June 7th, 2019

# ● Overview of Algorithms

- ○ A step-by-step procedure to solve a problem
- ○ Every program is the instantiation of some algorithm
  - ■ A Canonical Example: Sorting
  - ■ An algorithm solves a general, well-specified problem
    - ● Given a sequence of n keys, a1, . . . , an as input, produce an output reordering (i.e., a permutation) b1, . . . , bn of the keys so that b1 ≤ b2 ≤,...,bn.
    - ● The problem has specific instances
    - ● [Dopey, Happy, Grumpy] or [3, 5, 7, 1, 2, 3]
  - ■ An algorithm takes any possible instance and produces output that has the desired properties
    - ● e.g., insertion sort, quicksort, heapsort …
  - ■ It is hard to design algorithms that are:
    - ● • correct
    - ● • efficient
    - ● • (easily) implementable
  - ■ To do so effectively, we need to know about:
    - ● • algorithm design and modeling techniques
    - ● • existing resources (i.e., don't reinvent the wheel!)
  - ■ Algorithm Correctness
    - ● It produces the correct output on every possible input (!)
- ○ The Tour Finding Problem
  - ■ Given a set of n points in the plane, what is the shortest tour that visits each point and returns to the beginning?
  - ■ An Application
    - ● Consider a robot arm that solders contact points on a circuit board; we want to minimize the movement of the robot arm.
- ○ Finding a Tour: Nearest Neighbor
  - ■ Start by visiting any point
  - ■ While all points are not visited, choose an unvisited point closest to the last visited point and visit it, Return to the first point
  - ■ Nearest Neighbor Counter Example
- ○ So How Do We Prove Correctness?
  - ■ Formal methods (even automated tools) available for proving
  - ■ Informal reasoning - in this course
    - ● Check counter-examples to proposed algorithms as an important part of the design process
- ○ Algorithm Efficiency
- ○ Measuring Efficiency
  - ■ We generally focus on machine-independent measures
  - ■ We analyze a "pseudocode" version of the algorithm
  - ■ We assume an idealized model of a machine in which one instruction takes one unit of time
  - ■ **"Big-O" notation**
  - ■ Analyze the order of magnitude changes in efficiency as the problem size increases
  - ■ We often focus on worst-case performance
  - ■ This is safe, the worst case often occurs frequently, and the average case is often just as bad
- ○ Some Caveats
  - ■ There's no point in finding the fastest algorithm for parts of the program that are not bottlenecks.
  - ■ If the program will only be run a few times or time is not an issue (e.g., the program will be left to run overnight), the there's no real point in finding the fastest algorithm. Making it Easier
- ○ Cast your application in terms of well-studied data structures
- ○ Some Important Problem Types
  - ■ Sorting (a set of items)
  - ■ Searching (among a set of items)
  - ■ String processing (text, bit strings, gene sequences)
  - ■ Graphs (modeling objects and their relationships)
  - ■ Combinatorial (find desired permutation, combination, or subset)
  - ■ Geometric (graphics, imaging, robotics)
  - ■ Numerical (continuous math, solving equations, evaluating functions)
- ○ Algorithm Design Techniques
  - ■ Brute force and exhaustive search - follow definition / try all possibilities
  - ■ Divide and conquer - break problem into distinct subproblems
  - ■ Transformation - convert one problem into another one
  - ■ Dynamic programming • break problem into overlapping subproblems
  - ■ Greedy - repeatedly do what is the best right now
  - ■ Iterative improvement - repeatedly improve current solution
  - ■ Randomization - use random numbers

June 10th, 2019

# ● A Review of Discrete Mathematics

- ○ Set
  - ■ DEFINITIONS
    - ● a set is a collection of distinguishable objects, called members or elements
    - ● If x is an element of a set S, we write $x \in S$
    - ● If x is not an element of set S, we write $x \not\in S$
    - ● two sets are equal (i.e., A = B) if they contain exactly the same elements
    - ● some special sets:
      - ○ $\varnothing$ is the set with no elements

        - Z is the set of integer elements
        - R is the set of real number elements
        - N is the set of natural number elements
    - Set Operators
        - subset: if $x \in A$ implies $x \in B$, then $A \subseteq B$
        - Proper subset: if $A \subseteq B$ and $A \neq B$ then $A \subset B$
        - Intersection: $A \cap B = \{x | x \in A and x \in B\}$
        - Union: $A \cup B = \{x | x \in A \text{ or } x \in B\}$
        - Difference: $A - B = \{x | x \in A \text{ and } x \notin B\}$
- Relations
    - Relation Definitions
        - A binary relation R on two sets A and B is a subset of the Cartesian product $A \times B$.
            - If $(a, b) \in R$, we sometimes write a R b.
        - Consider the relations "=", "<", and "≤" for each of the following.
        - Properties of Relations
            - Reflexive: $R \subseteq A \times A$ is reflexive if $aRa$ for all $a \in A$
            - Symmetric: R is symmetric if a R b implies b R a for all $a, b \in A$
            - Transitive: R is transitive if a R b and b R c imply a R c for all $a,b,c \in A$
            - Antisymmetric: R is antisymmetric if a R b and b R a imply a = b.
    - Equivalence relation, equivalence class
        - A relation that is reflexive, symmetric, and transitive is an equivalence relation. If R is an equivalence relation on set A, then for $a \in A$, the equivalence class of a is the set $[a] = \{b \in A \mid a R b\}$.
        - EX. Consider $R = \{ (a, b) \mid a, b \in N \text{ and } a + b \text{ is an even number} \}$.
            - Is it reflexive? Is it symmetric? Is it transitive?
    - Partial Order
        - A relation that is reflexive, antisymmetric, and transitive is a partial order.
    - Total Order
        - A partial order on A is a total order if for all $a, b \in A$, a R b or b R a hold.
- Functions (definitions
    - Given sets A and B, a function f is a *binary relation* on A  B s.t. All $a \in A$, there exists exactly one $b \in B$ s.t. $(a, b) \in f$ .
    - A is the domain of f (a is an argument to the function)
    - B is the co-domain of f (b is the value of the function)
    - We often write functions as:
        - $f : A \rightarrow B$
        - If $(a,b) \in f$, we write b=f(a)
    - f assigns an element of B to each element of A. No element of A is assigned to two different elements of B, but the same element of B can be assigned to two different elements of A.
    - A **finite sequence** is a function whose domain is
        - $\{0, 1, \ldots, n - 1\}$, often written as $\langle f(0), f(1), \ldots, f(n - 1)\rangle$
    - An **infinite sequence** is a function whose domain is the set of *N natural numbers* ({0, 1,...}).
    - When the domain of f is a Cartesian product, e.g.,
        - $A = A1 \times A2 \times ... \times An$, we write f(a1,a2,...,an) instead of f((a1,a2,...,an))
    - We call each ai an argument of f even though the argument is really the n-tuple (a1, a2,..., an)
        If $f : A \rightarrow B$ is a function and b=f(a), then we say that b is the image of a under f.
    - The **range** of f is the image of its domain (i.e., f (A)).
    - A function is a **surjection** if its range is its codomain. (This is sometimes referred to as mapping A onto B.)
    - $f(n) = \lfloor n/2 \rfloor$ is a surjective function from N to N
    - f (n) = 2n is not a surjective function from N to N
    - f (n) = 2n is a surjective function from N to the even numbers
    - 
    - A function is an injection if distinct arguments ito f produce distinct values,
        - i.e., $a \neq a'$ implies $f(a) \neq f(a')$. (This is sometimes referred to as a one-to-one function.)
    - $f(n) = \lfloor n/2 \rfloor$ is not an injective function from N to N • f (n) = 2n is an injective function from N to N
    - A function is a bijection if it is both injective and surjective. (This is sometimes referred to a one-to-one correspondence.)
- Graphs
    - Types of Graphs
        - A directed graph (or digraph) G is a pair (V,E) where V is a finite set (of "vertices") and E (the "edges") is a subset of $V \times V$ .
        - An undirected graph G is a pair (V,E) where V is a finite set (of "vertices") and E (the "edges") is a set of unordered pairs of edges {u, v}, where u != v.
    - Properties of Edges
        - If (u,v) is an edge in a digraph G, then (u,v) is *incident from or leaves* u and is *incident to or enters* v.
        - If (u,v) is an edge in an undirected graph G, then (u,v) is *incident to both* u and v.
        - In both cases, v is **adjacent** to u; in a digraph adjacency is not necessarily symmetric.
        - The **degree** of a vertex in an undirected graph is the number of edges incident to it (which is the same as the number of vertices adjacent to it).
        - The **out-degree** of a vertex in a digraph is the number of edges leaving it.
        - The **in-degree** of a vertex in a digraph is the number of edges entering it.
    - Paths in Graphs
        - A path from u to v is a sequence of vertices $\langle v0,v1,...,vk\rangle$ s.t. u = v0, v = vk, and $(vi-1,vi) \in E$ for i = 1,2,...,k.
            - The length of a path is the number of edges
            - The path contains the vertices v0,v1,...,vk and the edges (v0, v1), (v1, v2), . . . , (vk−1, vk )
            - V is reachable from u if there is a path from u to v
            - A path is simple if all its vertices are distinct
            - A **subpath** of a path p is any $\langle vi,vi+1,...,vj\rangle$ where $0 \leq i \leq j \leq k$. (p is a subpath of itself)
            - In a digraph, $\langle v0,v1,...,vk\rangle$ is **a cycle** if v0 = vk and $k \geq 1$.
                - A cycle is **simple** if all vertices except v0 = vk are distinct.
            - In an undirected graph, a path $\langle v0,v1,...,vk\rangle$ forms **a cycle** if v0 = vk, $k \geq 3$ and v1,v2,...,vk are distinct.
            - An **acyclic** graph has no cycles.

- Connectivity in Graphs
  - An undirected graph is connected if each pair of vertices is connected by a path.
    - The connected components are the equivalence classes of vertices under the “is reachable from” relation
  - A directed graph is **strongly connected** if every two vertices are reachable from one another
    - The strongly connected components of a digraph are the equivalence classes of vertices under the “are mutually reachable” relation
    - A digraph is strongly connected if it has exactly one strongly connected component
- Graph Isomorphism
  - G = (V , E) is isomorphic to G = (V , E) if there is a 1-to-1 onto function f : V → V such that (u, v) ∈ E if and only if (f (u), f (v)) ∈ E
    - conceptually, we “relabel” G to get G
- Subgraphs and Transformations
  - The graph G’ = (V’ , E’) is a subgraph of G = (V , E) if V’ ⊆ V and E’ ⊆ E
    - Given V’ ⊆ V , the subgraph induced by V is G’ = (V’ , (V’ × V’ ) ∩ E), or, equivalently, E = {(u, v) ∈ E : u, v ∈ V’ }
  - Given an undirected graph G = (V , E), the directed version of G is the graph G’ = (V’ , E’), where (u, v) ∈ E’ if and only if (u, v) ∈ E
    - Conceptually, we introduce two edges for each original edge
  - Given a directed graph G = (V , E), the undirected version of G is the graph G’ = (V’ , E’) where (u, v) ∈ E’ if u != v and (u, v) ∈ E.
    - Conceptually, we remove directionality and self-loops
- Special Graphs
  - complete graph: an undirected graph in which every pair of vertices is adjacent
  - bipartite graph: an undirected graph in which the vertex set can be partitioned into two sets V1 and V2 such that every edge in the graph is of the form (x, y) where x ∈ V1 and y ∈ V2.
  - forest: an **acyclic undirected** graph
  - tree: a **connected, acyclic undirected** graph
  - dag: directed acyclic graph
  - multigraph: like an undirected graph but can **have multiple edges between vertices and self-loops**
  - hypergraph: like an undirected graph, but **each hyperedge can connect an arbitrary number of vertices**

**June 12th, 2019**

- Trees
  - Theorem (Properties of Trees)
  - Let G = (V , E) be an undirected graph. Then the following are equivalent statements:
    - 1. G is a tree.
    - *2. Any two vertices of G are connected by a unique simple path.*
    - *3. G is connected, but if any edge is removed from E, the resulting graph will not be connected.*
    - *4. G is connected and $|E| = |V| - 1$*
    - *5. G is acyclic and $|E| = |V| - 1$*
    - *6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle*
  - Rooted Tree
    - A rooted tree is a tree in which one vertex is distinguished from the others.
      - the distinguished vertex is called the root
      - a vertex in a rooted tree is often called a node
    - Let r be the root of a rooted tree T . For any node x, there is a unique path from r to x.
      - any node y on a path from r to x is an ancestor of x
      - if y is an ancestor of x, then x is a descendant of y
      - every node is its own ancestor and descendant
      - a proper ancestor (descendant) is an ancestor (descendant) that is not the node itself
      - the subtree rooted at x is the tree induced by the descendants of x
    - If the last edge of the path from r to x is (y, x), then y is the parent of x and x is the child of y
      - The **root** is the only node with no parent
      - **siblings**: two nodes that share the same parent
      - **leaf:** a node with no children (aka an external node)
      - **internal** node: a non-leaf node
    - The **number of children** of a node x in a rooted tree T is called the **degree** of x.
      - The length of a path from r to x is called the depth of x.
      - The largest depth of any node in T is the height of T
    - An ordered tree is a rooted tree in which the children at each node are ordered.
  - Binary Tree
    - Binary trees are defined recursively. A binary tree T is a structure defined on a finite set of nodes that either:
      - *1. contains no nodes (we call this empty or null or NIL)*
      - *2. is composed of three disjoint sets of nodes: a root node, a left subtree, and a right subtree*
    - If the left subtree of a binary tree is nonempty, its root is called the left child; similar definition of the right child.
    - A **full binary tree i**s a binary tree in which each node is either a leaf or has degree 2.
    - A binary tree is not just an ordered tree in which each node has degree at most two. ***Left and right children matter.***

# ● Proof

- Definition of proof
  - a statement is either true or false.
  - 1 = 0 is false
  - ∃t : cos(t) = t is true
  - ∀a, b, c, n : (n > 2) ∧ (a^n + b^n = c^n) ⇒ a = b = c = 0 is true (though it’s difficult to prove)
  - some statements may be true or false depending on the values assigned to variables:
  - 3x = 5
  - x2 + y2 − 4xy > 0
- Proofs
  - A mathematical proof is a “convincing” argument expressed in the language of mathematics
  - it should contain enough detail to convince someone with reasonable background in the subject

- Terminology
    - Definition: an unambiguous explanation of terms
    - Proposition: a statement that is claimed to be true
    - Theorem: a major result
    - Lemma: a minor result; often used on the way to proving a theorem
    - Corollary: something that follows from something just proved
    - Axioms: basic assumptions or truths
  - Forms of Theorems
    - A theorem can be reduced to stating “if A then B.” The following are all equivalent:
      - If A is true then B is true
      - A implies B
      - A ⇒ B
      - A only if B
      - A is sufficient for B
      - B is true whenever A is true
- The Forward-Backward Method
  - A good technique to approaching a proof is to work from both directions. Start by first writing both the statements A and B. In the forward direction: “given A, what else do I know?” In the backward direction: “how would I show B?”
    - **EX1** If a right triangle xyz with sides of length x and y and a hypotenuse of length z has area z^2 / 4, then the triangle xyz is isosceles.
      - **A right triangle xyz has area z2/4**
      - A1 xy/2 = z2/4 (area = 1/2 base × height) A2 x2 + y2 = z2 (Pythagorean theorem)
      - A3 (x2 + y2)/4 = xy/2 (substituting for z2) A4 (x2 + y2) = 2xy (multiplying through by 4) A5 x2 − 2xy + y2 = 0 (rearranging)
      - A6 (x − y)2 = 0 (factoring)
      - B2 (x − y) = 0
      - B1 x = y
      - **B triangle xyz is isosceles**

    A Condensed Proof:
      - From the hypothesis and the definition of the area of a triangle,
      - xy/2 = z2/4. By Pythagoras, x2 + y2 = z2. On substituting x2 + y2 for z2, we obtain (x − y)2 = 0. Hence x = y and the triangle is isosceles.
- Proof Tools
  - part of our proof is just algebraic manipulation
  - other pieces also drew upon external information
  - e.g., the definition of isosceles triangle, the theorem stating the area of a triangle, the Pythagorean theorem
  - in general, a proof will draw upon definitions, axioms, and previously proven theorems
  - be careful to avoid a circular proof (i.e., where a step in your proof relies on the theorem you’re trying to prove).
- Truth Table

**Notations**

- $A \Rightarrow B$: “implies”
- $\overline{B} \Rightarrow \overline{A}$: “contrapositive”
- $B \Rightarrow A$: “converse”
- $\overline{A} \Rightarrow \overline{B}$: “inverse”
- $A \Leftrightarrow B$: “equivalence” or “if-and-only-if” or “iff”

| $A$ | $B$ | $\overline{A}$ | $\overline{B}$ | $A \Rightarrow B$ | $\overline{B} \Rightarrow \overline{A}$ | $B \Rightarrow A$ | $\overline{A} \Rightarrow \overline{B}$ | $A \Leftrightarrow B$ |
|---|---|---|---|---|---|---|---|---|
| F | F | T | T | T | T | T | T | T |
| F | T | T | F | T | T | F | F | F |
| T | F | F | T | F | F | T | T | F |
| T | T | F | F | T | T | T | T | T |

- Qualifiers
  - Qualifiers
    - ∃ : there exists an object with a certain property such that something happens
    - ∀ : for all objects with a given property, something happens
  - Specialization
    - x’ has a certain property
    - ∀ x with a certain property, something happens
    - the something happens for x’
  - Choose
    - ∀ x with a certain property, something happens.
    - Let x’ be such that the certain property holds
    - something happens for x’
- **EX2**
  - If s and t are rational and t != 0, then s/t is rational.
    - **A s and t are rational and t = 0**
    - A1 ∃ integers p, q, q = 0 such that s = p/q
    - A2 Let a, b be integers such that b = 0 and s = a/b A3 ∃ integers p, q, q = 0 such that t = p/q
    - A4 Let c, d be integers such that d = 0 and t = c/d A5 t = 0 ⇒ c = 0
    - A6 t = ca//db = adbc
    - A7 Let p = ad and q = bc
    - B2 bc = 0, t = adbc
    - B1 ∃ integers p, q, q = 0 such that s/t = p/q
    - **B s/t is rational**
  - Condensed proof
    - Let a, b be integers such that s = a/b (b = 0). Such integers must exist because s is rational. Similarly, let c, d be integers such that t = c/d (d = 0). Since t = 0, it must be true that c = 0. Then, substituting, s/t = (a/d)/(c/d) = ad/bc.
    - bc = 0 (since both b and c are nonzero). Therefore, s/t is rational because there exist integers p, q such that s/t is p/q.
- **EX3**
  - Def: f : S → T is onto iff ∀t ∈ T , ∃s ∈ S : f (s) = t

- Def: Let $f : X \to Y$ and $g : Y \to Z$ be functions,
  then $g \bullet f : X \to Z$ is the function such that $(g \bullet f)(x) = g(f(x))$
- Proposition: if $f : X \to Y$ is onto and $g : Y \to Z$ is onto, then $g \bullet f : X \to Z$ is onto.
  - A $f : X \to Y$ , $g : Y \to Z$ are onto
  - A1 Let $c \in Z$
  - A2 $\forall z \in Z$ , $\exists y \in Y$ such that $g(y) = z$
  - A3 $\exists y \in Y$ such that $g(y) = c$
  - A4 Let b be such a y: $b \in Y$ , $g(b) = c$
  - A5 $\forall y \in Y$ , $\exists x \in X$ such that $f(x) = y$
  - A6 $\exists x \in X$ such that $f(x) = b$
  - A7 Let a be such an x: $a \in X$ , $f(a) = b$
  - A8 Let x of [B2] be a
  - A9 $(g \bullet f)(a) = g(f(a)) = g(b) = c$
  - B3 $(g \bullet f)(a) = c$
  - B2 $\exists x \in X$ such that $(g \bullet f)(x) = c$
  - B1 $\forall z \in Z$ , $\exists x \in X$ such that $(g \bullet f)(x) = z$
  - B $g \bullet f : X \to Z$ is onto
  - QED (quod erat demonstrandum)
- Condensed proof
  - For any $c \in Z$ , we can find a $b \in Y$ such that $g(b) = c$. (Such a b must exist because g is onto.) Similarly, let $a \in X$ be such that $f(a) = b$ (again, a must exist because f is onto). Then, given any selected $c \in Z$ , $(g \bullet f)(a) = c$, i.e., some $a \in X$ can be found to make the claim true. Therefore $g \bullet f : X \to Z$ is onto.

