Math Review

Why do we need math in a data structures course?

- ***To Analyze*** data structures and algorithms
  - Deriving formulae for time and memory requirements
  - Will the solution scale?
  - *Quantify* the results
- Proving algorithm correctness

Definition: Let $T(n)$ denote the time take by an algorithm on an input of size n.

Examples – how much “time” does each of these algorithms take?

// Assume A is an integer array of size n

```
Algorithm1 (A,n)        $T(n) \cong n$

    max = -infinity
    for (i=1 to n) {
      if(A[i]>max) max=A[i];
      }
      Output max;
```

```
Algorithm2 (A, start, end)    $T(n)$
    if (n<2) return
    mid = floor(n/2)
    if (condition#1)
            Algorithm2 (A,1,mid)  $T(n/2)$
    else
            Algorithm2 (A,mid+1,n) $T(n/2)$
```

```
Algorithm3 (A,n)     $T(n)$
    if (n<2) return
    x = floor(n/2)
    Algorithm3 (A,1,x)    $T(n/2)$
    Algorithm3 (A,x+1,n)  $T(n/2)$
```

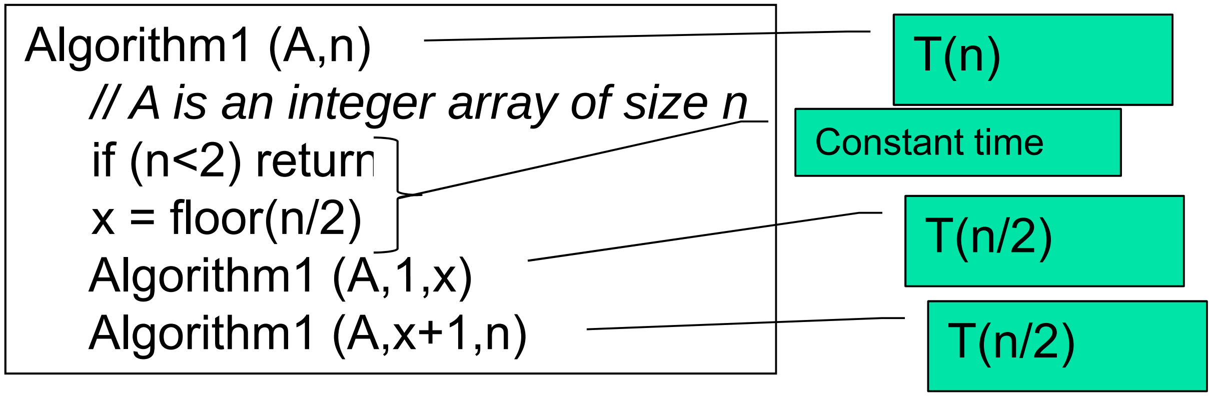
➔ $T(n) \cong T(n/2) + \text{const.}$

➔ $T(n) = 2\,T(n/2) + \text{const.}$

These are not a closed form yet.
If you solve the recurrences, you will get,
O(lg n) for Algorithm2, and
O(n) for Algorithm3

Example

- Consider Algorithm1 that divides the input array in half and calls Algorithm1 recursively on each half



```
Algorithm1 (A,n)
    // A is an integer array of size n
    if (n<2) return
    x = floor(n/2)
    Algorithm1 (A,1,x)
    Algorithm1 (A,x+1,n)
```

T(n)

Constant time

T(n/2)

T(n/2)

- What is the running time of Algorithm1?

T(n) = T(n/2) + T(n/2) + const.

This is not a closed form yet.

Floors and Ceilings

- $floor(x)$, denoted $\lfloor x \rfloor$, is the greatest integer ≤ x
- $ceiling(x)$, denoted $\lceil x \rceil$, is the smallest integer ≥ x
- Normally used to divide input into integral parts $\left\lfloor \frac{N}{2} \right\rfloor + \left\lceil \frac{N}{2} \right\rceil = N$

Exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq X^{2N}$$

$$2^N + 2^N = 2^{N+1}$$

Logarithms

$$\log_X B = A \Leftrightarrow X^A = B \quad \text{"logarithm of B base X"}$$

$$\log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0, A \neq 1$$

$$\log AB = \log A + \log B; \quad A, B > 0$$

$$\log \frac{A}{B} = \log A - \log B$$

$$\log A^B = B \log A$$

$$\log X < X \quad \text{for all } X > 0$$

$$\lg A = \log_2 A$$

$$\ln A = \log_e A; \quad e = 2.7182\ldots \quad \text{"natural logarithm"}$$

Our convention for the course:

lg n == $\log_2$ n

log n == $\log_{10}$ n

ln n == $\log_e$ n

PS: In Weiss book,
log n → $\log_2$n

- What is the meaning of the log function?
    - For example, lg 1024

Example

- How many times to halve an array of length n until its length is 1?

```
KeepHalving (n)
    i = 0
    while n ≠ 1
        i = i + 1
        n = floor(n/2)
    return i
```

What will be the value of i?

Factorials

$$n! = \begin{cases} 1 \text{ if } \; n = 0 \\ n * (n-1)! \text{ if } \; n > 0 \end{cases}$$

$$n! < n^n$$

$$\mathrm{n}! = \sqrt{2\pi \mathrm{n}}\,(n/e)^n (1 + \theta(1/n)) \quad \text{Stirling's approximation}$$

n! == how many ways to order a set of n elements?

Modular Arithmetic

$$A \bmod N = A - N * \lfloor A/N \rfloor$$

$$(A \bmod N) = (B \bmod N) \Rightarrow A \equiv B \pmod{N}$$

"A is congruent to B modulo N"

E.g., $81 \equiv 61 \equiv 1 \pmod{10}$

If $A \equiv B \pmod{N}$

Then $A + C \equiv B + C \pmod{N}$

and $AD \equiv BD \pmod{N}$

Basis of most encryption schemes: (Message mod Key)

Series

- General $\sum_{i=0}^{N} f(i) = f(0) + f(1) + \ldots + f(N)$
- Linearity $\sum_{i=0}^{N} (cf(i) + g(i)) = c\sum_{i=0}^{N} f(i) + \sum_{i=0}^{N} g(i)$
- Arithmetic series $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

Series

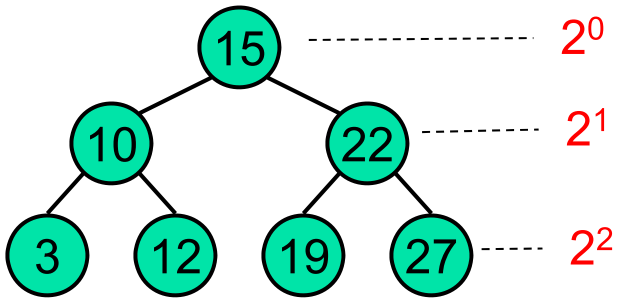
- Geometric series

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$

$$\sum_{i=0}^{N} A^i \le \sum_{i=0}^{\infty} A^i = \frac{1}{1-A}; \quad \text{if } 0 < A < 1$$

Example

How many nodes
in a <u>complete</u> binary tree
of depth D?



A=2, N=D=2 $\Rightarrow$ $(2^{2+1}-1) / (2-1)$

$\Rightarrow$ **7**

Proofs

- What do we want to prove?
  - Properties of a data structure always hold for all operations
  - Algorithm’s running time / memory will never exceed some threshold
  - Algorithm will always be correct
  - Algorithm will always terminate
- Techniques
  - Proof by induction
  - Proof by counterexample
  - Proof by contradiction

Variation:
Ind/hyp: All values < k,
Ind/step: show for value=k

Proof by Induction

- Goal: Prove some hypothesis is true
- Three-step process
  1. Base case: Show hypothesis is true for some initial conditions
  2. Inductive hypothesis: Assume hypothesis is true for all values ≤ k
  3. Inductive step: Show hypothesis is true for next larger value (typically k+1)

Inductive Proof: Example

- Prove arithmetic series

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

- Base case: Show true for N=1

$$\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$$

➔ Base case verified

Example (cont.)

