CI583: Data Structures and Operating Systems
Recursion, and some recursive problems

Outline

Recursion

Recursion occurs when an algorithm (or method, or function), A, requires us to execute A again, repeatedly.

So that the execution of A does not diverge (run forever) we need to define conditions under which the recursion will end.

We call this the base case: the conditions under which the recursion continues are called the recursive case.

Recursion is closely linked to mathematical induction.

Recursion

Any iterative algorithm can be expressed via recursion, and vice versa.

There are (Turing complete) programming languages which don't include any construct for looping, such as Haskell.

Sometimes an iterative solution is the clearest, but recursive code is often more concise and expresses the solution to the problem elegantly and directly.

Recursion and iteration are more general constructs than the algorithmic strategies we considered in the last lecture (greedy algorithms, dynamic programming, etc).

Recursion

We have seen recursion in OO style, when traversing tree structures:

```
1  class Node extends Tree {
2    void traverse() {
3      left.traverse();
4      this.visit();
5      right.traverse();
6    }
7  }
8  class Leaf extends Tree {
9    void traverse() {
10     this.visit();
11   }
12 }
```

Recursion

We can also use it directly within a single method. A simple (naive, in fact) way to compute the nth triangular number (the nth triangular number is found by adding n to the $(n-1)$th, so the sequence runs `[1, 3, 6, 10 ...]`):

```
1  int triangle(int n) {
2    if (n == 1) return 1;
3    else return n + triangle(n - 1);
4  }
```

The recursive call should normally be made with smaller input, so that we know the algorithm will terminate. In some recursive algorithms the input might grow before it shrinks but, obviously, shrink it must.

Recursion

The reason `triangle` can be considered naive is that it will use much more memory at runtime than an iterative solution. In fact, it could cause stack overflow errors for even fairly small values of n. Look at what happens when we run it:

```
1  triangle(5)
2  if (5==1) 1 else 5 + triangle(5-1)
3  if (false) 1 else 5 + triangle(5-1)
4  5 + triangle(4)
5  5 + (if (4==1) 1 else 4 + triangle(4-1))
6  5 + (4 + (if (3==1) 1 else 3 + triangle(3-1)))
7  5 + (4 + (3 + (if (2==1) 1 else 2 + triangle(2-1))))
8  5 + (4 + (3 + (2 + (if (1==1) 1 else 1 + triangle(1-1)
       ))))
9  5 + (4 + (3 + (2 + 1)))
10 15
```

Recursion

The solution to this is to make `triangle` tail-recursive, meaning that the recursive call is the last thing the method does.

In this way, we don't need to keep intermediate values hanging around.

Fortunately, every non-tail-recursive algorithm can be rewritten to a tail-recursive version.

Unfortunately, the resulting code is often much less intuitive.

Recursion

In terms of the algorithmic strategies from the last lecture, recursion is particular important in divide-and-conquer strategies.

These are strategies in which we divide the problem into two parts to solved separately (or discard one of them).

An example would be Binary Search, though the algorithm we gave in week 1 was iterative.

