



University of
Pittsburgh

Algorithms and Data Structures 1

CS 0445



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
 - Lab 5: next Monday @ 11:59 pm
 - No homework due this Friday
- Midterm Exam: Thursday 10/20
 - closed book, paper, in-person
- No recitations tomorrow: Fall Break!

Previous Lecture ...

- Recursion
 - More examples
 - Runtime analysis using proof by induction
- Using recursion to solve hard problems
 - Towers of Hanoi

Today ...

- Recursion Applications
 - Divide and Conquer
 - Backtracking
- Limitation of Recursion

Recursive Solution to Towers of Hanoi Problem

- Recursive algorithm to solve any number of disks.

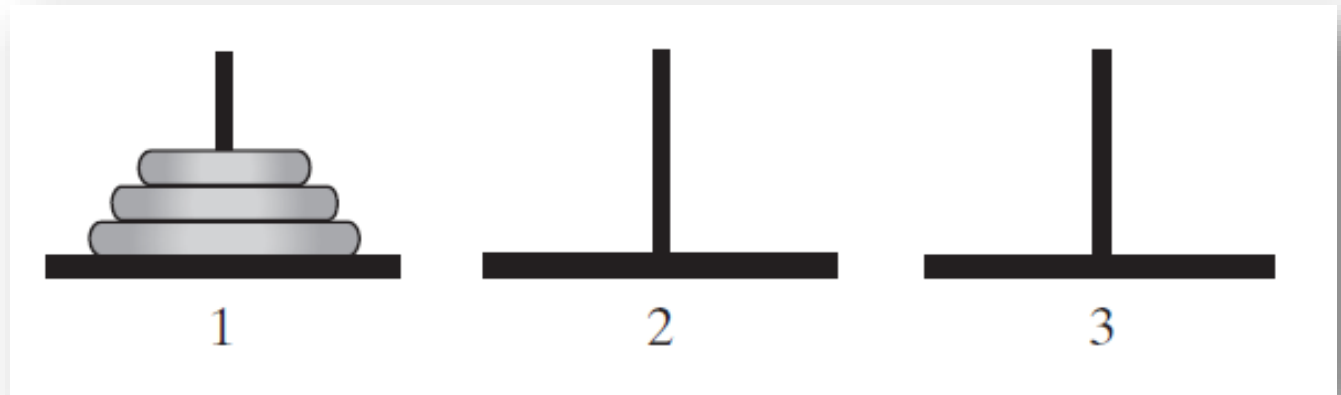
```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
 - Don't forget the base case (e.g., $T(1)$)
- Step 2: Build an intuition to make a guess for a closed form for $T(n)$
 - By evaluating $T(n)$ for small values of n
- Step 3: Try to prove your guess for $T(n)$ using induction
 - If not successful, go back to Step 2

Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
- What is n in this problem?
 - number of disks to move
- $T(1) = 1$



Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)

```
if (numberOfDisks == 1)
```

```
    Move disk from startPole to endPole
```

```
else
```

```
{
```

```
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
```

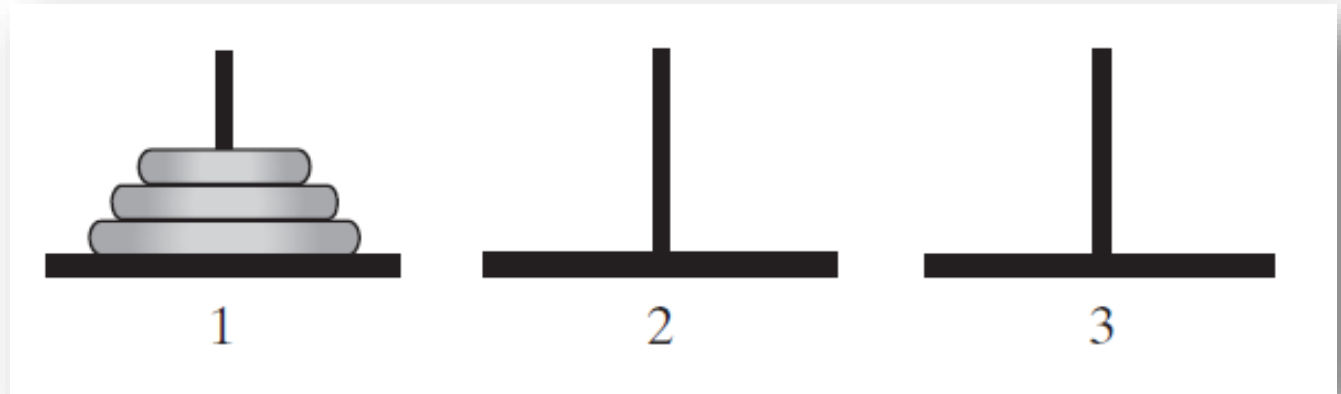
```
    Move disk from startPole to endPole
```

