Recursive Functions

Ajit Rajwade

Refer: Chapter 10 of the book by Abhiram Ranade

Recursion

- We have seen nested function calls in earlier slides - for example the function to determine the roots of equation via bisection made a call to function f.
- A function, however, can also **make a call to itself**, with either the same or different parameter values. This is called **recursion** or a **recursive function**.
- Recursion is a very useful tool in many computer programs, but must be used with care.
- Recursion typically divides the problem into **smaller** instances (same problem with smaller parameter values) and then **combines** the solutions of those instances to get the complete solution.

Example 1: Factorial

```
int factorial (int n){

if (n == 0) return 1;

int prod=1;

for (int i = n; i > 0; i=i-1){

    prod *= i;

}

return prod;

}
```

Non-recursive

```
int rec_factorial (int n){

if (n == 0) return 1;

else {

    int f = n*rec_factorial(n-1);

    return f;

}

}
```

Recursive: The line in blue color is called the base case. It is very important - without it, your program will not halt, leading to infinite recursion.

Execution and Activation Frames

- Suppose `main` makes the call: `int f = rec_factorial(3);`
- The AF (activation frame) of `main` is pushed onto the stack.
- Inside `rec_factorial(3)`, the `n == 0` check fails and you enter the `else` part where variable `f` is created and a call to `rec_factorial(2)` needs to be made.
- The AF of `rec_factorial(3)` is pushed onto the stack, and you enter `rec_factorial(2)`.
- Again the `n == 0` check fails and you enter the `else` part where variable `f` is created. The AF of `rec_factorial(2)` is pushed onto the stack and you enter `rec_factorial(1)`.
- When you enter `rec_factorial(1)`, the `n == 0` check fails and you need to make a call to `rec_factorial(0)`.
- The AF of `rec_factorial(1)` is pushed onto the stack and you enter `rec_factorial(0)`.
- Here the `n==0` check is successful and you return 1 to `rec_factorial(1)`. The AF of `rec_factorial(0)` is destroyed.

Execution and Activation Frames

- The AF of `rec_factorial(1)` is retrieved. Since `n` equals 1 and `rec_factorial(0)` returns 1, the value 1*1 is returned to `rec_factorial(2)`.
- The AF of `rec_factorial(1)` is destroyed.
- The AF of `rec_factorial(2)` is retrieved. Since `n` equals 2 and `rec_factorial(1)` returned 1, the value 2*1 is returned to `rec_factorial(3)`.
- The AF of `rec_factorial(2)` is destroyed.
- The AF of `rec_factorial(3)` is retrieved. Since `n` equals 3 and `rec_factorial(2)` returned 2, the value 3*2 is returned to `main`.

Example 2: GCD computation

```
int gcd(int m, int n)
{
      while (m%n !=0)
      {
            int remainder = m%n;
            m = n;
            n = remainder;
      }
      return n;
}
```

Non-recursive GCD
Assumption `m > n`

```
int rec_gcd(int m, int n)
{
      if (m%n ==0) return n;
      return rec_gcd(n,m%n);
}
```

Recursive GCD: Note the base case!
Assumption `m > n`

Example 3: Drawing a Tree

- In computer science, a tree is a structure which denotes hierarchy.
- For example, the director of an institute is the root of a tree.
- The heads of different departments are “children” of the root.
- The faculty members are “children” of the head of the department, and so on.
- The directors, heads and faculty members are all **nodes** of the tree.
- The nodes at the lowest level of hierarchy are called **leaf nodes**.
- In this example, we will consider a binary tree, where any node has at the most two children - a **left child** and a **right child**.
- There is an edge between a parent node and its child(ren).
- A node without any children is called a **leaf node**.
- We will draw such a tree, and recursion will be very useful here.

