Algorithms

Fida Dankar

Algorithms

Computers are mainly used to *store*, *retrieve* and *manipulate* data.

Data Structures

Algorithms

The goal of this course is to teach the most common algorithm design techniques through a wide variety of examples.

Why Study Algorithms?

Required course for Computer Science!

Lot of applications.

Cryptography, Web Search (Page Rank), Compression, Error Correcting Codes, Graphics, Recommendation Systems (e.g. Amazon, Netflix), Fourier Transform (e.g. JPEG, MP3), Shortest Paths (e.g. Google Maps), Auctions (e.g. Ebay), High Frequency Trading, Optimization, Compilers, and many more.

For any subject X, there is most likely a very practical area called 'Computational X' which is about algorithms related to X.

(*e.g. X = Physics, Chemistry, Biology, Music, Photography, Statistics, Geometry, Topology, Design etc.*)

Provides perspective on other areas.

(*e.g. Algorithmic Game Theory, Quantum Information Processing, Complexity Theory*)

Tech companies focus on Algorithms and Data Structures.

It is fun and good for your brain.

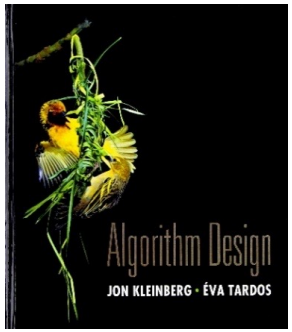
Why Study Algorithms?

"The Algorithm's coming-of-age as the new language of science promises to be the most disruptive scientific development since quantum mechanics."

Bernard Chazelle,
Professor of Computer Science,
Princeton University.

Administrativia

Algorithms Design by Kleinberg and Tardos.



- Video Lectures by Tim Roughgarden
  http://www.algorithmsilluminated.org/
- Course material will be uploaded on Brightspace.

- Assignments (30%) *Submissions in weeks 4, 6, 11, 13.*
- Midterm Exam (30%) *During class hrs of Lecture 14.*
- Final Exam (40%) *In the final exam period.*

Course Schedule

LECTURE	TOPIC	READING
1	Introduction	1.1
2	Analysis of Algorithms	2.1 – 2.4
3	Graphs, Graph Representation	3.1
4	Graph Traversal	3.2, 3.3
5	Graph Connectivity	3.5
6	Directed Acyclic Graphs and Topological ordering	3.6
7	Greedy Algorithms, Interval Scheduling	4.1, 4.2
8	Shortest paths: Dijkstra's algorithm	4.4
9	Minimum Spanning Trees: Prim's algorithm	4.5
10	Minimum Spanning Trees: Kruskal's algorithm	4.6
11	Huffman Codes and Data Compression	4.8
12	Divide and Conquer algorithms, Mergesort	5.1, 5.2
13	Closest Pair of Points	5.4
14	**MIDTERM EXAM**	
15	Integer Multiplication	5.5
16	Convolutions and Fast Fourier Transform	5.6
17	Dynamic Programming, Weighted Interval Scheduling	6.1
18	Subset Sum and Knapsack	6.4
19	Sequence alignment	6.6
20	Shortest Paths: Bellman-Ford Algorithm	6.8
21	Network Flows, Ford Fulkerson Algorithm	7.1
22	Maxmimum Flows and Minimum Cuts	7.2
23	Better Augmenting Paths	7.3
24	Computational Intractability, Polynomial time reductions	8.1, 8.2
25	NP and NP-complete problems	8.3, 8.4
26	Introduction to Approximation Algorithms	11.1 – 11.3
27	Introduction to Randomized Algorithms	13.1 – 13.4
28	Course Review	

Assignments

- A few rules about assignments:

  You can submit assignments in ***groups of at most*** 3***.***
  All members in a group should have full understanding of all solutions.

  The goal is learn from others and ***not to distribute workload****.*

  Never submit solutions that you don't understand.

  Solutions should be written in neat handwriting or typed.
  Figures can be drawn by hand.
  Code (rarely required) needs to be submitted as text files.

  All submissions are via Brightspace.
  Never submit solutions by email.