- Methodologies
  - Proof by Contradiction
    - We assume that the negation of our proposition is true and show that it leads to a contradictory statement.
    - **EX4**
    - Theorem: There are infinitely many prime numbers.
    - Proof:
      - Suppose there is a finite number of prime numbers. Then you can list them in order: p1, p2, . . . , pn. Consider the number q = p1p2 . . . pn + 1. The number q is either prime or composite. If we divide any of the listed primes pi into q, there would be a remainder of 1. Thus q cannot be composite. Therefore q is a prime number that is not listed among the primes listed above, contradicting the assumption that our list p1, p2, . . . , pn lists all of the prime numbers.
  - Proof by Induction
    - Three Steps to an Inductive Proof
      - Start with verifying the base case.
      - Then assume the nth case.
      - And use that to prove the (n + 1)st case.
    - **EX5**
    - Prove that $0 + 1 + 2 + \cdots + n = n(n+1)/2$
      - Base case: show it's true for n = 0: 0 = 0(0+1)
      - Inductive step: show that if it holds for n then it holds for n + 1. That is, use: $0 + 1 + 2 + \cdots + n = n(n+1)/2$ to show that:
      - $0 + 1 + 2 + \cdots + (n + 1) = (n+1)((n+1)+1)/2$
      - Substituting in the right hand side of the equation for the sum to n to most of the left hand side of the equation for the sum to n + 1 gives us:
        $n(n+1)/2 + (n+1) = (n+1)((n+1)+1)/2$ which is true.
    - EX6
    - Prove that the sum of the first n odd positive integers is n2.
      - Base case: the sum of the first one odd positive integers is 1^2. This is true since the sum of the first odd positive integer is 1.
      - Inductive step: show that if it holds for n, then it holds for n + 1. If the proposition is true for n, then
      - $1 + 3 + 5 + \cdots + (2n - 1) = n2$. Then we must show that $1 + 3 + 5 + \cdots + (2n - 1) + (2n + 1) = (n + 1)2$. We can prove this algebraically.
    - EX7
    - Prove that if S is a finite set with n elements, then S has 2^n subsets.
      - Base case: a set S of size 0 has one subset (the empty set); 2^0 = 1.
      - Inductive step: assume that every set with n elements has 2^n subsets. Prove that by adding one element to the set S, we increase the number of subsets to 2^(n+1). Let T be a set with n + 1 elements. Then it is possible to express $T = S \cup \{a\}$ where a is one of the elements of T and $S = T - \{a\}$. The subsets of T can be obtained by the following. For each subset X of S, there are exactly two subsets of T , namely X and $X \cup \{a\}$. Since there are 2^n subsets of S, there are $2 \times 2^n$ subsets of T , which is 2^(n+1). 19/20

June 14, 2019

- Algorithms
  - The Informal Problem
    - Consider the problem of optimally matching a set of applicants to a set of open positions.
      - Applicants to summer internships
      - • Applicants to graduate school
      - • Medical school graduate applicants to residency programs
      - • Eligible males wanting to marry eligible females
    - Actually, it seems like it should be easy. Why is it a hard problem, practically? Let's think about how to solve the problem if we have perfect information…
  - Stability and Instability
    - Given a set of preferences among hospitals and medical students, define a self-reinforcing admissions process.
    - Unstable Pair - Applicant x and hospital y are unstable if:
      - x prefers y to its assigned hospital
      - y prefers x to one of its admitted students
    - Stable Assignment - A stable assignment is one with no unstable pairs.

        - This is a natural and desirable condition.
        - Individual self-interest will prevent any applicant/hospital deal being made.
- Formulating the Problem
    - The Problem, Formally
        - Consider a set M = {m1, . . . , mn} of n men and a set W = {w1, . . . , wn} of n women.
            - • A matching S is a set of ordered pairs, each from M × W , s.t. each member of M and each member of W appears in at most one pair in S.
            - • A perfect matching S is a matching s.t. each member of M and each member of W appears in exactly one pair in S.
            - • Each man m ∈ M ranks all of the women; m prefers w to w if m ranks w higher than w. We refer to the ordered ranking of m as his preference list.
            - • Each woman ranks all of the men in the same way.
            - • An instability results when a perfect matching S contains two pairs
            - (m, w) and (m, w) s.t. m prefers w to w and w prefers m to m.
        - GOAL: A perfect set of marriages with no instabilities.
    - EX1
        - Consider the following pairs exist in matching set S,
            - • (m, w)
            - • (sm, w)
        - when the preference list is
            - • m prefers w to w
            - • w prefers m to m
            - (m, w) is an instability with respect to S
- Questions About Stable Marriage
    - 1. Does there exist a stable matching for every set of preference lists?
    - 2. Given a set of preference lists, can we efficiently construct a stable matching if there is one?
- The Gale-Shapley Algorithm
    - Pseudoode

```
GALE-SHAPLEY()
 1  Initially all m ∈ M and w ∈ W are free
 2  while ∃m who is free and hasn't proposed to every w ∈ W
 3        do Choose such a man m
 4           Let w be the highest ranked in m's preference list
               to whom m has not yet proposed
 5           if w is free
 6             then (m, w) become engaged
 7             else w is currently engaged to m′
 8                  if w prefers m′ to m
 9                    then m remains free
10                    else w prefers m to m′
11                         (m, w) become engaged
12                         m′ becomes free
13  return the set S of engaged pairs
```

    -
- But Does it Work?
    - It's never a bad idea to convince yourself informally on a simple example…
    - Some Axioms
        - • w remains engaged from the point at which she receives her first proposal
        - • the sequence of partners with which w is engaged gets increasingly better (in terms of her preference list)
        - • the sequence of women to whom m proposes get increasingly worse (in terms of his preference list)
    - Observations
        - Men propose to women in decreasing order of preference (they're "optimistic").
        - Once a woman is matched, she never becomes unmatched (she only "trades up").
- Termination
    - Theorem
        - The G-S algorithm terminates after at most n2 iterations of the while loop.
        - What is a good measure of progress?

- • the number of free men?
- • the number of engaged couples?
- • the number of proposals made?

- Proof
  - Each iteration consists of one man proposing to a woman he has never proposed to before. So we count the number of proposals. After each iteration of the while loop, the number of proposals increases by one; the total number of proposals is upper bounded by n2.

- Towards a Perfect Matching
  - Theorem - If m is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.
    - Proof
      - If, at some point, m is free but has already proposed to every woman. Then every woman must be engaged (because once engaged, they stay engaged, and they would have said yes to m if they weren't engaged when he proposed). Since all n women are engaged there must be n engaged men. This contradicts the claim that m is free.
  - Theorem - The set S returned at termination is a perfect matching.
    - Proof
      - Suppose the algorithm terminates with a free man m. Then m must have proposed to every woman (otherwise the while loop would still be active, and we wouldn't be at termination). But this contradicts the previous theorem, which stated that there cannot be a free man that has proposed to every woman.
  - Theorem - Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.
    - Proof
      - Assume there is an instability. Then there exist two pairs (m, w) and (m, w) in S s.t.
      - m prefers w' to w (1)
      - w' prefers m to m' (2)
        - By definition of w, w'
      - Since m is matched with w. During execution, m's last proposal must have been to w. Had m proposed to w at some earlier time?
      - If no, w must be higher than w' on m's preference list else m would have proposed to w'. This is a contradition to eq. 1.
      - If yes, then he was rejected by w in favor of some other guy m''.
      - Either m'' = m' or w' prefers m' to m'' (since the quality of her match only goes up). Either way, this is a contradicts eq. 2
      - Therefore, S is a stable matching.

- Implementation
  - Summary
    - The Gale-Shapley algorithm guarantees to find a stable matching for any problem instance.
    - • How do we implement the Gale-Shapley algorithm efficiently?
    - • If there are multiple stable matchings, which one does the algorithm find?
  - We can describe a O(n2) implementation.
    - Representing women and men: Assume men are named 1 . . . n and women are named 1 . . . n
    - Engagement
      - • maintain a list of free men in a queue
      - • maintain two arrays of length n, wife[m] and husband[w]
      - • set entry to 0 if unmatched
      - • if m matched to w, then wife[m] = w and husband[w] = m
    - Proposal
      - • For each man, maintain a list of women, ordered by preference
      - • Maintain an array count[m], the number of proposals made by m16/24
    - Women rejecting/accepting
      - • For each woman, create inverse of preference list.
      - • Allows constant time queries:
      - • A woman prefers m to m if inverse[m] < inverse[m]
    - Proposal process
      - • The first free man, m, in the queue proposes the woman at the front of his preference list, w
      - • He increments count[m] and removes w from his preference list
      - • w accepts the proposal if she is unengaged or prefers m to her current match
      - • if w accepts, her former match goes back on the queue of men; otherwise m proposes to his next favorite

June 17th, 2019

- Understanding solutions
  - For a given problem instance, there may be several stable matchings. Do all executions of Gale-Shapley yield the same stable matching? If so, which one?
  - EX 2
    - Given the following preference list:

      $m$ prefers $w$ to $w'$
      $m'$ prefers $w'$ to $w$
      $w$ prefers $m'$ to $m$
    - $w'$ prefers $m$ to $m'$
    - In any execution of G-S algorithm,

      $m$ becomes engaged to $w$
    - $m'$ becomes engaged to $w'$
    - But, there is another possible stable matching (m', w) and (m, w').
    - However, this possibility is not attainable in the version of G-S algorithm where men propose. (only female propose can result in such a pair)
  - w is a **valid** partner for a man m if there is a stable matching that contains the pair (m, w).

- w is the **best valid partner** of a man m if w is a valid partner of m and no woman whom m ranks higher than w is a valid partner of his.
- A man-optimal assignment is one in which every man receives the best valid partner
- Claims
- Claim 1: All executions of GS yield man-optimal assignment, which is a stable matching.
- Proof - by contradiction**********
- Claim 2: All executions of GS yield woman-pessimal assignment, which is a stable matching (i.e., each woman receives the worst possible valid partner).
- Proof - ********
- Other similar problems
  - Stable roommate matching
- Steps in Algorithm Design
  - Formulate the problem precisely.
  - Design an algorithm for the problem.
  - Prove the algorithm correct. - thm&proofs
  - Give a bound on the algorithm's running time. - O(N^2) for stable matching
  -

# ● Basics

- An algorithm is (variations on definitions)
  - a sequence of computational steps that transform the input into an output
  - a tool for solving a well-specified computational problem
  - said to be correct if, for every input instance, it halts with the correct output
  - said to solve a computational problem if it is correct
- Issues
  - Analysis - what we stress on here
  -
  - Design
- Analyzing an algorithm refers to predicting the resources (both amount of time and space) that the algorithm requires, and how these resources scale with increasing input sizes.
- Goal - Our goal is to develop (correct) efficient algorithms as solutions to well-defined problems.
- Efficient?
  - Want: platform-independent && instance-independent
    - of predictive value with respect to increasing instance sizes
  - EX
    - • What is the size of the problem? Before we can answer that, we need to know:
    - • What must be input to run the algorithm? The input preference lists of all participants.
    - A problem instance has a "size" N, which is the total size of the input preference lists (what must be input to run the algorithm).
      - Size of 2n^2
  - Input size
    - The definition of input size depends on the particular computational problem being studied.
      - As a general rule, the running time grows with the size of the input
  - Running time
    - The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
    - • running time should be machine independent
    - • we assume a constant amount of time is required to execute each line of pseudocode
  - Worse-case running time
    - The worst-case running time of an algorithm is the worst possible running time the algorithm could have over all inputs of size N.
      - • Is worst-case the best measure? Generally yes.
      - • Average-case analysis—study of performance of an algorithm over "random" instances. How does one pick random instances?
      - • What is a reasonable analytical benchmark to compare against to tell us whether the running time is good or not?A simple guide is to compare against a brute-force search over the space of all possible solutions.
  - Is better than brute-force efficient?
    - Ex in stable matching
      - Brute-force - n! Many possible perfect matchings
      - Worst-case running for finding a possible instability matching: n^2
      - Total: n! * n^2
      - Vs. G-S Algo: n^2
    - …
      - Polynomial running time
      - What is polynomial time?
      - For any algorithm, if there exists c > 0 and d > 0, such that for every input instance of size N, the running time is bounded by cN^d computational steps.
      - If this running-time bound holds, then we say the algorithm has a polynomial running time.
      -
  - Asymptotic Notation
    - We study the asymptotic efficiency of algorithms; i.e., how the running time scales with increasing input size in the limit.
    - An algorithm with the best asymptotic performance will be the best choice for all but very small inputs.
    - The goal is to identify broad classes of algorithms with similar behavior.
    - We measure running times in the number of primitive "steps" an algorithm must perform. We will be counting steps in pseudocode.
    - Interested in '**rate of growth**'

June 19th, 2019

- O(g(n)) - big O
  - Definition

        - 0<= f(n) <= c*g(n) - there exist c
    - Omega ω(g(n))
        - Definition
        - 0 <= c*g(n) <= f(n)
- (Asymptotic) Tight bound
    - Theta(g(n))
        - Definition
        - Such that 0 ≤ c1g(n) ≤ f (n) ≤ c2g(n) for all n ≥ n0
- Loose bound
    - o(g(n)) - little o
        - • Consider 2n2 = O(n2) and 2n2 = O(n3).
        - • Both are asymptotic upper bounds for f (n) = 2n2.
        - • For 2n2 = O(n2), we can say f (n) = 2n2 does not grow faster than g(n) = n2. While, for 2n2 = O(n3), we can say f (n) = 2n2 grows much slower than t (n) = n3.
        - • We use o-notation to refer to upper bounds that are loose.
        -
        - Definition
            - Given g(n), we denote by o(g(n)) the set of functions:
                o(g(n)) = {f (n) : for any positive constant c > 0, there
            - exists a constant n0 > 0 such that 0 ≤ f (n) < cg(n) for all n ≥ n0}
        - **Loose upper bound**
        - For all c, 0<= f(n) < c*g(n)
            - Strictly smaller
    - Little omega ω( g(N) )
        - Definition
            - Given g(n), we denote by ω(g(n)) the set of functions:
                ω(g(n)) = {f (n) : for any positive constant c > 0, there
            - exists a constant n0 > 0 such that 0 ≤ cg(n) < f (n) for all n ≥ n0}
    - Definition summary
        -

| Definition | $?$ $c \geq 0$ | $?$ $n_0 \geq 0$ | $f(n)$ $?$ $c \cdot g(n)$ |
|---|---|---|---|
| $O(\cdot)$ | $\exists$ | $\exists$ | $\leq$ |
| $o(\cdot)$ | $\forall$ | $\exists$ | $<$ |
| $\Omega(\cdot)$ | $\exists$ | $\exists$ | $\geq$ |
| $\omega(\cdot)$ | $\forall$ | $\exists$ | $>$ |

- The Limit Theorem
    - !!!
    - **Theorem**

      $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ (where } 0 < c < \infty) \implies f(n) = \Theta(g(n))$$

      *$f(n)$ grows at the same rate as $g(n)$*
    - **Theorem**

      $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ (where } 0 \leq c < \infty) \implies f(n) = O(g(n))$$
    -
    - **Theorem**

      $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ (where } 0 < c \leq \infty) \implies f(n) = \Omega(g(n))$$
    - **Theorem**

      $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \omega(g(n))$$

      *$f(n)$ grows faster than $g(n)$*
    - **Theorem**

      $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = o(g(n))$$

      *$f(n)$ grows slower than $g(n)$.*
    -
- **If and only if statement for all limit <-> O/Omega/Theta/o/omega**
- Add on both side, still true?
    - Log - false
    - 2^n - false
    - Square - true
- Properties of Asymptotic

- **Transpose Symmetry**

  $$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

- **Symmetry**

  $$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

- **Reflexivity**

  $$f(n) = O(f(n))$$
  $$f(n) = \Omega(f(n))$$
  $$f(n) = \Theta(f(n))$$

- **Transitivity**

  $$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \implies f(n) = O(h(n))$$
  $$f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$$
  $$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$$

- **Proof**

  Given, for some constant $c > 0$ and $n_0 > 0$, we have $f(n) \leq cg(n),\ \forall n \geq n_0$ .
  Also, $c' > 0$ and $n_0' > 0$, we have $g(n) \leq c'h(n),\ \forall n \geq n_0'$.
  So, $\forall n \geq \max(n_0, n_0')$, we have $f(n) \leq cg(n) \leq cc'h(n)$
  $\implies f(n) \leq \tilde{c}h(n)$, where $\tilde{c} = cc' > 0$.
  $\implies f(n) = O(h(n))$.

- **Sum of Functions**

  Suppose that $f(n)$ and $g(n)$ are such that $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n) + g(n) = O(h(n))$

- **In general**

  Let $k$ be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and $h$ be functions such that $f_i = O(h)\ \forall i$ Then $f_1 + f_2 + \cdots + f_k = O(h)$.

- Thm. Suppose that f (n) and g(n) (taking non-negative values) are such that g(n) = O(f (n)), then f (n) + g(n) = Theta (f (n))

■ Standard notation
- Monotonicity
  - f (n) is monotonically increasing if m ≤ n implies f (m) ≤ f (n) - non-decreasing
  - f (n) is monotonically decreasing if m ≤ n Implies f (m) ≥ f (n) - non-increasing
  - f (n) is **strictly increasing** if m < n implies f (m) < f (n)
  - f (n) is **strictly decreasing** if m < n implies f (m) > f (n)
- Floors and Ceilings
  - For any real number x, we denote the greatest integer less than or equal to x by |_x_|
  - … the least integer greater than or equal to x by |`x`|
- Modular Arithmetic
  - Remainder of the quotient a/n
- Polynomials
  - Polynomial in n of degree d is a function p(n) of the form: p(n) = Sum(i=0 to d) ai * n^i
- Exponentials
  - Any exponential function with base strictly greater than 1 grows faster than any polynomial function
- Basic Logarithm notations

■ Bounds for common functions
- Polynomials
  - **Polynomials**

    Given a nonnegative integer $d$, a **polynomial in n of degree d** is a function $p(n)$ of the form:

    $$p(n) = \sum_{i=0}^{d} a_i n^i$$

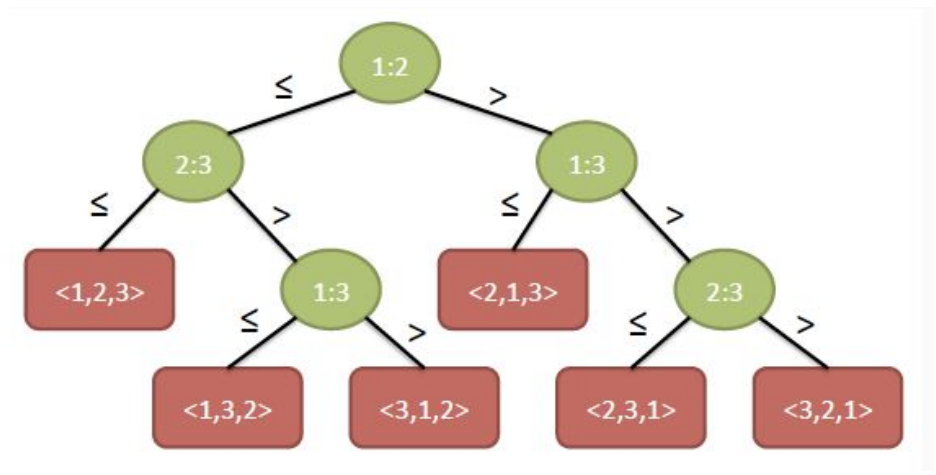
    where the constants $a_0, a_1, \ldots a_d$ are the **coefficients** of the polynomial and $a_d \neq 0$

    - For an asymptotically positive polynomial $p(n)$ of degree $d$, we have $p(n) = \Theta(n^d)$.
  - Note: An algorithm can be polynomial time even if then running time is of the form O(n^x), where x is not an integer
- Logarithms
  - You can ignore the base in logarithms
  - Log grow slower than polynomials

- Exponentials
    - Every exponential grow faster than every polynomial
    -

June 21st, 2019

- Common running time
    - Linear Time O(N)
        - Definition - running time is at most a constant factor times the size of the input
            - EX. Check sorted, find max/min, sumn elements, merge two sorted lists,
    - Linearithmic TIme O(NlogN)
        - Commonly arises in divide and conquer algorithms
            - Ex. Sort (nlogn)
            -
    - Quadratic time O(N^2)
            - Ex. Nested loop; Enumerate all pairs of elements. If there are n elements, then there are (n choose 2) pairs
            - Closest pair of points
    - Cubic time O(N^3)
        - Set disjointness
        - Independent set of size k
        - Matrix multiplication
            - Fastest: https://en.wikipedia.org/wiki/Strassen_algorithm
    - Polynomial Time: O(n^k)
        - Eg. Independent Set of Size k - the number of subset S of k nodes is sum of n choose 0 to n choose k
    - Exponential time
        - Expensive!
            - Eg. Independent Set - Given a graph, what is the size of the largest independent set? - O(n^2 * 2^n)
                Enumerate all subsets, and for each subset S check if independent
    - Sublinear time
        - Less than linear time
        - Eg. Binary Search
        -
- Lower bound of sorting
    - Comparison sorting - The sorted order of the output is based on a comparison between input elements. Such sorting algorithms are called comparison sorting.
        - Lower bound:
            - All inputs are distinct.
            - Without loss of generality we consider <= comparisons.
            - **Decision Trees**
                - a full binary tree
                - represents the comparisons between elements performed by a sorting algorithm
                - each internal node is annotated by i : j for some i and j in 1 ≤ i, j ≤ n, for n elements in the input sequence
                - each leaf is annotated by a permutation π(1), π(2), . . . , π(n)
                - executing the sorting algorithm equates to tracing a path through the decision tree
                -
                - Each internal node indicates the comparison ai ≤ aj .
                - The left subtree captures subsequent comparisons, once we know that ai ≤ aj .
                - The right subtree captures subsequent comparisons, once we know that ai > aj .
                - When we come to a leaf the sorting algorithm has established the desired ordering.
            - Any correct sorting algorithm must be able to generate each of the n! peruations on n elements; each permutation must appear as a leaf in the decision tree.