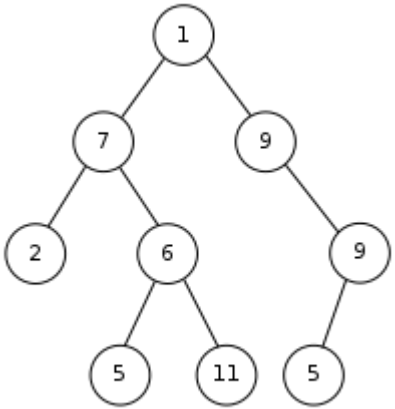
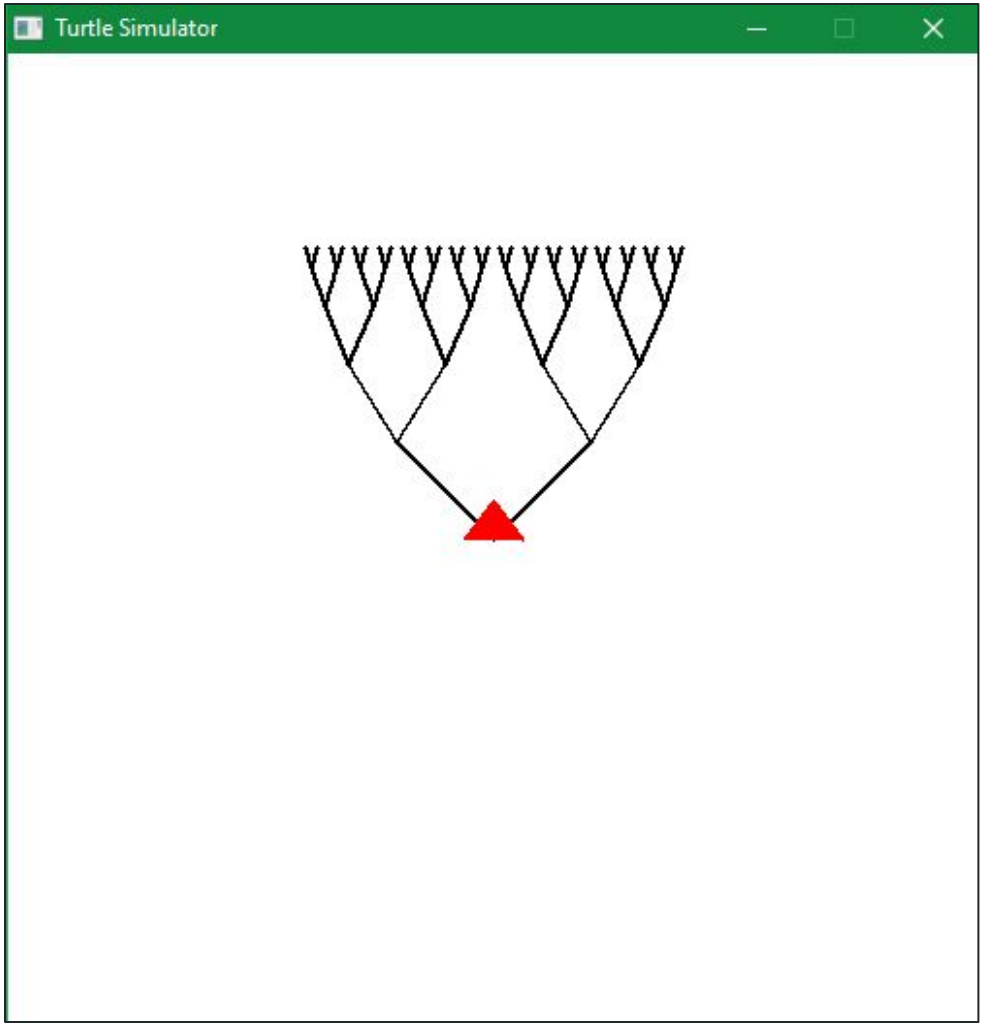


Image source



We want to write a program which will produce a tree like this.

Recursion is very natural for a task like this.

Consider you draw a root and its two children.

Each child is itself the root of a subtree.

You are thus splitting the problem into smaller parts, in a recursive manner.

