Data Structures & Algorithms

Lecture 2: Asymptotic Analysis & Recursion

Many slides by Kevin Buchin

Recap: Algorithms

- A complete description of an algorithm consists of three parts:
  1. the algorithm, *expressed in whatever way is clearest and most concise*
  2. a proof of the algorithm's correctness
  3. a derivation of the algorithm's running time

Recap: Correctness Proofs for Loops

first formulate a …

Loop invariant: an assertion that we prove to be true each time a loop iteration starts

… then prove correctness in 3 steps …

Initialization

Invariant is true prior to the first iteration of the loop.

Maintenance

If the invariant is true before an iteration of the loop, it remains true before the next iteration.

Termination

The loop terminates, and when it does, the loop invariant, along with the reason that the loop terminated gives us a useful property.

Recap: Finding a Loop Invariant

- Loop Invariant (generally) proves something that is growing
  - I.e., $A[1..i-1]$

- What do you want to know at the end?

- Think about a specific iteration.
  - What do you **know**?

- What is the algorithmic technique used, e.g., are you incrementally computing the solution?

  (Typical) Loop Invariant

  *At the start of iteration i, **answer** contains "the answer" for $A[1..i-1]$.*

Efficiency

Efficiency analysis

1. Determine running time as function T(n) of input size n
2. Characterize rate of growth of T(n)

- Focus on the order of growth
  ignore all but the most dominant terms

```
Linear-Search(A, n, x)

1.Set answer to Not-Found
2.For each index i, going from 1 to n, in order:
      A.   If A[i] = x, then set answer to the value of i
3.Return the value of answer as the output
```

n single integer, x as large as array element, hence insignificant …

- input size is n: number of elements in A

Efficiency analysis

1. Determine running time as function T(n) of input size n
   - best case
   - average case
   - worst case

> An algorithm has worst case running time T(n) if for any input of size n the maximal number of elementary operations executed is T(n).

elementary operations

add, subtract, multiply, divide, load, store, copy, conditional and unconditional branch, return …

Efficiency analysis

Linear-Search(A, n, x)

1. Set answer to Not-Found
2. For each index i, going from 1 to n, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Assumption

each execution of step i takes t_i time (independent of n)

1. t_1
2. t_2' (test i against n) + t_2'' (increment i)
   A. t_{2A}' (test A[i] = x) + t_{2A}'' (set answer to i)
3. t_3

Efficiency analysis

Linear-Search(A, n, x)

1. Set answer to Not-Found
2. For each index i, going from 1 to n, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

---

1. t_1
2. t_2' (test i againt n) + t_2'' (increment i)
   A. t_{2A}' (test A[i] = x) + t_{2A}'' (set answer to i)
3. t_3

Running time between

$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot 0 + t_3$ and

$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3$

Efficiency analysis

Linear-Search(A, n, x)

1. Set answer to Not-Found
2. For each index i, going from 1 to n, in order:
   A. If A[i] = x, then set answer to the value of i
3. Return the value of answer as the output

Running time between

$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot 0 + t_3$ and

$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3$

lower bound $(t_2' + t_2'' + t_{2A}') \cdot n + (t_1 + t_2' + t_3)$ ➡ $c_1 \cdot n + d$

upper bound $(t_2' + t_2'' + t_{2A}' + t_{2A}'') \cdot n + (t_1 + t_2' + t_3)$ ➡ $c_2 \cdot n + d$

➡ bounded below and above by linear function in n ➡ $\Theta(n)$

Θ-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

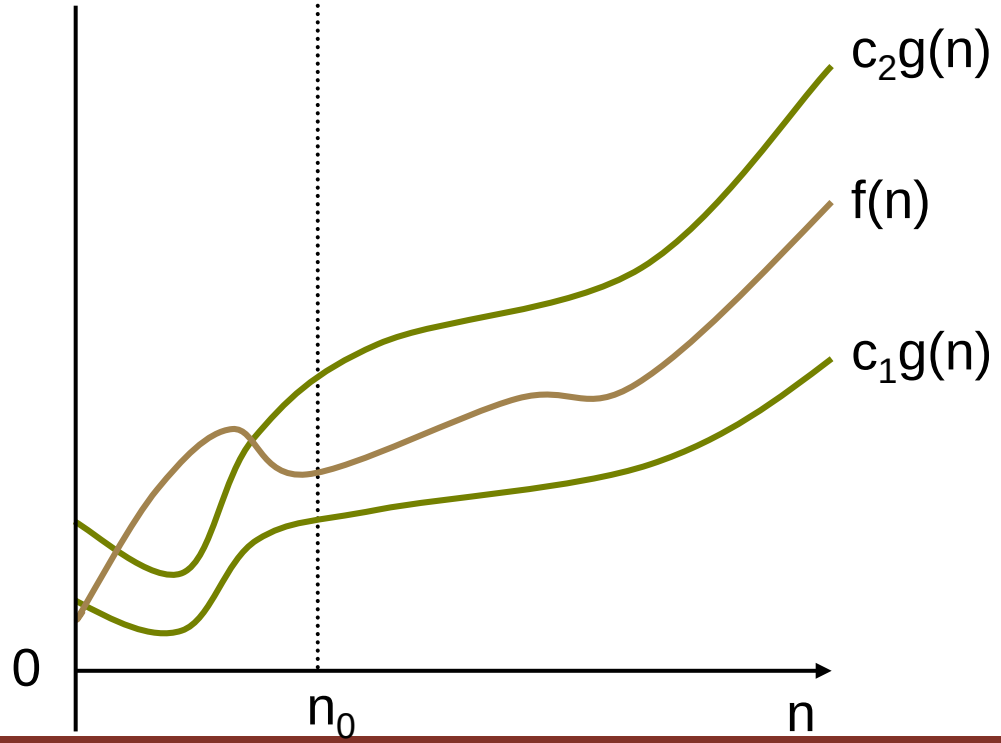
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0
such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

