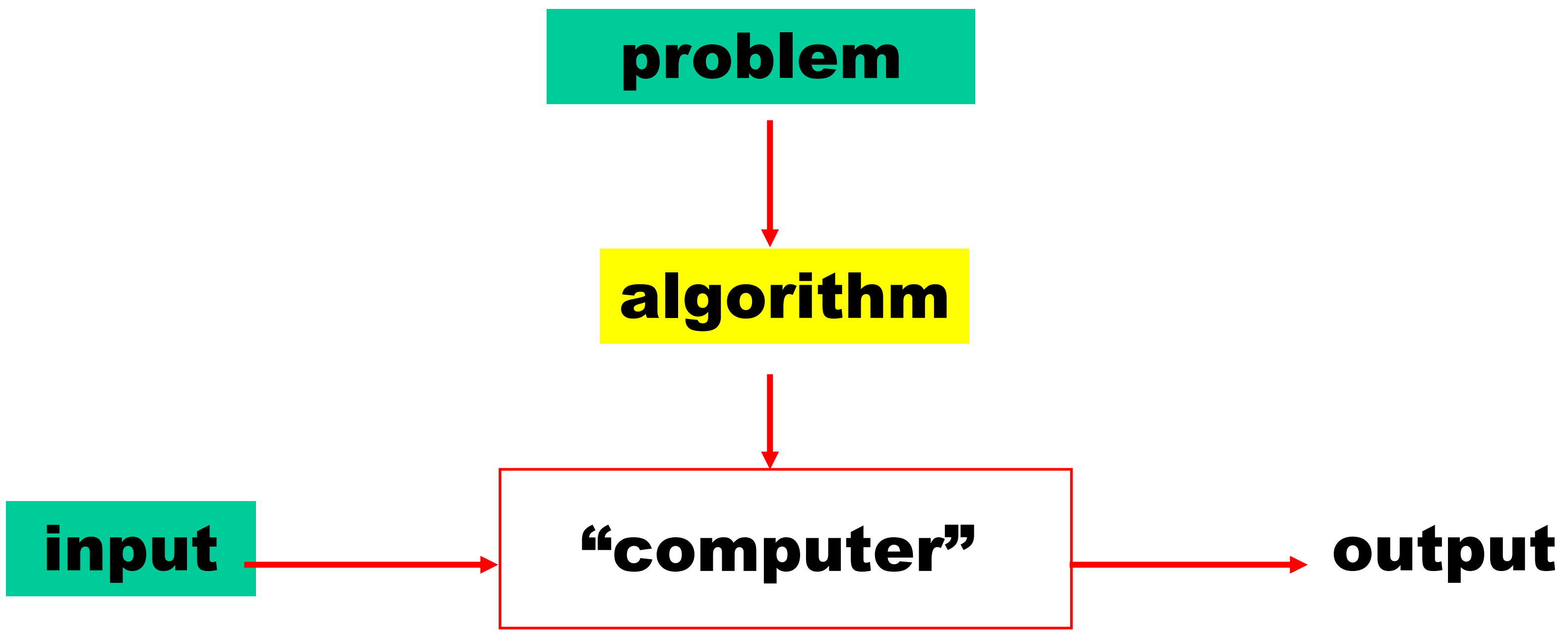
Review For the Final Exam

An ***algorithm*** **is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.**



Some Important Points

Each step of an algorithm is unambiguous

The range of inputs has to be specified carefully

The same algorithm can be represented in different ways

The same problem may be solved by different algorithms

Different algorithms may take different time to solve the same problem – we may prefer one to the other

Some Well-known Computational Problems

Sorting

Searching

String matching

Shortest paths in a graph

Minimum spanning tree

Polynomial time

Traveling salesman problem

Knapsack problem

Assignment problem

Towers of Hanoi …

Algorithm Design Strategies

Brute force: bubble sort, selection sort

Divide and conquer: mergesort, quicksort

Decrease and conquer: insertion sort, DFS traversal, and topological order

Transform and conquer: presorting, balanced binary search tree, and heap

Greedy approach: Prim's algorithm for minimum spanning tree and Dijkstra's algorithm for single-source shortest paths

Dynamic programming: Warshall's Algorithm for transitive closure and Floyd's algorithm for all-pairs shortest paths

Backtracking and branch and bound: n-queen problem, assignment problem, and traveling salesman problem

Space and time tradeoffs: hashing and shift table for string matching

Analysis of Algorithms

How good is the algorithm?

- Correctness
- Time efficiency
- Space efficiency

Does there exist a better algorithm?