Example 3: Drawing a Tree: Height of a node

- There are some nodes in a tree which have no children. These are called **leaf nodes**.
- The height of a node is the length of the longest path from the root node to any leaf. The length of a path = number of edges on the path.
- Height of a leaf node is always 0 in a complete binary tree (i.e. a tree in which every node has 2 children except the leaf nodes which have no children).

```
#include<simplecpp> // h below is the height of the node
void tree (int h, float H_b, float W_b, float rx, float ry){
    if (h > 0)
    {
        float LSRx = rx-W_b/4; // x coordinate of endpoint of left branch
        float RSRx = rx+W_b/4; // x coordinate of endpoint of right branch
        float SRy = ry-H_b/4; // y coordinate of endpoint of both branches

        // draw the left subtree, by means of a line
        Line Lbranch(rx,ry,LSRx,SRy); Lbranch.imprint();
        // draw the right subtree, by means of a line
        Line Rbranch(rx,ry,RSRx,SRy); Rbranch.imprint();
        tree(h-1,H_b-H_b/h,W_b/2,LSRx,SRy); // recursive call to left subtree
        tree(h-1,H_b-H_b/h,W_b/2,RSRx,SRy); // recursive call to right subtree
    }
    else return; // base case: do nothing for height = 0, i.e. leaf node.
}
main_program{
turtleSim();
left(90);
tree(5,200,200,250,250);
wait(15); }
```

The updates in `H_b` and `W_b` ensure that as height decreases, subtrees of smaller size are drawn.

Example 4: Computing powers efficiently

- Suppose we want to write a program to efficiently evaluate x^n where n is an integer and x is real-valued.
- One method is to multiply x with itself $n-1$ times, but this is inefficient and we will develop a better method.
- If n is even, then we know that $x^n = x^{n/2} * x^{n/2}$. If you evaluated $x^{n/2}$, then you just have to square its value to obtain x^n.
- Now $x^{n/2}$ can be evaluated by multiplying x with itself $n/2-1$ times.
- But you can further split $x^{n/2} = x^{n/4} * x^{n/4}$ and carry out similar operations recursively.
- Except for the case where n is an exact power of 2, you will encounter odd-valued n at some step or other of the recursion. What do you do?
- Just express $n = 2m + 1$ and write $x^n = x * x^m * x^m$.
- Base case: For $n = 0$, we have $x^n = 1$ and for $n = 1$, we have $x^n = x$.

Example 4: Computing powers efficiently

```
float rec_power (float x, int n) {
// n should be a non-negative integer
if (n == 0) return 1;  // base case 1
if (n == 1) return x;  // base case 2
float temp = rec_power(x,n/2);  // evaluate x^(n/2)
if (n%2 == 0) return temp*temp;  // n is even
else return temp*temp*x;  // n is odd
}
```

What if n was a negative integer? How would you compute powers in such cases?

Example 5: Virahanka Numbers

- **Prosody** is the study of poetic metres: the rhythmic structure of lines in a verse.
- While reciting poetry, some syllables are stressed and others are relatively unstressed.
- For example: The words in bold are stressed and the others are unstressed

  So **long** as **men** can **breathe**, or **eyes** can **see**,

  So **long** lives **this**, and **this** gives **life** to **thee**
- We will consider that **stressed** syllables have length **2 units** and unstressed ones have length 1 unit.
- Virahanka was a 7th century Indian prosodist who asked the question: how many different poetic metres (V_n) can you construct of some given length n?

Example 5: Virahanka Numbers

- The length of a poetic metre = sum of lengths of all syllables in that metre.
- For example, here are different poetic metres of length 4:
    - 1, 1 ,2
    - 1, 2, 1
    - 2, 1, 1
    - 1,1,1,1
    - 2,2
- The first syllable is either of length 1 or length 2.
- So V_n = number of metres of length $n-1$ with first syllable having length 1 + number of metres of length $n-2$ with first syllable with length 2 = $V_{n-1} + V_{n-2}$.
- A relationship of this form is called a **recurrence relation**.

Example 5: Virahanka Numbers

- The recurrence relation also requires base cases: $V_1 = 1$ and $V_2 = 2$.
- This is justifiable: there is only one poetic metre with one syllabus, and there are two metres of length 2 (1,1 and 2).
- Applying the recurrence relation, we have $V_3 = V_2 + V_1 = 3$, $V_4 = V_3 + V_2 = 5$, $V_5 = V_4 + V_3 = 8$, and so on.
- Given the recurrence relation, we now write a recursive program to compute the n-th Virahanka number.
- Virahanka numbers are popularly known as Fibonacci numbers, though Virahanka had discovered them much earlier.

