Recursion

AlgoManiaX

Made by Ravishankar Joshi

Topics to be discussed today:

1. Finding factorial(n)
2. a^n and (a^n) % m
3. Decimal to binary conversion
4. Printing a linked list
5. In-order traversal of a binary tree
6. Side effects of recursion
7. Loop invariants

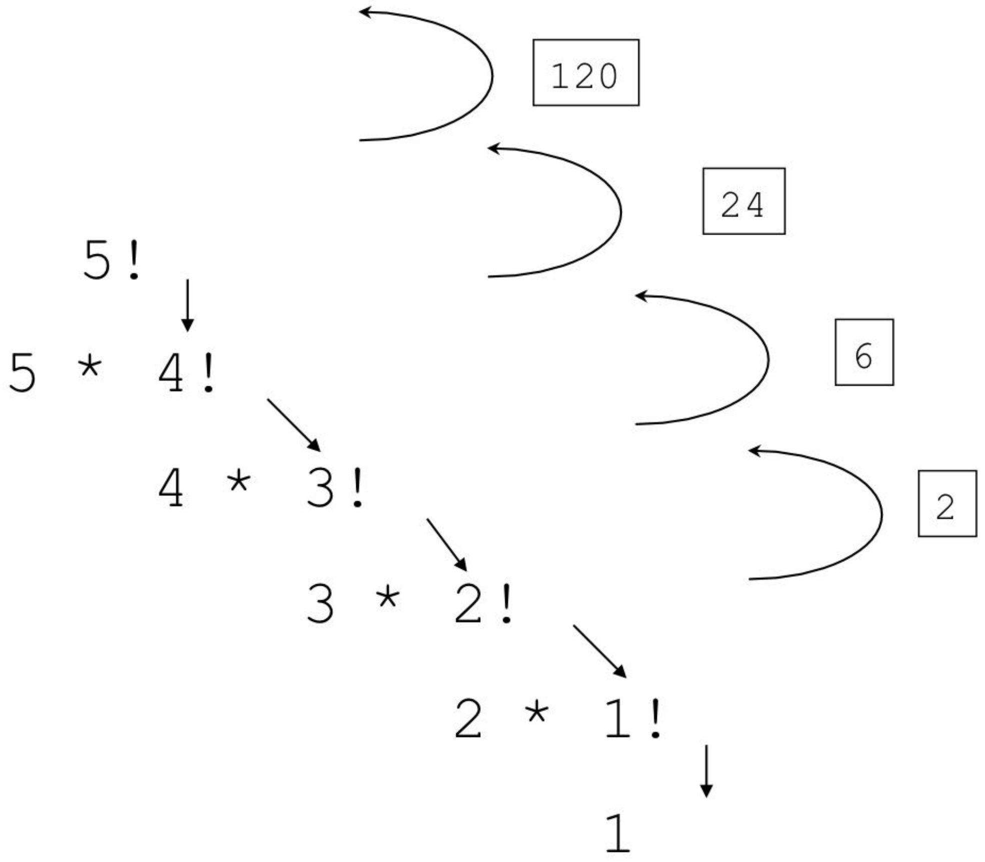
- Recursion is just a function calling itself.
- We can sometimes do things with recursion that we can’t do with loops, and sometimes same things, but easily.
- Sometimes, due to nature of the data structure or algorithm, implementing it iteratively becomes difficult, and recursion comes really handy.
- (Challenge that you can take up in future: Write mergesort iteratively and recursively.)

Factorial

One such example is factorial.
Since, we know that 0! =0, n! = (n-1)! *n, we can write it recursively as follows:

```
long long factorial(long long n)
{    if(n==0)    return 0;
     return factorial(n-1)*n;
}    //Time complexity: O(n), memory complexity: later.
//Since long long can be up to 10^18, we can find
//factorials up to 17!. If we need higher factorials, we
//have to store the factorials as strings.
```

Tree of Recursion calls for factorial:



Computing a^n i.e. pow(a, n)

We know that a^0 =1, a^n = a*(a^n-1), we can write it recursively as follows:

```
long long exp(long long a, long long n)
{    if(n==0)    return 1;
     return a*exp(a, n-1);
}
// long long can hold values only up to 10^18. :)
```

Computing (a^n) % m, i.e. pow(a, n)%m

```
long long exp(long long a, long long n, long long m)
{    if(n==0)    return 1;
     return (a*exp(a, n-1, m)) %m;
}
```

DO NOT CALL pow(a, n)%m. It will give WA.

