COSC 1P03- Introduction to Data Structures

Recursion

Instructor: Naser Ezzati

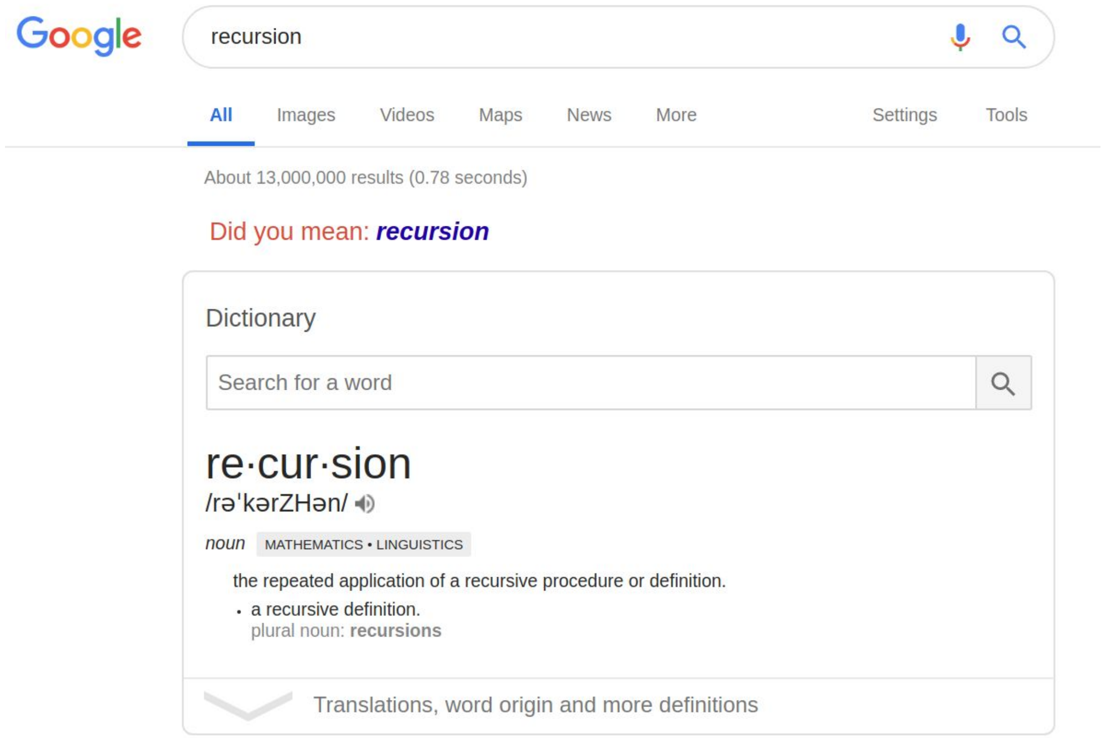
Brock University
www.github.com/naser/ubrock

Objectives

- What recursion is?
- How to write recursive algorithms?
- When to use recursion?

Definition

- Recursion is a problem solving approach
- Recursion solves a problem by reducing it to smaller subproblems;



Example1: Your Spot Number?

The Diagram

- Ask and wait
  - Ask and wait
    - Ask and wait …
      - ... first in line is reached. Tell person behind it is 0.
    - Tell person behind it is 0+1
  - Tell person behind it is 1+1 …
- Tell Person behind it is x +1

Example 2: Ground a Stone

BreakStone: You want to break a stone into dust (very small stones)