Example 5: Virahanka Numbers

```
int rec_virahanka (int n){

if ( n == 1) return 1;

if (n == 2) return 2;

return rec_virahanka(n-1) +
rec_virahanka(n-2);

}
```

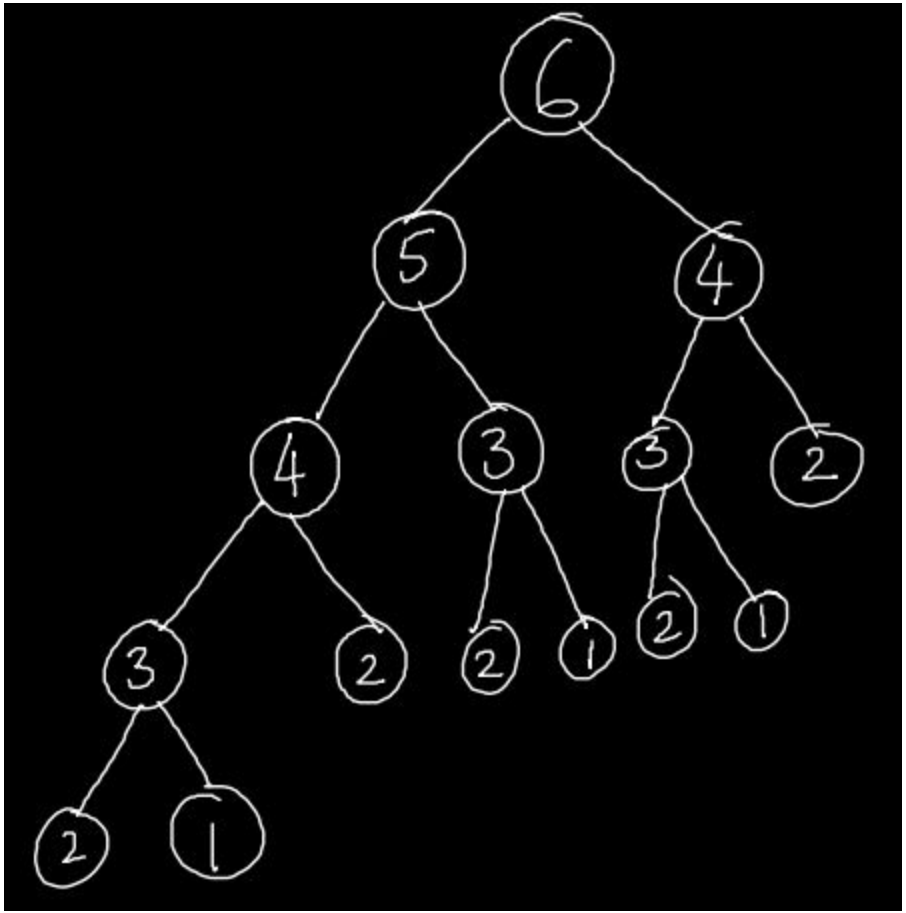
Recursive program

```
int virahanka (int n){

if ( n == 1) return 1;

if (n == 2) return 2;

int prev = 1, curr = 2,val;

for(int i = 0; i < n-2;i++){

val = prev + curr;

prev = curr;

curr = val;

}

return val;

}
```

Non-recursive program

Example 5: Virahanka numbers - a word of caution

- In this case, the recursive program is much slower than the non-recursive one.
- Consider the execution tree for `rec_Virahanka(6)` - see the next slide.
- You will observe that `rec_Virahanka(4)`, `rec_Virahanka(3)` are computed multiple times.
- There is no provision of storage or accumulation of these numbers.
- The tree for `rec_Virahanka(6)` has 14 branches, i.e. 14 calls to the `rec_Virahanka` function!
- In contrast, the loop-based function uses just 6 iterations of the for loop.
- It turns out that a recursive Virahanka function with parameter `n` will make $2^{n/2}$ calls to itself. Even for small values of `n`, this is very inefficient.