            -
        - Worst case: The length of the longest path from the root of a decision tree to any of its reachable leaves is the worst-case number of comparisons that the corresponding sorting algorithm performs.
        - Lower bound: A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.
            - Thm. Any comparison sort algorithm requires Ω(n lg n) comparisons in the worst case. (We have to make at least Ω(n lg n) comparisons for any comparison sorting algorithm.)
            - Pf. We have to determine the height of a decision tree with n!leaves. A binary tree of height h has no more than 2h leaves. Therefore n! ≤ 2h, so h ≥ lg(n!) = Ω(n lg n).
            - Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts.

(L5 Part3 End)
(L6 Start)

# ● Priority Queues

- Efficient Algorithms found first
    - Seek algorithms that are quantitatively better than brute force search.
    - Seek algorithms that are polynomial time.

        - Once we find this efficient algorithm, we can further improve runtime by taking care of the implementation details, sometimes through complex data structures.
    - A priority queue is a data structure that maintains a set of elements S, where each element v ∈ S has an associated value *key(v)* that denotes the priority of the element v. Smaller keys represent higher priority.
    - Operations on a priority queue
        - add elements to the set
        - delete elements from the set
        - Selection of an element with the smallest key
    - Problem
        - Motivation: Stable Marriage - The stable marriage algorithm needs a data structure that maintains the dynamically changing set of all free men. The algorithm needs to be able to:
            - add elements to the set
            - delete elements from the set
            - select an element from the set, based on some assigned priority
        - Eg. Schedule Processes on a Computer
            - Each process has a priority
            - Processes do not arrive in order of priority
            - When ready, we want to extract the process with the highest priority or key with lowest value
        - Eg. Sort - Sort a set of n elements
            - Possible Algorithm
                - Set up a priority queue H, and insert each value into H with it's value as the key.
                - Repeatedly find the smallest number in H, and output it ("find minimum" operation).
                - OR
                - Sort array in O(n) "find minimum" operations.
                - Comparison sorting algorithms have O(n log n) running time. If we want to achieve this bound each "find minimum" step must take O(log n) time.
    - Candidate Data Structure for Priority Queues
        - List
            - Insertion and deletion take O(1) time, but finding the minimum requires scanning the list and takes Ω(n) time
        - Sorted Array
            - Finding the minimum takes O(1) time, but locating where to insert or delete element from would take O(log n), and then inserting/deleting would take O(n)
            - (move all elements).
        - ...no O(log n)

- Heaps
    - Def.
        - Combine the benefits of both lists and sorted arrays
        - Conceptually, a heap is a balanced binary tree
        - The tree has a root, and each node can have up to two children.
        - ***Heap order: For every element v at node i, the element w at i's parent satisfies key(w) ≤key(v)***
    - Implement
        - in a pointer-based data structure
        - Alternatively, assume a maximum number N of elements is known in advance
        - Store nodes of the heap in an array
            - Node at index i has children at indices 2i and 2i + 1 and parent at index i/2
            - Index 1 is the root
            - How do you know that a node at index i is a leaf? If 2i > N, the number of elements in the heap.
    - Heapify-Up
        - 1. Heap H has n < N elements
        - 2. Insert a new element at i = n + 1, by setting H[i] = v.
        - 3. This may break the heap-order.
        - 4. Fix the heap order using Heapify-up(H; n + 1).

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]] < key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

        -

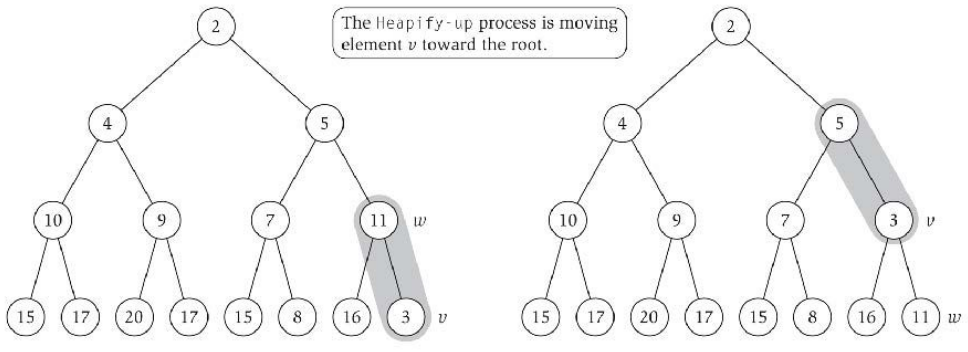



**Figure 2.4** The Heapify-up process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).

        -
    - Correctness of Heapify-Up
        - H ***is almost a heap with key of H[i] too small if there is a value*** alpha >= key(H[i]) such that increasing key(H[i]) to alpha makes H a heap

- **Claim**: The procedure Heapify-Up(H; i) fixes the heap property in O(log i) time, assuming that the array H is almost a heap with the key of H[i] too small.
    - Prove by induction on i.
    - **Base case:** i = 1. H[1] is the root, so if it's too small, then H is
    - already a heap.
    - **Inductive Hypothesis:** Heapify-Up(H; j), where j = floor(i/2) fixes the heap property in O(log j) time, assuming that the array H is almost a heap with the key of H[j] too small.
    - **Inductive step:** H is almost a heap with key of H[i] too small. Let j =parent(i) = floor(i/2) and beta be its key. Swapping the elements at H[i] and H[j] takes O(1) time, and now H[i] = beta. After the swap, H is a heap or almost a heap with the key of H[j] too small, since setting its key to beta would make H a heap. Finally, by the
    - inductive hypothesis, the recursive call to Heapify-Up(H, j) fixes the heap property.
- **Corollary:** Using Heapify-Up we can insert a new element in a heap of n elements in O(log n) time. (Why?)

Cost of `Heapify-Up` $(H, i)$

$$\begin{aligned} &= \log j + 1 \\ &= \log(\lfloor \tfrac{i}{2} \rfloor) + \log 2 \\ &= \log(2\lfloor \tfrac{i}{2} \rfloor) \\ &= \log i \end{aligned}$$

- Deleting an Element
    - Suppose H has n + 1 elements
    - 1. Delete element at H[i] by moving element at H[n + 1] to H[i]
    - 2. If element at H[i] is too small, fix heap order using Heapify-up(H; i)
    - 3. If element at H[i] is too large, fix heap order using Heapify-down(H; i)

```
Heapify-down(H,i):
Let n = length(H)
If 2i > n then
  Terminate with H unchanged
Else if 2i < n then
  Let left = 2i, and right = 2i + 1
  Let j be the index that minimizes key[H[left]] and key[H[right]]
Else if 2i = n then
  Let j = 2i
Endif
If key[H[j]] < key[H[i]] then
   swap the array entries H[i] and H[j]
   Heapify-down(H, j)
```

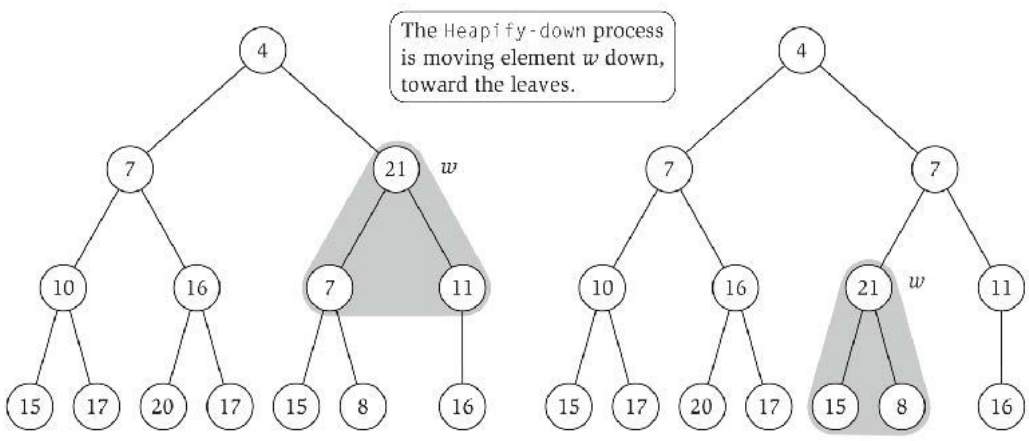



**Figure 2.5** The Heapify-down process:. Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).

- Correctness of Heapify-Down
    - Similar to Heapify-Up but opposite
    - Claim - prove by reverse induction on i
    - Corollary: Using Heapify-Down we can delete an element from a heap of n elements in O(logN) time
- In class exercise 1
    - Build a heap out of an arbitrary array
        - What is the height of an n-element heap?
            - O(log n) (it's a (nearly) complete binary tree).
        - How many nodes are there at height h of an n-element heap?
            - Key Observation
                - The number of leaves in a complete binary tree is dn=2e.
            - Proposition
                - In an n-element heap, there are dn=2h+1e nodes at height h.
                - Proof @ p23-25/27
- HeapSort
    - Sort
        - Instance: Nonempty list x1, x2, …, xn of integers
        - Solution: A permutation y1, y2, …, yn of x1, x2, …, xn such that yi<= y(i+1) for all 1<=i<=n
    - Final Algorithm
        - Insert each number in a priority queue H
        - Repeatedly find the smallest number in H, output it, and delete it from H
    - Each insertion and deletion takes O(log n) time for a total running time of O(n log n)
- In class Exercise 2
    - Problem: One of your classmates claims that he built an alternative data structure (other than a heap) for representing a priority queue. He claims that, using his new data structure, INSERT, MAX, and EXTRACTMAX all take constant (O(1)) time in the worst case. Give a very simple proof that he is mistaken.

        - Solution: If this were true, we could comparison sort in O(n) time. But we've already proven that this is not possible.

(L6 End)
(L7 Part I Starts)

# Graph

## Definition and Applications

- Undirected Graph G = (V, E)
  - V : nodes
  - E: edges between pairs of nodes
  - captures pairwise relationships between objects
  - graph size parameters: n = |V |, m = |E|
- Directed Graph G = (V, E)
  - Almost the same as undirected Graph, expect: captures on-way relationships between objects
- Applications

| Graph | Nodes | Edges | Directed |
|---|---|---|---|
| transportation | intersections | highways | no |
| communication | computers | fiber optic cables | no |
| World Wide Web | web | pages hyperlinks | yes |
| social | people | relationships | maybe |
| food web | species | predator/prey | yes |
| software systems | functions | function calls | yes |
| scheduling | tasks | precendences | yes |

## Graph Presentations

- Two ways:
  - Adjacency list
    - ArrayList of nodes, each nodes also contains a ArrayList of other nodes that it connects to, i.e. edges
    - Memory requires: Theta(n + m)
    - Undirected graph vs. directed graph on edges number: 2m vs. m
    - Weights store in w(u, v), u and v are two nodes connected by an edge (u, v)
  - Adjacency matrix
    - Always use Theta(N^2) memory
    - Mark node a(i,j) 1 if there exists an edge, otherwise mark 0
    -
- Usually use list, list usually needs less storages
- Checking if an edge exist:
  - O(1) for finding if an edge exist using adj matrix.
  - O(n) for finding in adj list because we need to go through all nodes that connect to the nodes we are looking at, at most n nodes.
- Path and connectivity
  - Definition: Path
    - A path in an undirected graph G = (V;E) is a sequence P of nodes v1, v2, …, v_k-1, v_k with the property that each consecutive pair vi, v_i+1 is joined by an edge in E.
  - Definition: Simple Path
    - A path P is simple if all nodes in P are distinct.
  - Definition: A Connected Undirected Graph
    - An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.
- Cycle

**Definition: Cycle**

A cycle is a path $v_1, v_2, \ldots v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.

  -
- Tree
  - Connected acyclic undirected graph
  - m = n - 1
- Rooted trees
  - With a special node - root
  - Given a tree T, choose a root node r and orient each edge away from r . This enables one to model hierarchical structure.

## Graph Traverse

- Connectivity
  - s to t Connectivity Problem
    - Given two nodes s and t, is there a path between s and t?
  - s to t Shortest Path Problem
    - Given two nodes s and t, what is the length of the shortest path between s and t?
  - Applications
    - Social network connections (e.g., Kevin Bacon number)
    - Maze traversal
    - Fewest number of hops in a communication network
- Connected Components
  - Related problem: find all nodes reachable from node s

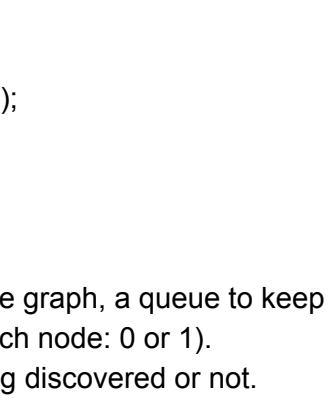
```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

- Theorem: Upon termination, R is the connected component containing s.
- Practical problem: flood fill - coloring

■ Breadth First Search
- Solve problem: Given a graph G = (V;E) and a specific source vertex s, what vertices can be reached from s?
  - a node is “discovered” the first time the BFS encounters it; at this point BFS colors the node gray
  - the complete set of gray nodes is the “frontier”
  - to proceed, BFS looks at each of the gray nodes, examines each of its outgoing edges, to see if they’re connected to any white (undiscovered) nodes
    - if so, color that node gray and insert this node at the end of the queue of the frontier vertices
    - when we’ve examined all of a node’s outgoing edges, remove it from the frontier queue and color it black
- Intuition: search each layer
- Thm: For each i, Li consists of all nodes at distance exactly i from s. There is a path from s to t if and only iff t appears in some layer.
- Thm: Let T be a breadth first search tree, let x and y be nodes in T belonging to layers Li and Lj respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.
- Pseudocode
  - n = |V|, m = |E|;
  - // Create an array (discovered array) of size n;
  - Discovered array [n] = 0;
  - frontier = (s); // fifo queue
  - while (frontier not empty){
  - next = frontier.pop();
  - if(discovered[next] = 0){
  - discovered[next] = true;
  - frontier.append(neignbors(next));
  - }
  - }
  -
- Analysis
  - Assume that we use adjacency lists to store the graph, a queue to keep track of frontier nodes, and an array to keep track of which nodes were “discovered” (for each node: 0 or 1).
    - O(n) time to initialize the array indicating discovered or not.
    - Each node is discovered at most once; queue operations are O(1) at most; at most O(m) time is spent interacting with the queue each adjacency list is scanned at most once (when the node is explored); so the total time spent looking at adjacency lists is O(2m) = O(m)
    - So the total running time of breadth first search is O(n + m), or linear in size to the adjacency list representation.
- BFS and Shortest Path
  - The level of a node in a breadth first search is the distance computed by the breadth first search algorithm from s to u. We define the shortest-path distance, d(s, v) from s to v as the minimum number of edges in any path from s to v
    - if there is no path from s to v, then d(s, v) = ∞
  - It is a non-trivial fact that the levels computed in breadth first search are the shortest distances from s to any node u. We’ll revisit this problem in Chapter 4.

■ Depth First Search
- An alternative to exploring across the entire frontier at the same time is to explore a single path as far as it can go, then explore a different one.
  - depth-first search explores “deeper” into the graph whenever possible
  - edges are explored out of the most recently discovered vertex (v) until there are no more
  - then the search backtracks, exploring other paths out of v’s parent

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```




**Theorem**

Let $T$ be a depth-first search tree, let $x$ and $y$ be nodes in $T$, and let $(x, y)$ be an edge of $G$ that is not an edge of $T$. Then one of $x$ or $y$ is an ancestor of the other in $T$.

■ Comparing BFS and DFS

**Similarities**

- Both build the strongly connected component of $G$ that contains $s$.
- Both have similar efficiency

**Differences**

- They explore the vertices of $G$ in very different orders.
- They result in trees rooted at $s$ that have very different structure (bushy vs. tall)

- 

## ○ Incident Matrix

- An incidence matrix can also be used to represent a graph. An incidence matrix M has a row for each vertex and a column for each edge. M[i, j] = 1 if vertex i is part of the edge j.
  - Give an efficient algorithm to convert an adjacency list representation to an incidence matrix. What is the time complexity of your algorithm?
  - O(|V|+|E|)

**Convert adjacency list to an incidence matrix:**

```
1  create M of size |V| × |E|
2  for every v ∈ G.V
3       do lookup i for v
4            for every u ∈ Adj[v]
5                 do lookup j for (u, v)
6                      set M[i, j] = 1
```

  - 
  - Give an efficient algorithm to convert an incidence matrix to an adjacency list representation. What is the time complexity of your algorithm?
  - O(|V||E|)

**Convert an incidence matrix to an adjacency list:**

