

Chapter 11

Recursion

Recursion

Recursive functions are functions that call themselves. For a problem to lend itself to a recursive solution it must be one that may be broken into sub-problems of the same kind.

Recursive Algorithm - an algorithm that has two solutions

- one in which the algorithm is expressed as a smaller version of itself (the recursive case) and,
- one in which the solution may be found directly (the base case, the case that stops the recursion)

We're going to examine a few examples of problems that we may have already solved using iteration and compare them side by side with a recursive solution.

Example 1: Calculating the sum of the squares in the range $m - n$

Given two positive integers, m and n , where $m \leq n$, we want to find

$$\text{SumOfSquares}(m,n) = m^2 + (m+1)^2 + (m+2)^2 + \dots + n^2$$

$$\text{SumOfSquares}(3,8) = 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 = 199$$

Note: There are easier algebraic methods for calculating this sum but this is a great example to compare iteration and recursion side by side.

```
// iterative sum of the squares
int SumOfSquares(int m, int n)
{

}

}
```

Iterative solutions typically build up (accumulate) the final solution in stages using a repetitive process. Each step gets closer and closer to the final solution and this solution is returned at the end of the function. To obtain a recursive solution to this problem we need to find a way of solving the problem by breaking it into sub-problems that can be solved using the same method. The solutions to the sub-problems are then combined to get the overall solution.

```

// recursive sum of the squares
int SumOfSquares(int m, int n)
{
    if(there is only one number in the range m..n, => m=n)
        the solution is the square of m
    else
        there is more than one number in the range m..n and the solution is obtained
        by adding the square of m to the sum of the squares in the range m+1..n
    endif
}

```

```

// recursive sum of squares calculation
int SumOfSquares(int m, int n)
{
    if(
        {
            // the base case
        }
    else
    {
        // the recursive case
    }
}
}

```

So, the overall pattern of a recursive function is that it breaks the problem into smaller and smaller sub-problems, which are solved by calling the function recursively, until these sub-problems get small enough that they become base cases that can be solved directly. The above is an example of *going-up recursion*. The calls progress upward until they hit the base case, the case where the sub-range is (n..n). This "breaking down" process is called *decomposition*. We are going to look at two methods used for visualizing the recursive process, a *call tree* and a *call trace*.

Call Tree

Call Trace

Rewrite SumOfSquares using *going down recursion*.

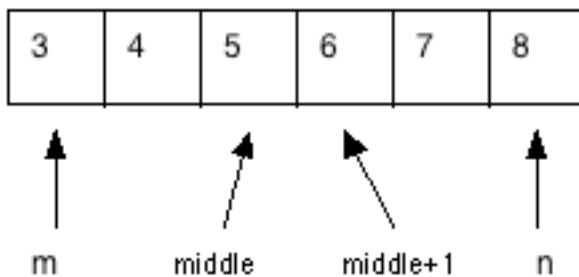
```
int SumOfSquares(int m, int n)
{
    if(m == n)
    {
        // the base case
    }
    else
    {
        // the recursive case
    }
}
```

Given the following:

```
int SumOfSquares(int m, int n)
{
    int middle;

    if(m == n)
    {
        return m*m;
    }
    else
    {
        middle = (m+n) / 2;
        return SumOfSquares(m, middle) + SumOfSquares(middle+1, n);
    }
}
```

This example uses a decomposition method that involves splitting the overall problem into two halves. If a range of numbers (m..n) contains just one number then the solution is simply m^2 . Otherwise, the range (m..n) is split into two half ranges. Assuming that middle is the mid point, the left half range would be (m..middle), and the right half range would be (middle+1..n). So, the recursive case simply says that the sum of the squares of the entire range of integers (m..n) can be obtained the adding the sum of the squares from the left half range to the sum of the squares from the right half range.



Call Tree

Write an iterative function and then a recursive function to calculate N!

```
long RecFac(int n)
{
```

```
}
```

```
long ItFac(int n)
{
```

```
}
```

Infinite Regress

A common problem with recursive functions is called *infinite regress*. It is similar to an infinite loop and it occurs when a recursive function calls itself endlessly. There are two reasons a recursive program can call itself endlessly

- 1) there is no base case to stop the recursion, or
- 2) a base case never gets called

What would happen given the following call

```
cout << RecFac(0);
```

ALWAYS save your program before you attempt to run it if you have any recursive calls in it.