“Recursion tree” when rec_virahanka(6) is called.

Example 6: Fractals

- A fractal is a shape that repeats itself at many different scales. Even at different scales, the repeating patterns appear very similar.
- Many shapes occurring in nature can be expressed in the form of fractals: trees, ferns, coastlines, snowflakes, etc.

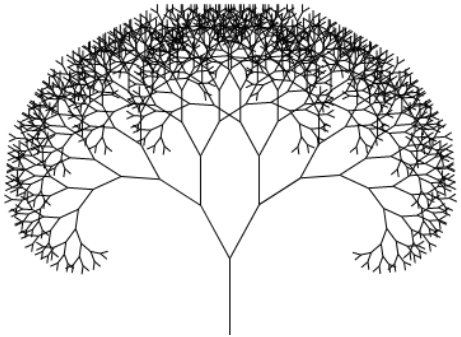
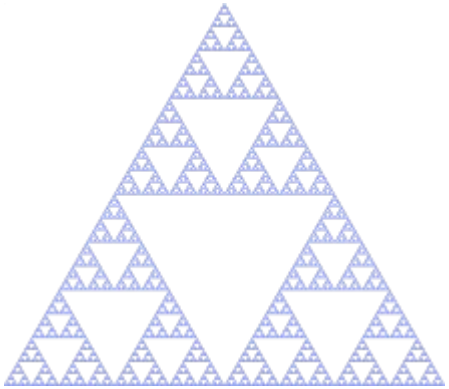


Image source

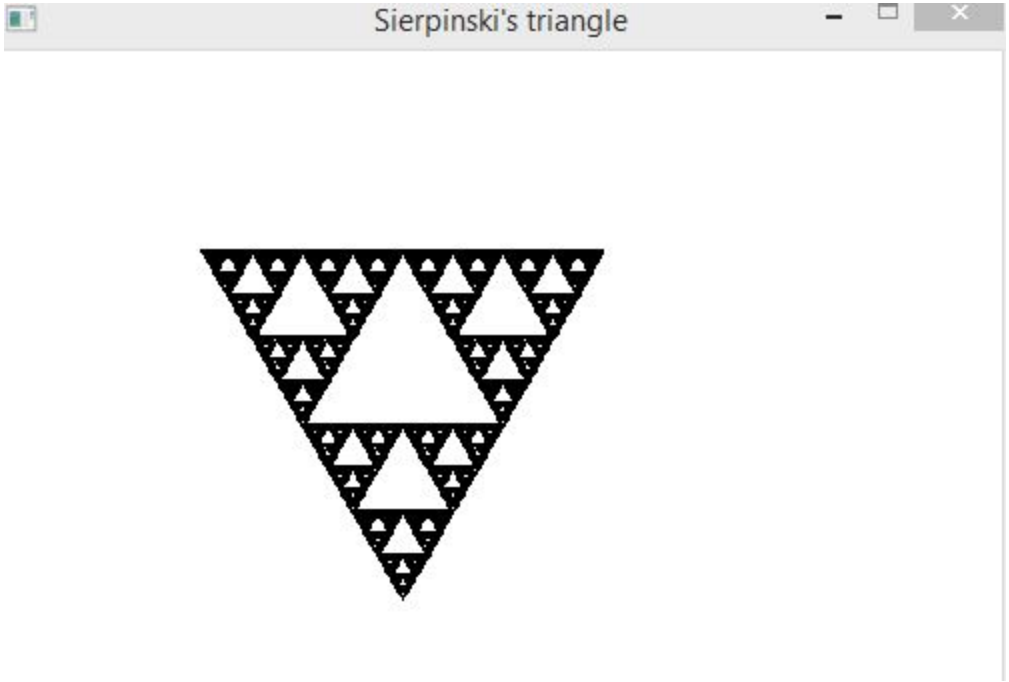
Example 6: Fractals: Sierpinski Triangle

