The next topic in the course is *recursion*. Since recursion is closely related to a proof technique in mathematics called *mathematical induction*, which many of you are familiar with, this is a good place to start.

Mathematical induction

Suppose we would like to prove some statement about the natural numbers (non-negative integers). For example we would like to prove: for any $n \geq 1$,

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}.$$

Here we are stating some *proposition* about the number n — call it $P(n)$ – and we want to prove the statement that $P(n)$ is true for all $n \geq 1, P(n)$".

The proof of the statement can be done using the technique of *mathematical induction*, which requires us to prove two things:

1. *base case*: the statement $P(n)$ is true for some $n = n_0$. In our example here, $n_0 = 1$.
2. *induction step*: for any $k \geq n_0$, *if* $P(k)$ is true, *then* $P(k+1)$ must also be true.

   The statement $P(k)$ is called the "induction hypothesis".

The logic is that if you can show $P(n)$ is true for $n = n_0$ (base case), then the induction step would imply that the statement is true for $n = n_0 + 1$ (since the induction hypothesis holds for $n = n_0$ which is the base case), which in turn implies that it must be true for $n = n_0 + 2$, and so on.

Example 1

Prove the formula for the well-known *arithmetic series*:

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

The base case is $n_0 = 1$. It is easy to prove:

$$\sum_{i=1}^{1} = \frac{1 \cdot (1+1)}{2} = 1.$$

We next prove the induction step. For any $k \geq 1$, we assume $P(k)$ is true. We want to show that $P(k+1)$ must also be true.

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= \left(\sum_{i=1}^{k} i\right) + (k+1) \\
&= \frac{k(k+1)}{2} + (k+1), \text{ by induction hypothesis} \\
&= (k+1)(\frac{k}{2} + 1) \\
&= \frac{(k+1)(k+2)}{2} \quad \text{whis proves the induction step!}
\end{aligned}$$

Example 2

Claim: for all $n \geq 1$,

$$1 + 3 + 5 + \cdots + (2n-1) = n^2$$

Proof: The base case of $n_0 = 1$ is obvious, since there is only a single term on the left hand side, i.e. $1 = 1^2$. The induction hypothesis is the statement $P(k)$:

$$P(k) : 1 + 3 + 5 + \cdots + (2k-1) = k^2$$

To prove the induction step, we hypothesize that $P(k)$ is true, and we show that it follows that $P(k+1)$ must also be true.

$$\begin{aligned} \sum_{i=1}^{k+1} 2i - 1 &= \left(\sum_{i=1}^{k} 2i - 1 \right) + 2(k+1) - 1 \\ &= k^2 + 2(k+1) - 1, \quad \text{by the induction hypothesis} \\ &= k^2 + 2k + 1 \\ &= (k+1)^2. \end{aligned}$$

Thus, the induction step is proved.

Example 3

Claim: for all $n \geq 3$, $2n + 1 < 2^n$.

The induction hypothesis $P(k)$ is the statement "$2k + 1 < 2^k$". Note that this statement is not true for $k = 1, 2$.

Proof:
The base case $n_0 = 3$ is easy to prove, i.e. $7 < 8$.
Next we prove the induction step. Let k be some integer such that $k \geq n_0 = 3$. Suppose $P(k)$ is true. We want to show that it follows that $P(k+1)$ is true. We write $P(k+1)$ on the left side:

$$\begin{aligned} 2(k+1) + 1 &= 2k + 3 \\ &= (2k+1) + 2 \\ &< 2^k + 2, \quad \text{by induction hypothesis} \\ &< 2^k + 2^k \\ &< 2^{k+1} \ . \end{aligned}$$

The induction step is proved.

Example 4

Claim: For all $n \geq 5, \quad n^2 < 2^n$.

Proof:
The base case $n_0 = 5$ is easy to prove, i.e. $25 < 32$.
Next we prove the induction step. The induction hypothesis $P(k)$ is the statement "$k^2 < 2^k$". We show that if $P(k)$ is true, then $P(k+1)$ must also be true.

$$\begin{aligned} (k+1)^2 &= k^2 + 2k + 1 \\ &< 2^k + 2k + 1, \quad \text{by induction hypothesis} \\ &< 2^k + 2^k, \quad \text{from Example 3} \\ &= 2^{k+1} \end{aligned}$$

which proves the induction step.

