29. Recursion

Objectives

- Write a recursive function
- Trace a recursive function by hand

In this set of notes, you will see that I'm not going to introduce new C/C++ syntax. Instead we will be using recursion to solve problems. When used appropriately, recursion sometimes results in shorter code. It's also easier to check that a recursive problem works correctly than one that is not recursive.

For those of you who are mathematically inclined (read: math geeks), the foundation of recursion is the principle of mathematical induction. This means that if you have a recursive program, to prove that the program is correct you have to prove the correctness of the code using mathematical induction.

You will see a curious thing: many of the code snippets that require loops (for-loop or while-loop) can be rewritten without these loops using recursion! In fact some programming languages that support recursion do not have loops at all!

What is recursion?

A **recursive function** is easy to define: It's a function that calls itself. (There's another kind of recursion called recursive data structure.)

Here's an example:

```
int f(int x)
{
    std::cout << "entering f() with x = " << x
              << std::endl;
    if (x == 0)
    {
        return 0;
    }
    else
    {
        return f(x - 1);
    }
}


int main()
{
    int x = f(3);
    std::cout << "main() ... x = " << x << std::endl;
    return 0;
}
```

We say that `f()` is a **recursive function**.

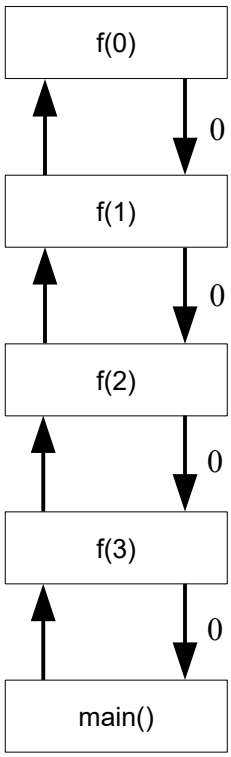
Let's trace the program slowly. The difficulty is that `f()` is being called several times.

`main()` calls `f(3)`
`f(3)` calls `f(3 - 1)`, i.e., `f(2)`
`f(2)` calls `f(1)`
`f(1)` calls `f(0)`

Note that

`f(0)` returns 0 to `f(1)`
`f(1)` receives 0 and return it to `f(2)`
`f(2)` receives 0 and return it to `f(3)`
`f(3)` receives 0 and return it to `main()`

Sometimes it's helpful to visual this stack of function calls:



Our C++ function `f()`:

```
int f(int x)
{
    if (x == 0)
    {
        return 0;
    }
    else
    {
        return f(x - 1);
    }
}
```

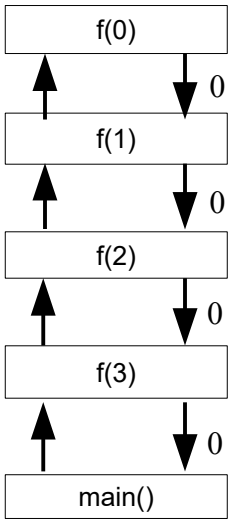
is written mathematically like this:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1) & \text{if } x > 0 \end{cases}$$

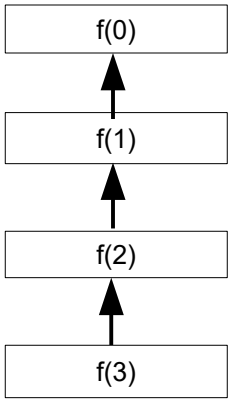
Now let's do a recursion that does something useful ...

Function call graph

The diagram above that describes function call:



is called a **function call graph**, Usually a function call graph is used to describe the calling, so the return path is usually not shown. If I'm only interested in the recursive calls say of f(3), then I would not include the box for main(). So the function call graph for f(3) looks like this:



is important because, if there are no loops in the function, then the number of boxes in the function call graph more or less denotes the amount of work that has to be done to compute f(3). For instance, omitting the box for `main()`, f(3) involves 4 boxes.

Sum from 1 to n

Suppose we write

sum(n)

to denote “the sum of all integers from 0 to n”, i.e.,

sum(n) = 0 + 1 + 2 + 3 + … + n

Of course we know how to sum from 1 to n without using recursion but using a loop:

```
int sum(int n)
{
    int s = 0;
    for (int i = 0; i < n; ++i)
    {
        s += i;
    }
    return s;
}
```

In fact the

…

in sum(n) = 1 + 2 + 3 + … + n more or less described a loop.

Let me think of the sum(n) computation recursively … let’s try some examples to get a feel for the problem. Here's an example:

sum(5) = 0 + 1 + 2 + 3 + 4 + 5

Note that

sum(4) = 0 + 1 + 2 + 3 + 4

Therefore you have this:

sum(5) = 0 + 1 + 2 + 3 + 4 + 5
= (0 + 1 + 2 + 3 + 4) + 5
= sum(4) + 5

Clearly we also have

sum(8) = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
= (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7) + 8
= sum(7) + 8

See you should quickly see that in general if n is a positive integer greater than 0, then

sum(n) = sum(n – 1) + n

You see that sum(n) is related to sum(n – 1). Also note that

$$sum(0) = 0$$

So if we define sum(n) to be "the sum of all integers from 0 to n", then we get

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

The relationship (when n > 0)

$$sum(n) = sum(n-1) + n$$

is called a **recursion** because the sum function called **itself**. The case when n = 0

$$sum(0) = 0$$

is called the **base case** of the recursion.

Recursion is an **extremely** important phenomenon in math, computer science, …, in nature. So listen up.

First of all, if I ask you to give me sum(5), you would say, well it's just

$$sum(5) = 1 + 2 + 3 + 4 + 5$$

and then you evaluate the express to get 15. But you can also use

$$sum(n) = \begin{cases} 0 & \text{if } n = 0 \quad (B) \\ sum(n-1) + n & \text{if } n > 0 \quad (R) \end{cases}$$

to get the same value. (Note that I've labeled the two cases.) Here's how you use (B) and (R) to compute sum(5):

Since 5 > 0, from (R) we get

$$sum(5) = sum(5 - 1) + 5$$

So??? Well if we simplify that we get

$$sum(5) = sum(4) + 5$$

That doesn't give us a value of sum(5). But wait, sum(4) can be written in a different way: Because 4 > 0, by (R):

$$sum(4) = sum(4 - 1) + 4$$

which simplifies to

sum(4) = sum(3) + 4

So altogether we have this:

sum(5) = sum(4) + 5
sum(4) = sum(3) + 4

That still does NOT tell us what the value of sum(5) is!!! But hang on, with the same reasoning (using (R)) we get three more facts:

sum(3) = sum(2) + 3
sum(2) = sum(1) + 2
sum(1) = sum(0) + 1

So altogether we have

sum(5) = sum(4) + 5
sum(4) = sum(3) + 4
sum(3) = sum(2) + 3
sum(2) = sum(1) + 2
sum(1) = sum(0) + 1

Now you might be tempted to write

sum(0) = sum(-1) + 0

That's wrong!!! Because the relationship (R)

(R): sum(n) = sum(n – 1) + n

applies only when n > 0. You cannot apply it to the case of n = 0. But wait … for the case of n = 0 we have this:

(B): sum(0) = 0

Therefore altogether we have

sum(5) = sum(4) + 5
sum(4) = sum(3) + 4
sum(3) = sum(2) + 3
sum(2) = sum(1) + 2
sum(1) = sum(0) + 1
sum(0) = 0

So what? Well, with these we can compute sum(5)!!! Let me show you how. First let me label the facts:

(5): sum(5) = sum(4) + 5
(4): sum(4) = sum(3) + 4
(3): sum(3) = sum(2) + 3
(2): sum(2) = sum(1) + 2
(1): sum(1) = sum(0) + 1
(0): sum(0) = 0

From equation (0), we know that sum(0) = 0. Substituting this into equation (1) we get

$$sum(1) = sum(0) + 1 = 0 + 1$$

And if we substitute this into equation (2) we get

$$sum(2) = sum(1) + 2 = 0 + 1+ 2$$

Substituting this into equation (3) we get

$$sum(3) = sum(2) + 3 = 0 + 1+ 2 + 3$$

And when this is put into equation (4) we get

$$sum(4) = sum(3) + 4 = 0 + 1+ 2 + 3 + 4$$

And of course finally we get

$$sum(5) = sum(4) + 5 = 0 + 1+ 2 + 3 + 4 + 5$$

So whether you define sum(n) as

$$sum(n) = 1 + 2 + \ldots + n$$

(which looks more like a loop) or

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

you get the same value for sum(n).

