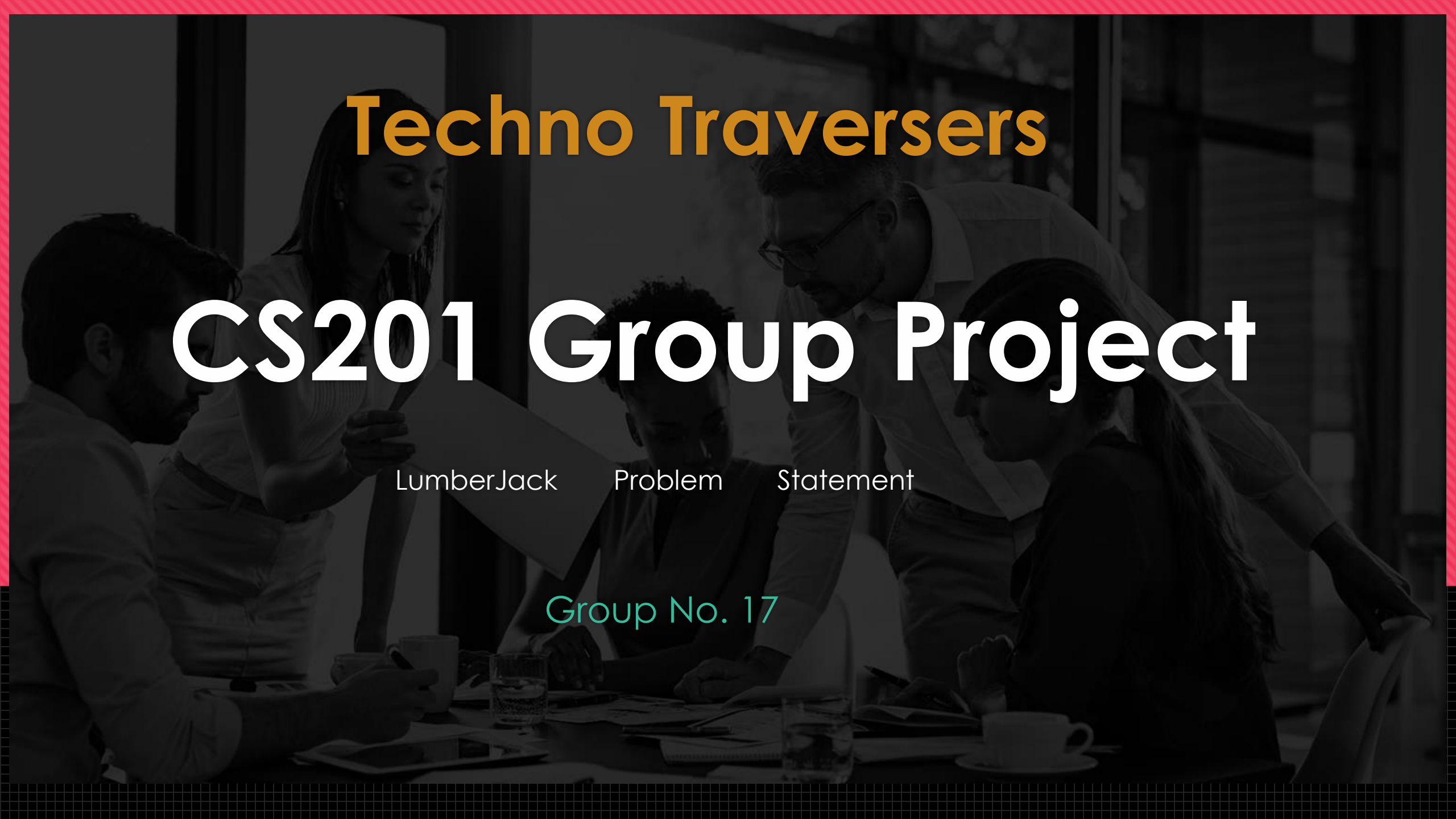
