Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 1

General Information

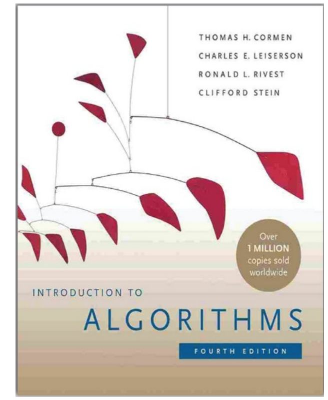
- Instructor: Dr. Monica Nicolescu
  - E-mail: monica@cse.unr.edu
  - Office hours: Tuesday 9:30-11:30am (these may change)
- Teaching assistants:
  - Maryam Ghaed: mghaed@nevada.unr.edu
    - Office hours: Mo, Wed 12:30 pm to 2:00 pm, SEM 340
  - Jeremy Mamaril: jmamaril@nevada.unr.edu
    - Office hours: Mo 10am-1pm, SEM 340
- Office hours scheduling
  - Due to large class size, appointments are highly recommended, use Canvas calendar (most up-to-date)
  - "Walk-in" also possible based on availability

COVID-19 Policy

- Follow UNR policies at
  - https://www.unr.edu/coronavirus/students-x318392
- Contact instructor as soon as:
  - you get a positive COVID-19 test, or
  - you know that you have been exposed to somebody who is COVID-19 positive and have symptoms

Class Policy

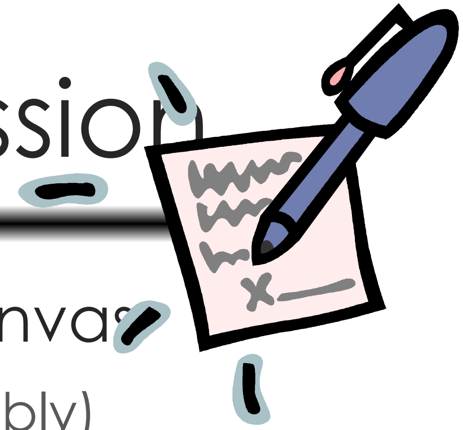
- Grading
  - 7-8 homework assignments (30%)
    - Extra-credit
    - Programming component (C/C++)
  - Two mid-term exams (20% each)
    - Closed books, closed notes
    - February 27, April 2
  - Final exam (30%)
    - Closed books, closed notes
    - May 14, 12:45-2:45pm
  - Extra credit for class participation (up to 3%)



Introduction to Algorithms, 4th edition

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

Homework/Exam Submission



- Homework: two types, all submitted in Canvas
  - Handwritten and/or scanned (please write legibly)
  - Electronic input directly in Canvas
- Homework due at the beginning of the class, late after that
  - 10% penalty for each day of delay, up to 3 days
- Exams:
  - In class, will need identification (student ID card/other) at test time

Academic Dishonesty

University Academic Standards policy: UAM 6,502:

1. *Plagiarism*: defined as: (1) the appropriation of another person's ideas, processes, results, or words without giving appropriate credit; (2) the submission of ideas, processes, results or words not developed by the student specifically for the coursework at hand without the appropriate credit being given; or (3) assisting in the act of plagiarism by allowing one's work to be used as described above.
2. *Cheating*: For purposes of this policy, cheating is defined as: (1) obtaining or providing unauthorized information while executing, completing or in relation to coursework, through verbal, visual or unauthorized use of books, notes, text and other materials; (2) unauthorized collaboration on coursework (3) turning in the same work in more than one class (or when repeating a class), unless permission is received in advance from the instructor; (4) taking an examination for another student, or arranging for another person to take an exam in one's place; (5) altering or changing test answers after submittal for grading; (6) altering or changing grades after grades have been awarded; (7) altering or changing other academic records once these are official; and/or (8) facilitating or permitting any of the above-listed items.

Additional Standards for Code

A student may receive academic and disciplinary sanctions for cheating, plagiarism or other attempts to obtain or earn grades under false pretenses. In addition to University definitions of academic dishonesty, the following rules define plagiarism and cheating for students in computer science and engineering classes:

1. Sharing ideas with other students is fine, but you should write your own code. Never copy or read other students' code, including code from previous years. Cosmetic changes such as rewriting comments, changing variable names, and so forth -- to disguise the fact that your work is copied from someone else is easy to detect and not allowed.

