The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated..

PSet – Recursion

Table of Contents

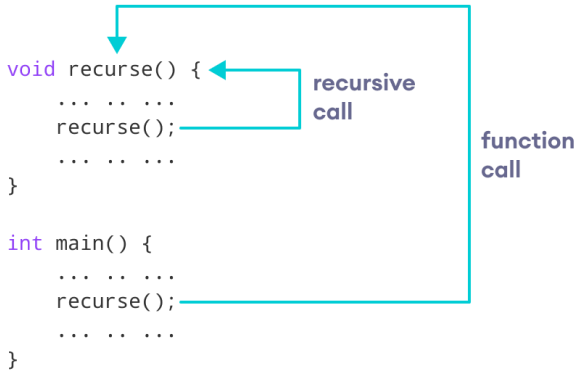
Getting Started - Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to **smaller instances** of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science. An infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions. Most computer programming languages support recursion by allowing a function to call itself within the program text.
(Resources: https://en.wikipedia.org/wiki/Recursion & www.codingBat.com)

Recursive algorithm is expressed in terms of

1. **base case(s)** for which the solution can be stated **non-recursively**,
2. **recursive case(s)** for which the solution can be expressed in terms of a smaller version of itself.

The following figure shows how recursion works by calling itself over and over again.



Files provided:

The following files are provided with this pset:

1. `recursion.cpp, driver.cpp` – a skeleton code
2. `recursionx.exe, recursionx` - a sample solution

Step 1: Implementing Recursion Functions

As you have seen in the lecture, you practice the recursion with a simple factorial function.

- Create a file called `factorial.cpp` and practice this example.
- There are four different styles of the same function `factorial()`. You are supposed to try all those styles and turn in the last form, "Code Example 4:" as a part of pset.

Example 1: Factorial

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

factorial(n) – iterative solution

```
long long unsigned factorial(n) {
  long long total = 1;
  for (int i = n; i > 1; i--) total  *= i;
  return total;
}
```

factorial(n) – recursive solution

input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]
1. if *n* is 0, **return** 1
2. otherwise, **return** [*n* × factorial(*n*-1)]
end factorial

factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000

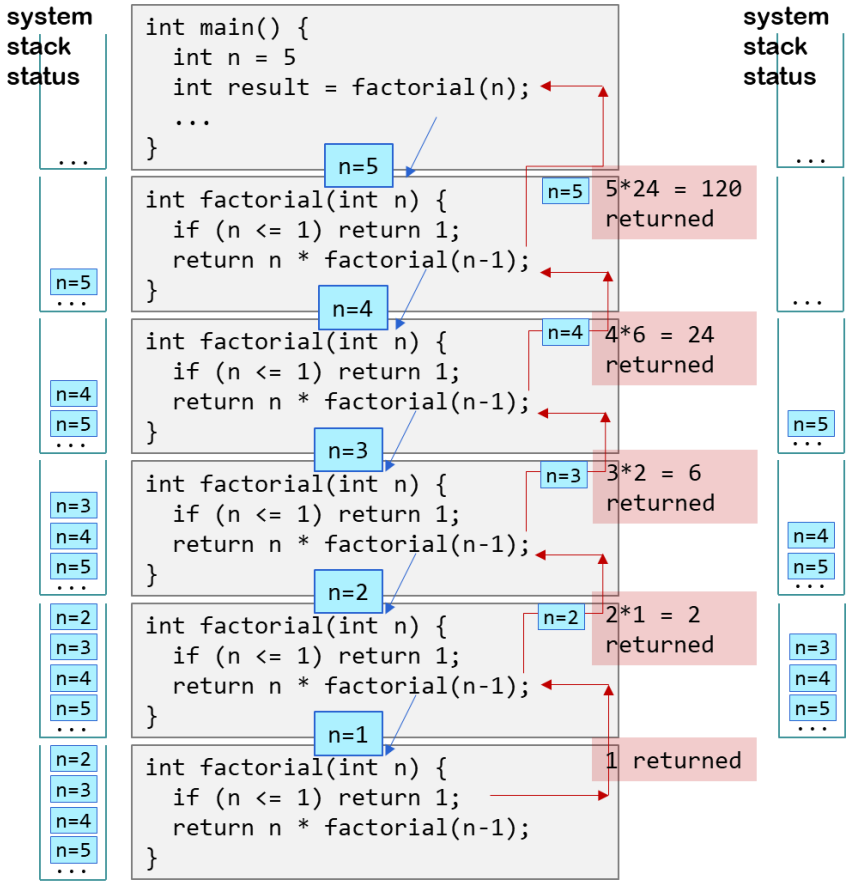
Hint:
First, detect the "base case", a case so simple that the answer can be returned immediately (here when `n==1 or n==0`). Otherwise make a recursive call of `factorial(n-1)` (towards the base case). Assume the recursive call returns a correct value and fix that value up to make our result.