```
1  for each v ∈ G.V
2       do create empty list Adj[v]
3          lookup i for v
4          for 0 ≤ k ≤ |E|
5               do if M[i, k] = 1
6                  then lookup (v, u) for k
7                       Adj[v] = Adj[v] ∪ (u)
```

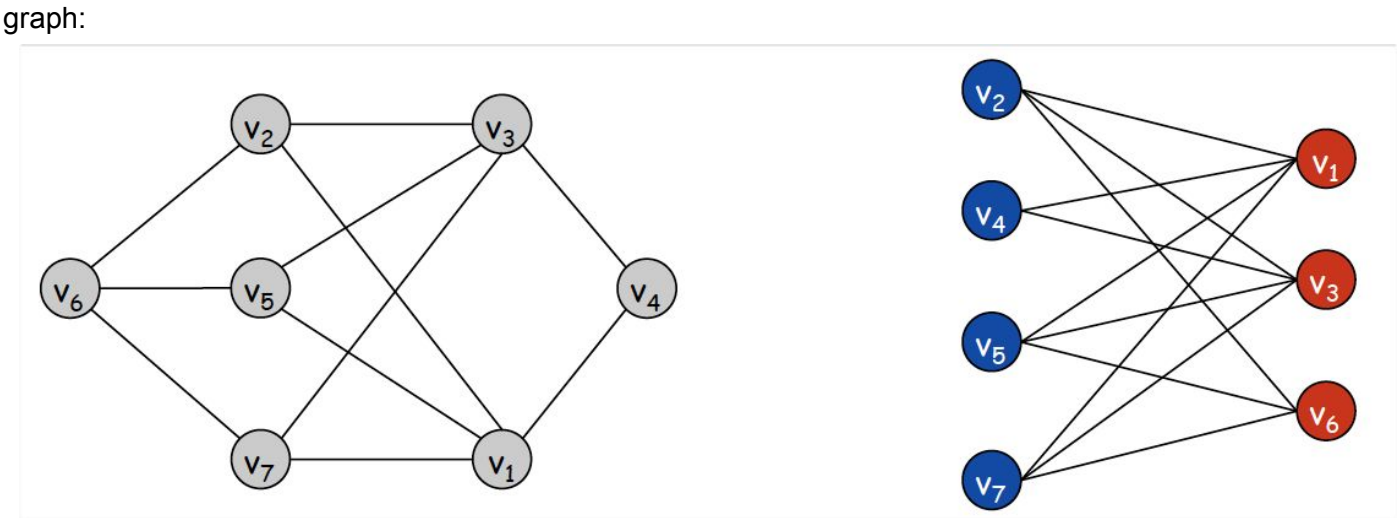
  - 

## ○ Exercise

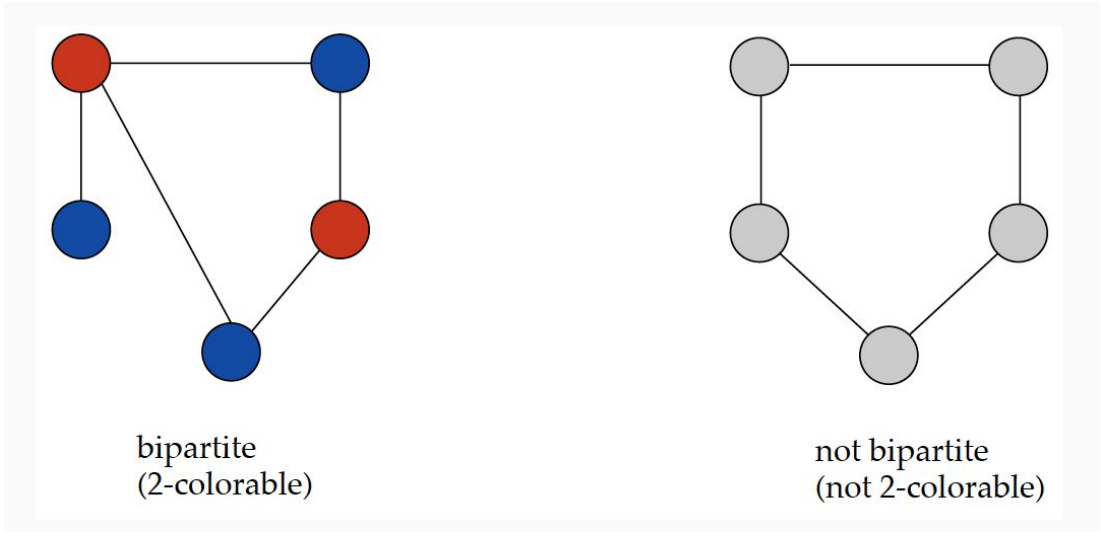
- We have a connected graph G = (V , E) and a specific vertex u ∈ V . Suppose that we compute a depth first search tree rooted at u and obtain a tree T that includes all of the nodes of G. Suppose we then compute a breadth first search tree rooted at u and obtain the same tree T . Prove that G = T .
- Pf. Suppose that G has an edge (a, b) that does not belong to T . Since T is a depth first search tree, one of the two ends must be an ancestor of the other. Let's say that a is an ancestor of b. Since T is also a breadth first search tree, the levels of the two nodes in T can differ by at most one. But if a is an ancestor of b and the distance from the root to b in T is at most one greater than the distance from the root to a, then a must be b's parent in T and (a, b) ∈ T . This is a contradiction.

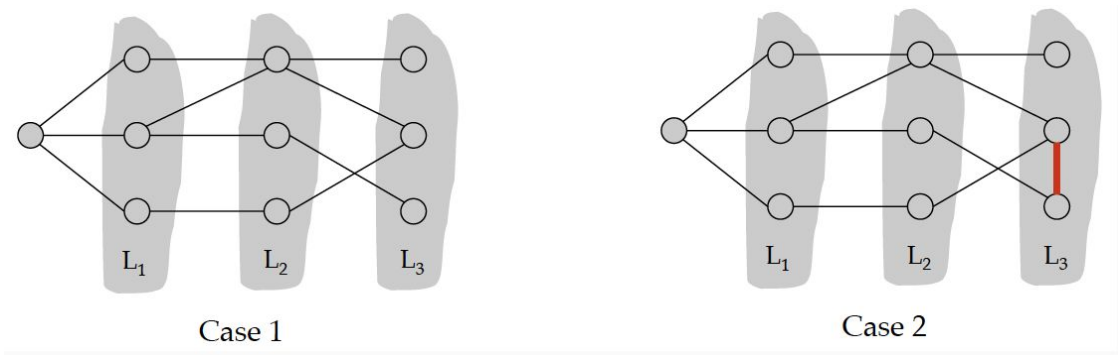
## ○ Testing Bipartiteness

- Definition of Bipartite Graph
  - An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.
  - Applications.
    - Stable marriage: men = red, women = blue.
    - Scheduling: machines = red, jobs = blue.
- Testing
  - Given a graph G, is it bipartite?
    - Many graph problems become easier if the underlying graph is bipartite (matching) tractable if the underlying graph is bipartite (independent set)
    - Before attempting to design an algorithm, we need to understand the structure of bipartite graphs
    - Same graph:
  - 

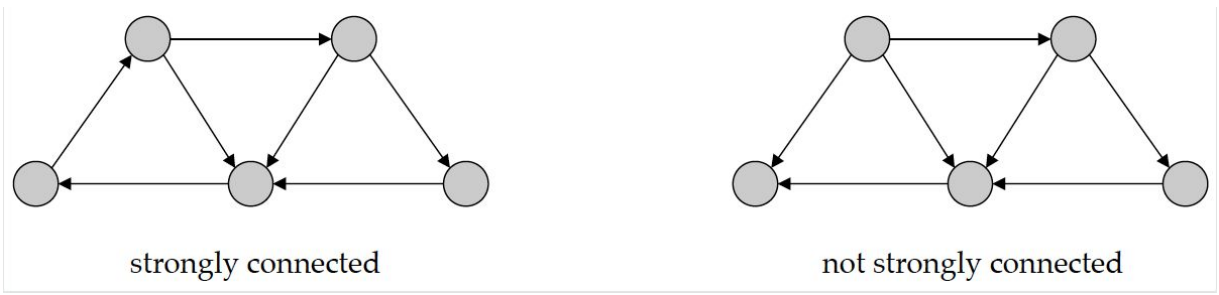



  - Lemma:

- If a graph G is bipartite, it cannot contain an odd length cycle.
- 


- Lemma2
  - Let G be a connected graph, and let L0 , . . . , Lk be the layers produced by BFS starting at node s. Exactly one of the following holds.
    - 1. No edge of G joins two nodes of the same layer, and G is bipartite.
      - Pf.
      - Suppose no edge joins two nodes in the same layer.
      - By the previous lemma (in BFS if (x, y) is an edge in G then the layer difference is at most 1.) this implies that all edges join nodes on adjacent levels.
      - Then the bipartition is such that nodes on odd levels are red; nodes on even levels are blue.
      -
    - 2. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).
      - Pf.
      - Suppose (x, y) is an edge with x and y in the same level Lj .
      - Let z be the lowest common ancestor of x and y. Let Li be the level containing z.
      - Consider the cycle that takes the edge from x to y, then the path from y to z, then the path from z to x.
      - It's length is 1 + (j - i) + (j - i) = 2(j - i) + 1, which is odd.
      -
  - 


## Connectivity in Directed Graph

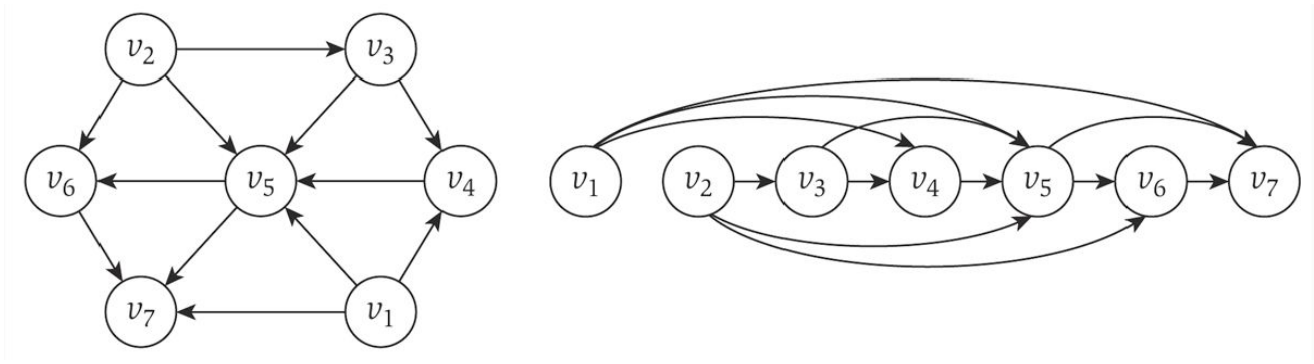
- Directed Graph Def.
  - In a directed graph, G = (V, E), an edge (u, v) goes from node u to node v.
- Application
  - Directed Reachability
    - Given a node s, find all nodes reachable from s.
  - Directed s - t Shortest Path Problem
    - Given two nodes s and t, what is the length of the shortest path between s and t?
  - Graph Search
    - Breadth first search (and depth first search) extend naturally to directed graphs.
    - Set of nodes t such that there is a path from s to t.
    - How do I find a set of nodes that have a path to s?
  - Web Crawler
- Strong Connectivity
  - Definition
    - Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u.
  - Definition
    - A graph is **strongly connected** if every pair of nodes is mutually reachable.
  - Lemma
    - Let s be any node. G is strongly connected iff every node is reachable from s and s is reachable from every node.
- Determine Strong Connectivity
  - We can determine if G is strongly connected in O(m + n) time.
  - Idea: We have to show every node that is reachable from s, and every node is reachable to s.
  - 

- Algorithm
  - Pick any node s.
  - Run BFS from s in G.
  - Run BFS from s in G_rev (the reverse orientation of every edge in G)
  - Return true iff all nodes reached in both BFS executions

        - Correctness follows from the previous lemma
    - Strong Component
        - Def. The strong component containing a node s in a directed graph is the set of all v such that s and v are mutually reachable.
        - The previous algorithm is really computing the strong component containing s.
        - Thm. For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

## ○ DAGs and Topological Ordering

- Def. A DAG is a directed graph that contains no directed cycles.
- Def. A topological order of a directed graph G = (V , E) is an ordering of its nodes as v1, v2, . . . , vn so that for every edge (vi , vj ) we have i < j.
    - Same graph:

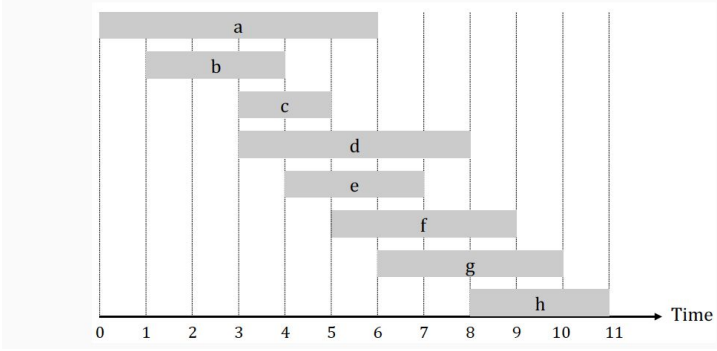



- Precedence Constraints
    - Edge (vi, vj ) means task vi must occur before vj .
- Applications
    - Course prerequisite graph: course vi must be taken before vj .
    - Compilation: module vi must be compiled before vj.
    - Pipeline of computing jobs: output of job vi needed to determine input of job vj
- Lemma.
    - If G has a topological order, then G is a DAG.
    - Pf.
        - Suppose that G has a topological order v1, …, vn and that G also has a directed cycle.
        - Let vi be the lowest-indexed node in the cycle and let vj be the node just before vi . Thus (vj, vi ) is an edge in E.
        - By our choice of i, we have i < j.
        - On the other hand, since (vj, vi ) is an edge and v1, …, vn is a topological order, we must have j < i, a contradiction.
- Lemma.
    - If G is a DAG, then G has a node with no incoming edges
    - Pf.
        - Suppose G is a DAG and every node has at least one incoming edge.
        - Pick any node v and begin following edges backward from v. Since v has at least one incoming edge (u, v), we can walk backward to u.
        - Then since u has at least one incoming edge (x, u), we can walk backward to x.
        - Repeat until we visit a node, say w, twice.
        - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle.
- Lemma.
    - If G is a DAG, then G has a topological ordering.
    - Pf.
        - Base case: true if n = 1.
        - Given a DAG on n > 1 nodes, find a node v with no incoming edges.
        - G - {v} is a DAG, since deleting v cannot create cycles.
        - By inductive hypothesis, G - {v} has a topological ordering.
        - Place v first in the topological ordering, then append the nodes of G - {v} in topological order. This is valid since v has no incoming edges.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

- The algorithm finds a topological order in O(m + n) time.
    - Pf.
        - Maintain the following information:
            - count[w]: the remaining number of incoming edges
            - S: the set of remaining nodes with no incoming edges
        - Initialization: O(m + n) via a single scan through the graph
        - Update: to delete v:
            - remove v from S
            - decrement count[w] for all edges v to w, and add w to S if count[w] hits 0
            - This is constant time per edge, and we go over each edge once
- Using DFS
    - DFS(u)
    - 1 mark u as explored and add to R
    - 2 for each (u, v) incident to u
    - 3 do if v is not explored
    - 4 DFS(v)
    -

- Greedy Algorithm
  - Intro
    - An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.
    - Locally, incrementally optimizing some measure on its way to a solution.
    - We will prove that a greedy algorithm produces an optimal solution to a problem. Two approaches
      - That the greedy algorithms stays ahead.
      - The exchange argument: Any possible solution and gradually transform it into a solution found by the greedy algorithm without hurting its quality.
    - We will illustrate these approaches using examples of Interval Scheduling.
    - Applications of greedy algorithms: Shortest path in a graph, Minimum Spanning Tree, Huffman codes.
  - Interval Scheduling
    - Job $j$ starts at $s_j$ and finishes at $f_j$
    - Two jobs are compatible if they do not overlap
    - The goal is to find the maximum subset of mutually compatible jobs




    - Greedy: Earliest finish time
      - Sort jobs by finish times so that f1 <= f2 <= ... <= fn.
      - A set as an empty set - job selected
      - for j = 1 to n {
        if (job j compatible with A)
        A = A U {j}
      - }
      - return A
    - Implementation: O(n log n)
      - Job j is compatible with A if sj >= fj*
    - Analysis
      - To show optimality of the greedy algorithm, we need to show that for the set of intervals A returned by the greedy algorithm
        - All the intervals are compatible.
        - Comparable to the optimal set of intervals O.
      - Fact: A is a compatible set of requests.
      - Ideally, we would like to show that A = O. But there may be many optimal solutions, so at best we can show |A| = |O|. (*O is the optimal and |O| is the maximum mutually compatible tasks in the optimal solution).
      - Strategy
        - We will try to show that our greedy algorithm “stays ahead” of the optimal solution O.
        - Compare partial solution of our greedy algorithm to initial segments of O, and we will show that the greedy algorithm is doing better in a step-by-step fashion.
      - Notation
        - Let i1, i2, …, ik be the set of requests in A in the order that they were added to A.
        - Therefore, |A| = k.
        - Similarly, let j1, j2, …, jm be the set of requests in O and |O| = m.
        - Our goal is to prove k = m.
      - Greedy stay ahead of the optimal solution
        - When i counts indices in the greedy solution and j counts indices in the previous solution, for all indices r ≤ k, fir ≤ fjr .
        - Pf.
        - Base case: r = 1; the claim is clearly true, since the greedy algorithm selects the interval with the earliest finish time.
        - Inductive step: Let r > 1, and assume the claim is true for r − 1. By the
        - inductive hypothesis, we have fir−1 ≤ fjr−1 . For our greedy algorithm to fall behind, the next interval would have to finish later. But by the definition of the greedy algorithm, at each step, the algorithm will choose the next non-conflicting job with the earliest finish time. The greedy algorithm would have the option of picking jr as it’s next request, thus fulfilling the inductive step.
      - Thm. The greedy algorithm returns an optimal set A.
        - Pf.
          - A is not optimal, then O must have more requests than A, that is m > k.
          - By previous theorem, with r = k, we have fik ≤ fjk .
          - Since, m > k, there is a request jk+1 in O. This request starts after request jk ends and hence after ik ends (Greedy algorithm always ahead).
          - That means there is one more request that is compatible to A, but the greedy algorithm terminated, which is a contradiction.
    - Additional potential concerns
      - the algorithm doesn’t know all of the requests a priori —online algorithms
      - different requests may have different weights — weighted interval scheduling
  - Interval Partition
    - Consider the problem in which one must schedule all requests using the fewest “resources” (e.g., processors, classrooms)

- Lecture j starts at sj and finishes at fj .
- Goal: find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

- Lower Bound
  - Definition
    - The depth of a set of open intervals is the maximum number that contain any given time.
  - Key Observation
    - The number of classrooms needed <= depth
  - Example
    - The depth of the schedule on the previous slide was three; the schedule is optimal.
  - Question
    - Does there always exist a schedule equal to the depth of intervals?
    - EX. 3 classrooms needed for both 3 & 4 classroom
- Greedy Algo
  - Consider lectures in increasing order of start time; assign lecture to any compatible classroom.
  - Pseudocode:
    - Sort intervals by starting time so that s1 <= s2 <= .. <= sn.
    - d <- 0 // number of allocated classroom
    - for j = 1 to n {

```
if (lecture j is compatible with some classroom k)
        schedule lecture j in classroom k
Else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1
        d = d + 1
}
```

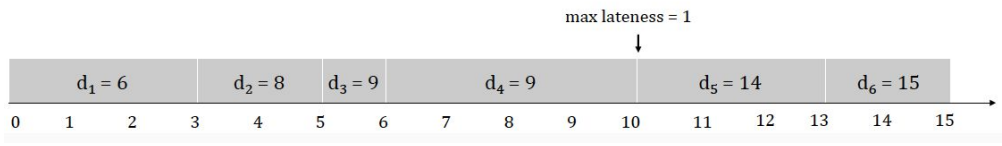
    - Implementation
    - For each classroom k, maintain the finish time of the last job added.
    - Keep the classrooms in a priority queue.
    - Running time: O(n log n)
- Greedy Analysis
  - Observation
    - The greedy algorithm never schedules two incompatible lectures in the same classroom.
  - Theorem
    - The greedy algorithm is optimal.
  - Proof
    - Let d be the number of classrooms the greedy algorithm uses.
    - Classroom d is opened because we need to schedule a job, say, j, that is incompatible with all d - 1 other classrooms.
    - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than sj .
    - Thus, at time sj + e, we have d overlapping lectures.
    - Key observation: all schedules use >= d classrooms

## ○ Scheduling Minimize Lateness

- Description
  - Single resource processes one job at a time
  - Job j requires tj units of processing time and is due at time dj
  - If j starts at time sj , it finishes at time fj = sj + tj
  - Lateness = lj = max{0, fj - dj}
  - Goal: Schedule all jobs to minimize maximum lateness L = max lj
- Greedy Algorithm
  - Intuition: Earliest Deadline

