Chapter 2

Algorithms

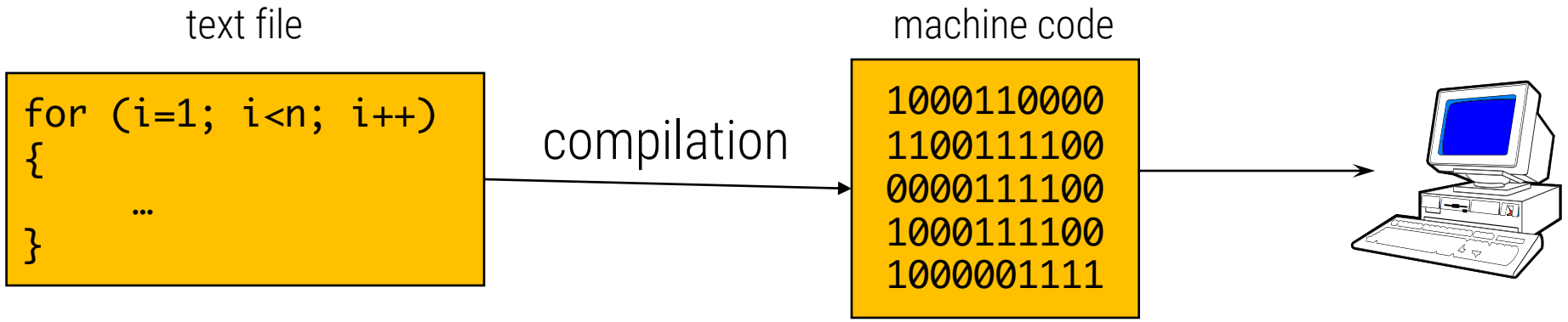
"Named after an Iranian mathematician,

Al Khawarizmi"
Dictionary of Computing

COMP2119

Programs

A program specifies in the exact syntax of some programming language the computation one expects a computer to perform. It specifies *a finite sequence of computer instructions* which tells the machine what to do exactly (fine details).



Algorithms

An algorithm is similar to a computer program in that it may specify essentially the same computational steps as a specific program written in a specific language such as Basic, Pascal, or C++.

Yet, the purpose of an algorithm is to *communicate a computation procedure not to computers but to people*. (We tend to skip fine details.)

Algorithms

An algorithm is the (programming) *language independent* and *machine independent* strategy that a program uses. Algorithms are to programs like plots are to novels.

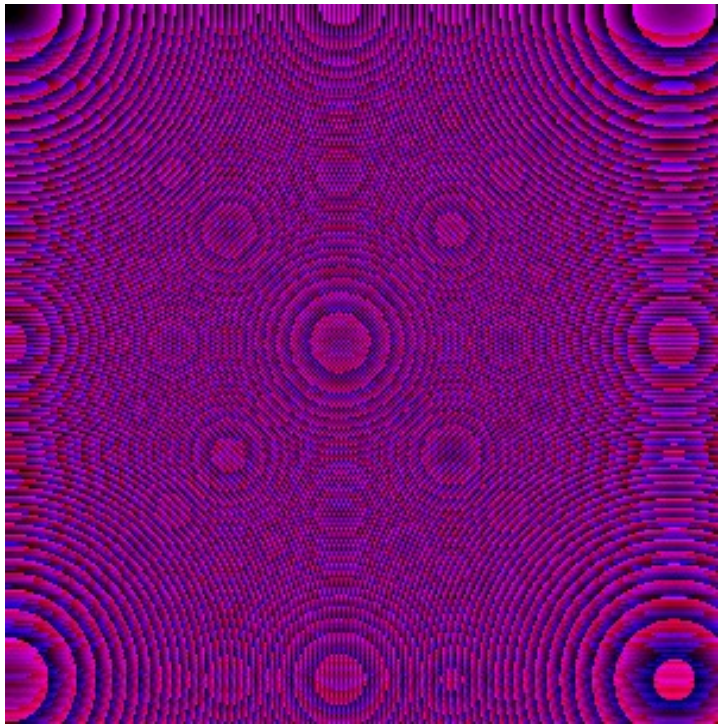
We can consider an algorithm to be a finite sequence of *well defined operations* that *halts in a finite amount of time*.

Given a computational problem, our approach is to design an algorithm and refine/transform it into a computer program.

Given a choice of algorithms to solve a computational problem, we *analyze the algorithms* to find the best design.

Example

Problem: generate an $n \times n$ pixels wallpaper pattern.



Example

Attempt 1:

```
For each pixel “d” do
    decide whether “d” should be “on” or “off”
```

Attempt 2:

```
For each pixel “d” with coordinates (x,y) {
    compute a function c = f(x,y)
    if (c ≤ threshold) then
      plot (d)
}
```

Example

Attempt 3:

```
for i=1 to n
  for j = 1 to n {
      c = (i^2 + j^2) mod 10;
      if c <= 3
          plot (i,j);
  }
```

Program:

```
#include <iostream>
using namespace std;
…
```

Data structure

In the algorithm, we need to handle certain amount of data such as the n^2 pixels, the value of c etc.

We organize the data using *data structures* for efficient retrieval. E.g., the pixels can be organized as a 2-dimensional array so that each pixel can be accessed directly.

Program = data structure + algorithm

A program thus consists of 2 parts:

- *Algorithm*: *the computational procedure* which transforms an input into the desired output as specified by the problem statement.
- *Data Structures*: the *representation*, *organization* and *storage* of information.

Choosing an algorithm

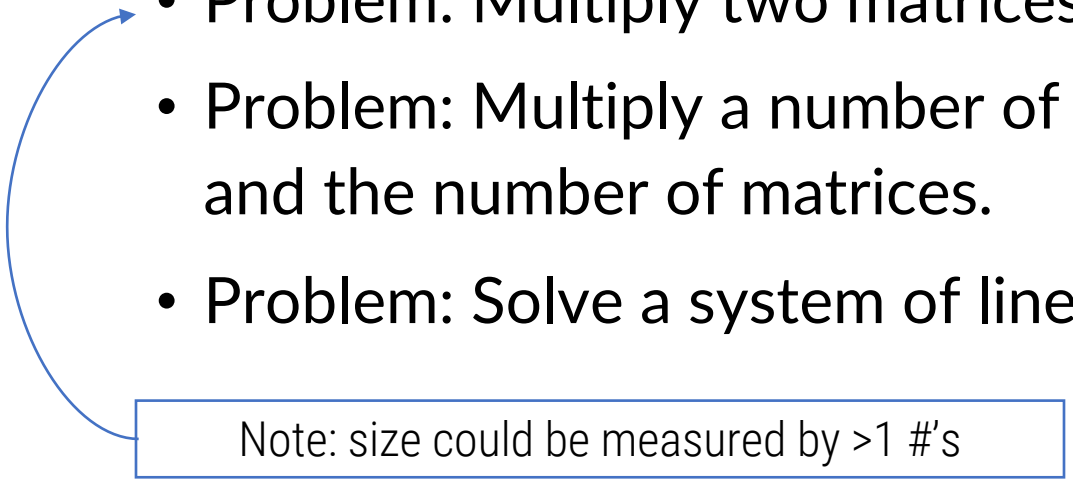
Usually, we get to choose an algorithm to solve a problem. How do we measure which algorithm is better? We need to pick a measure for

- *problem size*, and
- computational *resources* required by the algorithm (expressed in terms of the problem size).