2. It is your responsibility to keep your code private. Sharing your code in public is prohibited and may result in zero credit for the entire assignment.

3. If you find some external code (such as an open-source project) that could be re-used as part of your assignment, you should first contact the instructor to see whether it is fine to reuse it. If the instructor permits it, she/he may announce it to the entire class so all students can use it. If you decide to reuse the external code, you should clearly cite it in comments and keep the original copyright in your code, if applicable.

4. You should be prepared to explain any code you submit, including code copied/modified from external sources.

5. Every student will be asked to include the following statement with every programming assignment: **"This code is my own work, it was written without consulting online resources, a tutor, or code written by other students."**

How to be Successful in Class

- Review prerequisites:
  - Data structures
  - Mathematical background:
    - Algebraic manipulation
    - Logarithms, exponential functions, mathematical series
    - Proofs: induction, proof by contradiction
- Study class material before starting work on homework assignments
  - Work through lecture examples
  - Consult textbook, TAs/instructor office hours
- Study homework & study guides for exams
- Keep pace with material

An algorithm is…

… a step-by-step procedure for solving a problem or accomplishing some end

… a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation

Muhammad ibn Musa **al-Khwarizmi**

Persian scientist (c. 780 – 850)

Father of algebra

Al-Jabr (Arabic): The Compendious Book on Calculation by Completion and Balancing

Timeline of Ancient Algorithms

Multiplication table

up to 6 decimal places

- 1700-2000 BC — Egyptians: multiplying two numbers
- 1600 BC — Babylonians: factorization, square roots
- 300 BC — Euclid’s algorithm: computing greatest common divisor
- 200 BC — Sieve of Eratosthenes: finding prime numbers up to a limit
- 263 AD — Liu Hui: solving systems of linear equations (Gaussian elimination)
- 628 AD — Brahmaguptra: Chakravala method for solving indeterminate quadratic equations
- 820 AD — Al-Khwarizmi: solving linear and quadratic equations
- 825 AD — Al-Khwarizmi: performing arithmetic in place value form
- 85o AD — Al-Kindi: algorithms for breaking encryptions and cyphers
- 1025 AD — Ibn al-Haytham: derived the formula for the sum of fourth powers

Algorithms and Computing

Steps in development of an algorithm

1. Problem definition
2. Development of a model
3. Specification of the algorithm
4. **Designing an algorithm**
5. **Checking the correctness of the algorithm**
6. **Analysis of algorithm**
7. *Implementation of algorithm*
8. *Program testing*
9. *Documentation preparation*

Natural language
Pseudocode
Flowcharts

Why Study Algorithms?

- Necessary in any computer programming problem
  - Improve algorithm **efficiency**: run faster, process more data, do something that would otherwise be impossible
  - **Scalability**: solve problems of significantly large size
  - Technology only improves things by a constant factor
- Compare algorithms
- Algorithms as a field of study
  - Learn about a standard set of algorithms
  - New discoveries arise
  - Numerous application areas
- Learn techniques of **algorithm design** and **analysis**

Applications

- Multimedia
  - CD player, DVD, MP3, JPG, DivX, HDTV
- Internet
  - Packet routing, data retrieval (Google)
- Communication
  - Cell-phones, e-commerce
- Computers
  - Circuit layout, file systems
- Science
  - Human genome
- Transportation
  - Airline crew scheduling, UPS deliveries

Data Formats

- The format in which the data is coded such that it can be recognized, read and used by a program or application
- Ex.: Data Format Description Language (binary & text)
  - Text data types such as strings, numbers, zoned decimals, calendars and Booleans
  - Binary data types such as two's complement integers, BCD, packed decimals, floats, calendars and Booleans
  - Fixed length data and data delimited by text or binary markup
  - Industry standards such as CSV, SWIFT, FIX, HL7, X12, HIPAA, EDIFACT, ISO8583
  - Bit data of arbitrary length
  - Pattern languages for text numbers and calendars, & others
- "*Categories of data structures,*" P. Falley – reading in Canvas
  - Storage structures (arrays, linked structures, hash tables)
  - Process oriented data structures (stacks, queues, priority queues, iterators)
  - Descriptive data structures (collections, sets, linear lists, binary trees, etc.)