Example 5: upper bound on Fibonacci numbers

Consider the Fibonnacci[13] sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where

$$F(0) = 0, \qquad F(1) = 1,$$

and, for all $n \geq 2$, we define $F(n)$ by

$$F(n) = F(n-1) + F(n-2).$$

Claim:

$$\text{for all } n \geq 0, \quad F(n) < 2^n. \qquad (*)$$

Proof:
We take the base cases to be $n_0 \in \{0, 1\}$. Note $F(0) = 0, F(1) = 1$ and so (*) holds for the base cases since $F(0) = 0 < 2^0$ and $F(1) = 1 < 2^1$.

The induction hypothesis $P(k)$ for $k \geq 1$ is that $F(0) < 2^0, F(1) < 2^1, ..., F(k) < 2^k$. For the induction step, we assume that $P(k)$ is true and we show $P(k+1)$ must also be true.

$$\begin{aligned} F(k+1) &= F(k) + F(k-1) \\ &< 2^k + 2^{k-1} \quad \text{by induction hypothesis} \\ &< 2^k + 2^k \\ &= 2^{k+1} \end{aligned}$$

and so the induction step is proven.

[13] `http://en.wikipedia.org/wiki/Fibonacci_number`

Induction and Recursion

Induction is used to prove many such statements in mathematics. It is also used to prove the correctness of certain computer programs. For example, "n factorial" is defined

$$n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n$$

Here is an algorithm (written in Java) for computing it. The algorithm just applies the definition so there is nothing to prove about correctness.

```
int factorial(int n){  //  assume  n >= 1
   int result = 1;
   for (int i = 1;  i <= n;  i++)
       result *= i;
   return result;
}
```

But here is another way to define $n!$ which is more subtle, namely if $n > 1$, then

$$n! = n \cdot (n-1)!$$

The corresponding algorithm (written in Java) is:

```
int factorial(int n){   //   algorithm assumes argument:  n >= 1
   if  (n == 1)
     return 1;
   else
      return  n * factorial(n - 1);
}
```

Notice that the definition of the algorithm `factorial` involves a call to itself. Such an algorithm is said to be *recursive*.

Claim: The recursive algorithm `factorial(n)` computes $n!$ (as defined at the top of this page) for any input value $n \geq 1$.

Proof:

First, the base case: If the parameter $\texttt{n} = 1$, then the algorithm returns 1, so the base case is true.

Second, the induction step: Suppose that the algorithm returns $k!$ for $k \geq 1$. (the induction hypothesis). We show that it follows that the algorithm returns $(k+1)$! when input argument is $n = k+1$. But this is easy, since when the argument is $k+1$, the algorithm returns $\texttt{n} * \texttt{factorial}(\texttt{n} - 1) = (k+1) \cdot k!$ by induction hypothesis, which is just $(k+1) \cdot k \cdot ... \cdot 1 = (k+1)!$ by definition.

Recursion

In this course the concept of recursion will be used in several contexts: in definitions, in algorithms and in running time analyses. However, all three steps do not to be recursive at the same time. A function may be defined recursively, but its best implementation may be iterative even if it is very tempting to implement it recursively. We are about to see such an example...

factorial

Let's write the recursive version a bit differently, by inserting a tmp variable named `factn` to store the returned result of the recursive call. Its not necessary to do this, but it makes it a bit easier to see what's going on.

```
int factorial(int n){   //   algorithm assumes argument:  n >= 1
   int factn = 0
   if  (n == 1)
     return 1;
   else
      factn = n * factorial(n - 1);
      return  factn
}
```