Problem size

The *measure of size* should reflect the *complexity* of the problem instance.

- Problem: Find the sum of a set of *n* integers. Size: *n*.
- Problem: sort a list of *n* numbers. Size: the number of entries in the list (i.e., *n*).
- Problem: Multiply two matrices. Size: the dimensions of the matrices.
- Problem: Multiply a number of matrices. Size: the dimensions of the matrices and the number of matrices.
- Problem: Solve a system of linear equations. Size: the number of unknowns.



Resources

The *measure of resources* required should have at least the following properties:

- it should measure the *resources we care about* (e.g., memory space, running time, ...)
- it should be *quantitative*, so that it is easy to compare two algorithms.
- it should be a good *predictor*, so that it is easy to predict the resource requirement when the algorithm is applied to a large set of problem instances.

Measuring time resource

We may buy space, but we may not buy time!

Time is money! Let's assume the primary performance metric is *running time*. How can we compare the speeds of two algorithms solving the same problem?

Measuring time resource

Method 1: write 2 programs using the different algorithms and time their speeds.

- Advantage: straightforward
- Disadvantage: too much "*noise*": programmer, language, compiler, OS, machine, input.
- Hard to get a *predictor function*.

Measuring time resource

Method 2: discover a function of the problem's size that reflects the work the algorithm must do to solve the problem of that size.

In particular, we can *count some set of operations* that the algorithm performs, then derive a rough estimate of the number of such operations required.

Comparing algorithms

Example: compute the square of a positive integer *n*

Alg 1
```
sum = n * n
```

Alg 2
```
sum = 0;
for i = 1 to n
   sum = sum + n;
```

Alg 3
```
sum = 0;
for i = 1 to n
   for j = 1 to n
      sum++;
```

Comparing algorithms

Assume arithmetic operations take *constant times*,

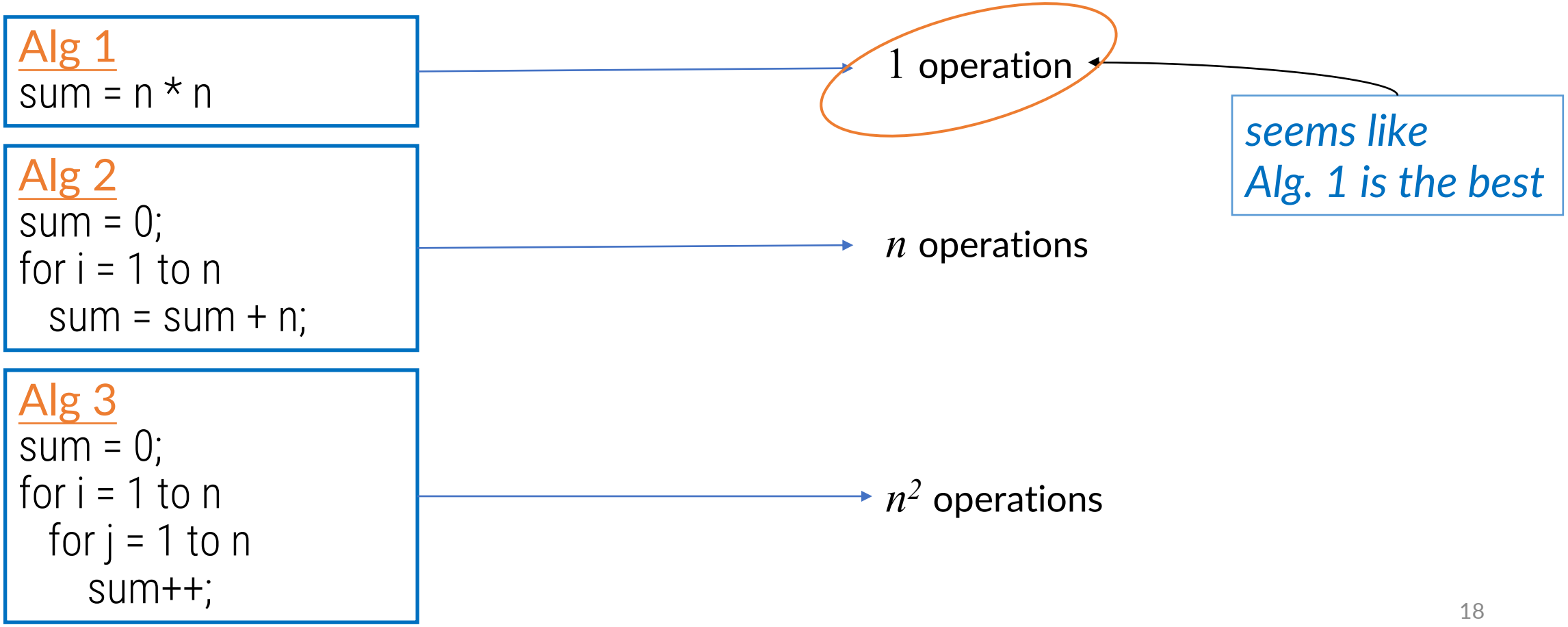
Alg 1
```
sum = n * n
```
→ 1 operation

Alg 2
```
sum = 0;
for i = 1 to n
   sum = sum + n;
```
→ n operations

Alg 3
```
sum = 0;
for i = 1 to n
   for j = 1 to n
      sum++;
```
→ n^2 operations

Comparing algorithms

Assume arithmetic operations take *constant times*,



That’s not fair!

Different operations may require different amounts of time to complete. It is likely that a computer performs an increment much faster than it performs a multiplication.

Comparing algorithms

True. Let t_1, t_2, and t_3 be the time taken by a computer to perform an increment, an addition, and a multiplication, respectively. Times taken by the 3 algorithms are:

- Alg1: t_3
- Alg2: nt_2
- Alg3: n^2t_1

Comparing algorithms

As an example, assume that:

- $t_3 = 20t_2 = 200t_1$

Times taken by the algorithms are:

- Alg1: $200t_1$
- Alg2: $10nt_1$
- Alg3: n^2t_1

Q: under what condition is Alg1 the best? What about Alg2 and Alg3?

Comparing algorithms

We can deduce that:

n	best algorithm
1-10	Alg3
10-20	Alg2
≥ 20	Alg1

So, as long as n is large, Alg1 wins. Only when the problem size is small will the other algorithms be better.

Plan to solve a computational problem

1. We start by *recognizing a problem*. The problem must be of the restricted kind that we can solve on digital computers.
2. We build an *abstract model* -- the set of operations allowed.
3. We *design an algorithm* to solve the problem within the model (i.e., the algorithm should use *only* the operations allowed by the model.)

Plan to solve a computational problem

4. We *analyze the algorithm* to see how much "effort" it takes to solve the problem. This gives us an *upper bound* on the work needed to solve the problem.
5. If possible, we *analyze the problem* to see at least how much work is needed to solve the problem. This gives us a *lower bound* to the problem.

Plan to solve a computational problem

6. If we are not satisfied with the algorithm (in terms of the amount of effort needed), try to find a better alternative.
7. We build *data structures* that allow the algorithm to be implemented efficiently. (Sometimes, steps (3) and (7) are integrated, i.e., the algorithm and data structures are designed together.)

