Objective

- Programs, which let you think recursive.

Task – 1:

```
//Define a recursive function int sumUpTo(int n) that computes the sum of the first n
integers without using a loop.
//E.g. sumUpTo(3) is 1+2+3 which is 6.

int sumUpTo(int n)
{
    // base case, nothing to add
    if (        )
        return    ;
    else
    {
        // make a recursive call to n-1
        int rec_result =                    ;
        // add the missing number
        int total =                  ;
        // return the total
        return        ;
    }
}

int main()
{
    int n;
    cin >> n;
    cout << sumUpTo(n);
}


}
```

Task – 2:

Write a recursive method sumDigits that has one integer parameter and returns the sum of the digits in the integer specified. Remember, your method should not use loops. For example, if the integer is 15121, then this method should return 10.

Task – 3:

Write a method printSquares that has an integer parameter n, and prints the squares of the integers from 1 to n, separated by commas. It should print the squares of the odd integers in descending order first and then following with the squares of the even integers in ascending order. It does not print a newline character.

For example, printSquares(4) should print 9, 1, 4, 16 printSquares(1) should print 1 printSquares(7) should print 49, 25, 9, 1, 4, 16, 36

You may NOT use helper methods to solve this problem; write a single method.

Task – 4:

Write a recursive function evenDigits that accepts an integer and returns a new number containing only the even digits, in the same order. If there are no even digits, return 0.

– Example: evenDigits(8342116) returns 8426

© Fareed Ul Hassan Balg

Scanned with CamScanner

Task – 5:

Define a function void pronounce(int n) that prints out (to cout) the English spelling of n. For example,

- pronounce(25); prints twenty-five
- pronounce(103); prints one hundred three
- pronounce(2014); prints two thousand fourteen
- pronounce(999999); prints nine hundred ninety-nine thousand nine hundred ninety-nine

```
void pronounce(int n)
{
    // base cases
    if (n<20)
    {
        const char* units[] = {"zero",
"one", "two", "three", "four",
            "five", "six", "seven",
"eight", "nine",
            "ten", "eleven", "twelve",
"thirteen",
            "fourteen", "fifteen",
"sixteen", "seventeen",
            "eighteen", "nineteen"};
        cout << units[n];
    }
    // more base cases
    else if (n%10==0 && n<100)
    {
        const char* tenfolds[] = {"",
"ten", "twenty", "thirty", "forty",
            "fifty", "sixty", "seventy",
"eighty", "ninety"};
        cout << tenfolds[n/10];
    }
    else if (n<100)
    {
        // pronounce the tens place
            pronounce(------------);
        // pronounce the ones place
        if (n%10 != 0)
        {
            cout << "-";
            pronounce(------------);
        }
    }
    // exact multiple of 100
    else if (n%100==0 && n<1000)
    {
        ----------------
        cout << " hundred";
    }
    else if (n<1000)
    {
        pronounce(n - (n%100));
        if (n%100 != 0)
        {
            cout << " ";
            pronounce(n%100);
        }
    }
    else if (n<1000000)
    {
        // how many thousands?
        ----------------
        cout << " thousand";
        if (n%1000 != 0)
        {
            cout << " ";
            pronounce(------------);
        }
    }
    else if (n<1000000000)
    {
        pronounce(n/1000000);
        cout << " million";
        if (n%(int)1000000 != 0)
        {
            cout << " ";
            pronounce(n%(int)1000000);
        }
    }
    else
    { // INT_MAX is less than a trillion
        pronounce(n/1000000000);
        cout << " billion";
        if (n%(int)1000000000 != 0)
        {
            cout << " ";

pronounce(n%(int)1000000000);
        }
    }
}

int main()
{
    int n;
    cin >> n;
    pronounce(n);
}
```


© Fareed Ul Hassan Balg

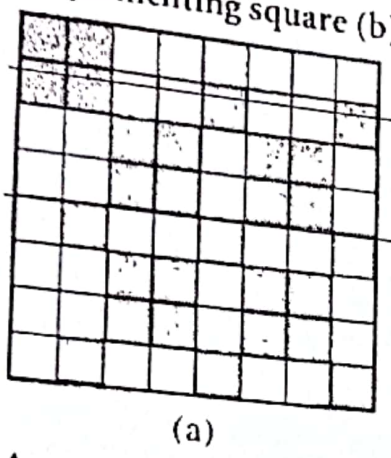
Scanned with CamScanner

Issue Date: 26-Aug-2022

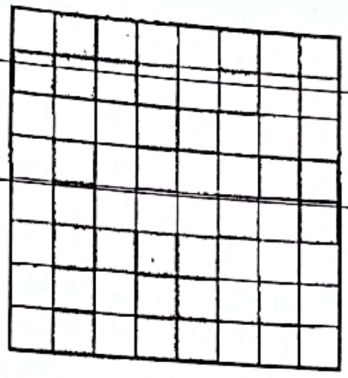
Task – 6:
Write a recursive function reverseLines that accepts a file input stream and prints the lines of that file in reverse order.

