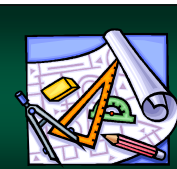
# Analysis of Algorithms

Part 1

1



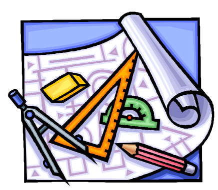
# Analysis of Algorithms

Looking at how well it works

2

## What is an Algorithm?

- An *algorithm* is a sequence of unambiguous instructions that solves a problem
- Can be represented various forms – i.e. languages
- Each unique set of data fed into an algorithm specifies an *instance* of that algorithm





3

## Analysis of Algorithms

- Algorithms must to analyzed to determine whether it should be used
- This field is called *algorithmics*
- How it is analyzed:
  - correctness
  - unambiguity
  - effectiveness
  - finiteness/termination - does it in a *finite* amount of time



4

## Correctness

- Correctness means the algorithm obtains the required output with *valid* input
- In other words, does it do what it is supposed to do
- *Proof of Correctness* can be easy for some algorithms – and quite difficult for others
- *Proof of Incorrectness* is quiet easy – find one instance where it fails on valid input



5

## Effectiveness

- How good is the algorithm?
- Two major areas of interest:
  - *time efficiency* defines how long the algorithm will take to complete
  - *space efficiency* defines how much memory and resources will be needed
- … and how these algorithm react as the data set grows



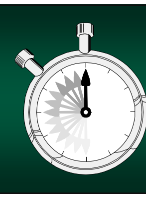
6

## Effectiveness

- Knowing this, we can determine if there is a better algorithm
- Does there exist a better algorithm?
  - better time complexity
  - better space efficiency
- *Efficiency is a **HUGE** part of creating professional programs*



7



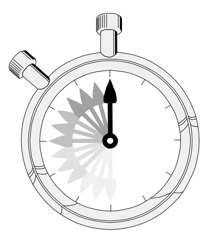
## Time Complexity Basics

Basically, time is complex

8

## Time Complexity

- One of the important aspects of analyzing an algorithm is to determine how reacts to the size of data
- Analyzed by the number of repetitions of the *basic operation* as a function of input size

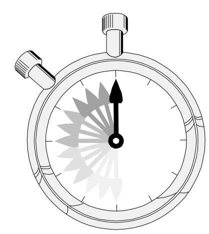




9

## Time Complexity

- The basic operation is what contributes the ***most*** towards the running time of the algorithm
- It is essentially the operations that are repeated in our algorithm





10

## Size and Basic Operation Examples

| Problem | Input size measure | Basic operation |
|---|---|---|
| Searching for key in a list of *n* items | Number of list's items, i.e. *n* | Key comparison |
| Multiplication of two matrices | Matrix dimensions or total number of elements | Multiplication of two numbers |
| Checking primality of a given integer *n* | *n*'size = number of digits (in binary representation) | Division |
| Typical graph problem | # of vertices and edges | Visiting a vertex or traversing an edge |



11

## Theoretical Analysis of Time Efficiency

Total execution time

Number of times the basic operation executes

$$T(n) \approx c_{op} C(n)$$

number of items in the data set

Cost: execution time of a basic operation



12

## Empirical analysis of time efficiency

- Analysis can be performed by observation
- Select a specific (typical) sample of inputs
- Use….
  - physical unit of time and / or
  - count actual number of basic operation's executions
- Analyze the empirical data to determine T, $C_{op}$, and C(n)



13

## Time Complexity Cases

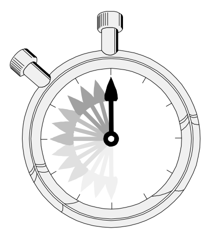
- Worst case: $C_{worst}(n)$
  - maximum executions over a set of size n
  - can be linear, quadratic, or even exponential!
  - the worst case can be exceedingly rare
- Best case: $C_{best}(n)$
  - minimum executions over a set of size n
  - best case can also be exceedingly rare



14

## Time Complexity Cases

- Average case: $C_{avg}(n)$
  - how it executes using typical data…
  - …in other words, the type of data the algorithm will normally encounter
  - this is **NOT** the average of worst and best case





15

## Order of Growth

Uh, "O"

16

## Order of Growth

- What is important is how an algorithm's time grows as $n \to \infty$
- Examples:
  - how will it run on a computer that is twice as fast?
  - how long does it take with twice the input?