Roadmap

- Different classes of problems
  - Sorting
  - Searching
  - String processing
  - Graph problems
  - Geometric problems
  - Numerical problems
- Different design paradigms
  - Divide-and-conquer
  - Incremental
  - Dynamic programming
  - Greedy algorithms
  - Randomized/probabilistic

Analyzing Algorithms

- Predict the amount of resources required:
  - memory: how much space is needed?
  - computational time: how fast the algorithm runs?
- FACT: running time grows with the size of the input
- Input size (number of elements in the input)
  - Size of an array, polynomial degree, # of elements in a matrix, # of bits in the binary representation of the input, vertices and edges in a graph

Def: Running time = the number of primitive operations (steps) executed before termination

- Arithmetic operations (+, -, *), data movement, control, decision making (*if, while*), comparison

Algorithm Efficiency vs. Speed

E.g.: sorting **n** numbers

Sort 10^6 numbers!

Friend's computer = 10^9 instructions/second

Friend's algorithm = $2n^2$ instructions

Your computer = 10^7 instructions/second

Your algorithm = $50n\lg n$ instructions

Your friend = $\frac{2 * (10^6)^2 \; instructions}{10^9 \; instructions/second} = 2000 \; seconds$

You = $\frac{50 * (10^6) lg(10^6) \; instructions}{10^7 \; instructions/second} \approx 100 \; seconds$

20 times better!!

Algorithm Analysis: Example

- *Alg.:* MIN (a[1], …, a[n])

```
m ← a[1];
for i ← 2 to n
    if a[i] < m
        then m ← a[i];
```

- **Running time**:
  - the number of primitive operations (steps) executed before termination

$T(n) = 1$ [first step] + (n) [for loop] + $(n-1)$ [if condition] + $(n-1)$ [the assignment in then] = $3n - 1$

- Order (rate) of growth:
  - The leading term of the formula
  - Expresses the asymptotic behavior of the algorithm

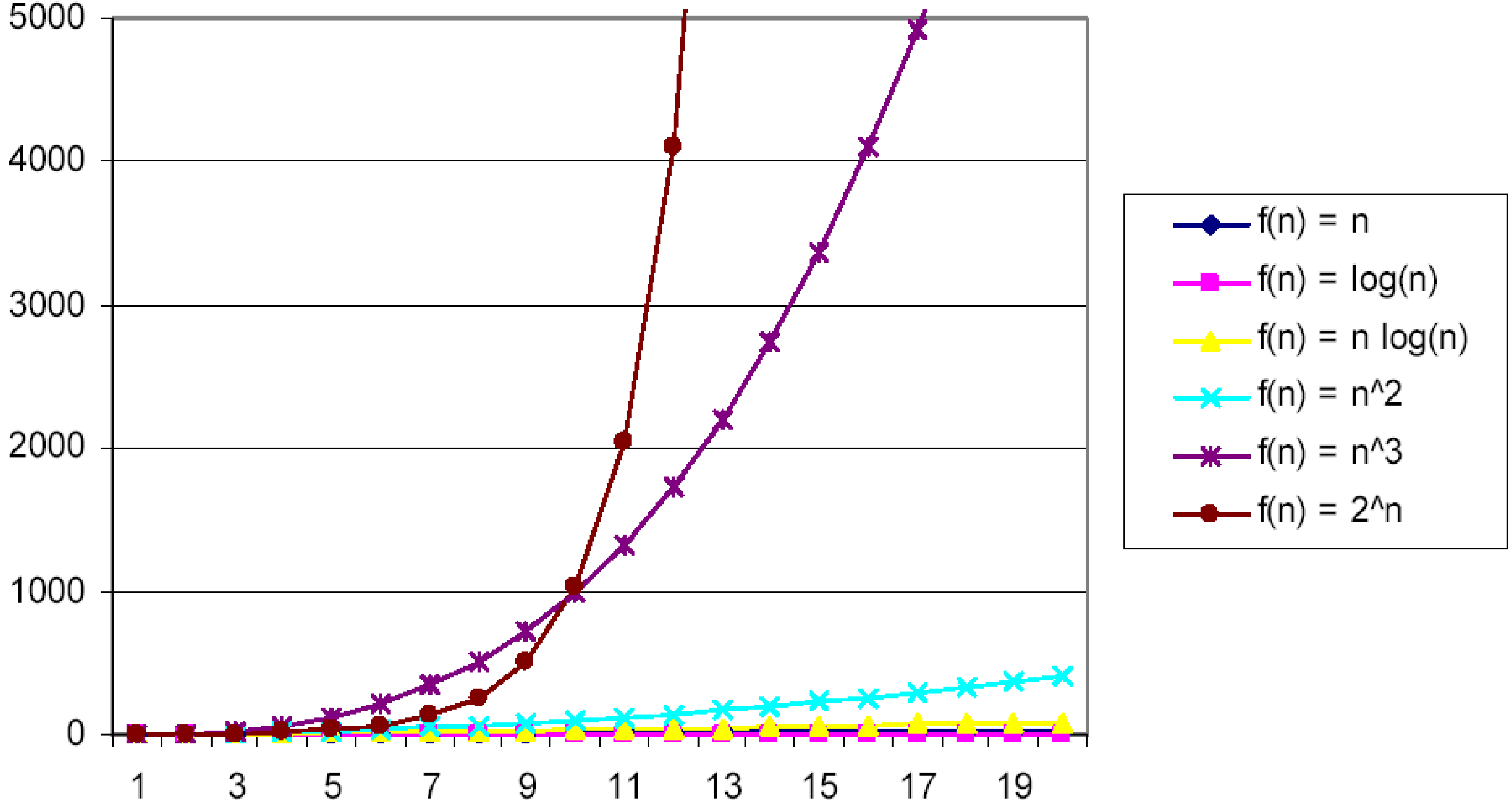
Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