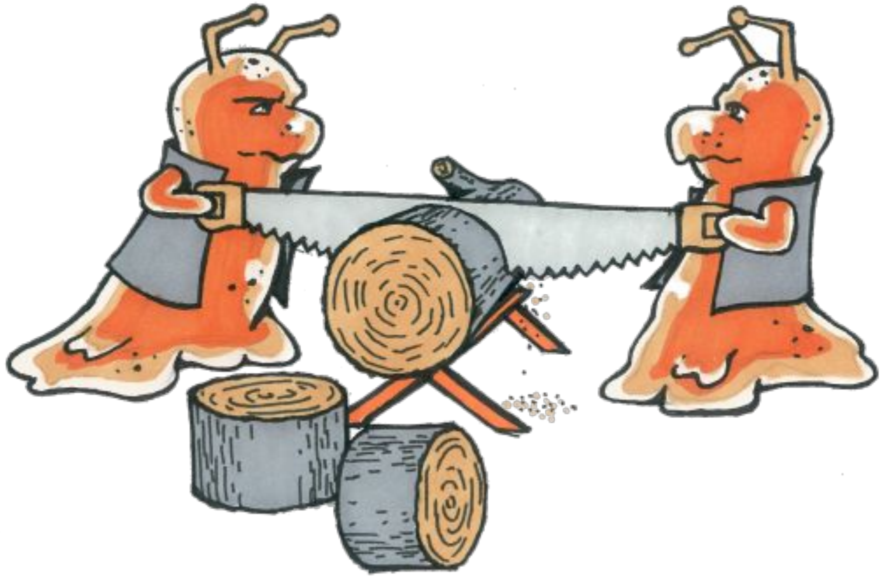
- Use a hammer and strike the stone
- If a resulting piece is small enough, we are done with that part
- For pieces that are too large, repeat the **BreakStone** process

Example 3: Fundraising

- Problem: Collect $10,000.00 for charity
  - Assumption: everyone is willing to donate $1
- Iterative Solution:
  - Find 10,000 people
- Recursive:
  - Donate your share and ask a friend to donate the rest!

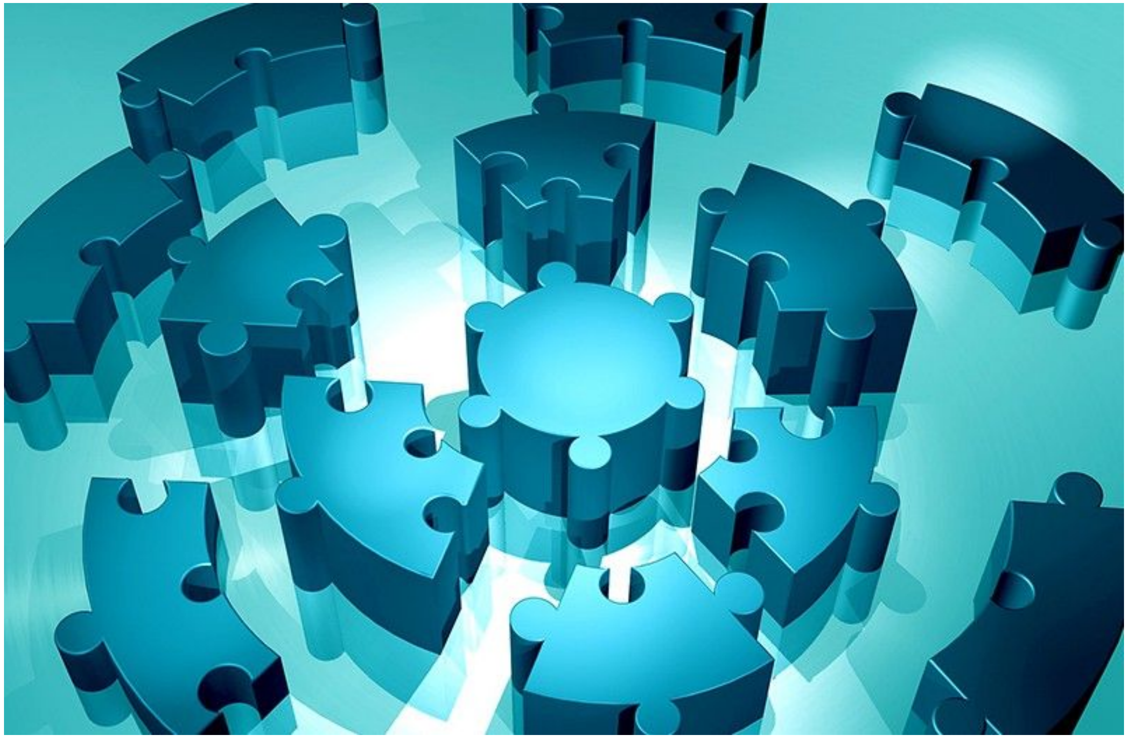
Definition

- Recursion is a problem-solving approach
  - Generates simple solutions to certain kinds of problems
- Recursive algorithm splits a complex problem
  - Into one or more simpler versions of itself



Recursion

Recursive solutions solve a problem by applying the same algorithm to each piece and then combining the results.