When we think about solving this problem recursively, we need to figure out what our subproblems will be. Let's break it down:

1. We know `factorial(5) = 5 * factorial(4) aka 5! = 5 * 4!.`
2. To continue, `factorial(5) = 5 * (4 * factorial(3))` which equals `5 * (4 * (3 * factorial(2))` and so on…
3. …Until you get `5 * 4 * 3 * 2 * 1` and the only remaining subproblem is `1!.`
4. `factorial(1)` and `factorial(0)` always equals 1 so this will be our base case.

As we can see in the figure below, the `factorial()` function is calling itself. However, during each call, we have decreased the value of n by 1. When n is less than 1, the `factorial()` function ultimately returns the output.

Remember that when a function keeps on calling another without finishing, the current status of function or n value is pushed (or saved) in the system stack. Then as soon as it gets a returned value from `factorial(n-1),` the n value is popped retrieved for the computation.



Using this line of thinking, we can write a recursive solution to our factorial problem as shown below:

Code Example 1:

```
long long unsigned factorial(int n) {
  if (n == 1 || n == 0) return n;
  return n * factorial(n-1);
}
```

Code Example 2:

```
long long unsigned factorial(int n) {
   return (n == 1 || n == 0) ? 1 : n * factorial(n - 1);   // using ternary operator
}
```

Code Example 3:

```
#include <iostream>

long long unsigned factorial(int n) {
  std::cout << "n= " << n << std::endl;
  if (n == 1 || n== 0) return n;
  auto result = n * factorial(n - 1);
  std::cout << "n= " << n << "\tn!= " << result << std::endl;
  return result;
}

int main() {
  factorial(5);
}
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset02recursion> g++ factorial.cpp -o factorial
PS C:\GitHub\nowicx\psets\pset02recursion> ./factorial
n= 5
n= 4
n= 3
n= 2
n= 1
n= 2    n!= 2
n= 3    n!= 6
n= 4    n!= 24
n= 5    n!= 120
PS C:\GitHub\nowicx\psets\pset02recursion>
```

The first function call begins with `n = 5`, but it never ends until all the factorials from 4, 3, 2, 1 are computed first. In the `main ()`, it calls the function only one, but the function calls itself until `n = 1` or the base case is reached.

Code Example 4:

```
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000
```

```
#include <iostream>

long long unsigned factorial(int n) {
  if (n == 1 || n == 0) return n;
  auto result = n * factorial(n - 1);
  return result;
}

int main() {
    int n[] = {1, 2, 3, 4, 5, 8, 12, 20};
    for (auto x: n)
    std::cout << "factorial(" << x << ") = " << factorial(x) << std::endl;
    return 0;
}
```

Sample Run:

```
PS C:\GitHub\nowicx\psets\pset2a-recursion> ./factorial
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(8) = 40320
factorial(12) = 479001600
factorial(20) = 2432902008176640000
PS C:\GitHub\nowicx\psets\pset2a-recursion>
```

I recommend that you follow the code by hand or step through the debugger and understand the concept of the recursion.

Follow the code style of "Code Example 4" and complete the rest recursion code of this PSet.

Step 2: Implementing Recursion Functions

Once you practiced the recursive function using **`factorial(),`** now we implement more of recursive functions..

1. Two recursive functions, **`factorial()`** and **`gcd(),`** are already coded in **`recursion.cpp`** as examples to follow.
2. Implement the rest of examples including **`factorial()`** in **`recursion.cpp`** and while referencing Example 1 ~ 2 source code in **`recursion.cpp`** and **recusrionDriver.cpp**, and running **`recursionx.exe`**

Example 2: GCD (Great Common Divisor)

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x,y)) & \text{if } y > 0 \end{cases}$$

Recurrence relation for greatest common divisor, where `x%y` expresses the remainder of `x/y`:

$$\begin{aligned} gcd(x,y) &= gcd(y, x\%y) \qquad if\ y\ !=0 \\ &= gcd(x,0) = x \qquad if\ y = 0 \end{aligned}$$

gcd(x, y)

input: integer x, y such that x >= y, y > 0
output: gcd of x and y
1. if y is 0, **return** x
2. otherwise, **return** [*gcd* (y, x%y)]
end gcd

