

7.11 — Recursion

BY ALEX ON AUGUST 13TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

A **recursive function** in C++ is a function that calls itself. Here is an example of a poorly-written recursive function:

```
1  #include <iostream>
2
3  void countDown(int count)
4  {
5      std::cout << "push " << count << '\n';
6      countDown(count-1); // countDown() calls itself recursively
7  }
8
9  int main()
10 {
11     countDown(5);
12
13     return 0;
14 }
```

When `countDown(5)` is called, “push 5” is printed, and `countDown(4)` is called. `countDown(4)` prints “push 4” and calls `countDown(3)`. `countDown(3)` prints “push 3” and calls `countDown(2)`. The sequence of `countDown(n)` calling `countDown(n-1)` is repeated indefinitely, effectively forming the recursive equivalent of an infinite loop.

In lesson [7.9 – The stack and the heap](#), you learned that every function call causes data to be placed on the call stack. Because the `countDown()` function never returns (it just calls `countDown()` again), this information is never being popped off the stack! Consequently, at some point, the computer will run out of stack memory, stack overflow will result, and the program will crash or terminate. On the author’s machine, this program counted down to -11732 before terminating!

Recursive termination conditions

Recursive function calls generally work just like normal function calls. However, the program above illustrates the most important difference with recursive functions: you must include a recursive termination condition, or they will run “forever” (actually, until the call stack runs out of memory). A **recursive termination** is a condition that, when met, will cause the recursive function to stop calling itself.

Recursive termination generally involves using an if statement. Here is our function redesigned with a termination condition (and some extra output):

```
1  #include <iostream>
2
3  void countDown(int count)
4  {
5      std::cout << "push " << count << '\n';
6
7      if (count > 1) // termination condition
8          countDown(count-1);
9
10     std::cout << "pop " << count << '\n';
11 }
12
13 int main()
14 {
15     countDown(5);
16     return 0;
17 }
```

Now when we run our program, `countDown()` will start by outputting the following:

```
push 5
push 4
push 3
push 2
push 1
```

If you were to look at the call stack at this point, you would see the following:

```
countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
main()
```

Because of the termination condition, `countDown(1)` does not call `countDown(0)` -- instead, the "if statement" does not execute, so it prints "pop 1" and then terminates. At this point, `countDown(1)` is popped off the stack, and control returns to `countDown(2)`. `countDown(2)` resumes execution at the point after `countDown(1)` was called, so it prints "pop 2" and then terminates. The recursive function calls get subsequently popped off the stack until all instances of `countDown` have been removed.

Thus, this program in total outputs:

```
push 5
push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4
pop 5
```

It's worth noting that the "push" outputs happen in forward order since they occur before the recursive function call. The "pop" outputs occur in reverse order because they occur after the recursive function call, as the functions are being popped off the stack (which happens in the reverse order that they were put on).

A more useful example

Now that we've discussed the basic mechanics of recursive function calls, let's take a look at another recursive function that is slightly more typical:

```
1 // return the sum of all the integers between 1 (inclusive) and sumto (inclusive)
2 // returns 0 for negative numbers
3 int sumTo(int sumto)
4 {
5     if (sumto <= 0)
6         return 0; // base case (termination condition) when user passed in an unexpected param
7     else if (sumto == 1)
8         return 1; // normal base case (termination condition)
9     else
10        return sumTo(sumto - 1) + sumto; // recursive function call
11 }
```

Recursive programs are often hard to figure out just by looking at them. It's often instructive to see what happens when we call a recursive function with a particular value. So let's see what happens when we call this function with parameter `sumto = 5`.

```
sumTo(5) called, 5 <= 1 is false, so we return sumTo(4) + 5.
sumTo(4) called, 4 <= 1 is false, so we return sumTo(3) + 4.
sumTo(3) called, 3 <= 1 is false, so we return sumTo(2) + 3.
sumTo(2) called, 2 <= 1 is false, so we return sumTo(1) + 2.
sumTo(1) called, 1 <= 1 is true, so we return 1. This is the termination condition.
```

Now we unwind the call stack (popping each function off the call stack as it returns):

```
sumTo(1) returns 1.
sumTo(2) returns sumTo(1) + 2, which is 1 + 2 = 3.
sumTo(3) returns sumTo(2) + 3, which is 3 + 3 = 6.
sumTo(4) returns sumTo(3) + 4, which is 6 + 4 = 10.
sumTo(5) returns sumTo(4) + 5, which is 10 + 5 = 15.
```

At this point, it's easier to see that we're adding numbers between 1 and the value passed in (both inclusive).

