FUNDAMENTALS

ANALYSIS OF ALGORITHMS

Ruben Acuña

Spring 2018

2

CHARACTERIZING ALGORITHMS

UNDERSTANDING PROGRAMS

- Suppose for a moment, that you have managed to obtain a internship position at a coveted company.
- Unfortunately for you, the company brought in several interns but only has a single developer position to offer at the end of the semester.
- As part of your work, you and the rest of the interns are asked to write an algorithm to solve a problem.
- At the end, you need to present your solution and, ideally, argue that it is better than the solutions produced by other people, thus securing the fulltime position!

- But how?

UNDERSTANDING PROGRAMS

- Once we have an implementation, we simply have a way to solve the problem. We do not explicitly know how (e.g., fast execution, minimal memory) it acts.
- There is no way to argue one implementation versus another!
- What we need is some way to characterize the behavior of a program or method – we need to study *analysis of algorithms*.
  - Characterization might be as simple as putting programs into "buckets" where each bucket is a family of programs that act roughly the same.
  - If there is an order on the "buckets", then that order applies to the implementations that fall into them.
- A generic way would best, so it can model all implementations in all languages.

PERSPECTIVES ON ALGORITHM ANALYSIS

Analyzing algorithms is useful from both theoretical and applied perspectives.

A theorist would be interested in understanding:

- What is the "time complexity" or "space complexity" of a problem?
  - Is this the fastest possible algorithm?
- Can the problem be solved? Efficiently or otherwise.

PERSPECTIVES ON ALGORITHM ANALYSIS

While designing a solution to a problem, we often encounter multiple solutions, and will need to select one.

A developer would be interested in applying:

- Have a way to communicate properties of an algorithm at a high level.
- Have a way to rank algorithms for efficiency.
- Can guarantee performance. (To who?)

Ultimate question, can my program solve the size of problems I need it to solve?

REVIEWING TERMINOLOGY

Let’s review three basic ideas in algorithm analysis that you should have heard before: *n*, *growth function*, and *O(n) notation*.

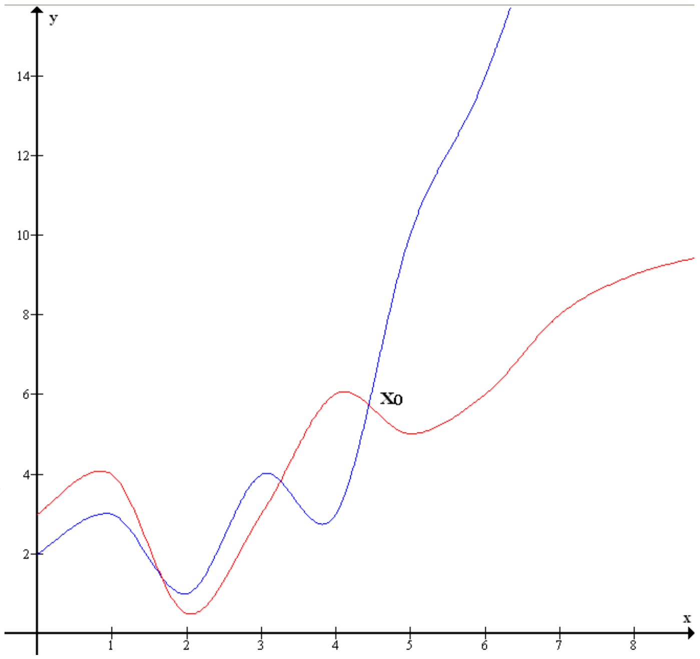
- We use *n* as a variable presenting problem size.
  - Typically algorithms are parameterized on some kind of input which is collection whose size impacts how long it takes.
  - Algorithms may have many parameters but if the parameter is always “the same size”, we don’t need to worry about it. (Why?)
- The problem size (n), can be many things:
  - size of an array (search)
  - value of integer (factorial)
  - length of a string (reverse)

Image from Wikipedia Commons.

REVIEWING TERMINOLOGY

growth function: computes the number of operations* for an implementation to execute on a problem of a size *n*. May written as *f(n)*, *T(n)*, etc.

- *f(n)=1*. A method that always executions one operation (e.g., an assignment).
- *f(n)=1+n*. A method that performs a fixed operation and then a operation that depends on the problem size (e.g., a loop)



* we should plan to be more exact later. We need to define *cost*.

Image from Wikipedia Commons.

REVIEWING TERMINOLOGY

Big-Oh: a term like $O(1)$, $O(n)$, or $O(n^2)$ that represents a function that can be as large as a constant factor of something.

- For example, $2n + 15 = O(n)$ indicates the fact that the LHS growth function has an upper bound (i.e., a value always larger than it) that looks like $c * n$.
- Will be used to focus on the long term behavior of a function.

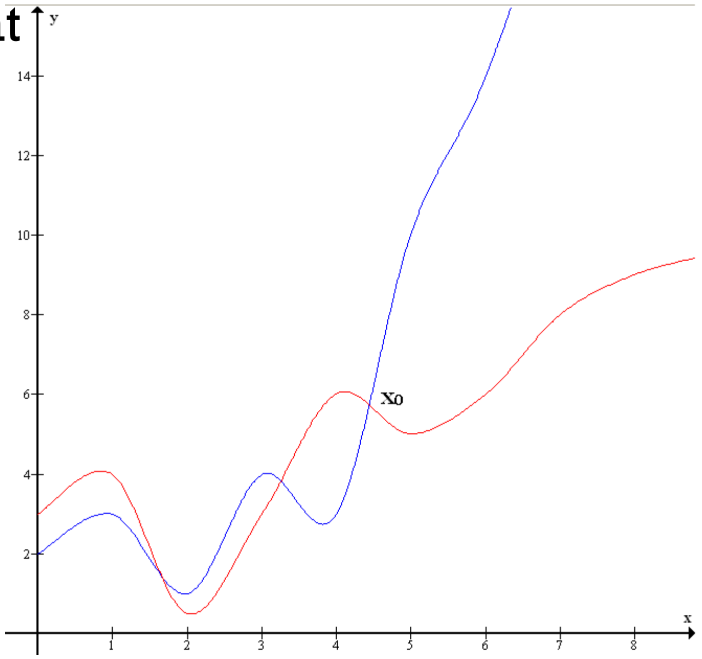


Image from Wikipedia Commons.

ANALYZING PROBLEMS THEMSELVES

- Consider: different solutions to the *same* problem may act *differently*.
- Consider the two methods at right, which can be used to check if a sorted array contains an element.
  - How are they the same?
  - How are they different?

```
boolean searchLinear(int target,
                     int[] pool) {
  for(int i = 0; i < pool.length; i++)
    if(pool[i] == target)
      return true;

  return false;
}

boolean searchBinary(int target,
                     int[] pool) {
    int lo = 0;
    int hi = pool.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (target < pool[mid])
            hi = mid - 1;
        else if (target > pool[mid])
            lo = mid + 1;
        else
            return true;
    }
    return false;
```

ANALYZING PROBLEMS THEMSELVES

- The first is *O(n)*, the second is *O(log(n))*.
- These are correct characterizations of the respective pieces of code…
- What of the problem itself?

```
boolean searchLinear(int target,
                     int[] pool) {
  for(int i = 0; i < pool.length; i++)
    if(pool[i] == target)
      return true;

  return false;
}

boolean searchBinary(int target,
                     int[] pool) {
    int lo = 0;
    int hi = pool.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (target < pool[mid])
            hi = mid - 1;
        else if (target > pool[mid])
            lo = mid + 1;
        else
            return true;
    }
    return false;
```

