# Extended Introduction to Computer Science CS1001.py 

## Chapter E <br> Lecture 10

Recursion(cont.): Quick-Sort<br>Merge-Sort<br>Towers of Hanoi

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-pr.wikidot.com

* Slides based on a course designed by Prof. Benny Chor


## מעבנה (נוֹא (באדום - חומר שירד בשל קיצור הסמסטר)



## Comic Relief *



* אנו מזמינים אתכם לשלוח לנו הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר


## Recursion: Plan

- Definition and basic examples
- Fibonacci
- factorial
- Recursive binary search
- Sorting

$\square$
- Quick-Sort


## You are here

- Merge-Sort
- Towers of Hanoi (and the "monster of Hanoi")
- Improving recursion with memoization
- An example from-Game theory-Chomp! (removed this semester)


## Recursion: Definition and First Example

- A function $f$, whose definition contains a call to $f$ itself, is called recursive.
- A simple example is the following function that computes the factorial of a natural input number, $n!=n \cdot(n-1) \cdot \ldots \cdot 2 \cdot 1$ (and it is convenient to define $0!=1$ )
- Observe that $n!=n \cdot(n-1)$ !, for $n \geq 1$.
- It can be coded in Python, using recursion, as follows:

```
def factorial(n):
    if n==0:
    return 1
    else:
    return n * factorial(n-1)
```


## What Happens at Run Time?

- For example, here is a record of the execution for factorial(2):



## Fibonacci Numbers

- A second simple example are the Fibonacci numbers:

$$
\begin{aligned}
& F_{0}=1, F_{1}=1, \\
& \text { and for } n>1, \quad F_{n}=F_{n-1}+F_{n-2}
\end{aligned}
$$

- A function that computes the $n$ 'th Fibonacci number can be programmed in Python, using recursion:

```
def fibonacci(n):
    if n<=1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

- Sanity check:

```
>>> [fibonacci(n) for n in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```


## Recursion Trees - Order of Execution

- Note the order of execution as animated in the following gif:

https://commons.wikimedia.org/wiki/File:Fibonacci call tree 5.gif


## Back to Sorting

- We saw one simple sorting algorithm - Selection sort, whose time complexity for both best and worst cases is $O\left(n^{2}\right)$.
- We will now see another approach to sorting (out of very many), called Quicksort, which employs both randomization and recursion.


## Quicksort - Description

- Our input is an unsorted list, say

$$
[28,12,32,27,10,12,44,20,26,6,20,21]
$$

- We choose a pivot element, simply one of the elements in the list. For example, suppose we chose 20 (the second occurrence).
- We now compare all elements in the list to the pivot. We create three new lists, termed smaller, equal, greater. Each element from the original list is placed in exactly one of these three lists, depending on its size with respect to the pivot.
- smaller $=[12,10,12,6]$
- equal $=[20,20]$
- greater $=[28,32,27,44,26,21]$
- Note that the equal list contains at least one element, and that both smaller and greater are strictly shorter than the original list (why is
10 this important?)


## Quicksort - Description (cont.)

- What do we do next?
- We recursively sort the sub-lists smaller and greater
- And then we append the three lists, in order (+ means list concatenation). Note that equal requires no sorting.

```
return quicksort(smaller) + equal + quicksort(greater)
```

- quicksort(smaller) $=[6,10,12,12]$
equal $=[20,20]$
quicksort(greater) $=[21,26,27,28,32,44]$

$$
\begin{aligned}
& {[6,10,12,12]+[20,20]+[21,26,27,28,32,44]} \\
& \quad=[6,10,12,12,20,20,21,26,27,28,32,44]
\end{aligned}
$$

- The original list was [28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21]


## Quicksort: A Graphical Depiction



## Quicksort: Python Code

```
import random # a package for (pseudo) random generation
def quicksort(lst):
    if len(lst) <= 1: # empty lists or length 1 lists
        return lst
    else:
        pivot = random.choice(lst) # random element from lst
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
                        # ain't these selections neat?
        return quicksort(smaller) + equal + quicksort(greater)
                        # two recursive calls
```


## Implementation Comment:

## The Use of List Comprehension

- The use of list comprehension is a very convenient mechanism for writing code, and Python is extremely good at it.
- This convenience is good for quickly developing code. It also helps to develop correct code. But this simplicity and elegance do not necessarily imply an efficient execution.
- For example, our quicksort algorithm goes three times over the original list. Furthermore, it allocates new memory for the three sublists.
- There are versions of quicksort that go over the list only once, and swap original list elements in-place, reusing the same memory. These versions ate more efficient, yet more error prone and generally take longer to develop.
- Eventually you will choose, on a case by case basis, which style of programming to use.