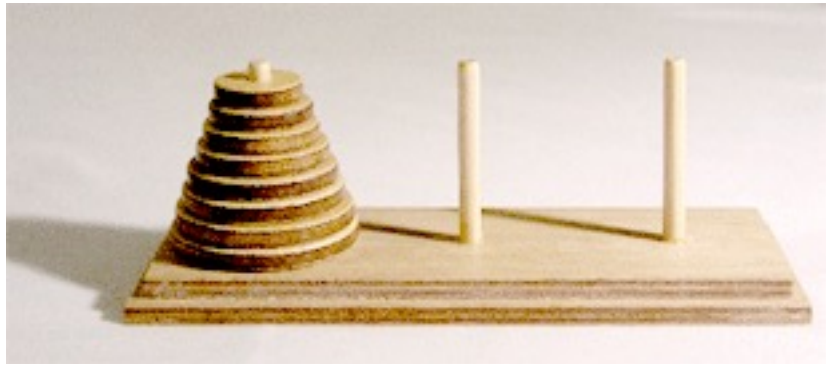
The Towers of Hanoi

In the great temple at Benares … beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed 64 disks of pure gold, the largest disk resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. …

https://ianparberry.com/TowersOfHanoi/index64.html

The Towers of Hanoi

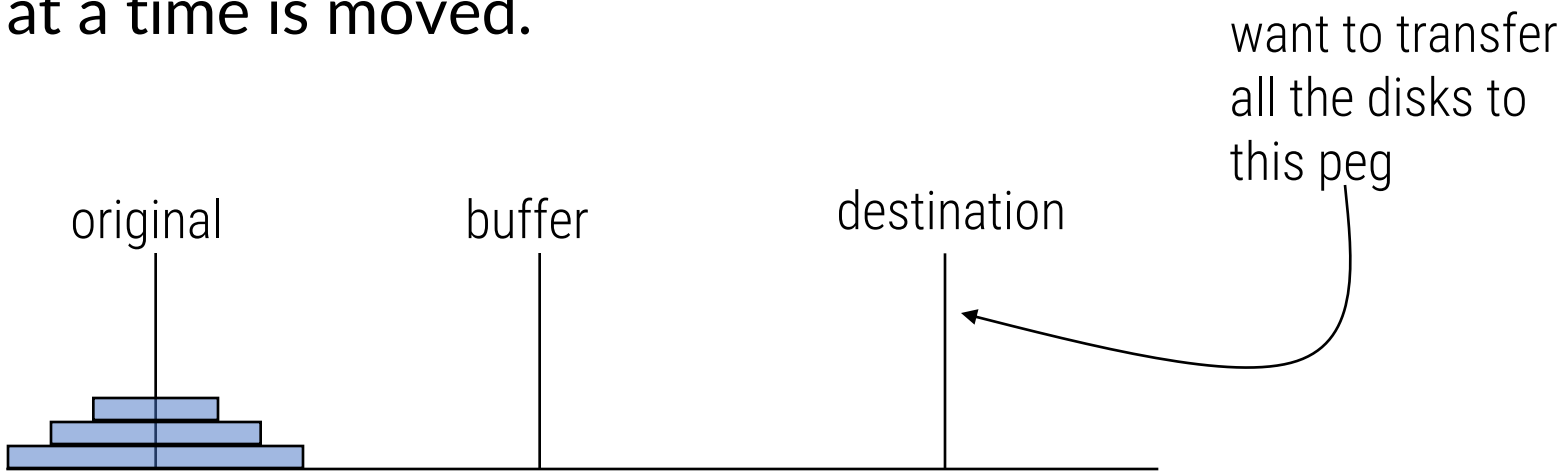
Day and night unceasingly the priests transfer the disks (one at a time) from one diamond needle to the other in such a way that no disk is ever placed on top of a smaller one .... When the 64 disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, then tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.



The Towers of Hanoi

Problem: Given 3 pegs and *n* disks of different sizes placed in order of size on one peg, transfer the disks from the original peg to another peg with the constraints that:

- each disk is on a peg
- no disk is ever on a smaller disk, and
- only one disk at a time is moved.



The Towers of Hanoi

Model: The set of operations = {“move one disk from one peg to another”}

We choose to measure the running time of an algorithm for the problem by the number of “moves” required.

Problem size = *n*.

The Towers of Hanoi

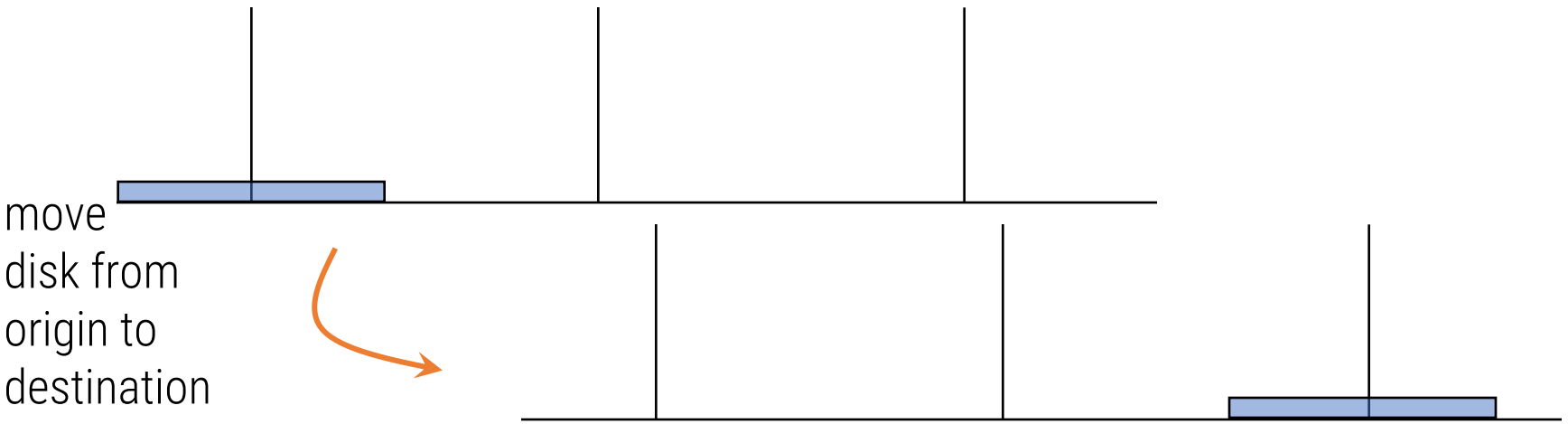
Let us think recursively. Again, we need a *base case* and a way for making *progress*.

The Towers of Hanoi

Let us think recursively. Again, we need a *base case* and a way for making *progress*.