17

## Time Complexity Cases

- For some algorithms, efficiency depends on the *form* of input
  - sometimes, the order of data, or the type of data can drastically increase cost
  - some algorithms are sensitive to certain criteria
- This will appear again and again with lists, trees, and, *especially*, sorting



18

## Order of Growth

- One property of functions that we are interested in its rate of growth
- *Rate of growth* doesn't simply mean the "slope" of the line associated with a function
- Instead, it is more like the *curvature* of the line



19

## Several Growth Functions

- There are several functions
- In increasing order of growth, they are:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - Log Linear ≈ n log n
  - Quadratic ≈ $n^2$
  - Exponential ≈ $2^n$



20

## Growth Rates Compared

| n = | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **1** | 1 | 1 | 1 | 1 | 1 |
| **log n** | 0 | 1 | 2 | 3 | 4 |
| **n** | 1 | 2 | 4 | 8 | 16 |
| **n log n** | 0 | 2 | 8 | 24 | 64 |
| **$n^2$** | 1 | 4 | 16 | 64 | 256 |
| **$n^3$** | 1 | 8 | 64 | 512 | 4096 |
| **$2^n$** | 2 | 4 | 16 | 256 | 65536 |



21

## Classifications

- Using the known growth rates…
  - algorithms are classified using three notations
  - these allows you to see, quickly, the advantages/disadvantages of an algorithm
- Major notations:
  - Big-O
  - Big-Theta
  - Big-Omega



22

## Order of Growth

| Notation | Name | Meaning |
|---|---|---|
| O ( *n* ) | Big-O | class of functions f(n) that grow no faster than n |
| Θ ( *n* ) | Big-Theta | class of functions f(n) that grow at same rate as n |
| Ω ( *n* ) | Big-Omega | class of functions f(n) that grow at least as fast as n |



23

## Big-O

- So, Big-O notation gives an *upper bound* on growth of an algorithm
- We will use Big-O almost exclusively rather than the other two



24

## Big-O

- The following means that the growth rate of f(n) is no more than the growth rate of n
- This is one of the classifications mentioned earlier

$$f(n) \text{ is } O(n)$$



25

## Why it is O-some!

- These classes make it is easy to…
  - compare algorithms for efficiency
  - making decisions on which algorithm to use
  - determining the scalability of an algorithm
- So, if two algorithms are the same class…
  - they have the same rate of growth
  - both are equally valid solutions



26

## O(1)

- Represents a constant algorithm
- It does not increase / decrease depending on the size of n
- Examples
  - appending to a linked list *(with an end pointer)*
  - array element access
  - practically all simple statements



27

## O(log n)

- Represents logarithmic growth
- These increase with n, but the rate of growth diminishes
- For example: for base 2 logs, the growth only increases by one each time n doubles



28

## O(log n) Examples

- Searching for an item on a sorted array – (e.g. a binary search)
- Traversing a sorted tree



29

## O(n)

- Represents an algorithm that grows linearly with n
- Very common in programming – for iteration
- Examples:
  - finding an item in a linked list
  - merging two sorted arrays



30

## O(n log n)

- Represents an algorithm that has "log linear" growth
- These algorithms grow based on both n and n's log value



31

## O(n log n) Examples

- Quick Sort
- Heap Sort
- Merge Sort
- Fourier transformation



32

## O($n^2$)

- Represents an algorithm that has "quadratic" growth
- These algorithms grow dramatically fast depending on the size of n
- **Do <u>NOT</u> use for large values of n**



33

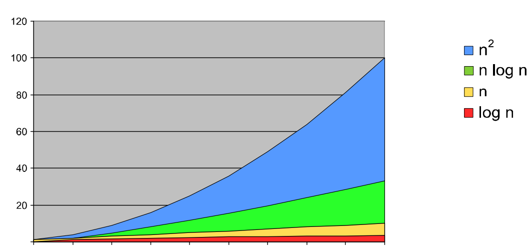
## O($n^2$) Examples

- Bubble Sort, Selection Sort, etc.…
- matrix multiplication
- merging unsorted arrays