Because recursive functions can be hard to understand by looking at them, good comments are particularly important.

Note that in the above code, we recurse with value `sumto - 1` rather than `--sumto`. We do this to this because operator`--` has a side effect, and using a variable that has a side effect applied more than once in a given expression will result in undefined behavior. Using `sumto - 1` avoids side effects, making `sumto` safe to use more than once in the expression.

Recursive algorithms

Recursive functions typically solve a problem by first finding the solution to a subset of the problem (recursively), and then modifying that sub-solution to get to a solution. In the above algorithm, `sumTo(value)` first solves `sumTo(value-1)`, and then adds the value of variable `value` to find the solution for `sumTo(value)`.

In many recursive algorithms, some inputs produce trivial outputs. For example, `sumTo(1)` has the trivial output 1 (you can calculate this in your head), and does not benefit from further recursion. Inputs for which an algorithm trivially produces an output is called a **base case**. Base cases act as termination conditions for the algorithm. Base cases can often be identified by considering the output for an input of 0, 1, "", "", or null.

Fibonacci numbers

One of the most famous mathematical recursive algorithms is the Fibonacci sequence. Fibonacci sequences appear in many places in nature, such as branching of trees, the spiral of shells, the fruitlets of a pineapple, an uncurling fern frond, and the arrangement of a pine cone.

Here is a picture of a Fibonacci spiral:



Each of the Fibonacci numbers is the length of the side of the square that the number appears in.

Fibonacci numbers are defined mathematically as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases}$$

Consequently, it's rather simple to write a (not very efficient) recursive function to calculate the nth Fibonacci number:

```

1  #include <iostream>
2
3  int fibonacci(int count)
4  {
5      if (count == 0)
6          return 0; // base case (termination condition)
7      if (count == 1)
8          return 1; // base case (termination condition)
9      return fibonacci(count-1) + fibonacci(count-2);
10 }
11
12 // And a main program to display the first 13 Fibonacci numbers
13 int main()
14 {
15     for (int count=0; count < 13; ++count)
16         std::cout << fibonacci(count) << " ";
17
18     return 0;
19 }
```

Running the program produces the following result:

```
0 1 1 2 3 5 8 13 21 34 55 89 144
```

Which you will note are exactly the numbers that appear in the Fibonacci spiral diagram.

Memoization algorithms

The above recursive Fibonacci algorithm isn't very efficient, in part because each call to a Fibonacci non-base case results in two more Fibonacci calls. This produces an exponential number of function calls (in fact, the above example calls `fibonacci()` 1205 times!). There are techniques that can be used to reduce the number of calls necessary. One technique, called **memoization**, caches the results of expensive function calls so the result can be returned when the same input occurs again.

Here's a memoized version of the recursive Fibonacci algorithm:

```

1  #include <iostream>
2  #include <vector>
3
4  // h/t to potterman28wxcv for a variant of this code
5  int fibonacci(int count)
6  {
7      // We'll use a static std::vector to cache calculated results
8      static std::vector<int> results{ 0, 1 };
9
10     // If we've already seen this count, then use the cache'd result
11     if (count < static_cast<int>(std::size(results)))
12         return results[count];
13     else
14     {
15         // Otherwise calculate the new result and add it
16         results.push_back(fibonacci(count - 1) + fibonacci(count - 2));
17         return results[count];
18     }
19 }
20
21 // And a main program to display the first 13 Fibonacci numbers
22 int main()
```

```

23 {
24     for (int count = 0; count < 13; ++count)
25         std::cout << fibonacci(count) << " ";
26
27     return 0;
28 }

```

This memoized version makes 35 function calls, which is much better than the 1205 of the original algorithm.