Base case:

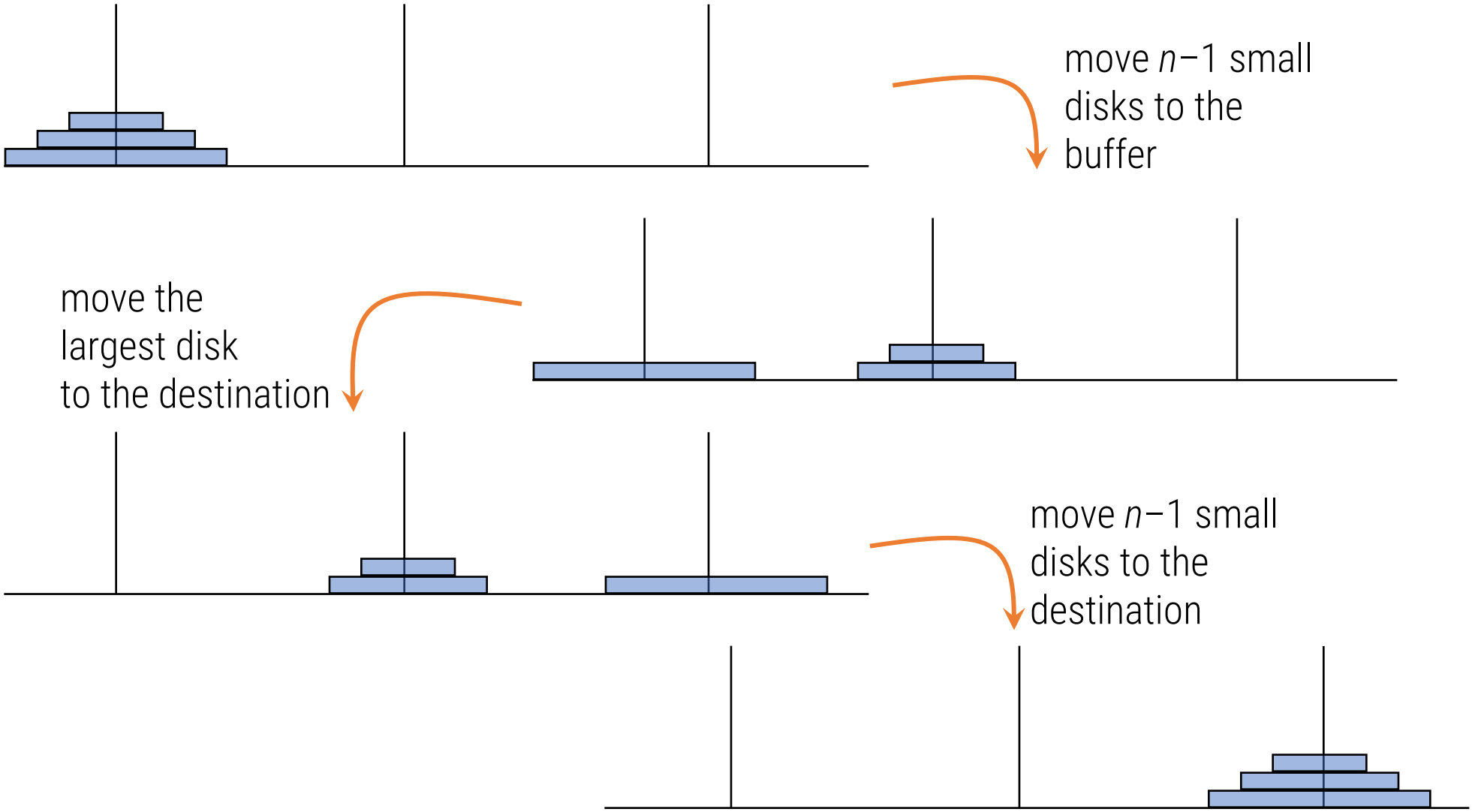
- if there is only 1 disk (i.e., $n = 1$)



The Towers of Hanoi

If I know how to move $n-1$ disks without violating the rules, can I come up with a strategy for moving n disks?

The Towers of Hanoi



The Towers of Hanoi

Algorithm:

```
Hanoi (A, B, C, n) {
  /* to move n disks from peg A to peg C using peg B as an intermediate peg. */

  if (n=1)
     move A's top disk to C;
  else {
     Hanoi(A, C, B, n-1);
     move A's top disk to C;
     Hanoi(B, A, C, n-1);
} }
```

The Towers of Hanoi

Analysis:

- Can we *prove* that our algorithm is correct?
- How many moves does our algorithm take (*upper bound*)?
- What is the least number of moves required to solve the problem (*lower bound*)?

The Towers of Hanoi

Analysis:

- Note: the lower bound analysis is done on the problem and not on the algorithm; *a lower bound must apply to all algorithms* allowed within the model that solve the problem, not just the one we designed.
- An *upper bound* is a measure of *how bad* a particular algorithm can be;
- A *lower bound* is a measure of *how hard* a problem is.

The Towers of Hanoi

Analysis: How many moves does our algorithm take?

Let $f(n)$ be the number of moves required by our algorithm to move *n* disks from a peg to another. We know that:

$$f(n) = \begin{cases} 1, & n = 1 \\ 2f(n-1) + 1, & n > 1 \end{cases}$$

The Towers of Hanoi

Solving the recurrence gives

$$f(n) = 2^n - 1$$

Our algorithm thus takes $2^n - 1$ moves and this is an *upper bound* of the Tower of Hanoi problem with n disks.

The Towers of Hanoi

What about a lower bound?

0th attempt: a stupid lower bound: 0
Ben: "It takes at least 0 moves to transfer the disks!"

1st attempt: an obvious lower bound: n
Jerry: "To move n disks from one peg to another, one at a time, you need at least n moves!"

2nd attempt: a better lower bound: $2n-1$
Slump: "Each disk except the largest one must be moved at least twice, and the largest disk must be moved at least once."