ANALYSIS APPROACHES

- In the past, you have likely been asked to derive a growth function directly from a piece of code:

```
int pool = new int[n];
for(int i = 0; i < pool.length; i++)
    pool[i] = random.nextInt(100);
```

- $f(n) = 1 + ([1] + [n+1] + [n] + [n]) = 3 + 3n$

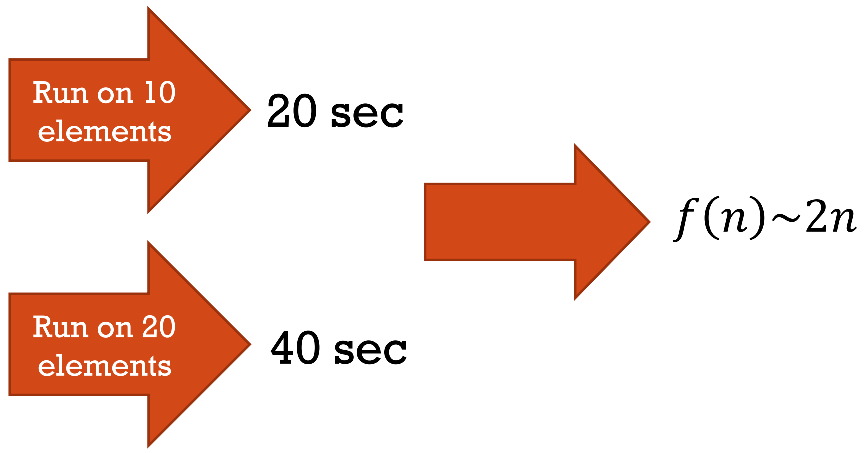
  where *n* is the problem size (size of pool) and *f(n)* is the number of operations.

- This is an *analytical approach* – from the code, there are specific rules to determine the algorithm's growth function.
- The goal is to produce an *exact* solution.

ANALYSIS APPROACHES

- Problem: using that level of detail ends up being arduous!
- We can also try using observation:

```
int pool = new int[n];
for(int i = 0; i < pool.length; i++)
    pool[i] = random.nextInt(100);
```



- This is an *empirical* approach - we treat the code as a black box, feed it inputs, and from observation of how it behaves, approximate its growth function.
- The goal is to produce an *approximate* solution.

ANALYSIS APPROACHES

- Both analytical and empirical approaches are useful and we will be covering both.
  - What are some advantages of an analytic approach?
  - What are some disadvantages of an analytic approach?
  - What are some advantages of an empirical approach?
  - What are some disadvantages of an empirical approach?

EMPIRICAL ANALYSIS

THE EMPIRICAL PROCESS

1) Benchmark the runtime of the application.

2) Construct a mathematical model (e.g., regression).

Then of course:

3) Use model to predict results.

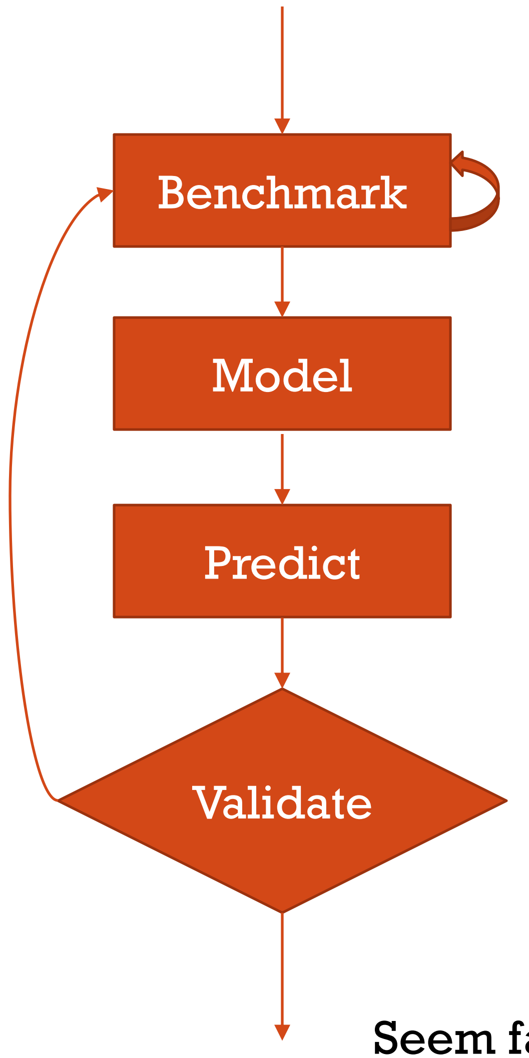
4) Validate model against results.

Advantages:

- Hides the internals of the implementation.
- Does not require source code to analyze.
- Fast.

Disadvantages:

- Inexact.
- Very much "black box".
- Dependent on evaluation environment.



Seem familiar?

THE THREESUM PROGRAM

- Let's start by reviewing a method to analyze.
- A brute force technique to check for triples that sum to zero in an input list.

```
//Sedgewick and Wayne
public static int threeSum(int[] a) {
   int N = a.length;
   int count = 0;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
         for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
               count++;
    return count;
}
```

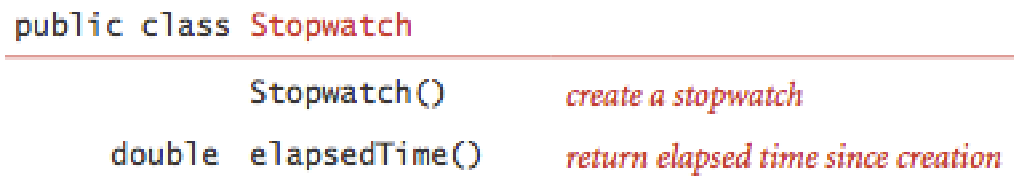
```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4
```

a[i]	a[j]	a[k]	Sum
30	-40	10	0
30	-20	-10	0
-40	40	0	0
-10	0	10	0

Output and table from Sedgewick and Wayne.

BENCHMARKING A PROGRAM

- A simple way to time execution: object creation records time, method returns the difference.
- Be aware that there may be some accuracy caveats listed in the documentation.



public class Stopwatch

	Stopwatch()	*create a stopwatch*
double	elapsedTime()	*return elapsed time since creation*

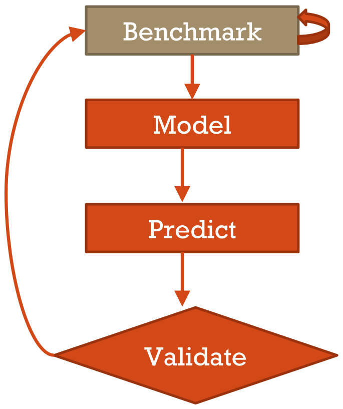
```java
//Sedgewick and Wayne
public class Stopwatch {
  private final long start;

  public Stopwatch() {
    start = System.currentTimeMillis();
  }

  public double elapsedTime() {
    long now = System.currentTimeMillis();
    return (now - start) / 1000.0;
  }
}
```

- Code will look like:

```java
Stopwatch stopwatch = new Stopwatch();
//Do something.
double time = stopwatch.elapsedTime();
```

Table from Sedgewick and Wayne.

BENCHMARKING THREESUM



- To use Stopwatch, simply drop it into the code and run it.