"$\Theta(g(n))$ is the set of functions that grow as fast as $g(n)$"

Θ-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$



Notation: $f(n) = \Theta(g(n))$

Θ-notation

Example

$n^2/4 + 100\,n + 50 = \Theta(n^2)$

$n > 400$ ➡ $n^2/4 > 100\,n + 50$

$n = 1000$ ➡ $n^2/4 = 250{,}000$ and $100\,n + 50 = 100{,}050$

$n = 2000$ ➡ $n^2/4 = 1{,}000{,}000$ and $100\,n + 50 = 200{,}050$

…

Efficiency analysis

Better-Linear-Search(A, n, x)

1. For i = 1 to n:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

- Running time?
- How often does line 1A execute?
  in the worst case and in the best case …

Worst case: x is not in the array ➡ Θ(n)

Best case: A[1] = x ➡ Θ(1)

- A linear function is an upper bound ➡ O(n)

the running time might be better than linear, but never worse …

O-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

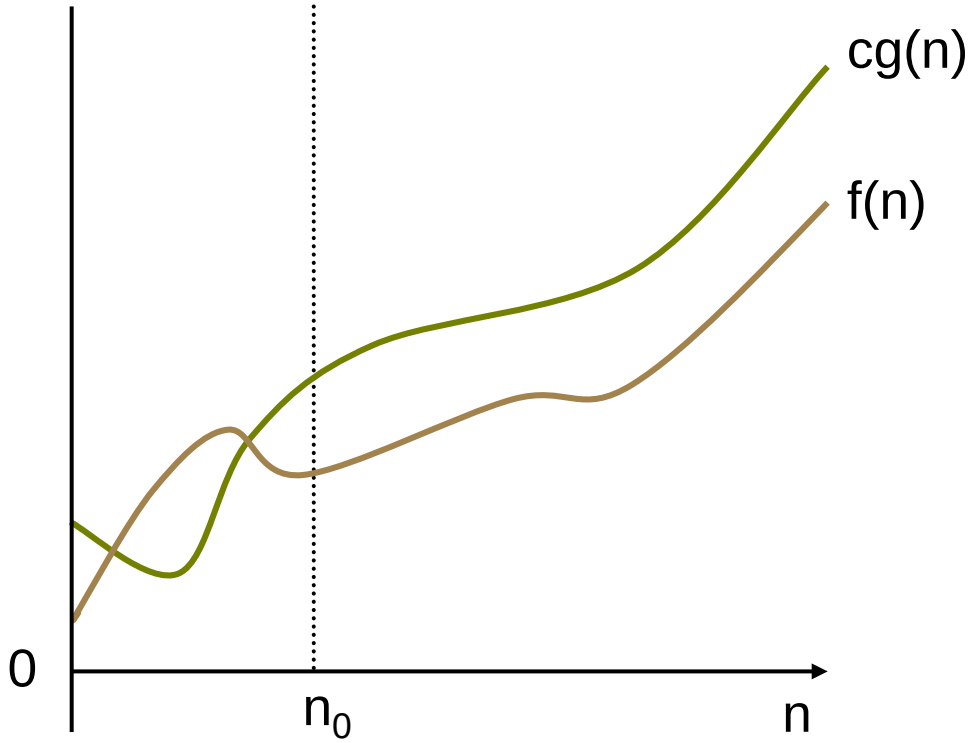
$O(g(n))$ = { $f(n)$: there exist positive constants c and n_0
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }

"$O(g(n))$ is the set of functions that grow
at most as fast as $g(n)$"

O-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0
such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$



Notation: $f(n) = O(g(n))$

Efficiency analysis

- O-notation
  running time is never worse than given function upper bound
- Lower bound: the running time is never better than a given function
- Better-linear-search: Ω(1)
- *How meaningful is best-case running time?*

Whenever I ask you to analyze the running time,

- you should analyze the worst-case running time,
- and aim at a tight upper bound