## Quicksort: Complexity

To analyze the time complexity of quicksort, we will use recursion trees.

## Worst-Case Analysis

- Worst partition: at each recursive call the pivot is selected to be the minimal or maximal value in the input list, dividing the problem into sizes $n-1$ and 0 .


Pivot is maximum


Pivot is minimum

- In terms of time complexity, the cases are equivalent.


## Worst-Case Analysis (cont.)

- Work at each step: besides the recursive calls, each step requires $\leq c \cdot n$ time for $n>0$ and some constant $c$. For a problem size of $n=0$ the amount of work $\leq c$.


```
def quicksort(lst):
if len(lst) <= 1: # empty lists or length 1 lists
            return lst
else:
    pivot = random.choice(lst) # random element from lst }\quadO(\operatorname{log}n
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot] 
        return quicksort(smaller) + equal + quicksort(greater)
                            # two recursive calls
```


## Worst-Case Analysis (final)

- The recursion tree, annotated with the amount of time spent at each step:

- Total amount of time:

$$
T(n) \leq c\left(\sum_{i=1}^{n} i\right)+c(n-1)=O\left(n^{2}\right)
$$

## Complexity using Recurrence Relations

- The worst case run time satisfies the recurrence relation

$$
T(n) \leq c \cdot n+T(n-1)
$$

where $c$ is some constant.

- The solution for this equation is, as we saw,

$$
T(n)=O\left(n^{2}\right)
$$

## Best-Case Analysis - Example

- At each recursive call the pivot is selected to be the median
- There are 2 sub-problems to solve, of sizes at most $n / 2$.
- Here are two such recursion trees for example:

$$
n=7
$$



## Best-Case Analysis

- The recursion tree (with minor simplifications), annotated with the amount of time spent at each step:

- Total amount of time:

$$
T(n)=\text { time per layer } \times \text { number of layers } \leq c n \cdot O(\log n)=O(n \log n)
$$

## Complexity using Recurrence Relations

- The best case run time satisfies the recurrence relation

$$
T(n) \leq c \cdot n+2 \cdot T(n / 2)
$$

where $c$ is some constant.

- The solution for this equation is, as we saw,

$$
T(n)=O(n \cdot \log n)
$$

## Quicksort: Average Complexity

- A more complicated analysis can be done for the average running time.
- Average over what?
- It can be shown that the average running time to sort a list of $n$ elements is also $O(n \cdot \log n)$
- The "best-case" constant in the "big $O$ " notation is slightly smaller than the "average-case" constant.
- Rigorous analysis is deferred to the data structures course.


## Deterministic Quicksort

- We could take the element with the first, last, middle or any other index in the list as the pivot.
- For example:

$$
\text { pivot }=\text { lst }[0]
$$

- This would usually work well (assuming some random distribution of input lists).
- However, in some cases this choice would lead to poor performance (even though the algorithm will always converge).
- For example, if the input list is already sorted (or close to sorted), and the pivot is the first or last element.


## Quicksort: Pivot Selection, cont.

- Instead of a fixed choice, the recommended choice is to pick the pivot at random.
- With high probability, the randomly chosen pivot will be neither too close to the minimum nor too close to the maximum.
- This implies that both the lists smaller and greater are substantially shorter than the original list, and yields good performance with high probability (at this point this is an intuitive claim, nothing rigorous.)