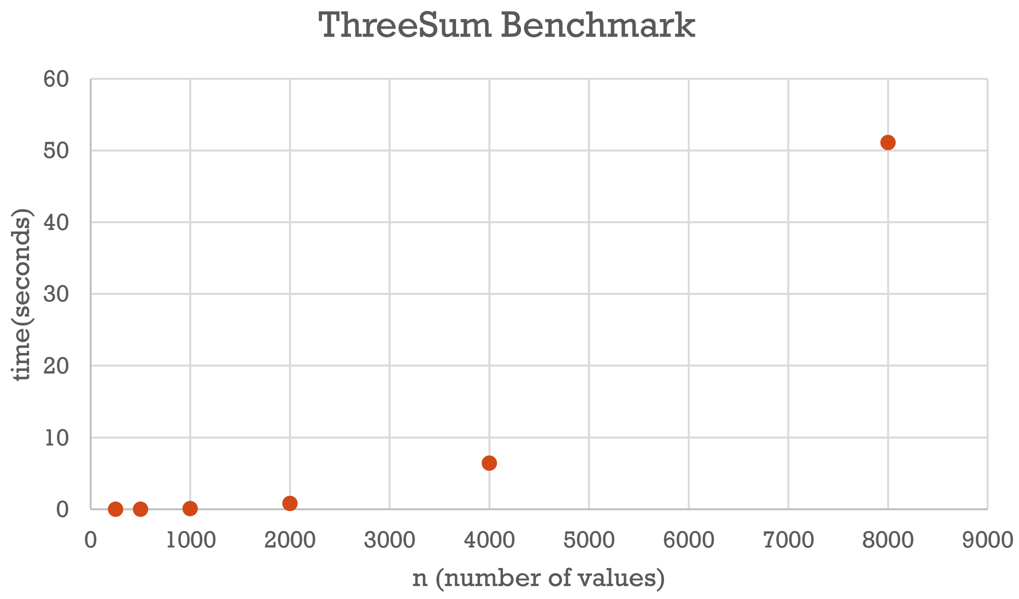
```
 //Sedgewick and Wayne
 public static void main(String[] args) {
    int[] a = {23, 32, 5, 103, 12, 10, …};
    Stopwatch stopwatch = new Stopwatch();
    System.out.println(ThreeSum.count(a));
    double time = stopwatch.elapsedTime();
    System.out.println("elapsed " + time);
 }
```

```
run:
0
elapsed time 0.001
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Now, we can think about running a program multiple times, knowing the input size, and using the results to create a growth function.

BENCHMARKING THREESUM

N	Time(seconds)
250	0
500	0
1000	.1
2000	.8
4000	6.4
8000	51.1
16000	?

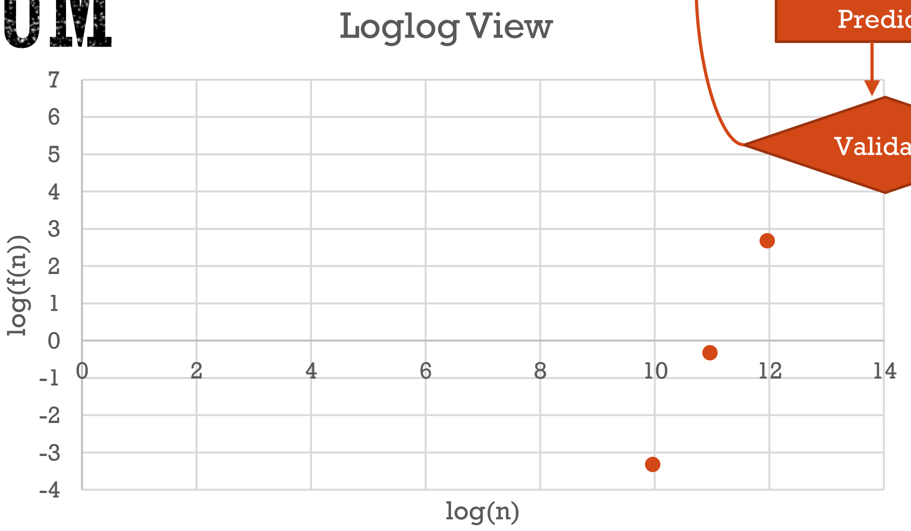


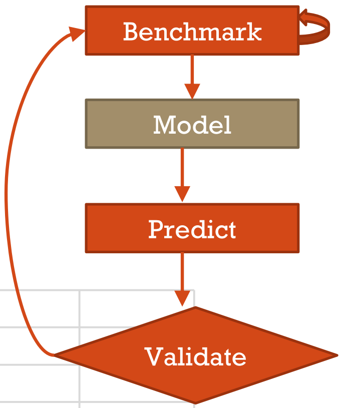
We have data so we could just use regression… can be messy though.

Another approach, when we suspect the answer will be a power function (… nested loops…) is to do a log-log plot.

MODELING THREESUM

- Start by creating a log-log plot, then do linear regression.
- Use the recovered values to compute T(N) = $a\,N^{b}$.
- Result: $1.006 \cdot 10^{-10} \cdot N^{2.999}$ seconds
- Note the function won't have a constant – there is an assumption happening that when n is low, then f(n) is 0.





Equation for line on a log-log plot:

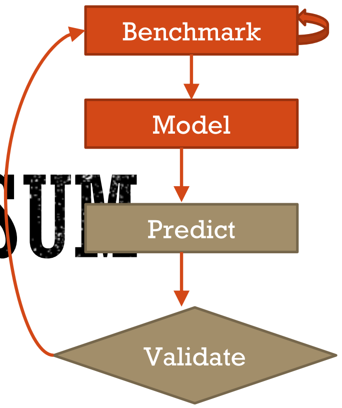
$\lg(f(n)) = b \lg n + c$

$b = 2.999$

$c = -33.2103$

$T(N) = a\,N^{b}$, *where* $a = 2^{c}$ *(2 is from lg)*

So: $T(N) = 1.006 \cdot 10^{-10} \cdot N^{2.999}$



PREDICTING AND VALIDATING THREESUM

- At this point we have a formula, that we could assume is correct.
- Let's be a little scientific though: let's predict (form a hypothesis) the time to execute on 16000 inputs by calculating f(16000). This gives 408.1.
- Okay – now we would need to benchmark the algorithm again…
- F(16000) confirmed?

N	T(N) (seconds)
250	0
500	0
1000	.1
2000	.8
4000	6.4
8000	51.1
16000	410.8

MODELING SMALL DATASETS

- Another way to determine the power function is by carefully scaling our input sizes and checking the impact on the resulting time.
- (We will assume $T(N) = aN^b$)
- Suppose we wrote:

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b}$$

$$\frac{T(2N)}{T(N)} = 2^b$$

$$lg^{\frac{T(2N)}{T(N)}} = b$$

N	T(N)	ratio	lg(ratio)
250	0	DNE	DNE
500	0	DNE	DNE
1000	.1	DNE	DNE
2000	.8	8	3
4000	6.4	8	3
8000	51.1	8.0	3

BTW, what would be the first sign that our T(N) assumption is wrong?

USING AN EMPIRICAL GROWTH FUNCTION

- So, is a empirically determined T(N) good enough?
- T(N) does provide an easy way to directly predict the run time of a program.

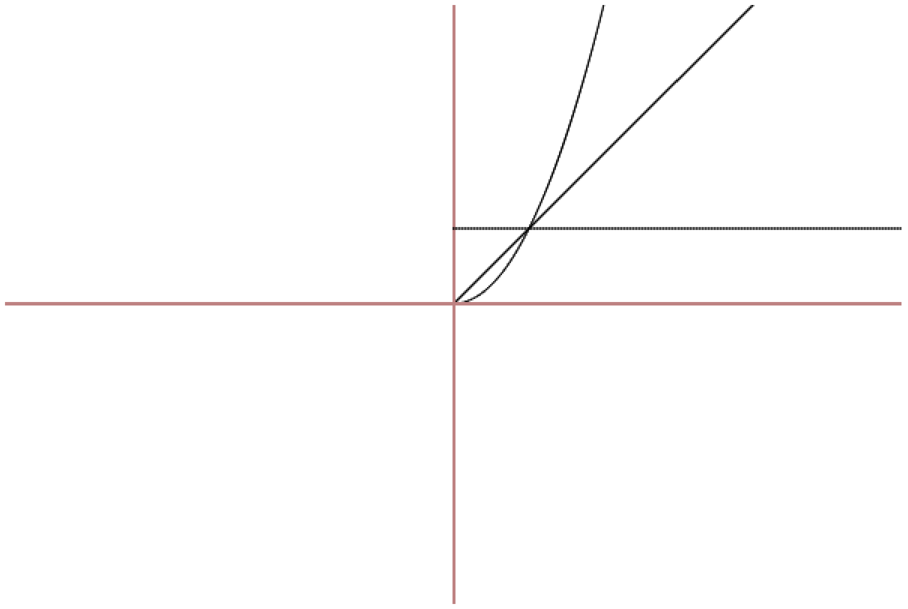
?

- However, the estimate is valid only for the specific context of where the benchmark was performed.
- Performance may not even be repeatable on the same system. (remember benchmark practices for computer parts?)
- Ultimately, the value of a *T(N)* analysis will be the power on *n*, not the whole expression.

ASYMPTOTICS

BOUNDING FUNCTIONS

- Our goal in this section will be to analyze the long-term character of a function by deducing some other functions which serve to *bound* it.
- Ex: they are always larger or smaller than it when *n* is large.
  - Another approach would be to find something identical to it…
- This is an *asymptotic* approach – we are examining the functions in the limit.
  - What makes asymptotics useful?



BOUNDING FUNCTIONS

- Until now, we've been using $T(N)$ – a growth function with output in units of time.
- However, we can be more precise: we can measure the number of operations in a computation.

- For now, let's assume we already these so-called growth functions, $f(n)$, available to us.

- Given a $f(n)$, we will explore several ways to analyze its behavior in the limit.

UNDERSTANDING UPPER BOUNDS

- On the right we have two (arbitrary) functions.
- Note: we say nothing about growth functions here. We are just talking about functions in general.
- The image has an interesting trend: the blue function appears to have a value that is always higher than the red function. This means that red is *upper bounded* by blue.
- Key thought: the bound will only hold once some threshold (x_0) is met. This doesn't restrict us – quite the opposite.
