Assignment #2

20190741 김 지수

Q1) N*M minimum tile calculator

Note: The code to solve the problem 1 used <math.h> standard c library. As a result, when te sting the code using putty in linux environment gcc HW2_20190741_1.c -o result -lm should b e used to link <math.h> library.

Q1-1. Code Description with Flow chart

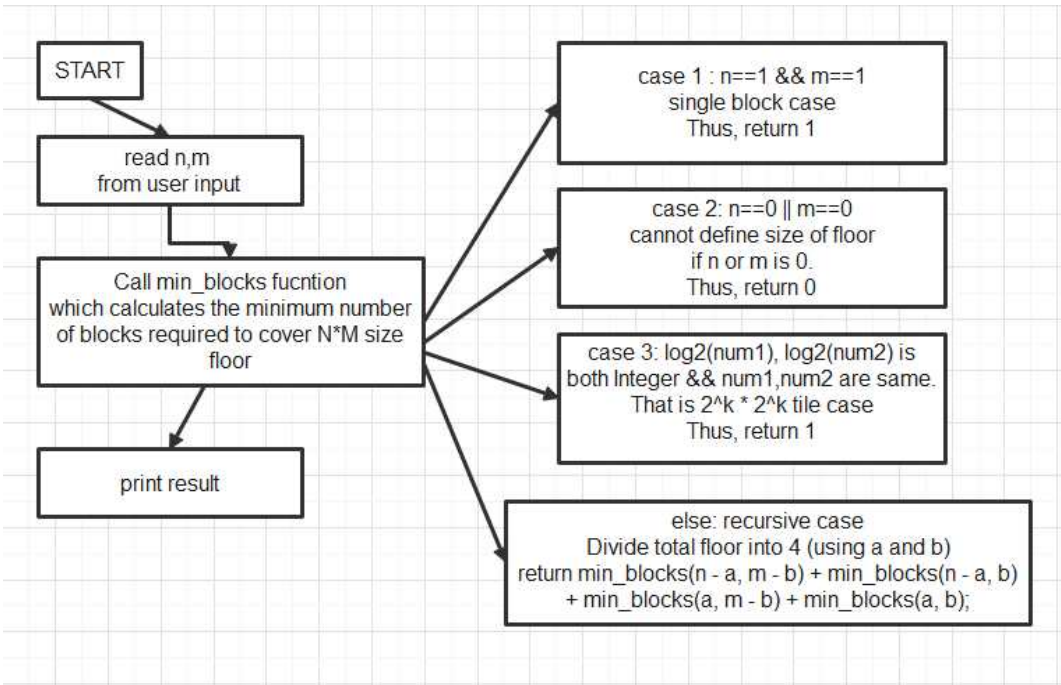
The flow chart of the Q1 is as follows. When the program starts, it read from user input th e size of the floor, N*M. Next, it calls min_blocks() function which calculates the minimum nu mber of blocks required to cover the received N*M size floor. In the min_blocks function ther e are 4 cases for the input number N and M. Three cases are particular cases that could dir ectly return the results, and the other is recursive case which calls the function itself recursi vely.

In case 1, if n==1 && m==1, it means the input case is a single block case. We needs one s ingle tile to cover a 1*1 block. Thus, return 1.

In case 2, if n==0 || m==0, it is the handling of exception case. The size of the floor cannot be defined as 0*0, 0*100(line). Thus, return 0.(No tile required to cover 0 size.)

In case 3, if log2(n), log2(m) is both integer and n equals to m, it is the 2^k *2^k tile case. As the question described that no matter the size of the tile it is considered as one tile, in th is case the program returns 1.

In recursive case, which is implemented in else statement, covers the case that the input N *M floor should be divided into several tiles to calculate the number of required tiles. So, in t his case, the program finds the maximum size of 2^k*2^k tile that fit into the given N*M floo r and divides the floor into 4 parts using the maximum size tile. As a result it returns the 4 recursive min_blocks function call. Finally, the program prints the result.