Typical Running Time Functions

- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (double nested loops)
- N^3 (cubic)
  - Processing of triples of data (triple nested loops)
- N^K (polynomial)
- 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

Why Faster Algorithms?



Asymptotic Notations

- A way to describe behavior of functions in the limit
  - Abstracts away low-order terms and constant factors
  - How we indicate running times of algorithms
  - Describe the running time of an algorithm as n grows to ∞
- O notation: asymptotic "less than and equal": f(n) "$\leq$" g(n)
- Ω notation: asymptotic "greater than and equal": f(n) "$\geq$" g(n)
- Θ notation: asymptotic "equality": f(n) "=" g(n)

Asymptotic Notations - Examples

- Θ notation
  - $n^2/2 - n/2 \quad = \Theta(n^2)$
  - $(6n^3 + 1)\lg n/(n + 1) \quad = \Theta(n^2 \lg n)$
  - n vs. $n^2 \quad n \neq \Theta(n^2)$
- Ω notation
  - n^3 vs. $n^2 \quad n^3 = \Omega(n^2)$
  - n vs. $\log n \quad n = \Omega(\log n)$
  - n vs. $n^2 \quad n \neq \Omega(n^2)$
- O notation
  - $2n^2$ vs. $n^3 \quad 2n^2 = O(n^3)$
  - n^2 vs. $n^2 \quad n^2 = O(n^2)$
  - n^3 vs. $n\log n \quad n^3 \neq O(n\lg n)$

Mathematical Induction

- Used to prove a sequence of statements (**S(1)**, **S(2)**, … **S(n)**) indexed by positive integers. **S(n)**: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Proof:
  - **Basis step**: prove that the statement is true for $n = 1$
  - **Inductive step:** assume that **S(n)** is true and prove that **S(n+1)** is true for all $n \geq 1$
- The key to proving mathematical induction is to find case **n** "within" case **n+1**

Recursive Algorithms

- Binary search: for an ordered array A, finds if **x** is in the array A[lo…hi]

Alg.: BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

  return FALSE

mid ← $\lfloor (lo+hi)/2 \rfloor$

if x = A[mid]

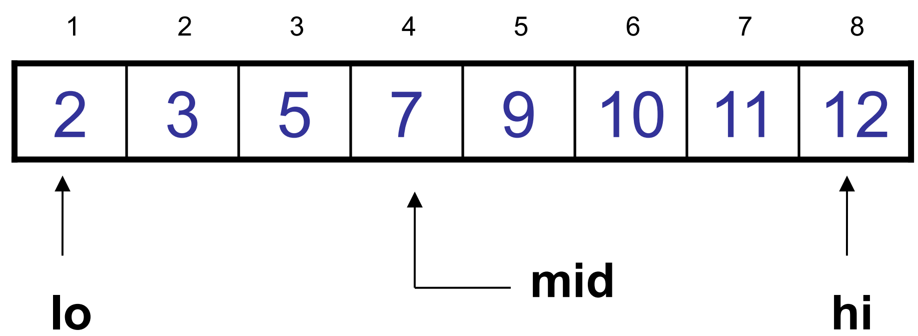
  return TRUE

if (x < A[mid])

