COMP 1020 - Recursion

UNIT 6

Recursion

- The basic concept of recursion is:

Recursion

- The basic concept of recursion is:
  - A method can call itself

Recursion

- The basic concept of recursion is:
  - A method can call itself
    - That's it?

Recursion

- The basic concept of recursion is:
  - A method can call itself
    - That's it?
    - Yes, that's it!

The traditional example

- n factorial (n!) can be defined and programmed using iteration:
- n! = n*(n-1)*(n-2)...*1

```
public static long fact(int n) {
long nfact=1;
if (n == 1 || n == 0)
      return nfact;
else{
      for(int i=n; i>0; i--)
            nfact *= i;
      return nfact;    }}
```

iterative approach (i.e. uses a loop)

The traditional example

- Or n factorial (n!) can be defined and programmed using recursion:

  $n! = 1$ (if $n \leq 1$)

  $n! = n(n-1)!$ (if $n>1$)

```
public static long fact(int n) {
    if(n==1||n==0)
        return 1;
    else
        return n * fact(n-1);
}
```

recursive approach (i.e. method calls itself)

see Factorial.java

The base / easy case

- Any recursive method
  - Cannot always call itself, or else the chain of calls will never end → infinite recursion
    - Always results in a "stack overflow"
    - The recursive equivalent of an infinite loop
- You need a “stop condition” → a base case where you know the answer and can stop calling yourself

The base / easy case

- Must have some easy, non-recursive "base case" or "easy case"

- The recursive calls must always lead, sooner or later, to this base case

- Try omitting the if(n<=1) case from the fact method

The base / easy case

- Must have some easy, non-recursive "base case" or "easy case"

- The recursive calls must always lead, sooner or later, to this base case

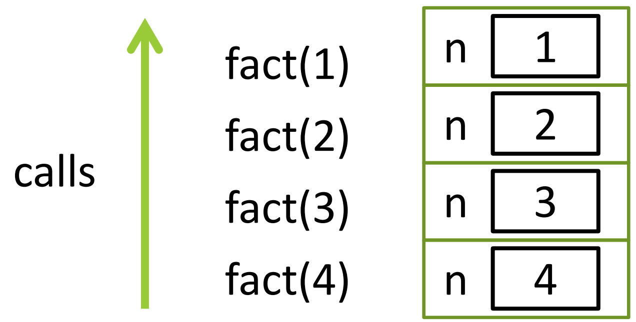
- Try omitting the if(n<=1) case from the fact method
  - Stack Overflow!

How recursion works

- Each time a method is called, a whole new set of local variables (including parameters) are created
  - Many instances of one method can all be running at the same time, each with its own variables
  - These sets of variables are stored on the stack

How recursion works

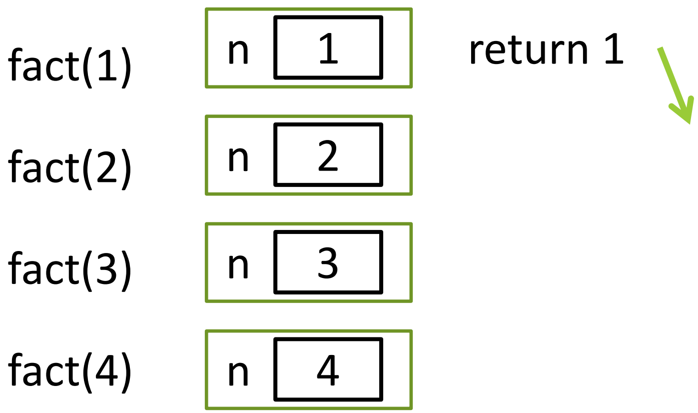
- If you call fact(4), the stack will become:



- There are four separate versions of the fact method running, each with its own parameter n