<Flow chart, Q1>

Q1-2. Main Algorithms with Pseudo code

The main algorithm is in the min_blocks function which calculates the minimum number of tiles needed to fill the given n*m size floor. As described above in the flow chart part, case 1 through case 3 are particular cases that could directly return the result while the other case, the recursive case, divides the original floor into 4 parts by recursive function call. As case 1 through case 3 were described specifically at the above part, the recursive part will be explai ned in below.

1. Find the maximum size of 2^k*2^k tile that fit into the given n*m size floor

In this recursive case, the given n*m floor is not the case that can be covered by a single til e. So, we first find the maximum size tile that fit into the given floor as we have to minimize the number of tiles used to cover the floor. We propose the maximum size tile that fit into th e given floor is 2^k*2^k size.

2. Divide the floor into 4 parts using the maximum size tile, 2^k*2^k.

We have to fill the empty floor that is not covered by the maximum size tile. Thus, we calcul ate the minimum number of tiles to cover the part of the floor that is not covered by recursi ve function call,

```
return min_blocks(n - a, m - b) + min_blocks(n - a, b) + min_blocks(a, m - b) + min_blocks(a, b);
```

For easier understand, the below chart is the replication of the above return statement. The total box size is n*m. a and b is each side of the maximum size tile, 2^k.

min_blocks(a, b)	min_blocks(n - a, b)
min_blocks(a, m - b)	min_blocks(n - a, m - b)

By calculating 4 divided parts including the maximum size tile that fit into given n*m floor, we can finally calculate the final result using recursive function call. The pseudocode of Q1 is as follows.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int isInteger(double val);
int min_blocks(int n, int m);

int main() {
    receive n*m size of the floor
    num_blocks = min_blocks(n, m) //function call to calculate the result
    print result
    return 0;
}

int isInteger(double val) {
    if (the input val is Integer except 0) return 1;
    else return 0;
}

