Analysis of Algorithms

Section 1.4

Analysis of Algorithms

Looking at how well it works

What is an Algorithm?

- An *algorithm*:
  - a sequence of unambiguous instructions for solving a problem
  - can be represented various forms
- Each unique set of data fed into an algorithm specifies an *instance* of that algorithm

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

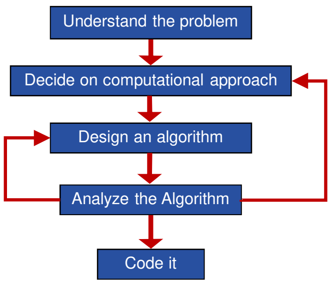
Correctness

- 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

Effectiveness

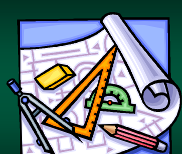
- How good is the algorithm?
  - What is the time efficiency?
  - What is the space efficiency?
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality
- Computational efficiency is a large part of creating professional programs

Design Process



Algorithm Design Strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer



Well Known Problems

Data structures and you

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problem
- Geometric problems
- Numerical problems

Searching

- Find a given value, called a search key, in a given set
- *Extremely* common in programs
- Examples of searching algorithms
  - Sequential search
  - Binary search
  - Tree search

Sorting

- It is useful (and efficient) to sort a list of data in ascending order
- Examples:
  - sorting scores by highest to lowest
  - sorting filenames in alphabetical order
  - sorting students by their student-id
- Two properties
  - Stable: preserves the order of any two equal elements
  - In place: requires lots of extra memory

Sorting

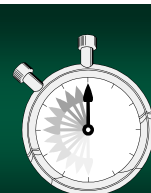
- There are multiple sorting algorithms which get complex as they become more efficient
- Some examples:
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort

String Processing

- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.

Graph Problems

- A *graph* is a collection of points called vertices, some of which are connected by line segments called edges.
- A common example of a graph is the network – Internet, phone, etc…
- Examples of graph algorithms
  - Graph traversal algorithms
  - Shortest-path algorithms

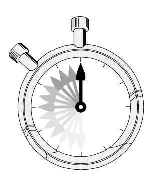


Time Complexity Basics

Basically, time is complex

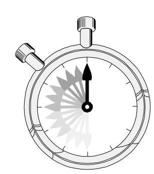
Time Complexity

- One of the most 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



Time Complexity

- The basic operation is what contributes the *most* towards the running time of the algorithm



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

Theoretical Analysis of Time Efficiency

Total execution time

Number of times the basic operation executes

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

Cost: execution time of a basic operation

Empirical analysis of time efficiency

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

Time Complexity Cases

- For some algorithms, efficiency depends on *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 when we deal with lists, trees, and, *especially*, sorting

Time Complexity Cases

- Worst case: $C_{worst}(n)$
  - maximum executions over a set of size n
  - can be linear, exponential or even geometric!
- Best case: $C_{best}(n)$
  - minimum executions over a set of size n
- Average case: $C_{avg}(n)$
  - "average" over a set of size n
  - times the basic operation will execute on *typical* data
  - *NOT* the average of worst and best case
  - the worst case can be exceedingly rare

Order of Growth

- How does the required time grow as $n \to \infty$
- Example:
  - How much longer does it take to solve problem of double input size?
  - Will it take twice as long? Is it linear?
- In computer science several types of growth occur.
- Algorithms will fall into one of these categories for worst-case, best-case, and average-case

Operational Growth for n

Function	10	100	1000
$\log_2 n$	3.32	6.64	9.97
n	10	100	1000
$n \log_2 n$	33.21	664.38	9965.78
n^2	100	10000	1,000,000

Order of Growth

Uh, "O"

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

Order of Growth

- What is important is how an algorithm's time grows as $n \to \infty$
- In computer science several types of growth occur

Order of Growth

- Algorithms will fall into one of these categories for worst-case, best-case, and average-case
- Examples:
  - how will it run on a computer that is twice as fast?
  - how long does it take with twice the input?

Several Growth Functions

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

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

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

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

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

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

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

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

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

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 <u>one</u> each time n <u>doubles</u>

O(log n) Examples

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

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

O(n log n)

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

O(n log n) Examples

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

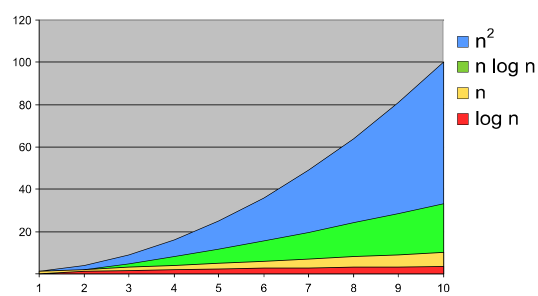
O(n^2)

- Represents an algorithm that has "exponential" growth
- These algorithms grow dramatically fast depending on the size of n
- Avoid for large values of n !

