Project: Lumberjack
Team[17]= Techno Traversers

Team Members

Sl. No.	*Student Name*	*Roll Number*
1	B Siddharth Prabhu	200010003
2	Siddharth Shankar	200030056
3	Rajat Lavekar	200010045
4	Patel Shahil Manishbhai	200010039

Problem Statement

To conduct a theoretical study of the "LUMBERJACK" problem listed in Optil.io

Algorithm of Approach and Main Idea

The design of algorithm was made to maximize the profit earned by cutting the trees as well as by the Domino effect.

The things to work on for maximizing profit were:

a. Maximum profit in the given time (greedy approach)
b. Taking advantage of Domino Effect and updating the profit values of trees whose profit might be affected by certain trees falling/being cut.

The main idea is to first go to the tree nearest to the origin, and then find the tree nearest to this tree which has maximum value of profit/time (including domino effect, if any). After cutting this tree with maximum value, we delete the information of this fallen tree from our list of all trees and update the current coordinates to the coordinates of the next tree which is to be cut in the next iteration. We repeat this process until the time is over.

Choice of Data Structure

a. Data Structures used: Vectors; Vectors of structures.
b. Method of accessing elements: Iterators and Indexes.
c. Reason of using the data structure: Easy Storage + Accessibility (Traversability) + Easy Manipulation.
d. Also, erasing a tree will not leave behind an empty space in the tree vector. There is no need to re-adjust tree indexes, they "automatically" update due to that nature of vectors

Environment Variables

a. Structure ***tree_gen***: used to store the data of a tree.
b. Variables ***value*** and ***weight***: used to store the value and weight of the tree. (value = p*h*d, weight = p*h*c)
c. Vector ***domino_list*** in ***tree_gen***: used to store list of trees that would be falling if the current tree is cut.
d. Variable ***dir***: used to store the directions {up, down, right, left} in which the next tree would be cut.
e. Vector ***<tree_gen> tree***: used to store all the trees in the forest.
f. Variables ***x_curr*** and ***y_curr***: used to store the coordinates of currently visited tree.
g. Variables ***x_next*** and ***y_next***: used to store the coordinates of the tree to be visited next.
h. Variable ***TOTAL***: used to store the values of all the trees in the vector tree.
i. Variable ***t***: the time given to cut down the trees.
j. Variables ***n*** and ***k***: the size of given grid and number of trees respectively.

A Brief Description of Functions

1. ***void profit_finder()*** - This function gives the direction in which the profit is maximum and updates the profit of each tree in that direction. It works with the help of the directional profit functions.

2. ***int profit_up()*** - This function returns the total profit that can made by when the tree is cut in the up-direction. It finds the nearest neighbour of the current tree in up direction and checks whether that tree allows for domino effect. Then, it makes a recursive call to find whether the next nearest neighbour of the given tree (in the up direction) can take part in the domino effect. (Similarly, ***profit_down(), profit_left(), profit_right()***)

3. ***void find_next()*** - This function chooses the next tree to be cut. It first computes the feasibility of cutting a tree, i.e., the "time cost" of reaching that tree plus the time to cut that tree. If this can be done within the time left, then it calculates the profit/time for that tree. Then it finds the maximum of this quantity among all the trees and the next tree to be cut will be the tree which has the maximum profit/time ratio.

4. ***void print_moving()*** - This function prints the path that is to be followed to reach the required tree, from (x_curr, y_curr) to (x_next, y_next).

5. ***void print_cutting()*** - This function prints the cutting of the tree that we have arrived at. It chooses the direction that maximizes profit due to domino effect. Also, it updates the max_profit of trees in the same row/column as the tree cut, since their domino effect might be affected

Note (An explanation for when we return to main() after cutting a tree):
1. After calling print_cutting() function, we delete the information of the tree that is being cut, and that of the fallen trees, from the vector tree.
2. Since the domino sums of the trees in line of fallen trees may get affected, we call profit_finder() to update their directional profits.

Drawbacks of our method

1. Because of the greedy approach, we aren't cutting small trees that we may encounter on the way, i.e., trees that wouldn't take long to cut; we are missing contribution due to these trees in our total profit.
2. Our solution to this problem (locally optimal solution) may not be the best solution to this problem (globally optimal solution). This can happen as we are following a greedy approach.

Time Complexity Analysis

We will start by finding complexities of the smaller component functions before building up towards main function!

1. ***int profit_up()*** – The for loop that finds domino effect in upward direction first looks for nearest tree in upward direction (O(k) time complexity) and it calls itself recursively for a maximum of k-1 times, which is also O(k). So, net worst case time complexity is $O(k^2)$ for the directional profit functions (Similarly, ***profit_down(), profit_left(), profit_right()***)

2. ***void profit_finder()*** – For a given tree of index 'i', this function essentially calls all the 4 directional profit functions and checks which direction has the best profit. There aren't any extra loops within, so its time complexity is $O(4k^2)$ (= $O(k^2)$)

3. ***void find_next()*** – This function has a for loop that runs statements a maximum of O(k) times, and the nested-if statements don't impact time complexity in a way to affect its big O value. So, this function is O(k).

4. ***void print_moving()*** – Since the grid is of (n x n) dimensions, this function has O(n) time complexity.

5. ***void print_cutting()*** – There is one for loop in this function that updates max profit of in-line trees, that calls profit_finder() a maximum of 2n-1 times, which would give it a time complexity of $O(nk^2)$.

6. ***int main()*** – Taking inputs and finding tree closest to origin are *O(k)* each. Function profit_finder() is called for all k trees, so that is $O(k^3)$. Through one iteration of the while(t>0) loop, in the worst case, t would decrement by just 2 units (1 moving, 1 cutting). So, the while loop occurs a maximum of t/2 times. Within it, find_next(), print_moving() and print_cutting() are called (*O(k)*, *O(n)* and $O(nk^2)$ respectively). Fallen trees are removed in a for loop that occurs O(n) times, with another nested O(k) for loop, which gives *O(nk)*. Another for loop of *O(k)* removes the cut tree from the all-info vector.

Hence, net time complexity would roughly be:

$O(k) + O(k^3) + O((t/2)*(k + n + nk^2 + nk + k)) =$ ***$O(k^3 + tnk^2)$***

Role of team members

Student Name	***Role***
Siddharth P	Implementing main code, Compilation and coordination.
Siddharth S	Implementing functions 1, 4; Report writing.
Rajat	Code Organization; Directional Profits; Documentation.
Shahil	Implementing functions 3, 5; Report writing.