  Late policy. $score' = score \cdot \left[1 - \left(\frac{t_{late}}{t_{max}}\right)^2\right]$ t_{late} *: how late your submission is.*
  $t_{max} = 1$ *day.*

Advice

- Lectures:

The <u>main ideas and concepts</u> will be explained in the lectures at a high level.

Advice

- Some advice:

Read the textbook regularly.

*There will be **mandatory** reading assignments.*

Try as many problems from the textbook as possible.

Ask a lot of questions in and outside class.

*Maintain a sporting spirit and **have fun**.*

- Warning: *Serious "head-banging" and patience required!*

Celebrate Failure!

Go forth and fail!

Fear of failure is the main reason for failure!

Reading

Mandatory reading. Section 1.1 of the textbook.

Read the whole chapter if you feel excited.

In the next lecture, we will start with Chapter 2.

Optional.

From Roughgarden's video lectures,
watch Section 1.1 : Why Study Algorithms?

http://www.algorithmsilluminated.org/

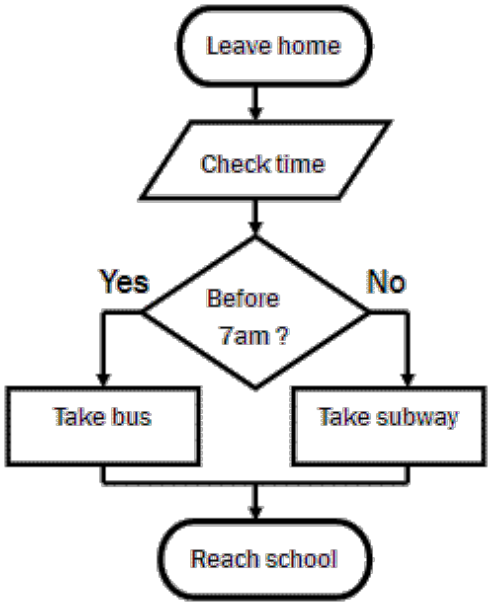
Questions

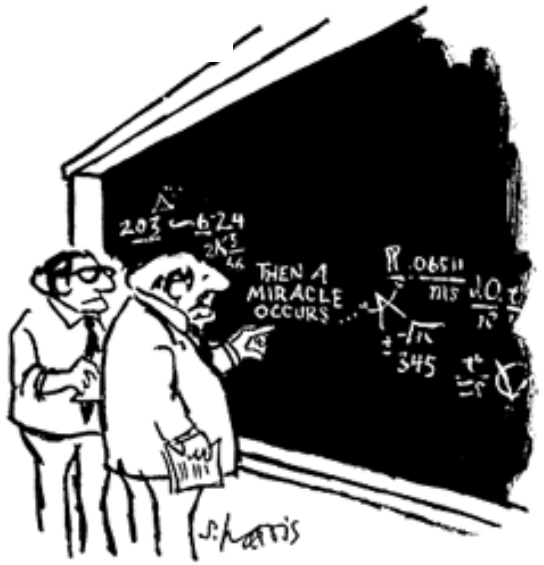
- What is an algorithm?
- What is a program?
- What are data structures?

Problems, Algorithms, Programs, Data Structures

Algorithm: A recipe for producing the output from the input.

It must be *finite and concrete (non-ambiguous)*.





"I think you should be more explicit here in step two."

Problems, Algorithms, Programs, Data Structures

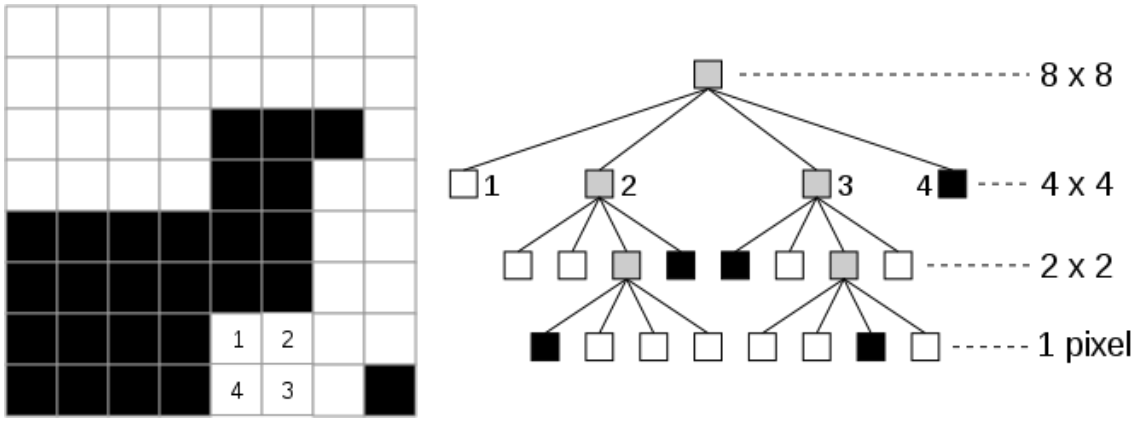
Program: implementation of an algorithm in a programming language.

