**Minimal and Maximal Reward Routes in a Directed Acyclic Graph**

**BY KAUSHIK. PIDUGU**

**16187629**

**Objective:-**

Designing greedy algorithm and dynamic programming algorithms for finding the maximum reward points and minimum rewards points. Analyzing the time complexities for greedy and dynamic approach for both maximum and minimum rewards and also analyzing/deciding which approach gives the optimal solution for maximum and minimum rewards points.

Source file, Executable file, Output result files, Input file has been zipped to this folder.

**Project Execution:**

gcc –Wall <code\_file.c> -o <exe.o> //Compiling

./<exe.o> <Input\_file> //Execution

1. **STRUCTURES USED IN THE PROJECT**

int m = 0, n = 0; //Two Global for M rows and N colums

int \*\*arr = NULL , \*\*Dir\_arr = NULL ;

arr -> array of pointers (2-D array to store the input Grid values)

Dir\_arr -> array of pointers (2-D array to summation cell (Dynamic approach))

arr = (int \*\*) malloc((m+1) \* sizeof(int \*));

Dir\_arr = (int \*\*) malloc((m+1) \* sizeof(int \*));

Dir\_arr1 = (int \*\*) malloc((m+1) \* sizeof(int \*));

for(i = 0; i <= m; i++) {

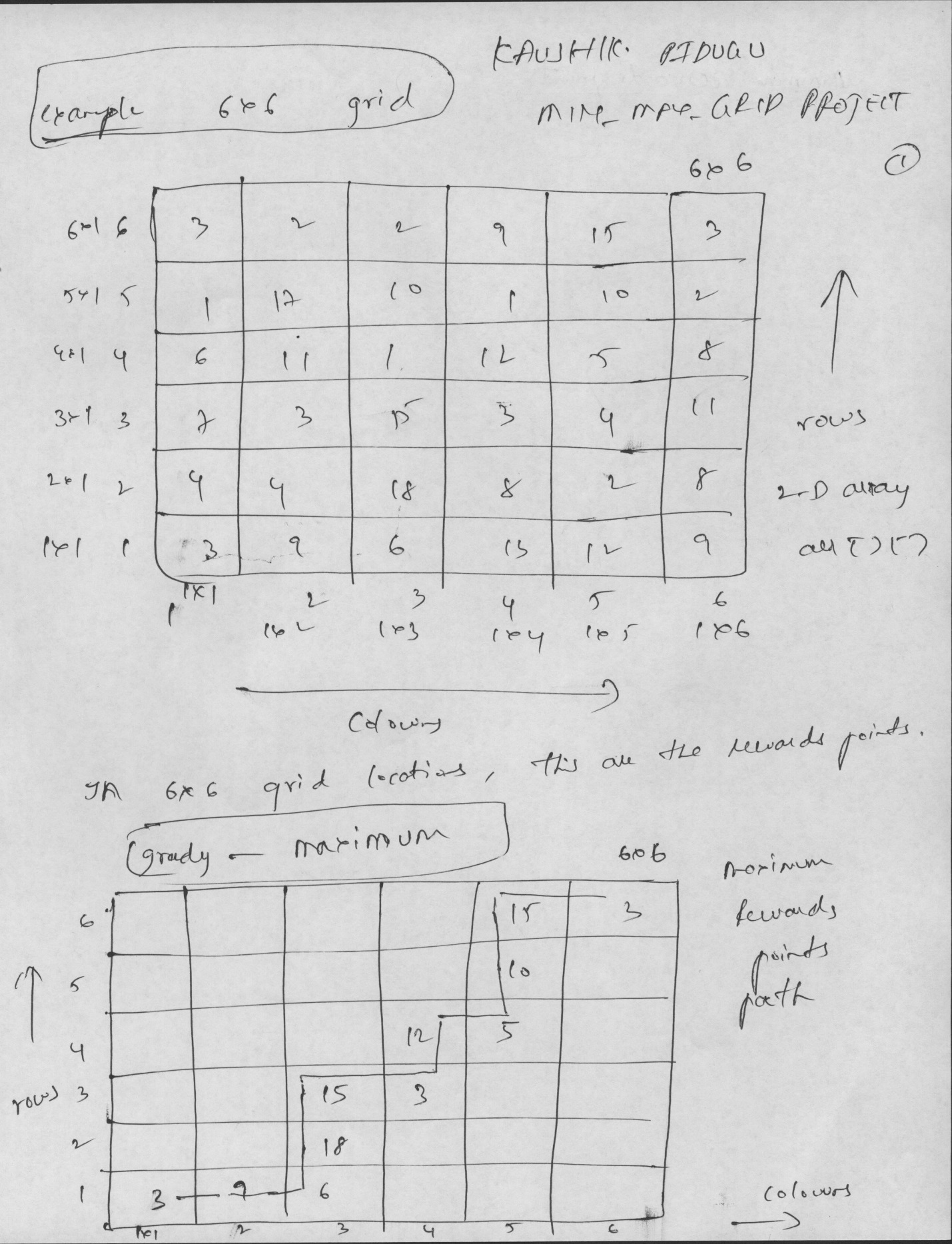
arr[i] = (int \*) malloc((n+1) \* sizeof(int));