Recursive vs iterative

One question that is often asked about recursive functions is, "Why use a recursive function if you can do many of the same tasks iteratively (using a *for loop* or *while loop*)?". It turns out that you can always solve a recursive problem iteratively -- however, for non-trivial problems, the recursive version is often much simpler to write (and read). For example, while it's possible to write the Fibonacci function iteratively, it's a little more difficult! (Try it!)

Iterative functions (those using a for-loop or while-loop) are almost always more efficient than their recursive counterparts. This is because every time you call a function there is some amount of overhead that takes place in pushing and popping stack frames. Iterative functions avoid this overhead.

That's not to say iterative functions are always a better choice. Sometimes the recursive implementation of a function is so much cleaner and easier to follow that incurring a little extra overhead is more than worth it for the benefit in maintainability, particularly if the algorithm doesn't need to recurse too many times to find a solution.

In general, recursion is a good choice when most of the following are true:

- The recursive code is much simpler to implement.
- The recursion depth can be limited (e.g. there's no way to provide an input that will cause it to recurse down 100,000 levels).
- The iterative version of the algorithm requires managing a stack of data.
- This isn't a performance-critical section of code.

However, if the recursive algorithm is simpler to implement, it may make sense to start recursively and then optimize to an iterative algorithm later.

Rule: Generally favor iteration over recursion, except when recursion really makes sense.

Quiz time

1) A factorial of an integer N (written N!) is defined as the product (multiplication) of all the numbers between 1 and N (0! = 1). Write a recursive function called factorial that returns the factorial of the input. Test it with the first 7 factorials.

Hint: Remember that $(x * y) = (y * x)$, so the product of all the numbers between 1 and N is the same as the product of all the numbers between N and 1.

Show Solution

2) Write a recursive function that takes an integer as input and returns the sum of each individual digit in the integer (e.g. $357 = 3 + 5 + 7 = 15$). Print the answer for input 93427 (which is 25). Assume the input values are positive.

Show Solution

3a) This one is slightly trickier. Write a program that asks the user to enter a positive integer, and then uses a recursive function to print out the binary representation for that number. Use method 1 from lesson **0.4 -- Converting between binary and decimal**.

Hint: Using method 1, we want to print the bits from the "bottom up", which means in reverse order. This means your print statement should be `_after_` the recursive call.

3b) Update your code from 3a to handle the case where the user may enter 0 or a negative number.

```
Enter an integer: -15  
11111111111111111111111111110001
```

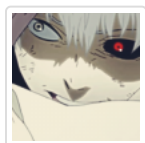
Show Solution



7.10 -- std::vector capacity and stack behavior

276 comments to 7.11 — Recursion

[« Older Comments](#) [1](#) [2](#) [3](#) [4](#)



Mn3m
February 1, 2020 at 3:57 am · Reply.
2)

```
1  #include <iostream>
2
3
4
5  int divAndSum(int number)
6  {
7
8      if (number == 0)
9          return 0;
10     else
11     {
12         int remainder = number % 10;
13         return divAndSum(number / 10) + remainder;
14     }
15
16
17 }
18
19
20
21 int main()
22 {
23
24
25
26     std::cout<<divAndSum(93427);
27
28
29
30     return 0;
31 }
```



cnoob

December 21, 2019 at 6:07 am · Reply

Hi! I wanted to solve the factorial function without having to use any iteration. Is this a legit way to solve it?:

```
1  #include <iostream>
2
3  int factorial(int n)
4  {
5      static int factVal{ 1 };
6      if (n < 1)
7          return 1;
8      else
9      {
10         factVal = factorial(n - 1) * factVal;
11         std::cout << factVal << '\n';
12         return (n);
13     }
14 }
15
16 int main()
17 {
18     factorial(7);
19 }
```

I made an iterative version for the fibonacci numbers too:

```
1  #include <iostream>
2
```

```

3  void fibonacci()
4  {
5      static int fibo{ 1 };
6      static int tempVal{ 0 };
7      static int tempVal2{};
8
9      std::cout << fibo << '\n';
10     tempVal2 = fibo;
11     fibo = fibo + tempVal;
12     tempVal = tempVal2;
13 }
14
15 int main()
16 {
17     for (int count{ 0 }; count < 7; count++)
18         fibonacci();
19 }

```



nascar driver

December 21, 2019 at 6:18 am · Reply

The solution used a loop to test the function. The function itself shouldn't print anything. Yours doesn't work, because you used a `static` variable. Try calling `factorial` again and it will produce wrong results.