Time complexity : O(n)

Question that we must ask ourselves:

Can we do better?

Answer is : Yes, this can be done in time O(log n)

Computing (a^k) % MOD, i.e. pow(a, k)%MOD

```
long long ex(long long a, long long k, long long MOD)
{       if(k==0)  return 1;
        if(k==1)  return a%MOD;
        if(k%2)                                 // If k is odd.
        {       long long x=ex(a, (k-1)/2, MOD)%MOD;
                x= (x*x)%MOD;
                return (a*x)%MOD;
        }
        else                                    // If k is even.
        {       long long y=ex(a, k/2, MOD)%MOD;
                y= (y*y)%MOD;
                return y;
        }
}       //Time complexity : O(log n)
```

Decimal to binary conversion

```
string tobinary(int x)
{    if(x==0)    return “”;
     string ans= tobinary(x>>1);
     if(x%2 ==1)
          ans+= “1”;
     else
          ans+= “0”;
     return ans;
}
//Time complexity : O(log x)
```

Printing a linked list

Given a pointer to the *head* node of a linked list, print its elements in order, one element per line. If the head pointer is NULL (indicating the list is empty), don't print anything.

Sample Input

```
NULL
1->2->3->NULL
```

Sample Output

```
1
2
3
```

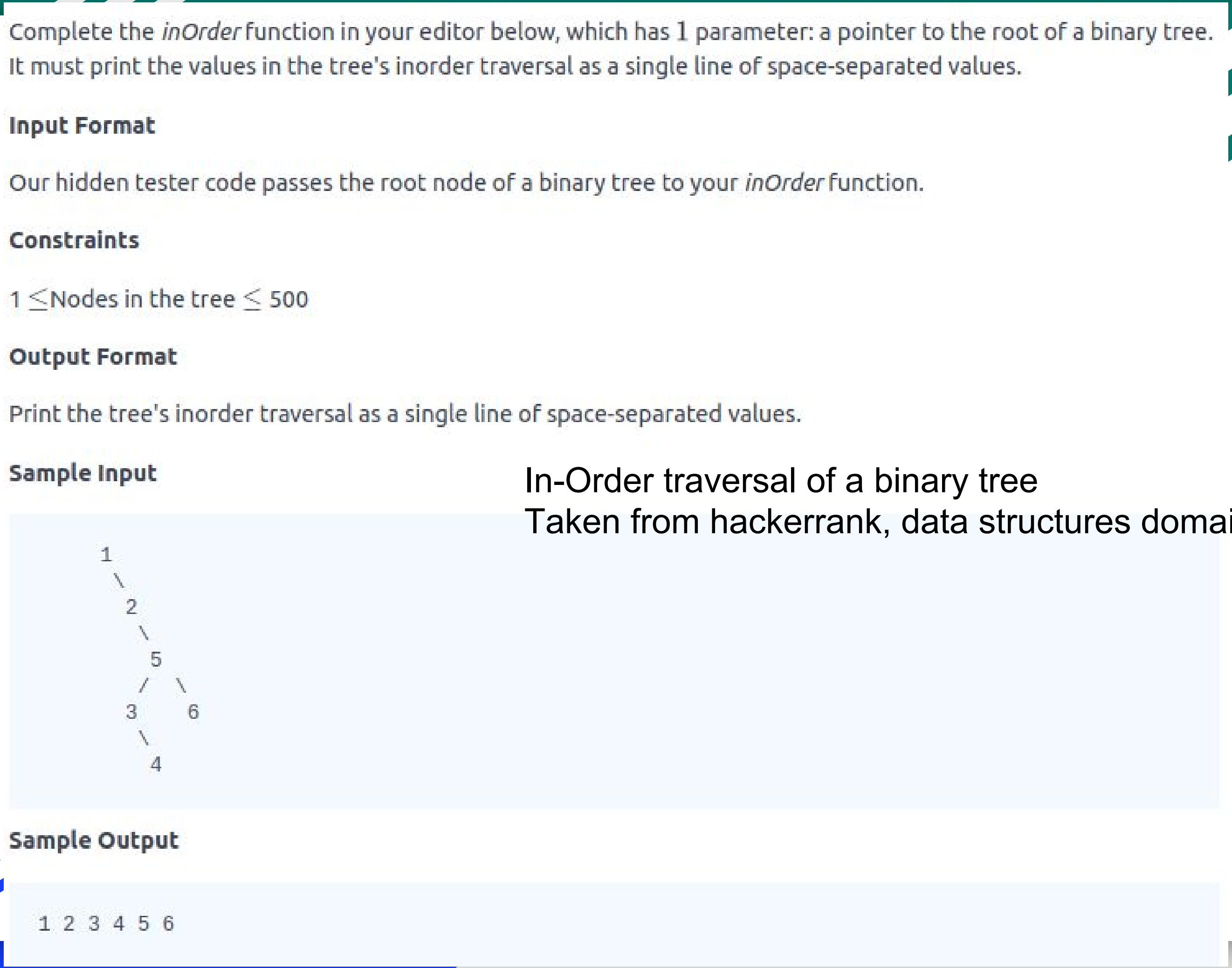
Explanation

Test Case 0: `NULL`. An empty list is passed to the method, so nothing is printed.

Test Case 1: `1->2->3->NULL`. This is a non-empty list so we loop through each element, printing each element's data field on its own line.