- This lets us neglect constant overhead that a method may need for any size input.

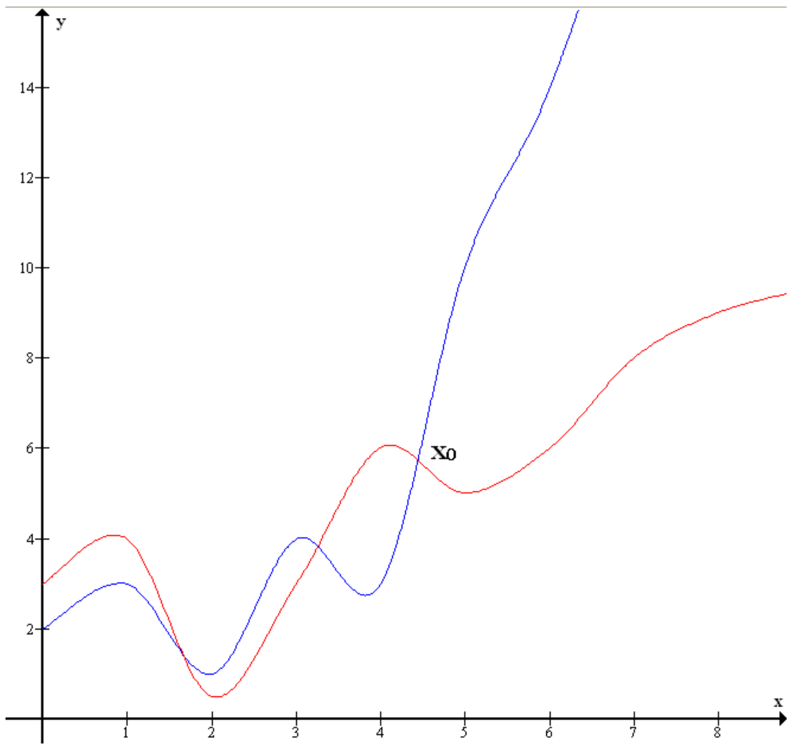


Image from Wikipedia Commons.

EXPRESSING UPPER BOUNDS: BIG-OH

- Definition: We write $\mathrm{f(n)} = \mathrm{O}\big(\mathrm{g(n)}\big)$ to indicate that $f(n)$ has an *upper bound* (i.e., will run with at most some number of operations). Formally: $f(n) = O\big(g(n)\big)$ iff $|\mathrm{f(n)}| \leq c|g(n)|$ (for $n > x_0$, where x_0 is a constant).

- Process: to find g(n) for f(n) in f(n)=O(g(n)), we take $f(n)$, eliminate all but the dominant term, and remove any constant factor from the dominant term.

- For example:
  - $f_1(n) = 2n+3$ ----> $g_1(n)=n$ (let c=5, assume $n > 0$, then $2n+3 \leq 5*n$ holds)
  - $f_2(n) = lg(n)*35+n^2$ ----> $g_2(n)=n^2$ (let c=36, assume $n > 1$, then $lg(n)*35+n^2 \leq 36n^2$)

Big-Oh is the typical way algorithms are characterized. Unfortunately, the "upperboundedness" of Big-Oh means people can get lazy: $O(2^n)$.

EXPRESSING UPPER BOUNDS: BIG-OH LOGS

It is the case that:

- $log_2(n) = O(log_2(n))$ and,
- $log_{10}(n) = O(log_2(n))$

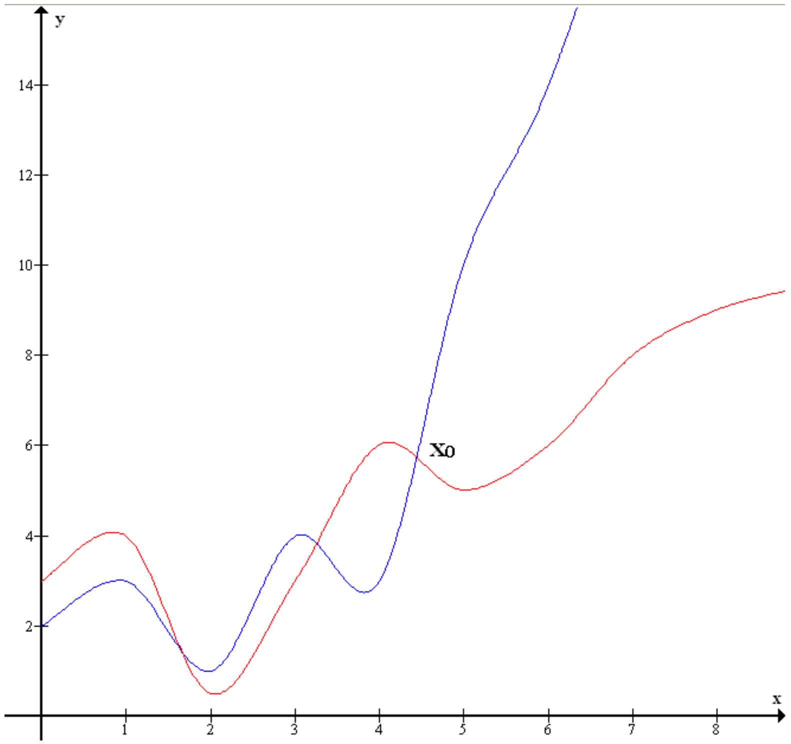
Why?

EXPRESSING LOWER BOUNDS: OMEGA

The opposite of Big-Oh is Omega: a lower bound.

- Definition: We write $\mathrm{f(n)} = \Omega(\mathrm{g(n)})$ to indicate that f(n) has a *lower bound* (i.e., will run at least this long on an input): $f(n) = \Omega(g(n))$ iff $|\mathrm{f(n)}| \geq c|g(n)|$ (for $n > x_0$, where x_0 is a constant).

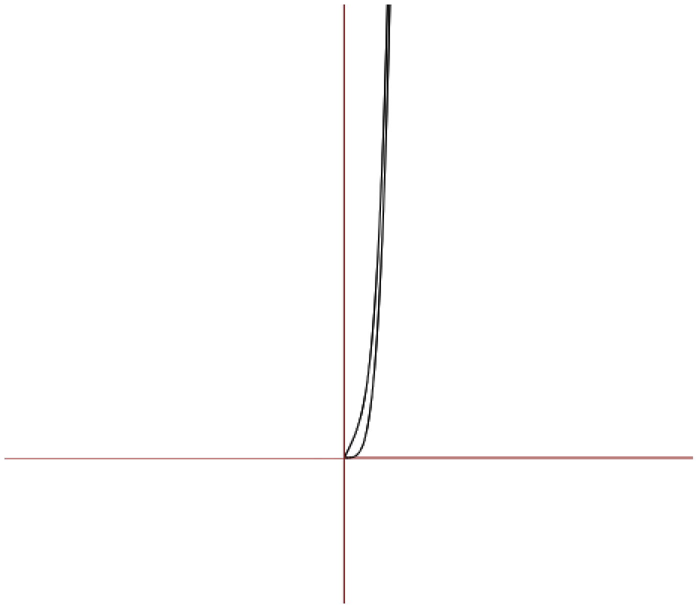
Omega won't be too useful for us – why?



UNDERSTANDING TIGHT BOUNDS

- Consider the functions x^4 and x^4+2x; seen on right.
- Instead of an upper or lower bound, they "exactly" follow each other for large values of *n*.

- Looking at the functions, we can see that the 2x term will, for large values of *n*, be negligible in terms of x^4.
- Here's a thought: what if we just used x^4?



EXPRESSING TIGHT BOUNDS: TILDE

- *Definition: We write ~f(N) to indicate a function that, when divided by f(N), approaches 1 as N grows, and so we write g(N) ~f(N).*

- Process: to find g(n) for f(n) in g(n) ~ f(n), we take f(n) and eliminate all but the dominant term.
- For example:
  - $f_1(n) = 2n+3$ ----> $g_1(n)=2n$ (since $\lim_{n\to\infty} \frac{2n}{2n+3} = 1$)
  - $f_2(n) = lg(n)*35+n^2$ ----> $g_2(n)=n^2$ (since $\lim_{n\to\infty} \frac{n^2}{n^2+35\lg(n)} = 1$)

The authors like ~ notation for the simple reason: it doesn't obscure the constant like Big-Oh. In fact, we can choose to think of the above expressions as:

- $f_1(n) = g_1(n)+O(1)$
- $f_1(n) = g_1(n)+O(lg(n))$

That is, we are upper bounding the value of the eliminated terms.

EXPRESSING TIGHT BOUNDS: THETA

- There is also a second (more common) tight bound: theta.
  - We will prefer tilde over theta as it is a little easier to find.
- *Definition:* We write $\mathrm{f(n)} = \theta\big(\mathrm{g(n)}\big)$ to indicate that f(n) an upper and lower bound represented by the same function: $\mathrm{f(n)} = \theta\big(\mathrm{g(n)}\big)$ iff $\mathrm{f(n)} = \mathrm{O}\big(\mathrm{g(n)}\big) \wedge f(n) = \Omega\big(g(n)\big)$.

(theta acts like a case of ~ approximation where *c=1*.)

RANKING BOUNDS

- Once you have a bound, use the following hierarchy to rank terms:
- $1 < logn < n < nlog < n^c < c^n < c^{c^n}$ with $1 < c$.

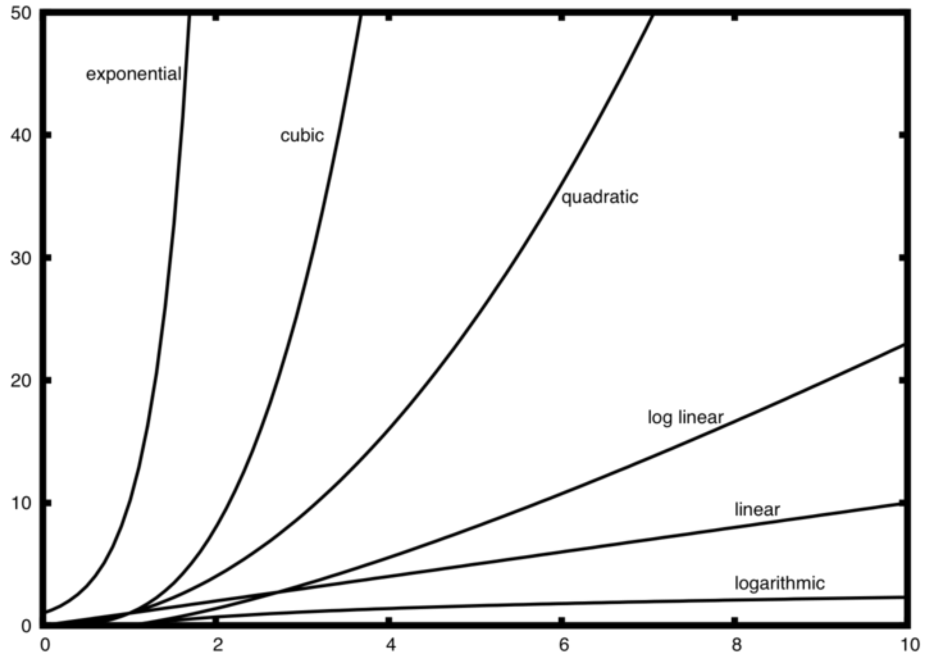


Image from http://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

ANALYTICAL ANALYSIS

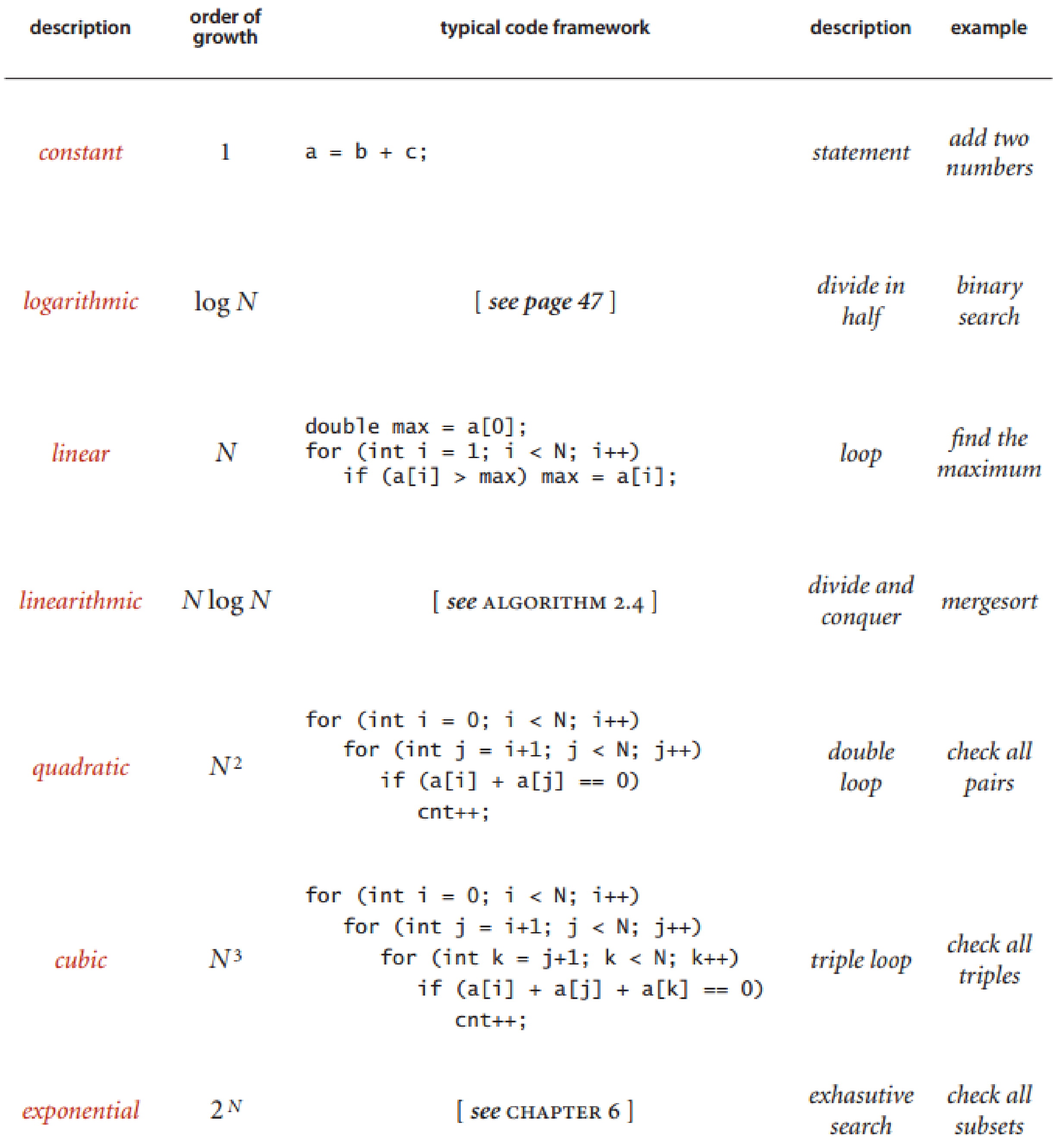
DEFINING F(N)

- We haven't done a good job at defining *f(n)*.
  - So far we've waved our hand and say that it is the number of operations (lines?) that occur.
- Really, we are better off thinking in terms of a specific ***cost*** metric.
  - For example, the number of additions, the number of array accesses, etc.
- It is nice (but not required) that our cost measure the most "expensive" or "frequent" operation.
- Having a specific cost metric generally makes our life easier, WLOG, should we choose to do direct analysis (i.e., can skip around).

DEFINING F(N)