Common Elements

- If the problem is small enough to be solved directly, do it.
- If not, find a smaller problem and use its solution to create the solution to the larger problem

Common Elements

- If the problem is small enough to be solved directly, do it.
  - Base Case: The point where you stop applying the recursive case
    - *Person 0. You do not do execute the above.*
- If not, find a smaller problem and use its solution to create the solution to the larger problem
  - Recursive Case: The set of instructions that will be used over and over
    - *Tap person in front of you. Ask #people in front of him. Wait for his answer and add 1.*

Common Elements: Code

Method (…) {

If *the problem is small enough {*

- *solve it directly* ***without recursion****.*

}

else *{*

- ***Recurs with a simpler subproblem***
  - ***Divide*** *the problem into one or more subproblems that have the same form.*
  - ***Solve*** *each of the problems by calling this method* ***recursively****.*
- ***Combine*** *and return the solution from the results of the various subproblems.*

}

}

Recursive Algorithm Design Strategy

- Identify the base case(s) (for direct solution)
  - The one for which you know the answer
- Devise a problem splitting strategy (Decomposition)
  - A way of breaking the problem down into one or more smaller subproblems of same kind.
    - Find self-similarity
    - Subproblems must work towards a base case
- Devise a combining strategy (Composition)
  - A clear connection between small and large cases

Splitting Strategy: Onion Method!

- Model your problem as an onion
  - Find its layers!

Combining Strategy

- Create and analyze smaller cases of the problem
- Try to construct larger cases using smaller cases
- Make a guess about how small cases are generally related to larger cases
- Translate it into a general formula that uses the answers to smaller/simpler cases to find the answer to a larger/more difficult case.

Key Rules

- Always have a base case that doesn't recurse
- Make sure recursive case always makes progress, by solving a smaller problem
- Trust recursive solution!
  - Just consider one step at a time.

Question 1


```
int mystery(int n)
{
        if(n == 0)
            return 1;
        else
          return n * mystery(n + 1);
}
```

1- fibonacci 2- factorial 3- sum of numbers 4- none

Question 1


```
int mystery(int n)
{
        if(n == 0)
            return 1;
        else
          return n * mystery(n + 1);
}
```

Recursive methods must eventually terminate!

1- fibonacci   2- factorial   3- sum of numbers   4- none

Question 2


```
int mystery2(int n)
{
        if(n == 1)
            return 1;
        else
          return n + mystery2(n - 1);
}
```

1- fibonacci   2- factorial   3- sum of numbers   4- none

Question 2


```
int mystery2(int n)
{
        if(n == 1)
            return 1;
        else
          return n + mystery2(n - 1);
}
```

1- fibonacci   2- factorial $\boxed{\text{3- sum of numbers}}$ 4- none

Algorithm Design Example

Problem: Length of a String

Example: Length("abcdef") → 6

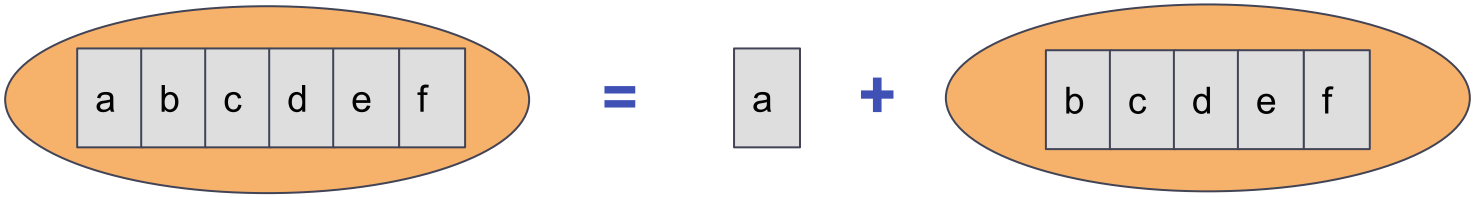
How we can solve this recursively?

Recursive Algorithm

Question? How can we describe this algorithm in terms of a smaller or simpler version of itself?

Is it solvable by Recursion?