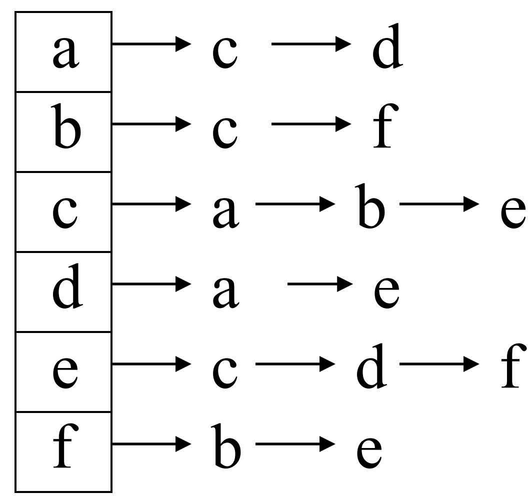
- Lower bounds
  - Trivial lower bound
  - Information-theoretic lower bound

Data Structures

Array, linked list, stack (DFS traversal), queue (BFS traversal), priority queue (heap, greedy approaches), tree (heap, binary search tree, AVL tree), undirected/directed graph, set

Graph Representation: adjacency matrix / adjacency linked list

$$\boxed{\begin{matrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{matrix}}$$



Binary Tree and Binary Search Tree

Tree

- Connected and acyclic graph
- $|E|=|V|-1$

Binary tree – each vertex has no more than two children

Binary search tree – the number associated with the parent is larger than its left subtree and smaller than its right subtree.

The height of a binary tree (the length of the longest path from the root to the leaf) is

$$\lfloor \log_2 |V| \rfloor \le h \le |V| - 1$$

Theoretical Analysis of Time Efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

Basic operation: **the operation that contributes most towards the running time of the algorithm.**

input size

$$T(n) \approx c_{op}C(n)$$

running time

execution time
for basic operation

Number of times
basic operation is
executed

Best-case, Average-case, Worst-case

For some algorithms efficiency depends on type of input:

Worst case: **W(*n*) – maximum over inputs of size *n***

Best case: **B(*n*) – minimum over inputs of size *n***

Average case: **A(*n*) – "average" over inputs of size *n***

- Number of times the basic operation will be executed on typical input
- NOT the average of worst and best case
- Expected number of basic operations repetitions considered as a random variable under some assumption about the probability distribution of all possible inputs of size *n*

Order of growth

Order of growth as $n \to \infty$

Asymptotic Growth Rate: A way of comparing functions that ignores constant factors and small input sizes

- O(*g*(*n*)): class of functions *f*(*n*) that grow *no faster* than *g*(*n*)
- Θ (*g*(*n*)): class of functions *f*(*n*) that grow *at same rate* as *g*(*n*)
- Ω(*g*(*n*)): class of functions *f*(*n*) that grow *at least as fast* as *g*(*n*)

Establishing rate of growth – using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c>0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

L'Hôpital's Rule

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

Basic Asymptotic Efficiency Classes

1	constant
log n	logarithmic
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Analyze the Time Efficiency of An Algorithm

Nonrecursive Algorithm

ALGORITHM $Factorial(n)$

$f \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

  $f \leftarrow f * i$

return f

Recursive Algorithm

ALGORITHM $Factorial(n)$

if $n = 0$

  return 1

else

  return $Factorial(n-1) * n$

Time efficiency of Nonrecursive Algorithms

Steps in mathematical analysis of nonrecursive algorithms:

- Decide on parameter *n* indicating *input size*
- Determine *worst*, *average*, and *best* case for input of size *n*
- Find all the loops
- The operation in the innermost loop is the basic operation
- Write the complexity in the form of summations
- Simplify the expression using formulas in Appendix A

Useful Formulas in Appendix A

Make sure to be familiar with them

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1}) \ldots$$

Time Efficiency of Recursive Algorithms

- Find the recurrence relations and initial conditions
- Find the closed-form solution (Appendix B)
  - Forward substitution
  - Backward substitution
  - Linear 2^{nd} order with constant coefficients (homogenous and inhomogenous cases)
  - Properties of smooth functions
    - $f(2n) \in \Theta(f(n))$
  - Master Theorem

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^k)$

$a < b^k$ $\quad T(n) \in \Theta(n^k)$

$a = b^k$ $\quad T(n) \in \Theta(n^k \log n)$

$a > b^k$ $\quad T(n) \in \Theta(n^{\log_b a})$

Important Recurrence Types:

One (constant) operation reduces problem size by one.