Exercise. Using the above method for evaluating a recursive function, compute for sum(7). You should get 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7.

The two ways of looking at sum(n), using a loop and using recursion, are gives us the same result. The really important question to ask is this:

Why should

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

be a good alternative view of

$$sum(n) = 0 + 1 + 2 + \ldots + n?$$

The reason is because of this … pay attention! … many problems in

nature present themselves naturally in a **recursive way**. For more complex problems, the recursive way is the way that leads to a simpler solution. And simple mean less bugs.

(For those of you who have taken Physics and Differential Equations, you know that many problems in Physics requires the description of a function or behavior of a physical phenomenon, but the function takes part in a relationship of a special form – the function appears in a differential equation. For CS, many problems gives rise to functions which are recursive.)

Of course we still need code … but don't worry: We can easily translate the recursive sum(n)

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

into a C++ function:

```
int sum(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else
    {
        return sum(n - 1) + n;
    }
}
```

Note that the **recursive version does not have loops**. Make sure you study and test it carefully.

Exercise. The following version includes print statements so that you know what the function is doing:

```
#include <iostream>

int sum(int n)
{
    std::cout << "entering sum(" << n << ") ...\n";
    if (n == 0)
    {
        std::cout << "exiting sum(" << n
                  << ") returning 0 ...\n";
        return 0;
    }
    else
    {
        int ret = sum(n - 1) + n;
```

```
        std::cout << "exiting sum(" << n
                  << ") returning " << ret
                  <<  "...\n";
        return ret;
    }
}


int main()
{
    sum(5);
    return 0;
}
```

Make sure you run the above. I've already mentioned this: putting print statements inside a function, recursive or not, helps in understanding the function and, if you have an error, helps in debugging the function.

In this case, the sum-from-1-to-n is easy so recursion is not really necessary. The point of the section is to show you that it's possible to write a recursion function for sum-from-1-to-n. In the real world many problems are described more naturally in a recursive form and solving such problems using loops is extremely untidy and painful.

Exercise. Draw the function call graph for sum(5). How many boxes do you see? How many boxes are there for the function call graph of sum(n)?

Exercise. Let f(n) denote the sum of squares $0^2 + 1^2 + 2^2 + ... + n^2$.

(a) Write down f(4) and f(3). What is the relationship between f(4) and f(3)?
(b) Write down f(5) and f(4). What is the relationship between f(5) and f(4)?
(c) In general if n > 0 is a positive integer, what is the (recursive) relationship between f(n) and f(n-1)?
(d) What is f(0)?

Write a C/C++ function for f(n). Write a `main()` to verify your f(n).

Exercise. Let `h(n)` be 2 to the n-th power. Consider h(5):

$$h(5) = 2^5$$

Do you see h(4) in h(5)? Write a recursive function for `h(n)`. Test it (of course).

Exercise. Continuing the above … Let `h(a, n)` be a to the n-th power. Consider h(3, 5):

$$h(3, 5) = 3^5$$

Do you see h(3, 4) in h(3, 5)? Write a recursive function for `h(a, n)`. Test it (of course).

Exercise. You are given the following mathematical description of a recursion:

$$k(n) = 2 + 3 * k(n - 1) \text{ if } n > 0$$
$$k(0) = 1$$

Compute by hand the value of k(1), k(2) and k(3). Write a C++ recursive function for k(n). Test your code. Make sure your computation by hand matches the output of your program.

Exercise. You are given the following mathematical description of a recursion:

$$j(n) = 1 + j(n - 1) * n \text{ if } n > 0$$
$$j(0) = 2$$

Compute by hand the value of j(1), j(2) and j(3). Write a C++ recursive function for j(n). Test your code. Make sure your computation by hand matches the output of your program.

Factorial

First let's just do math.

Let f(n) denote the factorial function. In other words f(3) is 3x2x1; f(5) is 5x4x3x2x1. In math the factorial of 3 is written 3!; the factorial of 5 is written 5!. This is not new – I have already talked about this long time ago.

First of all there is a **recursion** here. Look at f(n).

$$f(n) = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

Note that

$$f(n-1) = (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

Do you see f(n – 1) in f(n)? Putting the above together we have

$$\begin{aligned} f(n) &= n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \\ &= n \times \left[(n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1\right] \end{aligned}$$

in other words:

$$\mathbf{f(n) = n \times f(n-1)}$$

Right? You can use this mathematical fact to compute f(3) by hand:

$$f(3) = 3 \times f(2)$$
$$f(2) = 2 \times f(1)$$

WAIT!!! When do we stop?

$$f(1) = 1 \times f(0)$$
$$f(0) = 0 \times f(-1)$$
$$f(-1) = -1 \times f(-2)$$

Not good ... this will go on forever. But hang on. Look at this:

$$f(3) = 3 \times f(2)$$
$$f(2) = 2 \times f(1)$$
$$f(1) = 1 \times f(0)$$

This means

$$f(3) = 3 \times f(2) = 3 \times 2 \times f(1) = 3 \times 2 \times 1 \times f(0)$$

Now ... if we insist that f(0) is defined to be 1 we get

$$f(3) = 3 \times 2 \times 1 \times f(0) = 3 \times 2 \times 1 \times 1$$

So we have these properties for our factorial function:

f(n) = n x f(n-1), if n > 0
f(0) = 1

The fact

f(0) = 1

is called the _____ **case** of f(n) while

f(n) = n x f(n-1), if n > 0

is called the _____ **case**.

This is how you translate the above function:

f(n) = n x f(n-1), if n > 0
f(0) = 1

into C++ code:

```
int f(int n)
{
    if (n == 0)
        return 1;
    else
        return n * f(n - 1);
}


int main()
{
    for (int n = 0; n < 5; n++)
    {
        std::cout << n <<  "! = " << f(n)
                  << std::endl;
    }
    return 0;
}
```

/Of course there's another way (non-recursive way) of computing factorials … using loops …:

```
int f(int n)
{
    int p = 1;
    for (int i = 1; i <= n; i++)
    {
        p *= i;
    }
```

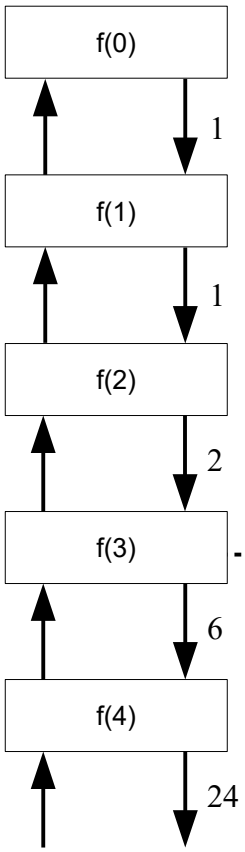
```
    return p;
}
```

Note that just like the sum(n) example the **recursive version does not have loops**.

To see the program trace its steps run this:

```
int f(int n)
{
    std::cout << "enter f(" << n << ")" << std::endl;
    if (n == 0)
    {
        std::cout << "base case ... returning 1"
                  << std::endl;
        return 1;
    }
    else
    {
        int x = n * f(n - 1);
        std::cout << "recursive case ... returning "
                  <<  x << std::endl;
        return x;
    }
}


int main()
{
    int x = f(5);
    std::cout << "main() ... " << x << std::endl;
    return 0;
}
```

Here's a diagram that might help:



In `f(3)`,

```
int f(int n)
{
 if (n == 0)
  return 1;
 else
  return n * f(n - 1);
}
```

On receiving `f(2)`, i.e., `2`, `f(3)` returns `n * f(n-1)`, i.e. `3 * 2`, i.e. `6`.

Notice that for the function call graph of (4), there are 5 boxes. You should see quickly that the function call graph for f(n) has n + 1 boxes.

How to discover recursion: look for smaller subproblems inside a problem

This is really important … !!!

Let's step back and look at the recursive version of the computation of "sum from 0 to n" function and the "factorial of n".