Printing a linked list.

```
/*
  Print elements of a linked list on console
  head pointer input could be NULL as well for empty list
  Node is defined as
  struct Node
  {
     int data;
     struct Node *next;
  }
*/
void Print(Node *head)
{   if(head==NULL)
        return;
    else
    {   printf("%d\n",head->data);
        return Print(head->next);
    }
}
```



Complete the *inOrder* function in your editor below, which has 1 parameter: a pointer to the root of a binary tree. It must print the values in the tree's inorder traversal as a single line of space-separated values.

Input Format

Our hidden tester code passes the root node of a binary tree to your *inOrder* function.

Constraints

$1 \leq$ Nodes in the tree ≤ 500

Output Format

Print the tree's inorder traversal as a single line of space-separated values.

Sample Input

```
     1
      \
       2
        \
         5
        /  \
       3    6
        \
         4
```

Sample Output

```
1 2 3 4 5 6
```

In-Order traversal of a binary tree
Taken from hackerrank, data structures domain.

In-order traversal of a binary tree

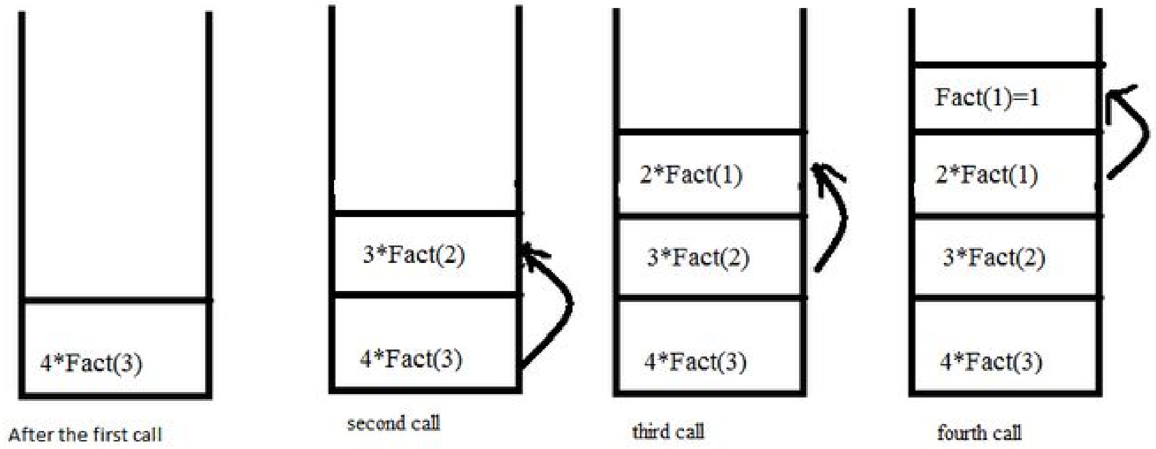
```
/* you only have to complete the function given below.
Node is defined as
struct node
{
    int data;
    node* left;
    node* right;
};
*/

void inOrder(node *root)
{
    if(root->left != NULL)
        inOrder(root->left);
    cout << root->data << ' ';
    if(root->right != NULL)
        inOrder(root->right);
}
```

