Recursion et friends.

Suppose you are given a 'rule' such as

$$a_n = n\, a_{n-1}$$

and also know that

$$a_1 = 1.$$

Read pages 106-108 and 618-622 of the Skansholm text

Then we can use this to build a sequence of numbers...

$$1\ ,\ 2\ ,\ 6\ ,\ 24\ ,\ 120\ ,\ 720\ ,\ \ldots$$

which you recognise as factorials. It's easy to see this by building up from the bottom...

$$1\ ,\ 2\times 1\ ,\ 3\times(2\times 1)\ ,\ 4\times(3\times 2\times 1)\ ,\ \ldots$$

$1 = a_1$, $2\times 1 = 2\,a_1$, $3\times(2\times 1) = 3\,a_2$, $4\times(3\times 2\times 1) = 4\,a_3$

Although showing that this will really only produce factorials would take an infinite amount of time!!

We could prove this in an intuitively rigourous inductive way by . . .

① Remark that $a_1 = 1 = (1!)$.

② Notice that (if) we were to assume that

$$a_n = n!$$

then

$$a_{n+1} = (n+1) \cdot a_n \quad \leftarrow \text{by our 'rule'}$$

$$= (n+1) \cdot (n!) \quad \leftarrow \text{by our assumption}$$

$$= (n+1)!$$

This is rather like saying . . .

① I can put my foot on the first rung of a ladder.

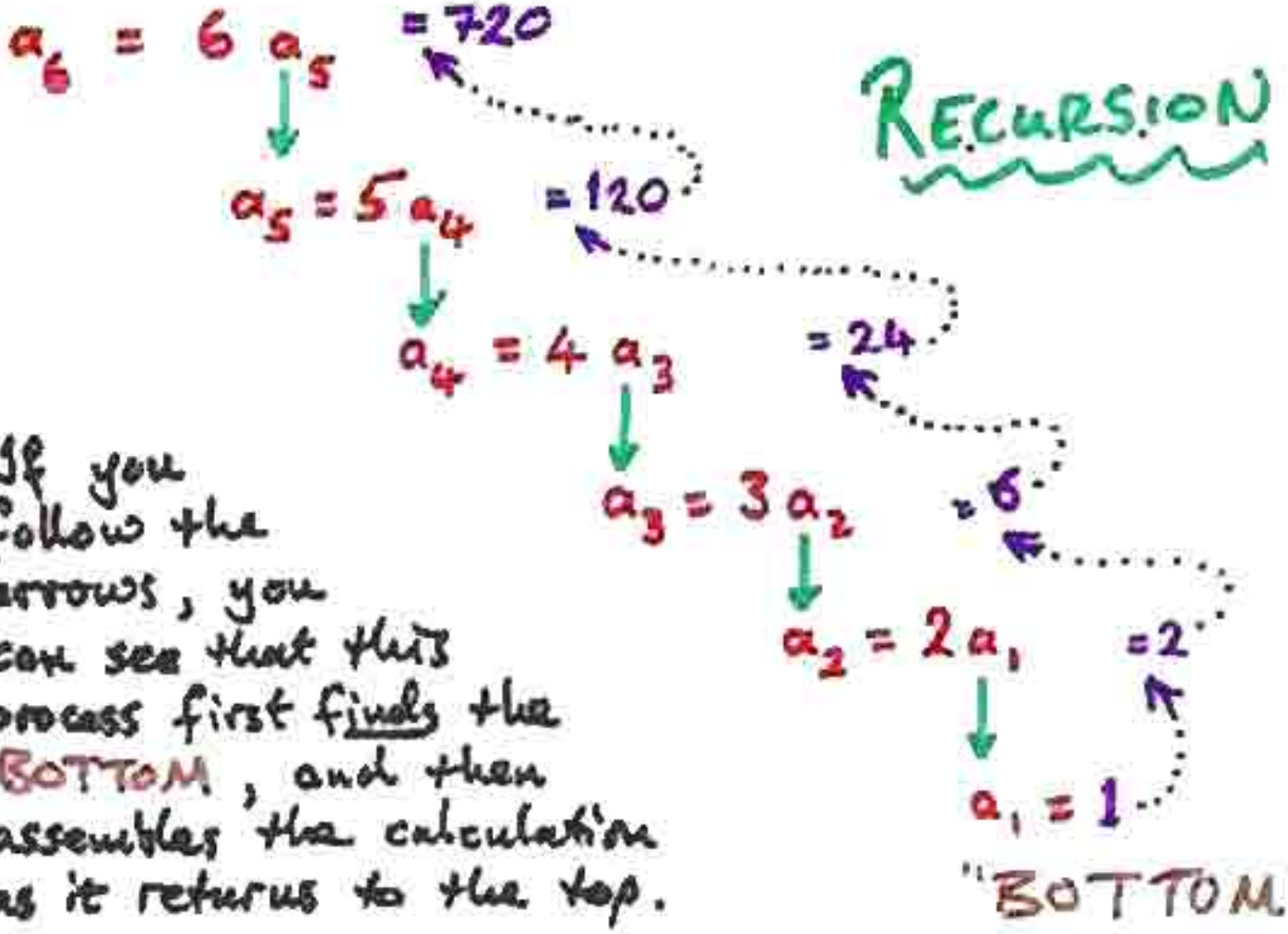
② IF I'm on any rung of the ladder THEN I can step onto the next rung.

This way of arguing, called **induction**, is very nice because...

a/ We don't have to do infinitely many steps.

b/ It's "jolly obvious" that we've covered every case.

Instead of working from the bottom up, we could work from the top down (provided our 'top' is only 'finitely high')....



If you follow the arrows, you can see that this process first finds the BOTTOM, and then assembles the calculation as it returns to the top.

Obviously, if there is no bottom, then we will be waiting a jolly long time for any results!

Let's see what the coded version of this looks like ...

```
public static int fact (int n)
        {
            if (n == 1) return 1;
            else        return n * fact (n-1);
        }
```

Then our fact(n) behaves just like our a_n, and it would be invoked by ...

```
int ans = fact (6);
```

for example — producing the same bottom-hungry routine we saw for a_n.

In fact, any sequence defined by a recurrence relation can be converted into recursive code very easily. Without making any comments about efficiency, recursive code is typically very short.

As experiments, first you should run the above code to compute fact(20).

After that, try to find the 100th term in the following Fibonacci-like sequence ...

$$a_n = 2\,a_{n-1} - 3\,a_{n-2}$$

$$a_1 = 1 \quad , \quad a_2 = 1$$

by running the following code ...


```
fibby ( n )
  {
    if (n == 1)
          return 1;
    else if (n == 2)
          return 1;
    else
      return 2*fibby(n-1) - 3*fibby(n-2);
  }
```

Obviously, in both this and the preceding factorial example, you will have to add the appropriate extra stuff to make this into a real Java program — and don't forget to catch the exceptional cases where $n \leq 0$ or is not an integer! (You might find the second example runs a little slowly.)

As you can see, writing this kind of code in these examples is pretty easy.