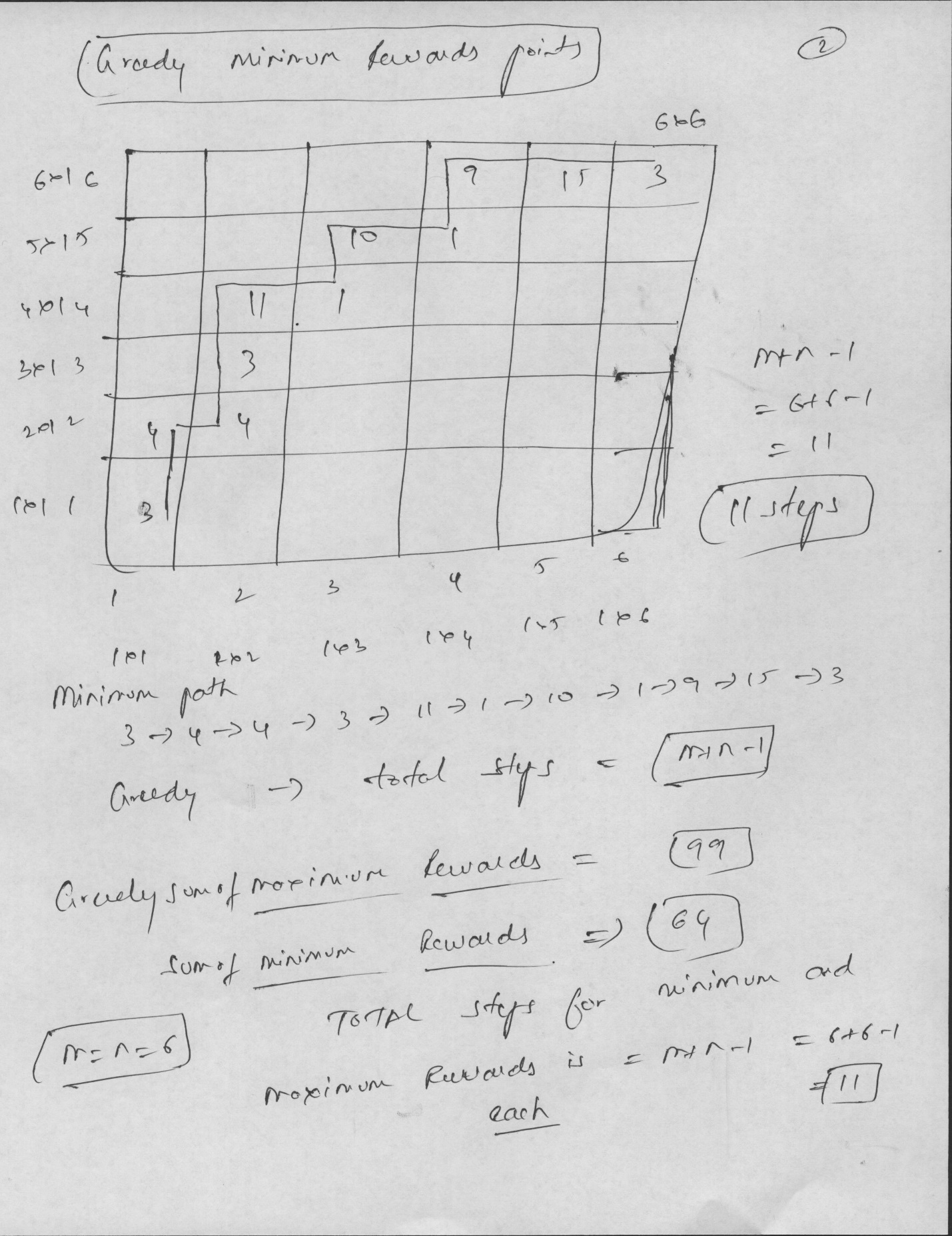
Dir\_arr[i] = (int \*) malloc((n+1) \* sizeof(int));