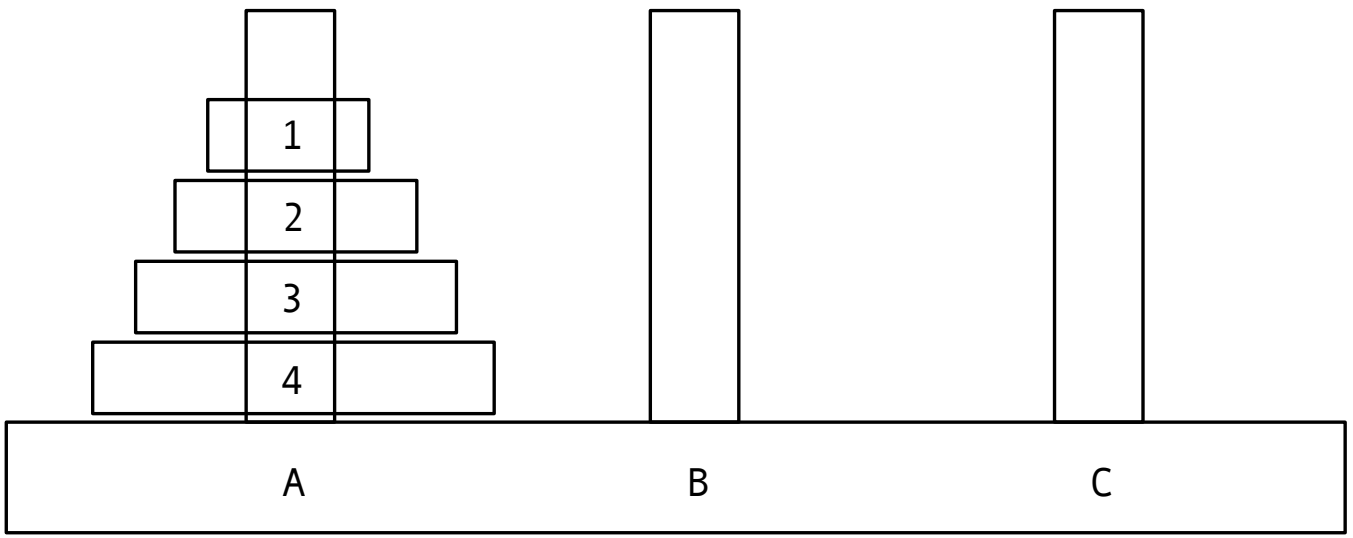
We will look at two efficient sorting algorithms that use a divide-and-conquer approach, MergeSort and QuickSort.

The Towers of Hanoi

The Towers of Hanoi is an ancient puzzle. Move the discs from the first peg to the last. You may only move one disc at a time, and you can never put a disc on top of a smaller one. Solving it manually, a pattern soon emerges – see the applet on studentcentral.

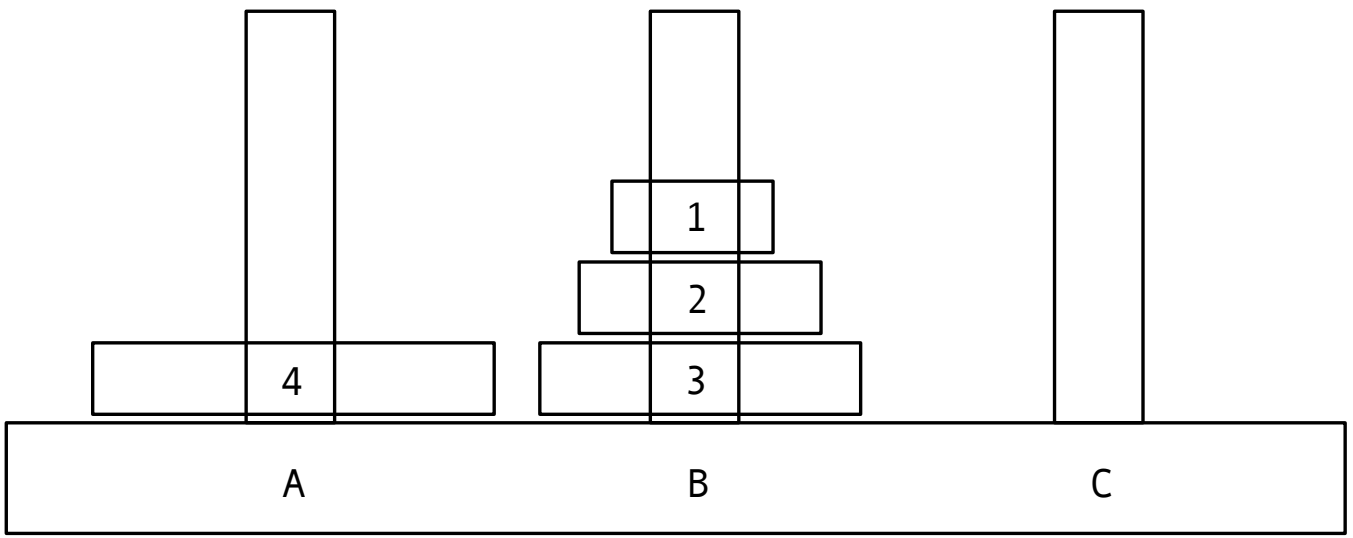
The Towers of Hanoi

We will call the set of all discs, in their right order, the stack.
(Nothing to do with the data structure.)