```
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
```

```
}
```

Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
- What is n in this problem?
 - number of disks to move
- $T(1) = 1$
- $T(n) = ??$
- $T(n) = 2T(n-1) + 1$
 - why?



```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```


Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
 - $T(1) = 1$
 - $T(n) = 2T(n-1) + 1$ for $n > 1$
- Step 2: Build an intuition to make a guess for a closed form for $T(n)$
 - By evaluating $T(n)$ for small values of n
 - $T(2) = 2T(1) + 1 = 3$
 - $T(3) = 2T(2) + 1 = 7$
 - $T(4) = 2T(3) + 1 = 15$
 - $T(5) = 2T(4) + 1 = 31$
 - ...
 - not linear
 - increase by 1 in $n \rightarrow$ almost doubling of $T(n)$
 - exponential function
- $T(n) = 2^n$
 - would that work?
 - Nope, why?
- $T(n) = 2^n - 1$

Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
 - $T(1) = 1$ (1)
 - $T(n) = 2T(n-1) + 1$ for $n > 1$ (2)
- Step 2: Build an intuition to make a guess for a closed form for $T(n)$
 - $T(n) = 2^n - 1$
- Step 3: Try to prove your guess for $T(n)$ using induction
- Let's try to prove that $T(n) = 2^n - 1$ for all $n \geq 1$ (3)
- Base Case: when $n=1$
 - From (1) $\rightarrow T(1) = 1$
 - From (3) $\rightarrow T(1) = 2^1 - 1 = 2 - 1 = 1$
 - (3) holds for $n=1$

Runtime analysis of Recursive Algorithms

- Step 1: **Recurrence Relation** for the running time function $T(n)$
 - $T(1) = 1$ (1)
 - $T(n) = 2T(n-1) + 1$ for $n > 1$ (2)
- Step 2: Build an intuition to make a guess for a closed form for $T(n)$
 - $T(n) = 2^n - 1$
- Step 3: Try to prove your guess for $T(n)$ using induction
- Let's try to prove that $T(n) = 2^n - 1$ for all $n \geq 1$ (3)
- Inductive Step:
 - Assume (3) holds for all values $n < k$ (**Inductive Hypothesis**)
 - $T(n) = 2^n - 1$ for all $n < k$ (4)
 - Given that assumption, prove that (3) holds for $n=k$
 - i.e., prove that $T(k) = 2^k - 1$
 - From (2) $\rightarrow T(k) = 2 T(k-1) + 1$
 - $T(k) = 2 (2^{k-1} - 1) + 1$ (Using the inductive hypothesis (4))
 - $T(k) = 2^k - 2 + 1 = 2^k - 1$
 - End of proof!

Recursion Applications

- So, what is recursion good for?
 - For some problems, a **recursive approach is more natural and simpler to understand** than an iterative approach
 - e.g., traversing a linked list backwards
 - e.g., reversing a linked list

Reversing a Linked List Recursively

- You have seen in recitations how to reverse a linked list iteratively
 - pretty much involved
- Let's try a recursive solution!
- Imagine you have a friend who can reverse a linked list
 - what should friend return to us?
 - first node? last node? nothing?
 - last node! we can attach the first node after that last node
 - We must return what we expect friend to return
 - That friend is us after all!
- You can use your friend's help
- But ...
- You can't ask your friend to reverse the same list given to you
 - has to ask for help on a "smaller" list

Reversing a Linked List Recursively

- Plan of work:
 - Keep track of the first node
 - Ask friend to reverse the rest of the list ...
 - i.e., starting from second node
 - ... and return the last node in the reversed list
 - Attach the first node after the node returned by friend
 - return the first node, which is now the last node
 - Non-recursive cases
 - reversing an empty list → return null
 - reversing a single node → make it the firstNode
 - Let's code!