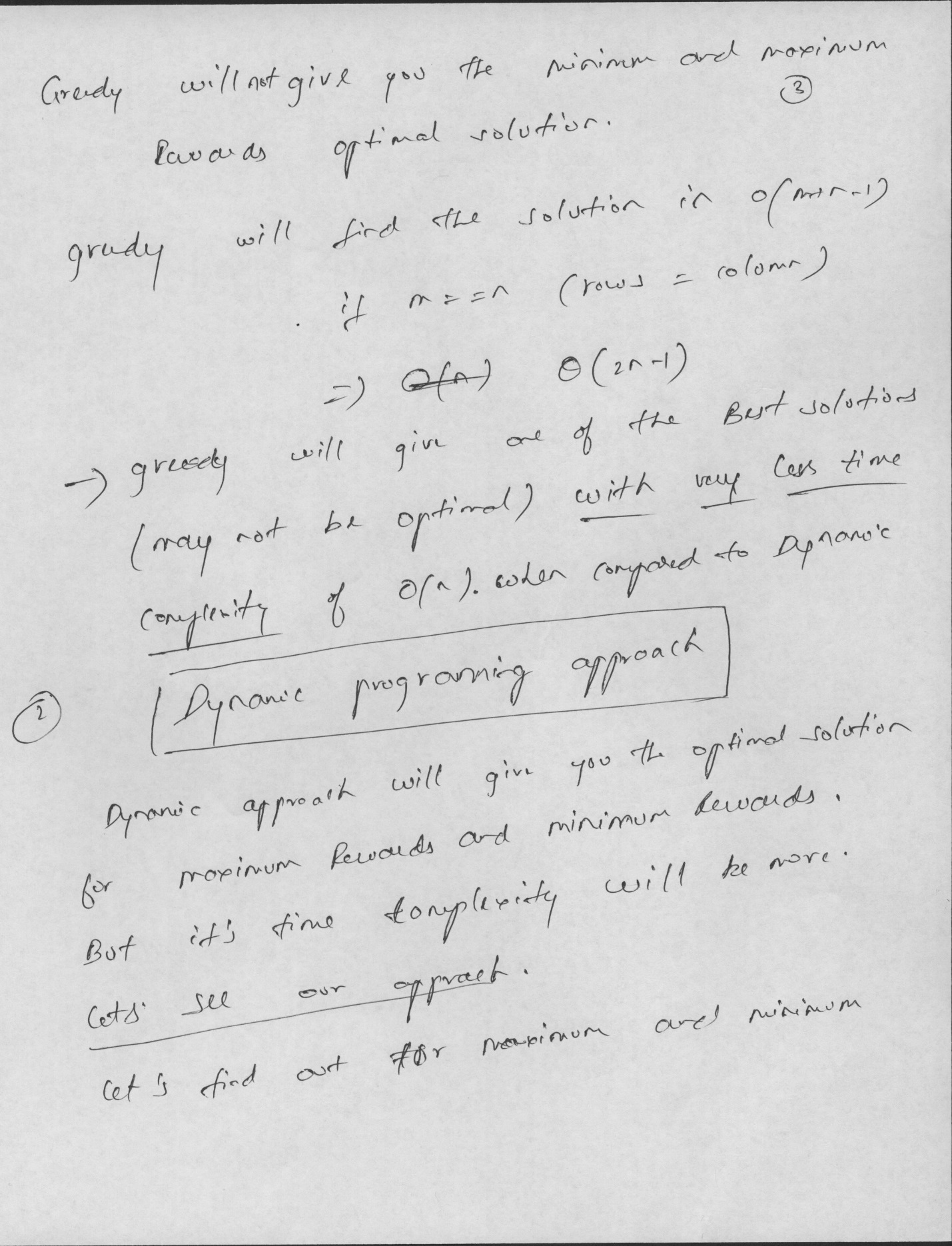
Dir\_arr1[i] = (int \*) malloc((n+1) \* sizeof(int));