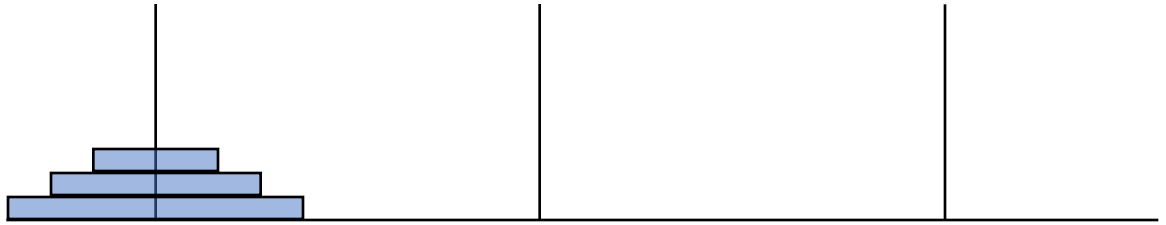
The Towers of Hanoi

3rd attempt: the best lower bound: $2^n - 1$

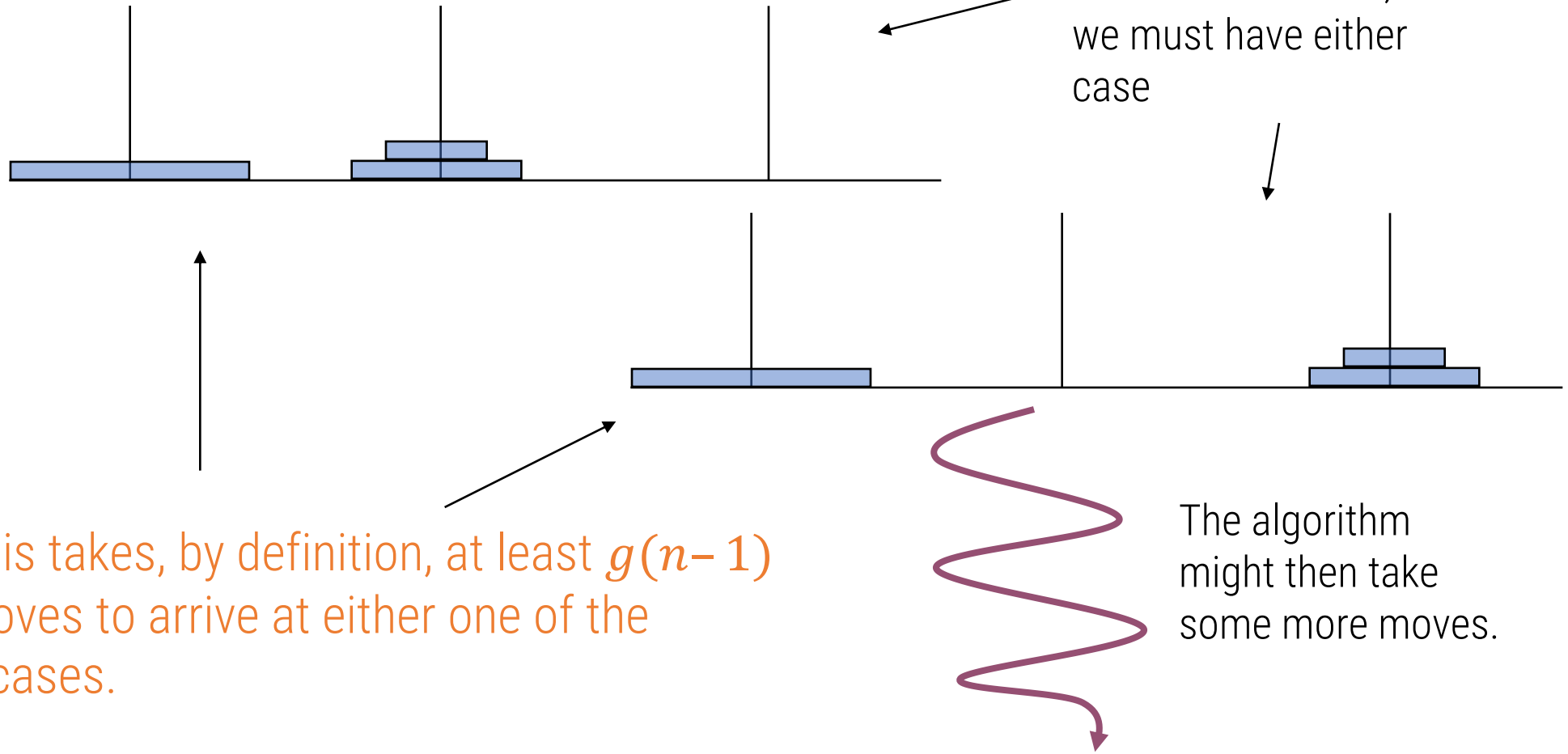
Let

$g(n)$ = number of moves that are necessary to solve the problem with n disks

Don't confuse this with $f(n)$



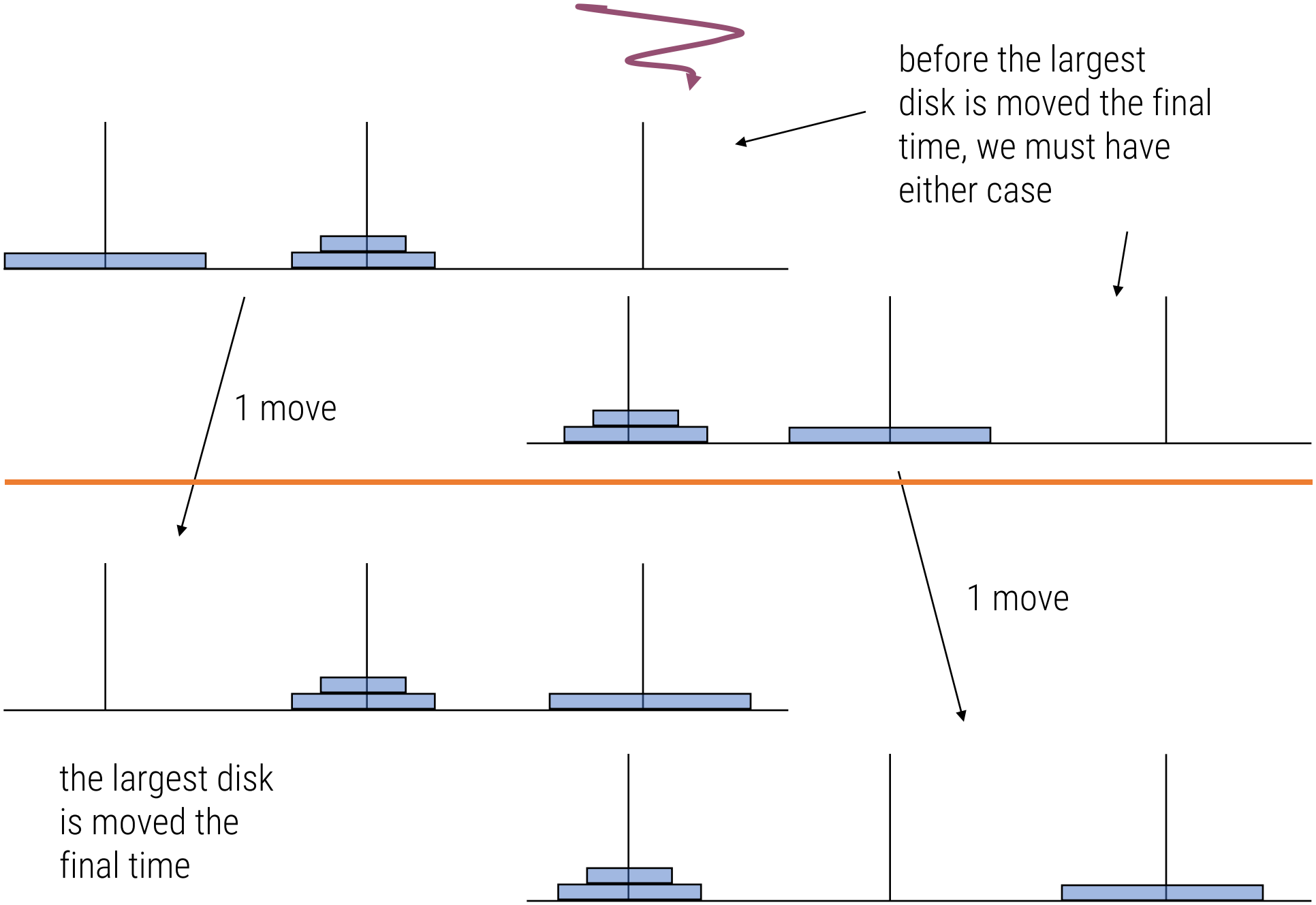
original configuration

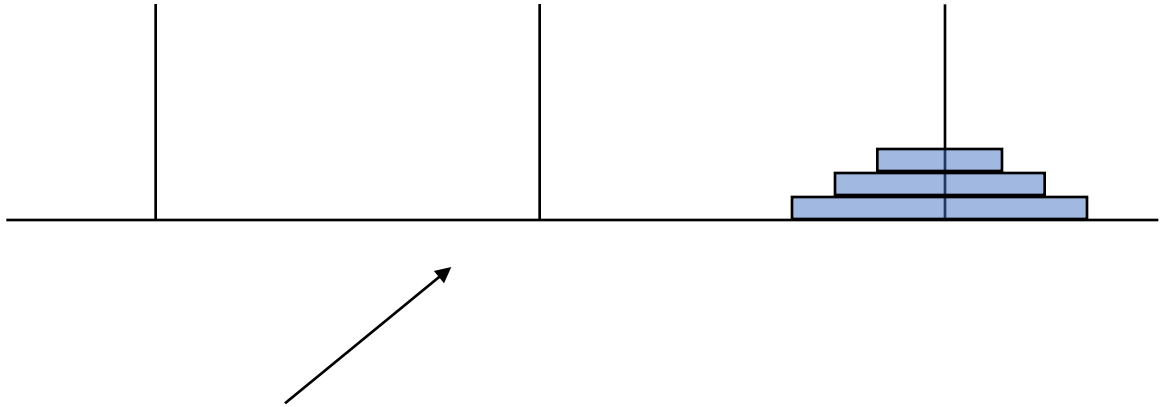


before the largest disk can be moved, we must have either case

This takes, by definition, at least $g(n-1)$ moves to arrive at either one of the 2 cases.

The algorithm might then take some more moves.


before the largest disk is moved the final time, we must have either case
1 move
1 move
the largest disk is moved the final time



It takes at least $g(n-1)$ moves to finish.

final configuration, i.e., all $n-1$ disks are moved from the same peg to the destination peg

The Towers of Hanoi

So, we have

$$g(n) \geq \begin{cases} 1, & n = 1 \\ 2g(n-1) + 1, & n > 1 \end{cases}$$

Solving the recurrence (inequality), we get

$$g(n) \geq 2^n - 1$$

The Towers of Hanoi

Our algorithm is *optimal* in the sense that it uses the minimum number of moves that is necessary to solve the problem.

The Towers of Hanoi

Shall we be afraid of the end of the world?

If we have a super monk who can transfer one disk in 1 second nonstop, the end of the world will come in
$2^{64} - 1$ seconds
= 18,446,744,073,709,551,615 seconds
= 385 billion years!!!

(The universe is about 15 billion years old.)
(An average species lives 4 million years before extinction.)

The Towers of Hanoi

Shall we be afraid of the end of the world?

If we have a super mo disk in 1 second nonstop, the end of th

$2^{64} - 1$ seconds
= 18,446,744,073,709,551,615 seconds
= 385 billion years!!!

Our solution takes *exponential* time w.r.t. the problem size!

(The universe is about 15 billion years old.)
(An average species lives 4 million years before extinction.)

Difficulties in algorithm analysis

The Tower of Hanoi problem is *easy* to analyze in the sense that

- for every problem size, there is only 1 problem instance.
- it is clear that “move” is the only operation in the model that we care about.

In many cases, we do not have this luxury.

Searching

Given an array of $n \geq 1$ integers and a "target" integer x, determine if x is in the array.

Algorithm: let A[1..*n*] be the array

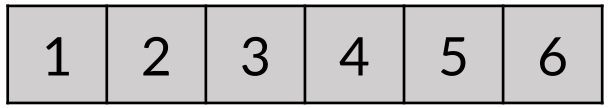
```
array_search(A,x,n) {
i = 1;
while (x != A[i])
   if (i == n)  return ("not found");
   else  i++;
return ("found"); }
```

Inputs

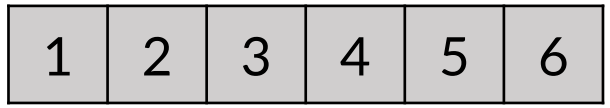
The first problem: the running time of the algorithm depends on the input.

Q: how many comparisons are needed for the following problem instances?

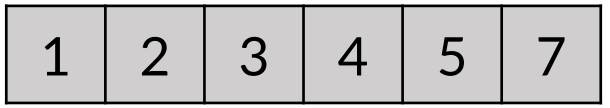
find ‘1’



find ‘6’



find ‘6’



Inputs

