CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 24

New Idea - Recursive Function Calls

New Idea - Recursive Function Calls

Can a function invoke itself?

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?
Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number *n*.

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?
Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n.
- We have not written `fact` function yet, but we want to write it using itself.

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?
Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number *n*.
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :
Write the function `fact` in such a way that :

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?
Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number *n*.
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :
Write the function `fact` in such a way that :

- it returns 1, if the argument is 1.

New Idea - Recursive Function Calls

Can a function invoke itself? Yes ! - but why would it want to?
Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number *n*.
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :
Write the function `fact` in such a way that :

- it returns 1, if the argument is 1.
- else it returns *n* times the result of itself when called with argument `n-1`.

`fact` function : Iterative vs Recursive

```
int fact(int n){
   int i;
   int result;
   result = 1;
   for (i = 1; i <= n; i++)
     result = result * i;
   return result;
}
```

`fact` function : Iterative vs Recursive

```
int fact(int n){
   int i;
   int result;
   result = 1;
   for (i = 1; i <= n; i++)
     result = result * i;
   return result;
}
```

invocation : `f = fact(4);`

fact function : Iterative vs Recursive

```
int fact(int n){
   int i;
   int result;
   result = 1;
   for (i = 1; i <= n; i++)
     result = result * i;
   return result;
}
```

```
int fact(int n){
   if (n == 1) return(1);
   return (n*fact(n-1));
}
```

invocation : `f = fact(4);`

`fact` function : Iterative vs Recursive

```
int fact(int n){
   int i;
   int result;
   result = 1;
   for (i = 1; i <= n; i++)
     result = result * i;
   return result;
}
```

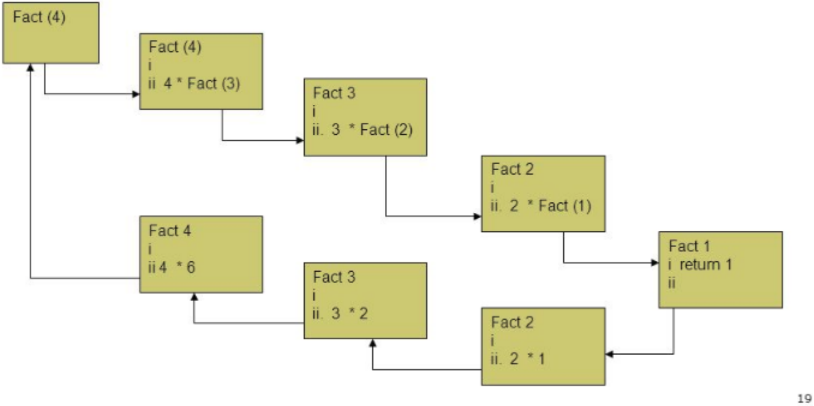
invocation : `f = fact(4);`

```
int fact(int n){
   if (n == 1) return(1);
   return (n*fact(n-1));
}
```

- `(n == 1)` case is called the **base case**. If it not provided, it will turn out to be an infinite recursion.

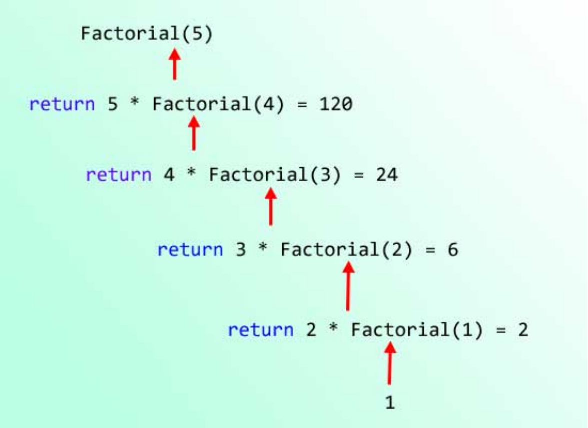
A Graphical Demo of `fact(4)` : Control Flow

Assume that we invoked `fact` with argument as the number 4.