## Comic Relief*



Merge-Sort

## Merging Sorted Lists (reminder)

```
def merge (A, B) :
    ''' Merge list A of size \(n\) and list B of size m
            A and B must be sorted!
    \(\mathrm{n}=\operatorname{len}(\mathrm{A})\)
    \(\mathrm{m}=\) len(B)
    \(C=[\) None for \(i\) in range \((n+m)]\)
    \(a=0 ; \quad b=0 ; c=0\)
    while \(\mathrm{a}<\mathrm{n}\) and \(\mathrm{b}<\mathrm{m}\) : \#more element in both A and B
        if \(A[a]<B[b]:\)
        \(C[c]=A[a]\)
        a \(+=1\)
        else:
        \(C[c]=B[b]\)
        b \(+=1\)
        c \(+=1\)
    C[c:] = A[a:] + B[b:] \#append remaining elements
    return C
```


## Merge-Sort: Description

- Merge-sort is a recursive sorting algorithm (i.e. like quicksort it also follows a "divide and conquer" approach).
- Merge-Sort is deterministic
- An input list (unsorted) of size $n$ is split to two halves:

$$
0 \ldots\lfloor n / 2\rfloor \text { and }\lfloor n / 2\rfloor+1 \ldots(n-1)
$$



- If we sorted these 2 halves, we would then only need to merge them.

- Does anybody know a nice sorting algorithm for the 2 halves?


## Merge-Sort: Python Code

```
def mergesort(lst):
    """ recursive mergesort """
    n = len (lst)
    if }\textrm{n}<=1
        return
        else:
            return merge(
            工)
                            # two recursive calls, then merge
```


## Merge-Sort: Python Code

def mergesort(lst):
""" recursive mergesort """
$\mathrm{n}=$ len (lst)
if $n<=1$ :
return lst
else:
return merge(mergesort(lst[0:n//2]), mergesort(lst[n//2:n]))
\# two recursive calls, then merge

## Merge-Sort: example

| 38 | 43 | 27 | 3 | 9 | 82 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- |

$\longrightarrow$ recursive call

-     - recursion fold
$+\quad$ merge


## Merge-Sort: example

## $\longrightarrow$ recursive call


$-\boldsymbol{-} \rightarrow$ recursion fold
merge

## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: example



## Merge-Sort: Recursion Tree



## Merge Sort: Complexity Analysis

- Given a list with $n$ elements, mergesort makes 2 recursive calls. One to a list with $n / 2$ 〕 elements, the other to a list with $\lceil n / 2\rceil$ elements.
- The two returned lists are subsequently merged.
- Question: Is there a difference between worst-case and best-case for mergesort?
- Recursion tree and time complexity analysis - discussion in class.


## Merge Sort: Complexity Analysis

- The runtime of mergesort on lists with $n$ elements, for both best and worst case, satisfies the recurrence relation

$$
T(n) \leq c n+2 T\left(\frac{n}{2}\right)
$$

where $c$ is a constant.

- We already saw that the solution to this relation is

$$
T(n)=O(n \cdot \log n)
$$

- Note that the mergesort function uses slicing which adds an overhead of $O(n)$ to the time complexity of each level in the recursion tree.
- Asymptotically, however, this overhead is negligible.


## A Three-Way Race

```
import time
import random
from quicksort import * # need quicksort.py
from mergesort import * # need mergesort.py
print("3 way race")
for func in [quicksort, mergesort, sorted]:
    print(func.___name___)
    for n in [2000, 4000, 8000]:
                print("n=", n, end=" ")
                rlst = [random.randint(0,n) for i in range(n)]
                t0 = time.perf_counter()
        for i in range(100):
        func(rlst) # not inplace, lst unchanged !
        t1 = time.perf_counter()
        print(t1-t0)
```


## A Three-Way Race (time in seconds)

```
quicksort
n= 2000 0.4635085
n= 4000 0.9853195
n= 8000 2.1039044
mergesort
n= 2000 0.9249551
n= 4000 1.7947224000000004
n= 8000 3.9287744999999994
Sorted (Python's built-in sorted)
n= 2000 0.0193020999999991
n=4000 0.0430652000000000914
n= 8000 0.09296229999999994 (I think we have a winner!)
```

- The results speak for themselves. Conclusions?