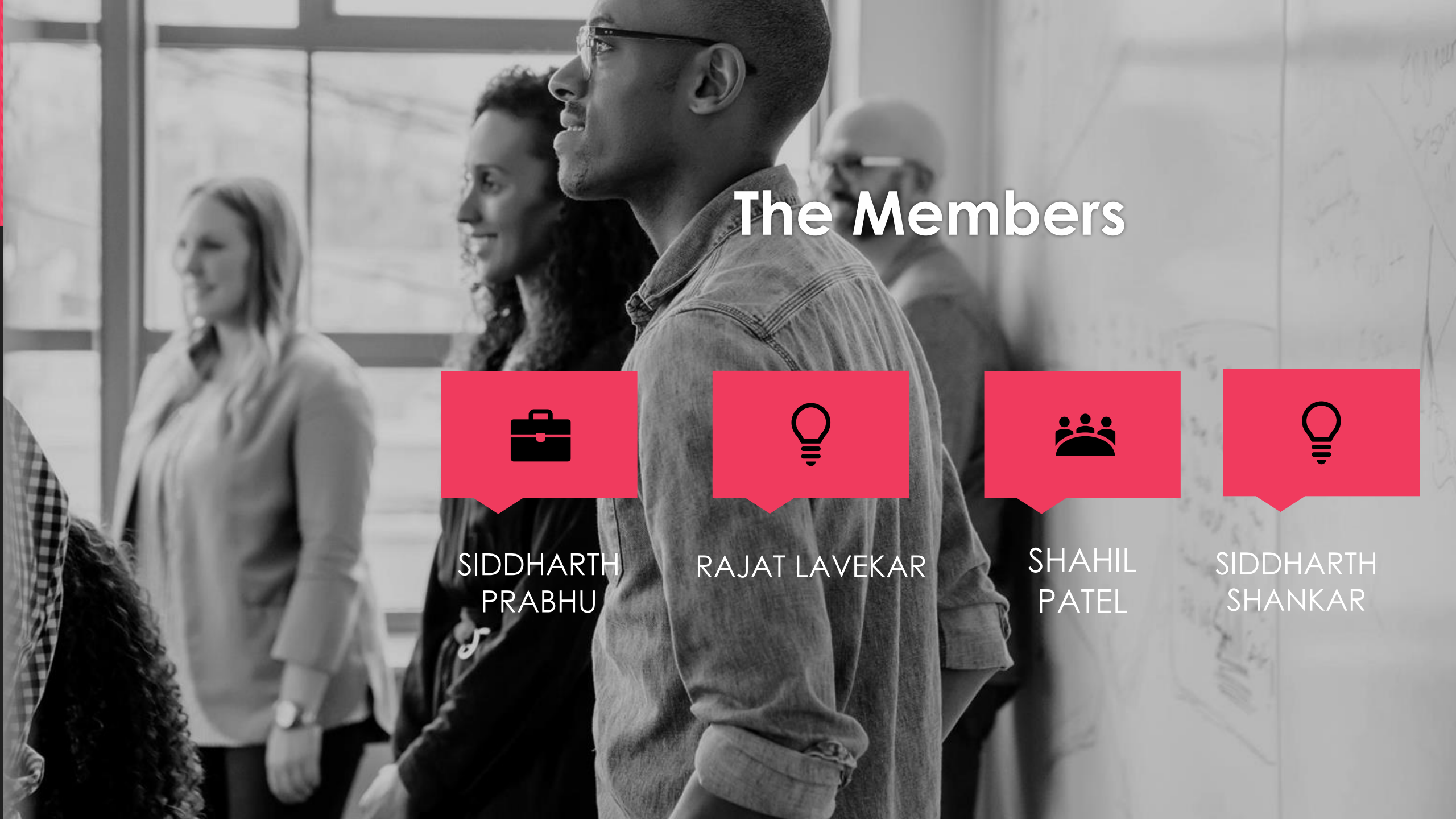