```
boolean searchBinary(int target,
                              int[] pool) {
       int lo = 0;
       int hi = pool.length - 1;
       while (lo <= hi) {
           int mid = lo + (hi - lo) / 2;
           if (target < pool[mid])
               hi = mid - 1;
           else if (target > pool[mid])
               lo = mid + 1;
           else
               return true;
       }
       return false;
```

TRENDS



description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`   if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      if (a[i] + a[j] == 0)`
		`         cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      for (int k = j+1; k < N; k++)`
		`         if (a[i] + a[j] + a[k] == 0)`
		`            cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhasutive search*	*check all subsets*

More details: Sedgewick p186-188.

Table from Sedgewick and Wayne.

ANALYSIS OF THREESUM

- For our cost model, we will do the analysis as a count of the number of comparisons with 0. (This is proportional to the number of array accesses.)

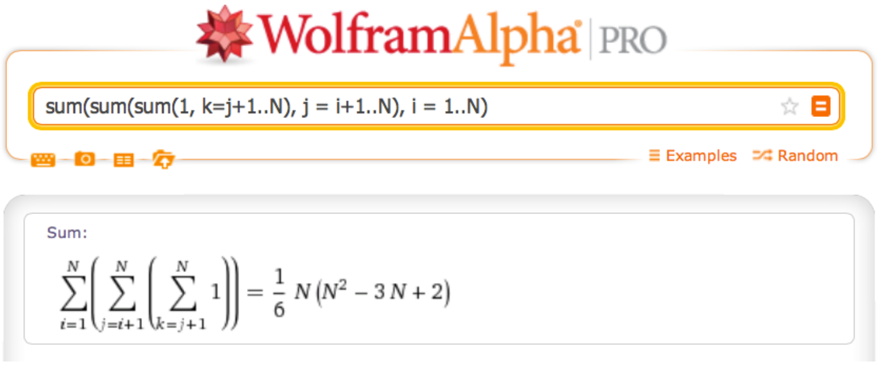
```
//Sedgewick and Wayne
 public static int threeSum(int[] a)
   {
      int N = a.length;
      int count = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }
```

$$\sum_{i=1}^{N}\sum_{j=i+1}^{N}\sum_{k=j+1}^{N} 1$$

- Could we instead pick the number of times count is incremented?

Analysis of ThreeSum

- Methods:
  - Write it out and try to find a pattern.
    - Can try using an integral: $\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \approx \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dzdydx \approx \frac{1}{6}N^3$
  - Use tables of summation formulas.
  - Use a CAS:



```
sum(sum(sum(1, k=j+1..N), j = i+1..N), i = 1..N)
```

Sum:

$$\sum_{i=1}^{N}\left(\sum_{j=i+1}^{N}\left(\sum_{k=j+1}^{N} 1\right)\right) = \frac{1}{6} N (N^2 - 3N + 2)$$

Some times the answer will be messy – it can be fair game to modify the bounds original summations to get a more reasonable solution.

INPUT DEPENDANCE

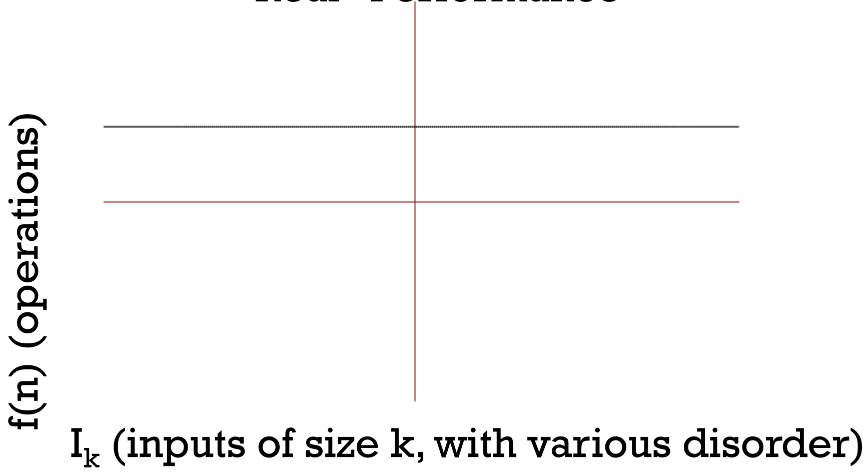
- Until now we've assumed that an algorithm's runtime behavior can be determine solely from the problem size *n*…
- Consider the linear algorithm for finding an element in an array.
- Is it possible for us to write a growth function to count the number of times pool[i] is compared?