Task – 7:

(a–b) Two $n \times n$ squares of black and white cells and (c) an $(n + 2) \times (n + 2)$ array implementing square (b).



(a)



(b)

b b b b b b b b b b
b w b b w w b w w b
b b b b b w b w w b
b w w w b b w w w b
b w b w b w w b b b
b w b w w w b w b b
b w b b b b w w w b
b w b w b b w w w b
b w b w b b w w w b
b b b b b b b b b b

(c)

An $n \times n$ square consists of black and white cells arranged in a certain way. The problem is to determine the number of white areas and the number of white cells in each area. For example, a regular 8×8 chessboard has 32 one-cell white areas; the square in Figure 5.22a consists of ten areas, two of them of ten cells, and eight of two cells; the square in Figure 5.22b has five white areas of one, three, twenty-one, ten, and two cells.

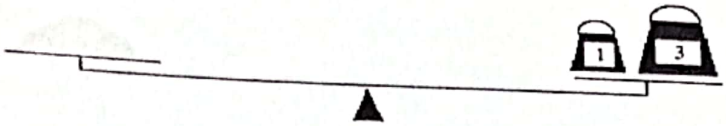
Write a program that, for a given $n \times n$ square, outputs the number of white areas and their sizes. Use an $(n + 2) \times (n + 2)$ array with properly marked cells. Two additional rows and columns constitute a frame of black cells surrounding the entered square to simplify your implementation. For instance, the square in Figure 5.22b is stored as the square in Figure 5.22c.

Traverse the square row by row and, for the first unvisited cell encountered, invoke a function that processes one area. The secret is in using four recursive calls in this function for each unvisited white cell and marking it with a special symbol as visited (counted).

Task – 8:
In Dickens's time, merchants measured many commodities using weights and a two-pan balance—a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:

Scanned with CamScanner



It is somewhat more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:

Write a recursive function

bool isMeasurable (int target, int * weights, int N)

that determines whether it is possible to measure out the desired target amount with a given set of weights. The available weights are stored in the integer array 'weights'.

Task – 9:

You're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: via three small strides or one small stride followed by one large stride or one large followed by one small. A staircase of four steps can be climbed in five different ways (enumerating them is an exercise left to reader :-).

Write the recursive function *int countWays(int numStairs)* that takes a positive *numStairs* value and returns the number of different ways to climb a staircase of that height taking strides of one or two stairs at a time.

Here's a hint about the recursive structure of the problem: consider the options you have at each stair. You must either take a small stride or a large stride; either will take you closer to the goal and therefore represents a simpler instance of the same problem that can be handled recursively. What is the simplest possible situation and how is it handled?

int CountWays(int numStairs)

Task – 10:

The subset sum problem is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to that target number. For example, given the numbers {3, 7, 1, 8, -3} and the target sum 4, the subset {3, 1} sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this function is

int canMakeSum(int array, int targetSum)*

Task – 11: **All Squares**

Geometrically, any square has a unique, well-defined centre point. On a grid this is only true if the sides of the square are an odd number of points long. Since any odd number can be written in the form 2k+1, we can characterize any such square by specifying k, that is we can say that a square whose sides are of length 2k+1 has size k. Now define a pattern of squares as follows.

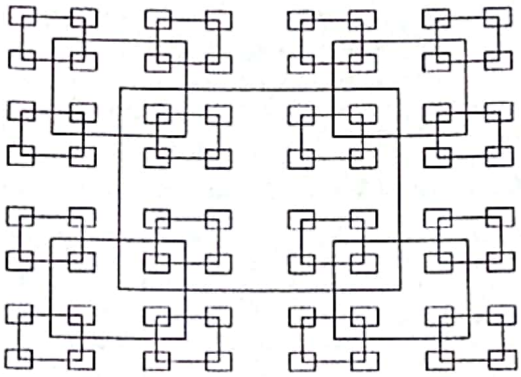
1. The largest square is of size k (that is sides are of length 2k+1) and is centred in a grid of size 1024 (that is the grid sides are of length 2049).

