

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

```
Search( "/home/potato", "fileH.txt" )  
> Search( "(...)/subfolder1", "fileH.txt" )  
>> Search( "(...)/subfolder1A", "fileH.txt" )
```

We have traversed as deep into **subfolder1** and **subfolder1A** as is possible, so now we need to terminate and begin returning from the recursed subfolder.

```
>> Search( "(...)/subfolder1A", "fileH.txt" )  
Returns "not found"
```

Notes

# *Intro to Recursion*

# *About*

Recursion is another way to solve problems. The general idea is to take a task that needs to be done and break it into smaller, identical tasks.

It can be tricky to think in terms of recursion at first, so we will have an introduction to it.

# *Topics*

1. Breaking up tasks
2. Iterative vs. Recursive

*Breaking up tasks*

# I. *Breaking up Tasks*

For most classes, we use iterative techniques to process data multiple times – whether it's using while loops or for loops.

Recursion is a way to break up a task in such a way that you call the same function over and over and over and over until the task is finished. Each time the recursive function is called, the inputs are tweaked slightly, until some “end-point” is reached.

Notes

# I. *Breaking up Tasks*

For example, let's say you're creating a function to search for a file on a hard-drive.

You start at one path on the directory and, if the file isn't found there, you search each subdirectory, and then each subdirectory of each subdirectory, and so on until you hit the end of each sub-path.

How would you implement this?

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

This would be much easier to implement **recursively** (with a self-calling function) than iteratively (with loops).

The **Search()** function you implement would know how to search the current directory, and if the file isn't found, it will call **Search()** on each of the subdirectories.

Notes



# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

So say we call

**Search( "/home/potato", "fileH.txt" )**

To find "fileH.txt". First it would look through its own files – fileA.txt, fileB.txt, and fileC.txt.

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
- subfolder1/
  - fileD.txt
  - fileE.txt
- subfolder1A/
  - fileF.txt
  - fileG.txt
- subfolder2/
  - fileH.txt

So say we call

**Search( "/home/potato", "fileH.txt" )**

It doesn't find the file in any of these, so next it will run

**Search( "(...)/subfolder1", "fileH.txt" )**

It calls the Search function again with the same file, but a different folder.

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**  
**> Search( "(...)/subfolder1", "fileH.txt" )**

Now subfolder1's files are being investigated, but the file still isn't found.

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

```
Search( "/home/potato", "fileH.txt" )  
> Search( "(...)/subfolder1", "fileH.txt" )  
>> Search( "(...)/subfolder1A", "fileH.txt" )
```

Next, it checks subfolder1A by calling the Search function again.

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

```
Search( "/home/potato", "fileH.txt" )  
> Search( "(...)/subfolder1", "fileH.txt" )  
>> Search( "(...)/subfolder1A", "fileH.txt" )
```

The files within subfolder1A are checked, but we still haven't found fileH...

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**  
**> Search( "(...)/subfolder1", "fileH.txt" )**

We have traversed as deep into **subfolder1** and **subfolder1A** as is possible, so now we need to terminate and begin returning from the recursed subfolder.

**>> Search( "(...)/subfolder1A", "fileH.txt" )**  
**Returns "not found"**

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**

We have traversed as deep into **subfolder1** and **subfolder1A** as is possible, so now we need to terminate and begin returning from the recursed subfolder.

**>> Search( "(...)/subfolder1A", "fileH.txt" )  
Returns "not found"**

**> Search( "(...)/subfolder1", "fileH.txt" )  
Returns "not found"**

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**

We've returned back to the original Search function. We're not done with the subfolders here, so next we will call Search on subfolder2...

**Search( "(...)/subfolder2", "fileH.txt" )**

Notes



# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**  
**> Search( "(...)/subfolder2", "fileH.txt" )**

Our file is found here, so we can begin returning from here.

Let's say if a file is found, it will return the full path:

/home/potato/subfolder2/fileH.txt

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**

**> Search( "(...)/subfolder2", "fileH.txt" )**

**> Search( "(...)/subfolder2", "fileH.txt" )**

**Return /home/potato/subfolder2/fileH.txt**

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

**Search( "/home/potato", "fileH.txt" )**

**> Search( "(...)/subfolder2", "fileH.txt" )**

**> Search( "(...)/subfolder2", "fileH.txt" )**

**Return /home/potato/subfolder2/fileH.txt**

**Search( "/home/potato", "fileH.txt" )**

**Return result of**

**Search( "(...)/subfolder2", "fileH.txt" )**

Notes

# I. Breaking up Tasks

- /home/potato/
  - fileA.txt
  - fileB.txt
  - fileC.txt
  - subfolder1/
    - fileD.txt
    - fileE.txt
    - subfolder1A/
      - fileF.txt
      - fileG.txt
  - subfolder2/
    - fileH.txt

So the underlying structure is that the function calls itself over and over and over, generally reducing the task to be done.

Once the task is as small as possible (the last subdirectory to check in a path), it will hit some **terminating case**, causing it to return some value.

The terminating case is returned through each function call, and we return back to the original function call, and out of the function.

Notes

*Iterative vs.  
Recursive*

## 2. *Iterative vs. Recursive*

Let's cover some example problems in both an iterative and a recursive way to highlight the differences.

Notes

## 2. Iterative vs. Recursive

This function will count up from **start** to **end**, displaying each value.