Recursion Applications

- So, what is recursion good for?
 - For some problems, a recursive approach is more natural and simpler to understand than an iterative approach
 - For some problems, it is very difficult to even conceive an iterative approach, especially if multiple recursive calls are required in the recursive solution
 - e.g., Towers of Hanoi
 - e.g., Eight Queens

8 Queens Problem

- How can I place 8 queens on a chessboard such that no queen can take any other in the next move?
 - Recall that queens can move horizontally, vertically or diagonally for multiple spaces

Backtracking to solve 8 Queens

- Idea of **backtracking**:
 - Proceed **forward** to a solution until it becomes apparent that no solution can be achieved along the current path
 - At that point UNDO the solution (backtrack) to a point where we can again proceed forward

8 Queens Problem

- How can we solve this with recursion and backtracking?
 - We note that all queens must be in different rows and different columns, so each row and each column must have exactly one queen when we are finished
 - Complicating it a bit is the fact that queens can move diagonally
 - So, thinking recursively, we see the following
 - To place 8 queens on the board we need to
 - Place a queen in a legal (row, column)
 - Recursively place 7 queens on the rest of the board
 - Where does backtracking come in?
 - Our initial choices may not lead to a solution – we need a way to undo a choice and try another one

8 Queens Problem

- Using this approach, we come up with the solution as shown in BackTracking.java in the code handouts
- Idea of solution:
 - Each recursive call attempts to place a queen in a specific column
 - A loop is used, since there are 8 squares in the column
 - For a given call, the state of the board from previous placements is known (i.e., where are the other queens placed so far?)
 - This is used to determine if a square is legal or not
 - If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column
 - When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)
 - If a queen cannot be placed into column i , do not even try to place one onto column $i+1$ – rather, backtrack to column $i-1$ and move the queen that had been placed there

8 Queens Problem

- Why is this difficult to do iteratively?
 - We need to store a lot of state information as we try (and un-try) many locations on the board
 - For each column so far, where has a queen been placed?
- The run-time stack does this automatically for us via activation records
 - Without recursion, we would need to store / update this information ourselves
 - This can be done (using our own Stack rather than the run-time stack), but since the mechanism is already built into recursive programming, why not utilize it?
- There are many other famous backtracking problems
 - <http://en.wikipedia.org/wiki/Backtracking>
 - PIN Cracking
 - Sudoku