```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
  Assign job j to interval [t, t + t_j]
  s_j ← t, f_j ← t + t_j
  t ← t + t_j
output intervals [s_j, f_j]
```

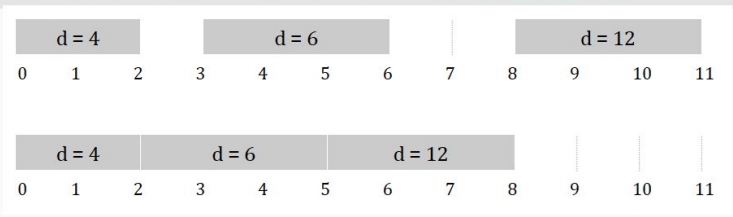



○

- No idle time

**Observation 1**

There exists an optimal schedule with no idle time.




**Observation 2**

The greedy schedule has no idle time.

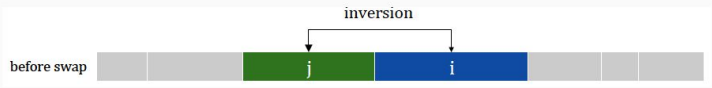
**Process**

Consider an optimal schedule $O$. Gradually modify $O$, preserving its optimality at each step, but eventually transforming it into a schedule $A$ that our greedy algorithm would give us. This is an exchange argument.

- No Inversions(for greedy)

**Definition**

An inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that $d_i < d_j$ but $j$ scheduled before $i$.




**Observation**

A greedy schedule has no inversions.

**Observation**

If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

- Claim
  - Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the maximum lateness.
  - Pf.

Let $l$ be the lateness before the swap, and let $l'$ be it after.

- $l'_k = l_k$ for all $k \neq i, j$ and $l'_i \leq l_i$
- If job $j$ is late:
  - $l'_j = f'_j - d_j$ (definition)
  - $l'_j = f_i - d_j$ ($j$ finishes at time $f_i$)
  - $l'_j \leq f_i - d_i$ ($i < j$)
  - $l'_j \leq l_i$

- Claim: All schedules with no idle time and no inversions have the same lateness.
  - Note: *Most of the reasoning is depend on 'no inversions' because that means only the jobs with same deadline that can have different order, but that order will not cause any change on lateness.
  - Pf.
  - If two different schedules have no inversions or idle time, then they differ in the order in which jobs that have identical deadlines can be scheduled.
  - Let such a deadline be d.
  - Jobs with the deadline d are scheduled consecutively after jobs with earlier deadlines and after jobs with later deadlines.
  - Among jobs with the deadline d only the last one has the greatest lateness, and this lateness does not depend on the order of the jobs.
- Claim: There is an optimal solution that has no inversions and no idle time.
  - Pf.
  - Already claimed there is an optimal solution with no idle time.
    - (a)
      - If O has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has dj < di. Consider an inversion in which job a is scheduled sometime before a job b, and da > db.
      - If we advance in the scheduled order of jobs from a to b one at a time, there has to come a point where the deadline we see decreases for the first time.
      - This corresponds to a pair of consecutive jobs that form an inversion.
    - (b) After swapping i and j we end up with a schedule with one less inversion. When we swap i and j the inversion is fixed and no new inversions are created.
    - (c) The new swapped schedule has a maximum lateness no larger than that of O. Proof on white board.
      - before fixing inversion: r sr fr lr' L' = maxr lr'
      - After fixing inversion: r s-r f-r l-r L- =maxr l-r
      - 
      - both tasks will have the same finish time, we only alter the lateness of job i and job j. we did not increase the lateness of j but potentially increase the lateness of i(i originally before j and di > dj, now changed to j before i)
      -

- l-i = f-i - di = fj - di = f-j - di < fj - dj = lj'
- 
- the new lateness of i is larger than the original lateness of j.

- Claim: The schedule A produced by the greedy algorithm has optimal maximum lateness L.
  - Pf.
  - By last claim, there is an optimal schedule with no inversions and idle time.
  - All schedules with no inversions and no idle time have the same lateness.
  - Therefore the greedy algorithm schedule A is optimal.
  - 
- Greedy schedule S is optimal
  - Pf.
  - Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens. We can assume that S* has no idle time.
  - If S* has no inversions, then S = S*. (There is only one schedule with no idle time and no inversions.)
  - If S* has an inversion, then let i-j be an adjacent inversion. Swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions. Therefore the swapped schedule is at least as good as S (it's also optimal) and has fewer inversions.

- Greedy Analysis Strategies
  - Greedy algorithm stays ahead
    - Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.
  - Exchange argument
    - Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
  - Structural
    - Discover a simple "structural" bound asserting that every possible optimal solution must have a certain value. Then show that your algorithm always achieves this bound.

- Shortest Paths
  - Overview
    - Given a graph and its vertices and edges, how do we find the shortest path from one vertex to another?
    - Clearly, enumerating all of the possible paths, summing the weights of the edges, and taking the minimum is excessively expensive.
    - A pervasive method
      - networking algorithms
      - maps and distances traveled
      - scheduling algorithms
      - or anything else where the weights can be costs, time, distances, etc.
  - Given a weighted, directed graph G = (V , E) with a weight function w : E → R, define the weight of a path p = v0, v1, ..., vk to be the sum of the weights of its constituent edges:

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

We define the **shortest path weight** from $u$ to $v$ as:

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

A **shortest path** from $u$ to $v$ is any path $p$ with weight $w(p) = \delta(u, v)$.

  - Variant of the Shortest Path Problem
    - We'll focus first on the single-source shortest-paths problem (i.e., find the shortest paths given a specific source vertex), but others exist:
      - single-destination shortest-paths problem: find the shortest paths to a given destination from all other vertices
      - single-pair shortest-path problem: given u and v, find the shortest path from u to v
      - all-pairs shortest-paths problem: find a shortest path for every pair of vertices
  - Shortest Path Substructures
    - The algorithms we'll discuss often rely on the fact that a shortest path contains within it other shortest paths.
    - Lemma 24.1
      - Given a weighted, directed graph G = (V , E) with weight function w : E → R, let p = <v0, v1, . . . vk> be a shortest path from v0 to vk and, for any i and j such that 1 ≤ i ≤ j ≤ k, let pij = <vi , vi+1, . . . , vj> be the subpath of p from vi to vj . Then pij is a shortest path from vi to vj .
  - Negative weight edges
    - At times, it makes sense for edges to have negative weights.
      - if there are no cycles in the graph that contain negative weights, then the shortest path weights are well-defined
      - if there are cycles that contain negative edges, the shortest path is not well-defined (since if I found one, I could always find one smaller by going around again)
      - if there is such a negative edge cycle on a path from u to v, we define δ(u, v) = −∞
    - Some algorithms assume that all edge weights are non-negative (so they're basically preceded by a phase that checks this); others can handle negative weights.
  - Representing Shortest Path
    - Usually when we construct the shortest paths, we want more than their weights, we want to know the actual paths, too. We maintain the predecessor subgraph Gπ = (Vπ , Eπ). Vπ is the set of vertices in G that end up having *non-NIL* predecessors (and the source, s):
      - Vπ = {v ∈ V : v.π = nil ∪ s}
      - Eπ is the set of edges induced by the π values for the elements of Vπ :
      - Eπ = {(v.π, v) ∈ E : v ∈ Vπ − {s}}

- Practically, we maintain v.π for each v and create a chain of predecessors that runs back along the shortest path to s. Conceptually, π gives us a tree rooted at s that is a subgraph of G, contains all vertices reachable from s with their shortest paths.

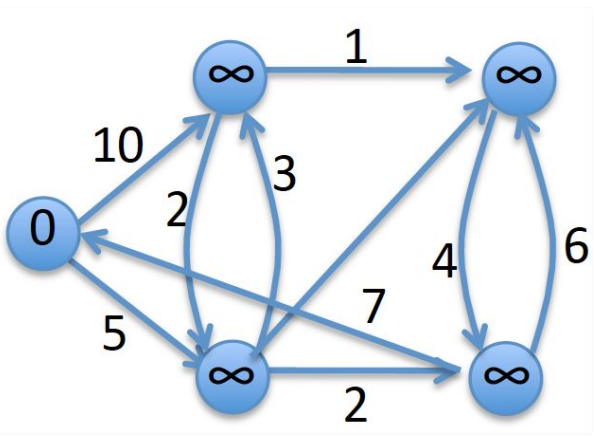
■ Initialization
- For each vertex, we maintain v:d to be an upper estimate on the weight of the shortest path from s to v.

- INITIALIZE-SINGLE-SOURCE(G, s)
- 1 for each vertex v in G.V
- 2 do v.d ← inf.
- 3 v.π ← NIL
- 4 s.d ← 0

■ Relaxation
- Relaxation of an edge (u; v) tests whether we can improve the shortest path known to v by going through u, and, if so, updating v.d and v.π

- RELAX(u, v, w)
- 1 if v.d > u.d + w(u, v)
- 2 then v.d ← u.d + w(u, v)
- 3 v.π ← u

■ Properties of Shortest Path
- Triangle Inequality: For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- Upper-bound property: We always have $v.d \geq \delta(s, v)$ for all $v \in V$ , and once v.d reaches $\delta(s, v)$, it never changes
- No-path property: If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$
- Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path, and if $u.d = \delta(s, u)$ prior to relaxing (u, v), then $v.d = \delta(s, v)$ after
- Path-relaxation property: If p = <v0, v1, . . . , vk> is a shortest path from v0 = s to vk and the edges of p are relaxed in order, then vk .d = δ(s, vk ), regardless of what other relaxations are performed, even if they are interleaved.
- Predecessor-subgraph property: Once $v.d = \delta(s, v)$ for all $v \in V$ , the predecessor subgraph is a shortest path tree rooted at s.

■ Dijkstra's Algorithm
- Dijkstra's algorithm maintains a set S of vertices to which the shortest path has been determined.
  - the algorithm selects a new vertex u with the minimum shortest path estimate of those left in V − S to add to S
  - then relax all of the edges leaving u and repeat
  - we use a minimum priority queue Q of the vertices that are keyed by the d estimates
  - 

  - DIJKSTRA(G, w, s)
    1 INITIALIZE-SINGLE-SOURCE(G, s)
    2 S ← ∅
    3 Q ← G.V
    4 **while** Q ≠ ∅
    5 **do** u ← EXTRACT-MIN(Q)
    6 S ← S ∪ {u}
    7 **for** each vertex v ∈ Adj[u]
    8 **do** RELAX(u, v, w)

  - Shortest path records includes the previous node that gives the shortest path. Therefore, if we chase back, we will know the whole path.

■ Claim: Dijkstra's algorithm terminates with u:d = (s; u) for all u 2 V.
- Pf.
- Base case: |S| = 1 is trivial.
- Inductive hypothesis: Assume true for jSj = k 1
- Let v be the next node added to S and let u-v be the chosen edge.
  - The shortest s-u path plus (u; v) is an s-v path of length d(v).
  - Consider any s-v path P. We'll see that it is no shorter than d(v).
  - Let x-y be the first edge in P that leaves S, and let P0 be the subpath s-x.
  - P is already too long as soon as it leaves S:
    - w(P) >= w(P') + w(x, y) >= d(x) + w(x, y) >= d(y) >= d(v)
  - The last step of the above is because the loop chose v instead of y as the next vertex to include.

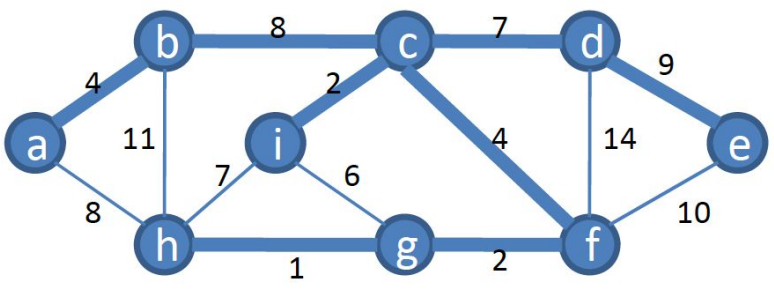
■ Dijkstra's Running Time
- Assume we store the v.d values in an array of size V , indexed by v.
  - initialization takes O(V ) time
  - inserting into Q and decreasing the key of an element of Q both take O(1) time (because we stored the keys in an array indexed by v)
  - EXTRACT-MIN takes O(V ) time
  - the while loop runs O(V ) times, each doing an EXTRACT-MIN, so it's O(V^2) overall
  - by aggregate analysis, the for loop runs O(E) times, and each RELAX call takes O(1) time
  - so the overall running time is O(V^2 + E) = O(V^2)

- Running Time using min-heap
  - If the graph is sparse, we can improve the running time by storing the priority queue in a min-heap
  - Analysis
    - initialization takes O(V )
    - the while loop runs O(V ) times, and EXTRACT-MIN takes O(lg V ) time each time it runs giving O(V lg V ) for the loop
    - by aggregate analysis, the internal for loop runs twice for each edge (once for every entry in an adjacency list), and RELAX contains an implicit DECREASE-KEY, so the time for the inner loop is O(E lg V )
    - so the total running time is O((V + E) lg V ) (or likely O(E lg V ) - because we talk about connected tree, so |E| >= |V| - 1.)

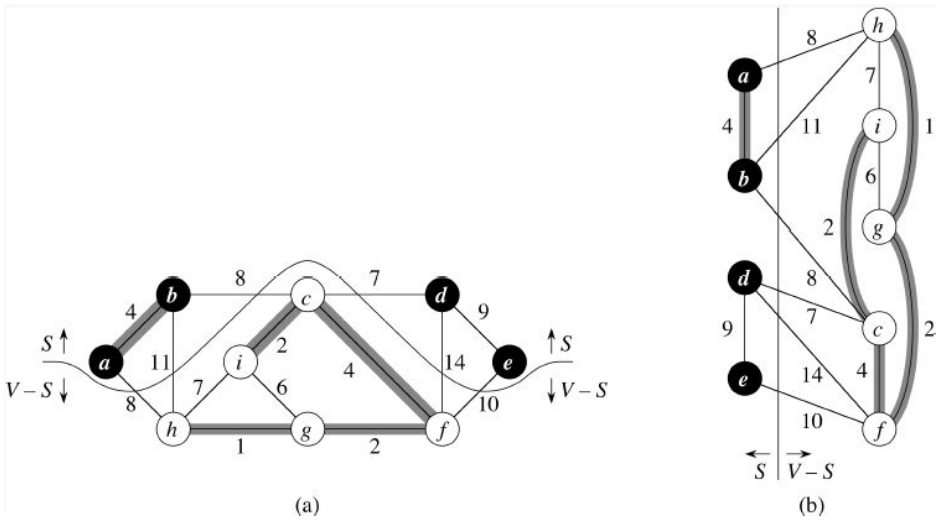
## Minimum Spanning Tree

- Minimum Weight Spanning Tree
- Intro
  - Given a connected undirected graph, $G = (V, E)$ where each edge $(u, v) \in E$ is associated with a weight $w(u, v)$ specifying the cost of the edge, find an acyclic subset $T \subseteq E$ that connects all of the vertices in $V$ and whose total weight:

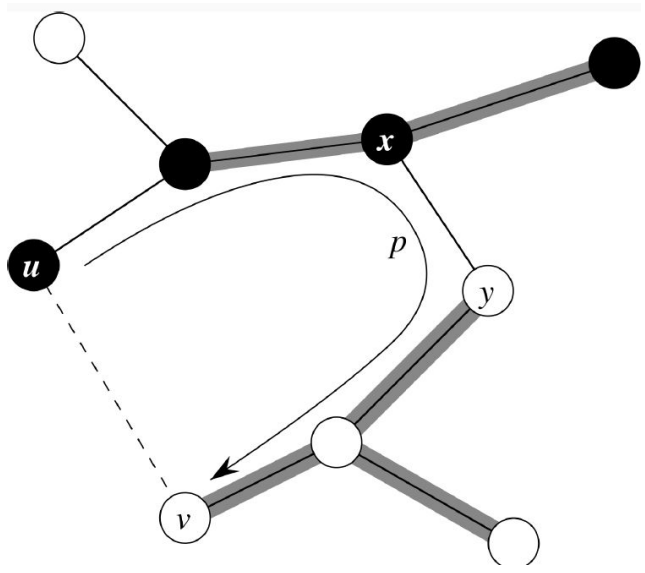
$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

    is minimized.
  - This is a common problem:
    - interconnect a set of pins in electronic circuitry using the least amount of wire
    - find the least latency paths in a network
  - The resulting tree is a spanning tree since it touches all of the nodes in G
    - we want the minimum-weight-spanning tree (or just minimum spanning tree)
  - We look at two algorithms to compute the minimum spanning tree. Both are greedy algorithms
    - greedy refers to the fact that they make the “best” choice at this moment
    - our two minimum spanning tree algorithms do end up minimizing the total weight in the spanning tree
- Example MST
  - The answer is not necessary unique because if we take away bc and add ah, the connected tree is still going to be minimum-weight-spanning, for example.
  - 

- Growing a Spanning Tree
  - First, let’s develop a generic algorithm for computing the minimum spanning tree (MST) by gradually growing the tree. Given a connected, undirected graph, G(V , E) and a weight function w : E → R, find the minimum spanning tree of G. Our approach is to grow a set A of edges that will eventually constitute our minimum spanning tree. What should our loop invariant be?
    - Prior to each iteration, A is a subset of some minimum spanning tree
  - Let a safe edge be any edge that can be added to A while maintaining the invariant. At each step, we select any safe edge (u, v) and add it to A so that A becomes A ∪ {(u, v)}
  - * loop invariant: A is the subset of minimum spanning tree.
  - GENERIC-MST(G, w)
  - 1 A ← Empty Set
  - 2 while A does not form a spanning tree
  - 3 do find an edge (u, v) that is safe for A
  - 4 A ← A U {(u, v)}
  - 5 return A
  - Can you prove it true for this generically stated algorithm?
    - Initialization: initially A is the empty set; it’s a subset of the minimum spanning tree
    - Maintenance: because the edge that we add is selected as a safe edge, it’s guaranteed to maintain the invariant when it’s added to A
    - Termination: upon termination, A is a subset of a minimum spanning tree by the loop invariant, and A is a spanning tree by the negation of the guard on the while loop, so A must be a minimum spanning tree upon termination
- Safe Edge
  - A cut (S, V − S) of an undirected graph G = (V , E) is a partition of V
    - an edge (u, v) **crosses** a cut if one of u and v is in S and the other is in V − S **- Crossing-edge cut**
    - a cut **respects** a set A of edges if no edges in A cross the cut - **A cut respecting A**
    - an edge is **a light edge** crossing a cut if its weight is the minimum of all of the edges that cross the cut




- Thm.
- Let G = (V , E) be a connected undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V − S) be any cut of G that respects A, and let (u, v) be a light edge crossing (S, V − S). Then (u, v) is a safe edge for A.



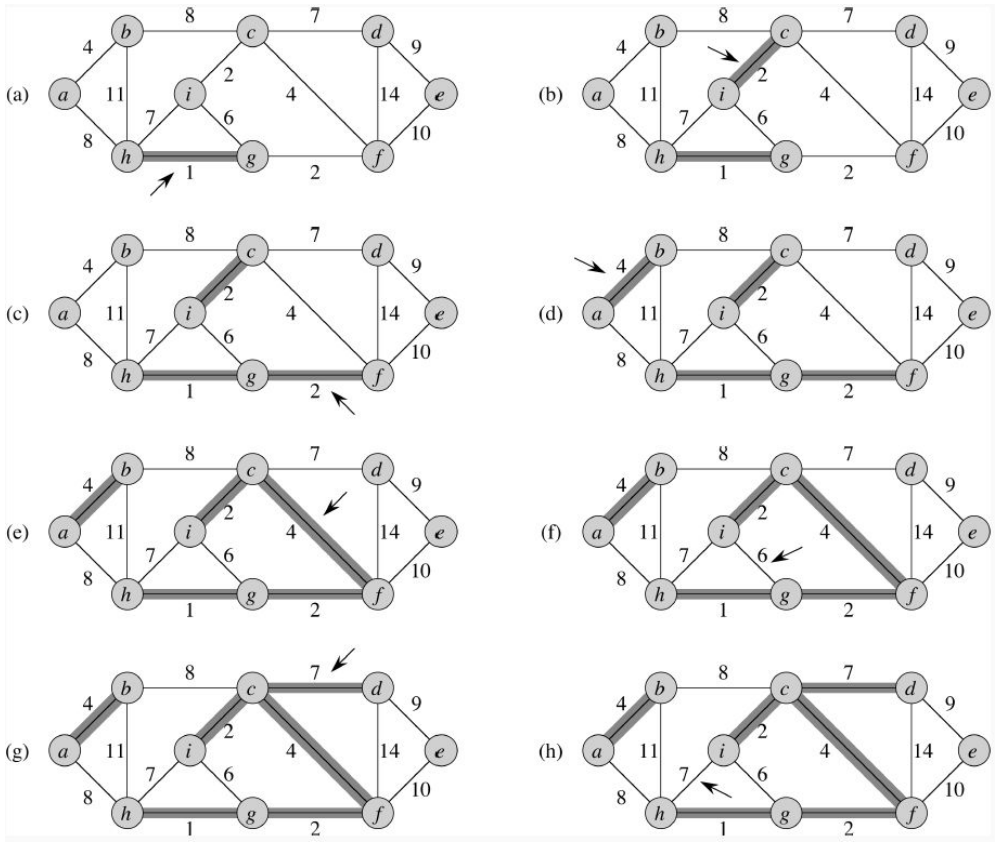

- Pf.
  - Suppose that T is some minimum spanning tree that includes A but does not contain (u, v).
  - We can construct an equivalent minimum spanning tree T  that includes A ∪ {(u, v)}, demonstrating that (u, v) is safe for A.
  - (u, v) forms a cycle with the edges on the path from u to v in T . u and v are on opposite sides of the cut (S, V − S); because T is a spanning tree, there must be at least one edge in T that crosses the cut. Call it (x, y).
  - (x, y) ∈/ A because the cut respects A.
  - Removing (x, y) and replacing it with (u, v) forms a new spanning tree T’ = T − {(x, y)} + {(u, v)}. But is it a minimum spanning tree?
  - (u, v) is a light edge crossing the cut (S, V − S), so w(u, v) ≤ w(x, y).
  - But T was a minimum spanning tree, so it must be that w(u, v) = w(x, y), and w(T’ ) = w(T ).

■ Generic Algorithm
- at any point, G_A = (V , A) is a forest, and each connected component in G_A is a tree
- any safe edge for A connects distinct components from G_A since A ∪ {(u, v)} must be acyclic
- Corollary
  - Let G = (V , E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let C = (V_C , E_C) be a connected component (tree) in the forest G_A = (V , A). If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

■ Kruskal’s Algorithm
- Kruskal’s algorithm builds directly on GENERIC-MST. At each step, the algorithm looks at all of the edges connecting any two trees in the forest G_A and chooses the smallest one.
  - it’s a greedy algorithm because at each step, it just takes the smallest of all of the possibilities

- We use the algorithm for computing the connected components of a graph.
  - we use a disjoint-set data structure to maintain several disjoint sets of elements (initially, each vertex is in its own set)
  - FIND-SET(u) returns a representative element from the set that contains u.
  - we can use FIND-SET to determine whether two vertices u and v belong to the same tree

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ G.V
3       do MAKE-SET(v)
4  sort the edges in G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, in nondecreasing order by weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```




- Running Time
  - initializing A is O(1)
  - initializing the disjoint sets in lines 2-3 is O(V )
  - sorting the edges in line 4 is O(E lg E)
  - the for loop runs for each edge, and does some amount of work that can be proven to be less than O(lg E) for each edge; the running time for the loop is therefore O(E lg E)

■ Prim's Algorithm

- Prim's algorithm differs in that it chooses an arbitrary vertex and grows the minimum spanning tree from there
- A is always a single tree.
- Prim's algorithm is greedy since it always picks the smallest edge that could grow the tree.
- The key challenge is storing and recalling the edges to make it easy to find the right edge to add to A.
- 
- We construct a min-priority queue Q that holds all of the vertices that are not yet in the minimum spanning tree under construction.
  - The priority of each vertex v in Q is the minimum weight of any edge connecting v to any vertex in the tree
  - we also store v.π, the parent on the other end of this minimum weight edge
- Prim's algorithm does not have to explicitly store A, it's just:
  - A = {(v, v.π) : v ∈ V − {r} − Q}
- where r is the provided root vertex

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2       do u.key ← ∞
 3          u.π ←NIL
 4  r.key ← 0
 5  Q ← G.V
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < v.key
10                    then v.π ← u
11                         v.key ← w(u, v)
```

- 
- Prim's Running Time
  - Initialization in lines 1-5 is O(V ) assuming we use a min-heap as our priority queue

- The while loop is executed |V | times, and each EXTRACT-MIN call takes O(lg V ) time, giving us a total here of O(V lg V )
- The for loop is executed O(E) times total, and the implicit call to DECREASE-KEY takes O(lg V ) time

- So the total is O(V lg V + E lg V ) = O(E lg V ), (because for connected graphs |E| ≥ |V | − 1) which is asymptotically the same as Kruskal's since |E| < |V |^2.