String Length: Code



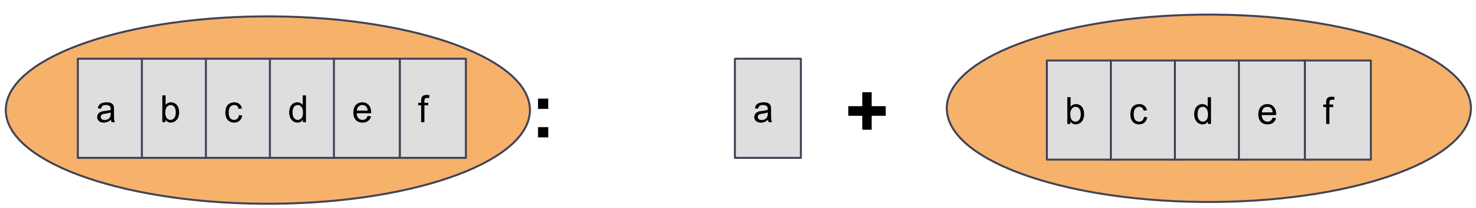
Length("abcdef"): 1 + Length ("bcdef")

- **Decomposition**
  - *First character*
  - *Remaining n-1 characters*
- **Composition**
  - *Add (1 + length of the rest)*
- **Base/stopping case**
  - *The input is null or empty*

String Length: Code


```
int Length(String str) {
    if (str == null || str == “”)
        return 0;
    else
        return 1 + Length(str.substring(1));
}
```

Length(“abcdef”): 1 + Length (“bcdef”)



Homework: Stars Problem!

How many red triangle tiles for the **1000th** layer?

23988 of them

layers

Towers of Hanoi

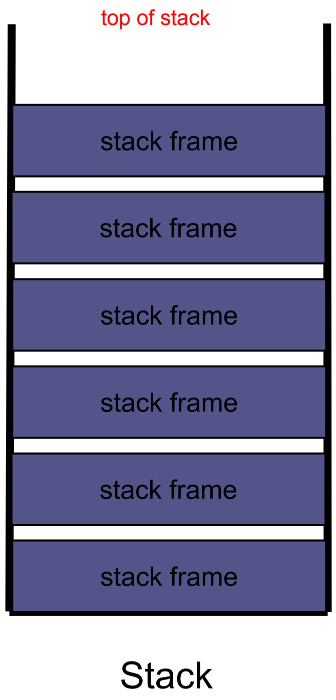
Rules:

- Move all disks from source to dest.
  - One disk at a time
  - Only onto a larger disk or an empty peg

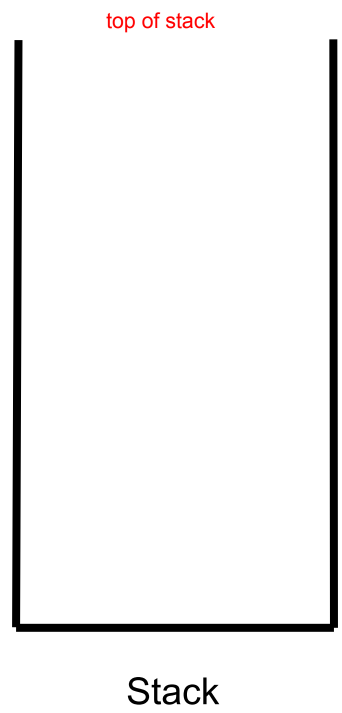
When to Use Recursion?

- Deep recursion
- *Recursive is less efficient*
  - *Overhead with function calls*
    - *Overhead of loop repetition is smaller than the overhead of a method call and return*
  - *Overlapping calculation*
    - *Repeated recursive calls with the same arguments*
      - *e.g., Fibonacci Sequence*
    - *Memoization and Dynamic Programming*
- *Recursive code is more intuitive: easier to design, write, read and debug*

Run-Time Stack

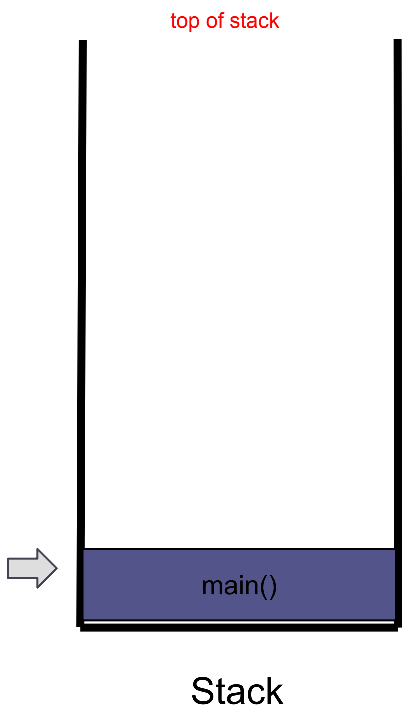


Run-Time Stack (cont.)