Each time a method calls another method (or a method calls itself, in the case of recursion), the computer needs to do some administration to keep track of the "state" of method at the time of the call. This information is called a "stack frame". Suppose we call `factorial(6)`, that is, we want to compute "6!". Then this leads to a sequence of calls and subsequent returns from these calls. For example, right before returning from the `factorial(3)` call, we have made the following sequence of calls and returns:

```
  factorial(6)
    factorial(5)
      factorial(4)
        factorial(3)
          factorial(2)
            factorial(1)
            return from factorial(1)
          return from factorial(2)
```

the stack looks like this,

```
    frame for factorial(3):  [factn = 6, n=3] <---- top of stack
    frame for factorial(4):  [factn = 0, n=4]
    frame for factorial(5):  [factn = 0, n=5]
    frame for factorial(6):  [factn = 0, n=6] <---- bottom of stack
```

You can inspect the stack frame with the Eclipse debugger. I strongly recommend that you do this for a simple example like factorial so that you can see for yourself how this works.

There are many other problems for which there is a non-recursive and a recursive solution. Sometimes the recursive way is more natural for expressing what you want to do, and other times the non-recursive way is more natural. Another issue is the computational cost. For factorial, the costs of the two algorithms are similar in that both run in time proportional to n. Let's now look at another problem in which the non-recursive vs. recursive solutions have quite different time costs.

Fibonacci numbers

Consider the Fibonnacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = F(n-1) + F(n-2),$$

where $F(0) = 0, F(1) = 1$. The standard way to calculate the Fibonacci numbers is just to iterate, starting at $n = 0$. Suppose you wanted the n^{th} Fibonacci number, where $n > 0$.

```
ALGORITHM:  fib(n)
      f0 = 0
      f1 = 1
      for i = 1 to n{
         f1 = f1 + f0      //  set  F(n+1) for next round
         f0 = f1 - f0      //  set  F(n) for next round
      }
      return f0
```

The method requires n passes through the "for loop". Each pass takes a small (fixed) number of operations. So we would expect the number of steps to be about cn for some constant[14] c.

A *recursive algorithm* for computing the nth Fibonacci number goes like this:

```
Algorithm:  fib(n)    //   assume n >= 0
//  Input:  the index of the Fibonacci number to be computed
//  Output:  the n-th fibonacci number
//
   if (n <= 1)
      return n
   else
      return fib(n-1) + fib(n-2)
```

The trouble with this recursive algorithm is that you end up calling `fib` on the same parameters *several* times. For example, suppose you are asked to compute the 247-th Fibonacci number. `fib(247)` calls `fib(246)` and `fib(245)`, and `fib(246)` calls `fib(245)` and `fib(244)`. But now notice that `fib(245)` is called twice. Similar redundancies occur each step of the way until you reach `fib(1)` and `fib(2)`. (Indeed the value of `fib(k)` will be computed $fib(n-k)$ times.) As we will see a few lectures from now, the number of steps required in the computation grows like ϕ^n, which takes a lot of longer than the iterative (for loop) algorithm !

[14]Note however that the values we are computing grow so fast that we cannot consider basic arithmetic at constant cost anymore, but we ignore that for now.

Tower of Hanoi

Let's now turn to an example in which recursion allows us to express a solution in a very simple manner. Unlike in the previous example, the issue here is expressibility, rather than computational efficiency. (The algorithm here still takes a long time for large n, as we'll see a few lectures from now.)

The problem is called *Tower of Hanoi*. There are three stacks (towers) and a number n of disks of different radii. (See `http://en.wikipedia.org/wiki/Tower_of_Hanoi`). We start with the disks all on one stack, say stack 1, such that the size of disks on each stack decreases from bottom to top. The objective is to move the disks from the starting stack (1) to one of the other two stacks, say 2, while obeying the following rules:

1. A larger disk cannot be put on top of a smaller disk (at any time).
2. Each move consists of popping a disk from one stack and pushing it onto another stack, or more intuitively, taking the disk at the top of one stack and putting it on another stack. (You cannot pop more than one disk at any time. You must push it back somewhere before poping another one.)