Ω-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

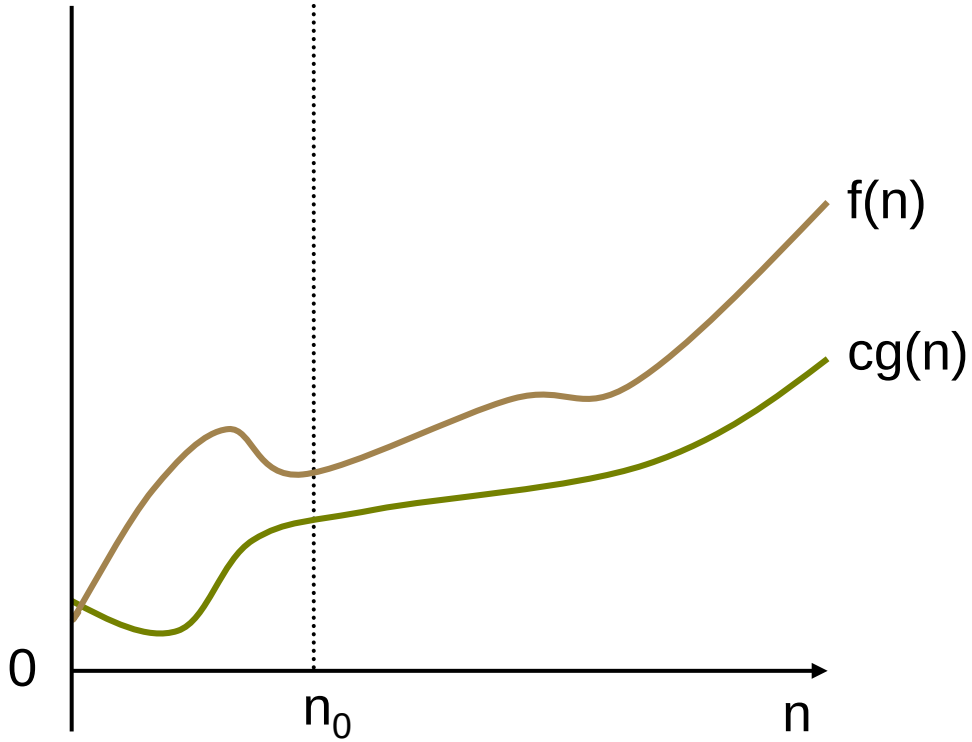
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

"$\Omega(g(n))$ is the set of functions that grow at least as fast as $g(n)$"

Ω-notation

Let $g(n) : N \mapsto N$ be a function. Then we have

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$



Notation: $f(n) = \Omega(g(n))$

Asymptotic notation

Θ(…) is an asymptotically tight bound "asymptotically equal"

O(…) is an asymptotic upper bound "asymptotically smaller or equal"

Ω(…) is an asymptotic lower bound "asymptotically greater or equal"

- other asymptotic notation

  o(…) → "grows strictly slower than"
  ω(…) → "grows strictly faster than"

Exercise: Running Time (revisited)

Exercise 3

We define the problem CountInInterval as follows: *Given an array A of n integers, and two integers p and q, count the number of elements of A that are at least p and at most q.*

Give an algorithm for CountInInterval. *Don't forget to prove correctness and analyze the running time.*

Quiz

1. $500\,n + 150 = O(n^2)$ — true
2. $4\,n \log^2 n + 20\,n = \Theta(n \log n)$ — false
3. $n \log n = \Omega(n)$ — true
4. $n \log^2 n = O(n^{4/3})$ — true
5. $57\,n^2 + 17 \log n = \Theta(n^2)$ — true
6. An algorithm with worst case running time $O(n \log n)$ is always slower than an algorithm with worst case running time $O(n)$ if n is sufficiently large. — false

Intermezzo: Sums

- Do you know the arithmetic series?
- $S_n = 1 + 2 + \ldots + n = \ldots$?
- $\ldots = n\,(n+1)\,/2$ [proof?]
- $S_n = \Theta(\ldots)$?
- $= \Theta(n^2)$

- Geometric series: $1 + q + q^2 + q^3 + \ldots + q^n = \ldots$?
- $= (q^{n+1}-1)/(q-1)$ if q is not 1
- $\Theta(\ldots)$?
- If $q<1$: $\Theta(1)$, if $q=1$: $\Theta(n)$, if $q>1$: $\Theta(q^n)$

Common Summations

- Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Geometric series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \left(x \neq 1\right)$$

  - Special case: $|x| < 1$:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

- Harmonic series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- Other important formulas:

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Example: Linear Search

Better-Linear-Search(A, n, x)

1. For i = 1 to n:
   A. If A[i] = x, then return the value of i as the output
2. Return Not-Found as the output

- Running time?