Solution: we usually find the "*worst cost*," that is, the max. number of times the algorithm performs the chosen operation(s).

Since the algorithm is a proof that we can solve the problem using at most that number of operations, even in the worst case, that gives an "*upper bound*" of the problem's worst cost as a function of the input size.

Note: the number of element-element comparison required is *n* in the worst case.

Best, worst, and average

A = algorithm;

I_n = set of all possible inputs to A, each of size n.

$f_A(I)$ = cost of A given $I \in I_n$ as the input, then

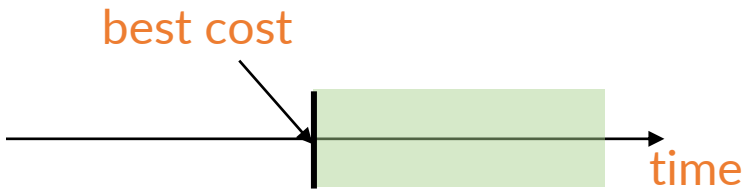
$$\text{WorstCost}(A) = \max_{I \in I_n} f_A(I)$$

$$\text{BestCost}(A) = \min_{I \in I_n} f_A(I)$$

Furthermore, if we know, $p(I) = prob[I \text{ occurs}]$ then

$$\text{AvgCost}(A) = \sum_{I \in I_n} p(I) \times f_A(I)$$

Choosing a cost



In many cases, the best cost isn't very telling. For example, the best cost of the search algorithm is 1 (comparison), which doesn't really reflect how hard the problem is.

The average cost is trickier to find. For one thing, $p(I)$ is usually unknown. Even if $p(I)$ is known, calculating the avgerage cost is usually harder than finding the worst cost.

So, given an algorithm, it make sense to first find its worst cost; then, if the problem is important enough, its average cost.

Operations

The 2nd problem: for many problems and models, there may be many different types of operations that are relevant in computing the running time of the algorithms.

Which operations shall we take into account in assessing the running time of an algorithm?

Matrix multiplication

Example. Matrix multiplication

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

```
Alg1 {  /* given a,b,c,d,e,f,g,h. */
   r = a*e + b*f;      s = a*g + b*h;
   t = c*e + d*f;      u = c*g + d*h;
   output (r,s,t,u);
}
```

Algorithm 1 takes 8 multiplications and 4 additions.

Matrix multiplication

```
Alg2{ /* given a,b,c,d,e,f,g,h */
    p1 = a * (g-h);      p2 = (a+b) * h;
    p3 = (c+d) * e;      p4 = d * (f-e);
    p5 = (a+d) * (e+h);  p6 = (b-d) * (f+h);
    p7 = (a-c) * (e+g)
    u = p5 + p1 - p3 - p7;
    r = p5 + p4 - p2 + p6;
    s = p1 + p2;         t = p3 + p4;
}
```

Algorithm 2 takes 7 multiplications and 18 additions.

Operations

Question: if we consider multiplication and addition having different costs, which algorithm is better?

The answer depends on which machine/computer you use. In other words, it would be difficult to compare algorithms without referring to a particular computer (not desirable).

A compromise

We shall relax our way of comparing algorithms. Instead of counting the exact number of execution for each type of operation, we are interested only in the *growth rate* of the *total number of operations* as a function of the input size.