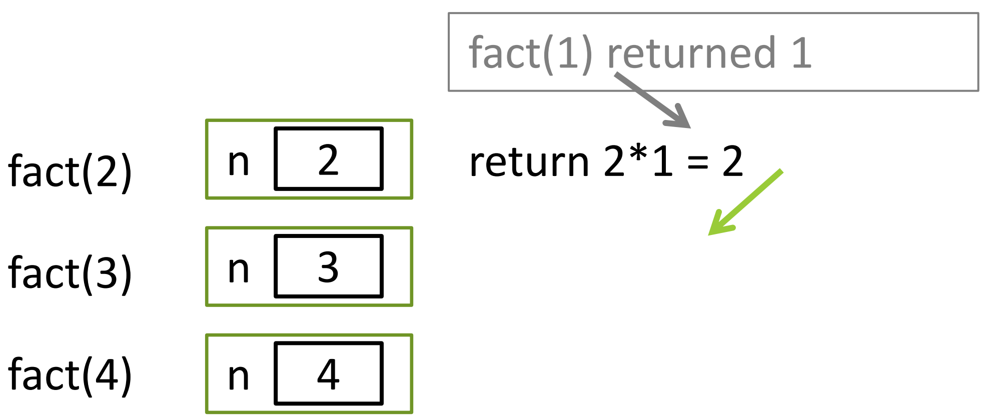
How recursion works

- Often, as in this method, the answer is built up as each method returns an answer to the previous one



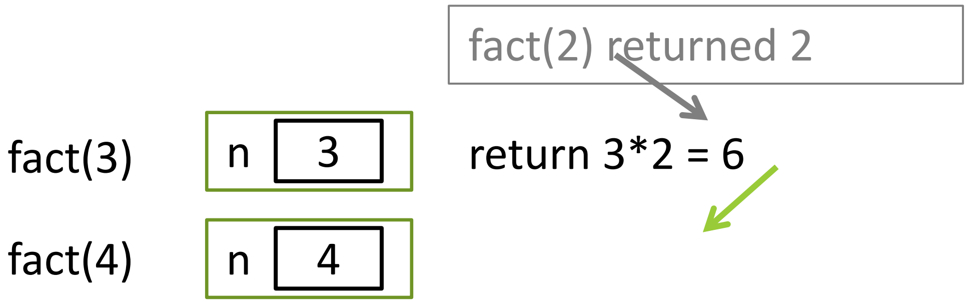
How recursion works

- Often, as in this method, the answer is built up as each method returns an answer to the previous one



How recursion works

- Often, as in this method, the answer is built up as each method returns an answer to the previous one



How recursion works

- Often, as in this method, the answer is built up as each method returns an answer to the previous one

fact(3) returned 6

fact(4) n 4 return 4*6 = 24

fact(4) gives 24

How recursion works

- Often, as in this method, the answer is built up as each method returns an answer to the previous one

fact(4) returned 24

result of fact(4) is 24

Don't worry about the stack

- You don't need to visualize the stack, and all of the calls, and all of the returns

    - The previous slides were just there to help you understand how recursion works and how the stack keeps track of all the recursive calls

Don't worry about the stack

- All you have to do to write a recursive method is focus on these two things:

  1) Find a way to solve the problem by using the solution to a slightly smaller version of the same problem

  2) Find an easy case (base case) that will always be reached if the problem keeps getting smaller

Don't worry about the stack

- All you have to do to write a recursive method is focus on these two things:

  1) Find a way to solve the problem by using the solution to a slightly smaller version of the same problem

  2) Find an easy case (base case) that will always be reached if the problem keeps getting smaller

- While writing methodX, simply assume that methodX will always work on any smaller case. Trust it.

Why are we doing this?

Why are we doing this?

- Using recursion can simplify a lot some problems
  - some problems are very complex and difficult to solve iteratively (using a loop)
  - a recursive approach can be much simpler, so it's an important tool to have in your toolbox

Why are we doing this?

- Using recursion can simplify a lot some problems
  - some problems are very complex and difficult to solve iteratively (using a loop)
  - a recursive approach can be much simpler, so it’s an important tool to have in your toolbox

- Recursive code can be much shorter than iterative code
  - keep in mind that shorter code is not always better code though (i.e. not necessarily faster; we’ll see an example of that)

Example: sum of an Array

- Consider a method

```
public static int sumOf(int[] data){
    //Returns the sum of all elements of an array
    //(0 if there are no elements)
}
```

- You know how to do this iteratively (just need a for loop)

- Can this be written recursively?