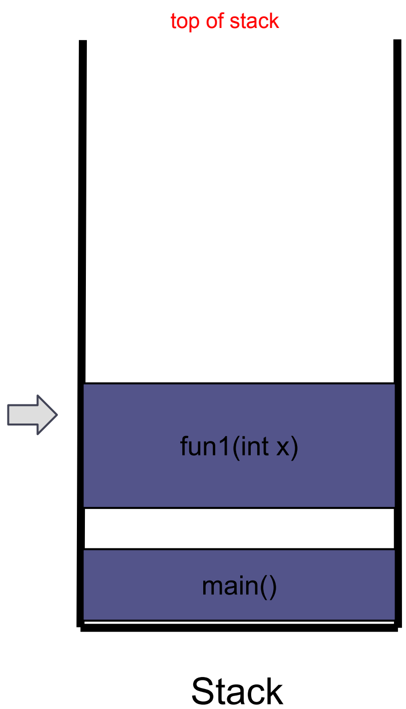
in the beginning!

Run-Time Stack (cont.)



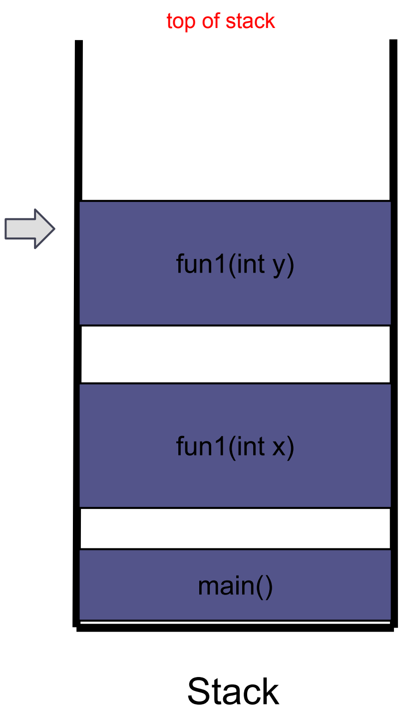
when a function is called

Run-Time Stack (cont.)



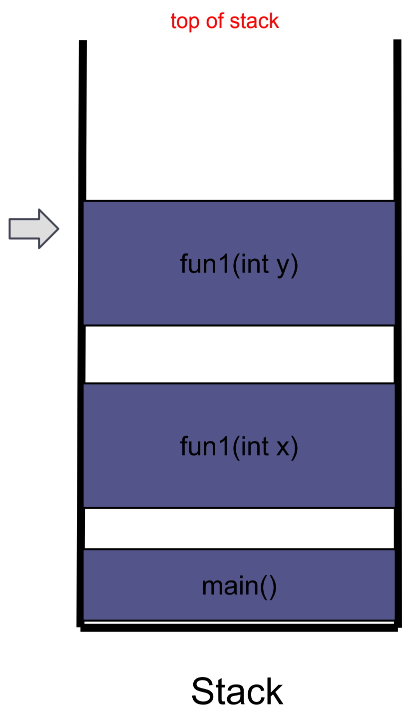
when main() calls another function

Run-Time Stack (cont.)



when func1() calls itself (or another function)

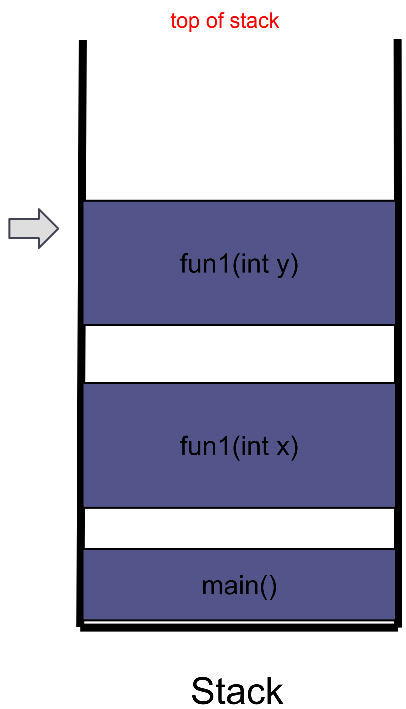
Run-Time Stack (cont.)



be careful!

the stack size is limited

Run-Time Stack (cont.)



make sure your recursive method terminates at some points!