- Draw an equilateral triangle.
- Divide into four equal-sized smaller equilateral triangles.
- Ignore the central triangle and divide the outer three into smaller triangles.
- Recurse up to a desired depth.



```
void sierp(int n, double p1x,double p1y,double p2x, double p2y, double p3x, double p3y){

Line l1(p1x,p1y,p2x,p2y); l1.imprint();// the lines of the equil. triangle

Line l2(p1x,p1y,p3x,p3y); l2.imprint();

Line l3(p2x,p2y,p3x,p3y); l3.imprint();

if (n > 0){

    double q1x = (p1x + p2x)/2; double q1y = (p1y + p2y)/2;// recursive calls to three
outer triangles

    double q2x = (p1x + p3x)/2; double q2y = (p1y + p3y)/2;

    double q3x = (p2x + p3x)/2; double q3y = (p2y + p3y)/2;

    sierp(n - 1, p1x, p1y, q1x, q1y, q2x, q2y);

    sierp(n - 1, p2x, p2y, q1x, q1y, q3x, q3y);

    sierp(n - 1, p3x, p3y, q2x, q2y, q3x, q3y);

}}
```

```
main_program{

int n;

double p1x,p1y,p2x,p2y,p3x,p3y;

cout << "Enter the level of recursion: "; cin >> n;

initCanvas ("Sierpinski's triangle",500,500); // inbuilt function to create
a new window

p1x = p1y = 100;

p2x = 300; p2y = 100;

p3x = 200; p3y = 100 + 100*sqrt(3);

sierp(n,  p1x,  p1y,  p2x,  p2y,  p3x,  p3y);

wait (15);

}
```



Output of the program for level = 6
(n = 6)

Optional problems

1. Write recursive and non-recursive functions to compute the sum of all integers from a to b, where a and b are taken as input from the user. The functions can have the form `int rec_findsum (int a, int b)` and `int findsum (int a, int b)` .

2. Write recursive and non-recursive functions to compute the sum of all integers from a to b in steps of c, where a,b,c are taken as input from the user. Here is what "in steps of c" means: you add the numbers a, a+c, a+2c, …., b. For example, if a = 10, b = 20 and c = 3, then you add the numbers 10,13,16,19. The functions can have the form `int rec_findsum_skip (int a, int b, int c)` and `int findsum_skip (int a, int b, int c)`.