- Analysis: lines 1.-1A. take constant (“O(1)”) time per iteration and are executed at most n+1 times; thus lines 1.-1A. take O(n) time. Line 2 takes O(1) time. The overall running time is O(n)+O(1)=O(n)

- A linear function is an upper bound ➡ O(n)

the running time might be better than linear, but never worse …

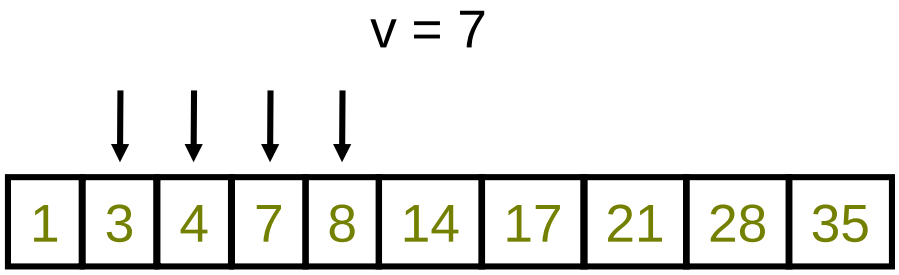
Iterative Binary Search

Input: **increasing** sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and value v

Output: an index i such that A[i] = v or NIL if v not in A

IterativeBinarySearch(A, v)

1. x = 0
2. y = n + 1
3. **while** x + 1 < y
4. **do** h = $\lfloor (x + y)/2 \rfloor$
5. **if** A[h] ≤ v **then** x = h
6. **else** y = h
7. **if** A[x] = v **then** return x **else** return NIL

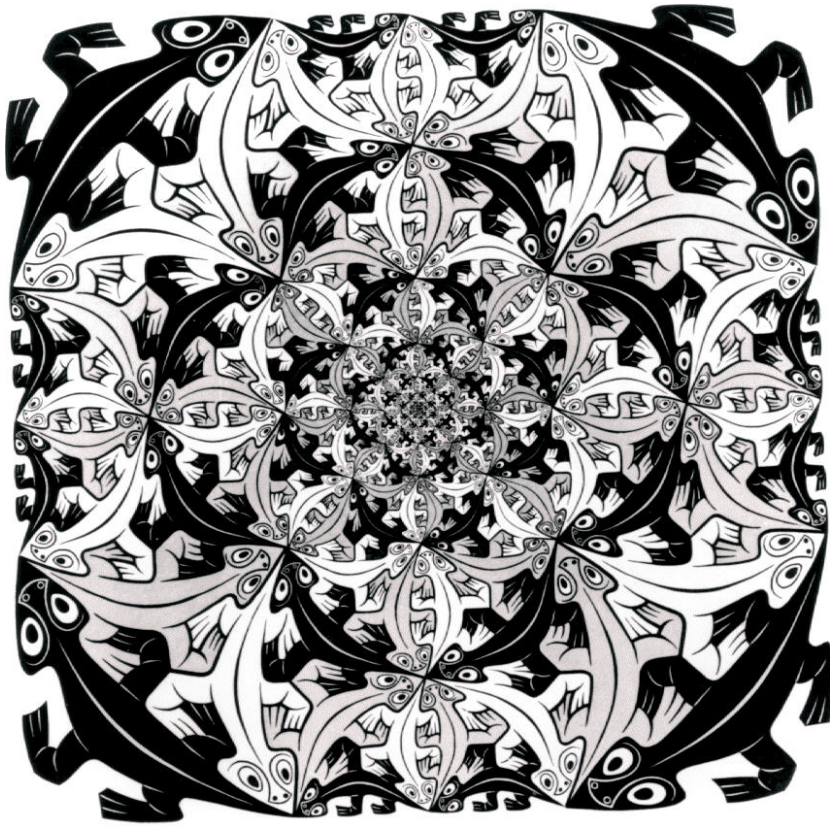


Running time: worst case O(log n)

- analysis: lines 1,2 and 7 take O(1) time, one iteration of the **while**-loop takes O(1) time. In each iteration the range from x to y is halved, therefore after log(y-x)=O(log(n)) iterations only one element is left.

RECURSIVE ALGORITHMS

Recursion



- “Recursion“: defining something in terms of itself:
  - *A directory is a collection of files and directories.*
  - *Words in dictionaries are defined in terms of other words.*

Recursive Algorithms

recursive algorithms are based on reduction:

solve a problem based on smaller instances

"In order to understand recursion, one must first understand recursion."

— *Anonymous*

“Do the hard jobs first. The easy jobs will take care of themselves.”

— *Dale Carnegie*

vs.