An illustration

Example: wallpaper algorithm

Statement		Cost
(1)	for i = 1 to n	c1
(2)	for j = 1 to n {	c2
(3)	c = (i^2+j^2) mod 10	c3
(4)	if (c <= 3)	c4
(5)	plot (i,j); }	c5

The exact values of these costs depend on the machine.

An illustration

There are 5 statements and 6 types of operations: assignment, addition, squaring, mod, plot, and comparison.

Statement		Number of executions
(1)	for i = 1 to n	n
(2)	for j = 1 to n {	n^2
(3)	c = (i^2+j^2) mod 10	n^2
(4)	if (c <= 3)	n^2
(5)	plot (i,j); }	? $(0 \leq ? \leq n^2)$

Execution time

Let total execution time = $T(n)$

$$\begin{aligned} T(n) &= (n+1)c_1 + n(n+1)c_2 + n^2c_3 + n^2c_4 + ?(c_5) \\ &\leq (n+1)c_1 + n(n+1)c_2 + n^2c_3 + n^2c_4 + n^2(c_5) \\ &= (c_2 + c_3 + c_4 + c_5)n^2 + (c_1 + c_2)n + c_1 \\ &\leq (2c_1 + 2c_2 + c_3 + c_4 + c_5)n^2 \\ &= C'n^2 \end{aligned}$$

Also,

$$\begin{aligned} T(n) &\geq (c_2 + c_3 + c_4)n^2 + (c_1 + c_2)n + c_1 \\ &\geq (c_2 + c_3 + c_4)n^2 \\ &= C''n^2 \end{aligned}$$

Execution time

Since $T(n)$ is bounded (above and below) by two functions that are proportional to n^2, it *grows like* the function n^2 and we say that $T(n)$ is $\Theta(n^2)$.

This is called the *order of growth*, or *complexity*.

Complexity

When comparing two algorithms, we only compare their orders of growth, or complexity. (The smaller the better!)

We aren't all that interested in the exact running time the algorithm takes. In many cases, what makes one algorithm more desirable than another is its growth rate relative to the other algorithm's growth rate.

Complexity

For example, an algorithm with a running time of $an^2 + bn + c$ for some constants a, b, c is said to have a growth rate of $\Theta(n^2)$. And that it is *more efficient* than an algorithm of order $\Theta(n^3)$ because n^3 grows much faster than n^2 does.

Complexity

Using growth rate allows us to not worry about the different types of operations having different costs. As long as we are sure that each operation takes constant amount of time to finish, we only need to count how many operations (in terms of an order, e.g., n^2) are needed.

Question: What is the complexity of the Hanoi algorithm?

Growth of Functions

We study the asymptotic efficiency of algorithms: how the running time of an algorithm increases with the size of the input as the size of the input grows. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small input.

Big Oh

O notation

For a given function $g(n)$

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$

We use O notation to give an upper bound on a function, to within a constant factor. We say $f(n)$ grows no faster than $g(n)$.

Ex. $f(n) = 2n^2 + 4n - 1$ is $O(n^2)$.

Big Omega

Ω notation

Ω gives us an asymptotic lower bound.

For a function $g(n)$

$$\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

We say $f(n)$ grows no slower than $g(n)$.

Big Theta

Θ notation

For a given function $g(n)$

$$\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0 \text{ s.t.} \\ \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

If $f(n) \in \Theta(g(n))$ we say that $g(n)$ is an asymptotically tight bound for $f(n)$.

Ex. $f(n) = 2n^2 + 4n - 1$ is $\Theta(n^2)$.

Abusing notation

Formally, $\Theta(g(n))$ is a *set of functions* that satisfy a certain condition. When we say that a function $f(n)$ is $\Theta(g(n))$ we really mean $f(n) \in \Theta(g(n))$, i.e., $f(n)$ satisfies certain condition.

Very often, however, we use $\Theta(g(n))$ informally to mean *"some function that is in the set $\Theta(g(n))$ ".*

For example, when we write $f(n) = \Theta(g(n))$, informally we mean

"$f(n)$ is equal to some function that is in the set $\Theta(g(n))$ ".

Abusing notation

As another example, when we write

- $4n^3 + \Theta(n^2) = \Theta(n^3)$

Informally, we mean

- *"$4n^3$ plus some function of order n^2 is equal to some function of order n^3."*

Semantically, $4n^3 + \Theta(n^2)$ refers to some function whose leading terms is $4n^3$ plus something of the order of $\Theta(n^2)$ (which we don't really care about).

Abusing notation

Sometimes we use the Θ-notation in formula to represent some function of lower order terms.

- E.g., $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

This saves us from inessential details.

Q: What is $\Theta(1)$?

Observation: with Θ, we can throw away lower order terms and the leading term's coefficient in mentioning a function.

Using O in algorithm analysis

Note that

- If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then

  $f_3(n) = f_1(n) + f_2(n)$ is also $O(g(n))$.

Using the O-notation, we can describe the worst case running time of an algorithm merely by inspecting its structure.

Using O in algorithm analysis

Example: wallpaper algorithm

Statement		Cost
(1)	for i = 1 to n	c1
(2)	for j = 1 to n {	c2
(3)	c = (i^2+j^2) mod 10	c3
(4)	if (c <= 3)	c4
(5)	plot (i,j); }	c5

Little o

o notation

We use o notation to denote an upper bound that is not asymptotically tight. E.g., the bound $2n = O(n^2)$ is not "tight".

o denotes a "proper" upper bound. Formally,

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq f(n) < cg(n)\}$$

E.g., $2n = o(n^2)$ but $2n^2 \neq o(n^2)$

We say that $f(n)$ grows slower than $g(n)$.

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Little ω

ω notation

We use ω to denote a “proper lower bound”.

Formally,

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq cg(n) < f(n)\}$$

E.g., $2n^2 = \omega(n)$ but $2n^2 \neq \omega(n^2)$

Little ω

We say that $f(n)$ grows faster than $g(n)$.

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

$$f(n) = \omega(g(n)) \Leftrightarrow g(n) = o(f(n))$$

Properties of the asymptotic notations:

Transitivity, reflexivity, symmetry, transpose

Associate the notations with the comparison operators:

- $O \Rightarrow \leq$
- $\Theta \Rightarrow =$
- $\Omega \Rightarrow \geq$
- $o \Rightarrow <$
- $\omega \Rightarrow >$

Manipulating O

Some simple rules:

$$n^m = O\left(n^{m'}\right) \quad \text{for } m \leq m';$$

$$O(f(n)) + O(g(n)) = O(f(n) + g(n));$$

$$f(n) = O(f(n));$$

$$c \cdot O(f(n)) = O(f(n)) \quad \text{where } c \text{ is a constant};$$

$$O(f(n))O(g(n)) = O(f(n)g(n));$$

$$O(f(n)g(n)) = f(n)O(g(n)).$$

Calculating running time

Some simple rules:

- Rule 1: *Loops*:
  The complexity of a loop is at most the complexity of the statement inside the loop (including tests) times the number of iterations.
- Rule 2: *Consecutive statements*:
  The complexity of 2 consecutive program segments (S1 and S2) = max{complexity of S1, complexity of S2}.

Calculating running time

- Rule 3: *If-then-else*:

  the complexity of

  ```
  if (cond) S1; else S2
  ```

  = max {complexity of (S1 + cond),

  complexity of (S2 + cond)}.

