. Depth-First Search (DFS):

- The algorithm performs a DFS traversal of the tree. In DFS, each node in the tree is visited exactly once.

2. Handling Each Node:

- At each node (whether it's a leaf, 'and' node, or 'or' node), the algorithm performs a constant amount of work.
- For leaf nodes, it's a simple check of their boolean value.
- For 'and' and 'or' nodes, it involves computing the sum or minimum of the costs of their two children, respectively.

3. Cost Computation:

- The cost computation for each node is a simple arithmetic operation, which is an O(1) operation.

- Since each node is only visited once, and the work done at each node is done in constant time, the overall time complexity is $O(n)$

2 Problem 2

Algorithm 2 Determine if the Number of Paths from s to t is Odd or Even

```
    function IsNumberOfPathsOdd(G, s, t)
2:      order ← TopologicalSort(G)
        dp ← array of length |V| initialized to 0
4:      dp[s] ← 1
        for all u ∈ order do
6:          for all v ∈ AdjacencyList(u) do
                dp[v] ← dp[v] ⊕ dp[u]
8:          end for
        end for
10:     return dp[t] == 1
    end function
```

2.1 Proof of correctness by induction

Inductive Hypothesis:

- For a vertex v in the DAG, assume that 'dp[v]' correctly represents the parity of the number of paths from s to v.

Proof by Induction:

Base Case:

For the vertex s, 'dp[s]' is set to 1, as there is exactly one path from s to itself (the trivial path). This correctly represents an odd number of paths (1 path). The base case holds.

Inductive Step:

- Consider a vertex u where the inductive hypothesis is true (i.e., 'dp[u]' correctly represents the parity of the number of paths from s to u).
- Now, consider an edge (u, v) in the DAG. After processing u, we update 'dp[v]' using 'dp[v] = dp[v] XOR dp[u]'.
- The XOR operation has the property that it maintains parity: XORing two even counts or two odd counts results in an even count (0 in binary), and XORing an odd count with an even count results in an odd count (1 in binary).
- This update correctly adjusts the parity for the number of paths to v considering the additional paths through u.
- Since the DAG is traversed in topological order, every path from s to v that goes through any predecessor of v is accounted for exactly once, maintaining the correct parity.
- Thus, if 'dp[u]' is correct for all predecessors u of v, then 'dp[v]' will also be correct after the update.

By induction, if the parity count is correct for all vertices preceding v, it will be correct for v after processing all incoming edges to v. Since the graph is a DAG and vertices are processed in topological order, the parity count will be correct for all vertices, including the target vertex t.

2.2 Time Complexity Analysis

Topological Sort:

- The first step is to perform a topological sort of the DAG.
- A topological sort on a DAG can be performed using depth-first search (DFS), which has a time complexity of $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

Dynamic Programming Initialization:

- Initializing the DP array is linear with respect to the number of vertices, which is $O(|V|)$.

Traversal and Updating DP Array:

- After topological sorting, the algorithm iterates through each vertex in the sorted order and updates the DP array based on the edges.
- In the worst case, every edge in the graph will be considered once. Since the graph is a DAG, there are no back edges or cycles, ensuring that each edge is processed only once.
- Therefore, this step has a complexity of $O(|E|)$.

- The dominating factors in this algorithm are the topological sort and the traversal/update step. Both have a complexity of $O(|V| + |E|)$.
- Since these steps are performed sequentially and not nested, the overall time complexity of the algorithm is also $O(|V| + |E|)$.

3 Problem 3

Algorithm 3 Maximize Length of Increasing Contiguous Subsequence

```
function MaxLengthIncSubseq(arr)
    N ← length(arr)
    leftSeq ← array of N elements, all initialized to 1
    rightSeq ← array of N elements, all initialized to 1
    maxLength ← 0
    for i ← 1 to N − 1 do
        if arr[i] > arr[i − 1] then
            leftSeq[i] ← leftSeq[i − 1] + 1
        end if
    end for
    for j ← N − 2 down to 0 do
        if arr[j] < arr[j + 1] then
            rightSeq[j] ← rightSeq[j + 1] + 1
        end if
    end for
    for i ← 0 to N − 1 do
        maxLength ← max(maxLength, leftSeq[i])
    end for
    for i ← 1 to N − 2 do
        if arr[i − 1] < arr[i + 1] then
            maxLength ← max(maxLength, leftSeq[i − 1] + rightSeq[i + 1])
        end if
    end for
    return maxLength
end function
```

3.1 Proof of Correctness by induction

Base Case: Length of input array = 1

- For ‘leftSeq[0]‘ and ‘rightSeq[N-1]‘ (where ‘N‘ is the length of the input array), the length of the LIS is 1, since each is a single-element subsequence.