When to Use Recursion?

- *Deep recursion*
- Recursive is less efficient ← Not always!
  - **Overhead with function calls**
    - Overhead of loop repetition is smaller than the overhead of a method call and return
  - *Overlapping calculation*
    - *Repeated recursive calls with the same arguments*
      - *e.g., Fibonacci Sequence*
    - *Memoization and Dynamic Programming*
- *Recursive code is more intuitive: easier to design, write, read and debug*

When to Use Recursion?

- *Deep recursion*
- *Recursive is less efficient*
  - ***Overhead with function calls***
    - *Overhead of loop repetition is smaller than the overhead of a method call and return*
  - **Overlapping calculation**
    - Repeated recursive calls with the same arguments
      - e.g., Fibonacci Sequence
    - *Memoization and Dynamic Programming*
- *Recursive code is more intuitive: easier to design, write, read and debug*

When to Use Recursion?

- *Deep recursion*
- Recursive is less efficient
  - ***Overhead with function calls***
    - *Overhead of loop repetition is smaller than the overhead of a method call and return*
  - **Overlapping calculation**
    - Repeated recursive calls with the same arguments
      - e.g., Fibonacci Sequence
    - Memoization and Dynamic Programming
- *Recursive code is more intuitive: easier to design, write, read and debug*

Next week!

When to Use Recursion?

- *Deep recursion*
- *Recursive is less efficient*
  - ***Overhead with function calls***
    - *Overhead of loop repetition is smaller than the overhead of a method call and return*
  - ***Overlapping calculation***
    - *Repeated recursive calls with the same arguments*
      - *e.g., Fibonacci Sequence*
    - *Memoization and Dynamic Programming*
- Recursive code is **more intuitive**: easier to design, write, read and debug

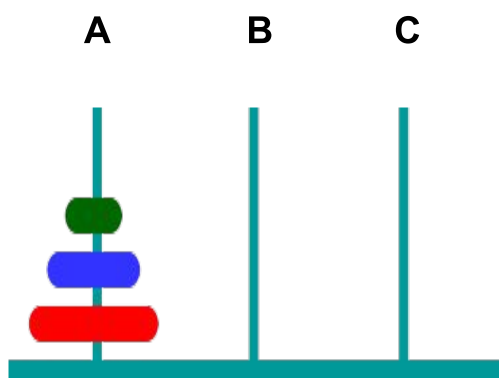
When to Use Recursion?

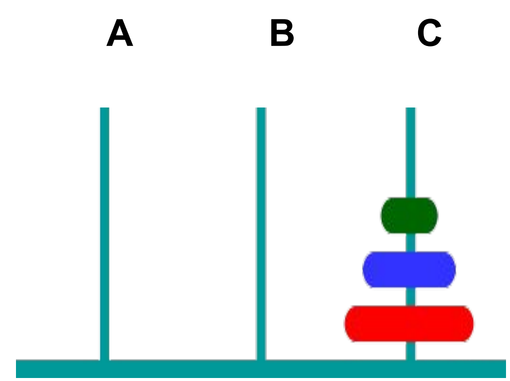
- Tradeoff between readability and efficiency
  - Solutions/algorithms for some problems are inherently recursive
    - Iterative implementation could be more complicated
  - When efficiency is less important
    - it might make the code easier to understand