Same problem in `fibonacci`. Use pure function if possible (A function called with the same arguments produces the same behavior every time). Add a parameter to `fibonacci` which is the index of the number in the fibonacci sequence. Remove `static` and move the loop into `fibonacci`.



cnoob

December 21, 2019 at 9:45 am · Reply

Yes, my bad! So I suppose the fibonacci should look something like this:

```

1  #include <iostream>
2
3  void fibonacci(int index)
4  {
5      int fibo{ 1 };
6      int tempVal{ 0 };
7      int tempVal2{};
8
9      for (int count{ 0 }; count < index; count++)
10     {
11         std::cout << fibo << '\n';
12         tempVal2 = fibo;
13         fibo += tempVal;
14         tempVal = tempVal2;
15     }
16 }
17
18
19 int main()
20 {
21     fibonacci(7);
22     fibonacci(5);
23 }

```


You were right in the case of the factorials too. I still can not understand why does this work though:

```

1  #include <iostream>
2  namespace Fact
3  {
4      int factVal{ 1 };
5  }
6
7  int factorial(int n)
8  {
9      Fact::factVal = 1;
10     if (n < 1)
11         return 1;
12     else
13     {
14         Fact::factVal = factorial(n - 1) * Fact::factVal;
15         std::cout << Fact::factVal << '\n';
16         return (n);
17     }
18 }
19
20 int main()
21 {
22     factorial(7);
23     factorial(5);
24 }
```

Shouldn't "Fact::factVal = 1;" set the value of Fact::factVal back to 1 every time the function is called? And shouldn't the function just print the value of n then?



nascar driver

December 22, 2019 at 1:59 am · Reply

You don't need, and shouldn't use, any global or `static` variables.

Let's run your `factorial` manually. The letters indicate which call (depth) we're currently in.

```

1  (a) factorial(3);
2  (a) Fact::factVal = 1
3  (a) Fact::factVal = factorial(2) * Fact::factVal
4  (b) Fact::factVal = 1
5  (b) Fact::factVal = factorial(1) * Fact::factVal
6  (c) Fact::factVal = 1
7  (c) Fact::factVal = factorial(0) * Fact::factVal
8  (d) Fact::factVal = 1
9  // Now (n < 1) is true, the recursive calls return.
10 (c) Fact::factVal = 1 * Fact::factVal
11 (c) Prints Fact::factVal, which is 1
12 (c) return 1
13 (b) Fact::factVal = 1 * Fact::factVal
14 (b) Prints Fact::factVal, which is 1
15 (b) return 2
16 (a) Fact::factVal = 2 * Fact::factVal
17 (a) Prints Fact::factVal, which is 2
18 (a) return 3
```

cnoob

December 22, 2019 at 4:56 am · Reply



Yes, I understand it now! Thanks! I know that the globals are not necessary here (and in general), I just couldn't wrap my head around why it worked. But I get it now, thanks again!



Ged

December 11, 2019 at 2:37 am · Reply

Enter an integer: -15

111111111111111111111111110001

My program works, but I have a question.

For example, here's how we represent -5 in binary two's complement:

First we figure out the binary representation for 5: 0000 0101

Then we invert all of the bits: 1111 1010

Then we add 1: 1111 1011

15 binary: 0000 1111

We invert it: 1111 0000

Then we add 1: 1111 0001

The ending is correct but there are plenty of extra 1's. It's because our number starts from (unsigned int maximum number - 15). So my question is do we actually need to discard all the 1's and only leave 8 at the end?



nascardriver

December 11, 2019 at 5:51 am · Reply

If you want to display only 8 bits, then you need to discard the first 24 (Assuming 32-bit integers).