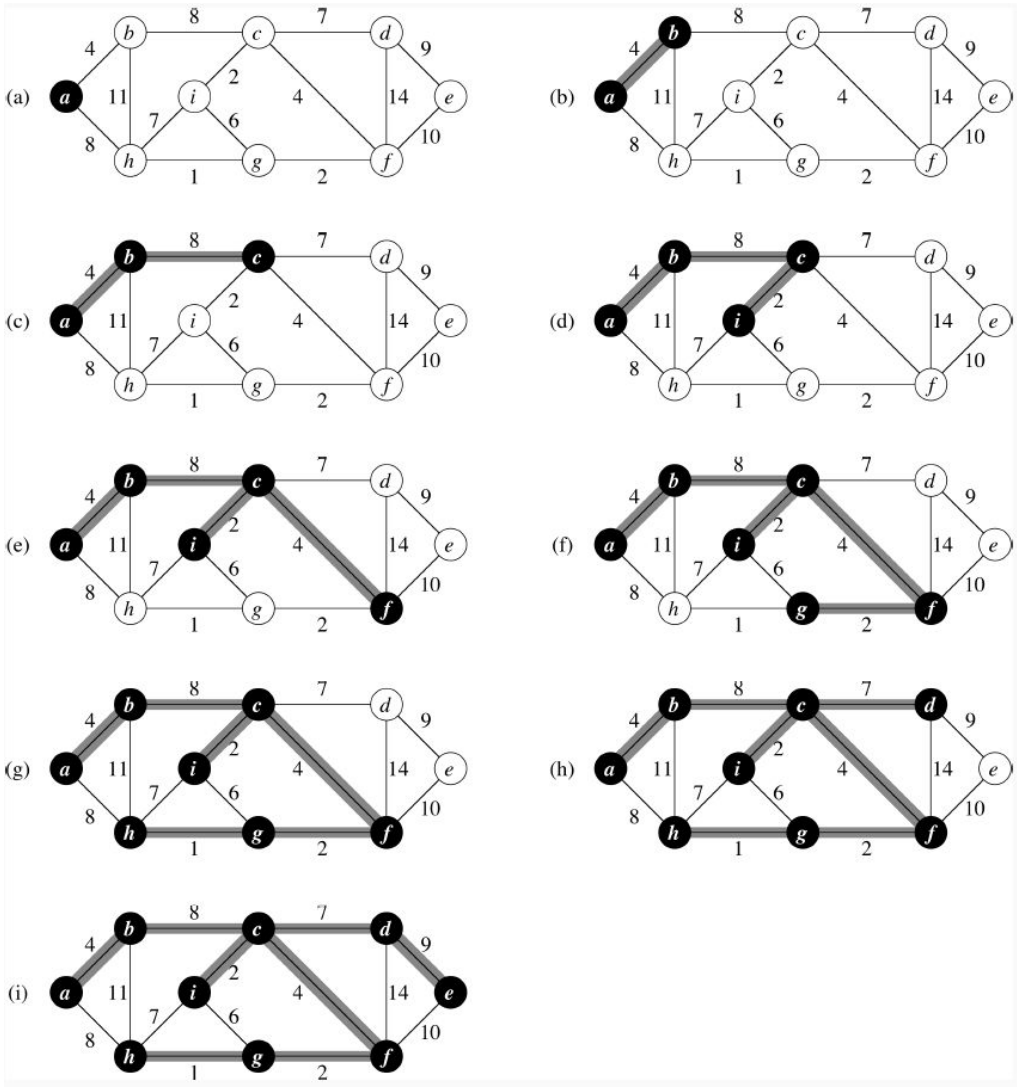
**In Class Exercise**

Let $G$ be a weighted undirected graph, where the edge weights are distinct. An edge is *dangerous* if it is the longest edge in some cycle, and an edge is *useful* if it does not belong to any cycle in $G$.

(a) Prove that any MST of $G$ contains every useful edge.
(b) Prove that any MST of $G$ contains no dangerous edge.

(a) Suppose not. Suppose there is an MST $T$ of $G$ that does not contain some useful edge $e = (u, v)$. $T$ must contain some path from $u$ to $v$. This path must not contain $e$. But then in the original graph, there is a cycle defined by this path (which goes from $u$ to $v$) when it is combined with $e$. This contradicts the fact that $e$ is a useful edge.
(b) Suppose not. Suppose there is an MST $T$ of $G$ that contains an edge $e$, which is part of some cycle. There is some edge $f$ in the same cycle that is not in $T$ (since a tree has no cycles). $f$ must have a lower weight than $e$ (because $e$ is a dangerous edge). Create a new tree $T'$ that contains $f$ but not $e$. $T'$ is a minimum spanning tree with a lower cost than $T$, which is a contradiction.

23/29

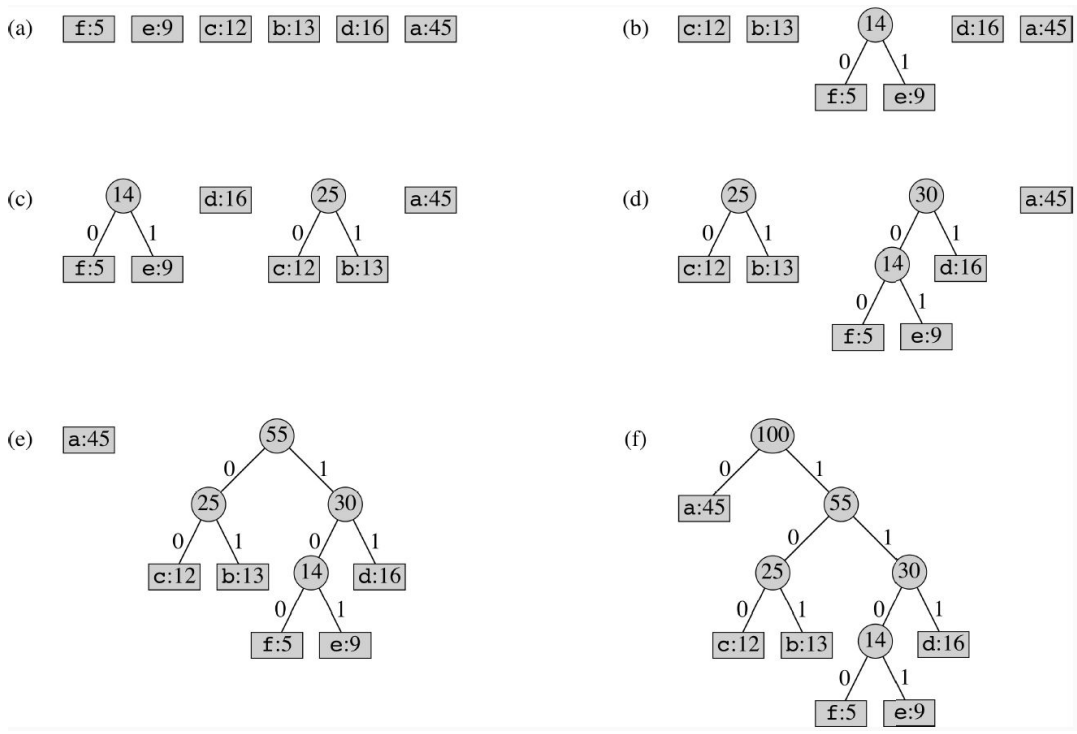


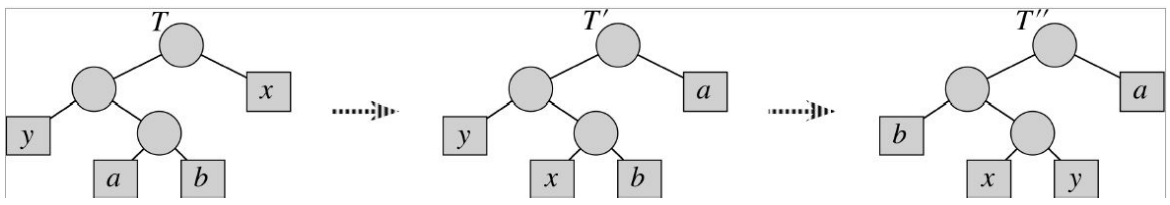

- Notes of Greedy Algorithm (In General)
  - Overview
    - As a general rule, we are pretty direct in applying the greedy method:
      - 1. Cast the optimization problem into one in which we make a choice and are left with only a single subproblem to solve.
      - 2. Prove that there is always an optimal solution to the original problem that makes the greedy choice (i.e., that the greedy choice is safe)
      - 3. Demonstrate that combining a solution to the remaining subproblem with the greedy choice yields an optimal solution to the original problem
  - Identify
    - In general, problems that can be solved by a greedy approach exhibit:
      - optimal substructure — if an optimal solution to a problem contains within it optimal solutions to subproblems
      - greedy-choice property — a globally optimal solution can be arrived at by making a locally optimal choice
  - The Knapsack Problem
    - 0-1 Knapsack Problem
      - A thief robbing a store finds n items; the ith item is worth vi dollars and weighs wi pounds (both integers). The thief wants to take as valuable a load as possible but can carry only W pounds. Which items should he take?
    - Fractional Knapsack Problem
      - The problem is the same as the 0-1 Knapsack problem, but the thief can take fractions of items; he doesn't have to take all or none of a particular item
    - Does the problem display the optimal substructure property?
      - Yes. Take w of item j; then we're left with finding an optimal solution for weight W − w given the remaining n − 1 items plus wj − w of item j
      - Solution
        - calculate vi /wi for each item (the value per pound)
        - sort the items by value per pound

            - take items from the front of the list until the knapsack is full
    - 0-1 Knapsack Problem
        - Does the problem display the optimal substructure property?
            - Yes. If we take item j, we're left with finding an optimal solution for weight W − wj given items {1, 2, . . . , j − 1, j + 1, . . . n}.
        - Does the problem satisfy the greedy-choice property?
            - No. Consider a knapsack that can hold 50 pounds and a set of three possible items: one that weighs 10 pounds and is worth$60, one that weighs 20 pounds and is worth $100, and one that weighs 30 pounds and is worth $120.
        - What happened?
            - By choosing greedily, we didn't fill up the knapsack entirely, resulting in a lower effective value per pound of the knapsack.
        - What we should have done was compare the value of the subproblem that included the greedy choice with the value of the subproblem that didn't.
- Huffman Code
    - Overview
        - Huffman coding is a variable length data compression scheme that takes advantage of the fact that some characters are more common than others
            - characters that are more common get shorter codes
            - avoid ambiguity by requiring that no codeword is a prefix of another
        - For example, if the letters to encode are a, b, c, d, e, f, and the codes are, respectively, 0, 101, 100, 111, 1101, 1100:
            - encoding is easy; just look up the code for each letter and concatenate them
            - decoding is not bad since the coding is "prefix-free"
            - e.g., decode 001011101
    - Prefix code and Binary Tree
        - Any binary prefix code can be described using a binary tree in which codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1".
        - An encoding of S constructed from T is a prefix code. Why? (Proof required for Programming Assignment)
        - Decoding the prefix code: traverse the tree from root to leave, letting the input characters tell us which branch to take.
    - Code
        - Let C be a set of n characters, each with frequency f (ci ) in some file. Construct the optimal codebook of C that minimizes the number of bits needed to represent the file.
        - Given the codetree T , the number of bits needed to encode the file is
            - $B(T) = \sum_{c \in C} f(c) d_T(c)$ where d_T is the depth of c's leaf in T
    - Claim: The binary tree corresponding to the optimal prefix code is full.
        - Pf. by exchange argument
            - Let T be a binary tree corresponding to the optimal prefix code.
            - Suppose it contains a node u with exactly one child v.
            - If u is a root node, then delete u, and use v as the root.
            - If u is not the root, let w be the parent of u. If u is deleted, and make v the child of w. By doing so, the depth of the subtree rooted at u is decreased by 1, and it does not effect the number of leaves.
            - In both case, the new tree T  obtained by deleting u has a smaller average number of bits per leaf than T , contradicting the optimality of T .
    - The Optimal Tree Structure
        - If someone gave us the optimal tree for a prefix code, how would we assign the elements of S to the leaves?
        - Suppose that u and v are leaves of T ∗ (an optimal prefix code tree), and depth(u) < depth(v). Suppose that in a labeling of T ∗ corresponding to an optimal prefix code, leaf u is labeled with y ∈ S and leaf v is labeled with z ∈ S. Then fy ≥ fz .
    - Construct
        - Approach
            - Start with |C| leaves, perform |C| - 1 merging operations to create tree
            - use a min heap keyed on f to merge the two least frequent objects
            - the result is a new object with frequency that's the sum of the frequencies of the merged objects
        - Pseudocode

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9          return EXTRACT-MIN(Q)
```

        - Ex.




- Correctness
  - Lemma
    - Let C be an alphabet in which each c ∈ C has frequency f [c]. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.
  - The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.
  - 

  - Substructure

    **Lemma**

    Let $C$ be an alphabet with $f[c]$ defined for each $c \in C$. Let $x$ and $y$ be characters in $C$ with lowest frequencies. Let $C' = C - \{x, y\} \cup z$ where $f[z] = f[x] + f[y]$. Let $T'$ be any optimal prefix code for $C'$. Then the tree $T$ obtained by replacing the leaf node for $Z$ in $T'$ with an internal node having $x$ and $y$ as children is an optimal prefix code for $C$.
    - Pf.

      **Proof of Huffman Substructure**

      Let $d_T(c)$ be the depth of $c$ in $T$ (i.e., the length of $c$'s code). The cost with respect to all nodes other than $x$ and $y$ is unchanged from $T$ to $T'$.
      $d_T(x) = d_T(y) = d_{T'}(z) + 1$. So

      $$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

      $B(T) = B(T') + f[x] + f[y]$. Rewriting, $B(T') = B(T) - f[x] - f[y]$.

      **Proof by Contradiction.** Suppose $T$ was not an optimal prefix code for $C$. Then there is some $T''$, $B(T'') < B(T)$. Because $x$ and $y$ have the smallest frequencies in $C$, $x$ and $y$ are siblings in $T''$, too (that's the greedy choice property). Construct $T'''$ that is exactly $T''$ except that the subtree rooted at the parent of $x$ and $y$ is replaced by a node $z$, whose frequency is the sum of the frequencies of $x$ and $y$.

      $$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &< B(T') \end{aligned}$$

      But that's a contradition, since $T'$ was supposed to be optimal for $C'$, and now we've found something ($T'''$) for the same $C'$ that's better.

      14/20

July 17th, 2019

# Divide and Conquer

## Designing an algorithm

- Approach to Design
  - **incremental design** – given a sorted subarray, insertion sort inserts a single element into its proper place, generating a longer sorted subarray
  - **greedy choice** – frame the problem as a choice; make the greedy (obvious, na¨ıve) choice
  - **divide and conquer design** – break the problem into several subproblems that are similar to but smaller than the original
    - solve the subproblems recursively

        - combine solutions to subproblems to get a solution to the original problem
- Approach to Divide and Conquer

**Divide, Conquer, Combime**

**Divide:** the problem into a number of subproblems

**Conquer:** the subproblems by solving them recursively. If the subproblems are small enough, just solve the subproblems directly.

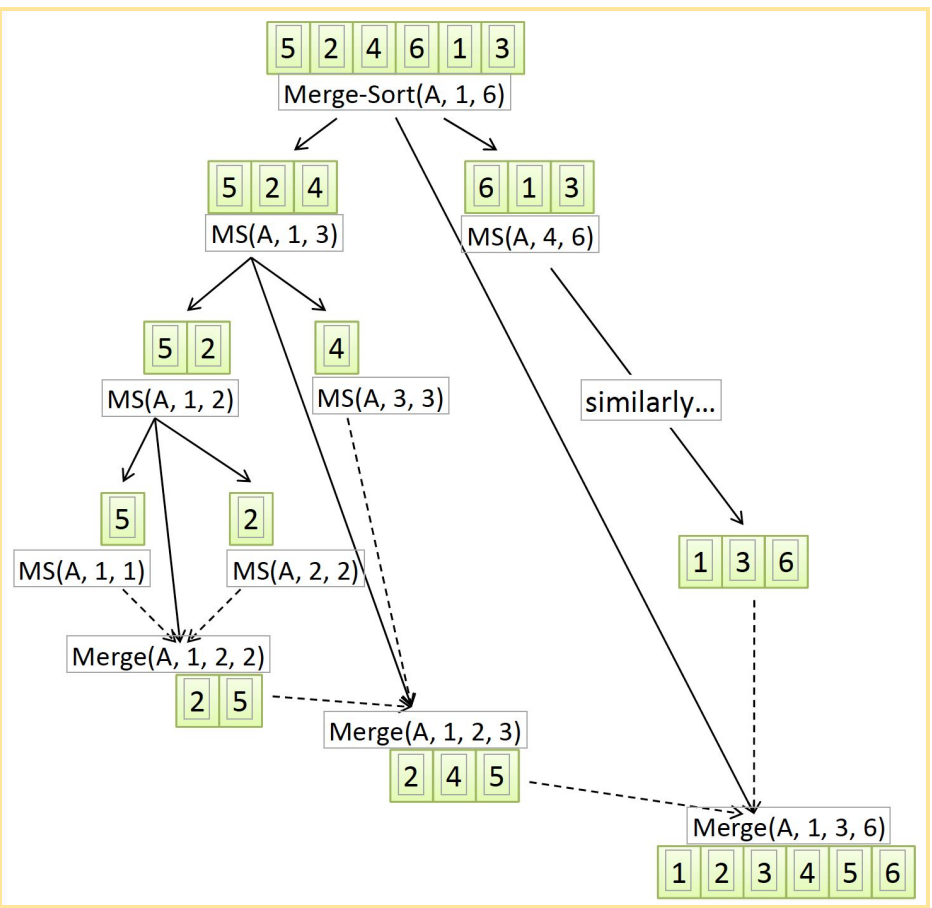
**Combine:** the solutions to the subproblems into the solution for the original problem.

- Ex. MergeSort, Number multiplication, Strassen matrix multiplication etc.
- Ex.1 a^n
  - Naive way
  - Divide and conquer way
- **Analyzing Divide and Conquer Algorithm (Generally)**
  - Idea
    - We often express the running time of a recursive algorithm using a recurrence equation or just recurrence.
    - A recurrence for a divide and conquer algorithm is based on the three steps:
      - divide
      - conquer
      - combine
  - Generic recurrence
    - Let T (n) be the running time of a problem of size n.
    - If the problem size is small enough (i.e., n ≤ c for some constant c), then the straightforward solution takes constant time, or Θ(1).
    - Suppose the divide step generates a subproblems, each of which are a fraction 1/b of the original problem size.
    - Assume that the divide step takes D(n) time and the combine step takes C(n) time.

**General Form**

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- MergeSort
  - Intuition
  - Pseudocode
    - If p < r
      - Then q <- floor of [(p+2)]/2
      - MergeSort(A, p, q)
      - MergeSort(A, q+1, r)
      - Merge(A, p, q, r)
  - Merge code:



```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```

- Explanation:
  - Line 1 computes the length of the subarray A[p . . q] Line 2 computes the length of the subarray, A[q + 1 . . r] Line 3 creates the arrays L and R Lines 4–5 copy the subarray A[p . . q] into L Lines 6–7 copy the subarray A[q + 1 . . r] into R Lines 8 and 9 put sentinels at the ends of L and R Lines 10–17 are the meat of the merging
- Merge Details
  - 1 for k  p to r
  - 2  do if L[i]  R[j]
  - 3  then A[k] <- L[i]
  - 4  i <-  i + 1
  - 5  else A[k] <- R[j]
  - 6  j <- j + 1
  - 
  - Loop variant A
    - At the start of each iteration of the for loop, the subarray A[p . . k − 1] contains the k − p smallest elements, in sorted order
    - L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.
  - Initialization: the loop invariant is true initially. Prior to the first iteration of the loop, k = p, so the subarray A[p . . k − 1] is empty.
  - Maintenance: the loop invariant remains true after each iteration of the loop. Suppose L[i] ≤ R[j]. Because A[p . . k − 1] already contains the k − p smallest elements, after L[i] is copied into A[k], A[p . . k] will

■ Runtime Analysis
- Idea
  - The assignments (Lines 1–3 and 8–11) all take constant time.
  - •  Copying into L and R from A (Lines 4–7) takes n1 + n2 = n where n is the number of elements in A.
  - •  The for loop (Lines 12–17) takes constant time and is executed n times (once for every element).
  - ∴ the running time of MERGE is Θ(n). What is the running time of MERGE-SORT?
- By steps:
  - Divide: this step simply computes the middle of the subarray: Θ(1)
  - Conquer: we define 2 subproblems, each of size n/2, which
  - contributes 2T (n/2) to the running time. Combine: the MERGE procedure takes Θ(n) times
  - Therefore, the worst case running time of MERGE-SORT is:
  - $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$
  - This is a final running time of Θ(n lg n).
    - This is intuitive in this case. We'll solve these generically using the master method.

## ○ Master Method

■ **The master method is a cookbook method for solving these common recurrences**

■ Consider:
- T (n) = aT (n/b) + f (n)
- where a ≥ 1 and b > 1 and f (n) is an asymptotically positive function.
- These often result from divide and conquer approaches.

■ The Master Theorem
- Let a ≥ 1 and b > 1 be constants, let f (n) be a function, and let T (n) be defined on the non-negative integers by the recurrence:
  T (n) = aT (n/b) + f (n)
- where we interpret n/b to be either floor(n/b) or ceiling(n/b). Then T (n) can be bounded asymptotically as follows:
  - 1.If f (n) = O(n^(log_b(a-epsilon)) for some constant epsilon > 0, then T (n) = Θ(n^(log_b a)).
  - 2.If f (n) = Θ(n^(log_b a)), then T (n) = Θ(n^(log_b a) lg n)
  - 3.If f (n) = Ω(n^(log_b(a+epsilon)) for constant epsilon > 0, and f (n/b) ≤ cf (n) for constant c < 1, and n sufficiently large, then T (n) = Θ(f (n)).
  - *The  factor is really n; this is required because the function f (n) must be polynomially different from n^(log_b a).

■ New Master Method
- Solve any recurrence of the form

        - T (n) = aT (n/b) + Θ(n^l (lg n)^k )
        - T (c) = Θ(1) for some constant c
            - where a ≥ 1, b > 1, l ≥ 0, and k ≥ 0.
    - The goal is to compare l and log_b a. The intuition is that n^(log_b a) is the number of times the termination condition (T (c)) is reached.
    - New Master Method cases
        - Case 1: l < log_b a. Then T (n) = Θ(n^(log_b a)).
        - Case 2: l = log_b a. Then T (n) = Θ(f (n) lg n). Which is
        - equivalent to T (n) = Θ(nl (lg n)k+1). Which is ultimately equivalent to (a more generic statement of) what we had before: T (n) = Θ(n^(logb a)* (lg n)^(k+1)).
        - Case 3: l > log_b a. Then T (n) = Θ(f (n)) = Θ(n^l *(lg n)^k ).