Ex: Computing the recurrence relation for x = 27 and y = 9

```
gcd(27, 9)   = gcd(9, 27 % 9)
             = gcd(9, 0)
```

Ex: Computing the recurrence relation for x = 111 and y = 259

```
gcd(111, 259)   = gcd(259, 111% 259)
                = gcd(259, 111)
                = gcd(111, 259% 111)
                = gcd(111, 37)
                = gcd(37, 111% 37)
                = gcd(37, 0)
                = 37
```

Code:

```
int gcd(int x, int y) {
  if (y == 0) return x;
  return gcd(y, x % y);
}
```

or

```
int gcd(int x, int y) {
   return y == 0 ? x : gcd(y, x % y);
}
```

Example 3: Fibonacci

The fibonacci sequence is a famous bit of mathematics, and it happens to have a recursive definition. The first two values in the sequence are 0 and 1 (essentially 2 base cases). Each subsequent value is the sum of the previous two values, so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on.

Define a recursive `fibonacci(n)` method that returns the nth fibonacci number, with n=0 representing the start of the sequence.

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(11) = 89
fibonacci(33) = 3524578
fibonacci(44) = 701408733 (It takes some time to compute.)

Code:

```
long long unsigned fibonacci(int n) {

  cout << "your code here\n";

}
```

Example 4: Bunny Ears

We have a number of bunnies and each bunny has two big floppy ears. We want to compute the total number of ears across all the bunnies recursively (without loops or multiplication).

bunnyEars(0) = 0
bunnyEars(1) = 2
bunnyEars(2) = 4
bunnyEars(3) = 6
bunnyEars(234) = 468

Hint:
First detect the base case `(bunnies == 0)`, and in that case just return 0. Otherwise, make a recursive call to `bunnyEars(bunnies-1).` Trust that the recursive call returns the correct value, and fix it up by adding 2.

Code:

```
int bunnyEars(int bunnies) {

  if (bunnies == 0) return 0;

  // Recursive case: otherwise, make a recursive call with bunnies-1
  // (towards the base case), and fix up what it returns.

  cout << "your code here\n";
}
```

Example 5: Funny Ears

We have bunnies and funnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even funnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny and funny line 1, 2, ... n (without loops or multiplication).

funnyEars(0) = 0
funnyEars(1) = 2
funnyEars(2) = 5
funnyEars(3) = 7
funnyEars(4) = 10
funnyEars(9) = 22
funnyEars(10 = 25

Code:

```
int funnyEars(int funnies) {
  // your code here
}
```

Example 6: Triangle

We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

triangle(0) = 0
triangle(1) = 1
triangle(2) = 3
triangle(3) = 6
triangle(4) = 10
triangle(7) = 28

Code:

```
int triangle(int rows) {
// your code here
}
```

Example 7: Sum of digits

Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

sumDigits(126) = 9
sumDigits(12) = 3
sumDigits(1) = 1
sumDigits(10110) = 3
sumDigits(235) = 10

Code:

```
int sumDigits(int n) {
  // your code here
}
```

Example 8: Count 8

Given a non-negative int n, return the count of the occurrences of 8 as a digit, so for example 818 yields 2. (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

count8(818) = 2
count8(8) = 1
count8(123) = 0
count8(881238) = 3
count8(48581) = 2
count8(888586198) = 5
count8(99899) = 1

Code:

```
int count8(int n) {
// your code here
}
```

Example 9: Power N

Given base and n that are both 1 or more, compute recursively (no loops) the value of base to the n power, so powerN(3, 2) is 9 (3 squared).

powerN(2, 5) = 32
powerN(3, 1) = 3
powerN(3, 2) = 9
powerN(3, 3) = 27
powerN(10, 2) = 100
powerN(10, 3) = 1000

Code:

```
long long powerN(int base, int n) {
  // your code here
}
```

A command line example to build

To build this program:

```
g++ recursion.cpp driver.cpp -I../../include -L../../lib –lnowic -o recursion
```

This command line works at pset3a folder and the following folder structures and files exist.

```
nowic/lib/libnowic.a
nowic/include/nowic.h
nowic/psets/pset3a/recursion.cpp
nowic/psets/pset3a/driver.cpp
```

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____ Section: _____ Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the problem sets partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again if it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

Submit the following files in piazza folder.

- **`recursion.cpp`**
- **`driver.cpp`**

Due and Grade

- Due: 11:55 pm