When to Use Recursion? (cont.)


```java
static public void main(String args[])
{
 int N = 4; // number of discs
 int nummoves,second=0,third,pos2,pos3,j,i = 1;
 int [] locations = new int[N+2]; // remembers which corner each disc is on
 for (j=0; j<N; j++) locations[i] = 0; // initially all are on 0
 locations[N+1]=2; // 2 is destination
 nummoves = 1;
 for (i=1; i<=N; i++) nummoves*=2;
 nummoves -= 1;
 for (i=1; i<= nummoves; i++){
    if (i%2==1){
        // odd numbered move - move disc 1
        second = locations[1]; // remember where disc 1 moved from
        locations[1] = (locations[1]+ 1) %3;
        System.out.print("Move disc 1 to ");
        System.out.println((char)('A'+locations[1]));
    }
    else{
        // even numbered move make only move possible not involving disc 1
        third = 3 - second - locations[1];
        // find smallest values on the other 2 corners
        pos2 = N+1; for (j=N+1; j>=2; j--) if (locations[j]==second) pos2=j;
        pos3 = N+1; for (j=N+1; j>=2; j--) if (locations[j]==third) pos3=j;
        System.out.print("Move disc ");
        // move smaller on top of larger
        if (pos2<pos3){
            System.out.print(pos2);
            System.out.print(" to ");
            System.out.println((char)('A'+third));
            locations[pos2]=third;
        }
        else{
            System.out.print(pos3);
            System.out.print(" to ");
            System.out.println((char)('A'+second));
            locations[pos3]=second;
        }
    }
  }
}
```

```java
void solve(int n, String start, String auxiliary,
String end) {
    if (n == 1) {
        System.out.println(start + " -> " + end);
    } else {
        solve(n - 1, start, end, auxiliary);
        System.out.println(start + " -> " + end);
        solve(n - 1, auxiliary, start, end);
    }
}
```





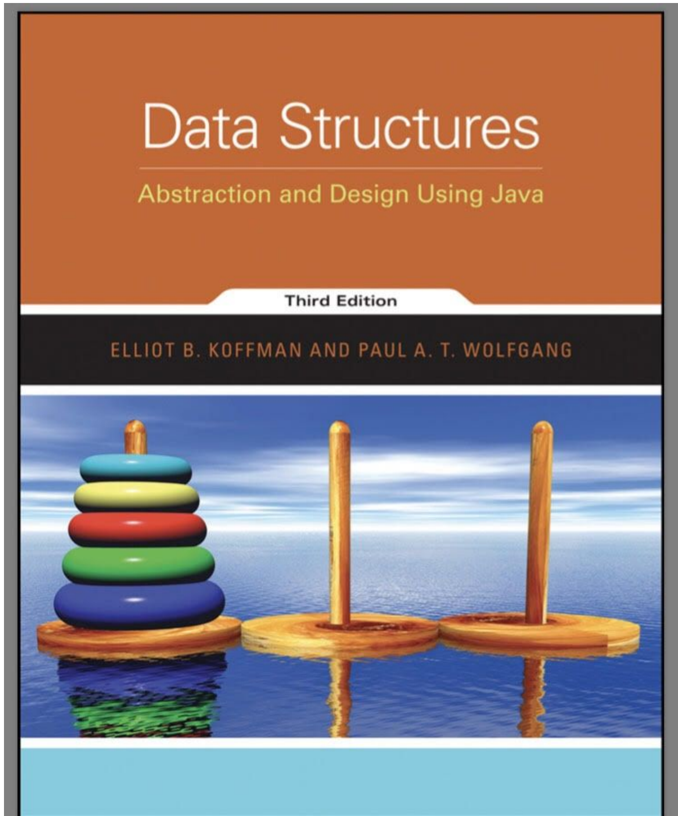
Iterative solution vs *Recursive solution*

Homework

Homework:

https://github.com/naser/ubrock

Due Friday May 31st



Any Questions?

ezzati@gmail.com

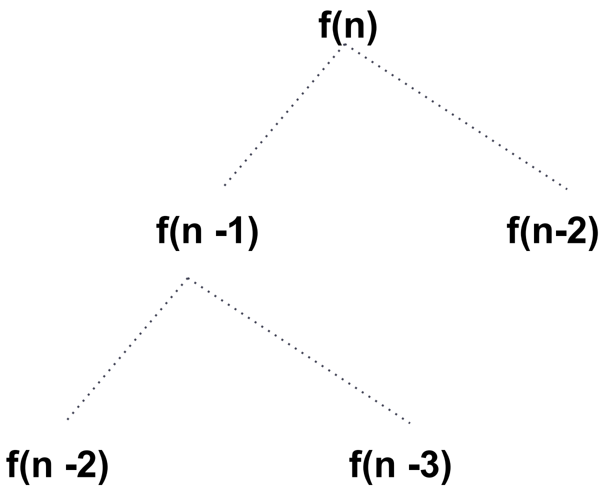
Thank You!