<u>Variation:</u>
Assume hyp for N<k, and
Check for N=k

- <u>Ind/Hyp:</u> Assume true for all N<=k
- <u>Ind/Step:</u> Now see if it is true for N=k+1

$$
\begin{aligned}
\sum_{i=1}^{k+1} i &= (k+1) + \sum_{i=1}^{k} i \\
&= (k+1) + \frac{k(k+1)}{2} \\
&= \frac{2(k+1) + k(k+1)}{2} \\
&= \frac{(k+1)(k+2)}{2}
\end{aligned}
$$

More Examples for Induction Proofs

- Prove the geometric series

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$

- Prove that the number of nodes N in a complete binary tree of depth D is $2^{D+1}-1$

Proof by Counterexample

Prove hypothesis is not true by giving an example that doesn't work

  - Example: **$2^N > N^2$?**
  - Proof: N=2 (or 3 or 4)

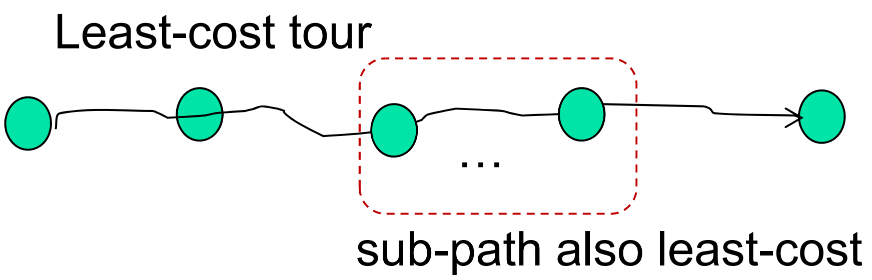
- ~~Proof by example?~~
- ~~Proof by lots of examples?~~
- Proof by all possible examples?
  - Empirical proof
  - Hard when input size and contents can vary arbitrarily

Another Example for a proof by Counterexample

Given N cities and costs for traveling between each pair of cities, a "***least-cost tour***" is one which visits every city exactly once with the least cost

Hypothesis: *Any sub-path within any least-cost tour will also be a least-cost tour for those cities included in the sub-path.*

Is this hypothesis true?

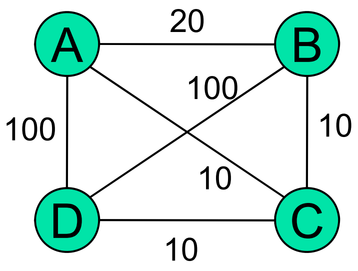


Proof by counterexample

- Counterexample
  - Cost (A→B→C→D) = 40 (optimal)
  - Cost (A→B→C) = 30
  - Cost (A→C→B) = 20

<u>Not</u> the least cost tour for {A,B,C}

Least cost tour for {A,B,C}



<u>Conclusion:</u> Least cost tours don't necessarily contain smaller least cost tours

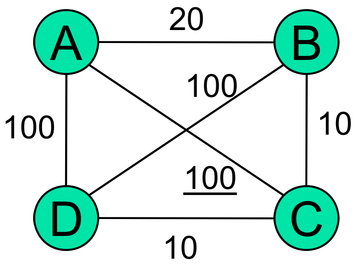
Proof by Contradiction

1. Start by assuming that the hypothesis is false
2. Show this assumption could lead to a contradiction (i.e., some known property is violated)
3. Therefore, hypothesis must be true

Example for proof by contradiction

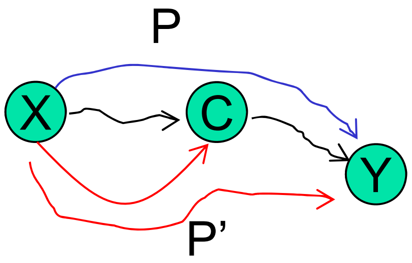
Single pair shortest path problem

  - Given N cities and costs for traveling between each pair of cities, find the least-cost path to go from city X to city Y
- *Hypothesis: A least-cost path from X to Y contains least-cost paths from X to every city on the path*
  - E.g., if X→C1→C2→C3→Y is a least-cost path from X to Y, then
    - X→C1→C2→C3 must be a least-cost path from X to C3
    - X→C1→C2 must be a least-cost path from X to C2
    - X→C1 must be a least-cost path from X to C1



Conclusion: Least cost paths should contain smaller least cost paths starting at the source

Proof by contradiction..



