# CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2009

Part I

# Administrivia

# Instructional Staff

- Instructor: Chandra Chekuri (`chekuri`)
  - Office Hours: 1-2pm Thursday/Friday, and by appointment.
- Teaching Assistants:
  - Alina Ene (`ene1`)
  - Kyle Fox (`kylefox2`)
  - Dan Schreiber (`dschrei2`)
  - Office Hours: See course webpage

# Electronic Bulletin Boards

- Webpage: `www.cs.illinois.edu/class/fa09/cs473`
- Newsgroup: `class.fa09.cs473` on server news.cs.illinois.edu

# Textbooks

- Prerequisites: All material in CS 173, CS 225 and CS 373
- Text-book: Algorithm Design by Kleinberg and Tardos
- Lecture Notes: Available on the web-page after every class
- Additional References
  - Algorithms: Dasgupta, Papadimitriou, and Vazirani.
  - Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein
  - Computers and Intractibility: Garey and Johnson
  - Previous class notes of Jeff Erickson, Sariel Har-Peled, Mahesh Viswanathan and the instructor.

# Prerequisites

- Asymptotic notation: $O(), \Omega(), o(), \omega()$
- Discrete Structures: sets, functions, relations, equivalence classes, partial orders, trees, graphs
- Logic: predicate logic, boolean algebra
- Proofs: **by induction**, by contradiction
- Basic sums and recurrences: sum of a geometric series, unrolling of recurrences, basic calculus
- Data Structures: arrays, multi-dimensional arrays, linked lists, trees, balanced search trees, heaps
- Abstract Data Types: lists, stacks, queues, dictionaries, priority queues
- Algorithms: sorting (merge, quick, insertion), pre/post/in order traversal of trees, depth/breadth first search of trees (maybe graphs)
- Basic analysis of algorithms: loops and nested loops, deriving recurrences from a recursive program
- Concepts from Theory of Computation: languages, automata, Turing Machine, undecidability, non-determinism
- Programming: in some general purpose language
- Elementary Discrete Probability: event, random variable, independence
- Mathematical maturity

# Grading Policy: Overview

- Homeworks: 25%
- Midterms: 40% ($2 \times 20$)
- Finals: 35%

# Homeworks

- One homework every week: Assigned on Monday and due the following Monday by midnight (in practice no later than 7am on Tue morning). Drop off in Elaine Wilson's office (3229 Siebel). **Exception:** HW 0 is due on Tuesday in class.
- Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
- Groups can be changed a *few* times only
- Homeworks will be "turned in" orally every third week; the rest of the times you will turn in a written homework
  - Oral: Explain (verbally) to TA the solution to problems that are asked for.
  - Written: Write solutions to every problem and turn in the written solutions.

# More on Homeworks

- No extensions or late homeworks accepted.
- To compensate, the homework with the least score will be dropped in calculating the homework average.
- Important: Read homework faq/instructions on website.

# Head Banging/Discussion Sessions

- 1 hour problem solving session led by TAs
- Three sections
  - Tuesday 5.00–5.50pm in Siebel 1302
  - Wednesday 2.00–2.50pm or 3.00–3.50pm in Siebel 1111

# Advice

- Attend lectures, please ask plenty of questions
- Attend head-banging sessions
- Don't skip homework and don't copy homework solutions
- Study regularly, not just before homeworks
- Ask for help promptly. Make use of office hours.

Part II

# Course Goals and Overview

# Topics

- Some useful basic algorithms and data structures
- Broadly applicable techniques in algorithm design
  - Understanding problem structure
  - Brute force enumeration and backtrack search
  - Reductions
  - Recursion
    - Divide and Conquer
    - Dynamic Programming
  - Greedy methods
  - Network Flows and Linear (Integer) Programming (optional)
- Analysis techniques
  - Correctness of algorithms via induction and other methods
  - Recurrences
  - Amortization and elementary potential functions
- Polynomial-time Reductions, NP-Completeness, Heuristics

# Goals

- Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
- Algorithmic thinking
- Understand/appreciate limits of computation (intractability)
- Learn/remember some basic tricks, algorithms, problems, ideas

# Goals

- Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
- Algorithmic thinking
- Understand/appreciate limits of computation (intractability)
- Learn/remember some basic tricks, algorithms, problems, ideas
- Have fun!!!