Department of Software Engineering (DSE), PU, Lahore, Pakistan.
© Fareed Ul Hassan Balg

Scanned with CamScanner

2. The smallest permissible square is of size 1 and the largest is of size 512, thus $1 \leq 512 \leq k$.
3. All squares of size $k > 1$ have a square of size k div 2 centred on each of their 4 corners. (Div implies integer division, thus 9 div 2 = 4).
4. The top left corner of the screen has coordinates (0,0), the bottom right has coordinates (2048, 2048).

Hence, given a value of k, we can draw a unique pattern of squares according to the above rules. Furthermore any point on the screen will be surrounded by zero or more squares. (If the point is on the border of a square, it is considered to be surrounded by that square). Thus if the size of the largest square is given as 15, then the following pattern would be produced.



Write a program that will read in a value of k and the coordinates of a point, and will determine how many squares surround the point.

Input and Output

Input will consist of a series of lines. Each line will consist of a value of k and the coordinates of a point. The file will be terminated by a line consisting of three zeroes (0 0 0).

Output will consist of a series of lines, one for each line of the input. Each line will consist of the number of squares containing the specified point, right justified in a field of width 3.

Sample input

```
500 113 941
300 100 200
300 1024 1024
0 0 0
```

Sample output

```
5
0
1
```

Scanned with CamScanner

Task – 12:

This question is about a one-dimensional puzzle which you can think about as an array of integers. For example,

(3)	6	4	1	3	4	2	5	3	0

The circle on the first cell in the above array indicates the position of a marker. At each step in the puzzle you are allowed to move the marker the number of squares indicated by the integer present in the location it currently occupies. The marker may move either to the left or to the right, but it can not move beyond the two ends of the array. For example, in the puzzle configuration given above the marker can only move 3 places to the right, because there is no room to move 3 places to the left.

The goal of the puzzle is to move the marker to the 0 which is present in the last location of the array. You can assume that 0 is always placed at the last location of the array and no other array location contains a 0 in it. You can also assume that all the numbers (except 0) in the array are positive integers.

The above configuration can be solved by making following moves:

Starting position	(3)	6	4	1	3	4	2	5	3	0
Step 1: Move right	3	6	4	(1)	3	4	2	5	3	0
Step 2: Move left	3	6	(4)	1	3	4	2	5	3	0
Step 3: Move right	3	6	4	1	3	4	(2)	5	3	0
Step 4: Move right	3	6	4	1	3	4	2	5	(3)	0
Step 5: Move left	3	6	4	1	3	(4)	2	5	3	0
Step 6: Move right	3	6	4	1	3	4	2	5	3	(0)

Although the above configuration is solvable, there are some configurations of this puzzle which can not be solved. For example, consider this initial configuration:

(3)	1	2	3	0

With this initial configuration the marker will go back and forth between the two 3's but it will never reach 0. So, the puzzle is unsolvable in this case.

Your task is to write a *recursive* function, which takes as an argument the array which represents an initial configuration of the puzzle and decides whether the puzzle is solvable or not (this function will return true if the puzzle is solvable and return false if it is not). The function prototype will look something like this:

```
bool Puzzle (const int array[], int n, int current);
```

Here, `array` contains the configuration of the puzzle (note that this array is constant and you can not change it inside the function), `n` is the number of elements in the array, and `current` indicates the current position of the marker. So, in the case of the first configuration given above the initial function call will be `Puzzle(array,10,0)` where array contains the ten elements that have been shown above.

Note: Be sure to keep track of the array elements that you have already visited otherwise you may end up with infinite recursion. (Hint: For this you can use a temporary array. You are allowed to change the above function prototype slightly to incorporate the temporary array).

© Fareed Ul Hassan Balg

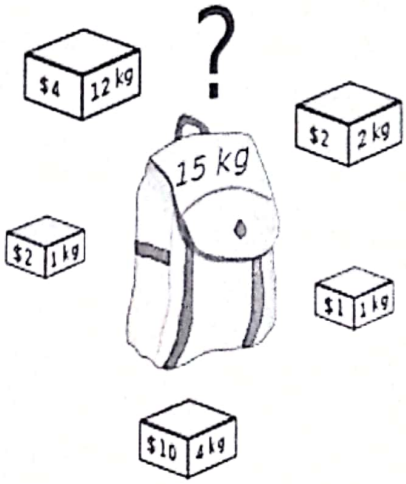
Scanned with CamScanner

Task – 13:

Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

So, formally describing the problem, you are given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).



int knapSack(int W, int weights[], int values[], int N);

Task – 14:

Google the following and attempt the solutions for it:

- Determinant of N order matrix
- 8 puzzle problem
- N Queens Problems
- Sudoku Solver
- .
- .
- .
- etc

Scanned with CamScanner