The Towers of Hanoi

If you solve the problem manually you will find that intermediate, smaller substacks form during the process of solving the puzzle. The only way to transfer disc 4 to tower C is to move everything on top of it out of the way!



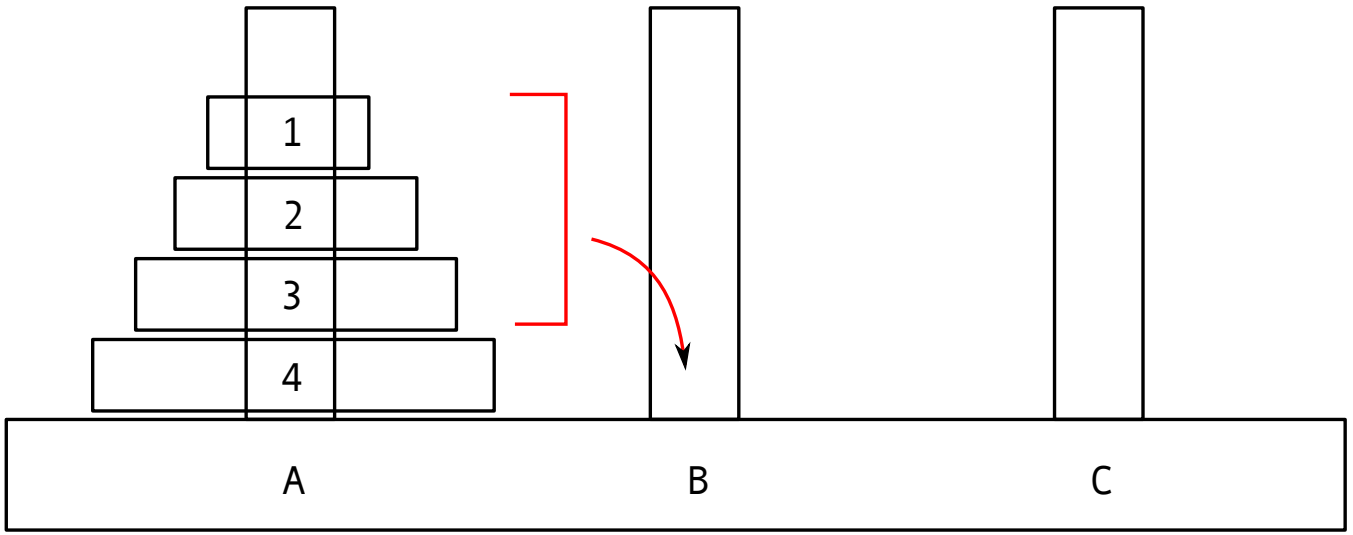
The Towers of Hanoi

A recursive solution to move n discs from a source tower, S, to a destination tower, D, via an intermediate tower, I:

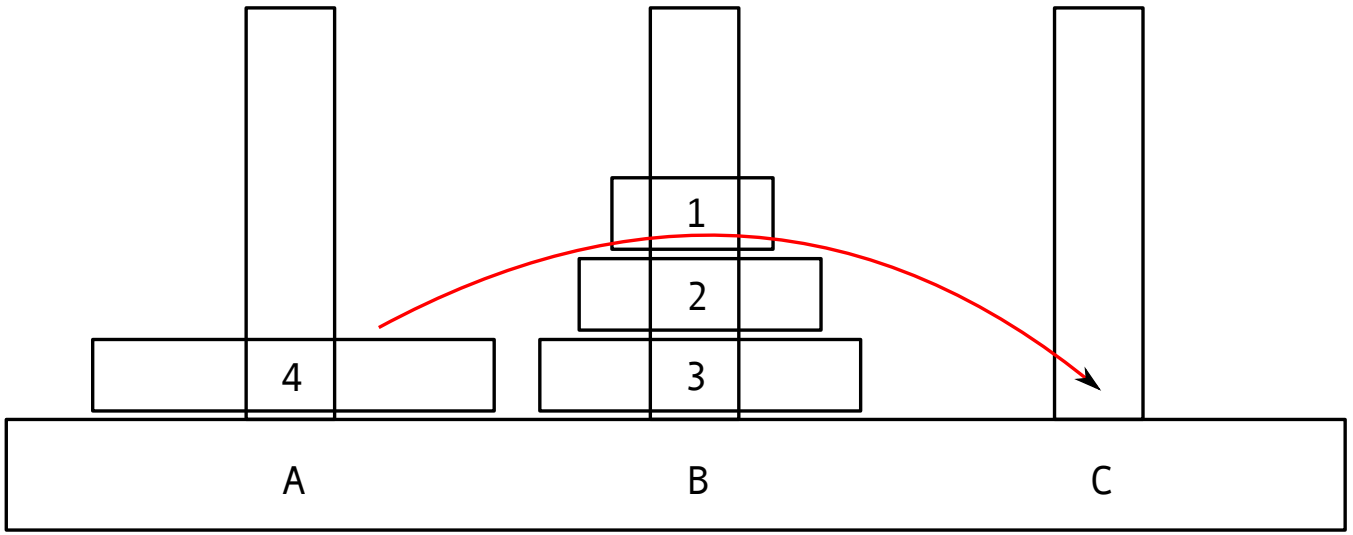
1. Move the substack of $n-1$ discs from S to I.
2. Move the largest disc from S to D.
3. Move the substack from I to D.

(Recursive definitions feel like cheating sometimes!) When we begin, $n = 4$, $S = A$, $I = B$ and $D = C$.