- Rule 4: *Function call*:

  The complexity of a function should be analyzed first before computing the complexity of the program fragment containing the call.

Example

```
array_search(A,x,n) {
i = 1;
while (x != A[i])
   if (i == n)  return (“not found”);
   else  i++;
return (“found”); }

test() {
   for (j=1 to m)
      array_search(A,j,n); }
```

Q: what is the running time of test() in O-notation?

Some typical growth rates

$O(1)$, i.e., constant time.

$O(\log n)$, i.e., logarithmic complexity. Commonly occurs in programs that solve a big problem by transformation into a series of smaller problems, cutting the problem size by some constant fraction at each step.

$O(n)$, i.e., linear complexity. Typical for algorithms that do constant amount of work on each input element.

Some typical growth rates

$O(n \log n)$. Arises when algorithms solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions.

$O(n^2)$, i.e., quadratic. Typically arises in algorithms that process all pairs of data items.

$O(n^3)$, i.e., cubic.

$O(2^n)$, i.e., exponential. Algorithms with exponential complexity are not likely to be practically useful for large input.

Hard problems

Generally, we say that a problem has a *tractable* solution if we have an algorithm to solve it taking time that is at worst a polynomial of the size of the problem, i.e., $O(n^c)$ for some constant c .

The current best solutions to thousand of important problems (e.g., the subset sum problem) are "exponential" in the size of the problems.

E.g., $O(r^n)$ for some $r > 1$. We call those *computationally hard problems*.

The Fibonacci numbers

Suppose you have a pair of rabbits and suppose every month each pair bears a new pair that from the second month on becomes productive. How many pairs of rabbits will you have in a year?

--- Leonardo Pisano Bigollo (Fibonacci) 1202

The Fibonacci numbers

Month (n)	Productive $g(n)$	not productive $h(n)$	Total $f(n)$
1	0	1	1
2	1	0	1
3	1	1	2
4	2	1	3
5	3	2	5
6	5	3	8

A recurrence equation

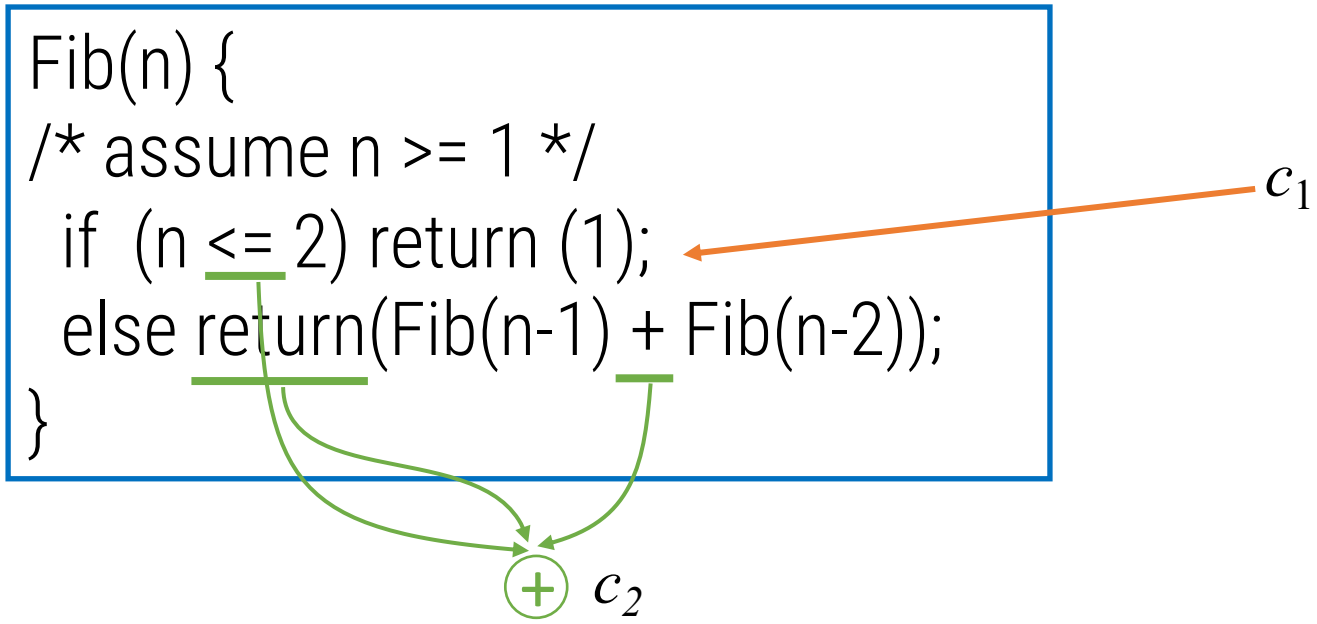
Let $f(n)$ be the number of pairs of rabbits you have in month $n \geq 1$.

$$f(1) = f(2) = 1;$$

$$f(3) = 1 + 1 = 2;$$

$$f(n) = f(n-1) + f(n-2), \quad n > 2.$$

A simple recursive algorithm



Let $T(n)$ be the execution time of Fib(n).

$$T(1) = T(2) = c_1$$
$$T(n) = T(n-1) + T(n-2) + c_2$$
$$\frac{T(n)}{T(n-2)} = \frac{T(n-1)}{T(n-2)} + \left(1 + \frac{c_2}{T(n-2)}\right) \approx \frac{T(n-1)}{T(n-2)} + 1 \text{ (for very large } n)$$
Suppose for large n, $\frac{T(n)}{T(n-1)} \to x$
$$\frac{T(n)}{T(n-1)} \times \frac{T(n-1)}{T(n-2)} = \frac{T(n-1)}{T(n-2)} + 1$$
$$\Rightarrow x^2 - x - 1 = 0$$
$$\Rightarrow x = \frac{1+\sqrt{5}}{2} = \phi$$
$$\therefore \frac{T(n)}{T(n-1)} = \phi \text{ (for large } n\text{, say for } n \geq n_0)$$
$$\forall n \geq n_0,\ T(n) = \phi^{(n-n_0)} T(n_0) = \left(\frac{T(n_0)}{\phi^{n_0}}\right)\phi^n$$
$T(n)$ is exponential!

Analysis

In fact, we could show that, for $n > 1$

$$T(n) = \left(\frac{\phi^{n-1} - \hat{\phi}^{n-1}}{\sqrt{5}} \right)(c_1 + c_2) - c_2,$$

$$\text{where, } \hat{\phi} = -1/\phi.$$

Recursion is not necessarily good

Recursions, if applied indiscriminately, could be very inefficient.

Q: how many times was Fib(2) computed in the previous algorithm?

An iterative algorithm

A more efficient algorithm:

```
Fib(n) {
  previous = present = 1;

  for (i=3; i<=n; i++) {
    past = previous;
    previous = present;
    present = previous + past;
  }
  return (present);}
```

Complexity = $\Theta(n)$

The *Golden Ratio*

In 1765, Leonhard Euler discovered the formula:

$f(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ where $\phi = \frac{1+\sqrt{5}}{2}$, and $\hat{\phi} = -1/\phi$

and $\phi \approx 1.61803$ is called the golden ratio.

It is believed to have magical power and is important in many parts of mathematics as well as in the art world.

E.g, The ratio of one's height to the height of one's navel is approx. 1.618. (doesn't apply to Asian, though)