Techno Traversers

CS201 Group Project

LumberJack Problem Statement

Group No. 17



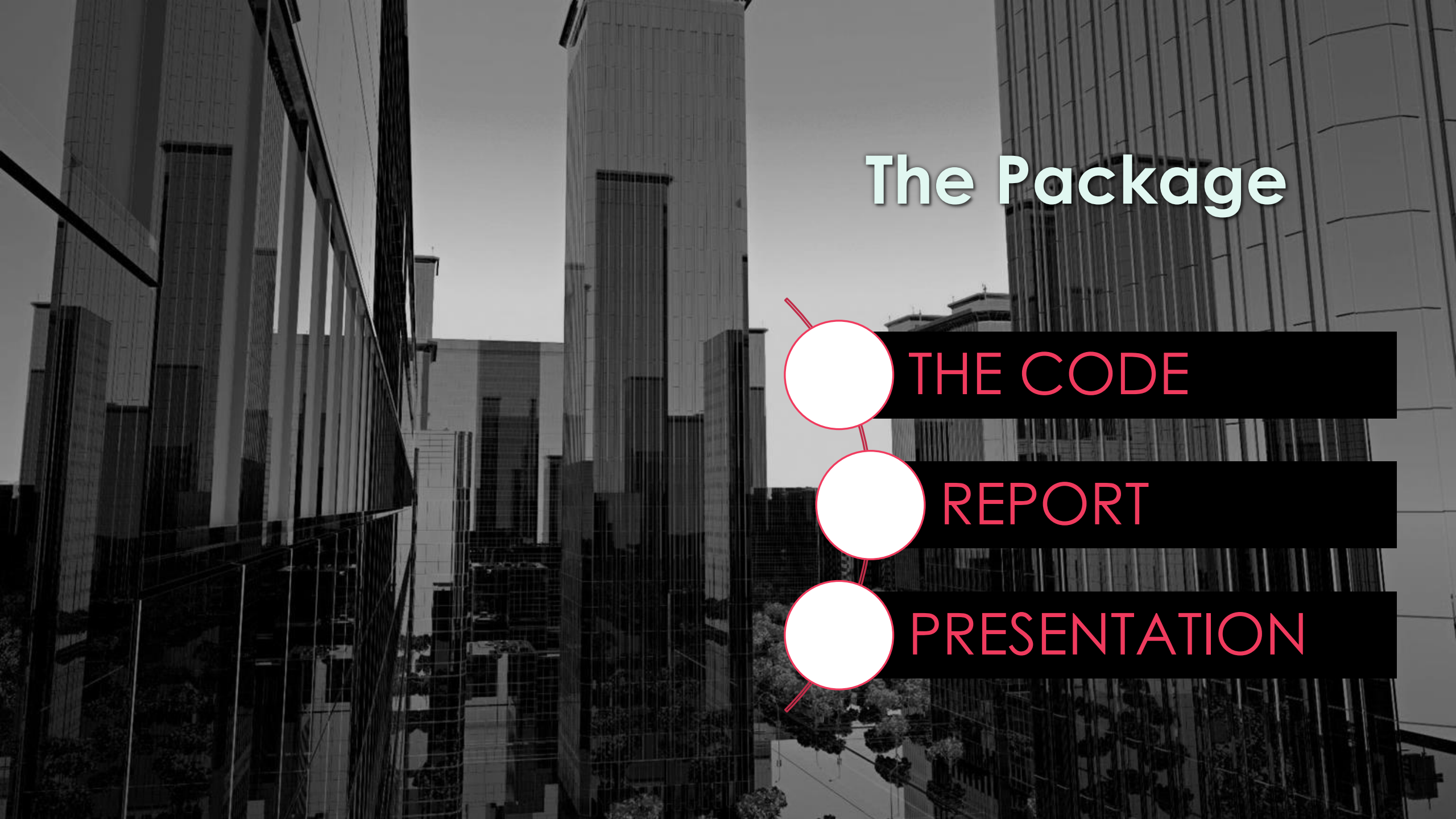
The Members

SIDDHARTH PRABHU

RAJAT LAVEKAR

SHAHIL PATEL

SIDDHARTH SHANKAR


The Package
THE CODE
REPORT
PRESENTATION

144129038

Our Generated Profit

Our Algorithm of Approach

The design of our algorithm is made in such a manner to maximize the profit earned by cutting trees and considering the domino effect induced by it.

The things to work on for maximizing profit were:

1. Maximum profit in the given time (greedy approach)
2. Taking advantage of Domino Effect and updating the profit values of those 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

1. Data Structures used: Vectors; Vectors of structures.
2. Method of accessing elements: Iterators and Indexes.
3. Reason of using the data structure: Easy Storage + Accessibility (Traversability) + Easy Manipulation.
4. 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

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

Functions

Profit_up (int curr, vector<int> up_vec)