Factorial

- Factorial: n! = n*(n-1)*(n-2)*...*1
  = n * (n-1)!

```
# Compute n!
def factorial(n):
  if n <= 1:
    return 1
  else:
    return n * factorial(n-1)
```

Factorial

- Why doesn't the following work?

```
def factorial(n):
  return n * factorial(n - 1)
```

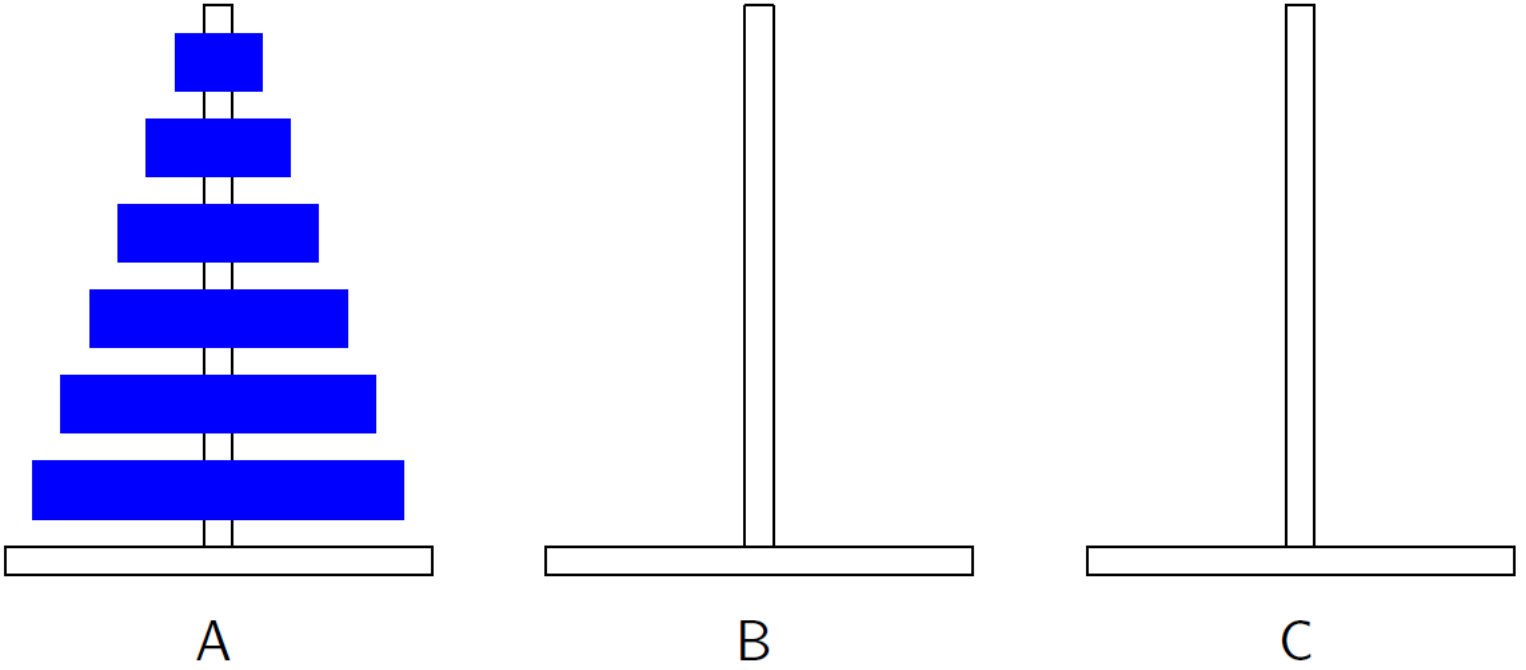
- and this one?

```
def factorial(n):
  if n <= 1:
    return 1
  else:
    return n * factorial(n)
}
```