# Part III

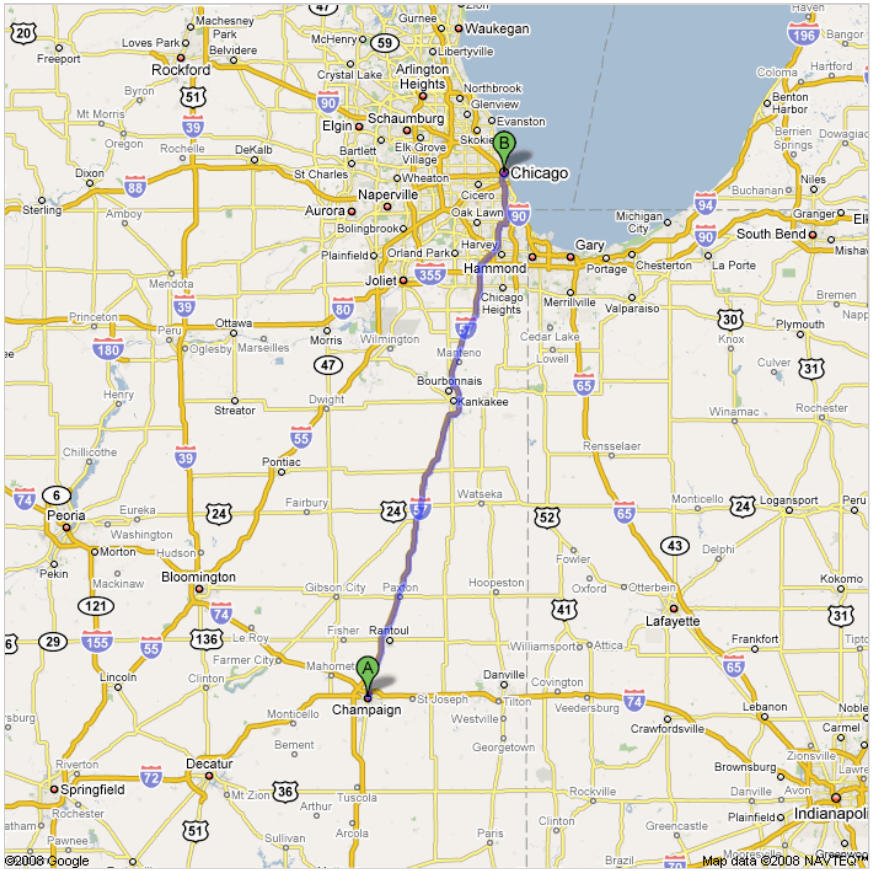
# Algorithmic Problems in the Real World

# Shortest Paths

Google Maps

Directions to Chicago, IL
**136 mi** – about **2 hours 20 mins**





# Data Compression

Lossless compression:

The gzip home page

# Data Compression

Lossless compression:

The gzip home page


Lossy compression: music, images, video (JPEG, MPEG standards)

# Search and Indexing: String Matching and Link Analysis

- Web search: Google, Yahoo!, Microsoft, Ask, ...
- Text search: Text editors (Emacs, Word, Browsers, ...)
- Regular expression search: grep, egrep, emacs, Perl, Awk, compilers

# Public-Key Cryptography

Foundation of Electronic Commerce

RSA Crypto-system: generate key $n = pq$ where $p, q$ are *primes*

# Public-Key Cryptography

Foundation of Electronic Commerce

RSA Crypto-system: generate key $n = pq$ where $p, q$ are *primes*

**Primality:** Given a number $N$, check if $N$ is a prime or composite.

**Factoring:** Given a composite number $N$, find a non-trivial factor

# Programming: Parsing and Debugging

[godavari: /temp/test] chekuri % gcc main.c

**Parsing:** Is main.c a syntactically valid C program?

**Debugging:** Will main.c go into an infinite loop on some input?

**Easier problem ???** Will main.c halt on the specific input 10?

# Optimization

Find the cheapest of most profitable way to do things

- Airline schedules - AA, Delta, ...
- Vehicle routing - trucking and transportation (UPS, FedEx, Union Pacific, ...)
- Network Design - AT&T, Sprint, Level3 ...

Linear and Integer programming problems

# Part IV

# Algorithm Design

# Four Important Ingredients in Algorithm Design

- What is the problem (really)?
  - What is the input? How is it represented?
  - What is the output?
- What is the model of computation? What basic operations are allowed?
- Algorithm design
- Analysis of correctness, running time, space etc.