Jack Overby

November 15, 2019 at 4:36 pm · Reply

Not sure if anyone has already come up with a similar solution, but here's a better way to handle negative numbers, that avoids unsigned int overflow:

```

1  void printBinary(int n)
2  {
3      if (n > 1)
4          printBinary(n / 2);
5      else if (n < -1)
6      {
7          std::cout << "-";
8          n = -n;
9          printBinary(n / 2);
10     }
11     std::cout << n % 2;
12 }
13
14 int main()
15 {
16     int x;
17     std::cout << "Enter an integer: ";
18     std::cin >> x;
19
20     printBinary(x);

```

21 | }

Works as you'd hope for both negative #s and 0!



teddye

November 13, 2019 at 5:39 am · Reply

"It turns out that you can always solve a recursive problem iteratively"

For the sake of being pedantic I'll point you towards the Ackermann Function, not particularly useful for anything, but truly can only be solved with recursion. Technically you can code it without calling the function within itself, but it involves dynamically allocating a stack and treating it like a call stack, so it's recursion by any other name.

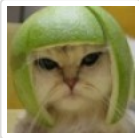


Parsa

September 26, 2019 at 4:24 pm · Reply

Can you make a lesson on returning functions? Does return have a temporary variable?

I am super confused right now



Alex

September 30, 2019 at 8:50 am · Reply

What do you mean by "returning functions"? Functions that return other functions? The quiz in lesson 7.8 shows an example of this.



potterman28wxcv

August 17, 2019 at 1:44 pm · Reply

Hello,

I was surprised at the Fibonacci example, especially since the implementation you give is the most naive one, of exponential complexity. This could induce a beginner into thinking their iterative version is much faster than the recursive one - when in fact, the recursive implementation you gave is algorithmically awful in terms of complexity (exponential complexity).

It would be much better (and not much more complicated) to use memoization such as this code:

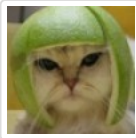
```
1  long fibonacci(int number)
2  {
3      static std::vector<int> results{0, 1};
4
5      if (number < static_cast<int>(std::size(results)))
6          return results[number];
7      else{
8          results.push_back(-1);
9          return (results[number] = fibonacci(number-1) + fibonacci(number-2));
10     }
11 }
```

(the .push_back part could be done a bit better, but it's just an example)

And indeed, if I try to run your example on fibonacci(10000) it runs on and on without finishing because of the exponential ; while my code seems to be much faster, below a second of execution.

(it is a bit silly i know to run fibonacci(10000) when a 64 bits integer isn't enough to contain it, but it's just to show the runtime complexity)

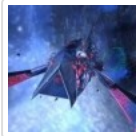
Cheers



Alex

[August 18, 2019 at 3:52 pm · Reply](#)

The lesson is primarily focused on C++ mechanics, not so much on algorithm design. Regardless, since you provided some code, I added a subsection to the lesson. Thanks for bringing this up!

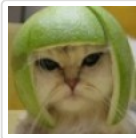


potterman28wxcv

[August 21, 2019 at 11:27 am · Reply](#)

Hey, sorry for the delay in the reply - i thought i would have been notified of replies by mail but that's apparently not the case, so actually i didn't check the comments much for further replies.

Thanks a lot for adding this subsection, it is very well explained and smooth, as usual! :)



Alex

[August 21, 2019 at 8:43 pm · Reply](#)

You should get replies by email when someone responds to your comment, assuming you left a valid email address. Check your spam box.



potterman28wxcv

[August 22, 2019 at 3:41 am · Reply](#)

You are right, it was all in the spam.

Thanks!