- always take care of the base case(s) and always reduce to simpler instances

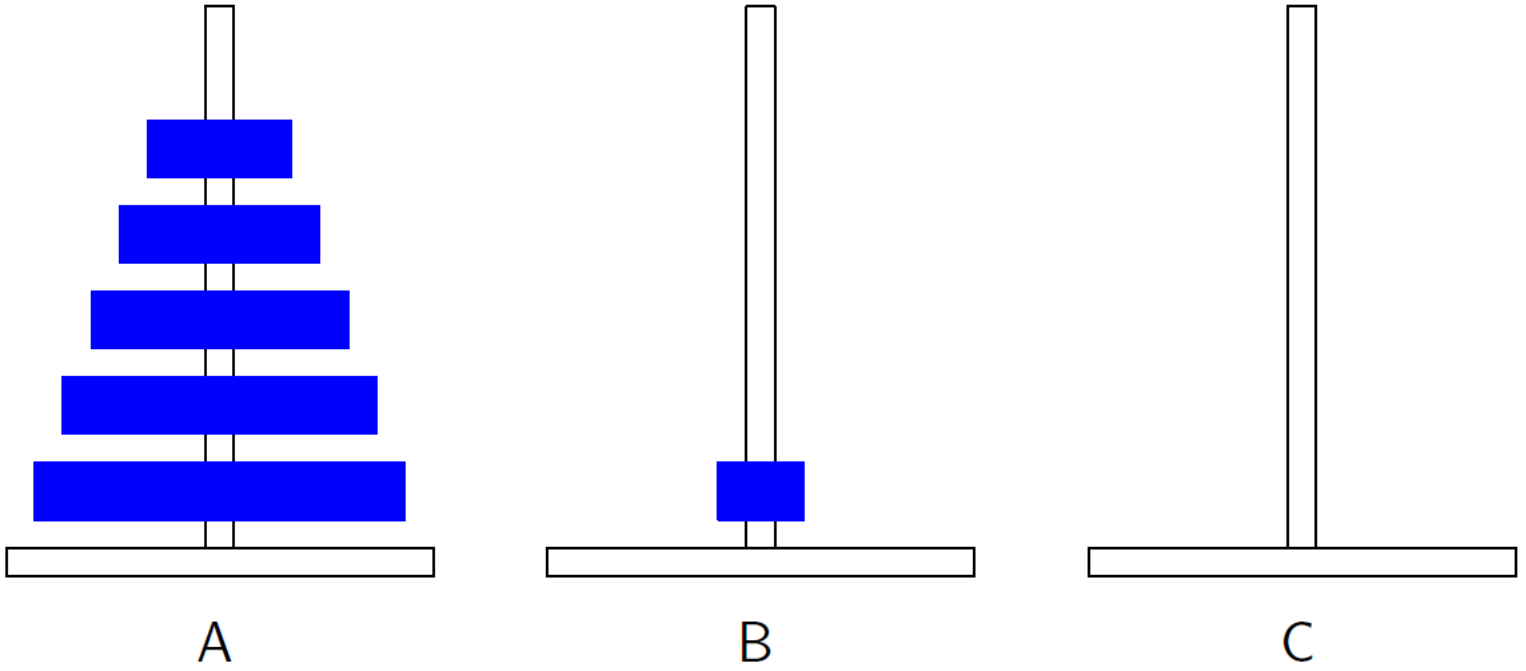
Towers of Hanoi

- Three poles, *n* disks



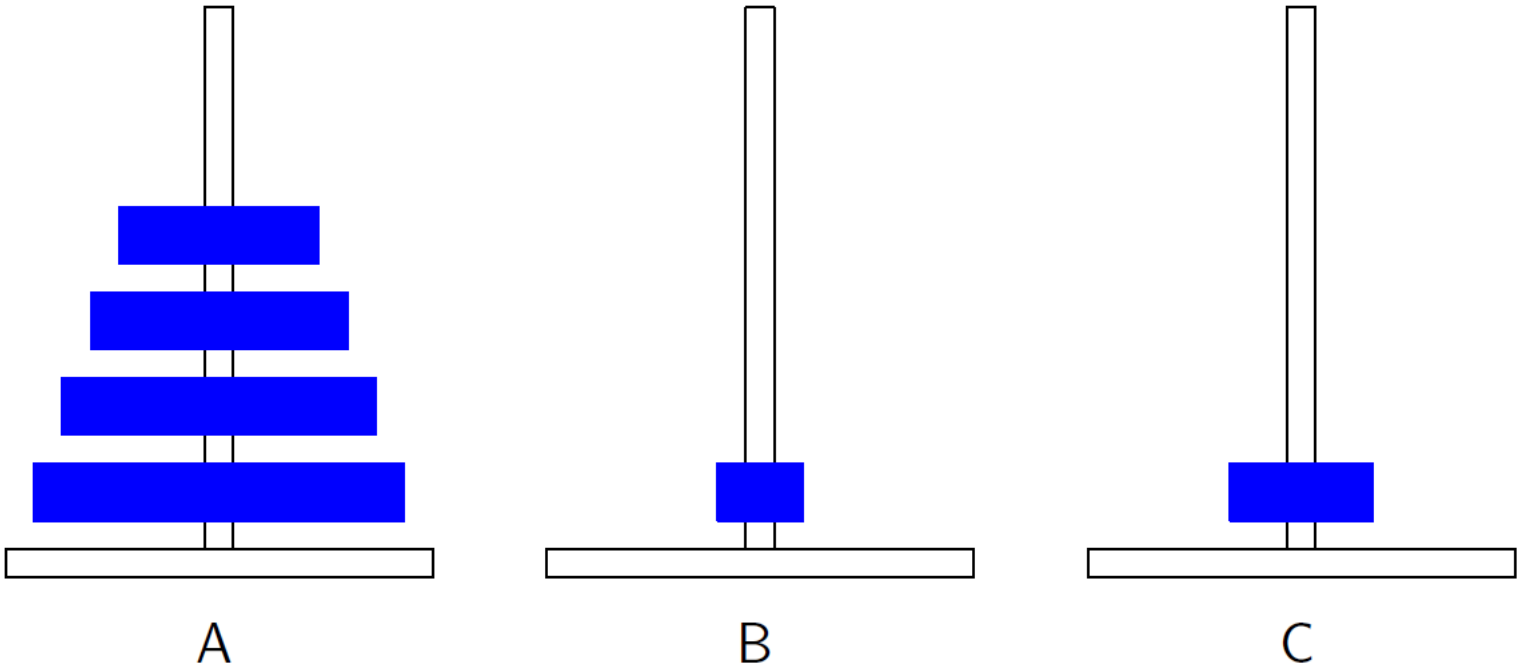
Towers of Hanoi

- Three poles, *n* disks
- One move: take the top disc from one pole and move it to another pole.



