CS1100 – Introduction to Programming

Instructor:

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

Lecture 22

Recursion

Drawing Hands by M. C. Escher.

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;
}
```

invocation : `f = fact(4);`

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

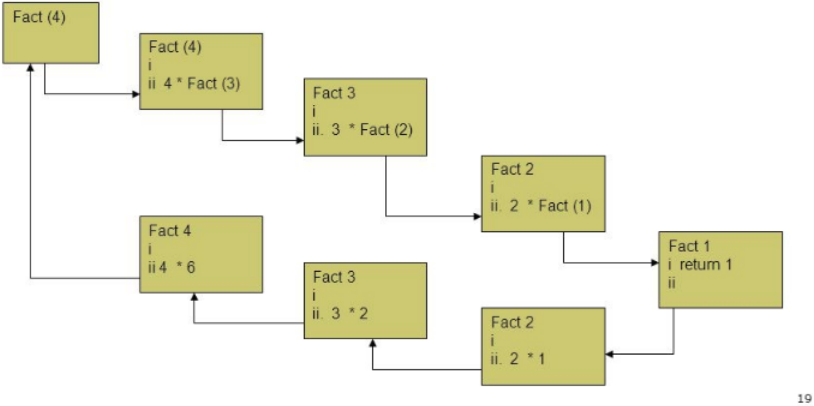
invocation : `f = fact(4);`

```
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.

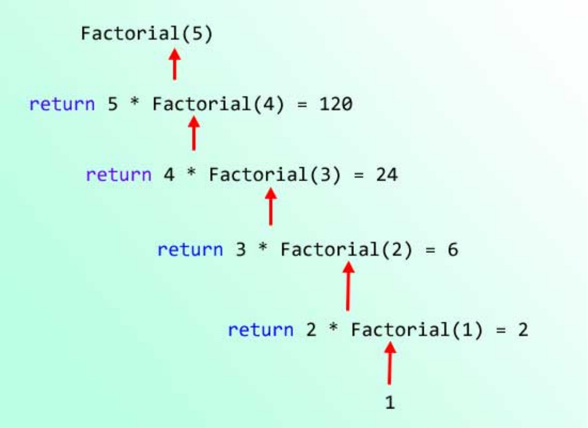
A Graphical Demo of `fact(4)` : Control Flow

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

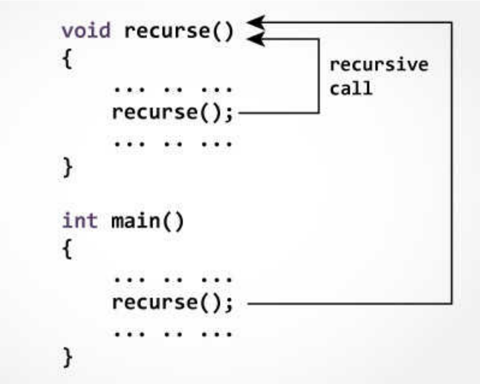


A Graphical Demo of `fact(5)` : Return Values

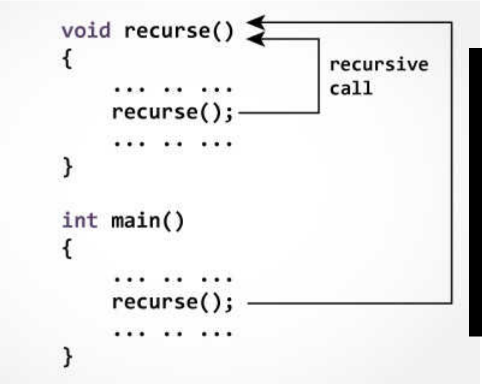
Assume that we invoked `fact` with argument as the number 5.

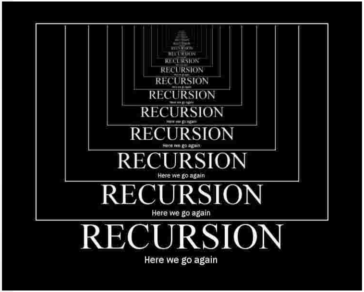


Recursion : Control Flow



Recursion : Control Flow





Recursion: Summary

- The function knows how to solve the *simplest* case. This is also called the base case.
- If the function is invoked using a *complex* case, it breaks the input into
  (i) a part that the function knows how to do.
  (ii) a part that it does not know how to do.
- For (ii) it again invokes the same function – this step is called the recursive step.

Multiplication using repeated addition

Mathematical formulation

Assume x, y are positive integers.