$T(n) = T(n-1) + c$ $\qquad$ $T(1) = d$

Solution: $T(n) = (n-1)c + d$ $\qquad$ *linear*

A pass through input reduces problem size by one.

$T(n) = T(n-1) + cn$ $\qquad$ $T(1) = d$

Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ $\qquad$ *quadratic*

One (constant) operation reduces problem size by half.

$T(n) = T(n/2) + c$ $\qquad$ $T(1) = d$

Solution: $T(n) = c \log_2 n + d$ $\qquad$ *logarithmic*

A pass through input reduces problem size by half.

$T(n) = 2T(n/2) + cn$ $\qquad$ $T(1) = d$

Solution: $T(n) = cn \log_2 n + d\, n$ $\qquad$ *$n \log_2 n$*

Design Strategy 1: Brute-Force

How to develop brute-force algorithms for these problems?

Bubble sort, sequential search, brute-force string matching, exhaustive search for TSP, knapsack, and assignment problem

$89 \overset{?}{\leftrightarrow} 45 \quad 68 \quad 90 \quad 29 \quad 34 \quad 17$

$45 \quad 89 \overset{?}{\leftrightarrow} 68 \quad 90 \quad 29 \quad 34 \quad 17$

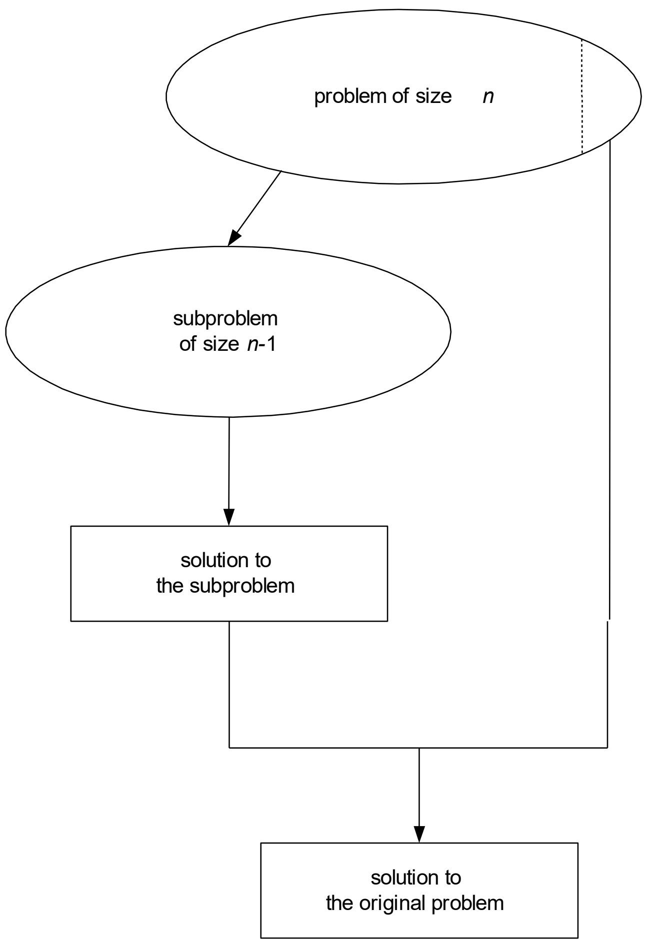
$45 \quad 68 \quad 89 \overset{?}{\leftrightarrow} 90 \overset{?}{\leftrightarrow} 29 \quad 34 \quad 17$

$45 \quad 68 \quad 89 \quad 29 \quad 90 \overset{?}{\leftrightarrow} 34 \quad 17$