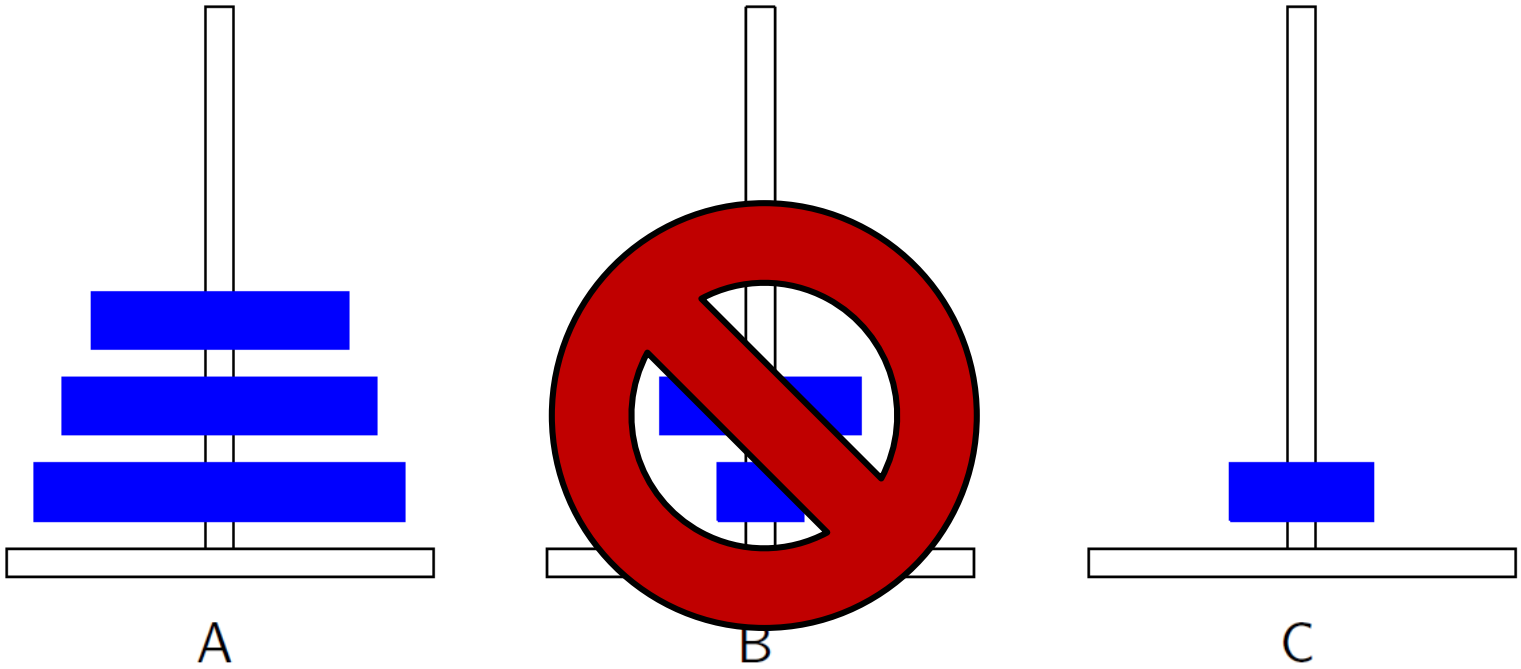
Towers of Hanoi

- Three poles, *n* disks
- One move: take the top disc from one pole and move it to another pole.



Towers of Hanoi

- Three poles, *n* disks
- One move: take the top disc from one pole and move it to another pole.
- A disc cannot lie on top of a smaller disc.



Towers of Hanoi

- Three poles, *n* disks
- One move: take the top disc from one pole and move it to another pole.
- A disc cannot lie on top of a smaller disc.

- Algorithm to move a stack of *n* disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*)?

```
Hanoi (n, src, dst, tmp):
1.      if n>0:
2.          Hanoi (n-1, src, tmp, dst)
3.          move disc n from src to dst
4.          Hanoi (n-1, tmp, dst, src)
```

Recursive Binary Search

Input: **increasing** sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and value v

Output: an index i such that A[i] = v or NIL if v not in A

```
BinarySearch(A, v, x, y)
1.  if x + 1 < y
2.      h = ⌊(x + y)/2⌋
3.      if A[h] ≤ v then return BinarySearch(A, v, h, y)
4.                   else return BinarySearch(A, v, x, h)
5.  else if A[x] = v then return x
6.                   else return NIL
```

Initial call: BinarySearch(A, v, 0, n+1)

Correctness (informal)

- When arguing about the correctness of a recursive method, always assume that the recursive call works.