  BINARY-SEARCH (A, lo, mid-1, x)

if (x > A[mid])

  BINARY-SEARCH (A, mid+1, hi, x)



Recurrences

Def.: Recurrence = an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases

- E.g.: $T(n) = T(n-1) + n$
- Useful for analyzing recurrent algorithms
- Methods for solving recurrences
  - Iteration method
  - Substitution method
  - Recursion tree method
  - Master method

Sorting – Analysis of Running Time

2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A

Iterative methods:

- Insertion sort
- Bubble sort
- Selection sort

Divide and conquer

- Merge sort
- Quicksort

Non-comparison methods

- Counting sort
- Radix sort
- Bucket sort

Types of Analysis

- Worst case (e.g. cards reversely ordered)
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case (e.g., cards already ordered)
  - Input is the one for which the algorithm runs the fastest
- Average case (general case)
  - Provides a prediction about the running time
  - Assumes that the input is random

Specialized Data Structures

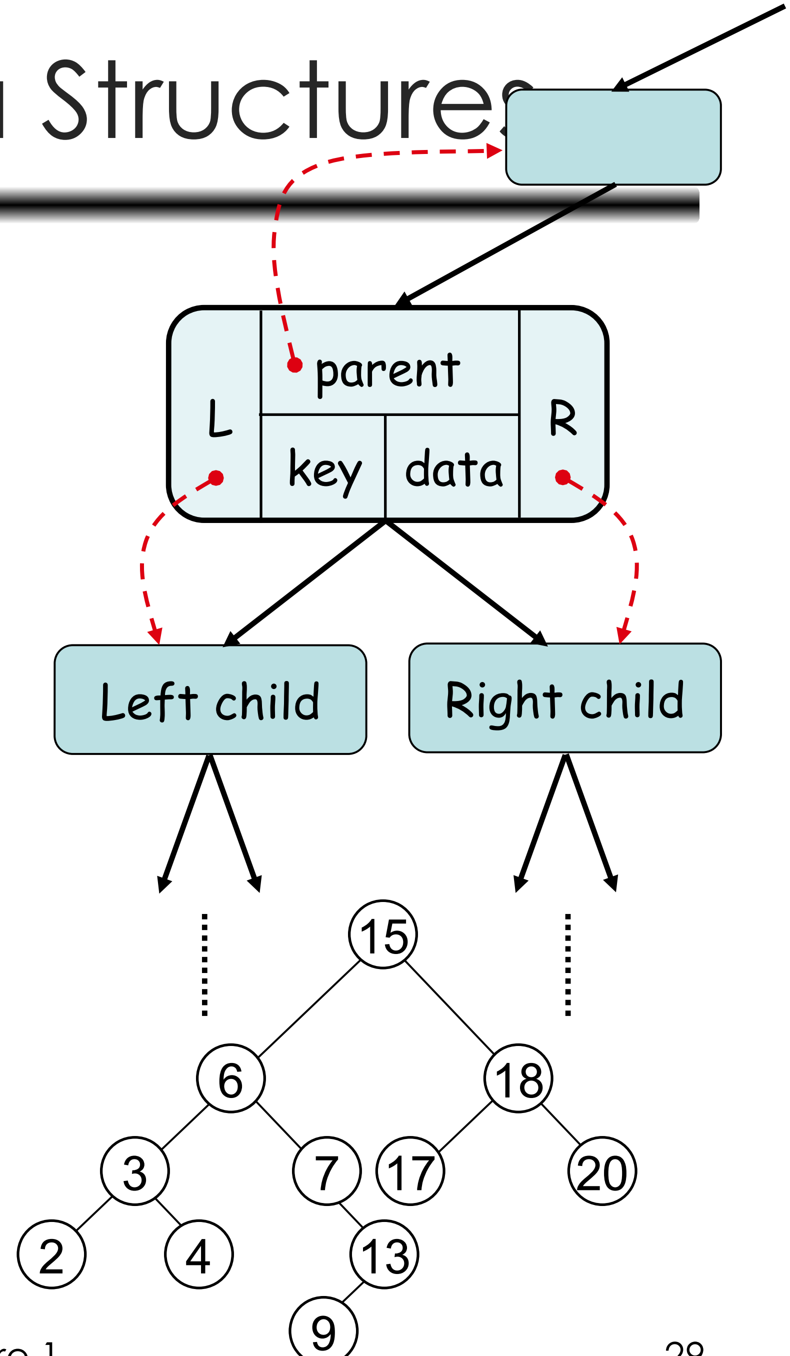
Problem:

- Keeping track of customer account information at a bank or flight reservations
- This applications requires fast search, insert/delete, sort

Solution: binary search trees

- If **y** is in left subtree of **x**, then **key [y] ≤ key [x]**
- If **y** is in right subtree of **x**, then **key [y] ≥ key [x]**

- Red-black trees, interval trees, OS-trees



Dynamic Programming

- An algorithm design technique (like divide and conquer)
  - Richard Bellman, optimizing decision processes
  - Applicable to problems with overlapping subproblems

E.g.: Fibonacci numbers:

- Recurrence: $F(n) = F(n-1) + F(n-2)$
- Boundary conditions: $F(1) = 0$, $F(2) = 1$
- Compute: $F(5) = 3$, $F(3) = 1$, $F(4) = 2$

- Solution: store the solutions to subproblems in a table
- Applications:
  - Assembly line scheduling, matrix chain multiplication, longest common sequence of two strings, 0-1 Knapsack problem

Greedy Algorithms

- Problem
  - Schedule the largest possible set of non-overlapping activities for SEM 234

	Start	End	Activity	
1	8:00am	9:15am	Numerical methods class	✓
2	8:30am	10:30am	Movie presentation (refreshments served)	
3	9:20am	11:00am	Data structures class	✓
4	10:00am	noon	Programming club mtg. (Pizza provided)	
5	11:30am	1:00pm	Computer graphics class	✓
6	1:05pm	2:15pm	Analysis of algorithms class	✓
7	2:30pm	3:00pm	Computer security class	✓
8	noon	4:00pm	Computer games contest (refreshments served)	
9	4:00pm	5:30pm	Operating systems class	✓

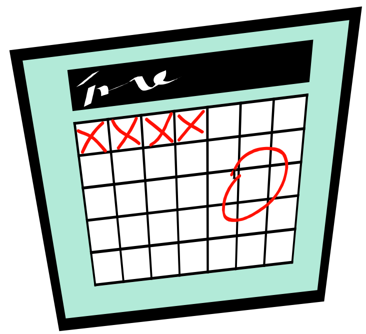
Greedy Algorithms

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems
- **Idea:** When we have a choice to make, make the one that looks best right now
  - Make a locally optimal choice in hope of getting a globally optimal solution
- Greedy algorithms don't always yield an optimal solution
- Applications:
  - Activity selection, fractional knapsack, Huffman codes

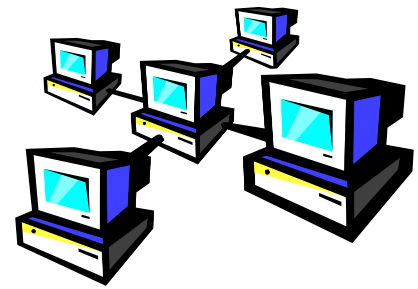
Graphs

- Applications that involve not only a set of items, but also the connections between them