## Problem

Given an integer $N > 0$, is $N$ a prime?

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

Correctness?

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

Correctness? If $N$ is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

Correctness? If $N$ is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$
Running time?

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

Correctness? If $N$ is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$
Running time? $O(\sqrt{N})$ divisions? Sub-linear in input size!

## Problem

Given an integer $N > 0$, is $N$ a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
endfor
return ``PRIME''
```

Correctness? If $N$ is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$
Running time? $O(\sqrt{N})$ divisions? Sub-linear in input size! Wrong!

How many bits to represent $N$ in binary? $\lceil \log N \rceil$ bits.

How many bits to represent $N$ in binary? $\lceil \log N \rceil$ bits.

Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.
*Exponential* in the input size $n = \log N$.

- Modern cryptography: binary numbers with 128, 256, 512 bits.
- Simple Algorithm will take $2^{64}$, $2^{128}$, $2^{256}$ steps!
- Fastest computer today: about 1 petaFlops: $2^{50}$ floating point ops/sec.

How many bits to represent $N$ in binary? $\lceil \log N \rceil$ bits.

Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.
*Exponential* in the input size $n = \log N$.

- Modern cryptography: binary numbers with 128, 256, 512 bits.
- Simple Algorithm will take $2^{64}$, $2^{128}$, $2^{256}$ steps!
- Fastest computer today: about 1 petaFlops: $2^{50}$ floating point ops/sec.

**Lesson**

Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

So, is there an *efficient/good/effective* algorithm for primality?

So, is there an *efficient/good/effective* algorithm for primality?

**Question**

What does efficiency mean?

So, is there an *efficient/good/effective* algorithm for primality?

**Question**

What does efficiency mean?

In this class *efficiency* is broadly equated to *polynomial time*. $O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where $n$ is size of the input.

So, is there an *efficient/good/effective* algorithm for primality?

**Question**

What does efficiency mean?

In this class *efficiency* is broadly equated to *polynomial time*. $O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where $n$ is size of the input.

Why? Is $n^{100}$ really efficent/practical? Etc.

So, is there an *efficient/good/effective* algorithm for primality?

**Question**

What does efficiency mean?

In this class *efficiency* is broadly equated to *polynomial time*. $O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where $n$ is size of the input.

Why? Is $n^{100}$ really efficent/practical? Etc.

Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

# Primes is in P!

**Theorem (Agrawal-Kayal-Saxena'02)**

*There is a polynomial time algorithm for primality.*

First polynomial time algorithm for testing primality. Running time is $O(\log^{12} N)$ further improved to about $O(\log^6 N)$ by others. In terms of input size $n = \log N$, time is $O(n^6)$.

Breakthrough announced in August 2002. Three days later announced in New York Times. Only 9 pages!

# Primes is in P!

**Theorem (Agrawal-Kayal-Saxena'02)**

*There is a polynomial time algorithm for primality.*

First polynomial time algorithm for testing primality. Running time is $O(\log^{12} N)$ further improved to about $O(\log^6 N)$ by others. In terms of input size $n = \log N$, time is $O(n^6)$.

Breakthrough announced in August 2002. Three days later announced in New York Times. Only 9 pages!

Neeraj Kayal and Nitin Saxena were undergraduates at IIT-Kanpur!

# What about before 2002?

Primality testing a key part of cryptography. What was the algorithm being used before 2002?
Miller-Rabin *randomized* algorithm:

- runs in polynomial time: $O(\log^3 N)$ time
- if $N$ is prime correctly says "yes".
- if $N$ is composite it says "yes" with probability at most $1/2^{100}$ (can be reduced further at the expense of more running time).

Based on Fermat's little theorem and some basic number theory.

# Factoring

- Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
- Relies on the difficulty of factoring a composite number into its prime factors.
- There is a polynomial time algorithm that decides whether a given number $N$ is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

# Factoring

- Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
- Relies on the difficulty of factoring a composite number into its prime factors.
- There is a polynomial time algorithm that decides whether a given number $N$ is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

**Lesson**

Intractability can be useful!

# Digression: decision, search and optimization

Three variants of problems.

- Decision problem: answer is yes or no. **Example:** Given integer $N$, is it a composite number?