int min_blocks(int n, int m) {
    if (1*1 size block case) return 1;
    else if (exceptional case which is not the size of a block) return 0;
    else if (2^k * 2^k size block case) return 1;
    else {
        Find the maximum size of 2^k*2^k tile that fit into the given n*m size floor
        Divide the floor into 4 parts using the maximum size tile, 2^k*2^k.
        a = 2^k and b = 2^k which is each side of the maximum size tile.
        Then return the 4 recusive min_blocks function using a and b.
        i.e., return min_blocks(n - a, m - b) + min_blocks(n - a, b) + min_blocks(a, m - b) + min_blocks(a, b);
    }
}
```

<Pseudocode, Q1>

Q1-3. Test cases simulated

Input	Output	Input	Output
5 6	9	10 10	10
11 11	31	1 1	1
0 0	0	0 1	0
1 0	0	100 120	54
1000 1000	358	128 581	166
6 5	9	548 999	1269
389 888	1209	987 654	1986
19 82	136	777 999	2529

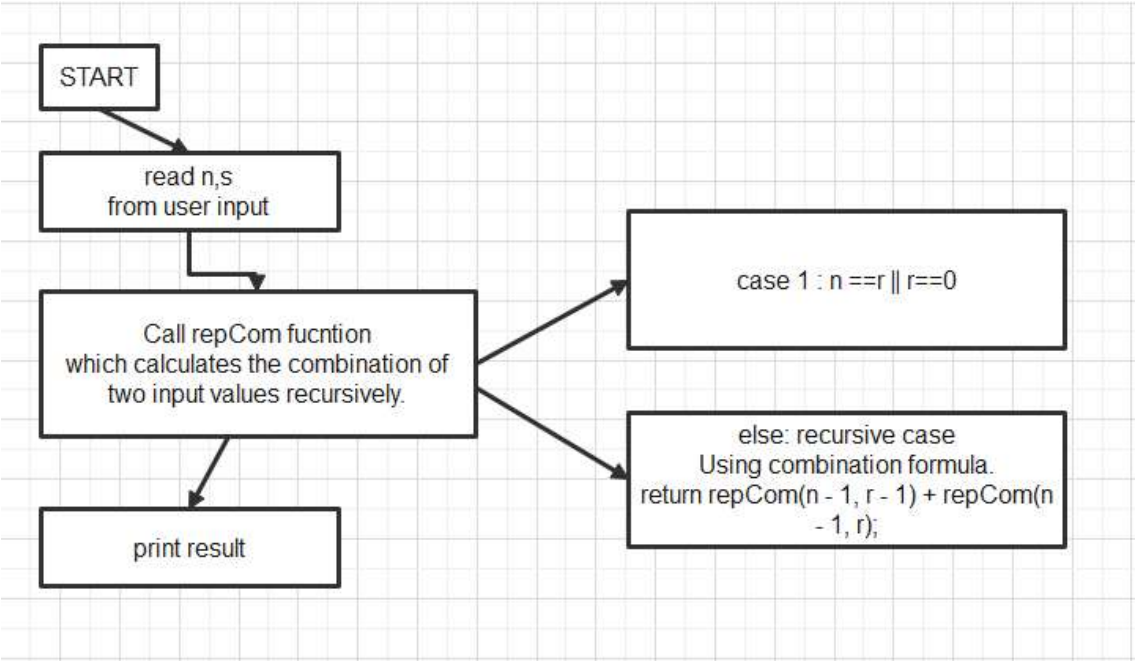
<Test results, Q1>

As simulating the random test cases could make the program more reliable by finding the corner cases, following cases are tested. Random test cases are generated as to test the corner case as well as random cases. As a result, there is no program error or incorrect result found for several test cases by comparing the test results with other students.

Q2)

Q2-1. Code Description with Flow chart

The flow chart of the Q2 is as follows. When the program starts, it read from user input th e n and s. Next, it calls repCom function which calculates the combination of two input value s recursively. In the min_blocks function, there are 2 cases. Case 1 is a particular case that could directly return the result which is return 1, and the other is recursive case which calls the function itself recursively.



<Flow chart, Q2>

Q2-2. Main Algorithms with Pseudo code

In Question 2, the main algorithms are the two mathematical formulas in regard to combina tion. The two formulas are as follows.

- $_nH_k = {}_{n+k-1}C_k$
- $_nC_r = {}_{n-1}C_{r-1} + {}_{n-1}C_r$

The first formula is used when we calculate combination with repetition by transforming the $_nH_k$ into $_{n+k-1}C_k$ which is a combination case. In the pseudocode, the program directly calls the repCom function to print the results with the transformed number, which are n+s-1 and s by using the first formula.

Next, the second formula is related to transforming the combination formula into recursive one using the Pascal's triangle. In this case, if (n is equal to r) or (r is 0) in the given $_nC_r$, t he program returns 1 as $_nC_n$ and $_kC_0$ is always 1. On the other hand, the remaining cases ex cept the upper one, returns the recursive function call statement by using the second formul a. The pseudocode of Q2 is as follows.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int repCom(int num, int sum);

int main() {
    receive n, s by standard input
    if (n == 0) {
        exception handling. Equation that has no variable cannot be defined in this question.
    }
    printf("result: %d\n", repCom(n+s-1, s));
    return 0;
}

int repCom(int num, int sum) {
    int n = num, r = sum;
    if (nCr is 1) return 1;
    else {
        recursive case using the combination formula.
        return repCom(n - 1, r - 1) + repCom(n - 1, r);
    }
}
```

<Pseudocode, Q2>

Q2-3. Test cases simulated

Input	Output	Input	Output
4 2	10	10 10	92378
11 11	352716	12 12	1352078
13 13	5200300	14 14	20058300
15 15	77558760	10 0	1
1 0	1	0 10	Exception handling (No variable)
14 7	77520	15 4	3060
9 1	9	11 9	92378
2 15	16	6 15	15504

As simulating the random test cases could make the program more reliable by finding the co rner cases, following cases are tested. Random test cases are generated as to test the corner case as well as random cases. As a result, there is no program error or incorrect result foun d for several test cases by comparing the test results with combination with repetition calcula tors.