Here's the sum(n) (i.e., sum from 0 to n):

```
sum(0) = 0
sum(n) = n + sum(n – 1) if n > 0
```

and here's the factorial (which I'll just call f):

```
f(0) = 1
f(n) = n * f(n – 1) if n > 0
```

How does one recognize the recursive form of a function? The crucial thing is to see a smaller subproblem within a bigger problem. You should also look at specific concrete examples. This is what I mean …

Let's look at the sum(n) and say you don't know how to write down the recursive form yet. You try a concrete example, say the problem of computing sum(5). It looks like this:

```
sum(5) = 0 + 1 + 2 + 3 + 4 + 5
```

Next, you ask yourself if you can see another sum(n) within sum(5). You see immediately that sum(4) appears within sum(5):

```
sum(5) = 0 + 1 + 2 + 3 + 4 + 5
       = (0 + 1 + 2 + 3 + 4) + 5
sum(4) = 0 + 1 + 2 + 3 + 4
```

i.e.,

```
sum(5) = sum(4) + 5
```

You then try a few more examples and you should see that in general

```
sum(n) = sum(n – 1) + n
```

Of course this is true for n > 0. You are pretty much done except for sum(0). You then ask yourself what is the meaning of sum(0). You see that it must be 0. The base case is not as abstract since it's just one case and is usually the easier to figure out.

It's really the same for the factorial too. You look at a concrete example like f(5):

```
f(5) = 5 * 4 * 3 * 2 * 1
```

and you try to see another factorial on the right. You see that it must be:

$$f(5) = 5 * (4 * 3 * 2 * 1) = 5 * f(4)$$

in other words

$$f(n) = n * f(n - 1) \quad (+)$$

If you don't see what f(0) is, you just try some examples using

$$f(n) = n * f(n - 1) \quad (+)$$

Let's try f(5).

```
f(5) = 5 * f(4)                    using (+) with n = 5
     = 5 * (4 * f(3))              using (+) with n = 4
     = 5 * (4 * (3 * f(2)))        using (+) with n = 3
     = 5 * (4 * (3 * (2 * f(1))))  using (+) with n = 2
     = 5 * (4 * (3 * (2 * (1 * f(0)))))  using (+) with n = 1
```

But you know concretely that

$$f(5) = 1 * 2 * 3 * 4 * 5$$

which means that

$$1 * 2 * 3 * 4 * 5 = 5 * (4 * (3 * (2 * (1 * f(0)))))$$

i.e., (removing the annoying parentheses):

$$1 * 2 * 3 * 4 * 5 = 5 * 4 * 3 * 2 * 1 * f(0)$$

which means, after canceling terms,

$$1 = f(0)$$

Remember this extremely important principle: To design a recursive formula, you must see a **smaller subproblem(s) within a bigger problem**.

For instance you must see sum(n – 1) within sum(n) and you must see f(n – 1) inside f(n).

There are times when instead of “n – 1” in the smaller problem you might see the smaller subproblem of g(n – 2) within g(n). Or you might see h(n – 1) and h(n – 2) with h(n). Etc.

Recursive function of two parameters

The number of ways to choose r objects from n is a very common problem in math and computer science. It occurs so frequently that a special symbol has been invented for it:

$$\binom{n}{r}$$

(We read this as “n choose r”). Let's do a simple example. Suppose we have 5 symbols A, B, C, D, E and we want to choose two symbols from this set of symbols. These are all the possible selections:

AB, AC, AD, AE, BC, BD, BE, CD, CE, DE

In other words “5 choose 2” is 10. Note that AB is the same as BA since BA is “choosing B and A”. We are not interested in the order of choosing the symbols; choosing A then B is the same as choosing B then A.

In fact there's a formula for “n choose r” in terms of factorials. It's given by this:

$$\frac{n!}{r!(n-r)!}$$

So for “5 choose 2” is

$$\frac{5!}{2!(5-2)!}$$

which is

$$\frac{5!}{2!3!}$$

which is

$$\frac{5.4.3.2.1}{(2.1).(3.2.1)}$$

and you can see easily that that is indeed 10. (I'm not going to explain why the above formula works. Take MATH225 if you want to find out more.)

But there's another way to compute “n choose r” that avoids the factorial ...

Let's write C(n,r) for “n choose r”. Here's a relations:

$$C(n, r) = C(n - 1, r) + C(n - 1, r - 1)$$

If you use this repeatedly to compute C(5, 2) (“5 choose 2”) you get this:

$$\begin{aligned} C(5, 2) &= C(4, 2) + C(4, 1) \\ &= (C(3, 2) + C(3, 1)) + (C(3, 1) + C(3, 0)) \end{aligned}$$

This formula is obviously helpful:

C(n, 0) = 1

Continuing the computation:

```
C(5, 2)  = C(4, 2) + C(4, 1)
         = (C(3, 2) + C(3, 1)) + (C(3, 1) + 1)
         = (((C(2, 2) + C(2, 1)) + (C(2,1) + C(2,0))
                  + ((C(2, 1) + C(2, 0)) + 1)
```

At this point it's useful to know this:

C(n, n) = 1

Continuing the computation:

```
C(5, 2)  = (((1 + C(2, 1)) + (C(2,1) + 1)
                  + ((C(2, 1) + 1) + 1)
         = (((1 + (C(1, 1) + C(1, 0)))) + ((C(1, 1) + C(1, 0)) + 1)
                  + (((C(1, 1) + C(1, 0)) + 1) + 1)
         = 10
```

In summary we have these relations:

C(n, r) = C(n – 1, r) + C(n – 1, r – 1) if r > 0 and r < n
C(n, 0) = 1
C(n, n) = 1

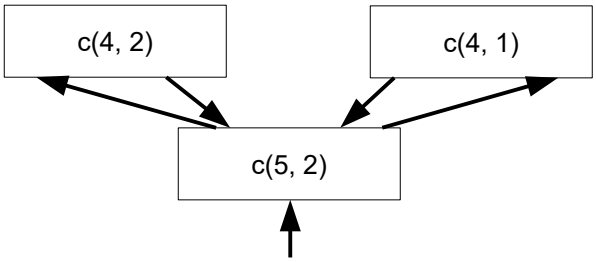
Here's the code implementing the above mathematical facts about C(n,r):

```
int c(int n, int r)
{
    if (n == r || r == 0)
    {
        return 1;
    }
    else
    {
        return c(n - 1, r) + c(n - 1, r - 1);
    }
}
```

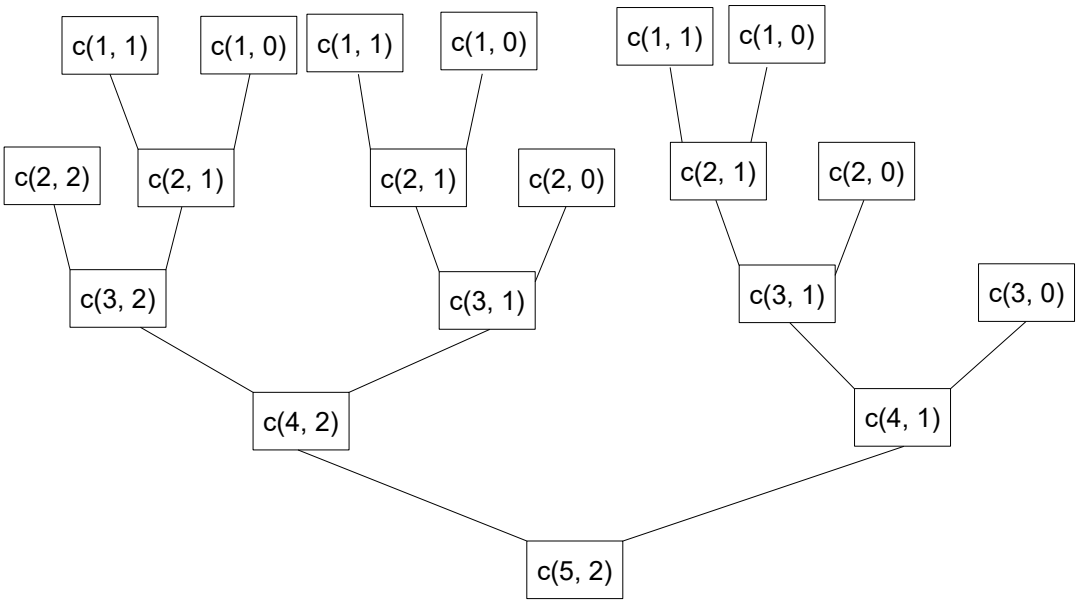
Note that the recursion is different from previous examples because c() calls itself **twice:**

```
return c(n - 1, r) + c(n - 1, r - 1);
```

For instance `c(5,2)` calls `c(4,2)` and `c(4,1)`, adds the return values, and return this sum. The mental picture is therefore slightly more complex. This is part of the diagram:



The whole picture looks like this. I will only draw a line for each function call and omit the line for return:



Isn't this beautiful?