## A Note on Space (memory) Complexity

- A measure of how much memory the algorithm needs to allocate - not including memory allocated for the input of the algorithm
- Assuming memory can be reused if needed, this is the maximal amount of memory needed at any time point during the algorithm's execution

- Compare to time complexity, which relates to the cumulative amount of operations made along the algorithm's execution


## Space Complexity and Recursion

- Recursion depth has an implication on the space (memory) complexity, as each recursive call requires opening a new environment in memory.
- At each recursive call we consider the space allocation requirements, just as we did earlier in the course. For example:
- copying (parts of) the input
- list / string slicing
- using + operator for lists (as opposed to += or Ist.append)
- To analyze the space complexity of a recursive function, we consider how much memory is required in the deepest leaf


## Comic Relief *



Cover of Ummagumma, a double album by Pink Floyd, released in 1969.
Taken from Wikipedia

## Towers of Hanoi



And the Monster of Hanoi


## Towers of Hanoi

Towers of Hanoi is a well-known mathematical puzzle, and no class on recursion, including this one (a recursive claim in itself :-) ), is complete without discussing it.

(figure from Wikipedia)

## Towers of Hanoi - Origin

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle will be completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.

## Towers of Hanoi - Description

- There are three rods, named $A, B, C$, and $n$ disks of different sizes which can be placed onto any rod.
- The puzzle starts with all $n$ disks in a stack in ascending order of size on one rod, say $A$, so that the smallest is at the top (see figure).

(figure from Wikipedia)


## Towers of Hanoi: Rules of Game

- The objective of the puzzle is to move the entire stack of all n disks to another rod, say C , obeying the following rules:
- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

(figure and some text from Wikipedia)


## Towers of Hanoi: Recursive View

- In order to think about a recursive solution, we should first have a recursive definition of a Hanoi tower:
- it is either empty, This is the base case
- or it is a tower on top of a larger disk (larger than all the disks in the tower on top).

Schematically:

Another possibility is to let the base case be a tower of one disk


## Towers of Hanoi: The algorithm

- We can now describe a recursive algorithm to move a stack of $n$ disks from rod $A$ to rod $C$ using rod $B$ as a helping rod.
- In the base case, when $n=0$, there is nothing to do.
- If we chose $n=1$ as the base case, then the base case would be to move the single rod from $A$ to $C$.
- The non- base case ( $n>0$ ) will be shown in the next slide.


## Towers of Hanoi: Recursive Algorithm



A


C

Move (recursively) top tower from $A$ to $B$ using $C$ as helping rod Move largest disk from $A$ to $C$

Move (recursively) top tower from $B$ to $C$ using $A$ as helping rod

## Towers of Hanoi: Another Picture



Move (recursively) top tower from
A to $B$ using $C$ as helping rod

Move largest disk from $A$ to $C$

Move (recursively) top tower from $B$ to $C$ using $A$ as helping rod


## Towers of Hanoi: Pseudo-Code

Input: number of disks, n
Output: sequence of moves needed to transfer $n$ disks from $A \rightarrow C$, using $B$ as a "helping rod"

Algorithm:

- If $\mathrm{n}=0$, there is nothing to do.
- Otherwise (namely $\mathrm{n}>0$ ):
(1) Move * $n$ - 1 disks from $A \rightarrow B$, using $C$ as a "helping rod"
(2) Move the largest disk (numbered $n$ ) directly from $A \rightarrow C$
(3) Move* $n$ - 1 disks from $B \rightarrow C$, using $A$ as a "helping rod"

[^0]
## Correctness of the Recursive Solution

- At no stage in the algorithm execution no rules are violated:
- During the entire stage (1), disk $n$ stays put on rod A. As it was the biggest of all $n$ disks, no rule will be violated if some of the $n-1$ disks are placed on top of it during the recursion in (1).
- In step (2), all n-1 smaller disks are on rod B, so moving disc $n$ directly from rod $A$ to rod $C$ is legal.
- The argument for step (3) is identical to the argument for step (1).