- Search problem: answer is a feasible solution if it exists. **Example:** Given integer $N$, if $N$ is composite output $a$ non-trivial factor $p$ of $N$.
- Optimization problem: answer is the *best* feasible solution (if one exists). **Example:** Given integer $N$, if $N$ is composite output the *smallest* non-trivial factor $p$ of $N$.

# Digression: decision, search and optimization

Three variants of problems.

- Decision problem: answer is yes or no. **Example:** Given integer $N$, is it a composite number?
- Search problem: answer is a feasible solution if it exists. **Example:** Given integer $N$, if $N$ is composite output $a$ non-trivial factor $p$ of $N$.
- Optimization problem: answer is the *best* feasible solution (if one exists). **Example:** Given integer $N$, if $N$ is composite output the *smallest* non-trivial factor $p$ of $N$.

For a given underlying problem,

$$\text{Optimization} \geq \text{Search} \geq \text{Decision}$$

# Quantum Computing

**Theorem (Shor'1994)**

*There is a polynomial time algorithm for factoring on a* quantum *computer.*

RSA and current commercial cryptographic systems can be broken if a quantum computer can be built!

# Quantum Computing

**Theorem (Shor'1994)**

*There is a polynomial time algorithm for factoring on a* quantum *computer.*

RSA and current commercial cryptographic systems can be broken if a quantum computer can be built!

**Lesson**

Pay attention to the model of computation.

# Problems and Algorithms

Many many different problems.

- Adding two numbers: efficient and simple algorithm
- Sorting: efficient and not too difficult to design algorithm
- Primality testing: simple and basic problem, took a long time to find efficient algorithm
- Factoring: no efficient algorithm known.
- Halting problem: important problem in practice, undecidable!

# Multiplying Numbers

**Problem** Given two $n$-digit numbers $x$ and $y$, compute their product.

## Grade School Multiplication

Compute "partial product" by multiplying each digit of $y$ with $x$ and adding the partial products.

$$\begin{array}{r} 3141 \\ \times 2718 \\ \hline 25128 \\ 3141 \\ 21987 \\ 6282 \\ \hline 8537238 \end{array}$$

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time
- Number of partial products: $\leq n$

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time
- Number of partial products: $\leq n$
- Adding partial products: $n$ additions each $\Theta(n)$ (Why?)

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time
- Number of partial products: $\leq n$
- Adding partial products: $n$ additions each $\Theta(n)$ (Why?)
- Total time: $\Theta(n^2)$

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time
- Number of partial products: $\leq n$
- Adding partial products: $n$ additions each $\Theta(n)$ (Why?)
- Total time: $\Theta(n^2)$

# Time Analysis of Grade School Multiplication

- Each partial product: $\Theta(n)$ time
- Number of partial products: $\leq n$
- Adding partial products: $n$ additions each $\Theta(n)$ (Why?)
- Total time: $\Theta(n^2)$

Is there a faster way?

# Fast Multiplication

Best known algorithm: $O(n \log n \cdot 2^{O(\log^* n)})$ time [Furer 2008]

Previous best time: $O(n \log n \log \log n)$ [Schonhage-Strassen 1971]

**Conjecture:** there exists and $O(n \log n)$ time algorithm

# Fast Multiplication

Best known algorithm: $O(n \log n \cdot 2^{O(\log^* n)})$ time [Furer 2008]

Previous best time: $O(n \log n \log \log n)$ [Schonhage-Strassen 1971]

**Conjecture:** there exists and $O(n \log n)$ time algorithm

We don't fully understand multiplication!
Computation and algorithm design is non-trivial!

# Course Approach

Algorithm design requires a mix of skill, experience, mathematical background/maturiy and ingenuity.

Approach in this class and many others:

- Improve skills by showing various tools in the abstract and with concrete examples
- Improve experience by giving many problems to solve
- Motivate and inspire
- Creativity: you are on your own!

# Topics

- Some useful basic algorithms and data structures
- Broadly applicable techniques in algorithm design
  - Understanding problem structure
  - Brute force enumeration and backtrack search
  - Reductions
  - Recursion
    - Divide and Conquer
    - Dynamic Programming
  - Greedy methods
  - Network Flows and Linear (Integer) Programming (optional)
- Analysis techniques
  - Correctness of algorithms via induction and other methods
  - Recurrences
  - Amortization and elementary potential functions
- Polynomial-time Reductions, NP-Completeness, Heuristics

# Part V

# Reductions and Recursion

# Reduction

Reducing problem $A$ to problem $B$:

- Algorithm for $A$ uses algorithm for $B$ as a *black box*

# Distinct Element Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

# Distinct Element Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
    endfor
endfor
return NO
```