Any Questions

Naser Ezzati-Jivan
ezzati@gmail.com

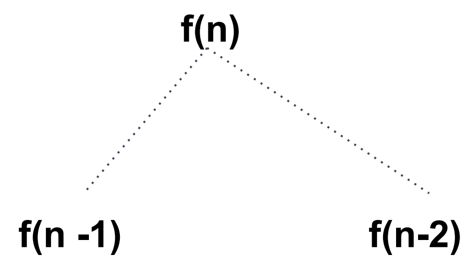
Overlapping Calculation

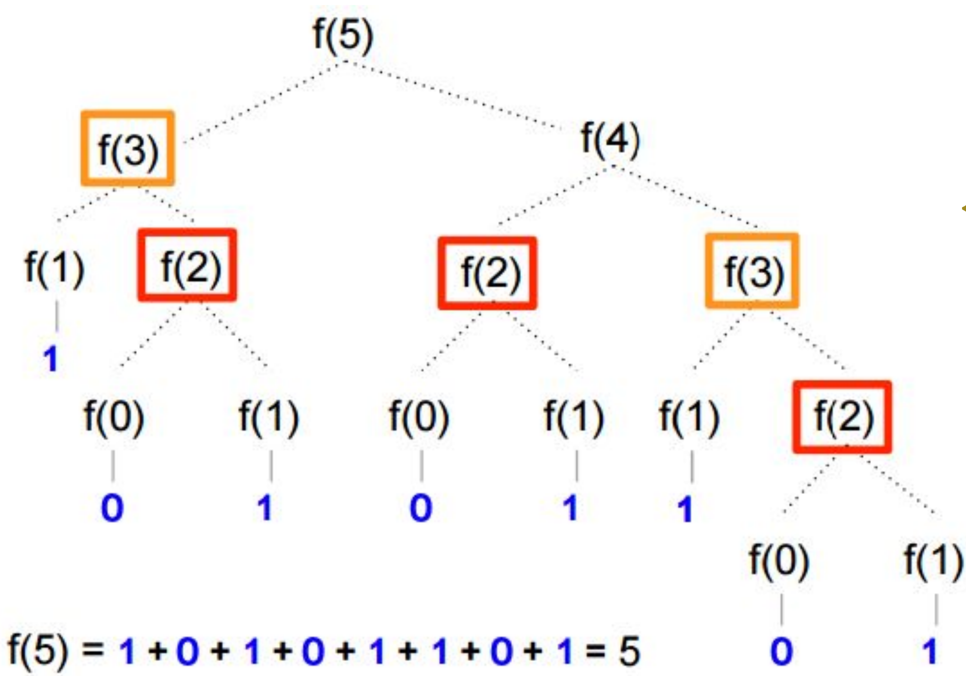

```
long f(int n) {
        if (n < 2)
            return n;
        else
            return f(n-1) + f(n-2);
    }
```



Overlapping Calculation


```
long f(int n) {
        if (n < 2)
            return n;
        else
            return f(n-1) + f(n-2);
    }
```





← f(5)

Extra Slide: Proving that a Recursive Method is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n, then it must be true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress towards the base case
  - Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Recursive vs Iterative Methods

- All recursive algorithms/methods can be rewritten without recursion

- Iterative methods use loops instead of recursion

- Iterative methods are generally faster! and use less memory (less overhead in keeping track of method calls)

Deciding whether to use a recursive solution

- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the nonrecursive version
- The recursive version is shorter and simpler than the nonrecursive solution

Common Patterns

- Divide in half, solve one half
- Divide in sub-problems, solve each sub-problem recursively, “merge”
- Solve one or several problems of size n-1
- Process first element, recurse on remaining problem
-

Software Debugging Course

- How debuggers work?
  - Debugging Tools and Techniques
  - Tracing Tools and Techniques
  - Profiling Tools and Techniques
  - Architecture and Design
- Debugging vs Testing
- Applications
  - Performance Analysis
  - Host-based Anomalies
  - Network Attacks
- Automated Debugging
  - Machine Learning- based Techniques
  - Data Mining-based Techniques
- Root-Cause Analysis