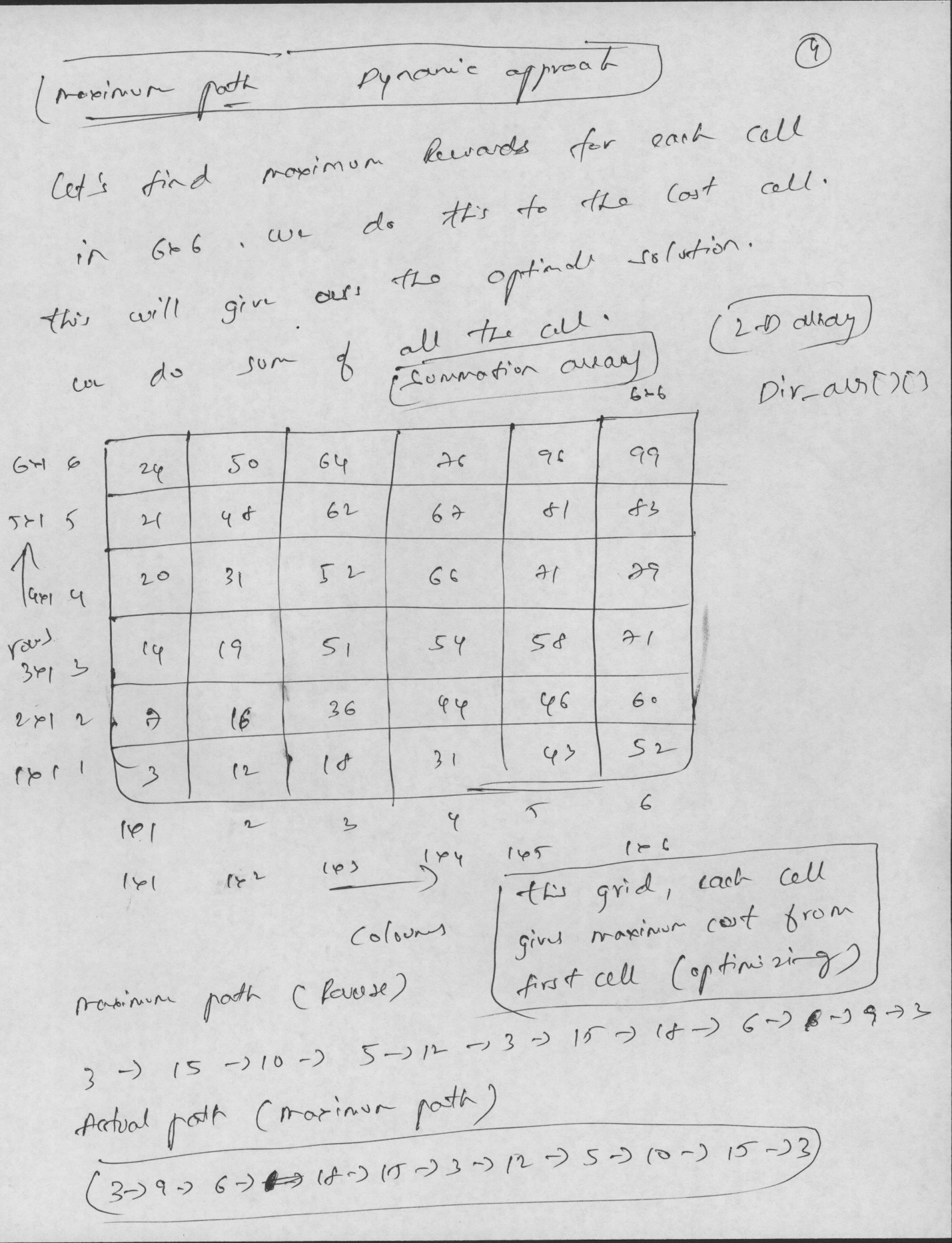
}

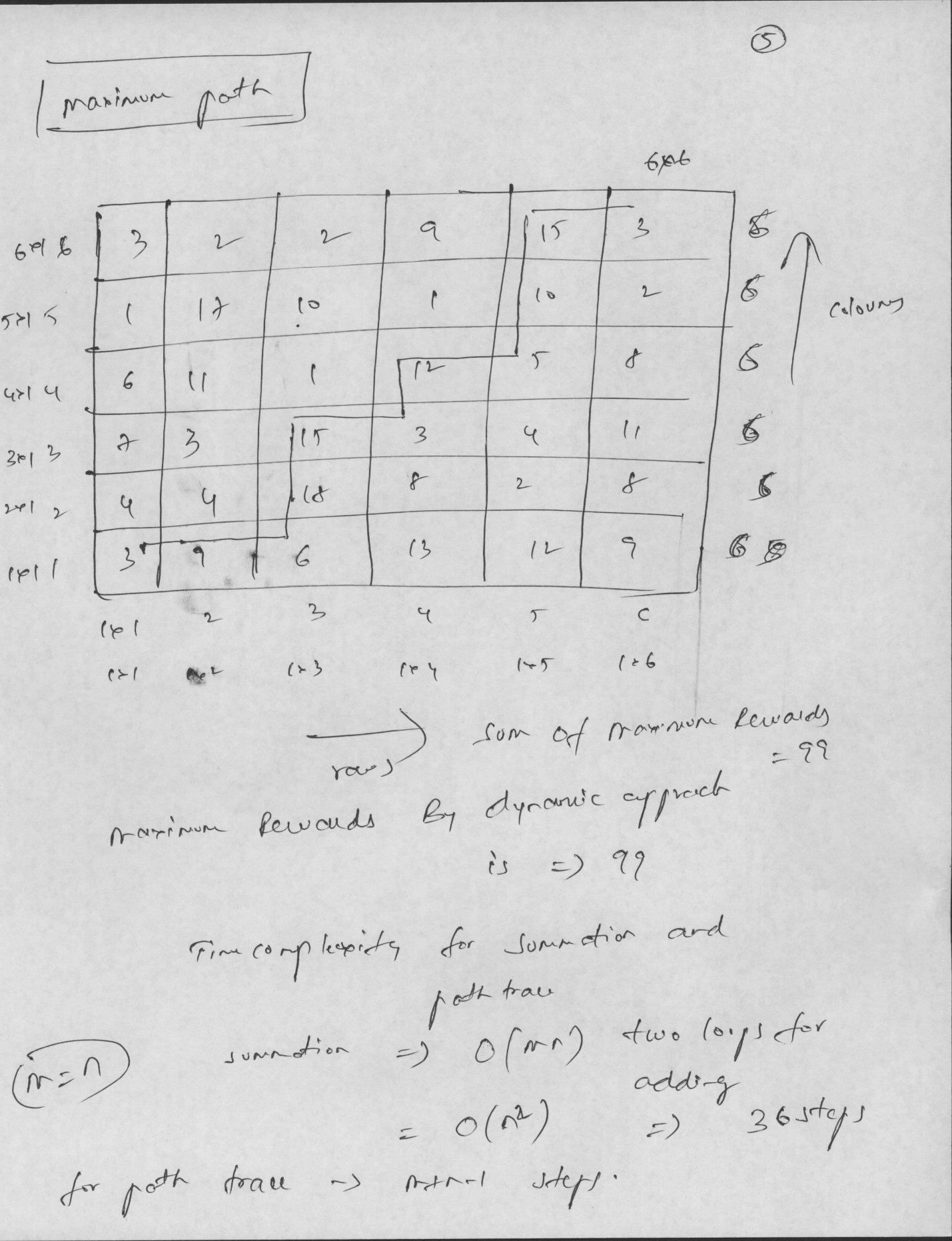
1. **BASIC DESIGN OF THE PROJECT WITH 6x6 GRID EXAMPLE AND ANALYSIS**