Do you notice that this looks like a tree? This structure is in fact called a **tree**. The box labeled c(5,2) is called the **root** of the tree. The boxes are called **nodes/vertices** of the tree. The boxes that corresponds to base cases are called **leaf nodes/vertices**. You will learn more about this mathematical structure in your other Math/CS classes (for instance MATH325, CISS350, CISS358, etc.)

Exercise. You should insert print statements in the above program to verify that you do get the above tree of function calls. Fill in the return

values of a function call for each return arrow.

Exercise. Given the following mathematically defined function

$$f(i, j) = \begin{cases} 5 & \text{if } i = 0 \text{ or } j = 0 \\ j * f(i - 1, j) + i * f(i, j - 1) & \text{otherwise} \end{cases}$$

Compute f(2, 3) and f(4, 2) by hand. Translate the above recursion into a C++ function and verify your computation with a program.

Exercise. Given the code for function `f()`:

```
int f(int x, int n)
{
    if (n == 0)
    {
        return 1;
    }
    else if (n == 1)
    {
        return x;
    }
    else
    {
        return f(x, n / 2) * f(x, n - n / 2);
    }
}
```

compute `f(2, 4)` by hand. Now run the program and verify your computation.

GCD

GCD stands for **greatest common divisor**. For instance the GCD of 10 and 35 is 5 because 5 is the largest divisor of both 10 and 35. The GCD of 100 and 30 is 10 since 10 is the largest divisor of both 100 and 30.

One way to compute the GCD of two numbers is to test all the numbers between 1 and the smaller of the two:

```
int min(int x, int y)
{
    if (x < y)
        return x;
    else
        return y;
}

int main()
{
    int x, y;
    std::cin >> x >> y;

    int gcd = 1;
    for (int i = 2; i <= min(x, y); i++)
    {
        if (x % i == 0 && y % i == 0)
        {
            gcd = i
        }
    }
    std::cout << "gcd = " << gcd << std::endl;
    return 0;
}
```

Try it out.

But there is a faster algorithm due to Euclid, a Greek mathematician. This is called **Euclid's algorithm**.

```
gcd(m, n):
        if n == 0:
                return m
        else:
                return gcd(n, m % n)
```

This is an extremely important algorithm and is used in CS, math, engineering, etc. in areas such as cryptography, data compression, etc. Here's the code:

```
int gcd(int m, int n)
{
    if (n == 0)
        return m;
    else
```

```
        return gcd(n, m % n);
}
```

Exercise. Compute `gcd(42, 66)` by hand. Draw the function call tree for `gcd(42, 66)`. Label the return arrows with the return values.

Exercise. Now compute `gcd(66, 42)` by hand.

Exercise. A brute force way to compute GCD is this:

> Given positive integer m and n, run d from 1 to the smaller of m and n, check if d is a divisor of both m and n. Compute the largest of such d.

Here's the pseudocode:

```
int max_d;
for d = 1, 2, 3, …, min(m,n):
        if d divides both m and n:
                if d < max_d:
                        max_d = d
```

The max_d is the GCD of m,n. (Why does this running max computation not require an initialization of max_d?) Since d keeps increasing, the above is really the same as

```
int max_d;
for d = 1, 2, 3, …, min(m,n):
        if d divides both m and n:
                max_d = d
```

I'll let you think about this … this is a better pseudocode:

```
int max_d;
for d = min(m,n), min(m,n) – 1, min(m,n) – 2, …, 1:
        if d divides both m and n:
                max_d = d and break the loop
```

Now you have two ways to compute GCD: by the earlier recursion or using this loop. Try to compute the GCD of several pairs of m,n and decide which algorithm is faster. (Hint: Euclid is smart.)

Degree 2 recursion

The **fibonacci numbers** are defined to be

1, 1, 2, 3, 5, 8, 13, 21, ...

i.e., the first two fibonacci numbers are 1 and a subsequent fibonacci number is defined to the the sum of the previous two. For instance 5 is the sum of 2 and 3. Therefore if fib() is the fibonacci function, we have this:

fib(n) = fib(n-1) + fib(n-2) if n > 1
fib(0) = 1
fib(1) = 1

This is called a **degree two** recursion because fib(n) is in terms of fib(n-1) and fib(n-2) and the largest difference between the parameters is 2 (i.e., the largest difference is between n and n − 2 which is 2).

Exercise. What is the degree of this recursive definition:

a(n) = 3a(n-1) + 4a(n-2) + 5a(n-3) if n > 2
a(0) = 0
a(1) = 1
a(2) = 2

What will happen if I only give you two base cases such as

a(n) = 3a(n-1) + 4a(n-2) + 5a(n-3) if n > 1
a(0) = 0
a(1) = 1

Exercise. What is the degree of this recursive definition:

b(n) = 5b(n-2) - 7b(n-4) + 12b(n-7)

How many base cases should there be?

Let's work out fib(2) using the formula.

fib(2) = fib(1) + fib(0) = 1 + 1 = 2

What about f(4)?

```
fib(4)  = fib(3) + fib(2)
        = (fib(2) + fib(1)) + (fib(1) + fib(0))
        = ((fib(1) + fib(0)) + 1) + (1 + 1)
        = ((1 + 1) + 1) + 2
        = (2 + 1) + 2
        = 3 + 2
        = 5
```

Note that previously we worked with degree one recursion. For instance look at this:

f(n) = n x f(n-1), if n > 0
f(0) = 1

The difference between n and n-1 is 1. The factorial is a recursive function of degree 1.

Back to our recursion. Here's the code:

```
int fib(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Exercise. Draw the function call graph for fib(4) – include the return call arrows as well. Remember to label the return arrows with the return value. How many nodes/vertices are there? Can you figure out how roughly many nodes/vertices there are for fib(n) in general?

Exercise. You are given the mathematical description of a degree 2 recursion:

a(n) = a(n-1) + 3 * a(n-2) if n > 1
a(0) = 1
a(1) = 2

Compute a(2), a(3) by hand. Next, write a degree 2 C++ recursive function for a(). Test your code by printing a(0), a(1), a(2), and a(3). Make sure your computation by hand matches the output of your program. Draw the function call diagram for a(4).

Exercise. In some books the fibonacci function is defined like this:

fib(n) = fib(n-1) + fib(n-2) if n > 1
fib(0) = 0
fib(1) = 1

Note that fib(0) is 0 and not 1. Write down the value of fib(n) for n = 0, 1, 2, 3, 4, 5, 6, 7. Implement this fib() function in C++ and print the values of fib(n) for n = 0, 1, 2, 3, 4, 5, 6, 7 and verify that you computations were done correctly. What is the relationship between this fibonacci function and ours earlier.

Exercise. What's wrong with this recursion?

```
int f(int x)
{
    if (x == 0)
        return 3;
    else
        return x * f(x - 2);
}
```

If you don't see it, try to compute f(5) by hand. Next, insert some print statements into the function like this:

```
int f(int x)
{
    std::cout << "f(" << x << ")" << std::endl;
    if (x == 0)
        return 3;
    else
        return x * f(x - 2);
}
```

and try to call f(5).

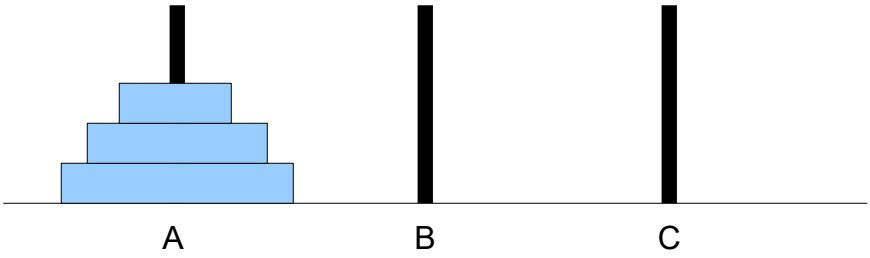
Tower of Hanoi

Most of the previous examples are easily solved without recursion. For instance the computation of the sum from 1 to a number, n say, can be easily achieved using a for-loop.

Now we come to a problem that is ***not*** that easily solved without recursion. The following is a very famous problem called … the

Tower of Hanoi.

Suppose you have a set of disks, say we have 3, with different sizes. They each have a hole in the center. On the player platform there are three stakes or needles, A, B, and C. Initially the three disks are placed on needle A through their holes.



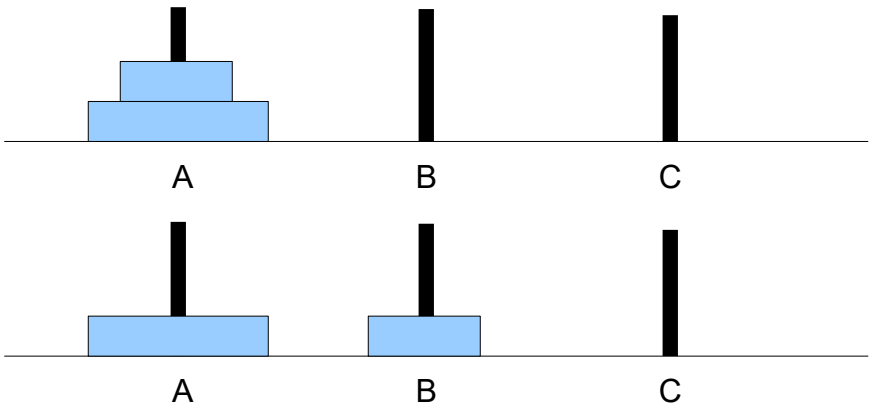
Here's the goal: We want to move all the disks from stake A to stake C. There are some rules:

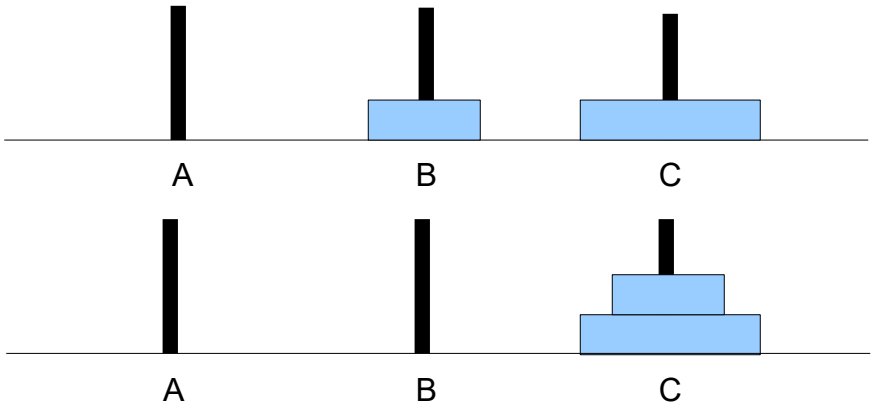
- You can move one disk at a time.
- You can only remove the topmost disk from a needle and place it on top of the disk of another needle.
- At all times, the disks on a needle must be arranged in descending sizes from bottom to top.

You also want to be as efficient as possible, i.e., use the least number of moves.

I'll do the simple **case of 2 disks**.

A → B (this means take the topmost disk of A and put it in B)
A → C
B → C





Do you get the point of the game now?

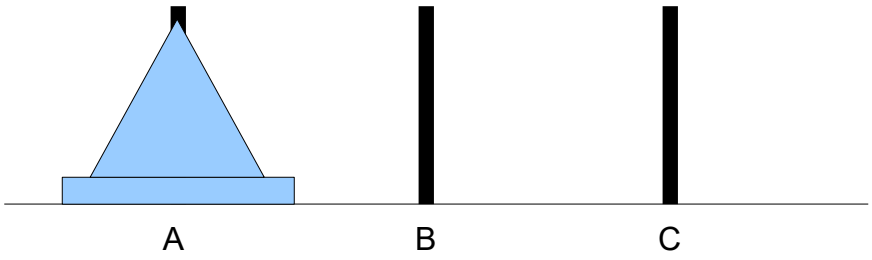
Exercise. Write down a list of steps to move the 3 disks from A to C. I'm giving you the first (Hint: There should be 7 steps).

1. A → C
2.
3.
4.
5.
6.
7.

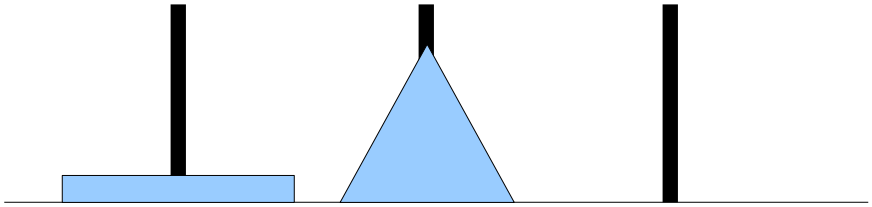
Exercise. Do the same for 4 disks. There should be 15 steps.

Of course the problem becomes increasingly more and more complex (if you don't see a common idea/pattern of the above solutions) when the number of disks n grows. For instance what is the solution when n = 64?

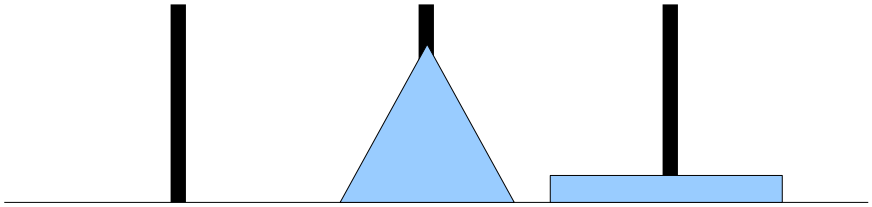
Let's solve the general problem. Suppose we have n disks. There are three needles. We can think of them as the "from" needle, the "helper" needle, and the "to" needle. We move all the disks from the "from" to to "to" using the "helper" as a helper. Think of the problem as one involving n-1 disks, and one disks:



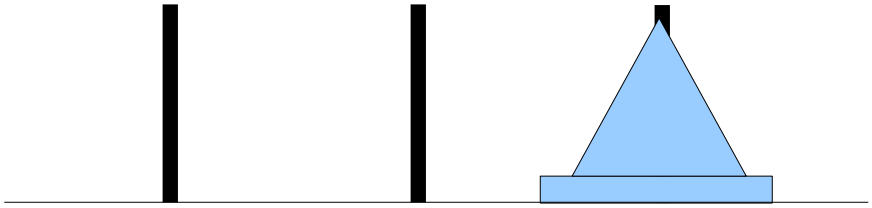
The triangle denotes the n − 1 disks. You want to solve the Tower of Hanoi problem for the n – 1 disks, but moving the n – 1 disks to the middle needle, using the C as a helper.



Next I move the last disks, which is a Tower of Hanoi problem of size 1, from A to C.



Now we solve the Tower of Hanoi problem of size n-1. Note that in this case the “from” needle is B, the “to” needle is C, and the “helper” needle is A.



Let's write down the pseudocode. Suppose hanoi(n, from, helper, to) prints the moves. The case of n = 1 is easy:

```
hanoi(1, from, helper, to):
        print from, “->”, to
```

and for n > 1:

```
hanoi(n, from, helper, to):
        hanoi(n – 1, from, to, helper)
        hanoi(1, from , helper, to)
        hanoi(n – 1, helper, from, to)
```

So the pseudocode is

```
hanoi(n, from, helper, to):
        if n == 1:
```

```
        print from, “--->”, to
    else:
        hanoi(n – 1, from, to, helper)
        hanoi(1, from , helper, to)
        hanoi(n – 1, helper, from, to)
```

Think about the pseudocode very carefully. Don't look at the code yet.

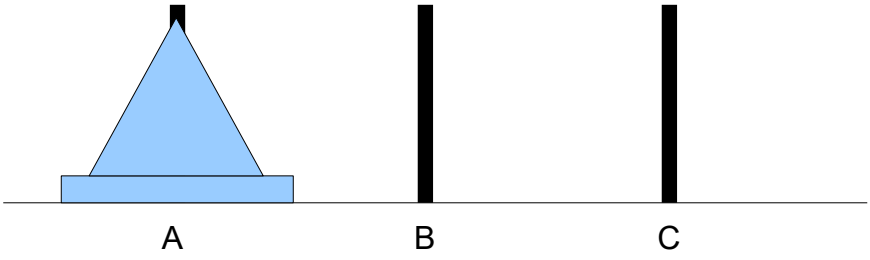
Exercise. Execute the pseudocode for the function call hanoi(3, 'A', 'B', 'C')

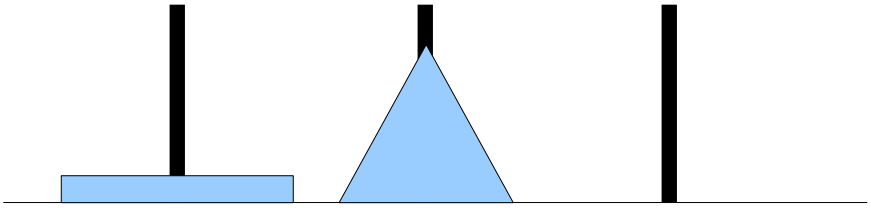
Here's the code (finally!):

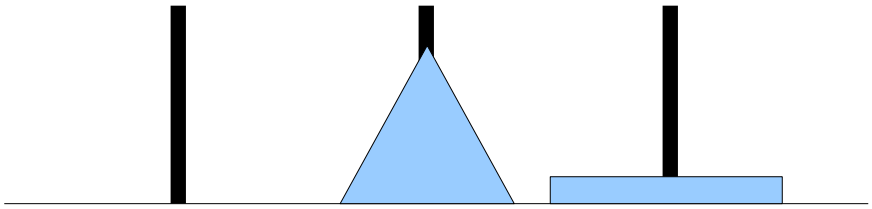
```
#include <iostream>