## ○ Recursive Multiplication

- Recursive multiplication I

We start recursive multiplication by observing:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

MULTIPLY$(x, y, n)$

```
1  if n = 1
2  return x × y
3     else
4  m ← ⌈n/2⌉
5  a ← ⌊x/10^m⌋; b ← x mod 10^m
6  d ← ⌊y/10^m⌋; c ← y mod 10^m
7  e ← MULTIPLY(a, c, m)
8  f ← MULTIPLY(b, d, m)
9  g ← MULTIPLY(b, c, m)
10 h ← MULTIPLY(a, d, m)
11 return 10^(2m) e + 10^m (g + h) + f
```

Can you write the recurrence? $T(n) = 4T(\lceil n/2 \rceil) + O(n)$

Can you solve the recurrence? Use the master method (revisited) where $a = 4$, $b = 2$, $l = 1$, and $k = 0$. $\log_b a = \log_2 4 = 2$. So $l < \log_b a$, so use Case 1. Then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

- Recursive Multiplication II

We can replace two multiplications with two we're already doing anyway and one additional one.

$$ac + bd - (a - b)(c - d) = bc + ad$$

FASTMULTIPLY$(x, y, n)$

```
1  if n = 1
2  return x × y
3     else
4  m ← ⌈n/2⌉
5  a ← ⌊x/10^m⌋; b ← x mod 10^m
6  d ← ⌊y/10^m⌋; c ← y mod 10^m
7  e ← MULTIPLY(a, c, m)
8  f ← MULTIPLY(b, d, m)
9  g ← MULTIPLY(a − b, c − d, m)
10 return 10^(2m) e + 10^m (e + f − g) + f
```

What's the recurrence for this one? $T(n) = 3T(\lceil n/2 \rceil) + O(n)$

Which solves to? $\Theta(n^{\lg 3}) = \Theta(n^{1.585})$.

- 3 recursion -> 2 recursion

## ○ Matrix Mulitplication

- Brute Force - O(n^3)
- Divide and Conquer
    - Divide: partition A and B into 1/2n by 1/2n blocks
    - Conquer: multiply 8 1/2n by 1/2n recursively
    - Combine: add appropriate products using 4 matrix additions
    - Example:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

    - Calculation

$$\mathrm{T}(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

- Key Idea
  - Multiply 2-by-2 block matrices with only 7 multiplications (7 multiplications and 18 additions/subtractions)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

**Fast matrix multiplication (Strassen 1969)**

- Divide: partition $A$ and $B$ into $\frac{1}{2}n$ by $\frac{1}{2}n$ blocks
- Compute: 14 $\frac{1}{2}n$ by $\frac{1}{2}n$ matrices via 10 matrix additions
- Conquer: multiply 7 $\frac{1}{2}n$ by $\frac{1}{2}n$ matrices recursively
- Combine: 7 products into 4 terms using 8 matrix additions

**Analysis**

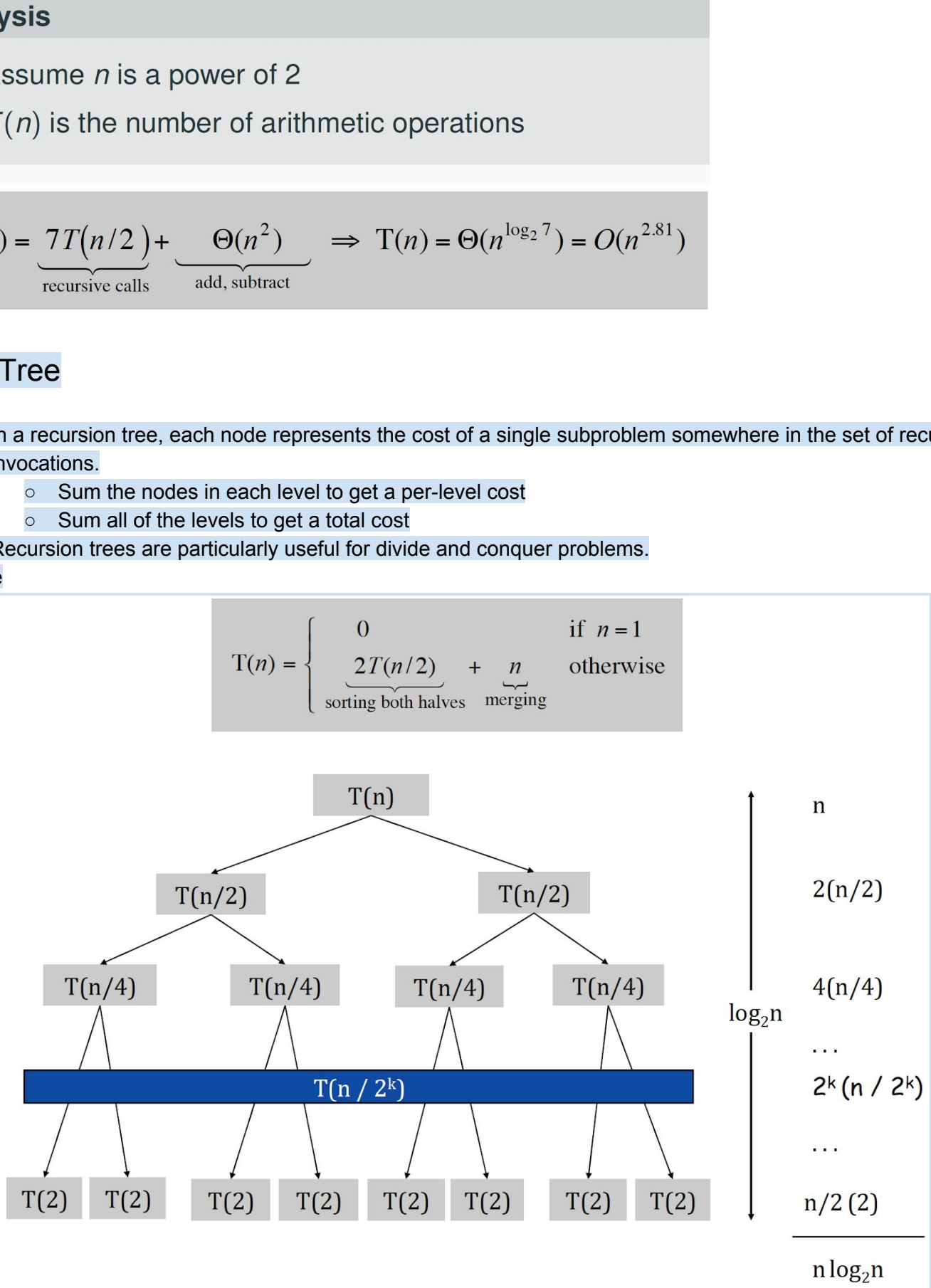
- Assume $n$ is a power of 2
- $T(n)$ is the number of arithmetic operations

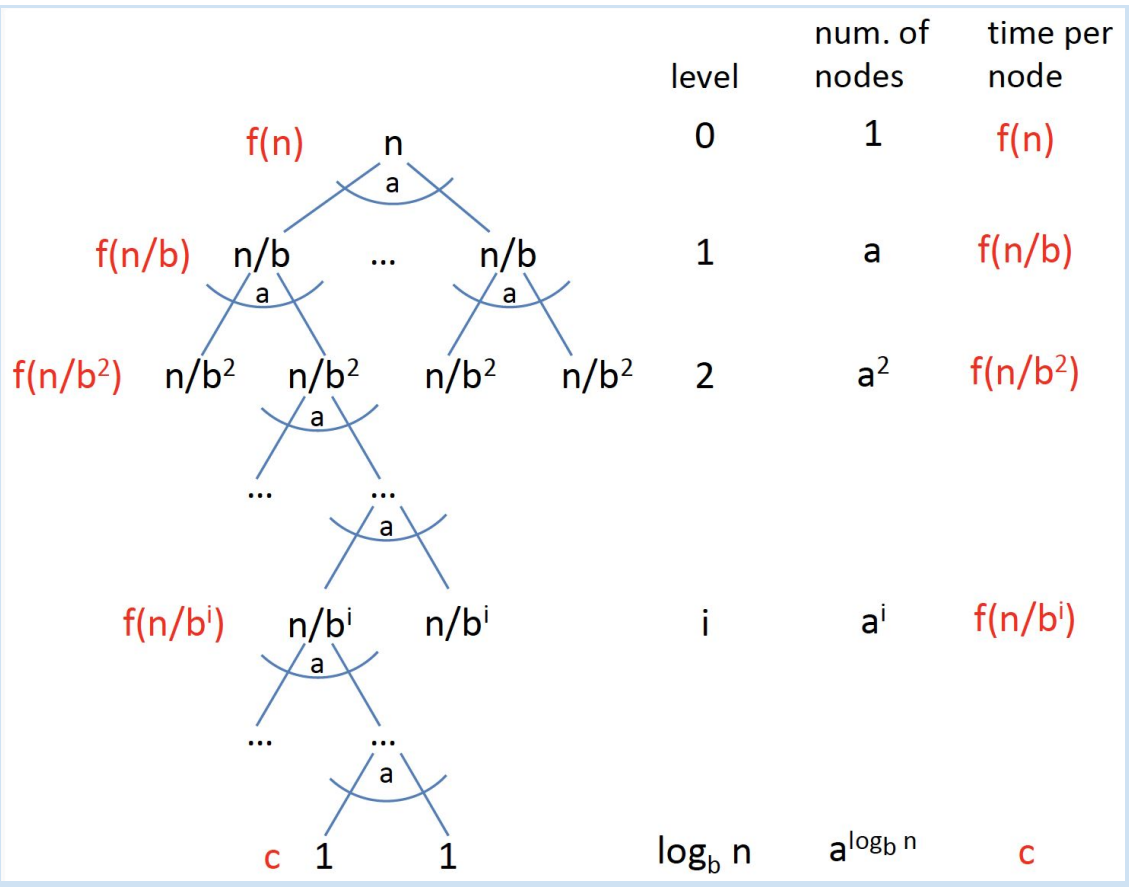
$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

## ○ Recursion Tree

- Idea
  - In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
    - Sum the nodes in each level to get a per-level cost
    - Sum all of the levels to get a total cost
  - Recursion trees are particularly useful for divide and conquer problems.
- Example

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$






- Recursion Tree Summation

Before we get to the full summation, we need the following fact:

$$a^{\log_b n} = n^{\log_b a}$$

You can verify this by taking the $\log_b$ of both sides.

So we can get the running time of the entire recurrence by summing up all of the levels:

$$T(n) = \left[ \sum_{i=0}^{(\log_b n)-1} a^i \times f(n/b^i) \right] + n^{\log_b a} \times c$$

$$T(n) = \left[ \sum_{i=0}^{(\log_b n)-1} a^i \times f(n/b^i) \right] + n^{\log_b a} \times c$$

The term $f(n/b^i)$ represents the running time of a single subproblem at level $i$ of the recursion tree. This is the second term of our general recurrence statement ($T(n) = aT(n/b) + f(n)$). From the master method, we know that it is useful to write $f(n)$ in terms of $l$ and $k$:

$$f(n) = \Theta(n^l (\lg n)^k)$$

Substituting $n/b^i$ for $n$, our sum becomes:

$$T(n) = \left[ \sum_{i=0}^{(\log_b n)-1} a^i \times \Theta((n/b^i)^l (\lg (n/b^i))^k) \right] + n^{\log_b a} \times c$$

- Example on p26 - merge sort
- Recurrences and Induction

### Verifying Recurrence Solutions

Let's say we wanted to verify that this was the correct answer. How would we do it? Induction!

Our recurrence is: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$; our guess is $O(n^2)$.

Our base case is taken care of by the fact that we assume $T(n) = \Theta(1)$ when $n = 1$.

### Inductive Step

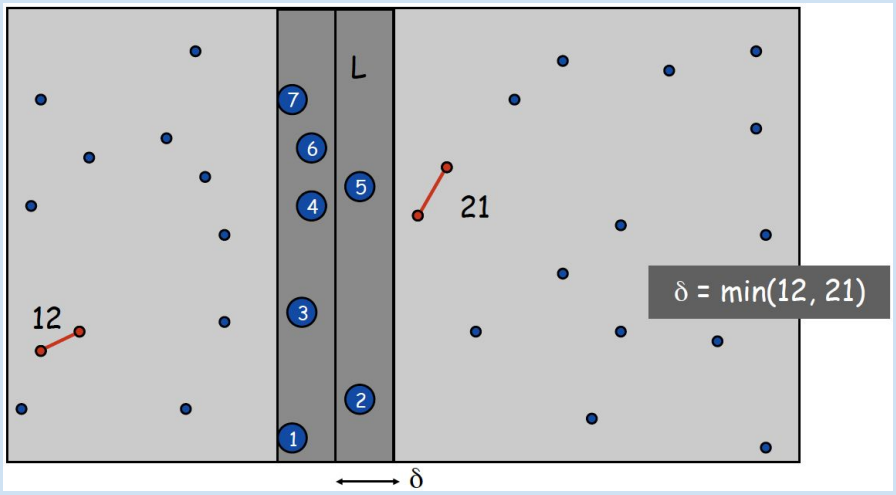
We need to prove that $T(n) \leq cn^2$ for an some $c$. We assume that the claim holds for $\lfloor n/4 \rfloor$, i.e., $T(\lfloor n/4 \rfloor) \leq c(\lfloor n/4 \rfloor)^2$. Then

$$\begin{aligned} T(n) &\leq 3(c(\lfloor n/4 \rfloor)^2) + \Theta(n^2) \\ &\leq 3cn^2/16 + dn^2 \\ &\leq cn^2 \end{aligned}$$

- which is true for values $c \geq (16/3)d$

## ○ Closest Pair

- ■ Defin
  - Given n points in the plane, find a pair with smallest Euclidean distance between them.
- ■ A Fundamental Geometric Primitive
  - Used in graphics, computer vision, geographic information systems molecular modeling, air traffic control
- ■ Brute Forth
  - O(n^2) - check all pairs
- ■ Divide and Conquer
  - Divide: Draw a vertical line L so that roughly n=2 points are on each side of L.
  - Conquer: Find the closest pair in each side recursively.
  - Combine: Find the closest pair with one point in each side. Return the best of the three solutions.
- ■ Combine Step ideas:
  - Find the closest pair with one point on each side, assuming that distance < delta (where delta is the minimum of the closest pair distance in the two halves.
  - We only need to consider points within delta of L
  - Sort points in 2*delta strip by their y coordinate
  - Only check distances of those within 7 positions in the sorted list
    - ○ Pf.
      - ■ Comparing become O(n) instead of O(n^2):
      - ■
      - ■ dl left, dr on the right
      - ■ delta = min(dl, dr)
      - ■ q in L, r in R
      - ■ d(q,r) < delta
      - ■ q=(qx, qy), r = (rx, ry)
      - ■ qx <= x*(the line) <= rx
      - ■ this implies:
      - ■ 1. x* - qx <= rx - qx <= d(q,r) < delta
      - ■ 2. rx - x* <= rx - qx <= d(q,r) < delta
      - ■
      - ■ S is a set of points that x < delta



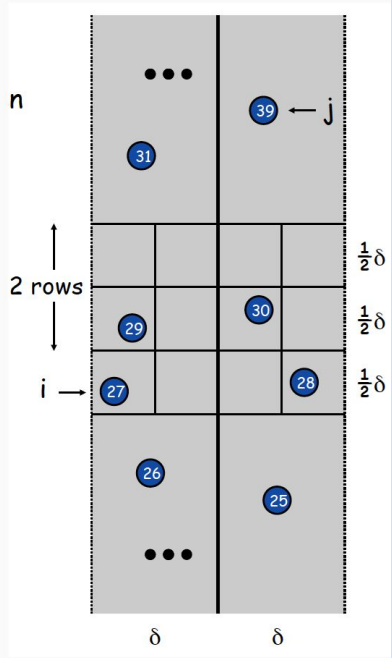
**Definition**

Let $s_i$ be the point in the $2\delta$-strip with the $i^{th}$ smallest $y$ coordinate.

**Claim**

If $|i - j| \geq 12$ then the distance between $s_i$ and $s_j$ is at least $\delta$.

- No two points lie in the same $\frac{1}{2}\delta$ by $\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$.




This is also true if you replace 12 with 7.

CLOSESTPAIR($p_1, p_2, \ldots, p_n$)

1 Compute $L$ s.t. half the points are on each side of $L$
2 $\delta_1$ = CLOSESTPAIR($p_1, \ldots, p_L$)
3 $\delta_2$ = CLOSESTPAIR($p_{L+1}, \ldots, p_n$)
4 $\delta = \min \delta_1, \delta_2$
5 Delete all points further than $\delta$ from $L$
6 Sort remaining points by $y$-coordinate
7 Scan points in $y$ order and compare distance between each point and next 11 neighbors. If any of these distances is less than $\delta$, update $\delta$.
8 **return** $\delta$

- Time to sort original points: $O(n \log n)$
- Divide: $O(n)$
- Conquer: 2 subproblems of size $n/2$
- Combine: $O(n \log n)$

$T(n) = 2T(n/2) + O(n \log n)$

Which solves, by the Master Method, to $O(n \log^2 n)$

Which we can reduce to $O(n \log n)$ by pre-sorting the $y$ coordinates before we start.

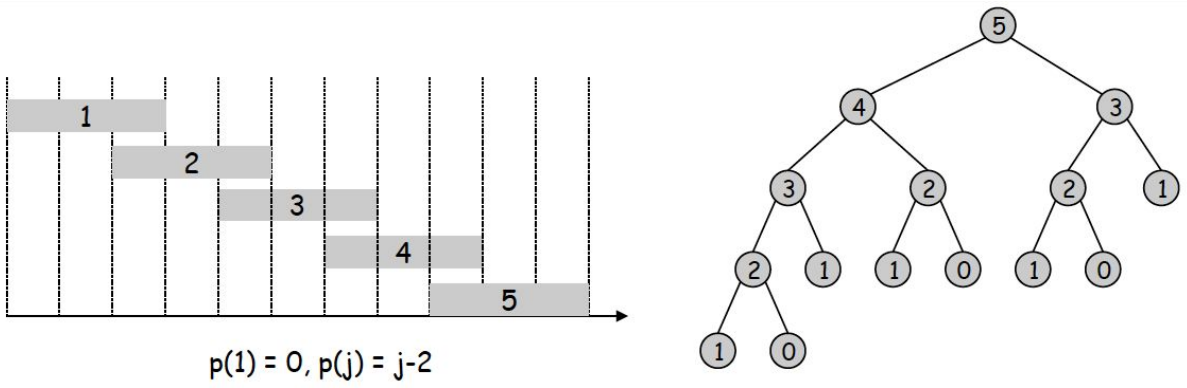
- Dynamic Programming
  - Introduction
    - **Break up a problem into a series of overlapping subproblems and build up solutions to larger and larger subproblems.**
    - Dynamic programming is commonly used for optimization problems in which many possible solutions exist; the algorithm helps us find one of the possibilities
      - every solution to the problem has a value
      - the algorithm helps us find a solution with the optimal value
    - Steps
      - characterize the structure of an optimal solution
      - recursively define the value of an optimal solution
      - compute the value of an optimal solution in a bottom-up fashion
      - construct an optimal solution from the computed information
  - Weighted Interval Scheduling
    - Problem defined
      - Job j starts at sj , finishes at fj , and has weight or value vj
      - Two jobs are compatible if they do not overlap
      - Goal: find maximum weight subset of mutually compatible jobs
    - Method
      - Label (and sort) jobs by finishing time (f1 <= f2 <= … <= fn). Define p(j) to be the largest index i < j such that job i is compatible with job j.
      - Two cases
        - Case 1: Select job i

        - then O can't include any of the jobs in the range p(j) + 1, p(j) + 2, . . . j − 1
        - O must include the optimal solution to the problem consisting of remaining compatible jobs 1, 2, . . . , p(j). If it didn't, we could replace O choice of requests with a better one from 1, 2, . . . , p(j) with no danger of overlapping requests n.
    - Case 2: Do not select job i
        - O must include the optimal solution to the problem consisting of remaining compatible jobs 1, . . . j − 1.
- Recursive Relation:
    - $$\mathrm{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1)\} & \text{otherwise} \end{cases}$$
- Code:
    - ```
      Input: n, s1,…,sn , f1,…,fn , v1,…,vn

      Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

      Compute p(1), p(2), …, p(n)

      Compute-Opt(j) {
         if (j = 0)
            return 0
         else
            return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
      }
      ```




- For faster computation and less repeated computations:
    - Memoization:
        - ```
          Input: n, s1,…,sn , f1,…,fn , v1,…,vn

          Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
          Compute p(1), p(2), …, p(n)

          for j = 1 to n
             M[j] = empty   ← global array
          M[j] = 0

          M-Compute-Opt(j) {
             if (M[j] is empty)
                M[j] = max(wj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
             return M[j]
          }
          ```
    - Analysis of memeoization

      What is the running time of the memoized algorithm? $O(n \lg n)$

      - Sort by finish time: $O(n \lg n)$
      - Computing $p(\cdot)$: $O(n)$ after sorting by start time
      - `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either:
          - returns an existing value `M[j]`
          - fills in one new entry `M[j]` and makes two recursive calls
      - Define a progress measure $\Phi$ as the number of nonempty entries in `M[]`.
          - initially $\Phi = 0$; throughout $\Phi \leq n$
          - a call to `M-Compute-Opt(j)` increases $\Phi$ by at most 1; $\Phi$ cannot be increased more than $n$ times; there are at most $2n$ recursive calls
      - so the overall running time of `M-Compute-Opt(n)` is $O(n)$

- Find solution: i.e. Find the jobs indices

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- Bottom-Up Computation Pseudocode