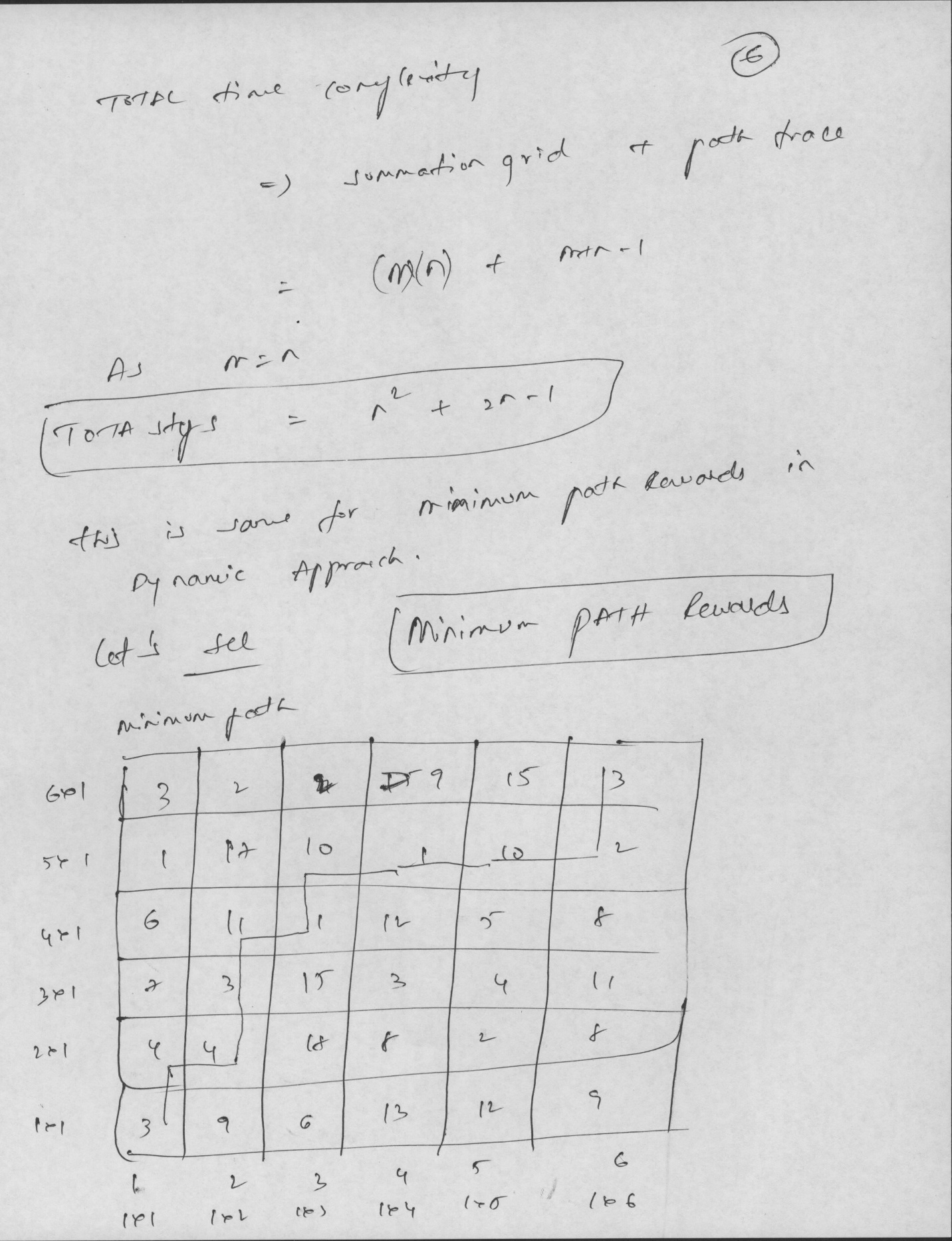




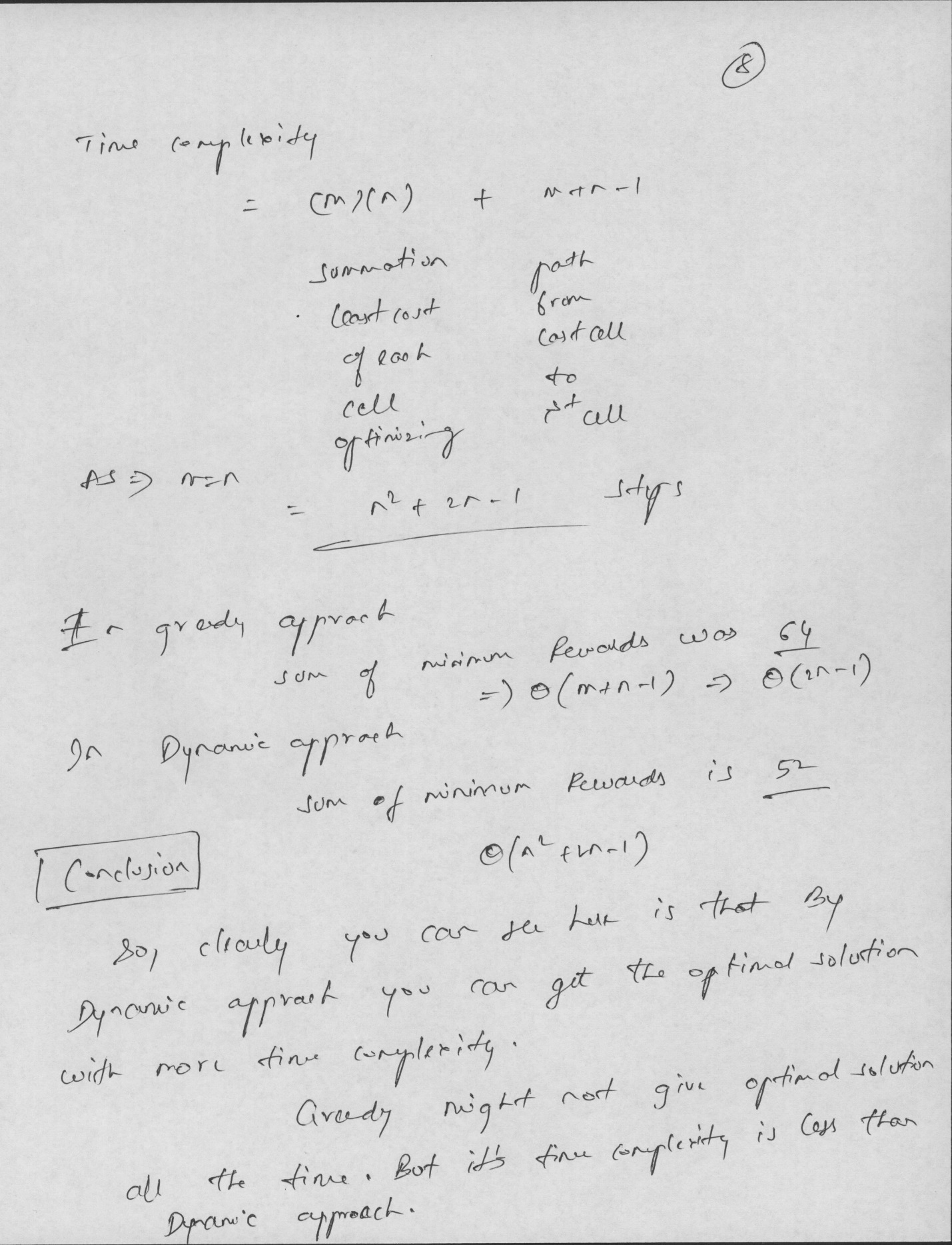




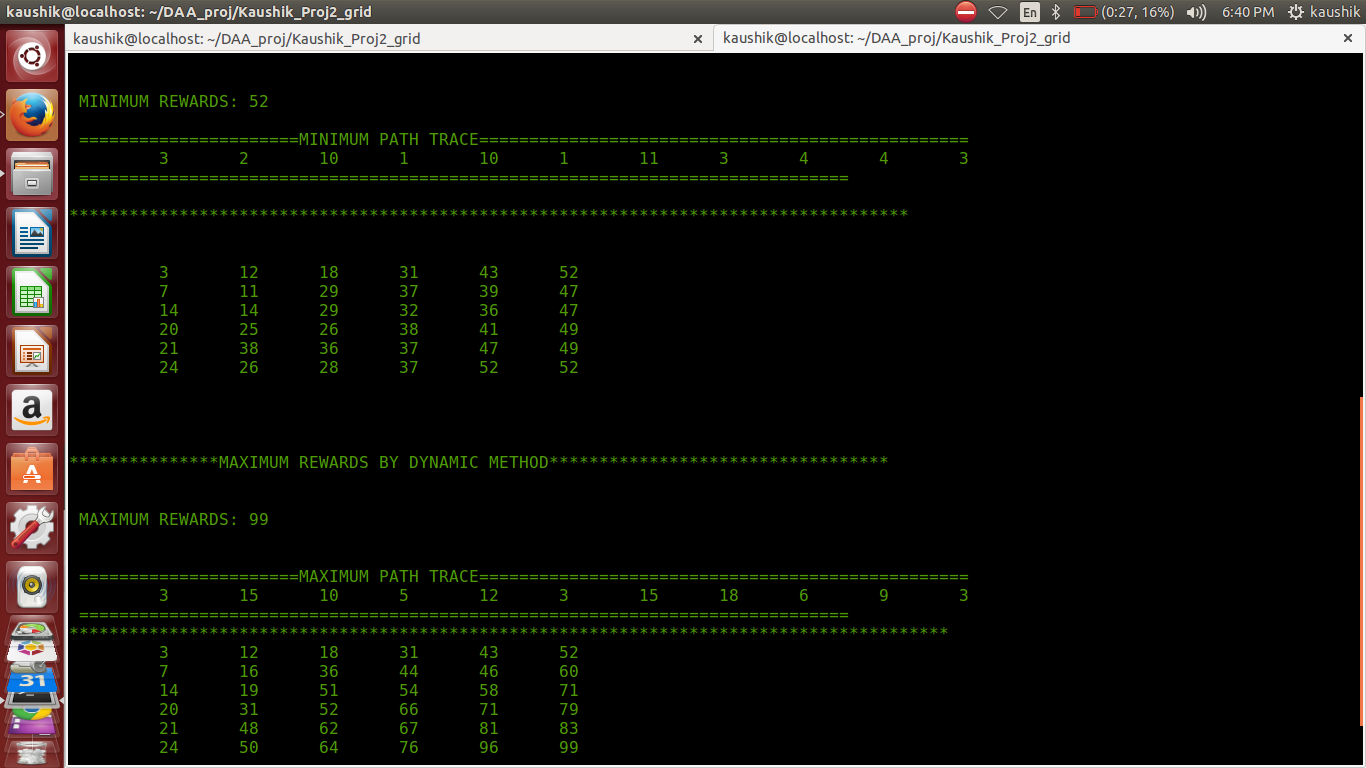








**SNAPSHOT:**



Read the input file and store and call the greedy and dynamic function to calculate the maximum and minimum rewards.