void hanoi(int n, char from, char helper, char to)
{
    if (n == 1)
    {
        std::cout << from << "--->" << to
                  << std::endl;
    }
    else
    {
        hanoi(n - 1, from, to, helper);
        hanoi(1, from, helper, to);
        hanoi(n - 1, helper, from, to);
    }
}

int main()
{
    int n = 0;
    std::cout << "how many disks? ";
    std::cin >> n;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

Exercise. Challenge: Solve this problem without recursion. (Yes it can be done.)

Exercise. The recursive idea behind the solution of the tower of Hanoi is made up of the following sequence of four pictures:







Note that the largest disk goes from the starting stake to the target stake. Suppose now I change the tower of Hanoi problem a little bit as follows:

- There are four stakes. The four stakes are called, A, B, C, D where A is the starting stake and D is the target stake.
- No disk can go from the starting stake to the target stake. For instance if there's only one disk (i.e., at A), then you cannot

move the disk from A to D – you have to move the disk from A to B (or to C) and then from B to D.

Write a program to solve this modified tower of Hanoi problem. (There are many possible solution.)

Exercise. Count the number of moves made for n = 1 disk, n = 2 disks, n = 3 disks, n = 4 disks, etc. Is there a formula for the **number of moves** for n disks? This can then be used as a measure of efficiency of the algorithm.

Exercise. Now let me modify the tower of Hanoi problem: What if you cannot move a disk from A to C? In other words, you can move from A to B, B to A, B to C, C to B, C to A. But you cannot move a disk from A to C. Is it possible to solve this version of tower of Hanoi? How many moves does your algorithm make for n disks?

Exercise. What about this version – what if you have 4 needles? You should be able to move a stack of n disks from A to C with fewer moves right?

Exercise. How suppose there are 5 stakes A, B, C, D, E. There two stacks of disks, one stack is at A and one stack is at B. Each stack is made up of n disks of radius 1, 2, 3, …, n. The goal is to move all the disks to E. (This means that when the procedure stops, all the disks are at E and the radii of the disks, from top to bottom, are 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, , …, n, n. Can you do this without stake D?

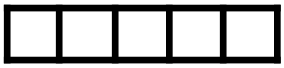
Many computational problems can be solved a lot easier with recursion than without. Another reason for knowing recursion is that it's frequently the case that ***correctness*** of a program can be ***mathematically proven*** much easier if it's recursive. (For many critical software – example: scientific software – correctness is crucial.) Note that from the above examples, it should be clear that recursion can do the work of loops. In fact there are programming languages that do not have loops!

Recursion and a tiling problem

Here's one such problem: How many ways are there to tile a 1-by-5 board

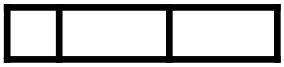
with 1-by-1 squares and 1-by-2 rectangles:

Here's one way:



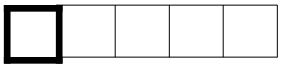
and here's another:

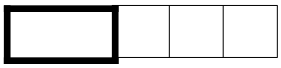
and another:



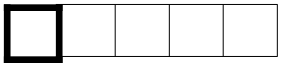
Of course this is not difficult: Just do it yourself. How many can you find? However if I say since the number of ways to tile not a 1-by-5 board but a 1-by-100 … then you'd find that a naïve and brute-force way of listing all the tilings is not the best way to solve this problem. Believe it or not … this problem is recursive. Why? Look at the 1-by-5 board.

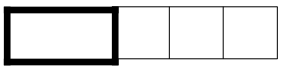
You have to start either with a 1-by-1 or a 1-by-2:



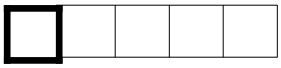


Suppose the number of ways to tile a 1-by-n board is written T(n). In other words I'm interested in T(5). Where's the recursion? Look at the above:



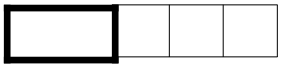


For the first case, after putting a 1-by-1 tile down,



I'm left with a 1-by-4. This means that there are T(4) ways to complete the first case!!!

Furthermore, for the second case, after putting down a 1-by-2



I'm left with a 1-by-3 space to tile. And there are T(3) ways to complete the tiling.

Hence altogether the number of ways to tile a 1-by-5 board, i.e. T(5), must be T(4) + T(3).

Get it? Here's the recursion:

$$T(5) = T(4) + T(3)$$

Convince yourself that

$$T(n) = T(n - 1) + T(n - 2)$$

if $n > 2$. It's easy to see that

$$T(1) = 1$$
$$T(2) = 2$$

(Make sure you see why!!!) Altogether we have

$$T(1) = 1$$
$$T(2) = 2$$
$$T(n) = T(n - 1) + T(n - 2), \text{ if } n > 2$$

Therefore to compute T(5) we have

$$T(5) = T(4) + T(3)$$
$$T(4) = T(3) + T(2) = T(3) + 2$$
$$T(3) = T(2) + T(1) = 2 + 1 = 3$$

Putting T(3) into the second equation we get

$$T(4) = T(3) + 2 = 3 + 2 = 5$$

and putting T(3) = 3 and T(4) = 5 into the first equation we get

$$T(5) = 5 + 3 = 8$$

Exercise. Draw all the tilings of a 1-by-5 board. Do you have 8?

Exercise. Draw all the tilings of a 1-by-6 board. How many are there? Now use

$$T(1) = 1$$
$$T(2) = 2$$
$$T(n) = T(n - 1) + T(n - 2), \text{ if } n > 2$$

to compute the number of tilings of the 1-by-6 board.

If you write down T(1), T(2), T(3), T(4), … you would see this sequence:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Of course note that after the first two terms, every term is the sum of the previous two. That's because of

$$T(n) = T(n - 1) + T(n - 2), \text{ if } n > 2$$

We can set T(0) = 1 (there is only one way to tile the 1-by-0 board: don't tile!) In that case the sequence of T(0), T(1), T(2), T(3), T(4), … is

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

This is the famous **Fibonacci sequence** that you have seen earlier!!!

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n - 1) + fib(n - 2), \text{ if } n > 1$$

This sequence of numbers appears in many areas of math, physics, chemistry, finance, … you name it.

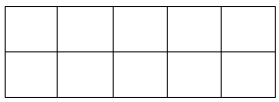
You see from this example that we try to solve a problem by restating it in terms of small subproblems:

Problem	Subproblems
tiling a 1-by-5 board	tiling a 1-by-4 board
	tiling a 1-by-3 board

Specifically:

a 1-by-5 tiling is	either	a 1-by-1 tile followed by a 1-by-4 tiling
	or	a 1-by-2 tile followed by a 1-by-3 tiling

Exercise. Let T(n) be the number of tilings of a 2-by-n board using 1-by-2 or 2-by-1 tiles. As an example, for the case of n = 5, how many ways are there to tile the following 2-by-5 board:

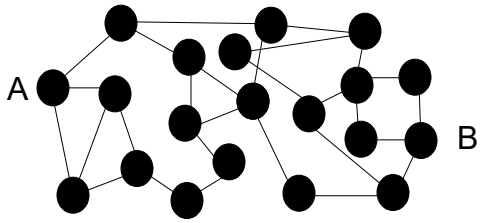


with the following types of tiles:

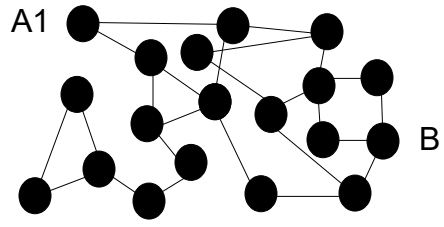
Let T(0) = 1. Complete this:

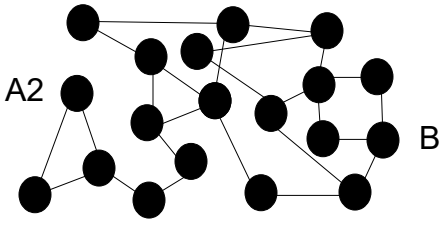
T(0) = 1
T(1) =
T(n) =

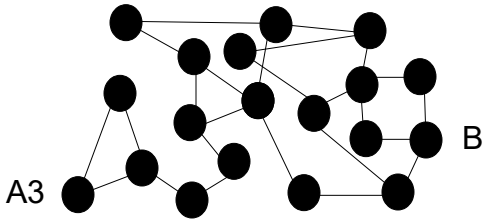
You might think “Aw.... this is just one of those academic math problems. I don't see how a puzzle like this can be used in the real world.” The essence of the technique, i.e. recursion, appears all the time in the real world. Here's one: Find the shortest path on this “simplified internet” from machine A to B in this configuration



where each dot is a machine and each line is a communication line between the two machines. This can be thought as a made up of finding shortest path from A1,A2,A3 to B in the following configurations or subproblems:







Not only that, this is more or less also similar to finding a winning chess strategy where each “machine” (each black dot) is replaced by a particular “chess board setup” and the lines are replaced by an arrow that depicts a chess move that changes one chess board state to another.

Also, tiling problems do appear in the real world. Tiling problems are related to “space packing” problems. For instance shipping companies are interested in optimizing the space in a cargo container. They are therefore interested in algorithms to solve problems like this: Given the (3-dimensional) measurement of cargo containers and a list of (3-dimensional) measurements of items to be placed in the container, find the best possible way to pack the items in the container. It also appears in diverse areas such as packing gates in a chips, protein folding, liquid crystal packing in an LCD screen, etc.

I will stop here for our first introduction to recursion. There is not a complete set of notes on recursion. So this is not the last time you will be thinking recursively. But … why did I bring up recursion at this point?

The reason is because you will see later (when we do recursion again), that loops can be achieved through recursive programming, i.e. using recursive functions.

Exercise. Suppose S(n) satisfies the following:

$$S(0) = 1$$
$$S(n) = 2\,S(n-1) + 1 \text{ if } n > 0$$

What is S(5)? Can you find a formula for S(n)?

Recursive computation on arrays: Part 1

Here's another example. You already know that you can write a loop to compute the max value in an array. Let's look at this problem recursively. Suppose I write x is an array of of value 2,3,5,6,4,1 and we want a function that looks like this:

max(x)

If you split the array x into the first value of the array

2

and the rest say we call it y of value 3,5,6,4,1, then we have this relation:

max(x) = maximum of 2 and max(y)

You see that the max function occurs on both sides. This is a recursive relation. Furthermore, we need the function max to accept index values. Then

```
// Returns max of x[start], x[1], x[2], ...,
// x[end - 1]
int max(int x[], int start, int end)
{
    if (start == end - 1)
    {
        return x[start];
    }
    else
    {
        if (x[start] < max(x, start + 1, end))
        {
            return max(x, start + 1, end);
        }
        else
        {
            return x[start];
        }
    }
}
```

Note that `max(x, start + 1, end)` is computed twice. To save computations, we can do this:

```
int max(int x[], int start, int end)
{
    if (start == end - 1)
    {
        return x[start];
    }
    else
    {
        int y = max(x, start + 1, end);
```

```
        if (x[start] < y)
        {
            return y;
        }
        else
        {
            return x[start];
        }
    }
}
```

Essentially you break the problem of finding the max of x[0],x[1],...,x[n-1] to

maximum between x[0] and max of x[1],x[2],...,x[n-1]

Exercise. Write a recursive version of a minimum function
`int min(int x[], int start, int end);`
that returns the minimum of `x[start], ..., x[end - 1]`.

Exercise. Write a recursive version of a linear search function
`int linearsearch(int x[], int start, int end, int target);`
that finds the first index where `target` is found in the array `x[start], x[start + 1], ..., x[end - 1]`.

Exercise. Write a recursive version of a count function
`int count(int x[], int start, int end, int target);`
that returns the number of times `target` occurs in `x[start], x[start + 1], x[start + 2], …, x[end - 1]`.

Exercise. Write a recursive version of a sum function
`int sum(int x[], int start, int end);`
that returns `x[start] + x[start + 1] + x[start + 2] + … + x[end - 1]`.

Exercise. A string is a palindrome if it's the same when you read it left-to-right and right-to-left. For instance here's one:

`madam`

Here's another one: What does Adam say to Eve?

`Madam, I'm Adam.`

If you remove spaces and punctuations and change everything to lowercase, you get

```
madamimadam
```

Write a recursive function with the prototypes

```
bool is_palindrome(char s[], start, end);
```

so that `is_palindrome("madamimadam", 0, 11)` returns `true`.

Recursive computation on arrays: Part 2

In the previous section, we break the problem of finding the max of x[0],x[1],...,x[n-1] to

maximum between x[0] and max of x[1],x[2],...,x[n-1]

But there's another way to do this … Suppose I have an array x with the following values:

{2,3,5,6,4,1}

I call max({2,3,5,6,4,1}). Now, max({2,3,5,6,4,1}) computes in the following way. It breaks up {2,3,5,6,4,1} into **two arrays of roughly the same length**:

{2,3,5} and {6,4,1}

and call max() on each of these pieces and wait for the result. When it receives the results, it will compare the two values and return the largest.

For instance max({2,3,5,6,4,1}) will wait for max({2,3,5}). What will max({2,3,5}) do? It will break up 2, 3, 5 into two pieces roughly of the same lengths to get

{2} and {3,5}

and then call max({2}) and max({3,5}) and wait for their results, compare them, and return the larger.

Clearly the maximum of an array with one value (such as 2) must be the value itself (i.e., 2). What about max({3,5})? max({3,5}) breaks up {3,5} to get:

{3} and {5}

and calls max({3}) and max({5}). The max({3}) will return 3 and max({5}) will return 5. So 3 and 5 is returned back to max({3,5}). max({3,5}) will return the larger of 3 and 5, i.e., it will return 5.

Now who will receive the return value of max({3,5})? You see that it is max({2,3,5}). Altogether, max({2,3,5}) will return 2 from max({2}) and 5 from max({3,5}). Therefore max({2,3,5}) will return the larger of 2 and 5, i.e., 5. Who does max({2,3,5}) return to? The 5 is returned back to max({2,3,5,6,4,1}). Don't forget that max({2,3,5,6,4,1}) calls max({2,3,5}) and max({6,4,1}). Only one value, i.e. 5, is returned.

You can check for yourself, using the above scheme, that max({6,4,1}) will be returning 6.

Therefore max({2,3,5,6,4,1}) will return the large of 5 and 6, i.e., it will return 6.

So the pseudocode looks like this:

```
int max(int x[], int start_index, int end_index)
{
        if start_index == end_index
        {
                // BASE CASE
                return x[start_index]
        }
        else
        {
                // RECURSIVE CASE
                int mid = (start_index + end_index) / 2
                int maxleft = max(x, start_index, mid)
                int maxright = max(x, mid+1, end_index)
                return the large of maxleft, maxright
        }
}
```

Make sure you study this VERY VERY CAREFULLY!!! It's EXTREMELY IMPORTANT!!!

As you know (and I've said this a gazillion times), tracing a progam/pseudocode helps in understanding the underlying logic behind the program/pseudocode. This is particularly important for recursive functions because recursive functions will be in some of your later classes. It's not just an interesting trick. Recursive functions are fundamental objects of study in both mathematical logic and CS. I expect you to study the following two traces VERY CAREFULLY.

To make sure you REALLY understand the above pseudocode, here is a trace of the computation of max({4}, 0, 0):

```
max({4}, 0, 0):
        x = {4}
        start_index = 0
        end_index = 0
        start_index == end_index is true:
                // BASE CASE
                return x[start_index], i.e., return x[0], i.e., 4
```

We're done.