Example: sum of an Array

public static int sumOf(int[] data)

- Can this be written recursively?

  - Almost (pseudocode):

```
if data.length is 0
    return 0
else if data.length is n>0
    return data[n-1] + the sum of the first n-1 elements
```

Example: sum of an Array

public static int sumOf(int[] data)

- Can this be written recursively?

  - Almost (pseudocode):

```
if data.length is 0
    return 0
else if data.length is n>0
    return data[n-1] + the sum of the first n-1 elements
```

  - But the method signature as defined above can't sum only the first n-1 elements – it can only sum the entire list

Example: sum of an Array

```
public static int sumOf(int[] data)
```

- Can this be written recursively?

  - But a more general version could be recursive:

    ```
    public static int sumOf(int[] data, int n) {
    //Returns the sum of the first n elements of data
    //(0 if n=0)
    ```

  - This is common. Sometimes you have to add some extra parameters before you can use recursion.

Array sum example

```
public static int sumOf(int[] data, int n) {
    //Returns the sum of the first n elements of data
    //(0 if n=0)
    if(n==0)
        return 0;
    else
        return sumOf(data,n-1) + data[n-1];
 }

//Don't forget the original goal was to write:
public static int sumOf(int[] data){
    //Returns the sum of all elements of an array
    //(0 if there are no elements)
    return sumOf(data,data.length);
}
```

see ArraySum.java

Array sum example

```
public static int sumOf(int[] data, int n) {
    //Returns the sum of the first n elements of data
    //(0 if n=0)
    if(n==0)
        return 0;
    else
        return sumOf(data,n-1) + data[n-1];
}

//Don't forget the original goal was to write:
public static int sumOf(int[] data){
    //Returns the sum of all elements of an array
    //(0 if there are no elements)
    return sumOf(data,data.length);
}
```

The goal of this method is to "hide" from the user the additional parameter (int n) necessary for the recursive method to work

see ArraySum.java

Array sum example

```
public static int sumOf(int[] data, int n) {
    //Returns the sum of the first n elements of data
    //(0 if n=0)
    if(n==0)
        return 0;
    else
        return sumOf(data,n-1) + data[n-1];
}

//Don't forget the original goal was to write:
public static int sumOf(int[] data){
    //Returns the sum of all elements of an array
    //(0 if there are no elements)
    return sumOf(data,data.length);
}
```

<u>This is the "interface" that we want to provide to the user</u>. The user shouldn't have to worry about how the recursion is implemented.

see ArraySum.java

Array sum example

```
private static int sumOf(int[] data, int n) {
    //Returns the sum of the first n elements of data
    //(0 if n=0)
    if(n==0)
        return 0;
    else
        return sumOf(data,n-1) + data[n-1];
 }
```

This is the method with extra parameters used internally for recursion. We should hide it from the user by making it private.

```
//Don't forget the original goal was to write:
public static int sumOf(int[] data){
    //Returns the sum of all elements of an array
    //(0 if there are no elements)
    return sumOf(data,data.length);
}
```

<u>This is the "interface" that we want to provide to the user</u>. The user shouldn't have to worry about how the recursion is implemented.

see ArraySum.java

One more example

- How can you convert an integer n>0 into a binary number?
  - We'll store it as an ArrayList<Integer> containing only 0's and 1's

- A simple recursive method:
  - If n=0 give an empty list (the base case)
  - If n>0 then
    - Convert n/2 to binary
    - Add a 0 to the end if n is even, or a 1 if n is odd
      - In other words, add n%2 to the end

One more example

```
public static ArrayList<Integer> binary(int n){
    if(n==0)
        return new ArrayList<Integer>();
    else {
        ArrayList<Integer> bin = binary(n/2);
        bin.add(n%2);
        return bin;
    }
}
```

see BinaryConversion.java

Iterative vs recursive

- For all of the previous examples, there is an easy iterative solution (using loops)
  - There is no clear advantage of choosing an approach over the other
- When a method contains multiple recursive calls, then there can be big advantages