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
    else:
      print '"];'
      children = []
      for n, child in enumerate(ast[1:]):
         children.append(dotwrite(child))
      print '    %s -> {' % nodename,
      for name in children:
         print '%s' % name,
```

Problems, Algorithms, Programs, Data Structures

Data Structure: a way to organize the data so that it can be used effectively by an algorithm.



More Questions

- What is an effective algorithm?
- How do we check effectiveness

Algorithms

What is an efficient algorithm?

One that uses less resources. Mainly time and space.

Most of the time, we will focus on the running time.

In particular, running time as a function of input size.

Given two algorithms, how do we know which is better?

Option 1: Find out empirically.

Option 2: Find out analytically.

Empirical Approach

- We need to implement both algorithms
- The results depend on quality of implementation and platform.
- Can only be tried on a limited number of inputs.
- Does not tell us how to design efficient algorithms.

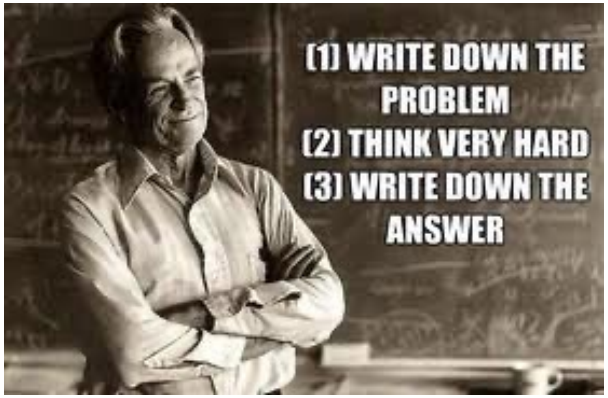
Analytical Approach

The goal is to ignore superficial differences between programs and concentrate on large components of running time.

Step 1: Start with a high level description of the algorithm

Step 2: Identify "basic operations" in the algorithm

Step 3: Count the number of basic operations required for an input of size n by **staring** at the algorithm.



High Level Description

Problem: Calculate the sum of n number stored in an array A

Algorithm ArraySum (A, n)

Input: Array A with n integers
Output: Sum of the n integers in A

sum:=0
for $i = 1$ to n **do**
    sum := sum + $A[i]$

return sum

pseudocode

Identifying Basic Operations

Algorithm ArraySum (A, n)

Input: Array A with n integers

Output: Sum of the n integers in A

sum:=0
for $i = 1$ to n **do**
    sum := sum + $A[i]$
return sum

Counting Basic Operations

Algorithm ArraySum (A, n)

Input: Array A with n integers
Output: Sum of the n integers in A

Note that incrementing is also an assignment In which 1 is added to the variable value.

```
sum:=0
for i = 1 to n do
        sum := sum + A[i]
return sum
```

1 assignment

1 assignments,
$n-1$ increments

n assignments,
n additions

1 return

Let c_1 = time for assignment, c_2 = time for increment,
c_3 = time for addition, c_4 = time for return

$$\text{Running time} = c_1(n+2) + c_2(n-1) + c_3 n + c_4 = \alpha n + \beta$$

Counting Basic Operations

Running time $= c_1(n+2) + c_2(n-1) + c_3 n + c_4 = \alpha n + \beta$

To make things simple, we assume that each operation takes *unit* time.

Furthermore, we only care about the no. of operations

In the above example, we ignore the constants α and β and simply say that the running time is $\propto n$ i.e., it is *linear*.