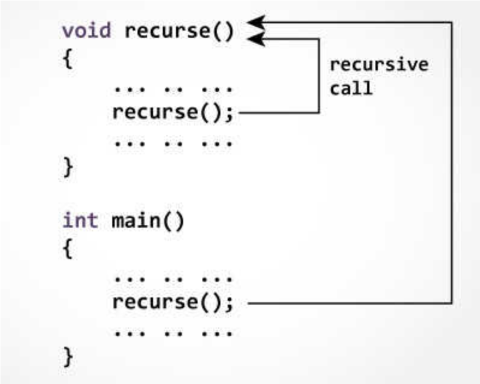


A Graphical Demo of `fact(5)` : Return Values

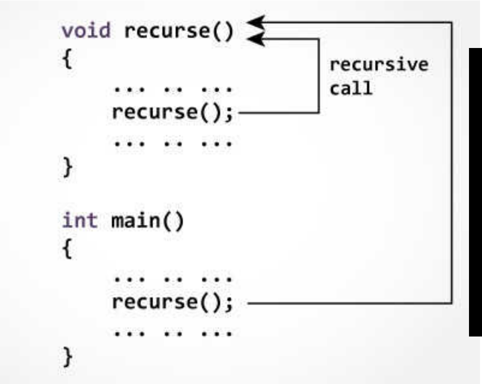
Assume that we invoked `fact` with argument as the number 5.

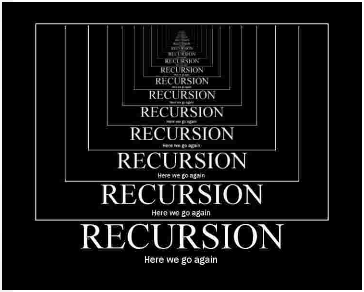


Recursion : Control Flow



Recursion : Control Flow





Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1 + x)^n$.

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Pascal's Identity :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Recursion Example 2 : The Binomial Coefficient

$\binom{n}{k}$ denotes :

- Number of ways of choosing k items from a collection of n items.
- Number of subsets of size k of a set of size n.
- Coefficient of x^k in the expansion of $(1+x)^n$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Task : Write a function which: given n and k computes $\binom{n}{k}$.

Pascal's Triangle

$$\begin{array}{ccccccccccc} &&&&&1&&&&& \\ &&&&1&&1&&&& \\ &&&1&&2&&1&&& \\ &&1&&3&&3&&1&& \\ &1&&4&&6&&4&&1& \\ 1&&5&&10&&10&&5&&1 \end{array}$$

Pascal's Identity :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Base : $\binom{n}{0} = \binom{n}{n} = 1.$

Recursion Example 2 : The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

Recursion Example 2 : The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

```
int binom(int n, int k)
{
 if(k == 0 || n == k)
    return 1;
 int s = binom(n-1,k-1);
 int t = binom(n-1,k);
 return (s+t);
}
```

Recursion Example 2 : The Binomial Coefficient

$$\texttt{bin(n,k)} = \begin{cases} 1 \text{ if } k = 0 \\ 1 \text{ if } n = k \\ \texttt{bin(n-1,k-1)} + \texttt{bin(n-1,k)} & \text{otherwise} \end{cases}$$

```
int binom(int n, int k)
{
 if(k == 0 || n == k)
    return 1;
 int s = binom(n-1,k-1);
 int t = binom(n-1,k);
 return (s+t);
}
```

```
#include<stdio.h>
int main()
{
 int n,k;
 printf("Entern n, k : ");
 scanf("%d %d",&n,&k);
 printf("%d\n",binom(n,k));
 return 0;
}
```

Exercise : Print the Pascal's Triangle

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$$

Exercise : Print the Pascal's Triangle

$\binom{0}{0}$

$\binom{1}{0}$ $\binom{1}{1}$

$\binom{2}{0}$ $\binom{2}{1}$ $\binom{2}{2}$

$\binom{3}{0}$ $\binom{3}{1}$ $\binom{3}{2}$ $\binom{3}{3}$

$\binom{4}{0}$ $\binom{4}{1}$ $\binom{4}{2}$ $\binom{4}{3}$ $\binom{4}{4}$

$\binom{5}{0}$ $\binom{5}{1}$ $\binom{5}{2}$ $\binom{5}{3}$ $\binom{5}{4}$ $\binom{5}{5}$

```
#include <stdio.h>
int binom (int n, int k);
int main()
{
 int i,j,n;
 printf("Enter n :");
 scanf("%d",&n);
 for (i=0;i <= n;i++)
 {
   for (j = 0;j <= i;j++) {
    printf("%6d",binom(i,j));
   }
   printf("\n");
 }
}
```

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Number of ways of building a tower of height n	=	Number of ways of building a tower of height n with bottom-most brick of height 1	+	Number of ways of building a tower of height n with bottom-most brick of height 2.

Recursion Example 3: Virahanka Numbers/Fibonacci Numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? Let this number be V_n.
Right answer comes from the right questions.