34

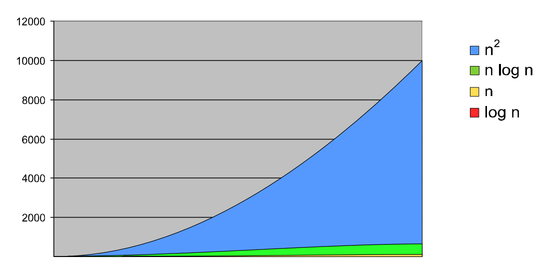
## Growth: 1 to 10






35

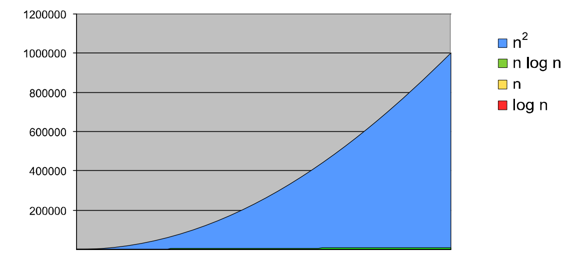
## Growth: 1 to 100






36

## Growth: 1 to 1000






37

## Seconds needed (Microsecond Op)

| n | O(log n) | O(n) | O(n log n) | O(n²) |
|---|---|---|---|---|
| 10 | 0.000003 | 0.000010 | 0.000033 | 0.000100 |
| 100 | 0.000007 | 0.000100 | 0.000664 | 0.010000 |
| 1,000 | 0.000010 | 0.001000 | 0.009966 | 1.000000 |
| 10,000 | 0.000013 | 0.010000 | 0.132877 | 100.000000 |
| 100,000 | 0.000017 | 0.100000 | 1.660964 | 2.8 hours |
| 1,000,000 | 0.000020 | 1.000000 | 19.931569 | 11.6 days |
| 10,000,000 | 0.000023 | 10.000000 | 232.534966 | 3.16 years |



38

## Big-O Summary

| | | |
|---|---|---|
| Good | O(1) | O(yeah) |
| | O(log n) | O(nice) |
| | O(n) | O(kay) |
| | O(n log n) | O(my!) |
| | O($n^2$) | O(no!) |
| Bad | O($2^n$) | O(sh*t!) |



39

## Big-O Math

Time for a "Big-O" headache

40

## Asymptotic Analysis

- Any algorithm can be analyzed, and its complexity/growth can be written as a simple mathematical expression
- *Asymptotic analysis* of an algorithm determines the running time in big-O notation



41

## Asymptotic Analysis

1. Find the worst-case number of primitive operations executed as a function of the input size
2. Eliminate meaningless values once the base rate in found



42

## Example

- If we analyze an algorithm and find it executes **12 * n – 1**
  - constant factors and lower-order terms dropped since they become meaningless for large values of n
  - remember, this is a *growth rate*
- It will be **O(n)**



43

## Examples

$3000n + 7 \rightarrow O(n)$

$2n^5 + 3n^3 + 5 \rightarrow O(n^5)$

$7n^3 - 2n + 3 \rightarrow O(n^3)$



44

## Test Your Might…

```
for(i = 0; i < 100; i++)
{
   total += values[i];
}
```

O(1)



45

## Test Your Might…

```
for(x = 0; x < array.size; x++)
{
    sum += array[x];
}

for(x = 0; x < array.size; x++)
{
    sum -= array[x];
}
```

O(n)



46

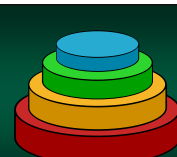
## Test Your Might…

```
for (x = 0; x < array.size; x++)
{
   for (y = 0; y < x; y++)
   {
      sum += x - y;
   }
}
```

$O(n^2)$



47



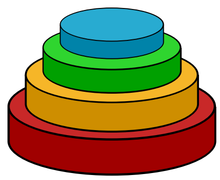
## Towers of Hanoi

A Classic Stack Puzzle

48

## Towers of Hanoi

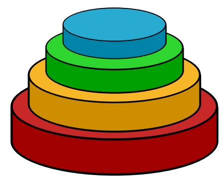
- *Towers of Hanoi* is a famous puzzle created by mathematician Edouard Lucas in 1883
- It is based on a "legend"





49

## The Puzzle



- Consists of a collection of discs with unique diameters
- Each disc has a hole in the center used to place it on one of 3 different pegs



