

Chapter 14: Recursion

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Recursion

- We've also talked about functions and the caller/callee relationship
- So what if we have a function call itself?
 - I.e. the caller is the same function as the callee
- This is known as *recursion* and it one of the most powerful ways to control a program.
- The intuition behind this is that we can solve a big problem by breaking it down into a smaller problem.
 - Recursion

Base Cases

- If a function is going to call itself, how will the function eventually stop calling itself?
- If the function doesn't have a way to stop calling itself, the function will call itself for forever (essentially an infinite loop) until your computer runs out of resources.
- We fix this problem by creating a **base case** that doesn't call the function again

Example: A factorial function

- Definition of a factorial:

$$n! = \prod_{k=1}^n k$$

- How can we write this in a recursive manner?
- How can we write the factorial of n as a function of the factorial of $n - 1$?

$$n! = n * (n - 1)!$$

- Ok, so we can write the factorial of n as a function of the factorial of $n - 1$. But what should the base case be?
 - When $n = 1$, we stop
- In C++, this code is incredibly simple to write:

```
1 int factorial(int n){
2     // base case
3     if(n == 1)
4     {
5         return 1;
6     }
7     // otherwise, recurs into factorial(n * 1) (this is called the
8     // recursive case)
9     else
10    {
11        return n * factorial(n-1);
12    }
13 }
```

- How would you write this function using a for loop?

```
1 int factorial_loop(int n){
2     int fac = 1;
3     for (int i = 1; i <=n; i++){
4         fac *= i;
5     }
6     return fac;
7 }
```

- The parallel to the **base case** is the **recursive case** where the function calls itself.
- The recursive case should make some progress towards the base case, otherwise the program may never terminate

The Fibonacci Sequence

- Fibonacci (introduced the idea in 1202) wondered a simple question has an interesting mathematical formulation: how many rabbits could be born in a year?
- He assumed the following conditions:
 - Begin with one male rabbit and female rabbit that have just been born.
 - Rabbits reach sexual maturity after one month.
 - The gestation period of a rabbit is one month. (How long it takes to give birth - for humans it's 9 months typically)
 - After reaching sexual maturity, female rabbits give birth every month.
 - A female rabbit gives birth to one male rabbit and one female rabbit.
 - Rabbits do not die.

- This is best shown with this diagram:

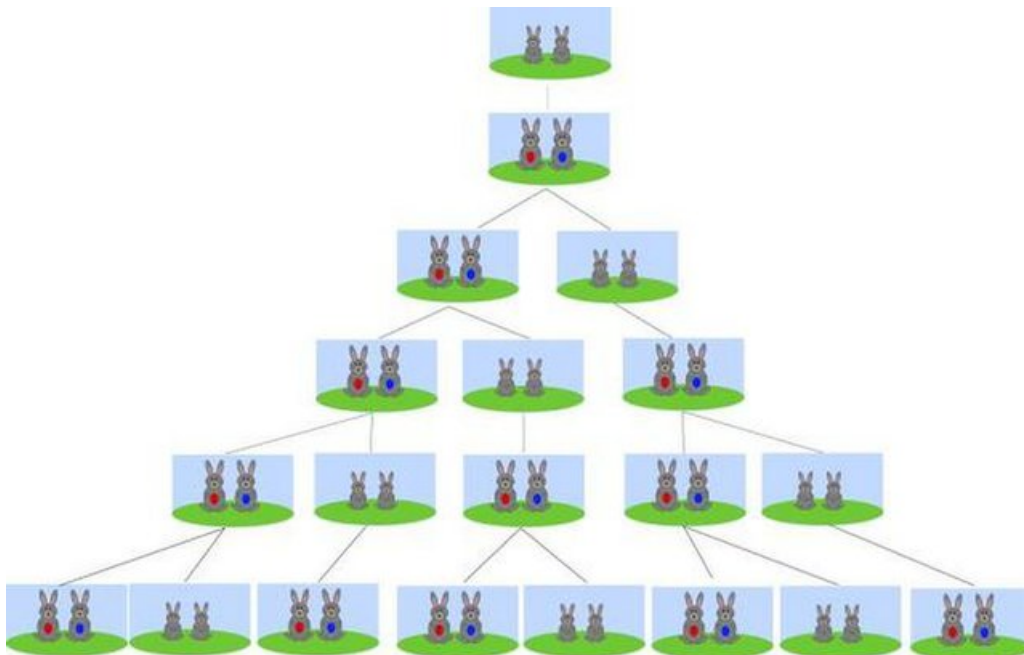


Figure 1: fibonacci_rabbits.jpg

- After one month, the first pair is not yet at sexual maturity and can't mate.
- At two months, the rabbits have mated but not yet given birth, resulting in only one pair of rabbits.
- After three months, the first pair will give birth to another pair, resulting in two pairs.
- At the fourth month mark, the original pair gives birth again, and the second pair mates but does not yet give birth, leaving the total at three pair.
- This continues until a year has passed, in which there will be 233 pairs of rabbits.
- Why Care?
 - Fibonacci's observation extends far beyond breeding rabbits. This pattern shows up in nature everywhere - growth pattern of sunflower seeds, hurricanes, galaxies. Tons of spirals in nature follow this pattern