Code is attached in the zip file

Below will be my greedy and dynamic algorithm approach and finding the maximum and minimum rewards.

1. **GREEDY APPROACH**

**Algorithm greedy\_min\_path\_and\_max\_path**(int \*\*arr)

{

//int - Integer

Int iMin = 1, jMin = 1, iMax = 1, jMax = 1;

int i = 1, j = 1, k = 1;

int \*grd\_max = NULL, \*grd\_min = NULL;

unsigned long long sum\_max = 0; //8 Bytes integer

unsigned long long sum\_min = 0; //8 Bytes integer

int count = 0;

//Allocating dynamic memory 1-D array for maximum path trace

grd\_max = (int \*)malloc((m+n) \* sizeof(int));

//Allocating dynamic memory 1-D array for maximum path trace

grd\_min = (int \*)malloc((m+n) \* sizeof(int));

grd\_max[0] = arr[1][1];

grd\_min[0] = arr[1][1];

**// In general, Greedy approach is 1 to k elements then compare n-k elements in O(n)**

**// 'k' is one.**

**// only m+n-1 steps.**

**for (k = 1; k <= 1; k++) {**

if (arr[i][j+1] > arr[i+1][j]) {

grd\_max[k] = arr[i][j+1];

grd\_min[k] = arr[i+1][j];

sum\_max += (unsigned long long)arr[i][j] + arr[i][j+1];

sum\_min += (unsigned long long)arr[i][j] + arr[i+1][j];

jMax = j + 1;

iMin = i + 1;

} else {

grd\_min[k] = arr[i][j+1];

grd\_max[k] = arr[i+1][j];

sum\_max += (unsigned long long)arr[i][j] + arr[i+1][j];

sum\_min += (unsigned long long)arr[i][j] + arr[i][j+1];

iMax = i + 1;

jMin = j + 1;

}

}

k = 2;

i = iMax;

j = jMax;

**// FOR MAXMIMUM PATH**

while (k < m+n)

{

if (i == m && j == n) {

break;

}

if (i == m) {

grd\_max[k] = arr[i][j+1];

sum\_max += (unsigned long long)arr[i][j+1];

j = j + 1;

++k;

continue;

}

if (j == n) {

grd\_max[k] = arr[i+1][j];

sum\_max += (unsigned long long)arr[i+1][j];

i = i + 1;

++k;

continue;

}

if (arr[i][j+1] > arr[i+1][j]) {

grd\_max[k] = arr[i][j+1];

sum\_max += (unsigned long long)arr[i][j+1];

j = j + 1;

} else {

grd\_max[k] = arr[i+1][j];

sum\_max += (unsigned long long)arr[i+1][j];

i = i + 1;

}

++k;

}

**// FOR MINIMUM PATH**

k = 2;

i = iMin;

j = jMin;

while (k < m+n)

{

if (i == m && j == n) {

break;

}

if (i == m) {

grd\_min[k] = arr[i][j+1];

sum\_min += (unsigned long long)arr[i][j+1];

j = j + 1;

++k;

continue;

}

if (j == n) {

grd\_min[k] = arr[i+1][j];

sum\_min += (unsigned long long)arr[i+1][j];

i = i + 1;

++k;

continue;

}

if (arr[i][j+1] < arr[i+1][j]) {

grd\_min[k] = arr[i][j+1];

sum\_min += (unsigned long long)arr[i][j+1];

j = j + 1;

} else {

grd\_min[k] = arr[i+1][j];

sum\_min += (unsigned long long)arr[i+1][j];

i = i + 1;

}

++k;

}

Print the trace based on grd\_max and grd\_min and Maximum rewards and Minumum rewards as sum\_max and sum\_min respectively.

-------------------------------------------------------------------------------------------------------------------------------

**BEST TIME COMPLEXITY = AVERAGE TIME COMPLEXITY = WORST TIME COMPLEXITY**

**TGreedy(maximum rewards) = TGreedy(minimum rewards)**

**TG(W) -> Worst Time complexity Greedy approach**

**Worst Time complexity :**

**TG(W) = k + (m+n-k– 1) (**

**-------------------------------------------------------------------------------------------------------------**

1. **DYNAMIC APPROACH MAXIMUM REWARDS**

**Algorithm Dynamic\_max\_path(int \*\*arr, int \*\*Dir\_arr)**

**{**

int i = 0, j = 0;

**for (i = 1; i <= m; i++) {**

**for (j = 1; j <= n; j++) {**

if ((i == 1) && (j == 1)) {

Dir\_arr[i][j] = arr[i][j];

} else if (i == 1) {

Dir\_arr[i][j] = Dir\_arr[i][j - 1] + arr[i][j];

} else if (j == 1) {

Dir\_arr[i][j] = Dir\_arr[i - 1][j] + arr[i][j];

} else if (Dir\_arr[i][j - 1] > Dir\_arr[i - 1][j]) {

//right cell selected

// selecting MAXIMUM path

Dir\_arr[i][j] = Dir\_arr[i][j - 1] + arr[i][j];

} else {

// upper cell selected

Dir\_arr[i][j] = Dir\_arr[i - 1][j] + arr[i][j];// upper cell selected

}

**} // END OF 2nd FOR LOOP**

**} // End OF 1st FOR LOOP**

i = m;

j = n;

**// MAXIMUM PATH TRACE**

printf("\t %d", arr[i][j]);