Let's look at a recursive algorithm to output the contents of an array.

```
void PrintArray(int theArray[], int top, int bottom)
{
    if(top>=bottom)
    {
        cout << theArray[bottom]<<endl;
        PrintArray(theArray,top,bottom+1);
    }
}
int main()
{
    int anArray[6]={1,2,3,4,5,6};
    int top=5;
    int bottom=0;
    PrintArray(anArray,top,bottom);
    return 0;
}
```

Note: Once the deepest call (last call) was reached, each of the calls prior to this call returned without doing anything. This is called *tail recursion*.

Tail recursion - an algorithm in which no statements are executed after the return from the recursive call to the function

This is usually an indication that the problem could be solved more easily using iteration.

Re-write the above function so it prints the list in reverse order.

```
if(
{

}
```

Fibonacci Numbers

Given the following recurrence relations:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

Let's look at an iterative solution to generate the *n*th Fibonacci number.

```
int IterFib(int n)
{
    int last, current, next;
    if( n == 0 )
    {
        current = 0;
    }
    else
    {
        last = 0;
        current = 1;
        for(int i = 2; i <= n; i++)
        {
            next = last + current;
            last = current;
            current = next;
        }
    }
    return current;
}
```

Write a recursive version below:

```
int RecFib(int n)
{

}

}
```

Examine the following iterative list function and describe what it does.

```
struct LstNode
{
    int val;
    LstNode* next;
};
typedef LstNode* LstPtr;

void Build(LstPtr& h)
{
    LstPtr p;
    for(int i=1; i<=5; i++)
    {
        p = new LstNode;
        p->val = 2*i;
        p->next = h;
        h = p;
    }
}

void WhatDoesThisDo(LstPtr& h)
{
    LstPtr p,q;
    p = NULL;
    while(h != NULL)
    {
        q = h;
        h = h->next;
        q->next = p;
        p = q;
    }
    h = p;
}

void Print(LstPtr h)
{
    LstPtr p;
    p = h;
    while(p != NULL)
    {
        cout << endl << p->val;
        p = p->next;
    }
    cout << endl;
}
```

Show the output produced after the following calls Build -> Print -> WhatDoesThisDo -> Print (assume the head has been initialized to NULL prior to the function call)

Recursive Linked List Exercise

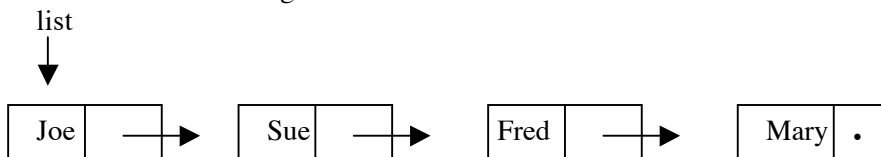
```
void FD(LstPtr& lst, LstPtr& head, LstPtr& tail)
{
    if(lst != NULL)
    {
        tail = lst->next;
        head = lst;
        head->next = NULL;
    }
}
```

```
LstPtr FC(LstPtr lst1, LstPtr lst2)
{
    LstPtr p;
    if(lst1 == NULL)
        return lst2;
    else
    {
        p = lst1;
        while(p->next != NULL)
            p = p->next;
        p->next = lst2;
        return lst1;
    }
}
```

```
LstPtr FR(LstPtr lst)
{
    LstPtr head, tail;
    if(lst == NULL)
        return NULL;
    else
    {
        FD(lst, head, tail);
        return FC(FR(tail), head);
    }
}
```

```
int main()
{
    LstPtr list, newList;
    list = NULL;
    Build(list);
    Print(list);
    newList = FR(list);
    Print(newList);
    return 0;
}
```

Assume the following list has been created after the call to Build.



Name _____

Due Date _____

Recursive Binary Search Lab

```
int BinSrch(int ar[ ], int key, int bott, int top)
```

```
{
    bool found = false;
    int mid;
    while(!found && top >= bott)
    {
        mid = (bott + top) / 2;
        if(ar[mid] == key)
            found = true;
        else
            if(ar[mid] < key)
                bott = mid + 1;
            else
                top = mid - 1;
    }
    if (found)
        return mid;
    else
        return -1;
}
```

```
int RecBinSrch(int ar[ ], int key, int bott, int top)
```

```
{
    int mid;

    if( ) // base case

    else
        if( ) // base case

        else
            if( ) // recursive case

            else

}
```