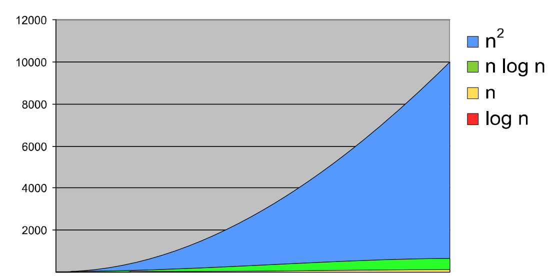
O(n^2) Examples

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

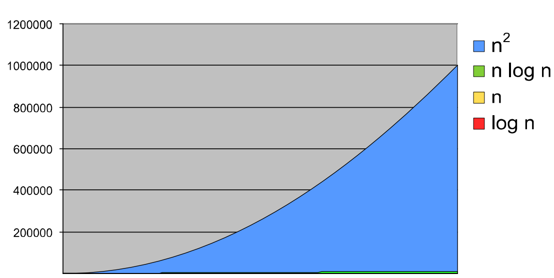
Growth: 1 to 10



Growth: 1 to 100



Growth: 1 to 1000



Time Given a 1 Microsecond Op

n	O(log n)	O(n)	O(n log n)	O(n^2)
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	6.94 days
1,000,000	0.000020	1.000000	19.931569	1.9 years
10,000,000	0.000023	10.000000	232.534966	190.2 years

Big-O Math

Time for a "Big-O" headache

Asymptotic Analysis

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

Asymptotic Analysis

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

Asymptotic Analysis

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

Examples

3000n + 7 is O(n)

$2n^5 + 3n^3 + 5$ is $O(n^5)$

$7n^3 - 2n + 3$ is $O(n^3)$

Test Your Might…

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

O(1)

Test Your Might…

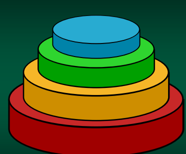
```
for(x = 0; x < n; x++)
{
      sum += score[x];
}

for(x = 0; x < n; x++)
{
      sum -= score[x];
}
```

O(n)

Test Your Might…

```
for (x = 0; x < n; x++)
{
   for (y = 0; y < x; y++)
   {
      sum += x - y;
   }
}
```

O(n^2)

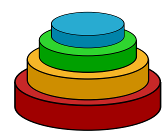


Towers of Hanoi

A Classic Stack Puzzle

Towers of Hanoi

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



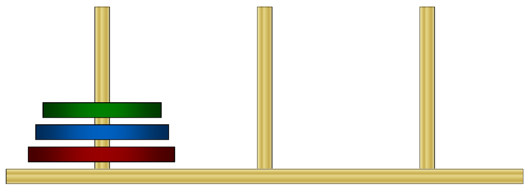
The Legend

- 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!*

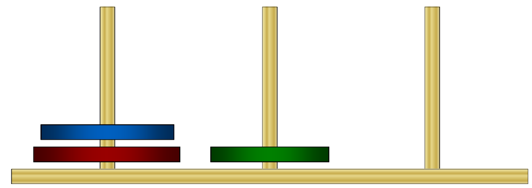
The Puzzle

- 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
- A disc cannot be placed onto a smaller disc
- Only one disc can be moved at a time
- Puzzle starts with all the discs stacked on one peg
- The goal is to move all the discs to another peg

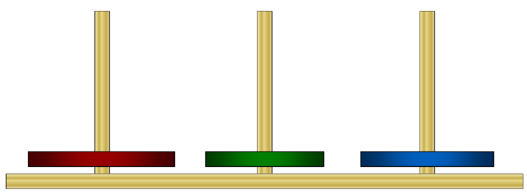
Hanoi: 3 Discs



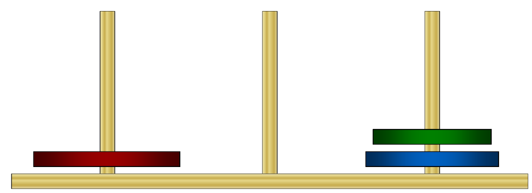
Hanoi: 3 Discs



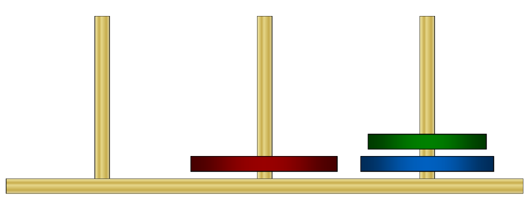
Hanoi: 3 Discs



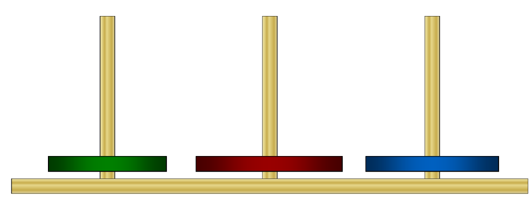
Hanoi: 3 Discs



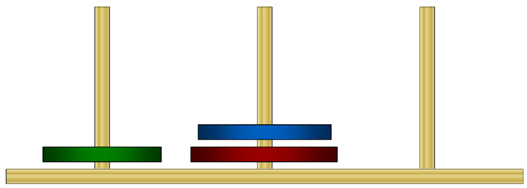
Hanoi: 3 Discs



Hanoi: 3 Discs



Hanoi: 3 Discs



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.

Hanoi: in Java

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

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);
   }
}
```


Hanoi: Demo Version

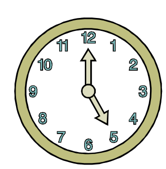
```
void hanoi(int disc, char F, char T, char D) {
   if (disc == 1) {
      System.out.println(disc + ": " + F + " to " + D);
   } else {
      hanoi(disc - 1, F, D, T);
      System.out.println(disc + ": " + F + " to " + D);
      hanoi(disc - 1, T, F, D);
   }
}

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


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
```


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!



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!