**while (i > 1 || j > 1) {**

if (i == 1) {

// j will be less than n j < n

printf("\t %d", arr[i][j-1]);

--j; // COMING FROM LEFT

// Guess we are in first row

} else if (j == 1) { // j == 1

// COMING FROM DOWN

printf("\t %d", arr[i-1][j]);

--i;

// Guess we are in first column

} else if (Dir\_arr[i][j-1] > Dir\_arr[i-1][j]) {

// minimum path

// 1. coming from left -> 1

// 2. coming from down -> -1

printf("\t %d", arr[i][j-1]); // COMING FROM LEFT BUDDY

--j;

} else {

printf("\t %d", arr[i-1][j]); // COMING FROM LEFT BUDDY

--i;

**} END OF WHILE LOOP FOR PATH TRACE**

**} // END OF ALGORITHM**

----------------------------------------------------------------------

**BEST TIME COMPLEXITY : ( INPUT IS REVERSED ORDER, Only one Stack)**

**Tsmallest(nB) = 1**

**//if there is only one stack and input is sorted list of ‘N’ numbers. For every sequence of ‘N’ number, this function is called only once.**

**Worst Time complexity : (INPUT IS SORTED ORDER so, N number , N stacks)**

**Tsmallest(nW) = N (N times loop to get to tail) + N (N times loop to back trace to**

**first node)**

* **N + N**
* **2N**

**------------------------------------------------------------------------------------------------------------**

1. **DYNAMIC APPROACH MINIMUM REWARDS**

**Algorithm Dynamic\_min\_path** (int \*\*arr, int \*\*Dir\_arr)

**{**

int i = 0, j = 0;

//

**for (i = 1; i <= m; i++) {**

**for (j = 1; j <= n; j++) {**

if ((i == 1) && (j == 1)) {

Dir\_arr[i][j] = arr[i][j];

} else if (i == 1) {

Dir\_arr[i][j] = Dir\_arr[i][j - 1] + arr[i][j];

} else if (j == 1) {

Dir\_arr[i][j] = Dir\_arr[i - 1][j] + arr[i][j];

} else if (Dir\_arr[i][j - 1] < Dir\_arr[i - 1][j]) {

//right cell selected

// selecting least path

Dir\_arr[i][j] = Dir\_arr[i][j - 1] + arr[i][j];

} else {

// upper cell selected

Dir\_arr[i][j] = Dir\_arr[i-1][j] + arr[i][j];// upper cell selected

}

**} // END OF 2nd FOR LOOP**

**} // End OF 1st FOR LOOP**

printf("\n\n **MINIMUM REWARDS**: %d\n", **Dir\_arr[m][n]);**

i = m;

j = n;

**// MINIMUM REWARDS PATH TRACE**

printf("\t %d", arr[i][j]);

**while (i > 1 || j > 1) {**

if (i == 1) {

// j will be less than n j < n

printf("\t %d", arr[i][j-1]);

--j; // COMING FROM LEFT

// Guess we are in first row

} else if (j == 1) { // j == 1

// COMING FROM DOWN

printf("\t %d", arr[i-1][j]);

--i;

// Guess we are in first column

} else if (Dir\_arr[i][j-1] < Dir\_arr[i-1][j]) {

// minimum path

// 1. coming from left -> 1

// 2. coming from down -> -1

printf("\t %d", arr[i][j-1]); // COMING FROM LEFT BUDDY

--j;

} else {

printf("\t %d", arr[i-1][j]); // COMING FROM LEFT BUDDY

--i;

}

**} // END OF WHILE LOOP**

**} // End of Algorithm**

**BEST TIME COMPLEXITY : ( Input : REVERSE ORDER N numbers, Only One stack)**

**Only one stack**

**TLargest(nB) = 0 + N-1 (while loop goes through N-1 times)**

* **N – 1**

**Worst Time complexity : (Input : SORTED ORDER so, N number , N stacks)**

**Or N/2 Stacks and each stack has 2 numbers**

**TLargest(nW) = N -1 + 0 + Sigma (N =1 to N-1) {2}**

* **N-1 + 2N -2**
* **3N - 3**

**-------------------------------------------------------------------------------------------------------------**

1. **TOTAL TIME COMPLEXITY OF THE WHOLE PROGRAM**

-------------------------------------------------------------------------------------------------------------------------------

**BEST TIME COMPLEXITY OF THE WHOLE PROGRAM :**

**T(nB) = N -1 + 1 + N -1 + 1**

* **2N (for one sequence, there are ‘N’ numbers)**

**Worst Time complexity of the program**

**T(nW) = N^2/2 – N/2 + 2N + 3N -3 + N – 1**

* **N^2/2 + 11n/2 – 4 (For one Sequence)**

**-------------------------------------------------------------------------------------------------------------**

1. **PROJECT LIMITATION**
2. As, testing failed for non decreasing numbers, only strictly increasing numbers are taken into consideration.
3. Testing failed for problem 4 to find the count of number of subsequences, still working on it. Included the code for that too but commentated that out.

**10) PROJECT IMPROVEMENTS**

a) Code can further optimized.

b) Instead of writing 4 functions for different problems, Only one function can be used to solve 4 problems. This can improve time complexity and space complexity.

c) No Memory being freed here. (Add this later).

**SREENSHOT FROM complete\_output\_file**

**END**