50

## The Puzzle

- Goal:
  - starts with all the discs stacked on one peg
  - goal is to move all the discs to another peg
- Gameplay:
  - a disc cannot be placed onto a smaller disc
  - only one disc can be moved at a time



51

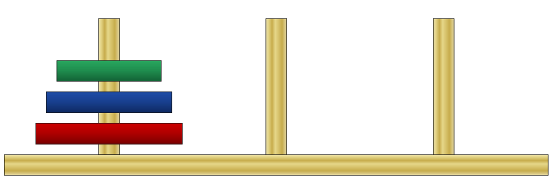
## The Legend

- Well, the legend was created along with the puzzle and expanded over time
- Basically, somewhere in a *hidden place*, priests are moving a stack of 64 discs
- The ancient prophecy states that when the entire stack is moved…the World ***ENDS!***



52

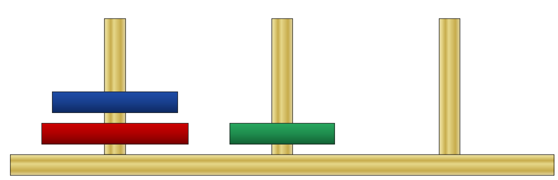
## Hanoi: 3 Discs





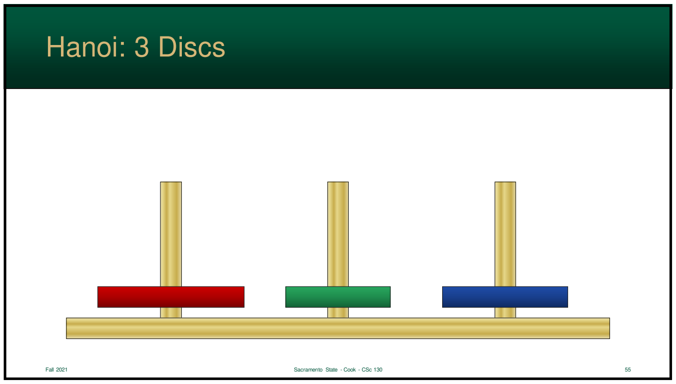
53

## Hanoi: 3 Discs

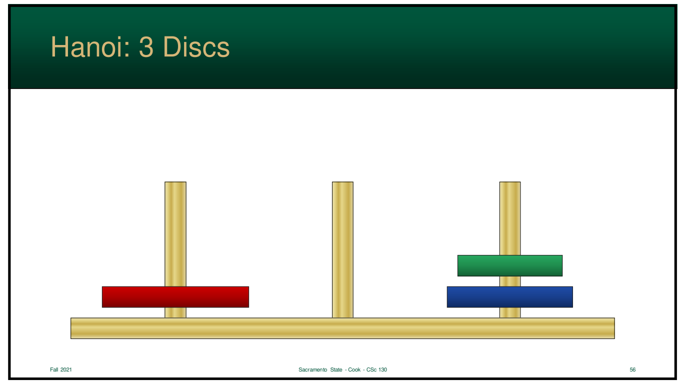




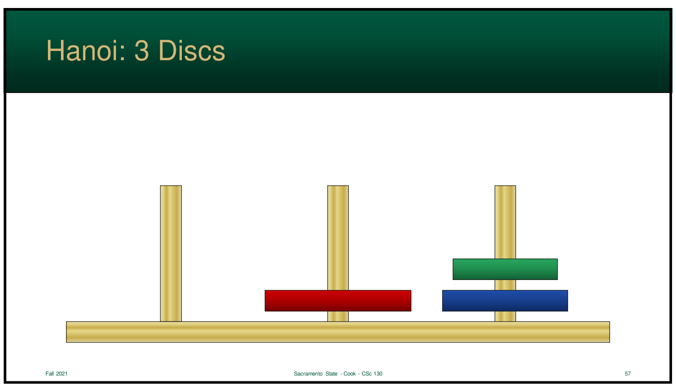
54



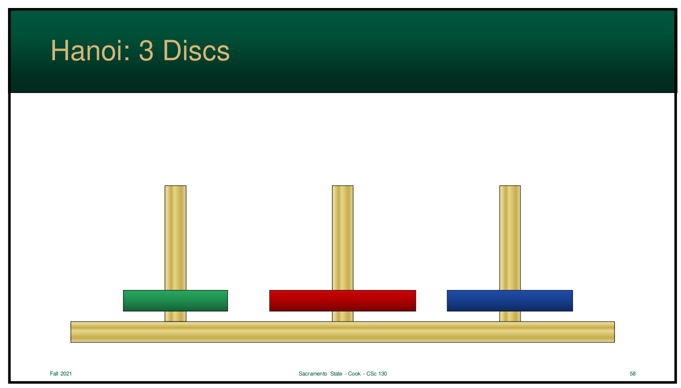

55



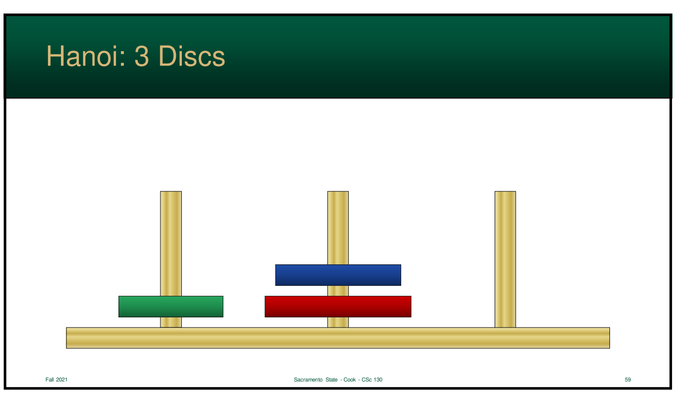

56



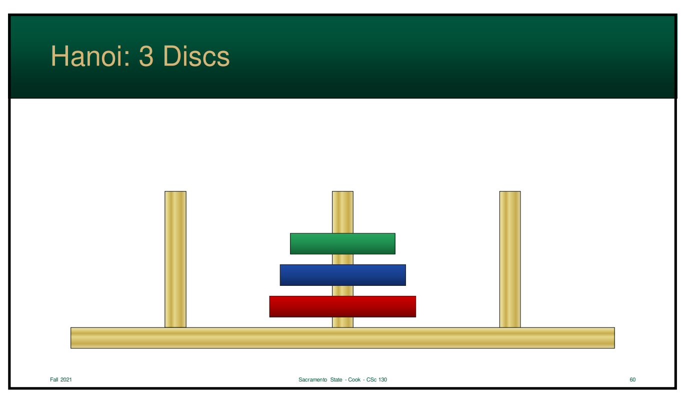

57




58




59




60

## Hanoi: Solution

- An elegant solution is to use recursion
- Since disks are move from each tower using LIFO, each tower can be represented as a stack
- The "classic" recursive solution just shows what actions to take, it doesn't move any values… but you could modify it easily to.



61

## Hanoi: in Java

```
void hanoi(int disc, Stack from, Stack temp, Stack dest)
{