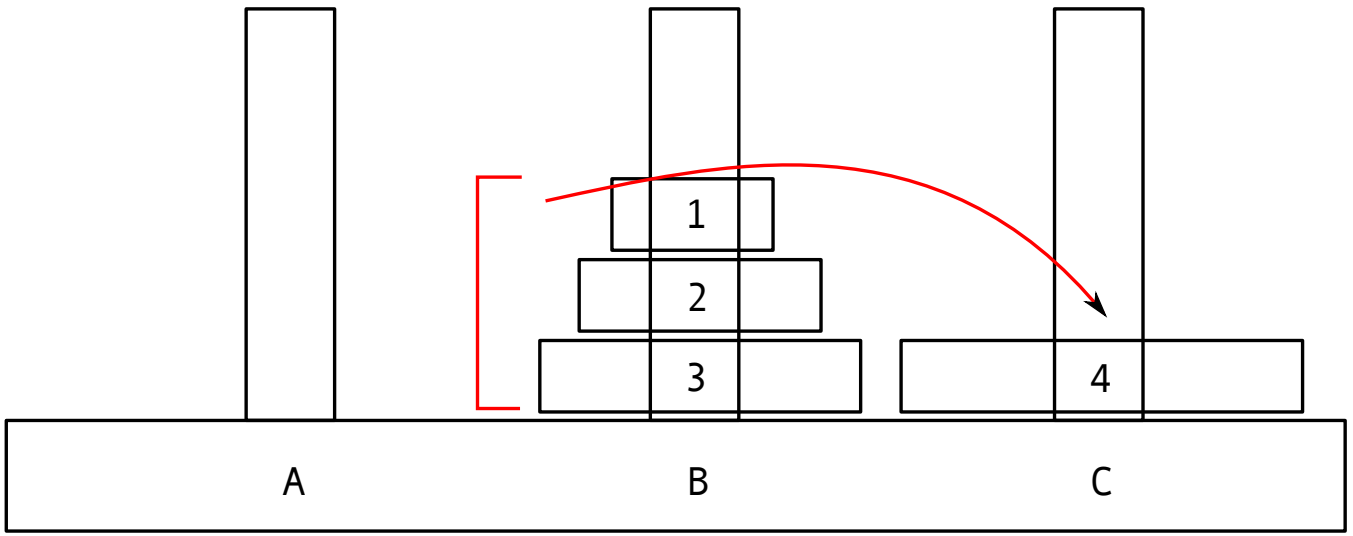
The Towers of Hanoi



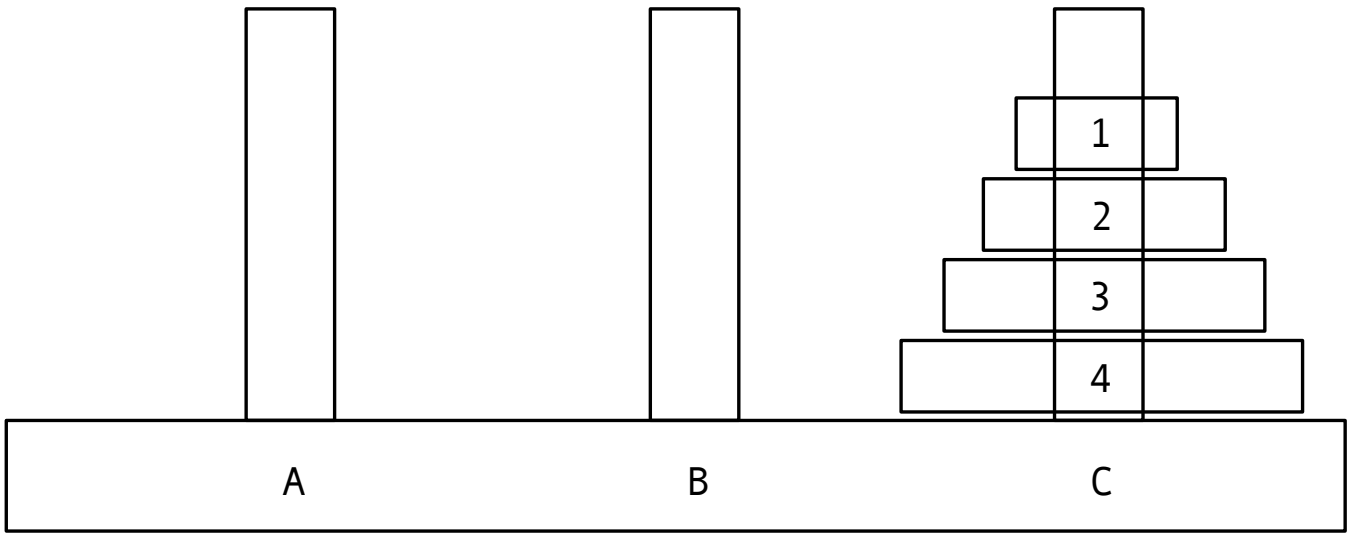
The Towers of Hanoi



The Towers of Hanoi



The Towers of Hanoi



The Towers of Hanoi

But we can't move the substack of discs [1,2,3] all at once. Still, it's easier than moving 4 discs...

In fact, we can move the three-disc substack from A to B in the same way as moving the entire stack but for different values of S, I and D: move substack [1,2] to intermediate tower C, then 3 to destination tower B, then substack [1,2] from C to B.

The problem is getting smaller...

The Towers of Hanoi

How do we move the substack [1,2] from A to C?

This one is quite easy: move 1 to B, then 2 to C, then 1 to C.

This is the base case: note that we have said exactly what we will do without hand-waving like "move the substack..."

The Towers of Hanoi

A recursive solution in Java:

```
1  class TowersApp {
2    static int nDiscs = 3;
3
4    public static void main(String[] args) {
5      doTowers(nDiscs, 'A', 'B', 'C');
6    }
7    //...
8  }
```

The Towers of Hanoi

A recursive solution in Java:

```
1  class TowersApp {
2    //...
3    public static void doTowers(int n, char from, char
         inter, char to) {
4      if (n==1) {
5        System.out.printf("Disc 1 from %c to %c\n", from
             , to);
6      } else {
7        doTowers(n-1, from, to, inter);//swap from and
             inter
8        System.out.printf("Disc %d from %c to %c\n", n,
             from, to);
9        doTowers(n-1, inter, from, to);//swap inter and
             to
10     }
11   }
12 }
```

The Towers of Hanoi

It seems astonishing at first that a problem that seems like it should be quite complicated can be solved with so little code!

This is certainly a case of the recursive solution being elegant and concise.

We can of course produce an iterative solution to the Towers of Hanoi problem, but it is longer and less clear (to my mind) what is going on.