Example: Towers of Hanoi

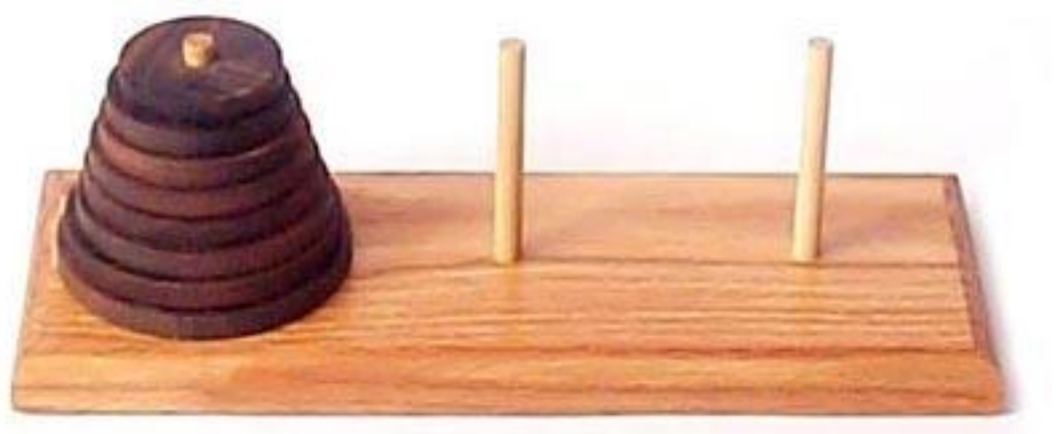


Image: woodenpuzzle.com

- Objective: Move the disks to the right (or center) peg
  - Only one can be moved at a time
  - Only the top one from one peg can be moved
  - No disk can ever be placed on a smaller one
- Good animation: towersofhanoi.info/Animate.aspx

Towers of Hanoi program

- Recursively, a solution is easy:
  - To move disks 1 to n from peg A to peg B (using peg C):
    - If n=1 just move disk 1 (base case)
    - Otherwise
      - Move disks 1 to (n-1) from A to C (using B temporarily)
      - Move disk n from A to B
      - Move disks 1 to (n-1) from C to B (using A temporarily)

Towers of Hanoi program

- Recursively, a solution is easy:

means move n discs from A to B using C

```
public static void solveHanoi(int n, String A, String B, String C)
{
    if(n==1) System.out.println("move 1 from "+A+" to "+B);
    else {
        solveHanoi(n-1,A,C,B);
        System.out.println("move "+n+" from "+A+" to "+B);
        solveHanoi(n-1,C,B,A);
    }
}
```

Binary search (recursive)

- Consider a method

```
public static int binSearch(int[] data, int key){
//Returns the index of key in data, or -1 if not there
```

- Can this be written recursively, with those parameters?

Binary search (recursive)

- Consider a method

```
public static int binSearch(int[] data, int key){
//Returns the index of key in data, or -1 if not there
```

- Can this be written recursively, with those parameters?
  - No, not quite
  - We would have to use it to search only the first half, or last half, of the array
  - As it is, it can't do that
  - This method signature can only search an entire array

Binary search (recursive)

- But a more general version could be recursive:

```
public static int binSearch(int[] data, int lo, int hi, int key)
{
    //Search the portion from data[lo] to data[hi] only.
    //Return the index of key in this range, or -1.
```

Let’s write the code for this!

Binary search code (recursive)

```
private static int binSearch(int[] data, int lo, int hi, int key){
	if(hi<lo) //There must be an easy non-recursive case
		return -1;
	else {
		int mid = (lo+hi)/2;
		if(data[mid]==key)
			return mid;
		else if(data[mid]<key)
			return binSearch(data,mid+1,hi,key); //Search top half
		else
			return binSearch(data,lo,mid-1,key); //Search bottom half
} }
public static int binSearch(int[] data, int key) {  //interface for the user
	return binSearch(data,0,data.length-1,key);
}
```

To use recursion

- Assume you can solve case n-1 of the problem. Just trust that it will work!