Inductive Step:

- Assume ‘leftSeq[i]‘ correctly stores the length of the LIS ending at ‘i‘ for all indices up to some ‘k‘ (i.e., for all ‘i k‘).
- When considering ‘leftSeq[k+1]‘, there are two cases:

1. If ‘arr[k+1] ¿ arr[k]‘, then ‘arr[k+1]‘ extends the LIS ending at ‘k‘. Therefore, ‘left-Seq[k+1] = leftSeq[k] + 1‘. This maintains the correctness as ‘leftSeq[k]‘ was assumed to be

correct.
2. If ‘arr[k+1] arr[k]‘, then the LIS ending at ‘k+1‘ is just the element ‘arr[k+1]‘ itself, making ‘leftSeq[k+1] = 1‘, which is correct by definition.

- when considering‘rightSeq[k-1]‘, there are two cases:

1. If ‘arr[k-1] ¡ arr[k]‘, then ‘arr[k-1]‘ can be included in the LIS starting at ‘k‘, extending it by one. Thus, ‘rightSeq[k-1]‘ should be ‘rightSeq[k] + 1‘.
2. If ‘arr[k-1] arr[k]‘, then the LIS starting at ‘k-1‘ does not include ‘arr[k]‘. Therefore, ‘rightSeq[k-1]‘ is just the element ‘arr[k-1]‘ itself, making its length 1.

- The maximum length of an LIS without removing any element is the maximum value in ‘leftSeq‘. This is straightforward since ‘leftSeq[i]‘ gives the length of the LIS ending at each ‘i‘, as proven above.

- When considering the removal of an element ‘arr[i]‘, if ‘arr[i-1] ¡ arr[i+1]‘, then removing ‘arr[i]‘ can potentially join the subsequences ending at ‘i-1‘ and starting at ‘i+1‘. The correctness of this step relies on the correctness of ‘leftSeq‘ and ‘rightSeq‘.

- The length of the LIS that would result from removing ‘arr[i]‘ is ‘leftSeq[i-1] + rightSeq[i+1]‘. This is correct because ‘leftSeq[i-1]‘ and ‘rightSeq[i+1]‘ correctly reflect the lengths of the LISs ending before and starting after ‘arr[i]‘, respectively.

3.2 Time Complexity Analysis

1. Initialization of Arrays: The algorithm starts by creating two arrays, ‘leftSeq‘ and ‘rightSeq‘, each of size ‘N‘, where ‘N‘ is the length of the input array ‘arr‘. This initialization step has a time complexity of O(N).

2. Building ‘leftSeq‘ Array: The algorithm iterates through the array ‘arr‘ from left to right, filling in the ‘leftSeq‘ array. This step involves a single pass through the array, with simple comparisons and arithmetic operations at each step. Therefore, this part also has a time complexity of O(N).

3. Building ‘rightSeq‘ Array: Similarly to the ’leftSeq’ array, the algorithm iterates from right to left through the array to fill in the ‘rightSeq‘ array. This is also a single pass through the array, resulting in a time complexity of O(N).

4. Calculating Maximum Length Without Removal: The algorithm makes another pass through the ‘leftSeq‘ array to find the maximum length of an increasing subsequence without removing any elements. This is a linear scan of the array, contributing O(N) to the time complexity.

5. Calculating Maximum Length With Single Removal: The algorithm considers the effect of potentially removing each element. It iterates once through the array, using both 'leftSeq' and 'rightSeq' to calculate the potential maximum length if an element were removed. This step is a linear scan of the array, adding another O(N) to the time complexity.

Total Time Complexity:

the total time complexity of the algorithm is O(N) + O(N) + O(N) + O(N) + O(N). these loops are not nested, so the total time complexity simplifies to O(N).

4 Problem 4

Algorithm 4 Determine Winner for Each Starting Position

```
1: function FindWinner(a, n)
2:     dp[1 ... n] ← array of size n                ▷ Dynamic programming array
3:     winner[1 ... n] ← array of size n
4:     for i ← n down to 1 do
5:         dp[i] ← False
6:         j ← i + a[i]
7:         while j ≤ n do
8:             if a[j] > a[i] then
9:                 dp[i] ← dp[i] ∨ ¬dp[j]
10:            end if
11:            j ← j + a[i]
12:        end while
13:        if dp[i] then
14:            winner[i] ← "Jane"
15:        else
16:            winner[i] ← "Joe"
17:        end if
18:    end for
19:    return winner
20: end function
```

4.1 Proof of correctness using induction

Base Case:

- For $i = n$:

- If n is the last card, no move is possible since there is no greater label to move to, making it a losing position for Jane (who starts the game). So, dp[n] should be False, which aligns

with the algorithm's initialization.