The (remarkably simple) recursive algorithm for solving the problem goes as follows. The three stacks are labelled $s1, s2, s3$. One of the stacks is where the disks "start". Another stack is where the disks should all be at the "finish". The third stack "other" is the only remaining one.

```
tower(n, start, finish, other)
  if n>0 then
    tower(n-1, start, other, finish)
    move from start to finish          // i.e.  finish.push(  start.pop() )
    tower(n-1, other, finish, start)
  end if
```

For example, `tower(1,s1,s2,s3)` would produce to the following sequence of instructions:

```
tower(0,s1,s3,s2)       //  don't do anything
move from s1 to s3
tower(0,3,2,1)          //  don't do anything
```

The two calls `tower(0,*,*,*)` would do nothing since the condition $n > 0$ is not met.

```
move from 1 to 2
```

What about `tower(2,1,2,3)` ? This would produce the following sequence of instructions:

```
tower(1,1,3,2)
move from 1 to 2
tower(1,3,2,1)
```

and the two calls `tower(1,*,*,*)` would each move one disk, similar to the previous example (but with different parameters). So, in total there would be 3 moves:

```
move from 1 to 3
move from 1 to 2
move from 3 to 2
```

Here are the states of the tower for `tower(3,1,2,3)` and the corresponding print instructions. Notice that we need to do the following:

```
tower(2,1,3,2)
move from 1 to 2
tower(2,3,2,1)
```

So first we do `tower(2,1,3,2)`:

```
    *
    **
    ***
    ---       ---       ---    (initial)


    **
    ***      *
    ---      ---       ---     (after moving disk from 1 to 2)



    ***      *         **      (after moving disk from 1 to 3)
    ---      ---       ---


                       *
    ***                **      (after moving from 2 to 3)
    ---      ---       ---

which completes the tower(2, 1, 3, 2) call.
Next we do "move from 1 to 2":

                       *
             ***       **
    ---      ---       ---     (after moving from 1 to 2)




 Then we call  tower(2, 3, 2, 1)
 *        ***       **
---       ---       ---     (after moving from 3 to 1)


          **
 *        ***
---       ---       ---     (after moving from 3 to 2)


           *
           **
           ***
---       ---       ---     (after moving from 1 to 2)

and we are done!
```

For any $n \geq 0$, the `tower` algorithm is correct for n disks

For the algorithm to be "correct", we need to ensure that a larger disk is never place on top of a smaller disk, and that the n disks are moved from the start tour to the finish tour. The proof is by induction.
Base case: The rule is obviously obeyed if $n = 1$ and the algorithm simply moves the one disk from `start` to `finish`.
Induction step: Suppose the algorithm is correct if there are $n = k \geq 1$ disks *in the initial tower*. This is the induction hypothesis. We need to show that the algorithm is therefore correct if there are $n = k + 1$ disks *in the initial tower*. For $n = k + 1$, the algorithm has three steps, namely,

- `tower(k,start,other,finish))`
- `move from start to finish`
- `tower(k,other,finish,start)`

The first recursive call to `tower` moves k disks from *start* to *other*, while obeying the rule for these k disks (by the induction hypothesis). The second step moves the biggest disk ($k + 1$) from `start` to `finish`. This also obeys the rule, since `finish` does not contain any of the k smaller disks (because these smaller disks were all moved to the `other` tower). Finally, the second recursive call to `tower` move k disks from `other` to `finish`, while obeying the smaller-on-bigger rule (by the induction hypothesis). This completes the proof.

Mergesort

In lecture 5, we saw the "insertion sort" algorithm for sorting n items. We saw that, in the worst case, this algorithm requires $\frac{n(n-1)}{2}$ or $\frac{n^2}{2} - \frac{n}{2}$ or operations. As discussed in the lecture slides, n^2 can be very prohibitively large if the number of items to be sorted is too large. e.g. if $n = 2^{20} \approx 10^6$, then $n^2 \approx 10^{12}$. Today's machines run at about 10^9 operations per second (i.e. GHz), and so this means thousands of seconds to sort such a list (worst case).