Here's the trace of max({2,4}, 0, 1):

```
max({2,4}, 0, 1):
   x = {2,4}
   start_index = 0
   end_index = 1
   start_index == end_index is false:
      // RECURSIVE CASE:
      mid = (start_index + end_index) = (1 + 0)/2 = 0
      maxleft = max(x, 0, 0) … WAITING
```

```
max({2,4}, 0, 0):
   x = {2,4}
   start_index = 0
   end_index = 0
   start_index == end_index is true:
      // BASE CASE:
      return x[start_index], i.e., return x[0], i.e., return 2
```

```
(CONTINUING max({2,4},0,1) ...
max({2,4}, 0, 1):
   x = {2,4}
   start_index = 0
   end_index = 1
   start_index == end_index is false:
   // RECURSIVE CASE:
      mid = (start_index + end_index) = (1 + 0)/2 = 0
      maxleft = max(x, 0, 0) = 2
      maxright = max(x, mid+1, end_index) = max(x, 1, 1) … WAITING
```

```
max(x, 1, 1):
   x = {2,4}
   start_index = 1
   end_index = 1
   start_index == end_index is true:
      // BASE CASE:
      return x[start_index], i.e., return x[1], i.e., return 4
```

```
(CONTINUING max({2,4},0,1) ...
max({2,4}, 0, 1):
   x = {2,4}
   start_index = 0
   end_index = 1
   start_index == end_index is false:
      // RECURSIVE CASE:
      mid = (start_index + end_index) = (1 + 0)/2 = 0
      maxleft = max(x, 0, 0) = 2
      maxright = max(x, mid+1, end_index) = max(x, 1, 1) = 4
      return larger of 2 and 4, i.e., return 4
```

max({2,4}, 0, 1) was the first function call. So the final end result is 4.

Exercise. Perform a trace similar to the above for the computation of max({5,1}, 0, 1).

Exercise. Perform a trace similar to the above for the computation of max({2,5,3}, 0, 2).

Exercise. Perform a trace similar to the above for the computation of max({2,5,6,1}, 0, 3).

Exercise. Perform a trace similar to the above for the computation of max({2,5,3,6,1}, 0, 4).

Exercise. Implement the pseudocode in C++. Test the code. Insert print statements if necessary to understand the execution of several examples.

In the previous section, we basically compute by breaking up an array into one value and the rest, something like this:

max of {5,3,1,2,4,6} = larger of 5 and max of {3,1,2,4,6}

Yes, it is recursion since max is calling max. In ***this*** section we break up the array this way:

max of {5,3,1,2,4,6} = larger of max of {5,3,1} and max of {2,4,6}

See the difference?

You can think the length of the array of a measure of the size of the problem to be solved. Obviously a large array will require more time. In the previous section a problem (array) of size 6 will give rise to a problem of size 1 and another size 5.

max of {5,3,1,2,4,6}) = larger of 5 and max of {3,1,2,4,6}
size 6 size 1 size 5

In ***this*** section a problem of size 6 will give rise to a problem of size 3 and another of size 3.

max of {5,3,1,2,4,6} = larger of max of {5,3,1} and max of {2,4,6}
size 6 size 3 size 3

In general, an algorithm that solves a problem using recursion so that a problem of size n is solved by recursing on two smaller problems, each of about half the size of the original is said to be a **divide-and-conquer** algorithm.

If your algorithm divides the problem of size n into three problems where each has size roughly n/3, then it is also called a divide-and-conquer algorithm (except that it divides into 3 equal parts instead of 2.)

Exercise. Develop a divide-and-conquer recursive algorithm to compute the minimum of an array.

```
int min(int x[], int start, int end);
```

Test it by tracing it with several cases. Once you're confident that it works, implement it in C++.

Exercise. Write a recursive function `print(int, int)` such that `println(1, 6)` prints 1 2 3 4 5 and a newline. `println(3, 8)` prints 3 4 5 6 7 and a newline.

Exercise. Develop a divide-and-conquer recursive algorithm to compute the smallest index value where a target occurs in an array:

```
int linearsearch(int x[], int start, int end, int
target);
```

If the target is not found, -1 is returned. Test it by tracing it with several cases. Once you're confident that it works, implement it in C++.

Exercise. Develop a divide-and-conquer recursive algorithm to compute the number of times a target value occurs an array.

```
int count(int x[], int start, int end);
```

Test it by tracing it with several cases. Once you're confident that it works, implement it in C++.

Exercise. Develop a divide-and-conquer recursive algorithm to compute the sum of the values an array:

```
int sum(int x[], int start, int end);
```

Test it by tracing it with several cases. Once you're confident that it works, implement it in C++.

Refer to the standard binary search in your previous set of notes. We will now write the recursive version of the program (in pseudocode):

```
// searches for target in a[left], …, a[right]
binarysearch(a, left, right, target):

        if left > right:
                return -1
        else:
                mid = left + (right - left) / 2
                if a[mid] == target:
                        return mid
                else if a[mid] < target:
```

```
                return binarysearch(a, mid + 1, right, target)
        else
                return binarysearch(a, left, mid – 1, target)
```

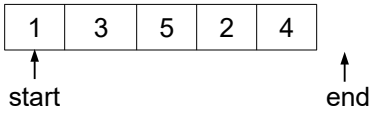
Note that the **binary search is also a divide-and-conquer algorithm**.

Exercise. Implement the binarysearch algorithm in C++. Test your function.

For divide-and-conquer, when you apply that to an integer array where you have two index values **`left`** and **`right`** and you want to compute the index in the middle of left and right, it's easy. It's just

```
int mid = left + (right - left) / 2;
```

Sometimes the subarray you are computing with is specified in terms of pointers, say **`start`** and **`end`** where **`start`** is the address of the first value and **`end`** is the address of the value *just outside* the subarray.



In this case you can compute the address of the value in the middle using

```
int * mid = start + (end – start) / 2;
```

Here's the max of array using divide-and-conquer where you specify the array using two pointers:

```
#include <iostream>

int max(int * start, int * end)
{
    if (start == end - 1)
    {
        return *start;
    }
    else
    {
        int * mid = start + (end - start) / 2;
        int leftmax = max(start, mid);
        int rightmax = max(mid, end);
        return (leftmax >= rightmax ? leftmax : rightmax);
    }
}

int main()
{
    int x[] = {1, 3, 5, 6, 4, 2};
    std::cout << max(&x[0], &x[6]) << std::endl;
```

```
    return 0;
}
```

Exercise. Write (and test) a `min` function using divide-and-conquer:

```
int min(int * start, int * end);
```

Exercise. Write (and test) a binary search function with the following prototype:

```
int * binarysearch(int * start, int * end, int target);
```

Exercise. Write an exponentiation function `power(double, int)` such that `power(2.0, 6)` return 2.0 to the power of 6. You must use the divide-and-conquer strategy.

Mutual Recursion

Sometimes recursion might involve more than one function.

Let f(n), g(n) be functions satisfying

f(0) = 1
f(n) = 2*g(n/2) if n > 0 and n is even
f(n) = f(n/2) if n > 0 and n is odd

g(0) = 2
g(n) = f(n/2) if n > 0 and n is even
g(n) = 3*g(n/2) if n > 0 and n is odd

(/ refers to integer division.) Do you see f(n) using g(n) and g(n) using f(n)?

Exercise. Compute f(10), g(10) by hand.

Here's the code. Study it well. Note that the functions are mutually recursive so we must use function prototypes. Verify your computation with the program. Insert print statements to see the function being called.

```
#include <iostream>

int f(int);
int g(int);

int f(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        if (n % 2 == 0) return 2 * g(n / 2);
        else return f(n / 2);
    }
}


int g(int n)
{
    if (n == 0)
    {
        return 2;
    }
    else
    {
        if (n % 2 == 0) return f(n / 2);
        else return 3 * g(n / 2);
```

```
    }
}

int main()
{
    std::cout << f(5) << std::endl;
    return 0;
}
```

Exercise. Write a pair of mutually recursive functions that prints the values of an array at the even index positions:

```
void print(int x[], int & index, int size);
void skip(int x[], int & index, int size);
```

Here's how they work together:

- `print(x, i, n)` prints `x[i]` and calls `skip(x, i + 1, n)`
- `skip(x, i, n)` calls `print(x, i + 1, n)`

Of course `n` is the size of the array `x`.

Exercise. Write a pair of function that checks if a string is an arithmetic expression of digits. An example would be the following string:

"5+2-3*2+7/8"

You should have two functions

```
bool consume_digit_or_end(char x[], int & index);
bool consume_operator(char x[], int & index);
```

(The name of the functions says it all.) Finally create a function to wrap everything up:

```
bool is_digit_expr(char x[])
{
    return consume_digit_or_end(x, 0);
}
```

(And thus we begin automata theory, language processing, and compilers …)