Time Complexity : $O(k^2)$

- This function finds the max profit which can be found cutting once in the up direction of a current tree.
- The Domino effect is taken into consideration by this function.
- It makes a recursive call to calculate the profit by the domino effect.
- The variable up_ind is used to denote the tree which lies directly above the current tree
- The current tree is initialized by int curr, which after recursion gets changed to up_ind
- This function returns up_profit, i.e. the max profit in the up direction.

```
int profit_up(int curr, vector<int> &up_vec)
{
    // if a tree is up from current tree on the map, we will mention it as being "above" current tree
    // we will find the index of the tree closest to current tree above it
    int up_ind = -1;
    // check for trees at same x-level but above the current tree
    for (int i = 0; i < (int)tree.size(); i++)
    {
        if (tree[curr].x == tree[i].x && tree[i].y > tree[curr].y)
        {                                       // if a tree is above curr,
            if(up_ind < 0)                      // if no tree has been found above curr yet,
            {
                up_ind = i;                     // then update up_ind
            }
            else{                               // if atleast one tree has been found above curr already,
                if(tree[up_ind].y > tree[i].y)  // if there is a tree in between that tree and curr,
                {
                    up_ind = i;                 // then update up_ind
                }
            }
        }
    }//up_ind is the direct above lying tree of the current tree
    int up_profit = 0;
    if (up_ind > 0) // if there exists a tree above the current tree
    {
        if (tree[curr].weight > tree[up_ind].weight && tree[curr].h > (tree[up_ind].y - tree[curr].y))
        {                                   // if the tree above will fall, given curr tree falls up,
            up_profit = tree[up_ind].value;         // add its value to profit
            up_vec.push_back(up_ind);               // and push that index into our up_vec vector
            int more_up_profit = profit_up(up_ind, up_vec); // domino effect
            up_profit = up_profit + more_up_profit;             // add domino profit to up_profit
            return up_profit;                                   // return up_profit
        }else return 0;
    }
    else return 0;
}
```

Profit_finder (int i)

Time Complexity : $O(k^2)$

- This function checks cutting in which direction produces the most amount of profit
- It calls the 4 functions – (profit_up, profit_right, profit_left, profit_down) and compares which direction results in greater profit.
- The vector d_tracker stores the falling trees in the most profitable direction, which is then copied into the vector tree.domino_list

```
void profit_finder(int i)
{
    tree[i].max_profit = 0;
    int time_till_it = abs(tree[i].x - x_curr) + abs(tree[i].y - y_curr) + tree[i].d;
    // make sure time to get to a tree is even viable
    if(time_till_it <= t)
    {
        vector<int> d_list;
        vector<int> d_tracker;
        // first assume up is best, then check if other dir^ns are better
        int uprofit = profit_up(i, d_list);
        tree[i].max_profit = uprofit;
        d_tracker = d_list;                          // copy constructor
        tree[i].dir = 1;
        int rprofit = profit_right(i, d_list);
        if(rprofit > tree[i].max_profit)
        {
            tree[i].max_profit = rprofit;
            d_tracker.clear();
            d_tracker = d_list;
            tree[i].dir = 2;
        }
        int dprofit = profit_down(i, d_list);
        if(dprofit > tree[i].max_profit)
        {
            tree[i].max_profit = dprofit;
            d_tracker.clear();
            d_tracker = d_list;
            tree[i].dir = 3;
        }
        int lprofit = profit_left(i, d_list);
        if(lprofit > tree[i].max_profit)
        {
            tree[i].max_profit = lprofit;
            d_tracker.clear();
            d_tracker = d_list;
            tree[i].dir = 4;
        }
        tree[i].domino_list = d_tracker;
    }
}
```

find_next()

Time Complexity : O(k)

- 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.

```
int find_next()
{
    float maximize_this = 0;        // this is the variable we want to
maximize here (profit/time left)
    int i_next = 0;                 // index of next tree
    for (int i=0; i<(int)tree.size(); i++) // O(k) times
    {
        int time_till_next = abs(tree[i].x - x_curr) + abs(tree[i].y -
y_curr) + tree[i].d;
        if (time_till_next <= t)    // if it is viable to cut tree
within time left,
        {
            //int check = 0;
            float qty= (float)tree[i].max_profit/time_till_next; // find
profit/time left
            if (qty >= maximize_this)                           // get
its max value
            {
                maximize_this = qty;
                i_next = i;                                     // keep
track of index of next
            }
        }
    }
    return i_next;
}
```

print_moving()

Time Complexity : O(n)

- 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).

```
void print_moving()
{
    // once we have current coords in x_curr, y_curr
    // and coords of next tree in x_next, y_next,
    // then: we can start printing moving upto that next tree
    if (x_next > x_curr)
    {
        // to move right until you reach x coords of next tree
        for (int i=0; i<(x_next - x_curr) && t>0; i++)
        {
            cout << "move right\n";
            t--;
        }
    }
    else{
        // to move left until you reach x coords of next tree
        for(int i=0; i<(x_curr - x_next) && t>0; i++)
        {
            cout << "move left\n";
            t--;
        }
    }
    if (y_next > y_curr)
    {
        // to move up until you reach y coords of next tree
        for (int i=0; i<(y_next - y_curr) && t>0; i++)
        {
            cout << "move up\n";
            t--;
        }
    }
    else{
        // to move down until you reach y coords of next tree
        for (int i=0; i<(y_curr - y_next) && t>0; i++)
        {
            cout << "move down\n";
            t--;
        }
    }
}
```

print_cutting()