### Function definition – iterative

```
void CountUp_Iter( int start, int end )
{
    for ( int i = start; i <= end; i++ )
    {
        cout << i << "\t";
    }
}
```

### Function definition – recursive

```
void CountUp_Rec( int start, int end )
{
    cout << start << "\t";

    // Terminating case
    if ( start == end )
        return;

    // recursive case
    CountUp_Rec( start+1, end );
}
```

Notes

## 2. Iterative vs. Recursive

This function will count up from **start** to **end**, displaying each value.

### Function definition – iterative

```
void CountUp_Iter( int start, int end )
{
    for ( int i = start; i <= end; i++ )
    {
        cout << i << "\t";
    }
}
```

The for loop creates a variable, *i*, which walks through a range of numbers one at a time.

### Function definition – recursive

```
void CountUp_Rec( int start, int end )
{
    cout << start << "\t";

    // Terminating case
    if ( start == end )
        return;

    // recursive case
    CountUp_Rec( start+1, end );
}
```

Notes



## 2. Iterative vs. Recursive

This function will count up from **start** to **end**, displaying each value.

### Function definition – iterative

```
void CountUp_Iter( int start, int end )
{
    for ( int i = start; i <= end; i++ )
    {
        cout << i << "\t";
    }
}
```

The for loop creates a variable, *i*, which walks through a range of numbers one at a time.

### Function definition – recursive

```
void CountUp_Rec( int start, int end )
{
    cout << start << "\t";

    // Terminating case
    if ( start == end )
        return;

    // recursive case
    CountUp_Rec( start+1, end );
}
```

The recursive function displays the “start” value. If “start” is the same as “end”, then we’re done. Otherwise, call the function again, passing in start+1.

Notes

## 2. Iterative vs. Recursive

Compute the value of  $n!$

### Function definition – iterative

```
int Factorial_Iter( int n )
{
    for ( int i = n-1; i > 0; i-- )
    {
        n *= i;
    }
    return n;
}
```

### Function definition – recursive

```
int Factorial_Rec( int n )
{
    // Terminating case
    if ( n == 0 )
    {
        return 1;
    }

    // Recursive case
    return n * Factorial_Rec( n-1 );
}
```

Notes

## 2. Iterative vs. Recursive

Compute the value of  $n!$

### Function definition – iterative

```
int Factorial_Iter( int n )
{
    for ( int i = n-1; i > 0; i-- )
    {
        n *= i;
    }
    return n;
}
```

In the for loop, start at  $n-1$  and go until the value of 1. Within the loop, multiply  $n$  by the current value of  $i$  each time.

Return  $n$  once finished.

### Function definition – recursive

```
int Factorial_Rec( int n )
{
    // Terminating case
    if ( n == 0 )
    {
        return 1;
    }

    // Recursive case
    return n * Factorial_Rec( n-1 );
}
```

Notes

## 2. Iterative vs. Recursive

Compute the value of  $n!$

Function definition – iterative

```
int Factorial_Iter( int n )
{
    for ( int i = n-1; i > 0; i-- )
    {
        n *= i;
    }
    return n;
}
```

In the for loop, start at  $n-1$  and go until the value of 1. Within the loop, multiply  $n$  by the current value of  $i$  each time.

Return  $n$  once finished.

Function definition – recursive

```
int Factorial_Rec( int n )
{
    // Terminating case
    if ( n == 0 )
    {
        return 1;
    }

    // Recursive case
    return n * Factorial_Rec( n-1 );
}
```

If the  $n$  value passed in is 0, then just return 1.

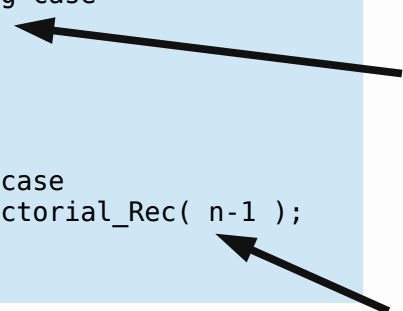
Otherwise, return the current value of  $n$  times Factorial of  $n-1$ .

Notes

## 2. Iterative vs. Recursive

```
int Factorial_Rec( int n )
{
    // Terminating case
    if ( n == 0 )
    {
        return 1;
    }

    // Recursive case
    return n * Factorial_Rec( n-1 );
}
```



For a recursive function, we need to have one or more **terminating cases**, where we return some value, and we need a **recursive case**, where the function is called again with different arguments.

Notes

## 2. Iterative vs. Recursive

```
int Factorial_Rec( int n )
{
    // Terminating case
    if ( n == 0 )
    {
        return 1;
    }

    // Recursive case
    return n * Factorial_Rec( n-1 );
}
```

```
void CountUp_Rec( int start, int end )
{
    cout << start << "\t";

    // Terminating case
    if ( start == end )
        return;

    // recursive case
    CountUp_Rec( start+1, end );
}
```

For each of these, the recursive case whittles down the problem set that we're working with.

We keep calling the function over and over until we finally hit a terminating case, which causes a chain-reaction of returns.

Notes

# *Conclusion*

We will work with recursion more in Chapter 5, and recursion will really come in handy once we're working with trees.

For now, it's good to get some practice with small problems to help you learn to think "recursively".