- Find a way to turn that into a solution for case n of the problem

- Add an easy non-recursive solution for the smallest possible n (base case)

- You're done. Program it!

To use recursion

- One pitfall to beware of:
  - Every instance of the method must be independent
  - With its own complete set of variables (received as parameters) → you need to receive parameters: that's how you can send information to each recursive call!

Complex example: Permutations

- Recursion is a particularly powerful way to generate all possible ways to do something
  - Great for permutations, subsets, game strategies, mazes, space filling, fractals, etc. etc.

- Example: Generate all possible permutations of an ArrayList
  - producing a list of ArrayLists – stored as an ArrayList
    - That's a 2D ArrayList!

Permutations

- Where's the recursion?

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
...
2 4 3 1
3 ...
4 ...
```

All permutations of 1 2 3 4

Permutations

- Where's the recursion?

1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
. . .
2 4 3 1
3 . . .
4 . . .

All permutations of 1 2 3 4

All permutations of 2 3 4

All permutations of 1 3 4

All permutations of 1 2 4

All permutations of 1 2 3

Permutations

- The recursive case (pseudocode):

```
for each element in the list
   choose that element to be first
   find all the sub-permutations of the rest
   add that first element to all the sub-permutations
   add them all to the list of solutions
```

Permutations

- What's the easy case?
  - The size of the list gets smaller with every call
  - The size must eventually reach 1
  - That's trivial and requires no recursion (there's only 1 permutation of size 1)
    - But be careful! You have to generate a list of 1 solution, and that 1 solution is a list of the 1 element.
    - It still must be a “2D ArrayList”. Always.

see Permutations.java

Subsets (or combinations)

- What about choosing all possible subsets of k things chosen from n things?
- Example: choose 3 things from 5 things.

  1 2 3
  1 2 4
  1 2 5
  1 3 4
  1 3 5
  1 4 5
  2 3 4
  2 3 5
  2 4 5
  3 4 5
- Where's the recursion?

Subsets (or combinations)

- What about choosing all possible subsets of k things chosen from n things?
- Example: choose 3 things from 5 things.

1 2 3
1 2 4
1 2 5 — 2 things from 2 3 4 5
1 3 4
1 3 5
1 4 5 Either you include the 1

2 3 4 Or you don't
2 3 5
2 4 5 — 3 things from 2 3 4 5
3 4 5

- Where's the recursion?

Subsets

- The main recursive case:
  - To choose k things from n things:
    1. Choose k-1 things from n-1 things
       - Tack on the first thing to all of these
    2. Choose k things from n-1 things

Subsets

- What's the easy case? Not as simple as usual.
  - In the top recursion (option 1), we'd go from:
    - 3 of 5 ⇒ 2 of 4 ⇒ 1 of 3 ⇒ 0 of 2.
      - Stop there! There's one solution to take 0 of anything.
      - Cannot continue to -1 of 1!
  - But in the bottom recursion (option 2), we'd go from:
    - 3 of 5 ⇒ 3 of 4 ⇒ 3 of 3 ⇒ 3 of 2.
      - Stop! 3 of 2 is clearly impossible. No solutions! Quit!

Subsets

- This time, we'll simply print the results as we get them
- Programming it can be a bit tricky for two reasons:
  - The main problem is not directly recursive any more. We have to pass the things we've already chosen as an extra parameter.

Subsets

- This time, we'll simply print the results as we get them
- Programming it can be a bit tricky for two reasons:
  - The main problem is not directly recursive any more. We have to pass the things we've already chosen as an extra parameter.
  - When we shrink the list, we have to make sure that the original list is NOT affected!
    - DON'T DESTROY YOUR PARAMETERS!
    - Objects can easily be changed! (Integer and String are OK, because immutable)

see Subsets.java

When recursion turns bad

- Fibonacci numbers are usually defined recursively:

  $\text{fib}(0) = \text{fib}(1) = 1$

  $\text{fib}(n) = \text{fib}(n-2)+\text{fib}(n-1)$ [for $n \geq 2$]

When recursion turns bad

- This is very easily programmed recursively:

```
public static long fibR(int n){
    if(n<=1)
        return 1;
    else
        return fibR(n-2)+fibR(n-1);
}
```

- But it's a horrible way to do it!

- Run FibonacciRecursive.java (before the next slide)

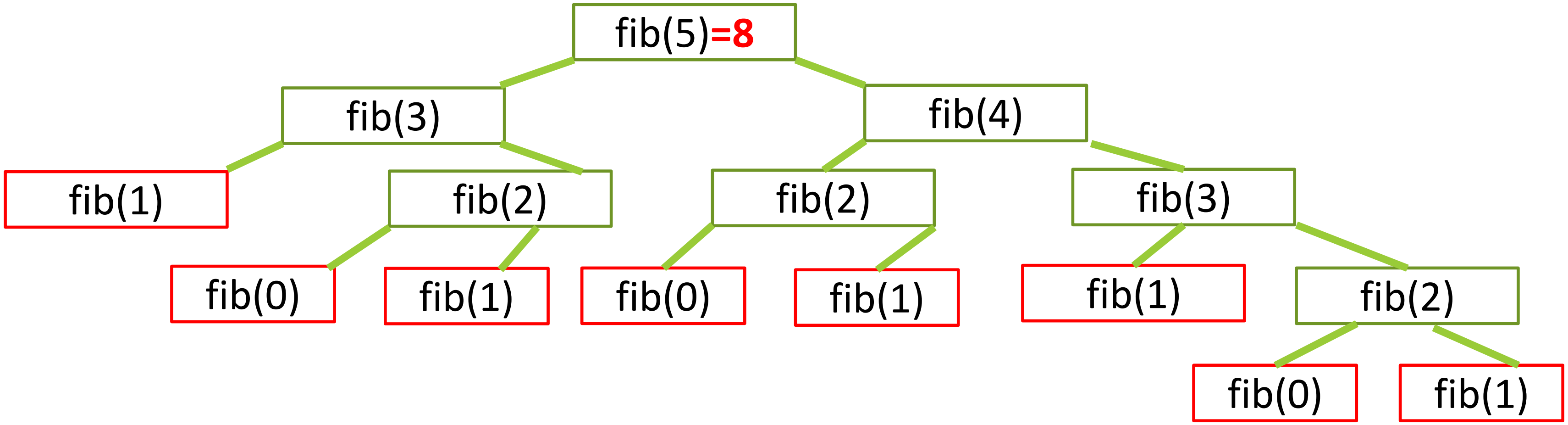
Why is it so slow?

- For fib(91)=7540113804746346429
  - A loop will find it in 0 sec
  - Recursion will take 762,142 years (roughly)

Why is it so slow?

- For fib(91)=7540113804746346429
  - A loop will find it in 0 sec
  - Recursion will take 762,142 years (roughly)

- Look at the calls that are made:



- To get an answer for fib(5), the base case must be called 8 times! (So for fib(91) it would be.....OMG....)

Why is it so slow?

- There is actually a way of implementing a recursive Fibonacci that will perform just like the iterative method

- We just need extra parameters that will allow us to transfer the previous two values

- See the “fibRecGood” method in FibonacciRecursive.java

Speed of algorithms

- The simplicity or complexity of the code is not an indication of the speed of the algorithm!

- You have to understand and analyze the number of actions/steps/operations that an algorithm will do

- This can require significant amounts of mathematical analysis at times

- We'll do a little bit in the next part of COMP 1020

- For the real story, take COMP 2080

Recursion: take-home message

- Building a recursive method involves answering two questions:
  - How can I solve a big problem by solving smaller instances of the same problem?
  - How can I stop the recursion with a simple/easy/base case, for which I know the answer?