Inductive Hypothesis:

- Assume that for all positions k where $n \geq k > i$, the algorithm correctly determines whether k is a winning or losing position (i.e., dp[k] is True for a winning position and False for a losing position).

Inductive Step:

- To prove the correctness for position i:

- The algorithm checks all positions j where $j > i$ and $(j - i) \mod a[i] = 0$ (all valid moves from i).

- If there exists any j for which dp[j] is False (indicating that j is a losing position for the opponent), then i is a winning position for the current player. This is because the player can move to j and leave the opponent in a losing position.

- If no such j exists (i.e., all valid moves lead to winning positions for the opponent), then i is a losing position, as any move from i will lead to a situation where the opponent can win.

- By induction, if the algorithm correctly identifies winning and losing positions for all $k > i$, then it can correctly determine if position i is a winning or losing position as well.

4.2 Time Complexity Analysis

1. Initialization of 'dp' and 'winner' Arrays:

- The algorithm starts by initializing two arrays of size n. This operation has a time complexity of $O(n)$ as it involves setting up each element in these arrays.

2. Backward Iteration Over Cards:

- The algorithm then iterates backward from card n to card 1. For each card i, it examines potential moves to other cards j.

3. Checking Possible Moves:

- For each card i, the algorithm checks all cards j where $j > i$ and $(j - i) \mod a[i] = 0$. The number of such cards j for each i varies.
- The number of multiples of a number within a range decreases roughly in proportion to the number itself. For example, a number k has about $\frac{n}{k}$ multiples within the range $[1, n]$.
- Therefore, when summing over all cards, the total number of operations for checking moves can be approximated by the sum of the harmonic series: $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$, which is $O(\log n)$.

4. Overall Time Complexity:

- The backward iteration itself is $O(n)$, as it involves a single pass through the array.
- Within this iteration, the checking of possible moves adds an $O(\log n)$ complexity for each card, leading to an overall complexity of $O(n \log n)$.

5 Problem 5

Algorithm 5 Maximize Total Reward in Snail Pairing

```
1: procedure MaxReward(M, n)
2:     Let DP[n][n][n] be a new 3D array
3:     for i = 1 to n do
4:         for j = 1 to n do
5:             DP[i][j][0] ← 0
6:         end for
7:     end for
8:     for k = 1 to n do
9:         for i = 1 to n − k + 1 do
10:            j ← i + k − 1
11:            DP[i][j][k] ← −∞
12:            for m = i to j − 1 do
13:                reward ← M[i][m] + DP[i + 1][m − 1][k − 2] + DP[m + 1][j][k − 1 − m]
14:                if reward > DP[i][j][k] then
15:                    DP[i][j][k] ← reward
16:                end if
17:            end for
18:        end for
19:    end for
20:    return DP[1][n][n]
21: end procedure
```

5.1 Proof of Correctness using Induction

Induction Hypothesis:

For a given segment of snails of length k (where $1 \leq k \leq n$), the value in the DP table, $DP[i][j][k]$, correctly represents the maximum reward achievable for the segment of snails starting from snail i and ending at snail j, where $j = i + k - 1$.

Base Case:

The base case occurs when $k = 1$. In this case, the segment consists of only one snail,

so no pairing is possible. Therefore, the reward for any segment of length 1 is 0. This is correctly initialized in the DP table.

Inductive Step:

Assume that the DP table correctly calculates the maximum reward for all segments of length k (the induction hypothesis). We need to show that the DP table also correctly calculates the maximum reward for segments of length $k + 1$.

For a segment of length $k + 1$, the algorithm considers all possible pairings of the first snail in the segment with another snail in the segment. For each such pairing, it calculates the total reward as the sum of the reward for this pair plus the maximum rewards for the two remaining segments (before and after the chosen pair) which are of length less than $k + 1$, so their rewards would have been calculated already.

Since by the induction hypothesis, we know that the DP table correctly calculates the rewards for segments shorter than $k + 1$, the sum of these rewards plus the reward for the chosen pair gives the correct maximum reward for the segment of length $k + 1$.