Time Complexity : $O(nk^2)$

- 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

```
int print_cutting(int i, vector<int> &fall_list)
{
    int profit_temp=0;
    if(t >= tree[i].d)
    {
        if (tree[i].dir == 1)
        {
            cout << "cut up\n";
            profit_temp= profit_up(i, fall_list);
        }
        else if (tree[i].dir == 2)
        {
            cout << "cut right\n";
            profit_temp= profit_right(i, fall_list);
        }
        else if (tree[i].dir == 3)
        {
            cout << "cut down\n";
            profit_temp= profit_down(i, fall_list);
        }
        else if (tree[i].dir == 4)
        {
            cout << "cut left\n";
            profit_temp= profit_left(i, fall_list);
        }
        t = t - tree[i].d;

        for(int k=0; k < (int)tree.size(); k++)
        {
            if(tree[k].x == tree[i].x || tree[k].y == tree[i].y)
            {
                profit_finder(k);
            }
        }
    }
    return profit_temp;
}
```

Int main()

```
int main()
{
    TOTAL = 0;
    cin >> t >> n >> k;
    int i;
    for(i=0; i<k; i++)
    {
        // to access a vector position, it should be populated using push_back() first.
        // also, we use the structure constructor, since the vector elements are of type tree_gen
        tree.push_back(tree_gen());

        // take all inputs for 'i'th tree
        cin >> tree[i].x >> tree[i].y >> tree[i].h;
        cin >> tree[i].d >> tree[i].c >> tree[i].p;

        // calculate value and weight using given formulas
        tree[i].value = (tree[i].p * tree[i].h * tree[i].d);
        tree[i].weight = (tree[i].c * tree[i].h * tree[i].d);

        tree[i].dir = 1;
    }

    // we want to start with tree closest to origin
    int man_dist;           // to hold manhattan distance between tree of index 0 and origin
    man_dist = abs(tree[0].x - 0) + abs(tree[0].y - 0);
    int start_i = 0;
    for(i = 1; i < k; i++)                              // traverse through all trees
    {
        if((abs(tree[i].x - 0) + abs(tree[i].y - 0)) < man_dist)
        {
            man_dist = abs(tree[i].x - 0) + abs(tree[i].y - 0);
            start_i = i;                                // index of tree closest to origin
        }
    }

    if(tree[start_i].x + tree[start_i].y + tree[start_i].d <= t)
        {
            TOTAL += tree[start_i].value;
        }

        // so far, we found nearest tree to origin, we also found the direction of max profit for all
        // trees, and the profits in each direction for each. We haven't actually moved or cut any
    trees yet!
        for(i=0; i<(int)tree.size(); i++)
        {
            profit_finder(i);
        }

        while (t > 0)
        {
            if (tree.size() > 0)
            {
                int next_i;
                // for test case 12, using profit doesn't seem to work
                if(TOTAL != 11475)
                {
                    next_i = find_next();
                }
                else{
                    next_i = find_next_price();
                }

                vector<int> fallen;

                x_next= tree[next_i].x;
                y_next= tree[next_i].y;
                print_moving();

                print_cutting(next_i, fallen);


                vector<tree_gen>:: iterator ptr;
                ptr = tree.begin();
```

Time Complexity :-

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

=

$O(k^3 + tnk^2)$

```
sort(fallen.begin(), fallen.end());        // sort indices of fallen trees
            for (int j=0; j < (int)fallen.size(); j++) // remove fallen trees from all-info
vector
            {
                int temp = fallen[j];
                int x_temp = tree[temp-j].x;
                int y_temp = tree[temp-j].y;
                tree.erase(ptr + fallen[j] - j);

                for(i=0; i < (int)tree.size(); i++)
                {
                    if(tree[i].x == x_temp || tree[i].y == y_temp)
                    {
                        profit_finder(i);
                    }
                }
                ptr = tree.begin();
            }
            for (int j=0; j < (int)tree.size(); j++)
            {
                if (x_next == tree[j].x && y_next == tree[j].y)
                {
                    tree.erase(ptr + j);
                    break;
                }
            }
            x_curr = x_next;
            y_curr = y_next;
        }
        else{
            break;        // if no trees left, end the loop
        }
    }
}
```

Thank You

It has been a pleasure presenting this project