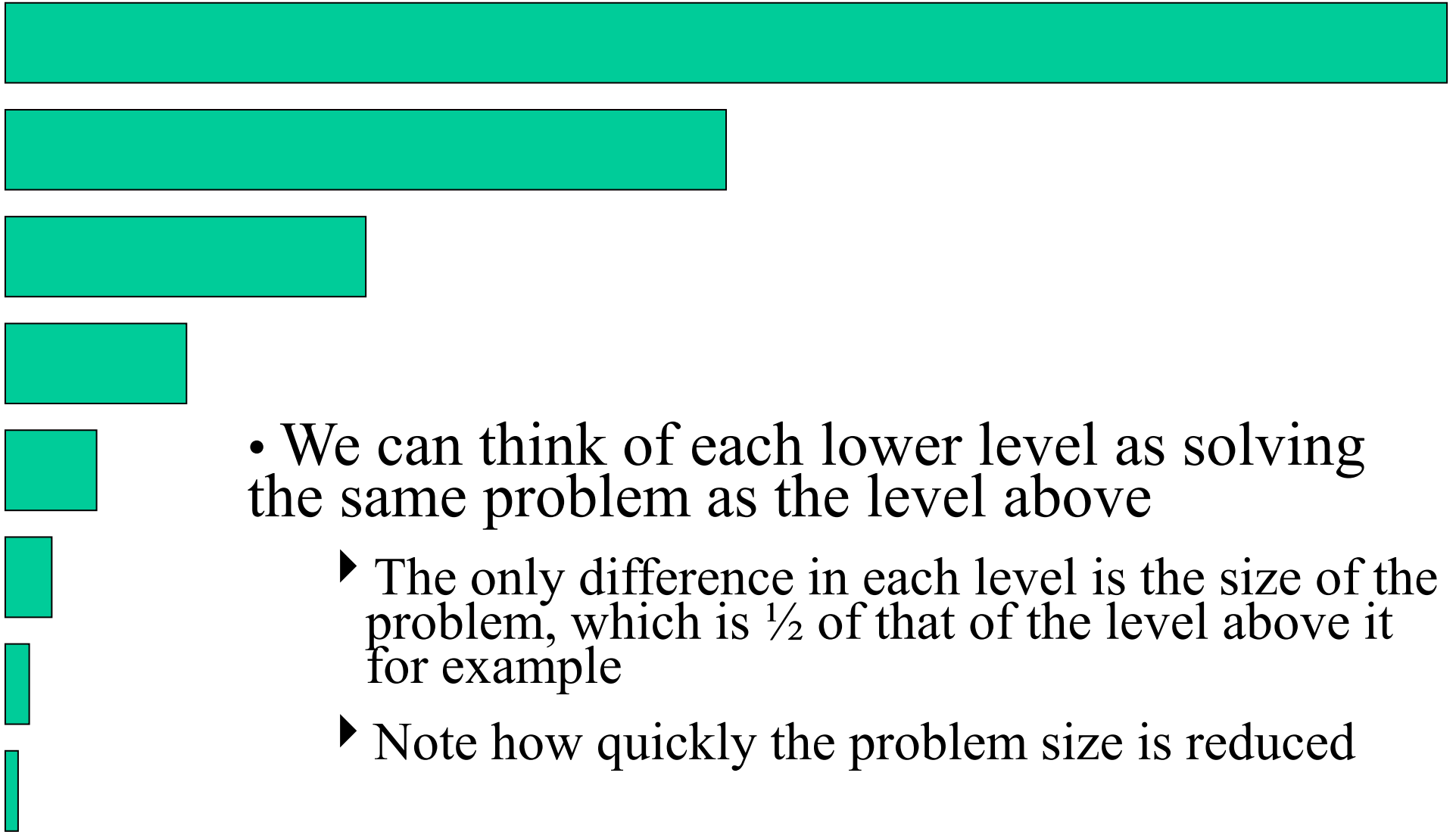
Recursion Applications

- So, what is recursion good for?
 - For some problems, a **recursive approach is more natural and simpler to understand** than an iterative approach
 - For some problems, **it is very difficult to even conceive an iterative approach**, especially if **multiple recursive calls** are required in the recursive solution
 - For some problems, a **recursive approach allows for a more efficient solution** than an iterative approach
 - e.g., exponentiation x^n
 - general approach is: Divide and Conquer
 - more examples later in the course

Recursion and Divide and Conquer

- **Divide and Conquer**
 - The idea is that a problem can be solved by breaking it down to one or more "smaller" problems in a systematic way
 - Usually the subproblem(s) are a fraction of the size of the original problem
 - Usually the subproblems(s) are identical in nature to the original problem
 - It is fairly clear why these algorithms can typically be solved quite nicely using recursion

Recursion and Divide and Conquer



Limitations of Recursion

- **Recursion Overhead**
- Recursion may lead a poor solution that an iterative approach

Overhead of Recursion

- Why do we care?
- Recursive algorithms have **overhead** associated with them
 - **Space:** each activation record (AR) takes up memory in the run-time stack (RTS)
 - If too many calls "stack up" memory can be a problem
 - **Time:** generating ARs and manipulating the RTS takes time
 - A recursive algorithm will always run more slowly than an equivalent iterative version
- Once a recursive algorithm is developed, in some cases we can convert it into a faster iterative version

Tail Recursion

- **Tail recursion**
 - Recursive algorithm in which the recursive call is the LAST statement of the method
 - No further processing after the return of the recursive call
- What are the implications of tail recursion?
 - Any tail recursive algorithm can be converted into an iterative algorithm in a methodical way
 - some compilers do this automatically
 - Some algorithms we have seen so far are tail recursive
 - make sure that the recursive call is inside an if statement
 - change the if statement to a while loop
 - replace the recursive call by a set of assignment statements in the form
 - parameter = argument

Example

- The recursive call is inside the if statement

```
public static void countDown(int integer)
{
    if (integer >= 1)
    {
        System.out.println(integer);
        countDown(integer - 1);
    } // end if
} // end countDown
```

Example

- Convert if to while

```
public static void countDown(int integer)
{ while
  if (integer >= 1)
  {
    System.out.println(integer);
    countDown(integer - 1);
  } // end if
} // end countDown
```

Example

- replace recursive call by assignment statement
 - parameter = argument
 - integer = integer - 1

```
public static void countdown(int integer)
{ while
  if (integer >= 1)
  {
    System.out.println(integer);
    countdown(integer - 1);
  } // end if
} // end countdown
```

integer = integer - 1

parameter

argument