## Towers of Hanoi: Python Code

- We write a function of four arguments: $n$, start, via, target.
- The first argument, $n$, is the number of discs. The next three arguments are the three rods' "names", given the default values "A", "B", and "C".
- The function prints the moves and does not return anything.

```
def HanoiTowers(n, start="A", via="B", target="C"):
    """ Prints a list of steps to move a stack
        of n disks from rod "start" to rod "target"
        employing intermediate rod "via"
    """
    if n>0:
    HanoiTowers(n-1, start, target, via)
    print("Move disk", n, "from", start, "to", target)
    HanoiTowers(n-1, via, start, target)
```

- Question: what is the base case here (it appears indirectly in the code).


## Towers of Hanoi: Running the Code

```
>>> HanoiTowers(1)
Move disk 1 from A to C
>>> HanoiTowers(2)
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
>>> HanoiTowers(3)
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```


## Towers of Hanoi: Running the Code

```
>>> HanoiTowers(1, "A", "B", "C")
disk 1 from A to C
```



## Towers of Hanoi: Running the Code

```
disk 1 from A to B
disk 2 from A to C
```

>>> HanoiTowers(2, "A", "B", "C")
disk 1 from $B$ to $C \quad A \rightarrow C$ (via $B)$


## Towers of Hanoi: Running the Code

>>> HanoiTowers (3,
disk 1 from A to C
disk 2 from A to B
disk 1 from C to B
disk 3 from A to C
disk 1 from B to A
disk 2 from B to C
disk 1 from A to C

## Towers of Hanoi: Running the Code

HanoiTowers(n, "A", "B", "C")


## Towers of Hanoi: Number of Moves

- The time complexity of our solution equals to the number of moves (as each iteration takes $\mathrm{O}(1)$ time)
- Let us denote by $H(n)$ the number of moves required to solve an $n$ disk instance of the puzzle.
- In the recursive solution outlined above, to solve an $n$ disks instance we solve two instances of $n-1$ disks, plus one direct move. This gives us the recursive relation

$$
\begin{aligned}
& H(0)=0 \\
& \text { For } n>0, H(n)=2 \cdot H(n-1)+1
\end{aligned}
$$

whose solution is $H(n)=2^{n}-1$ (You should be able to verify the last equality, using recursion trees or induction.)

- The recursion depth here is "just" $O(n)$. But the size of the recursion tree is $O\left(2^{n}\right)$, which is exponential in $n$. This is also the time complexity of the algorithm.


## Optimality of Number of Moves

- Hey, wait a minute. $H(n)=2^{n}-1$ is the number of moves in the solution presented above. Can't we find a more efficient solution?
- This is very good thinking in general.
- But in this case, we can argue that $H(n)=2^{n}-1$ moves are required from any solution strategy. (of course, more inefficient strategies do exist).
- Can you explain why?


## Comic Relief*



## "Monster of Hanoi" *

- Suppose a monster demanded to know what the $\left(3^{97}+381\right)^{\prime}$ th move in an $n=200$ disk Towers of Hanoi puzzle is, or else . . . .



## "Monster of Hanoi"

- Suppose a monster demanded to know what the $\left(3^{97}+381\right)$ 'th move in an $n=200$ disk Towers of Hanoi puzzle is, or else . . . .
- Having seen and even understood the material, you realize that either expanding all $H(200)=2^{200}-1$ moves, or even just the first $3^{97}+381$, is out of computational reach in any conceivable future, and the monster should try its luck elsewhere.
- You eventually decide to solve this new problem. The first step towards taming the monster is to give the new problem a name:

Hanoi_move(n, k, start="A", via="B", target="C")

## "Monster of Hanoi" - small example

>>> HanoiTowers(3)
1Move disk 1 from A to C 2Move disk 2 from A to B 3Move disk 1 from C to B 4Move disk 3 from A to C 5Move disk 1 from B to A 6Move disk 2 from B to C 7Move disk 1 from A to C


## "Monster of Hanoi" - small example

>>> HanoiTowers(3)
1Move disk 1 from A to $C$
2Move disk 2 from A to $B$
3 Move disk 1 from $C$ to $B$$\quad\left[2^{2}-1\right.$

- If $1 \leq k<2^{n-1}$ the move we are looking for is within the first part.


## "Monster of Hanoi" - small example

>>> HanoiTowers(3)


- If $k=2^{n-1}$ this is the move we want.


## "Monster of Hanoi" - small example

>>> HanoiTowers(3)


- If $2^{n-1}<k<2^{n}$ the move is within the third part
- In this case we want the $k-2^{n-1}$ th move of this part.
- For example, the $k=6^{\text {th }}$ move is the $6-2^{2}=2^{\text {nd }}$ move in this part.


## "Monster of Hanoi" with Recursion

- To compute the $k$-th move in the an $n$ disk Tower of Hanoi puzzle, we recall the solution of the Tower of Hanoi puzzle, and think recursively:
- The solution to HanoiTowers takes $2^{n}-1$ steps altogether (so $1 \leq k \leq 2^{n}-1$ ), and consists of three (unequal) parts:

1. In the first part, which takes $2^{n-1}-1$ steps, we move $n-1$ disks. If $1 \leq k<2^{n-1}$ the move we are looking for is within this part.
2. In the second part, which takes exactly one step, we move disk number $n$. If $k=2^{n-1}$ this is the move we want.
3. In the last part, which again takes $2^{n-1}-1$ steps, we again move $n-1$ disks. If $2^{n-1}<k<2^{n}$ the move is within this part

- In this case we want the $k-2^{n-1}$ th move of this part


## Hanoi Monster - Code