```
boolean searchLinear(int target,
                                 int[] pool) {
  for(int i = 0; i < pool.length; i++)
    if(pool[i] == target)
      return true;

  return false;
}
```

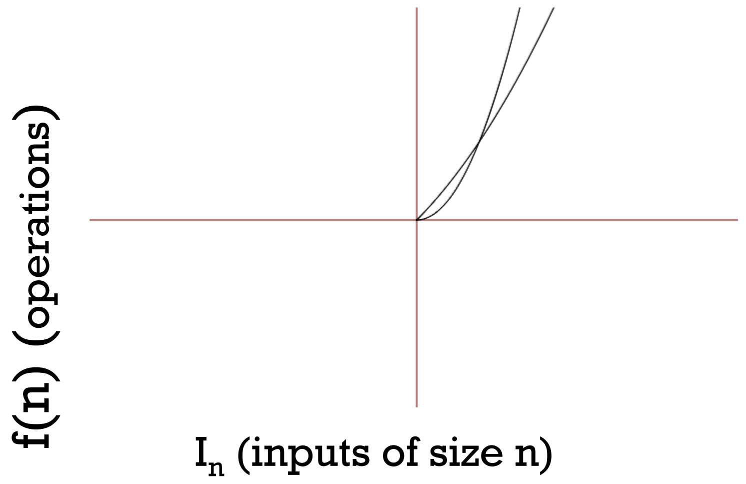
INPUT DEPENDANCE

- Let's assume the inputs of size k are sorted starting with the most sorted inputs.
- Although some methods lack direct dependence on input character (e.g., selection sort), others (e.g., insertion sort) are influenced.
- On a nearly sorted input, insertion sort will behavior more linearly than quadratically.

Selection Sort
"Real" Performance



Insertion Sort
"Real" Performance



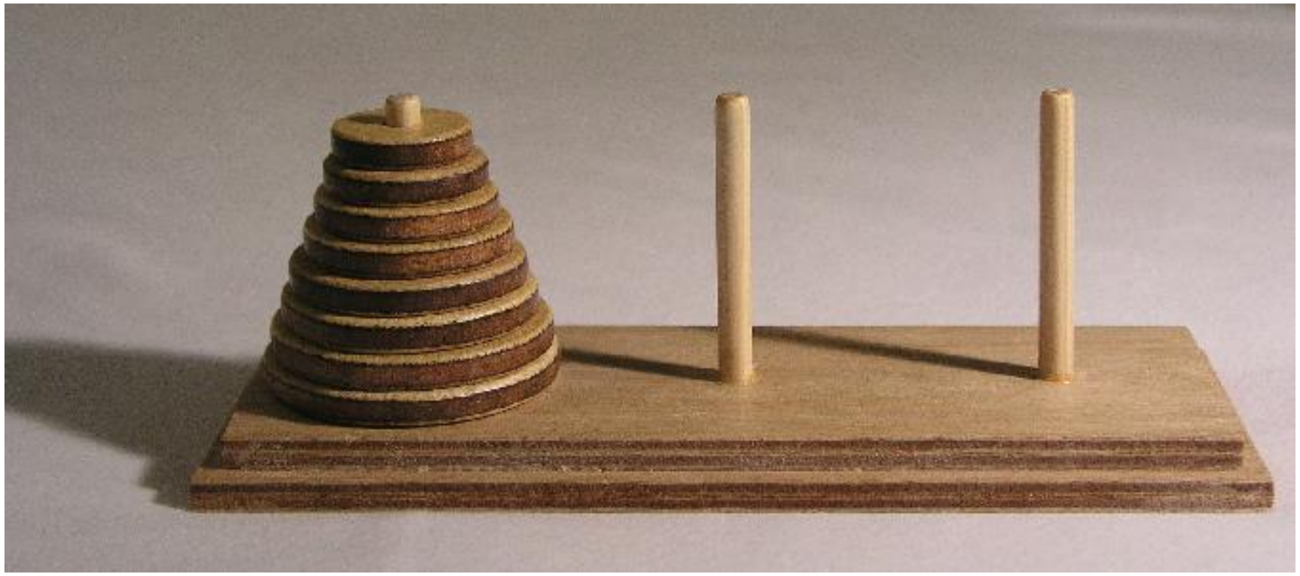
RECURRENCES

ANALYZING RECURSIVE METHODS

- So far, we haven't looked at any methods that call themselves.
- This is where things get tricky – we end up with growth functions expressed in terms of themselves… a ***recurrence***.
- Sure we can write them but what good do they do?
- We'll approach this problem by working through an example.
- In SER222, you won't be expected to do analysis on recursion methods from scratch. However, it is important that you are aware of the general process and be able to answer questions about it.

TOWERS OF HANOI

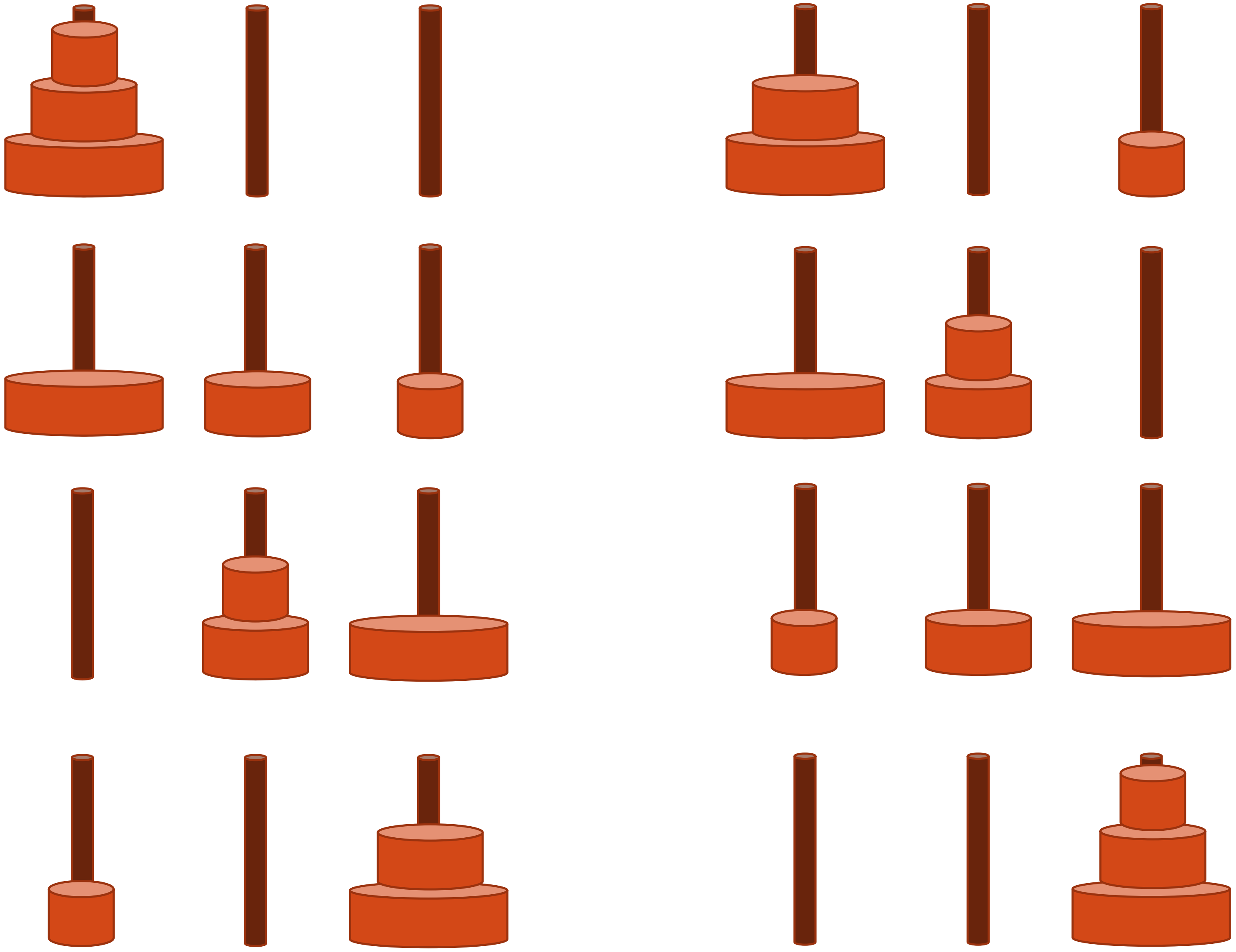
- Goal: move the tower of disks from first rod to the third rod.



Rules:
1) Disks are moved from the top position in a stack to the top in another stack.
2) Only one disk can be moved at a time.
3) Disks can only be placed on larger disks.

The question: how many moves are needed for a particular size (n) of stack?

TOWERS OF HANOI



Any ideas how to express this as an algorithm?

SOLVING TOWERS OF HANOI

- Assumptions:
  - The rods and the disks on them are already modeled by some data structure.
  - There is a *move* method that moves a disk between two rods.

For our analysis, we want to deduce some *f(n)* where *n* is the number of disks in a tower and *f(n)* is the number of moves to complete the game.