# Distinct Element Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
    endfor
endfor
return NO
```

Running time:

# Distinct Element Problem

**Problem** Given an array $A$ of $n$ integers, are there any *duplicates* in $A$?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
    endfor
endfor
return NO
```

Running time: $O(n^2)$

# Reduction to Sorting

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
endfor
return NO
```

# Reduction to Sorting

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
endfor
return NO
```

Running time: $O(n)$ plus time to sort an array of $n$ numbers

Important point: algorithm uses sorting as a black box

# Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

- Positive direction: Algorithm for $B$ implies an algorithm for $A$
- Negative direction: Suppose there is no "efficient" algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)

# Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

- Positive direction: Algorithm for $B$ implies an algorithm for $A$
- Negative direction: Suppose there is no "efficient" algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)

Example: Distinct element reduces to Sorting in $O(n)$ time

- An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct element problem.
- If there is *no* $o(n \log n)$ time algorithm for Distinct element problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

# Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

# Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*

# Recursion

- Recursion is a very powerful and fundamental technique
- Basis for several other methods
  - Divide and conquer
  - Dynamic programming
  - Enumeration and branch and bound etc
  - Some classes of greedy algorithms
- Makes proof of correctness easy (via induction)
- Recurrences arise in analysis

# Selection Sort

Sort a given array $A[1..n]$ of integers.

Recursive version of Selection sort.