Jonathan

[August 11, 2019 at 1:20 am · Reply](#)

My solution is slightly different from the sample solution for the binary question, seems like it works for negative and positive integers and zero.

```
1  #include <iostream>
2
3  int recursiveBinary(int n)
4  {
5      if (n == 0)
6          return 0;
7      if (n == 1)
8          return 1;
9      else
10         return (10 * recursiveBinary(n / 2) + (n % 2));
11 }
12 int main()
13 {
14     std::cout << "Enter an integer: ";
15     int n{ 0 };
16     std::cin >> n;
17
18     int nBinary{ recursiveBinary(n) };
```

```

19     std::cout << "The binary of " << n << " is " << nBinary;
20
21     return 0;
22 }

```



Caterpillar

[August 11, 2019 at 5:40 am · Reply.](#)

Check again, it does not work for negative numbers, nevertheless it's an interesting approach, but still, not very accurate for the task.



Jonathan

[August 11, 2019 at 7:36 am · Reply.](#)

true, I thought that for negative binary just add a negative sign in front of the binary...

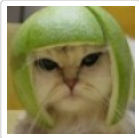


Nirbhay

[August 5, 2019 at 1:37 am · Reply.](#)

Typo:

Question 3 a) This one is slightly trickier. Write a program that asks the user to enter "an"(should be "a") positive integer.



Alex

[August 6, 2019 at 6:35 pm · Reply.](#)

Typo fixed. Thanks!



sasan.sn

[June 10, 2019 at 1:24 am · Reply.](#)

thank you.



sasan.sn

[June 10, 2019 at 12:24 am · Reply.](#)

hello!

unfortunately i can't use 8 bytes (64bit) even by long long int variable! why?

thank you for your answers.

```

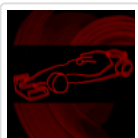
1  #include <iostream>
2  #include <string>
3
4  std::string binaryConv(long long int v)
5  {
6  /*
7
8      to convert negative numbers i do as follows:
9      1- conver negative to positive -> v * -1 or -v
10     2- Create a mask for invert all of the bits -> (65536LL * 65536 - 1)
11     3- use exclusive or to invert number -> -v ^ (65536LL * 65536 - 1)
12     4- at last add 1 to result.
13     5- now i have a positive number that in binary is equivalent of my negative number.
14     sorry if i have bad grammar or dictation.

```

```

14  */
15  if(v<0)
16      v = ((v * (-1)) ^ (65536LL * 65536 - 1)) + 1;
17
18  if(v == 0)
19      return "";
20
21  return binaryConv(v / 2) + ((v % 2 == 0)? "0" : "1");
22  }
23
24  int main()
25  {
26      long long int intVar {1};
27      int fixLen{0};
28      std::string myBinaryString;
29
30      while(intVar!=0)
31      {
32          std::cout <<  "\n\nEnter an integer number to convert to binary"
33                        "\nEnter 0 to exit: ";
34          std::cin >> intVar;
35          std::cout << "\n\n";
36          myBinaryString = binaryConv(intVar);
37          fixLen = 5 - (myBinaryString.length() % 4);
38          for(unsigned int i=0; i < myBinaryString.length(); i++)
39          {
40              std::cout << myBinaryString.at(i);
41              if((i+fixLen)%4 == 0)
42                  std::cout << " ";
43          }
44          std::cout << "\n\n";
45      }
46
47      system("pause");
48      return 0;
49  }

```



nascar driver

June 10, 2019 at 1:01 am · Reply

- Limit your lines to 80 characters in length for better readability on small displays.
- Don't use `system`, it won't work on other platforms.
- Use ++prefix unless you need postfix++.
- Initialize your variables with brace initializers.

The size of built-in numeric types is implementation-defined (with some restrictions). If you need a guaranteed size, use `std::int_least64_t` or `std::int_fast64_t`. See lesson 4.6 for more information. You can flip bits using the `~` operator.

```

1  char ch{ 0b0011'1010 };
2  ch = ~ch;
3  // ch:  0b1100'0101

```



عمرو رياض الجنيد

June 8, 2019 at 7:41 am · Reply

I want to solve marital and individual numbers using self-recall

**masterOfNothing**May 23, 2019 at 9:16 am · Reply

I just couldn't solve the second quiz question.

