CSC 611: Analysis of Algorithms

Lecture 1

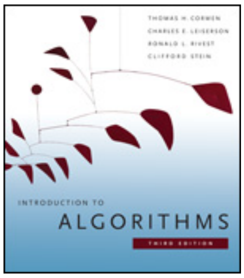
Introduction to Algorithms

General Information

- Instructor: Haidar Harmanani
  - E-mail: haidar@lau.edu.lb
  - Office hours: Tuesday 3 pm-5 pm, or by request
  - Office: Block A 810
- Class webpage:
  - http://Harmanani.github.io/csc611.html

Class Policy

- Grading
  - 5-10 homeworks (30%)
    - Extra-credit
    - Programming component
  - Mid-term exam (30%)
    - Closed books, closed notes
  - Final exam (40%)
    - Closed books, closed notes



Introduction to Algorithms,

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

Why Study Algorithms?

- Necessary in any computer programming problem
  - Improve algorithm efficiency: run faster, process more data, do something that would otherwise be impossible
  - 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

Roadmap

- Different 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 = $\dfrac{2 * (10^6)^2 \text{instructions}}{10^9 \text{instructions/second}} = 2000\text{seconds}$

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

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

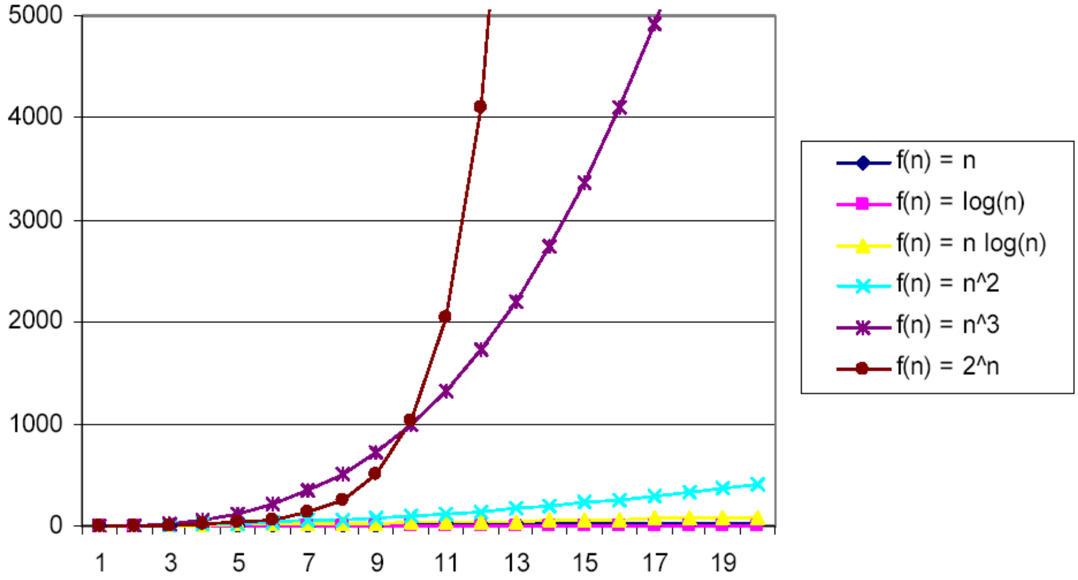
Typical Running Time Functions

- 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": $f(n)$ "≤" $g(n)$
- **Ω** notation: asymptotic "greater than and equal": $f(n)$ "≥" $g(n)$
- Θ notation: asymptotic "equality": $f(n)$ "=" $g(n)$

Asymptotic Notations - Examples

- Θ notation
  - $n^2/2 - n/2 \quad = \Theta(n^2)$
  - $(6n^3 + 1)\lg n/(n + 1) = \Theta(n^2 \lg n)$
  - 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

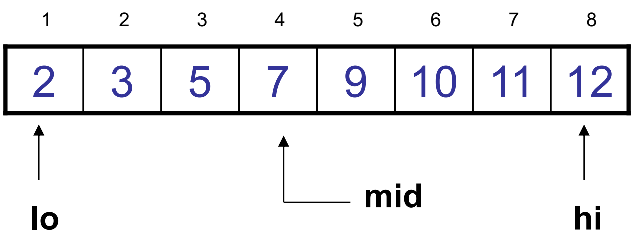
- Used to prove a sequence of statements (S(1), S(2), … S(n)) indexed by positive integers. S(n): $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Proof:
  - **Basis step**: prove that the statement is true for **n = 1**
  - **Inductive step:** assume that S(**n**) is true and prove that S(**n+1**) is true for all **n ≥ 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 ← ⌊(lo+hi)/2⌋
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)