**SelectSort**$(A[1..n])$ :

```
    If (n = 1) return
    Find smallest number in A.  Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

# Selection Sort

Sort a given array $A[1..n]$ of integers.

Recursive version of Selection sort.

**SelectSort**$(A[1..n])$:
    `If` $(n = 1)$ `return`
    `Find smallest number in` $A$. `Let` $A[i]$ `be smallest number`
    `Swap` $A[1]$ `and` $A[i]$
    **SelectSort**$(A[2..n])$

$T(n)$: time for SelectionSort on an $n$ element array.

$T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$

$T(n) =$

# Selection Sort

Sort a given array $A[1..n]$ of integers.

Recursive version of Selection sort.

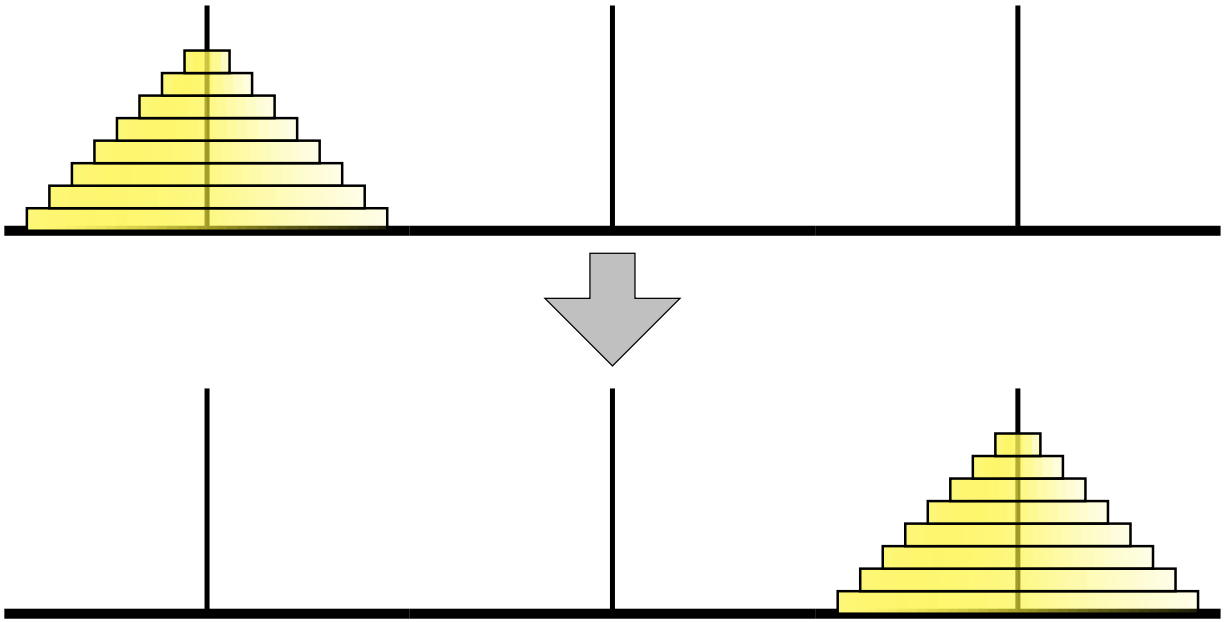
**SelectSort**$(A[1..n])$:
    `If` $(n = 1)$ `return`
    `Find smallest number in` $A$. `Let` $A[i]$ `be smallest number`
    `Swap` $A[1]$ `and` $A[i]$
    **SelectSort**$(A[2..n])$

$T(n)$: time for SelectionSort on an $n$ element array.

$T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$

$T(n) = \Theta(n^2)$.

# Tower of Hanoi



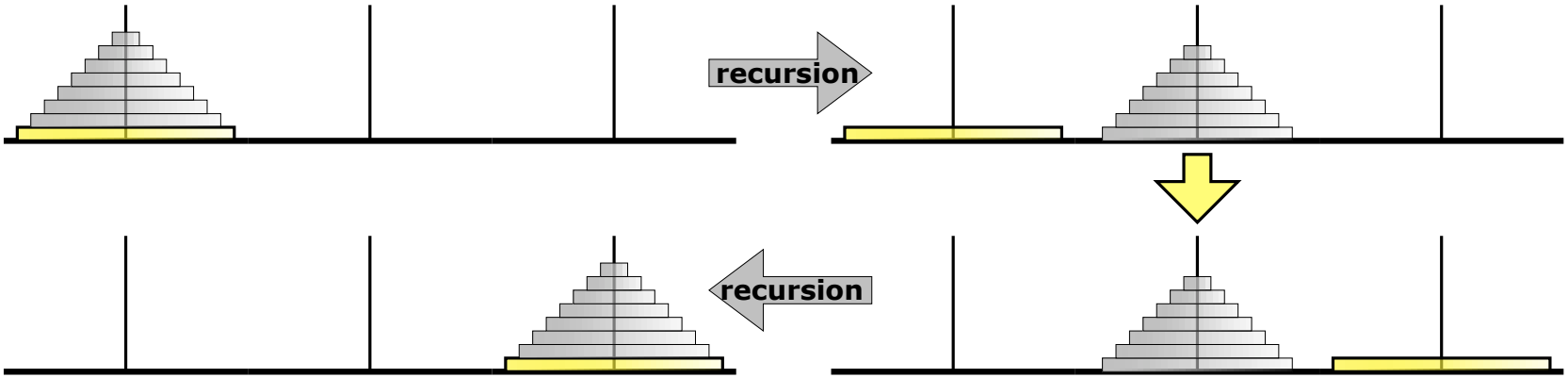
The Tower of Hanoi puzzle

Move stack of $n$ disks from peg 0 to peg 2, one disk at a time.

Rule: cannot put a larger disk on a smaller disk.

Question: what is a strategy and how many moves does it take?

# Tower of Hanoi via Recursion




The Tower of Hanoi algorithm; ignore everything but the bottom disk

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    If (n > 0) do
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    If (n > 0) do
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move $n$ disks via recursive strategy

# Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    If (n > 0) do
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move $n$ disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

# Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2^2 T(n-2)+2+1 \\
&= \ldots \\
&= 2^i T(n-i)+2^{i-1}+2^{i-2}+\ldots+1 \\
&= \ldots \\
&= 2^{n-1}T(1)+2^{n-2}+\ldots+1 \\
&= 2^{n-1}+2^{n-2}+\ldots+1 \\
&= (2^n-1)/(2-1) = 2^n-1
\end{aligned}
$$

# Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $0, 1, 2$

Non-recursive Algorithm 1:

- Always move smallest disk forward if $n$ is even, backward if $n$ is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

# Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $0, 1, 2$

Non-recursive Algorithm 1:

- Always move smallest disk forward if $n$ is even, backward if $n$ is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

Non-recursive Algorithm 2:

- Let $\rho(n)$ be the smallest integer $k$ such that $n/2^k$ is *not* an integer. Example: $\rho(40) = 4$, $\rho(18) = 2$.
- In step $i$ move disk $\rho(i)$ forward if $n - i$ is even and backward if $n - i$ is odd.

# Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $0, 1, 2$

Non-recursive Algorithm 1:

- Always move smallest disk forward if $n$ is even, backward if $n$ is odd.
- Never move the same disk twice in a row.
- Done when no legal move.

Non-recursive Algorithm 2:

- Let $\rho(n)$ be the smallest integer $k$ such that $n/2^k$ is *not* an integer. Example: $\rho(40) = 4$, $\rho(18) = 2$.
- In step $i$ move disk $\rho(i)$ forward if $n - i$ is even and backward if $n - i$ is odd.

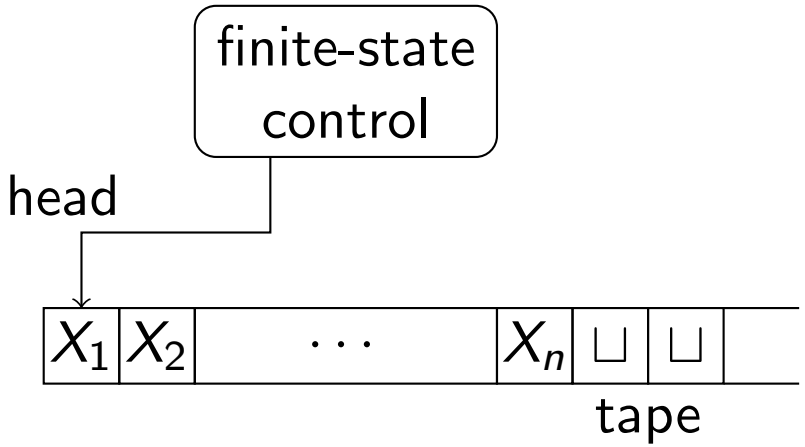