The Recursion Fairy takes care of that.

- There has to be a base case.
- And we need to be sure that we will reach the base case eventually there has to be some progress in each recursive call.

- In other words, the version given to the Recursion Fairy has to be easier than the original problem.
- To prove correctness of recursive algorithms use induction

Example: Recursive Binary Search

Input: **increasing** sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and value v

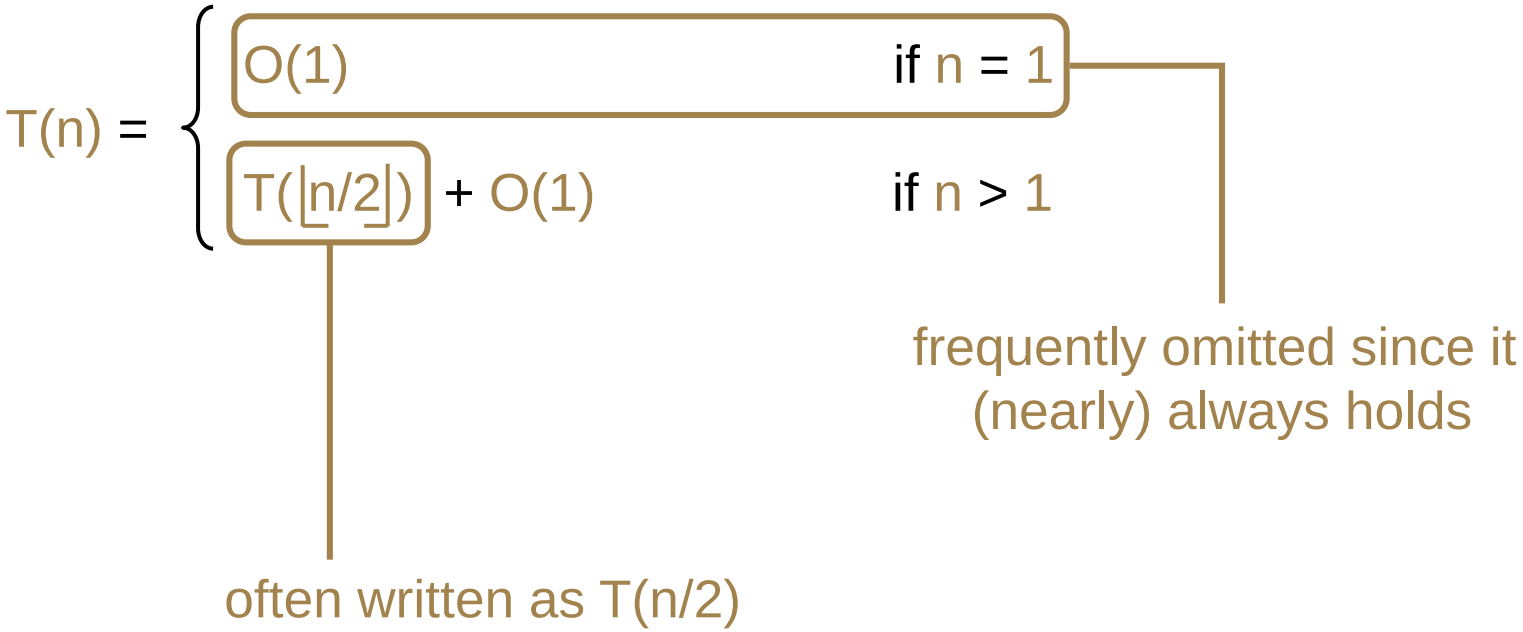
Output: an index i such that $A[i] = v$ or NIL if v not in A

```
BinarySearch(A, v, x, y)
1.  if x + 1 < y
2.      h = ⌊(x + y)/2⌋
3.      if A[h] ≤ v then return BinarySearch(A, v, h, y)
4.                   else return BinarySearch(A, v, x, h)
5.  else if A[x] = v then return x
6.                   else return NIL
```

Initial call: BinarySearch(A, v, 0, n+1)

Analysis of BinarySearch

- Let $T(n)$ be the worst case running time of BinarySearch on an array of length n. We have



Solving recurrences (a basic method)

- Use the following method
  - Iteration: expand the recursion, add the terms, and simplify the summation
    - Solve the following recursions
      1. $T(n) = T(n/2)$
      2. $T(n) = T(n/2) + n$

- Other methods (in CSc 140):
  1. Substitution: guess the solution and use induction to prove that your guess it is correct
  2. Recursion tree: carefully evaluate the recursion tree
  3. Master theorem (*caveat: not always applicable*)

Recap and preview

Today

- Efficiency analysis
- Recursive Algorithms
- Binary Search
- Correctness proofs via induction
- Outlook on running time analysis

Next lecture

All of what we have seen by the example of sorting algorithms

Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: a permutation of the input such that $\langle a_{i1} \leq \ldots \leq a_{in} \rangle$