We now consider an alternative sorting algorithm that runs much faster in the worst case. This algorithm is called *mergesort*. Here is the idea. If there is just one number to sort ($n = 1$), then do nothing. Otherwise, partition the array of n elements into two arrays of size about $n/2$ elements each, sort the two individual arrays (recurively, using mergesort), and then merge the two sorted arrays together.

For example, suppose we have an array

$$[8, 10, 3, 11, 6, 1, 9, 7, 13, 2, 5, 4, 12].$$

We partition it into two sub-arrays

$$[8, 10, 3, 11, 6, 1] \;\; [9, 7, 13, 2, 5, 4, 12].$$

and sort these (by applying mergesort recursively):

$$[1, 3, 6, 8, 10, 11] \;\; [2, 4, 5, 7, 9, 12, 13].$$

Then, we merge these two back together

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].$$

Algorithm: mergesort$(A, start, end)$
Input: Array A of elements and positions to be processed
Output: Sorted array

```
if ($start = end$) then
    initialize new array $B$ of $size \leftarrow 1$; $B[0] \leftarrow A[start]$; return $B$
else
    $mid \leftarrow (A.size - 1)/2$
    $A1 \leftarrow$ mergesort($A, start, mid$)
    $A2 \leftarrow$ mergesort($A, mid + 1, end$)
    return merge($A1, A2$)
end if
```

Algorithm: merge$(A1, A2)$
Input: Sorted sequences $A1$ and $A2$
Output: Sorted sequence B containing the elements from $A1$ and $A2$

```
initialize new array $B$ of $size \leftarrow A1.size + A2.size$
$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
while $i < A1.size$ & $j < A2.size$ do
  if $A1[i] < A2[j]$ then
    $B[k] \leftarrow A1[i]; i \leftarrow i + 1; k \leftarrow k + 1$
  else
    $B[k] \leftarrow A2[j]; j \leftarrow j + 1; k \leftarrow k + 1$
  end if
end while
while $i < A1.size$ do
  $B[k] \leftarrow A1[i]; i \leftarrow i + 1; k \leftarrow k + 1$
end while
while $j < A2.size$ do
  $B[k] \leftarrow A2[j]; j \leftarrow j + 1; k \leftarrow k + 1$
end while
return $B$
```

In order to process an entire array A one calls **mergesort**$(A, 0, A.length - 1)$. Note that this algorithm is not optimized for saving space as it uses a lot of extra space to sort this array: it creates an intermediary array for each recursive call of the size to be sorted.

There is however a clever way to organize these partitions and copies which allows you to just use one extra array (of the same size n as the original one). But the details on how to do that are not what I want to emphasize now, since they would obscure the more abstract ideas of the algorithm. So I have left them out. Similarly, for a linked list implementation, there would be details that one needs to address. One example is that the partitioning of a list into two sub-lists requires you to split the list at (roughly) the middle. But with a linked list, you don't have immediate access to the middle element. To find it, you have to scan for it by traversing the list from the beginning.

"mergesort is $\Theta(n \log n)$"

Another point to note is that there are $\log n$ levels of the recursion, namely the number of levels is the number of times that you can divide n by 2 until you reach 1 element per list. As we will discuss a few lectures from now, the mergesort algorithm requires about $n \log n$ steps, namely at each of the $\log n$ levels of the recursion, the total number of operations you need to do is proportional to n.

To appreciate the difference between the worst case number of operations for insertion sort (say n^2) versus the worst case[15] of mergeshort ($n \log n$), consider the following table.

[15]mergesort always takes $cn \log n$ operations, i.e. best case = worst case

n	$\log n$	$n \log n$	n^2
$10^3 \approx 2^{10}$	10	10^4	10^6
$10^6 \approx 2^{20}$	20	20×10^6	10^{12}
$10^9 \approx 2^{30}$	30	30×10^9	10^{18}
...	...	...	...

Thus, the time it takes to run mergesort becomes significantly less than the time it takes to run insertion sort, as n becomes large. Very roughly speaking, on a computer that runs 10^9 operations per second (which is typical these days), running mergesort on a problem of size $n = 10^9$ would take in the order of minutes, whereas running insertion sort would take centuries.