Formal definition

- $f_n = f_{n-1} + f_{n-2}$
- Initial values at 1 and 2 for f_{n-1} and f_{n-2} , respectively (this is a hint for our base case!
 - $f(0) = 0$
 - $f(1) = 1$
 - $f(n) = f(n-1) + f(n-2)$

- How would you write the recursive version to output

```
1 int fib(int n){
2     if (n == 0)
3     {
4         return 0;
5     }
6     else if (n == 1)
7     {
8         return 1;
9     }
10    else
11    {
12        return fib(n-1) + fib(n-2);
13    }
14 }
```

- Think about how this executes in terms of caller/callee
 - The recursive call chases down a ‘rabbit hole’ to get to the base cases, and then starts to return values up to the initial caller, where n is the initial input.
- How would you write this function using a for loop?

```
1 int fib_loop(int n){
2     int first = 0, second = 1, next;
3     for (int i = 0 ; i <= n ; i++ )
4     {
5         if ( i <= 1 )
6         {
7             next = i;
8         }
9         else
10        {
11            next = first + second;
12            first = second;
13            second = next;
14        }
15    }
16    return next;
17 }
```

- Possible to write using a loop, but less clear, and farther away from the underlying math.

When to use recursion?

- Often, a problem can be solved using iteration or recursion.
- I would recommend using recursion when writing the iterative solution is overly complex, but sticking to using iteration by default.
- Recursion is actually more resource intensive than iteration, because when a function is called, it enters the *call stack* which requires memory to be allocated. So every recursive call uses memory, something that isn't guaranteed to happen with iterative solutions
 - So iteration can be more efficient
- On the other hand, some problems can be beautifully solved with recursion, but are hard to write or hard to read in an iterative fashion.
 - Fibonacci is the classic example; a recursive solution is much easier to read and think about than the iterative version.

Exercises

1. Write a recursive function that computes the sum of all numbers from 1 to n, where n is given as parameter.

```

1  #include<iostream>
2
3  using namespace std;
4
5  int sum_of_range(int);
6
7  int main()
8  {
9      int n;
10     int sum;
11
12     cout << "Input the last number of the range starting from 1: ";
13     cin >> n;
14
15     sum = sum_of_range(n);
16     cout << "The sum of numbers from 1 to " << n << " : " << sum << endl;
17
18     return 0;
19 }
20
21 int sum_of_range(int n)
22 {

```

```
23     if (n == 1)
24     {
25         return 1;
26     }
27     else
28     {
29         return n + sum_of_range(n - 1);
30     }
31 }
```

2. Write a program in C to count the digits of a given number using recursion

```
1  #include <iostream>
2
3  using namespace std;
4
5  int num_digits(int n, int count);
6
7  int main()
8  {
9      int n, count = 0;
10     cout << "Input a number: ";
11     cin >> n;
12
13     count = num_digits(n, count);
14
15     cout << "The number of digits in the number is : " << count << endl;
16     return 0;
17 }
18
19 int num_digits(int n){
20     if (n < 10)
21     {
22         return 1;
23     }
24     else
25     {
26         return 1 + num_digits(n/10);
27     }
28 }
```

3. Write a program in C to convert a decimal number to a binary number using recursion.
-

```
1  #include <iostream>
2
3  using namespace std;
4
5  long convert_to_binary(int decimal, long binary, long factor);
6
7  int main()
8  {
9      long binary = 0;
10     int decimal;
11
12     cout << "Input any decimal number: ";
13     cin >> decimal;
14
15     // seed a binary value of 0 and a factor of 1
16     binary = convert_to_binary(decimal, 0, 1);
17     cout << "The Binary value of decimal number " << decimal << " is: "
18          << binary << endl;
19     return 0;
20 }
21 long convert_to_binary(int decimal, long binary, long factor)
22 {
23     long binary_digit;
24
25     if (decimal == 0)
26     {
27         return binary;
28     }
29     else
30     {
31         binary_digit = decimal % 2;
32         binary = binary + binary_digit * factor;
33         factor = factor * 10;
34         return convert_to_binary(decimal / 2, binary, factor);
35     }
```