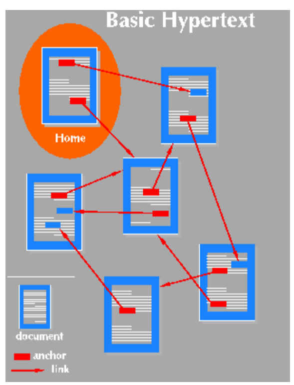
Maps



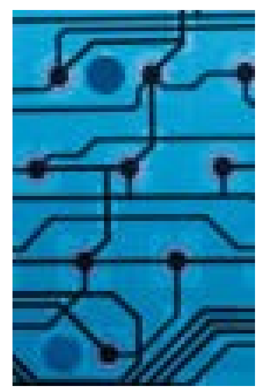
Schedules



Computer networks



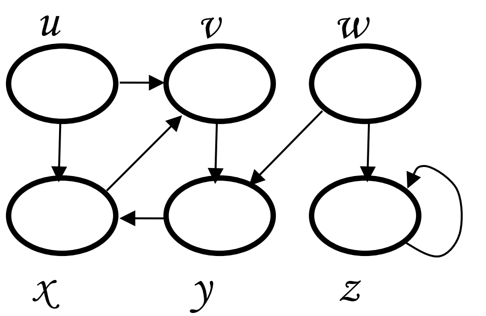
Hypertext



Circuits

Searching in Graphs

- **Graph searching** = systematically follow the edges of the graph so as to visit the vertices of the graph
- Two basic graph methods:
  - Breadth-first search
  - Depth-first search
  - The difference between them is in the order in which they explore the unvisited edges of the graph



- Graph algorithms are typically elaborations of the basic graph-searching algorithms

Strongly Connected Components

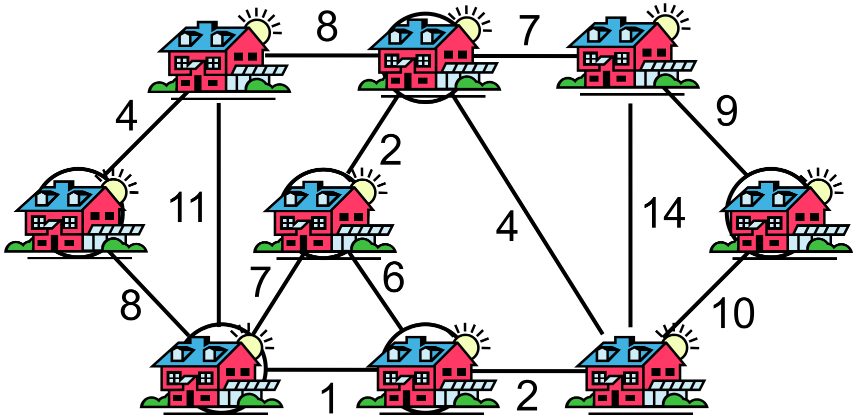
- Read in a 2D image and find regions of pixels that have the same color

Original

Labeled

Minimum Spanning Trees

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight w(u, v)** on each edge **(u, v)** ∈ E



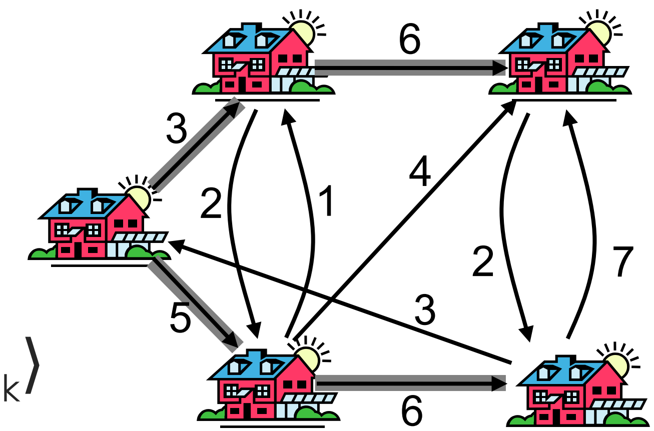
Find T ⊆ E such that:

1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is minimized

Algorithms: Kruskal and Prim

Shortest Path Problems

- **Input:**
  - Directed graph $G = (V, E)$
  - Weight function $w : E \rightarrow \mathbf{R}$



- **Weight of path** $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- **Shortest-path weight** from **u** to **v**:

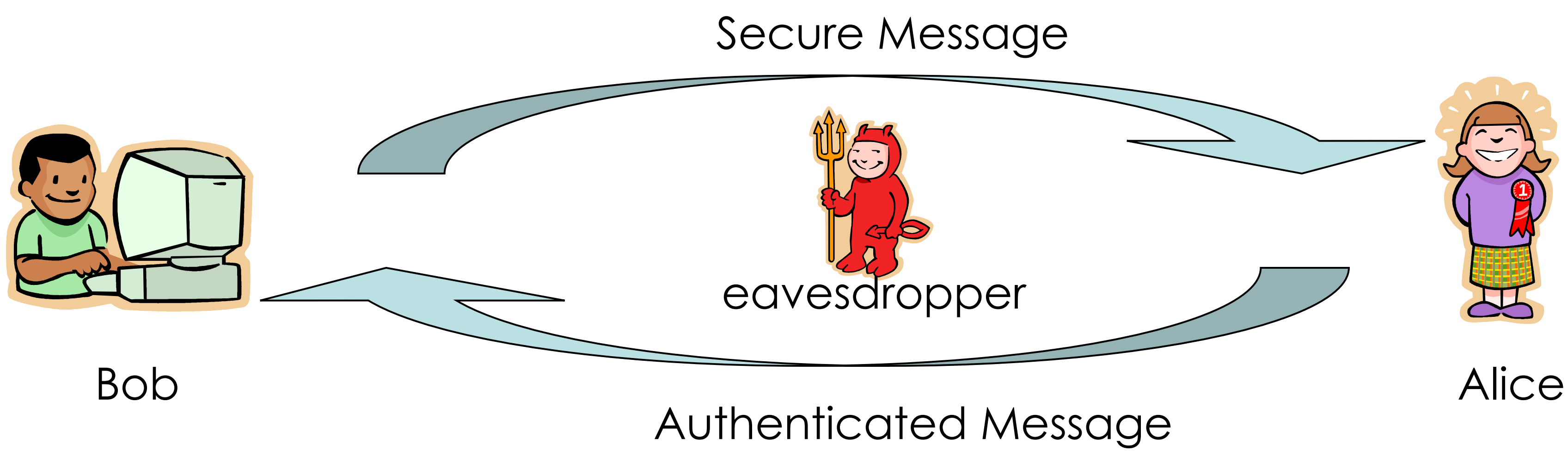
$$\delta(u, v) = \begin{cases} \min\{ w(p) : u \overset{p}{\rightsquigarrow} v \} & \text{if there exists a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Variants of Shortest Paths

- **Single-source shortest path (**Bellman-Ford, DAG shortest paths, Disjkstra**)**
  - $G = (V, E) \Rightarrow$ find a shortest path from a given source vertex s to each vertex $v \in V$
- **Single-destination shortest path**
  - Find a shortest path to a given destination vertex t from each vertex v
  - Reverse the direction of each edge $\Rightarrow$ single-source
- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
  - Solve the single-source problem
- **All-pairs shortest-paths (**Matrix multiplication, Floyd-Warshall**)**
  - Find a shortest path from u to v for every pair of vertices u and v

Number Theoretic Algorithms

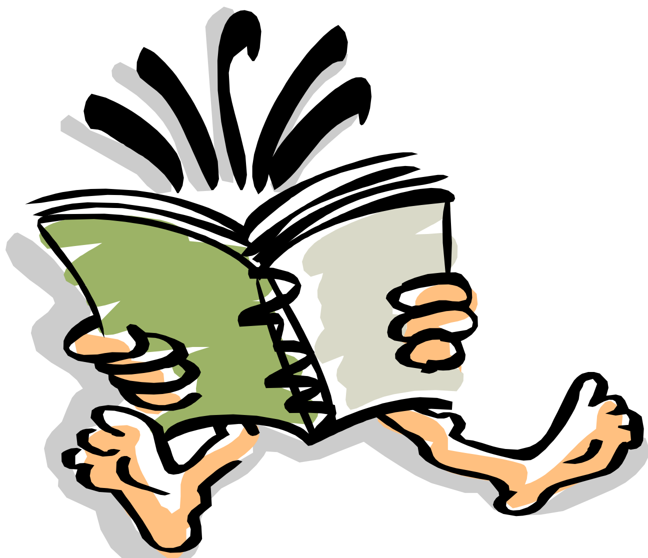
- Secured communication: RSA public-key cryptosystem
  - Easy to find large primes
  - Hard to factor the product of large primes



NP-Completeness

- Not all problems can be solved in polynomial time
  - Some problems cannot be solved by any computer no matter how much time is provided (Turing's Halting problem) – such problems are called **undecidable**
  - Some problems can be solved but not in $O(n^k)$
- Can we tell if a problem can be solved?
  - NP, NP-complete, NP-hard
- Approximation algorithms

Readings



- Chapter 1
- Appendix A
- "*Categories of data structures,*" P. Falley