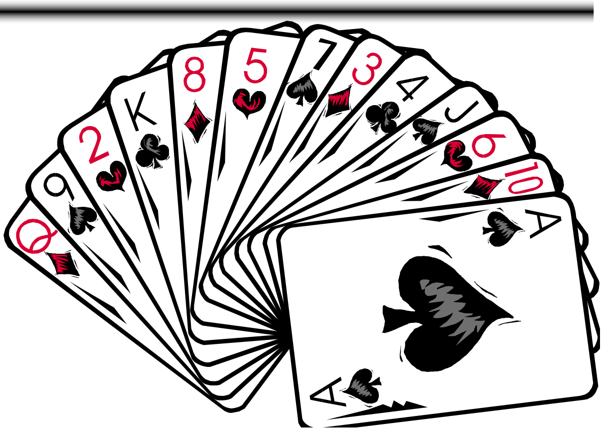
Recurrences

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

Iterative methods:
- Insertion sort
- Bubble sort
- Selection sort



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

Divide and conquer
- Merge sort
- Quicksort

Non-comparison methods
- Counting sort
- Radix sort
- Bucket sort

Types of Analysis

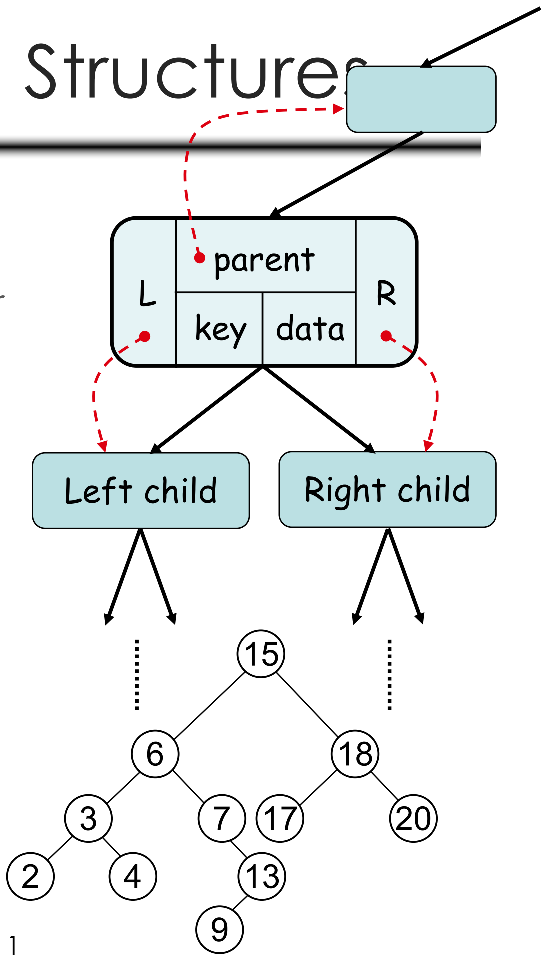
- Worst case (e.g. cards reversely ordered)
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case (e.g., cards already ordered)
  - Input is the one for which the algorithm runs the fastest
- Average case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

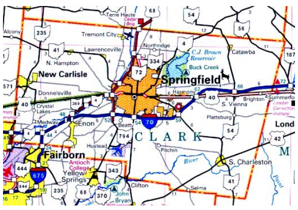
	Start	End	Activity	
1	8:00am	9:15am	Numerical methods class	✓
2	8:30am	10:30am	Movie presentation (refreshments served)	
3	9:20am	11:00am	Data structures class	✓
4	10:00am	noon	Programming club mtg. (Pizza provided)	
5	11:30am	1:00pm	Computer graphics class	✓
6	1:05pm	2:15pm	Analysis of algorithms class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