```
def Hanoi_move(n, k, start="A", via="B", target="C"):
    """ Finds the k-th move in an Hanoi Towers instance with n disks.
    Uses binary search on the sequence of steps
    """
    assert k>0 and k<2**n # k should satisfy 0<k<2**n
    if k==2**(n-1): # k is the middle step
    print("Move disk", n, "from", start, "to", target)
    elif k < 2**(n-1):
    Hanoi_move(n-1, k, start, target, via)
    else:
        Hanoi_move(n-1, k-2**(n-1), via, start, target)
```

- Note the changing roles of the rods, as in the HanoiTowers function.


## Recursive Monster Code: Executions

- We first test it on some small cases, which can be verified by running the HanoiTowers program.

```
>>> Hanoi_move(3, 1)
Move disk 1 from A to
>>> Hanoi_move(3, 4)
Move disk 3 from A to
>>> Hanoi_move(3, 6)
Move disk 2 from B to
\begin{tabular}{|c|c|}
\hline & >>> HanoiTowers(3) \\
\hline C & \(\rightarrow\) Move disk 1 from A to C \\
\hline & Move disk 2 from A to B \\
\hline C & Move disk 1 from C to B \\
\hline & Move disk 3 from A to C \\
\hline C & Move disk 1 from B to A \\
\hline & \(\rightarrow\) Move disk 2 from B to C \\
\hline & Move disk 1 from A to C \\
\hline
\end{tabular}
```

- Once we are satisfied with this, we solve the monster's question.

```
>>> hanoi_move(200, 3**97+381)
Move disk 11 from B to C' # saved from monster!
```



## Recursive Monster Solution and Binary Search

- The recursive Hanoi_move makes at most one recursive call each time.
- The way it "homes" on the right move employs the already familiar paradigm of binary search:
- it first determines if move number $k$ is exactly the middle move in the $n$ disk problem. If it is, then by the nature of the problem it is easy to exactly determine the move.
- If not, it determines if
- the move is in the first half of the moves' sequence or
- The move is in the second half,
- and makes a recursive call with the correct permutation of rods (in the latter case also k should change).


## Recursive Monster Solution - Complexity

- Each time we decrease the size of the problem by about half
- $2^{n}-1 \Rightarrow 2^{n-1}-1 \Rightarrow 2^{n-2}-1 \Rightarrow \ldots \Rightarrow 2^{1}-1$
- The number of steps is linear in $n$ (and not in $2^{n}-1$, the total length of the sequence of moves).
- Each step requires $O(1)$ time, so the complexity is $O(n)$.
- Another way to look at it - we use binary search on a search space of size $2^{n}-1$. So, the time complexity is $O\left(\log \left(2^{n}-1\right)\right)=O(n)$.


[^0]:    *recursively

