A Brief Introduction

Recursion
Many algorithms require that processes are repeated

- Iterating through the elements of an array
  - Computing statistics
  - Printing
  - ...
- Sorting
- Programming constructs for repetition
  - while
  - do ... while
  - for
Repetition with Functions

- In addition to loops there is another way to repeat a process
  - That uses function calling instead of loops
- Consider the factorial example
  - The factorial of 5 equals 5 *4!
- Let`s state this more generally
  - The factorial of x equals x * (x - 1)! where x > 1
  - And the factorial of 1 = 1
Let's write a function to compute factorials using the ideas presented previously.

- For all $x > 1$, $x! = x \times (x - 1)!$ and $1! = 1$

```cpp
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long factorial(int x){
    if(x == 1)
        {
            return 1;
        }
    else{
        return x * factorial(x-1);
    }
}
```
void recursionTest() 
{
    int x = 10;
    cout << x << "! = " << factorial(x);
}

incidentally, in case you were wondering why my factorial functions all returned long longs, here is 20!

10! = 3628800

20! = 2432902008176640000
INT_MAX = 2147483647
Recursion
Recursive Functions

- The *factorial* function is a *recursive* function
  - Because it *calls itself*

```c
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long factorial(int x){
    if(x == 1){
        return 1;
    }
    else{
        return x * factorial(x-1);
    }
}
```
The factorial function is recursive
- A recursive function calls itself
- Each call to a recursive function results in a separate call to the function, with its own input

Recursive functions are just like other functions
- The invocation is pushed onto the call stack
- And removed from the call stack when the end of the function or a return statement is reached
- Execution returns to the previous function call
Recursion and Memory

```c
int fact(int x)
{
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

int test = 4;
cout << x << "! = " << fact(test);

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c++
int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c
int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c
int fact(int x)
{
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c
int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c
int fact(int x)
{
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c
int test = 4;
cout << x << "! = " << fact(test);
```

Call stack – shown as 4 byte cells (since we only allocate space for ints)

<table>
<thead>
<tr>
<th>data</th>
<th>...</th>
<th>4</th>
<th>4</th>
<th>0</th>
<th>3</th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>...</td>
<td>2056</td>
<td>2060</td>
<td>2064</td>
<td>2068</td>
<td>2072</td>
<td>2076</td>
<td>2080</td>
<td>2084</td>
</tr>
</tbody>
</table>
```
Recursion and Memory

```c++
int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c++
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

```c
int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}
```

```c
int test = 4;
cout << x << "! = " << fact(test);
```

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursion and Memory

int fact(int x){
    int result = 0;
    if(x == 1)
        result = 1;
    else
        result = x * fact(x-1);
    return result;
}

int test = 4;
cout << x << "! = " << fact(test);

test

data | ... | 4 | ... |
address | ... | 2056 | 2060 | 2064 | 2068 | 2072 | 2076 | 2080 | 2084 | 2088 | ... |

call stack – shown as 4 byte cells (since we only allocate space for ints)
Recursive functions do not use loops to repeat instructions
- But use recursive calls, in if statements

Recursive functions consist of two or more cases, there must be at least one
- Base case, and one
- Recursive case
Base Case

- The base case is a smaller problem with a simpler solution
  - This problem’s solution must *not* be recursive
    - Otherwise the function may never terminate
- There can be more than one base case
  - And base cases may be implicit
The recursive case is the same problem with smaller input

- The recursive case must include a recursive function call
- There can be more than one recursive case
Finding Recursive Solutions

- Define the problem in terms of a smaller problem of the same type
  - The recursive part
  - e.g. `return x * factorial(x-1);`
- And the base case where the solution can be easily calculated
  - This solution should not be recursive
  - e.g. `if (x == 1) return 1;`
How can the problem be defined in terms of smaller problems of the same type?
  - By how much does each recursive call reduce the problem size?
    - By 1, by half, ...?
  - What is the base case that can be solved without recursion?
    - Will the base case be reached as the problem size is reduced?
Here is a recursive sum function (very similar to factorial):

```c
int sum(int x){
    if(x == 1)
        return 1;
    else{
        return x + sum(x-1);
    }
}
```

And here is what happens when you call `sum(5000)`
Recursive algorithms have more overhead than similar iterative algorithms.

- Because of the repeated function calls
- This may cause a *stack overflow*
  - The area of memory allocated to the call stack is used up

Some algorithms can still be implemented recursively in a safe way.

- Will the recursive factorial cause a stack overflow?
Some algorithms are naturally recursive
  - So writing a recursive solution is much easier than writing an iterative one
    - e.g. Quicksort

Recursion is a great problem solving tool
  - It is another way to reason about solutions
  - Even if we implement the solution using iteration