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

- Generalities
  - For an optimization problem to be a good candidate for dynamic programming, it should exhibit:
    - optimal substructure
    - overlapping problems
  - Optimal Substructure
    - A problem exhibits the optimal substructure property if the optimal solution contains within it optimal solutions to subproblems
      - be careful, though, optimal substructure may also make a problem a good candidate for a greedy solution
    - We use the optimal substructure by then solving the subproblems bottom-up, combining the subproblem solutions as we move up.
  - Overlapping Subproblems
    - If the recursive solution is going to solve the same subproblem multiple times, then we're wasting computation
      - this is in contrast to problems that generate unique subproblems at every split, for which divide and conquer is a good fit
      - one alternative for overlapping subproblems is *memoization*
      - it's like dynamic programming, but it fills the table in using a top-down approach that's more like traditional recursion you create a table to use in a recursive algorithm, and the recursive calls fill in the table as they return results can then be just computed once and looked up thereafter
      - more traditional dynamic programming is "bottom up"
- Segmented Least Square
  - **Least Squares**

    - Foundational problem in statistic and numerical analysis
    - Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find a line $y = ax + b$ that minimizes the sum of the squared error:

    $$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

  - **Solution**

    The solution from calculus tells us that the minimum error is achieved when:

    $$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

    and

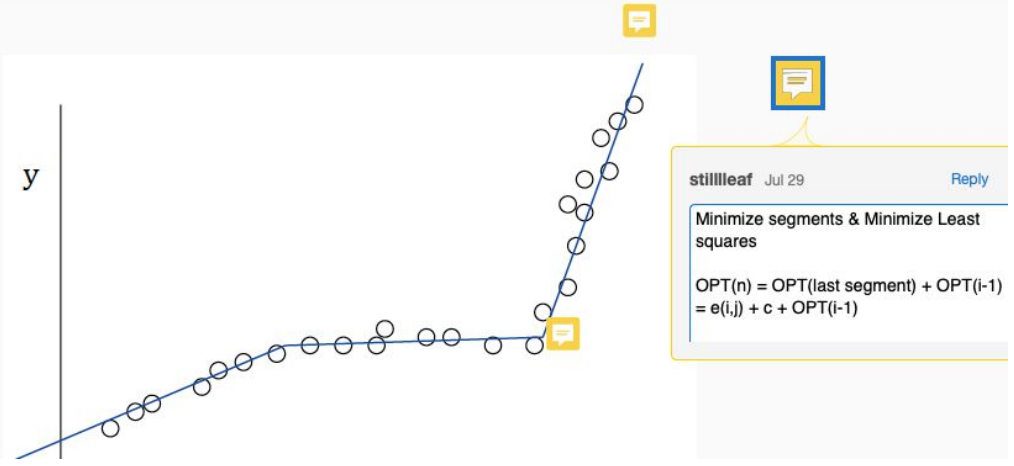
    $$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

## Segmented Least Squares

- Points lie roughly on a sequence of several line segments
- Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x)$

## Question

What is a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)




- **The Tradeoff**

  We have to tradeoff the number of lines for the summed error

  Find a sequence of lines that minimizes

  - The sum of the sums of the squared errors $E$ in each segment
  - The number of lines $L$

  This results in a tradeoff function $E + cL$ for some constant $c$

- **Notation**

  - $OPT(j) =$ minimum cost for points $p_1, p_2, \ldots, p_j$
  - $e(i, j) =$ minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$

- **To Compute $OPT(j)$**

  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$
  - Cost = $e(i, j) + c + OPT(i - 1)$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j}(e(i,j) + c + OPT(i-1)) & \textit{otherwise} \end{cases}$$

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error eij for
         the segment pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (eij + c + M[i-1])

   return M[n]
}
```

- **Running Time**

  - Bottleneck: computing $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ per pair using the previous formula

- Knapsack

**Knapsack Problem**

- Given $n$ objects and a "knapsack"
- Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$
- Knapsack has a capacity of $W$ kilograms
- Goal: fill the knapsack so as to maximize the total value

Definition: $OPT(i, w)$ is the max profit of items $1, \ldots, i$ with weight limit $w$

- Case 1: $OPT$ does not select item $i$
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$ using weight limit $w$
- Case 2: $OPT$ selects item $i$
  - new weight limit $w - w_i$
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max(OPT(i-1, w), v_i + OPT(i-1, w-w_i)) & \text{otherwise} \end{cases}$$

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

**Running Time**

- $\theta(nW)$
- Is this a polynomial-time algorithm?
- Nope. It's "pseudo-polynomial"

- RNA Secondary Structure

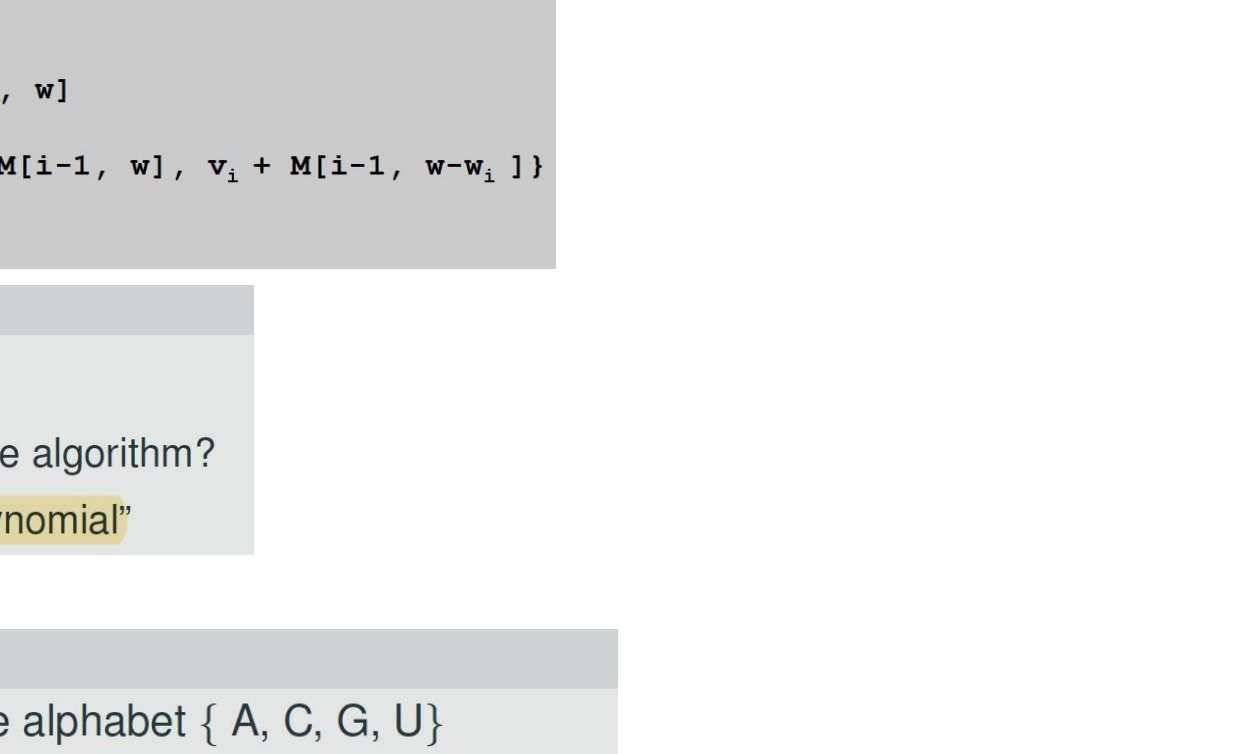
**RNA**

String $B = b_1 b_2 \ldots b_n$ over the alphabet { A, C, G, U}

**Secondary Structure**

RNA is single stranded, so it tends to loop back and form *base pairs* with itself. This structure is often essential to the function of the molecule. Legal base pairs are (A, U) or (C, G).

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



## Secondary Structure

A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- **Watson and Crick:** $S$ is a matching and each pair in $S$ is a Watson-Crick component (matching A to U or C to G)
- **No Sharp Turns:** The ends of each pair are separated by at least 4 intervening bases. That is, if $(b_i, b_j) \in S$, then $i < j - 4$.
- **Non-Crossing:** If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, the we cannot have $i < k < j < l$

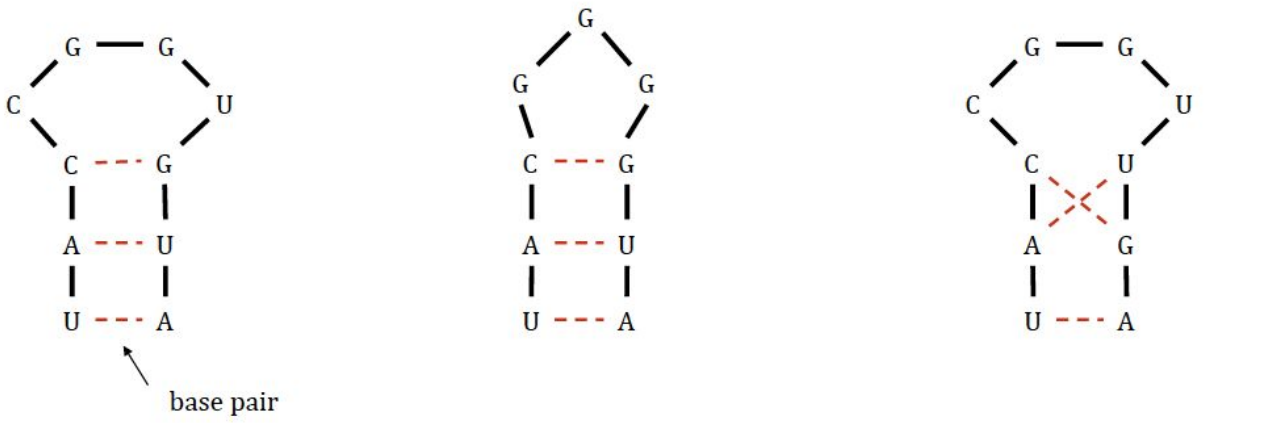
## Free Energy

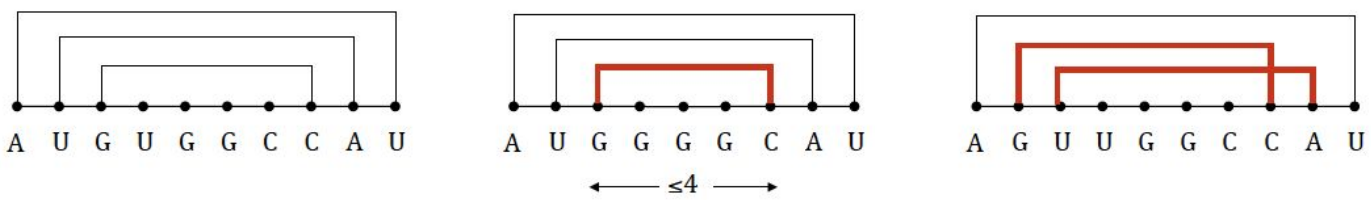
The usual hypothesis is that an RNA molecule will form the secondary structure that has the optimum total *free energy*, which we approximate by the number of base pairs.

## Goal

Given an RNA molecule $B = b_1 b_2 \dots b_n$, find a secondary structure $S$ that maximizes the number of base pairs.

Examples.




ok sharp turn crossing

## Notation

$OPT(i, j)$ is the maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$

- Case 1: if $i \geq j - 4$
  - $OPT(i, j) = 0$ by the no sharp turns condition
- Case 2: Base $b_j$ is not involved in a pair.
  - $OPT(i, j) = OPT(i, j - 1)$
- Case 3: Base $b_j$ pairs with $b_t$ for some $i \leq t < j - 4$
  - The non crossing constraint decouples the resulting subproblems
  - $OPT(i.j) = 1 + \max_t(OPT(i, t-1) + OPT(t+1, j-1))$ such that $i \leq t < j - 4$ and $(b_j, b_t)$ is a Watson and Crick pair

**Do the shortest intervals first**

```
RNA(b1,…,bn) {
    for k = 5, 6, …, n-1
        for i = 1, 2, …, n-k
            j = i + k
            Compute M[i, j]
                                  ↖
                                    using recurrence
    return M[1, n]
}
```

## Recipe

- Characterize the structure of the problem
- Recursively define the value of an optimal solution
- Compute the value of the optimal solution
- Construct the optimal solution from computed information

## Dynamic Programming Techniques

- Binary choice: weighted interval scheduling.
- Multi-way choice segmented least squares.
- Adding a new variable: knapsack
- Dynamic programming over intervals: RNA secondary structure

## Top-down vs. Bottom-up?

Different people have different intuitions

- Coin Changing Problem

## The Problem

You are given $k$ denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money $n$ using as few coins as possible.

## The recursive definition

Let $C[p]$ be the minimum number of coins of the $k$ denominations that sum to $p$ cents. There must exist some "first coin" $d_i$, where $d_i \leq p$. The remaining coins in the optimal solution must be the optimal solution to making change for $p - d_i$ cents. Then $C[p] = 1 + C[p - d_i]$. But which coin is $d_i$? We don't know, so the optimal solution is the one that maximizes $C[p]$ over all choices of $i$ such that $1 \leq i \leq k$ *and* $d_i \leq p$. Also, when $p = 0$, the optimal solution is clearly to have 0 coins:

$$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i:1 \leq i \leq k \wedge d_i \leq p}\{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

### The Algorithm to Compute the Optimal Value

**function** CHANGE($d, k, n$)
  $C[0] \leftarrow 0$
  **for** $p \leftarrow 1$ to $n$ **do**
    $min \leftarrow \infty$
    **for** $i \leftarrow 1$ to $k$ **do**
      **if** $d[i] \leq p$ **then**
        **if** $1 + C[p - d[i]] < min$ **then**
          $min \leftarrow 1 + C[p - d[i]]$
          $coin \leftarrow i$
        **end if**
      **end if**
    **end for**
    $C[p] \leftarrow min$
    $S[p] \leftarrow coin$
  **end for**
  **return** $C$ and $S$
**end function**

- 

### The Algorithm to Reconstruct the Solution

**function** MAKECHANGE($S, d, n$)
  **while** $n > 0$ **do**
    PRINT($S[n]$)
    $n \leftarrow n - d[S[n]]$
  **end while**
**end function**

- 

### Running Time

CHANGE is $\Theta(nk)$ (which is pseudo-polynomial in $k$, the input size and dependent on $n$, just like knapsack was dependent on $W$). MAKECHANGE is $O(n)$ since $n$ is reduced by at least 1 in every iteration of the **while** loop. The total space requirement is $\Theta(n)$

- 

- Longest Common Subsequence

This is a common problem in biological applications: find the longest common sequence of ACTG in two different pieces of DNA

- useful in finding similar proteins, etc.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of $X$ if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of $X$ such that for all $j = 1, 2, \ldots, k$, $x_{i_j} = z_j$.

(For example $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$.)

Our goal is, given two sequences $X$ and $Y$, find the longest common subsequence of the two sequences.

- 

- I.e. i1 = Z[0] = x2; i2 = Z[1] = x3; i3 = Z[2] = x5; i4 = Z[3] = x7.

## Theorem: Optimal Substructure of an LCS

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X$ and $Y$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is an LCS of $X_{m-1}$ and $Y$.
3. if $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$

## Proof (Part I)

If $z_k \neq x_m$, we could append $x_m = y_n$ to $Z$ to get a common subsequence with length $k+1$, which contradicts the premise. So $z_k = x_m = y_n$. And therefore $Z_{k-1}$, the prefix of $Z$ with $z_k$ removed is a longest common subsequence of $X_{m-1}$ and $Y_{n-1}$.

## Proof (Parts II and III)

If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there were a common subsequence $W$ of $X_{m-1}$ and $Y$ longer than $k$, then $W$ would also be a subsequence of $X$ and $Y$, which contradicts the premise that $Z$ was an LCS.

If $x_m = y_n$, there is one subproblem: find the LCS of $X_{m-1}$ and $Y_{n-1}$. If $x_m \neq y_n$, then there are two subproblems to solve: find the LCS of $X_{m-1}$ and $Y$ and the LCS of $X$ and $Y_{n-1}$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

We use dynamic programming to compute solutions to the $\Theta(mn)$ subproblems in a bottom-up fashion

- we create a table $c[0\,..\,m, 0\,..\,n]$ and compute its entries in row major order (filling in the first row, left to right, then the second row, etc.)
- we keep a second table $b[0\,..\,m, 0\,..\,n]$ whose entries point to the table entry that corresponds to the best subproblem for the problem $b[i, j]$ (to help in reconstructing the optimal solution)

```
LCS-LENGTH(X, Y)
 1  m = length[X]
 2  n = length[Y]
 3  for i = 1 to m
 4       c[i, 0] = 0
 5  for j = 1 to n
 6       c[0, j] = 0
 7  for i = 1 to m
 8       for j = 1 to n
 9            if x_i = y_j
10                 c[i, j] = c[i − 1, j − 1] + 1
11                 b[i, j] = ↖
12            else if c[i − 1, j] ≥ c[i, j − 1]
13                 c[i, j] = c[i − 1], j]
14                 b[i, j] = ↑
15            else c[i, j] = c[i, j − 1]
16                 b[i, j] = ←
17  return c and b
```

| $i$ | | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | | B | D | C | A | B | A |
| 0 | $x_i$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | C | | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | B | | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | D | | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | A | | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | B | | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

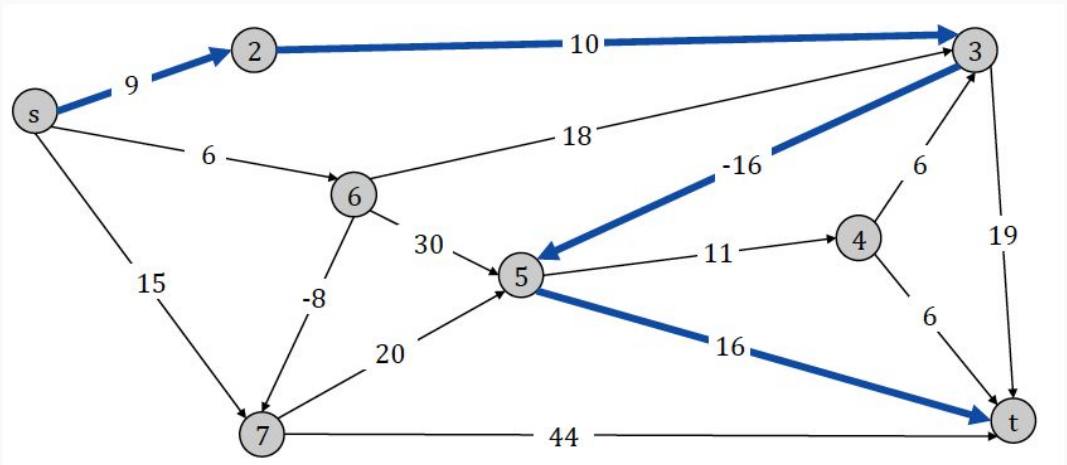
- Shortest Path(With negative weights)

**Shortest Path Problem**

Given a direct graph $G = (V, E)$ with the edge weights $c_{vw}$ (which can include negative edge weights), find the shortest path from node $s$ to node $t$.

**Example**

Nodes represent agents in a financial setting and $c_{vw}$ is the cost of a transaction in which we buy from agent $v$ and sell immediately to $w$




Definition: $OPT(i, v)$ is the length of the shortest $v$-$t$ path $P$ using at most $i$ edges.

- Case 1: $P$ uses at most $i - 1$ edges
  - $OPT(i, v) = OPT(i - 1, v)$
- Case 2: $P$ uses exactly $i$ edges
  - if $(v, w)$ is the first edge, the $OPT$ uses $(v, w)$, and then selects the best $w$-$t$ path using at most $i - 1$ edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ \min(OPT(i-1, v), \min_{(v,w) \in E}(OPT(i-1, w) + c_{vw})) & \text{otherwise} \end{cases}$$

**Remark**

If no negative cycles, then $OPT(n - 1, v) =$ length of shortest v-t path.

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }
}
```

## Complexity

$O(mn)$ time, $O(n^2)$ space

## Finding shortest path

Maintain a ”successor” for each table entry.

## Communication Network

- nodes = routers
- edges = direct communication link
- cost of edge = delay on link (which is naturally non-negative, but we use Bellman-Ford algorithm anyway!)

## Dijkstra’s Algorithm

Requires global information of network

## Bellman-Ford

Uses only local knowledge of neighboring nodes

## Synchronization

We don’t expect the routers to run in lock-step; Bellman-Ford can tolerate asynchronous routing updates and can be shown to still converge on the correct shortest path.

## Distance Vector Protocol

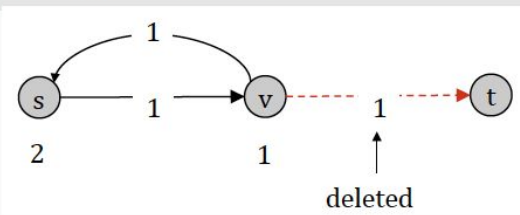
- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions)
- Algorithm: each router performs *n* separate computations, one for each potential destination node
- “Routing by rumor”

## Examples

RIP, Xerox XNS RIP, Novell’s IPX RIP, Cisco’s IGRP, DEC’s DNA Phase IV, AppleTalk’s RTMP

## Caveat

Edge costs may change during the algorithm; inconsistent router state can lead to the counting to infinity problem



- **Link State Routing**
  - Each router stores the entire path (not just the distance and next hop)
  - Based on Dijkstra's algorithm
  - Avoids the counting to infinity problem
  - Requires significantly more storage
- **Examples**
  - Border Gateway Protocol (BGP), Open Shortest Path First (OSPF)