Here's the 3A quiz question. Had to do it the hard way:

```

1  void printBinary(int x)
2  {
3      if (float(x)/2 < 1)
4      {
5          std::cout << x%2;
6          return;
7      }
8      else
9          printBinary(x/2);
10
11     std::cout << x%2;
12 }
13
14 int main()
15 {
16     std::cout << "Enter a positive integer: ";
17     int x;
18     std::cin >> x;
19     printBinary(x);
20     return 0;
21 }
```

**nascardriver**May 23, 2019 at 9:20 am · Reply

- * Don't use C-style casts.
- * Use float literals for floats.
- * Use your editor's auto-formatting feature.
- * Missing printed trailing line feed (Print a line feed when you program is done).

Line 3 can be simplified by math

```

1  x/2 < 1
2  // multiply each side by 2
3  x < 2
```

**masterOfNothing**May 28, 2019 at 8:39 am · Reply

Yes, the c-style casts... They seem so less complicated and so conveniently similar to how it's done in python. I did it due to laziness, I guess.

The f suffix?

I did use it, but for some reason the code I copied in kept copying with wrong indentations. So I fixed what I could in the comment section.

Good catch! (3rd line)

nascardriverMay 28, 2019 at 8:41 am · Reply



> The f suffix?
Indeed.



X
[April 6, 2019 at 2:04 am · Reply](#)

How important is it to learn or know Recursion? Is it used often in the workplace?

Asking because I understand the concept, I can follow the code of one (using hand-written diagrams) but creating one, as of right now, is something I can't do and I was wondering if I should spend more time on it now and learn it (if it is important) or if I can safely "skip" it and come back at some other point in time.



nascardriver
[April 6, 2019 at 2:24 am · Reply](#)

Skip it, recursion should be avoided.



X
[April 7, 2019 at 6:24 am · Reply](#)

Alright, thanks!



NXPY
[March 23, 2019 at 5:08 am · Reply](#)

Hey there guys !

How important is the concept of dynamic programming here ? Can it help reduce time complexity for the Fibonacci sequence to that of an iterative Fibonacci ?

(Note: I'm using Fibonacci as it's the only example I know correctly . If there are other examples I'll be grateful to know them)



nascardriver
[March 23, 2019 at 9:00 am · Reply](#)

> How important is the concept of dynamic programming here

Dynamic programming is nice, but we have iterative code, which is nicer. Dynamic programming is important in functional languages, where everything is done with functions, not variables.

> Can it help reduce time complexity for the Fibonacci sequence to that of an iterative Fibonacci ?

No. Dynamic programming speeds up recursion, but it can't be as fast as an iterative solution.

The slow part about recursion is that all local variables have to be stored while the recursive sub-call is running. For every recursive call, all local variables (and some extra information) have to be stored. With an iterative solution, every variable exists once.



NXPY
[March 23, 2019 at 11:09 pm · Reply](#)

Thanks for the reply !
Good thing I checked that doubt here .



NXPY
March 23, 2019 at 4:39 am · Reply.

What happens if we make a recursive function inline ?



nascardriver
March 23, 2019 at 8:52 am · Reply.

Hi!

The "inline" keyword is merely a suggestion to the compiler that you'd like this function to be inlined during compilation. The compiler is allowed to ignore your request and it's allowed to inline function which you didn't declare inline.

If you try to force the compiler to inline a function by using compiler-specific techniques, it might repeat the code over and over, or it aborts compilation.

clang++'s output:

```
1 | error: inlining failed in call to always_inline 'int fn(int)': function not considered
```




NXPY
March 23, 2019 at 9:06 am · Reply

Now I understand . Thanks for the reply !



Arthur
March 16, 2019 at 8:56 am · Reply.

 interesting. I have heard of the Fibonacci sequence but never seen it drawn as the Fibonacci sequence I have only seen that drawn as 'The Golden Mean' which is used in art. in photographic composition it is the 'rule of thirds'.



Demetrius
February 25, 2019 at 7:14 am · Reply.

Best explanation of recursive functions I have ever seen. Many thanks to the authors



Yiu Chung WONG
January 7, 2019 at 1:16 am · Reply

For 3b, so the code is essentially turning the negative integer into a really big positive integer, and then start dividing this big integer by 2 recursively? Is this where all the ones come from?