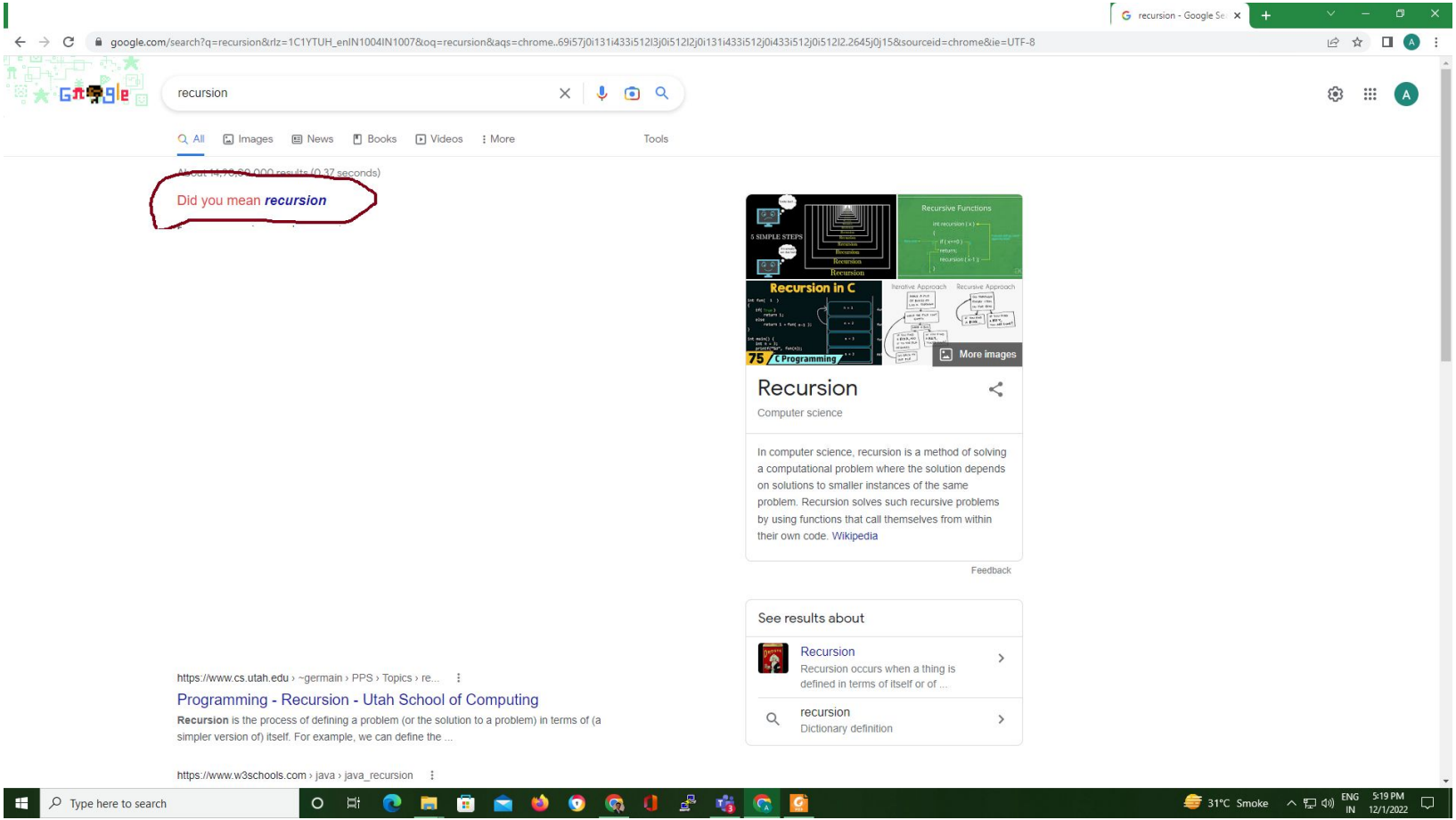
3. Write recursive and non-recursive functions to compute the n-th number of a sequence given as W_n = W_{n-1} + W_{n-2} + W_{n-3}. Take n as input from the keyboard.

4. **Interesting:** The towers of Hanoi is an interesting mathematical puzzle. It consists of three rods called R1, R2, R3 and a number of ciR3ular disks with different diameters and each with a central hole. The rods can slide into the disks. The rod R1 is inserted into n disks, with the bottommost disk having the largest diameter and the diameters successively decreasing up until the smallest disk on the top. The main task is to move all the disks into the rod R3 in such a way that no disk can ever be placed on top of a smaller one. At one step, only one disk can be moved. Any move must take the *topmost* disk from one of the rods and slide it to the topmost position of another rod. Your task is to write a recursive routine which prints the sequence of movements to achieve this. For each movement, you should print the string "Move disk <disk_num> from rod <rod_id> to rod <rodid_2>". For example, if there were three disks 1,2,3 slid into rod A, then the sequence of steps to move all disks to rod R2 via rod R3: (draw a diagram to understand this better)
   - Move disk 1 from rod R1 to rod R2
   - Move disk 2 from rod R1 to rod R3
   - Move disk 1 from rod R2 to rod R3
   - Move disk 3 from rod R1 to rod R2
   - Move disk 1 from rod R3 to rod R1
   - Move disk 2 from rod R3 to rod R2
   - Move disk 1 from rod R1 to rod R2

   To write the recursion, bear the following principle in mind to deal with n disks:
   - Move first n-1 disks from rod R1 to R3, using R2 as an intermediate place.
   - Then move the last disk from R1 to R2.
   - Then move the first n-1 disks from R3 to R2, using R1 as an intermediate place.

   Your function should have the form `void towers_hanoi (int n, int rod_source, int rod_destination, int rod_intermediate)` .

Joke on recursion in Google search engine.



Recursive humour

GNU stands for “GNU's not Unix”