- Let P be a least-cost path from X to Y
- Now, assume that the hypothesis is false:
  - ==> there exists C along X->Y path, such that, there is a better path from X to C than the one in P
  - ==> So we could replace the subpath from X to C in P with this lesser-cost path, to create a new path P' from X to Y
  - ==> Thus we now have a better path from X to Y
    - i.e., cost(P') < cost(P)
  - ==> But this violates the fact that P is a least-cost path from X to Y
  - (hence a contradiction!) ↯
- Therefore, the original hypothesis must be true

Mathematical Recurrence vs. Recursion

A recursive function or a recursive formula is defined in terms of itself

- Example:

$$n! = \begin{cases} 1 \text{ if } \ n = 0 \\ n * (n-1)! \text{ if } \ n > 0 \end{cases}$$

```
Factorial (n)
    if n = 0
    then return 1
    else return (n * Factorial (n-1))
```

Mathematical Recurrence

Recursion (code)

Basic Rules of Recursion

- Base cases
  - Must always have some base cases, which can be solved without recursion
- Making progress
  - Recursive calls must always make progress toward a base case
- Design rule
  - Assume all recursive calls work
- Compound interest rule
  - Try not to duplicate work by solving the same instance of a problem in separate recursive calls

Example

- Fibonacci numbers
  - F(0) = 1
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)

recurrence

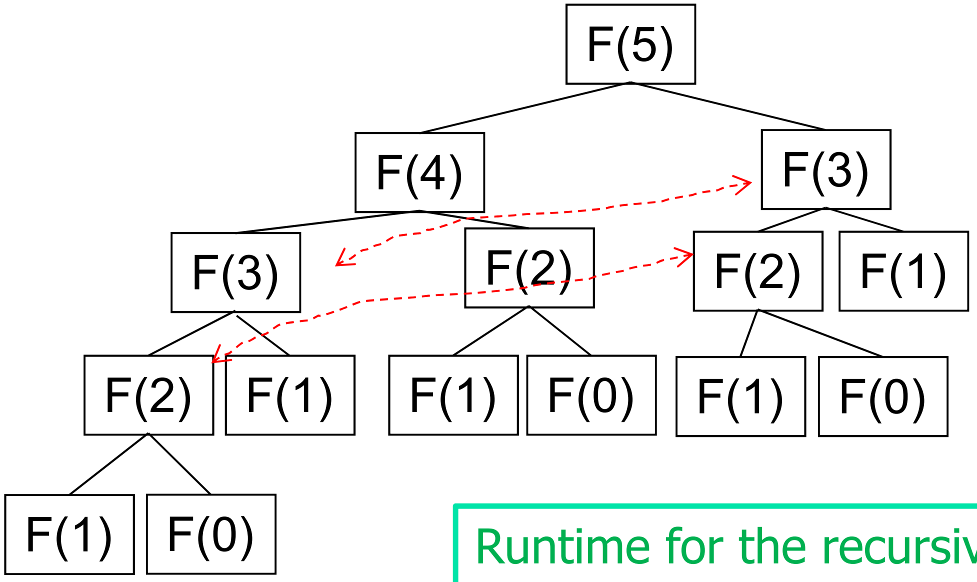
```
Fibonacci (n)
    if (n $\le$ 1)
    then return 1
    else return (Fibonacci (n-1) + Fibonacci (n-2))
```

recursive code

So, is there a better way to write the Fibonacci code?

Example (cont.)

- *Computation tree* for: Fibonacci (5)



Runtime for the recursive code (previous slide)
: is proportional to the size of the tree
: and that is a lot wasteful.
: Why?

Running time for Fibonacci(n)?

- Show that the running time T(n) of Fibonacci(n) is exponential in n
- Use mathematical induction
  - We can show that **$T(n) < (5/3)^n$** for n>=1
- Actually, this gives only an *upper bound* for T(n)
  - We also need to prove that T(n) is at least exponential

Solving recurrences

- Example:

```
Algo1(A,1,n)
    // A is an integer array of size n
    if(n<2) return;
    x = floor(n/2)
    Algo1(A,1,x)
    Algo1(A,x+1,n)
```

- How much time does Algo1 take?
  - Express time as a function of n (input size)
- Let T(n) be the time taken by Algo1 on an input size n
- Then, $T(n) = 1 + T(n/2) + T(n/2)$
- $= 2T(n/2) + 1$