Multiplication using repeated addition

Mathematical formulation

Assume x, y are positive integers.

$$\begin{aligned} f(x,y) &= y + f(x-1,y) && \text{if } x > 1 \\ &= y && \text{otherwise.} \end{aligned}$$

Multiplication using repeated addition

Mathematical formulation

Assume x, y are positive integers.

$$
\begin{aligned}
f(x,y) &= y + f(x-1,y) && \text{if } x > 1 \\
&= y && \text{otherwise.}
\end{aligned}
$$

```
int mult(int x, int y) {
    if (x == 1)
        return 1;
    else
        return ( y + mult(x-1, y));
}
```

Multiplication using repeated addition

Mathematical formulation

Assume x, y are positive integers.

$$\begin{aligned} f(x,y) &= y + f(x-1,y) && \text{if } x > 1 \\ &= y && \text{otherwise.} \end{aligned}$$

```
int mult(int x, int y) {
    if (x == 1)
        return 1;
    else
        return ( y + mult(x-1, y));
}
```

Is the program correct?

Multiplication using repeated addition

Mathematical formulation

Assume x, y are positive integers.

$$
\begin{aligned}
f(x,y) &= y + f(x-1,y) && \text{if } x > 1 \\
&= y && \text{otherwise.}
\end{aligned}
$$

```
int mult(int x, int y) {
    if (x == 1)
        return y;                    //notice the change.
    else
        return ( y + mult(x-1, y));
}
```

Length of a string : recursively

Length of a string : recursively

- Base case: if `str[0] == 0` return 0;

Length of a string : recursively

- Base case: if `str[0] == 0` return 0;
- Else add 1 to the length of string starting at str+1.

Length of a string : recursively

- Base case: if `str[0] == 0` return 0;
- Else add 1 to the length of string starting at str+1.

Length of a string : recursively

- Base case: if `str[0] == 0` return 0;
- Else add 1 to the length of string starting at str+1.

```
int reclen(char str[]) {
      if (str[0] == 0) {
           return 0;
      } else return(1+reclen(str+1));
}
```

Length of a string : recursively

- Base case: if `str[0] == 0` return 0;
- Else add 1 to the length of string starting at str+1.

```
int reclen(char str[]) {
      if (str[0] == 0) {
            return 0;
      } else return(1+reclen(str+1));
}
```

- Note the usage of `str+1`.
- Why is accessing str+1 valid?
  we know that str[0] != 0, hence we will find at least one more character after str[0].

Palindrome

Conceptual formulation

Assume inputs are a string s1, start index *start*, end index *end*.

Assume $start \leq end$.

Palindrome

Conceptual formulation

Assume inputs are a string s1, start index *start*, end index *end*. Assume $start \leq end$.

- if $s1$ is of length 1 it is a palindrome.
- if $s1$ is of length 2 and $s1[start] == s1[end]$, it is a palindrome.
- if $s1[start] == s1[end]$ and $s1[start + 1, end - 1]$ is a palindrome, then $s1$ is a palindrome.

Largest Element in an Array

- Till now - we computed only functions which were taught to us or known to us recursively.
- 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.

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.

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** :

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.

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)

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.

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** :

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.

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.)

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.

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.

Largest Element in an Array

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

Largest Element in an Array

Recursive thinking: Find the largest of elements 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;
}
```

Largest Element in an Array

Recursive thinking: Find the largest of elements 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.

Largest Element in an Array

Recursive thinking: Find the largest of elements 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;
}
```

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

Why is this algorithm better?

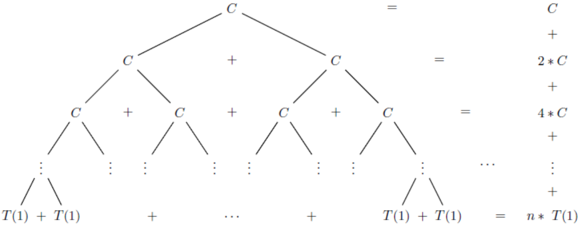
- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.
- Depth of the *recursion tree* is $\log n$, and total time taken is $\approx n \times K$ for some constant K.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.
- Depth of the *recursion tree* is $\log n$, and total time taken is $\approx n \times K$ for some constant K.



What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n - 1$, recursively call `largest` on the sub-array, and perform one comparison after they return.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n - 1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n - 1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n - 1) + C$.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.
- So total time is roughly the same. But depth of recursion tree is larger in one case.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.
- So total time is roughly the same. But depth of recursion tree is larger in one case.
- The *space* complexity of recursive algorithm is proportinal to maximum depth of recursion tree generated. So this is how the second algorithm is better than the first.