   if (disc == 1)
   {
      move(from, dest);   //base case
   }
   else
   {
      hanoi(disc - 1, from, dest, temp);
      move(from, dest);
      hanoi(disc - 1, temp, from, dest);
   }
}
```



62

## Hanoi: Demo Version

```
void hanoi(int disc, Stack from, Stack temp, Stack dest)
{
   if (disc == 1)
   {
      System.out.println(disc + ": " + from + " to " + dest);
   }
   else
   {
      hanoi(disc - 1, from, dest, temp);
      System.out.println(disc + ": " + from + " to " + dest);
      hanoi(disc - 1, temp, from, dest);
   }
}
```



63

## Hanoi: Demo Version

```
// Disc 1 is the *smallest* disc.
// We start recursion with the BIGGEST disc.

void main()
{
   hanoi(3, 'A', 'B', 'C');
}
```



64

## Hanoi: Demo Output

```
1: A to C
2: A to B
1: C to B
3: A to C
1: B to A
2: B to C
1: A to C
```



65

## Hanoi: Time Complexity

- The minimum number of moves required for a stack of N discs is is $2^N$-1
- So, the time complexity of the Towers of Hanoi puzzle is $O(2^n)$ - exponential!



66

## Hanoi: Is the World Ending?

- The "legend" states that the monks have to move 64 discs… order of $2^{64}$
- So…
  - if they take one second to move each disc, it will take them 584,542,046,090 years!
  - if a super-computer moves a disc once per microsecond, it still takes 584,542 years!



67