Moves are exactly same as those of recursive algorithm. Prove by induction.

# What model of computation do we use?

# What model of computation do we use?

Turing Machine?

# Turing Machines: Recap




- Infinite tape
- Finite state control
- Input at beginning of tape
- Special tape letter "blank" $\sqcup$
- Head can move only one cell to left or right

# Turing Machines

- Basic unit of data is a bit (or a single character from a finite alphabet)
- Algorithm is the finite control
- Time is number of steps/head moves

# Turing Machines

- Basic unit of data is a bit (or a single character from a finite alphabet)
- Algorithm is the finite control
- Time is number of steps/head moves

**Pros and Cons:**

- theoretically sound, robust and simple model that underpins computational complexity.
- polynomial time equivalent to any reasonable "real" computer: Church-Turing thesis
- too low-level and cumbersome, does not model actual computers for many realistic settings

# "Real" Computers vs Turing Machines

How do computers in use differ from TMs?

# "Real" Computers vs Turing Machines

How do computers in use differ from TMs?

- random access to memory
- pointers
- arithmetic operations (addition, subtraction, multiplication, division) in constant time

# "Real" Computers vs Turing Machines

How do computers in use differ from TMs?

- random access to memory
- pointers
- arithmetic operations (addition, subtraction, multiplication, division) in constant time

How do they do it?

# "Real" Computers vs Turing Machines

How do computers in use differ from TMs?

- random access to memory
- pointers
- arithmetic operations (addition, subtraction, multiplication, division) in constant time

How do they do it?

- basic data type is a word: currently 64 bits
- arithmetic on words are basic instructions of computer
- memory requirements assumed to be $\leq 2^{64}$ which allows for pointers and indirect addressing as well as random access

# Unit-Cost RAM Model

Informal description:

- Basic data type is an integer/floating point number
- Numbers in input fit in a word
- Arithmetic/comparison operations on words take constant time
- Arrays allow random access (constant time to access $A[i]$)
- Pointer based data structures via storing addresses in a word

# Example

Sorting: input is an array of $n$ numbers

- input size is $n$ (ignore the bits in each number)
- comparing two numbers takes $O(1)$ time
- random access to array elements
- addition of indices takes constant time

# Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in many situations so one has to be careful.

- For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two $n$-digit numbers, primality etc.
- Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by $2^k$ where $k$ is word length.
- Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very long numbers from initial numbers and it is not a unit time operation.

# Models used in class

In this course:

- Assume unit-cost RAM by default
- We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.

# Reading Assignment and Homework 0

Chapters 1-3 from the textbook. At least half is prereq material.

Solving recurrences - Jeff Erickson's notes (see link on website)

Homework 0 is due in class on Tuesday, September 1st.