```
Hanoi(Start, Temp, End, n) {
  if n=1
    //move from Start's top to End's top.
    move(Start, End)
else {
    //move n-1 disks from Start to End via Temp
    Hanoi(Start, End, Temp, n-1)
    //nth disk into final position
    move(Start, End) //nth disk
    //move n-1 disks from Temp to Start via End
    Hanoi(Temp, Start, End, n-1)
  }
}
```

Algorithm is from Compared to What? By Rawlins.

GUESSING A GROWTH FUNCTION

- Let *f(n)* be the moves required by a call to Hanoi for *n* disks.
- What would *f(1)* be?
- How could *f(n)* be written?

```
Hanoi(Start, Temp, End, n) {
  if n=1
    //move from Start's top to End's top.
    move(Start, End)
else {
    //move n-1 disks from Start to End via Temp
    Hanoi(Start, End, Temp, n-1)
    //nth disk into final position
    move(Start, End) //nth disk
    //move n-1 disks from Temp to Start via End
    Hanoi(Temp, Start, End, n-1)
  }
}
```

GUESSING A GROWTH FUNCTION

- By "hand"*, let's calculate out *f(n)* for some values of *n*.
- Powers of two?? Why?
- In fact, it looks just like $f(n)=2^n-1$.
- Since it seems to match exactly… are we done?

n	2f(n)+1		n	2^n
1	1		1	2
2	3		2	4
3	7		3	8
4	15		4	16
5	31		5	32
6	63		6	64
7	127		7	128
8	255		8	256
9	511		9	512
10	1023		10	1024
11	2047		11	2048
12	4095		12	4096
13	8191		13	8192
14	16383		14	16384
15	32767		15	32768
16	65535		16	65536

* Don't actually do stuff by hand. Use tools like Excel or WolframAlpha.

INTRODUCING INDUCTION

- Our next step is going to be proving that the close-form solution we guessed is correct.
- For this, we'll use mathematical induction.
  - Why?

- The purpose of induction is to show that some property holds for every element of a set (usually something like the nature numbers).
- Induction has two parts a base case and an inductive step:
  - Base case: the initial starting point (e.g., 0) in the set and an element where the property that can be shown to be true directly (i.e., P(0) is true).
  - Induction step: shows that if the property holds true for some k (i.e., P(k) is true), then that the property holds for k+1 (i.e., P(k+1) is true).

PROVING F(N)

Ahem…

- The known truth:
  - $f(1) = 1$
  - $f(n) = 2f(n-1)+1$

  and want to show that $f(n)=2^n-1$.

From guessing. We will call this the ***closed*** solution.

Proof by induction on n ∈ N:

Base case: n=1

Need to start somewhere. This seems okay. (Why?)

Immediate: $f(1)=1=2^1-1=1$

Evaluate our closed solution and compare.

Inductive Step:

Assume: $f(k)=2^k-1$

"Inductive hypothesis"

Show f(k) implies f(k+1):

The key! Show that if it worked for some value, it would work for the next one.

$f(k+1)=2[f(k)]+1$

Act like we want to use known solution to compute k+1.

$=2[2^k-1]+1$

Substitute our assumed solution.

$=[2^{k+1}-2]+1$

$=2^{k+1}-1$

We recover our closed solution but with k+1.

QED

PROVING F(N)

- Our goal was to find (and verify!) a closed formula for *f(n)*:
  - $f(n)=2^n-1$
- Now we have a proper growth function that we can use with Big-Oh and tilde approximation.

- What is the Big-Oh order of f(n)?

- What is the tilde approximation of f(n)?

GENERALIZING SOLUTIONS

- While we've managed to find a *closed solution* for the Hanoi recurrence, we were lucky.
- Remember that we (conveniently) guessed the exponential form of the solution.

- For other algorithms, there are a few general approaches:
  - Guessing. It can work, as we have seen.
  - The master theorem. A "presolved" formula that only requires a few constants.
  - Tables of common recurrences, sums, etc. Look up answer.
  - And of course: a CAS.

GENERALIZING SOLUTIONS

- Solving recurrences isn’t only useful for determining a functions growth. It can also be used to derive closed form formulas for polynomial time algorithms:
  - Fibonacci: $F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$

- But how? In essence, we are encoding the iteration of a problem into the character of a numerical sequence.

This is the idea of *Generating Functions* – a general approach to solving some kinds of algorithmic problems. (and also the main topic of the sequel to your textbook: *Analysis of Algorithms*.)

BIG-OH USAGE

SOFTWARE ENGINEERING USAGE

- Most of the time in programming, Big-Oh will be noted simply as "*… with O(n) running time on size of array*" or something similar.
- Other times, documentation may implicitly state the Big-Oh of a method by specifying the algorithm used to implement it, "*… is implemented with merge sort…*" In this case, one must be aware of the basic algorithms to judge performance.

MATHEMATICAL USAGE

- May also see as part of expressions:
  - $4n^2 + O(n)$
- In these cases, think about O(n) being an imprecise value, like the $\pm$ symbol. It indicates a quality plus some bounded value.

MATHEMATICAL USAGE

- A word of caution: just because Big-Oh notation can occur in expressions, does not mean the standard rules of algebra apply.
- For example
  - $1 = O(n)$: $c * n$ is clearly larger than 1 when n is large.
  - $2 = O(n)$: $c * n$ is clearly larger than 2 when n is large.
  - So then: $1 = O(n) = 2$, 1=2. Something is wrong…

Technically, an expression involving a Big-Oh is a set. A set of all functions which are bounded by the associated *f(n)*.

So, it is more correct to write $\{1\} \subseteq O(n)$. There is directionality!

APPLICATIONS OF BIG-OH

- Big-Oh is not only for analysis of algorithms.
- Useful in any case where you want to bound a value according to a function. Happens in Physics, Math, etc.
  - This where we would see expressions like $4n^2+O(n)$.

- Can use to identify long term error:

$$\sum_{k=0}^{n}(k^2+O(k)) = magichappens = \frac{1}{3}n^3+O(n^2)$$

- Can make life easier when writing inexact recurrences.