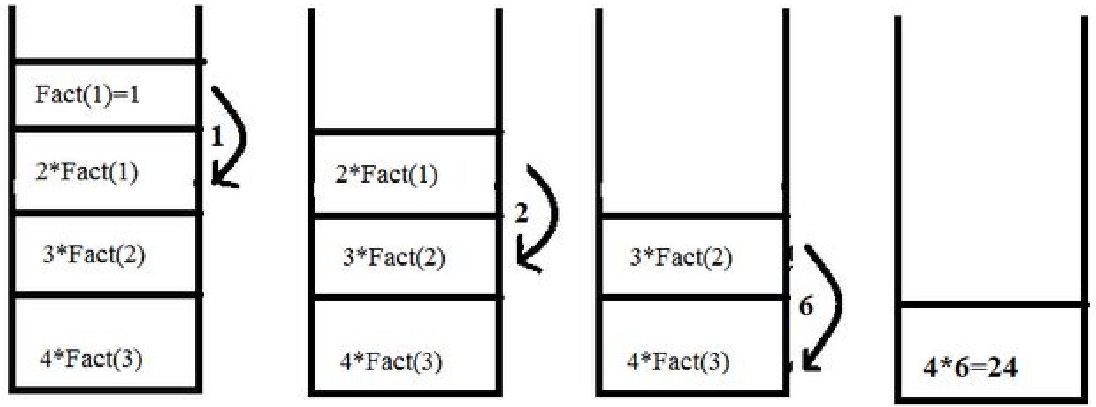
Side effects of recursion

- Like everything else in the world, recursion also has its pros and cons.
- **One con of recursion is its hidden memory complexity**.
- To implement recursion, languages internally make use a stack of memory.
- Temporary values of functions are stored in those stacks.
- When a function calls another function / itself, first the called function is evaluated, and this function returns its value, freeing itself and its local variables from the stack.
- **Another con of recursion is its little overhead to time complexity even if implemented properly.**
- Function calls have a little overhead. So, reduce any redundant function calls. If possible, declare tiny functions (max, min etc) there itself.

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Time overhead of function call:

It depends on your compiler settings and the way it optimizes code. Some functions are inlined. Others are not. It usually depends on whether you're optimizing for size or for speed.

Generally, calling function causes delay for two reasons: 1) The program needs to hook to some random location in memory where your function code starts. To do this, it needs to save the current cursor position into a stack so it knows where to return. This process consumes more than one CPU cycle.

2) Depending on your CPU architecture, there may be a pipeline, which fetches the next few instruction from memory into the CPU cache in parallel with your current instruction execution. This is to speed up execution speed. When you call a function, the cursor hooks to a completely different address and all the cached instructions are flushed from the pipeline. This causes further delays.

Source: https://stackoverflow.com/questions/31779335/why-is-there-overhead-when-calling-functions

Loop invariants

A loop invariant is a property that we assert will be true about a while loop, each time it is about to test its condition. We choose an invariant that we can use to convince ourselves that the program behaves as we intend, and we write the program so as to make the invariant true at the proper times.

Although the invariant is not part of the program text, it is a valuable intellectual tool for designing programs. Every useful while loop that we can imagine has an invariant associated with it. Stating the invariant in a comment can make a while loop much easier to understand.

Loop invariants

```
// invariant: we have written r rows so far
int r = 0;
string names[rows];
// setting r to 0 makes the invariant true
while (r != rows) {
        // we can assume that the invariant is true here
        // writing a row of output makes the invariant false
        std::cout << names[r] << std::endl;
        // incrementing r makes the invariant true again
        ++r;
}
// we can conclude that the invariant is true here.
```

Source: Accelerated C++ by Andrew Koenig and Barbara Moo

Loop invariants

Loop invariants should have 3 properties:

1. They should be true before we run the loop for the first time,
2. They should be true after each iteration / at the time of testing condition.
3. On completion of the loop, they should give us some result to prove correctness of our overall algorithm.

E.g. In the previous loop, invariant is true at the beginning, and after each iteration.

Because of the invariant, we can say that when control exits the loop, number of rows printed is r which “rows” variable, which proves correctness of overall program.

(It’s really like principle of mathematical induction.)

Source: Accelerated C++ by Andrew Koenig and Barbara Moo