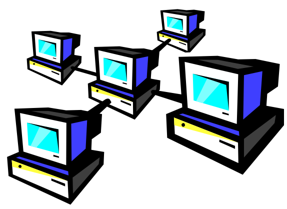
Graphs

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

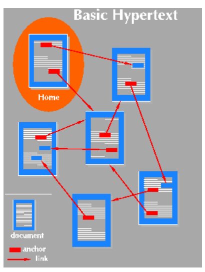


Maps

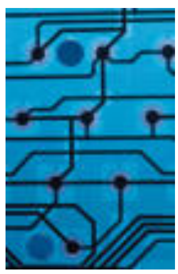
Schedules



Computer networks



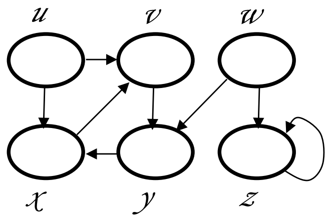
Hypertext



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

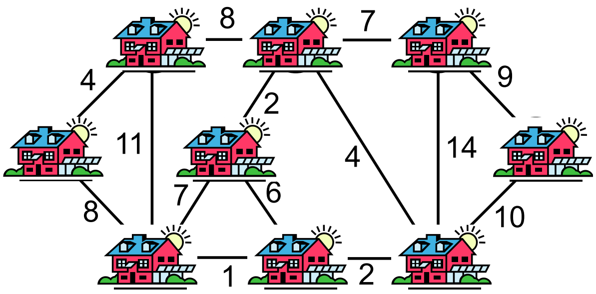
Labeled

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

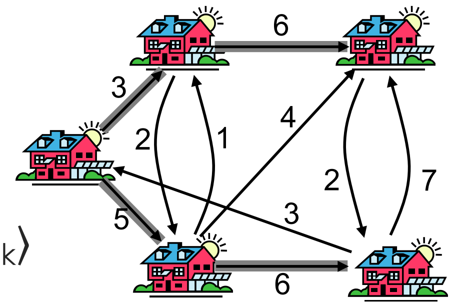


Algorithms: Kruskal and Prim

Shortest Path Problems

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**
- **Weight of path** p = ⟨$v_0, v_1, \ldots, v_k$⟩

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



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

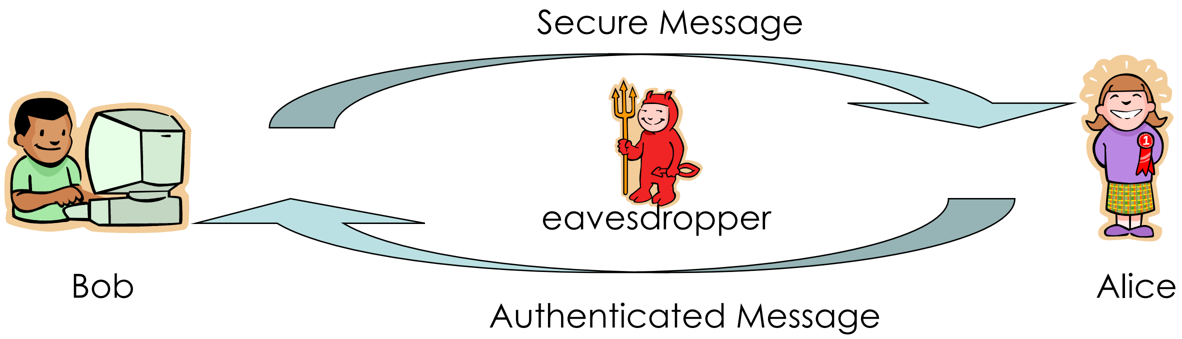
$$\delta(u, v) = \min \begin{cases} w(p) : u \overset{p}{\leadsto} v & \text{if there exists a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Variants of Shortest Paths

- **Single-source shortest path (Bellman-Ford, DAG shortest paths, Disjkstra)**
  - G = (V, E) ⇒ find a shortest path from a given source vertex **s** to each vertex **v** ∈ 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 ⇒ 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

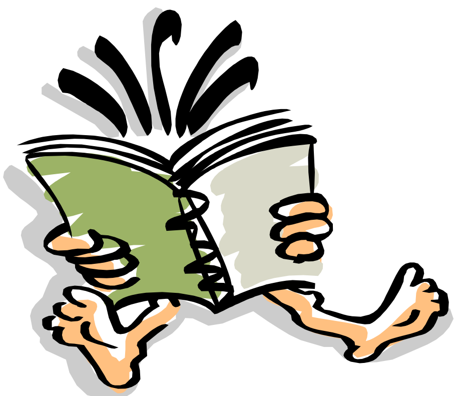
- Secured communication: RSA public-key cryptosystem
  - Easy to find large primes
  - Hard to factor the product of large primes



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