5.2 Time Complexity Analysis

1. Initialization:

- The algorithm initializes a 3D array ‘DP‘ of size ‘n x n x n‘, where ‘n‘ is the number of snails. This initialization is done in $O(n^2)$ time since it involves setting up a 2D slice for each snail.

2. Dynamic Programming Loop:

- The main part of the algorithm is a nested loop structure.
- The outer loop runs for each possible size of the segment, ‘k‘, which goes from 1 to ‘n‘. This gives us $O(n)$ iterations for the outer loop.
- The middle loop runs over the starting point of each segment, ‘i‘. Since the segment length is ‘k‘, the loop runs approximately ‘n - k‘ times for each value of ‘k‘. On average, this will also contribute to $O(n)$ complexity.
- The innermost loop considers potential mates within the segment. In the worst case, this loop runs $O(n)$ times.

3. Reward Calculation and DP Table Update:

- Inside the innermost loop, the algorithm calculates the reward for each possible pairing and updates the DP table. This calculation and update are done in constant time, $O(1)$, for each iteration of the loop.

Overall Time Complexity:

The overall time complexity is determined by the nested loops. Since each loop contributes $O(n)$ complexity, and there are three nested loops, the total time complexity of the algorithm is $O(n^3)$.

6 Problem 6

Algorithm 6 Longest Increasing Path in a Weighted Directed Graph with Handling of Equal Edge Weights

Input: Graph $G = (V, E)$ with weighted edges
Output: Length of the longest path where each edge weight is strictly greater than the previous edge

function LONGESTINCREASINGPATH(G)
    Sort E in ascending order of edge weights
    Initialize a Segment Tree/BIT, *tree*, with size $|V|$
    Initialize an array $lastEdgeWeight$ with size $|V|$ and set all values to a minimum possible value
    for each edge (u, v, w) in sorted E **do**
        $maxPath \leftarrow$ QUERYTREE($tree, u$)
        $lastWeight \leftarrow lastEdgeWeight[u]$
        if $lastWeight < w$ **then**
            UPDATETREE(tree, v, maxPath + 1, w)
            $lastEdgeWeight[v] \leftarrow w$
        end if
    end for
    return MAXVALUE(tree)
end function
procedure UPDATETREE($tree, v, value, weight$)
    if QUERYTREE(tree, v) ¡ value **then**
        Update $tree[v]$ to $value$ and update the corresponding last edge weight to $weight$
    end if
end procedure
procedure QUERYTREE($tree, v$)
    return Query the Segment Tree/BIT for the longest path ending at v
end procedure

6.1 Proof of correctness using induction

Base Case:

- Initially, when no edges have been processed, the longest path for each vertex is 0, and the last edge weight is set to a minimum value. This correctly represents the initial state with no paths.

- with a graph of only one node, there will be no paths and the correct result would be 0. The algorithm correctly handles this case.

Inductive Hypothesis:

- Assume that after processing the first k edges in the sorted list, the Segment Tree/BIT and 'lastEdgeWeight' array correctly maintain the length of the longest path and the weight of the last edge of this path for each vertex.

Inductive Step:

- Consider the $(k+1)$-th edge (u, v, w). The algorithm queries the Segment Tree/BIT for the longest path ending at u and checks the weight of the last edge in this path (using 'lastEdgeWeight[u]').
- If the weight of the $(k+1)$-th edge w is greater than the last edge weight, the algorithm updates the path information for v in the Segment Tree/BIT and 'lastEdgeWeight[v]'.
- This ensures that each edge in the path is strictly greater in weight than its predecessor, handling the case of equal weights in the sorted list of edges.

- By induction, after all edges have been processed, the Segment Tree/BIT and 'lastEdgeWeight' array correctly represent, for each vertex, the length of the longest path ending there and the weight of the last edge in this path. The maximum value in the Segment Tree/BIT gives the length of the longest path of increasing edge weight in the entire graph.

6.2 Time Complexity Analysis

1. Sorting the Edges:

- Sorting all edges by their weights is the first step. If the graph has $|E|$ edges, this sorting can be done in $O(|E| \log |E|)$ time. Since $|E|$ can be at most $|V|^2$ for a dense graph, this is also $O(|E| \log |V|)$ in the worst case since with the logarithm property $O(|E| \log |V^2|) = O(|E|2log|V|) = O(|E|log|V|)$

2. Initializing the Segment Tree/BIT and Last Edge Weight Array:

- Initializing the Segment Tree/BIT for $|V|$ vertices will take $O(|V|)$ time.
- Similarly, initializing the 'lastEdgeWeight' array for $|V|$ vertices is also $O(|V|)$.