Solving recurrences…

- Recurrence:
  $T(n) = 2T(n/2) + 1$
  $T(1) = 1$ (base case)
- Solution:
  - $T(n) = 2T(n/2) + 1$
  - $= 2[2T(n/2^2) + 1] + 1$
  - $= 2^2T(n/2^2) + 2 + 1$
  - $= 2^3T(n/2^3) + 2^2 + 2 + 1$
  - … (k steps)
  - $= 2^kT(n/2^k) + 2^{k-1} + \ldots + 2^2 + 2 + 1$
  - For termination, $n/2^k = 1 \Rightarrow k=\lg n$
  - $T(n) = 2^{\lg n}T(1) + n-1$
  - $= 2n-1$

This is the closed form for T(n)

Ponder this

1. Do constants matter for asymptotic analysis?

2. Recurrence vs. Recursion

   - A recurrence *need not* always be implemented using recursion
   - How?

Notion of a “recursion” as a function calling a function (same or not)

Recursive Function Calls

Code structure: (guess)?

```
M() {
  A()
  A()
  B()
  C()
}
```

```
A() {
}
```

```
B() {
  B()
}
```

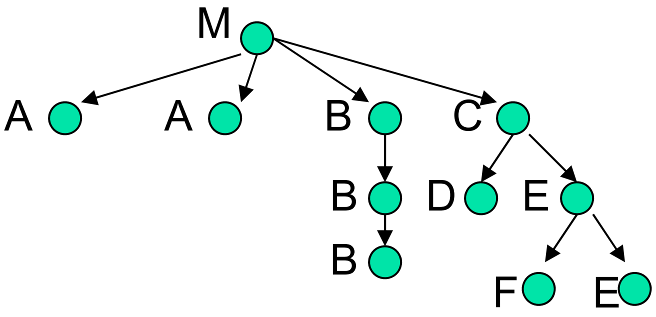
```
C() {
  D()
  E()
}
```

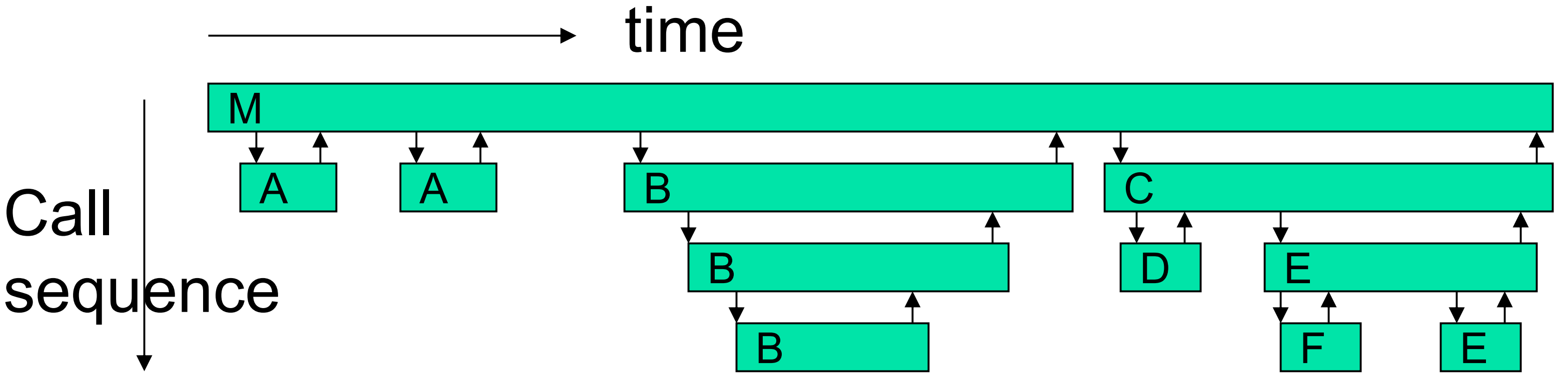
```
D() {
}
```

```
E() {
  F()
  E()
}
```

```
F() {
}
```

Call tree:



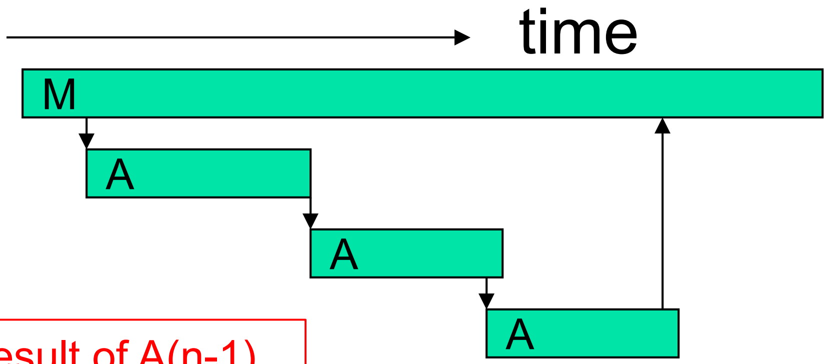


Why is iterative code more desirable than tail recursive code?

Refer to the note on tail recursion on the lecture notes web page

Tail Recursion → Iteration

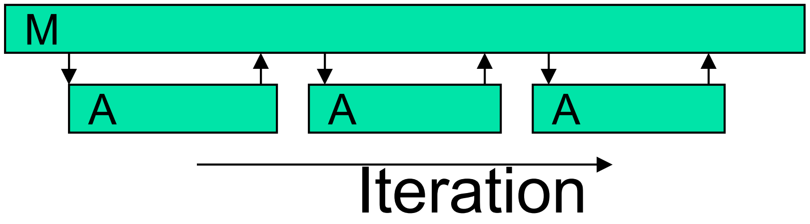
```
A(n) {
  …
  A(n-1)
}
```



Tail recursive code

The result of A(n-1) is NOT needed/used in A(n)

```
A(n) {
  for(i=n;i>=0;i--) {
    …
  }
}
```



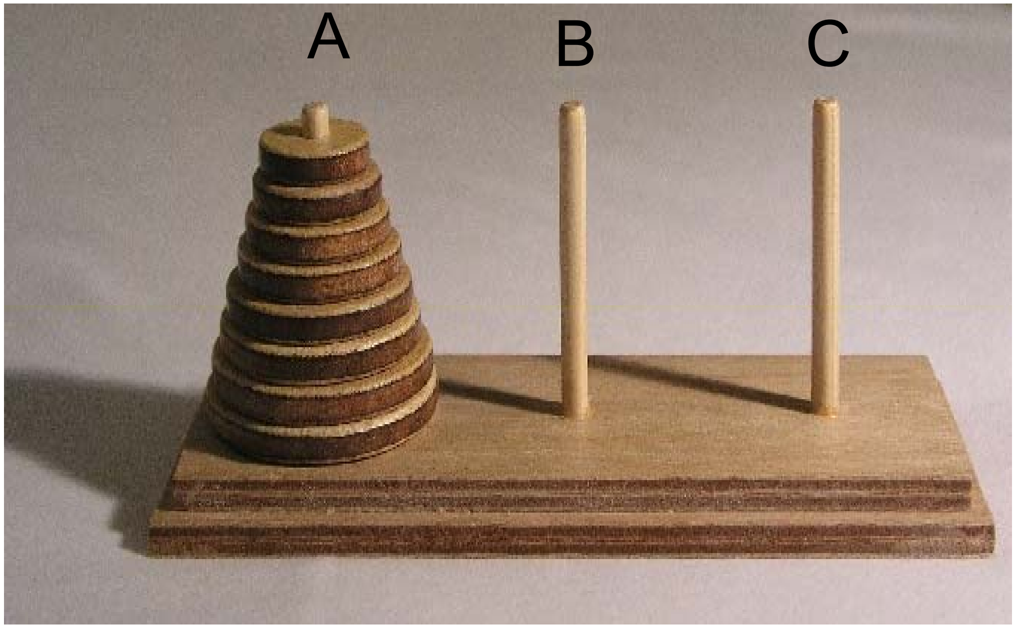
Last recursive call replaced with while() or for() loop

Tower of Hanoi

Goal: Move all disks from peg A to peg B using peg C

Rules:

1. Move one disk at a time
2. Larger disks cannot be placed above smaller disks

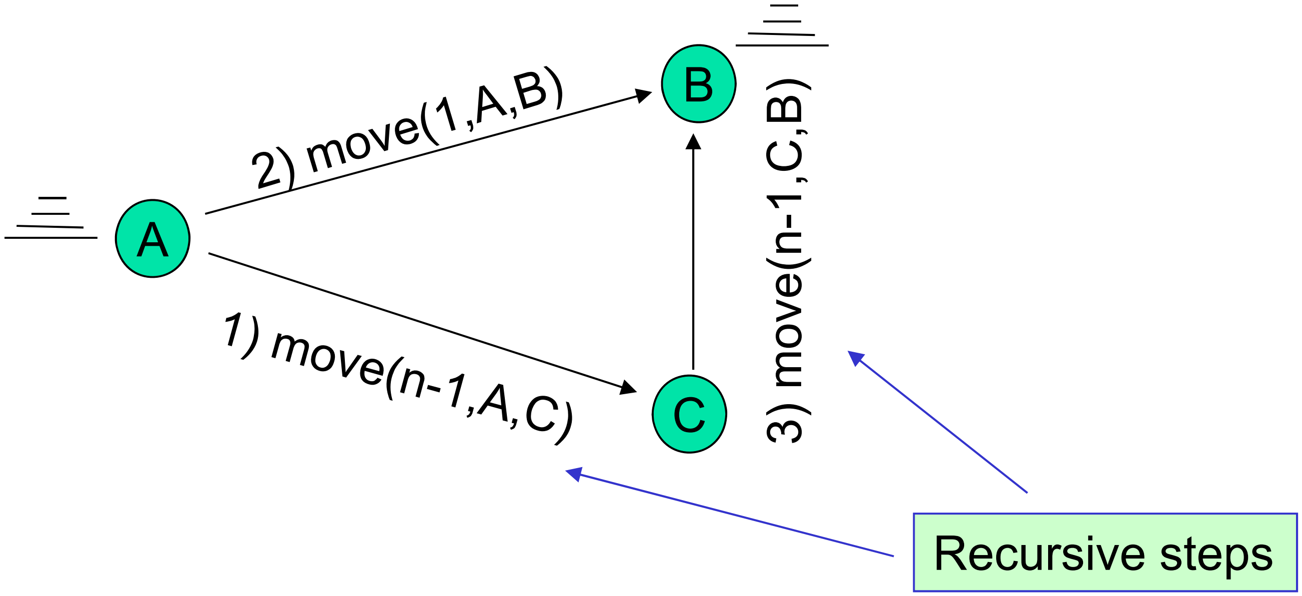


Invented by a French Mathematician Edouard Lucas, 1883

Question: What is the minimum number of moves necessary to solve the problem?

Tower of Hanoi: Algorithm

- <u>A Recursive Algorithm:</u>
  1. First, move the top n-1 disks, “recursively”, from A to C (using B)
  2. Move nth disk (ie., largest & bottom-most in A) from A to B
  3. Then, move all the n-1 disks, “recursively”, from C to B (using A)



Recursive steps

Recursive Algorithm for Tower of Hanoi (pseudocode)

src dst temp

```
Move (n: disk, A, B, C)
PRE: n disks on A; B and C unaffected
POST: n disks on B; A and C unaffected
BEGIN
    IF n=0 THEN RETURN
    Move (n-1, A,C,B)
    Move $n^{th}$ disk from A to B directly
    Move (n-1,C,B,A)
END
```

Tail Recursion

Tower of Hanoi: Analysis

- Let T(n)= minimum number of moves required to solve the problem
- Analysis:
  - T(1)=1 ➔ Base case
  - T(n) = 2.T(n-1)+1 ➔ recurrence
- Solving this yields $T(n)=2^n-1$ (how?)
- In the original Tower of Hanoi problem, n=8 & so T(n)=255 (which is fine!)
- For Tower of Brahma, n=64
  - ⇒ 2^{64}-1 moves made by a priest in a temple
  - ⇒ Assuming each move takes 1 second, this would take 5,000,000,000 centuries to complete
  - ⇒ So lots of time before the world ends!

Summary

- Floors, ceilings, exponents, logarithms, series, and modular arithmetic
- Proofs by mathematical induction, counterexample and contradiction
- Recursion
- Solving recurrences
- Tools to help us analyze the performance of our data structures and algorithms

Try it out yourself

- http://www.mazeworks.com/hanoi/index.htm