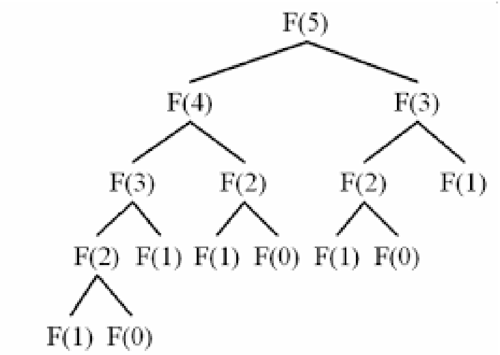
Question: In any given arrangement, what is the bottom-most brick, is it of height 1 or 2?
Two answers possible (It could be 1 or it could be 2, but not both).

Number of ways of building a tower of height n	=	Number of ways of building a tower of height n with bottom-most brick of height 1	+	Number of ways of building a tower of height n with bottom-most brick of height 2.

That is, $V_n = V_{n-1} + V_{n-2}$ Nice !. But how do we compute V_n

Recursion Example 3 : Virahanka Numbers

```
int Virahanka(int n)
{
  if(n == 0) return 1; // V_0
  if(n == 1) return 1; // V_1
  // returning V_{n-1} + V_{n-2}
  return Virahanka(n-1) + Virahanka(n-2);
}
```



Recursive Thinking : Largest Element in an Array

- Till now - we computed only functions which were taught to us or known to us recurively.
- We can solve problems that have a recursive structure using recursive programming. That is more fun !.
- **Key Part:** Formulate the problem recursively.

Example Task: Finding the largest element in an array.

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** : Take out the first element, find the largest of the remaining, and return the largest among the two.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n-1$ elements in the array)

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n-1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n-1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.
- **(Even Better) Recursive Thinking** :

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n - 1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.
- **(Even Better) Recursive Thinking** :
  - Divide the array into two.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n - 1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.
- **(Even Better) Recursive Thinking** :
  - Divide the array into two.
  - Recursively find the largest element in the first half and second half by invoking the same function and let the results by ℓ_1 and ℓ_2 resp.)

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n-1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.
- **(Even Better) Recursive Thinking** :
  - Divide the array into two.
  - Recursively find the largest element in the first half and second half by invoking the same function and let the results by ℓ_1 and ℓ_2 resp.)
  - Compare between the ℓ_1 and ℓ_2, and return the larger element as the largest in the array.

Recursive Thinking (Eg:#1): Largest Element in an Array

- **Iterative Thinking** : Keep the current largest, compare it with the next element. Update the largest with the largest among the two. Do this for all elements in the given order.
- **Recursive Thinking** :
  - Take out the first element.
  - Find the largest element (call it ℓ) in the remaining array recursively (with only $n-1$ elements in the array)
  - Compare between the first and ℓ, and return the larger element as the largest in the array.
- **(Even Better) Recursive Thinking** :
  - Divide the array into two.
  - Recursively find the largest element in the first half and second half by invoking the same function and let the results by ℓ_1 and ℓ_2 resp.)
  - Compare between the ℓ_1 and ℓ_2, and return the larger element as the largest in the array.

Recursive Thinking (Eg:#1): Largest Element in an Array

Recursive thinking: Find the largest of elmnts 2 to $n-1$. Compare it with first and return the largest.

Recursive Thinking (Eg:#1): Largest Element in an Array

Recursive thinking: Find the largest of elmnts 2 to $n-1$. Compare it with first and return the largest.

```
int largest(int i, int n)
{
  if (i == n) return arr[i];

  int l;
  l = largest(i+1,n);
  if (arr[i] > l)
       return arr[i];
  else return l;
}
```

Recursive Thinking (Eg:#1): Largest Element in an Array

Recursive thinking: Find the largest of elmnts 2 to $n - 1$. Compare it with first and return the largest.

```
int largest(int i, int n)
{
  if (i == n) return arr[i];

  int l;
  l = largest(i+1,n);
  if (arr[i] > l)
       return arr[i];
  else return l;
}
```

(Better) recursive thinking: Find the largest of the first half, then in the second half, and then return the largest of the two.

Recursive Thinking (Eg:#1): Largest Element in an Array

Recursive thinking: Find the largest of elmnts 2 to $n - 1$. Compare it with first and return the largest.

```
int largest(int i, int n)
{
  if (i == n) return arr[i];

  int l;
  l = largest(i+1,n);
  if (arr[i] > l)
       return arr[i];
  else return l;
}
```

(Better) recursive thinking: Find the largest of the first half, then in the second half, and then return the largest of the two.

```
int largest(int i, int j)
{
  if (i == j) return arr[i];

  int l1,l2;
  l1 = largest(i,(i+j)/2);
  l2 = largest((i+j)/2+1,j);
  if (l1 > l2)
       return l1;
  else return l2;
}
```