3. Processing the Edges:

- The algorithm iterates through each sorted edge. For each edge (u, v, w), it performs operations on the Segment Tree/BIT.
- Each operation (query and update) on the Segment Tree/BIT is $O(\log |V|)$. Since there are $|E|$ edges, and for each edge we perform at most two such operations, the total time for

processing all edges is $O(|E| \log |V|)$.

4. Finding the Longest Path:

- The final step involves finding the maximum value in the Segment Tree/BIT, which can be done in $O(\log |V|)$.

The overall time complexity is dominated by the sorting and the processing of edges, leading to a total complexity of $O(|E| \log |V|)$.

7 Problem 7

Algorithm 7 Minimum Deletions to Empty String with Identical Substrings

```
1: procedure MinDeletions(s)
2:     n ← length of s
3:     Let dp[0...n−1][0...n−1] be a new 2D array
4:     for i ← 0 to n − 1 do
5:         dp[i][i] ← 1
6:     end for
7:     for len ← 2 to n do
8:         for i ← 0 to n − len do
9:             j ← i + len − 1
10:            dp[i][j] ← ∞
11:            if s[i] = s[j] then
12:                dp[i][j] ← min(dp[i][j], 1 + SearchForOptimalSplit(i, j))
13:            end if
14:            for k ← i to j − 1 do
15:                dp[i][j] ← min(dp[i][j], dp[i][k] + dp[k + 1][j])
16:            end for
17:        end for
18:    end for
19:    return dp[0][n − 1]
20: end procedure
21: function SearchForOptimalSplit(i, j)
22:                    ▷ Perform binary search within s[i..j] to find the optimal split point
23:    return minimum number of deletions found through this search
24: end function
```

7.1 Proof of correctness using induction

Inductive Hypothesis:

- For a substring ‘s[i..j]‘ of the string ‘s‘, the algorithm correctly computes the minimum

number of deletions required to remove all characters in that substring.

Base Case:

- For a substring of length 1, i.e., a single character, the algorithm sets ‘dp[i][i] = 1‘, since one deletion operation is needed to remove a single character. This is correct and satisfies our base case.

Inductive Step:

- Assume the hypothesis is true for all substrings of length less than ‘l‘.

1. Substrings of Length l:

- Consider a substring ‘s[i..j]‘ of length ‘l‘. We need to find the minimum number of deletions to remove all characters in this substring.

2. Dividing the Substring:

- The algorithm considers every possible split point ‘k‘ within the substring. For each split point, it calculates the minimum number of deletions by either combining or separately deleting the parts divided by ‘k‘.

3. Optimizing Deletions:

- If ‘s[i] == s[j]‘, the algorithm tries to find an optimal way to combine these characters with the middle part to minimize deletions. This could potentially reduce the number of deletion operations if there's a way to delete the entire sequence at once.
- Otherwise, the algorithm computes the sum of the minimum deletions needed for ‘s[i..k]‘ and ‘s[k+1..j]‘. By our inductive hypothesis, these values have been correctly calculated for substrings shorter than ‘l‘.

4. Minimum of All Combinations:

- By considering all split points and calculating the minimum deletions for each scenario, the algorithm ensures that the minimum number of deletions for ‘s[i..j]‘ is found.

7.2 Time complexity Analysis

1. Initialization of the DP Table:

- The algorithm initializes a 2D array ‘dp‘ of size ‘n x n‘, where ‘n‘ is the length of the string ‘s‘. This initialization has a time complexity of $O(n^2)$.

2. Iterating Over Substring Lengths:

- The algorithm iterates over all possible lengths of substrings, which contributes an $O(n)$ complexity.

3. Iterating Over All Substrings for a Given Length:

- For each substring length, the algorithm iterates over all possible starting points of the substrings, leading to another $O(n)$ complexity.

4. Calculating Minimum Deletions for Each Substring:

- For each substring 's[i..j]', the algorithm performs calculations that include a binary search. The binary search operation contributes a $O(\log n)$ complexity.
- Additionally, there is a nested loop for considering all possible split points 'k' within the substring, which adds another $O(n)$ complexity.

The final time complexity of the algorithm is $O(n^3 \log n)$. This comes from the combination of:

- Two nested loops for choosing start and end points of substrings ($O(n^2)$).
- A loop for choosing split points and the binary search within each iteration ($O(n \log n)$).

$O(n^2) \times O(n \log n) = O(n^3 \log n)$.