[illegible]

nascardriver
January 7, 2019 at 7:37 am · Reply



Hi!

> turning the negative integer into a really big positive integer [...]

Yep

> I'm also confused by the ones in front

The two binary numbers are the same, they're just using a different amount of bits. Since two's complement (Lesson 3.7) will swap all bits, all 1s will turn into 0s.



ryder

December 20, 2018 at 12:04 pm · Reply.

Just want to share with anyone who interested in recursion:

I found a video very intuitive to show what's recursion: <https://www.youtube.com/watch?v=2SUvWfNJSSM&feature=youtu.be>



Piyush

April 27, 2019 at 11:46 pm · Reply.

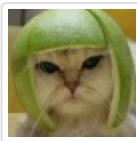
Thats great.



kapsch

December 5, 2018 at 4:57 am · Reply.

Hi there, just like you guys pointed out in one of the comments, the presented solution for 3a) won't print anything for input 0.



Alex

December 6, 2018 at 11:07 am · Reply.

3a said to assume the user entered a positive integer (which 0 is not). I've updated the question text to make this clearer.



kapsch

December 7, 2018 at 12:07 am · Reply.

My bad, sorry.



Michael Stef

September 6, 2018 at 3:02 am · Reply.

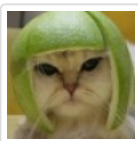
I tried it and really as u said the recursive solution is easier to design , understand and makes sense more, but

the speed decreased a lot when i increased the count to 40 in the recursive solution vs the iterative one, A question is there a way stop the recursion from making stack overflow just before stack runs out, by signaling that the function could not found a solution to the problem ?

```
1  int Fib(int N)
2  {
3      if (N == 0)
4      {
```

```

5         return 0;
6     }
7     else if (N == 1)
8     {
9         return 1;
10    }
11    else
12    {
13        int rv;
14        int x = 1;
15        int y = 0;
16
17        while (N > 1)
18        {
19            rv = x + y;
20            y = x;
21            x = rv;
22
23            --N;
24        }
25
26        return rv;
27    }
28 }
29
30 int main()
31 {
32
33     for (int count = 0; count < 40; ++count)
34         std::cout << Fibonacci(count) << " ";
35
36     std::cout << "\n\n";
37
38     for (int count = 0; count < 40; ++count)
39         std::cout << Fib(count) << " ";
40
41     std::cout << "\n\n";
42
43     return 0;
44 }
```



Alex

[September 6, 2018 at 2:51 pm · Reply.](#)

I'm not aware of any good way to determine when the stack is about to overflow.

The best thing to do here would be to specify a max recursion depth parameter, and either throw an exception or return some sentinel value if the answer can't be determined within that amount of depth.



Nigel Booth

[August 28, 2018 at 11:48 pm · Reply.](#)

#3a I suppose would be a bit like this:

```

1 // printBinary.cpp
2 //
3
4 #include "pch.h"
5 #include <iostream>
6
```

```

7  void printBinary(int x)
8  {
9      if (x == 0) //0 is 0 so this would be the terminator
10         return;
11     printBinary(x / 2); //divide each result by 2 and move on to the next
12     std::cout << x % 2; //and use the modulus to find and print the remainder.
13 }
14
15 int main()
16 {
17     int x{ 0 }; //declare int x here
18     std::cout << "Enter an integer: ";
19     std::cin >> x; //and get a value for x from the user.
20     std::cout << "The binary equivalent of " << x << " is: \n";
21     printBinary(x); //Then call recursive function to convert x to binary equivalent
22 }

```

and to accept a negative integer (3b) I would just change the call in the function declaration from

```
1 void printBinary(int x)
```

to

```
1 void printBinary(unsigned int x)
```

. I suppose this is correct?



nascardriver

August 29, 2018 at 7:07 am · Reply.

Hi Nigel!

* @main is missing a return value.

* @printBinary won't print anything for input 0.

> I suppose this is correct?

It is

[« Older Comments](#)

[1](#) [2](#) [3](#) [4](#)