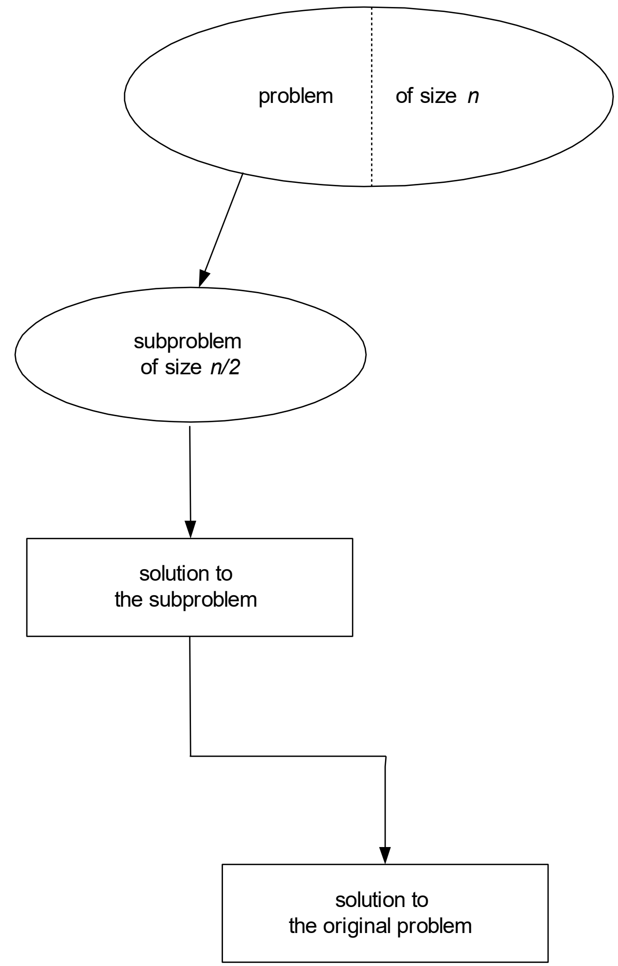
$45 \quad 68 \quad 89 \quad 29 \quad 34 \quad 90 \overset{?}{\leftrightarrow} 17$

$45 \quad 68 \quad 89 \quad 29 \quad 34 \quad 17 \mid \mathbf{90}$

Design Strategy 2: Decrease and Conquer



Decrease by one



Decrease by a constant factor

Insertion Sort

This is a typical decrease-by-one technique

Assume *A*[0..*i*-1] has been sorted, how to achieve the sorted *A*[0..*i*]?

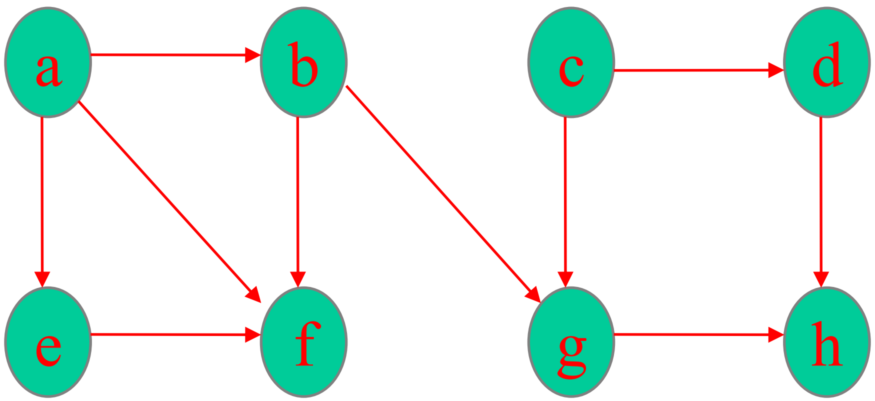
Solution: insert the last element *A*[*i*] to the right position

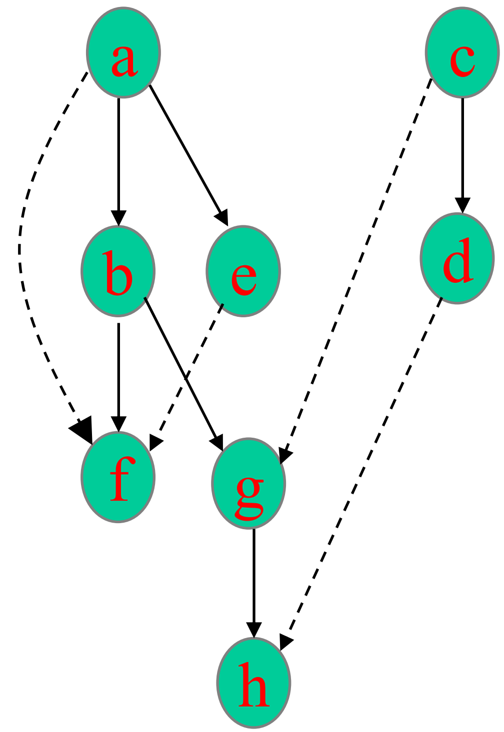
$$A[0] \le \ldots \le A[j] < \mathrm{A}[j+1] \le \ldots \le A[i\text{-}1] \mid A[i] \ \ldots \ A[n\text{-}1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Algorithm complexity: $T_{worst}(n) \in \Theta(n^2), \quad T_{best}(n) \in \Theta(n)$

DFS Traversal: DFS Forest and Stack



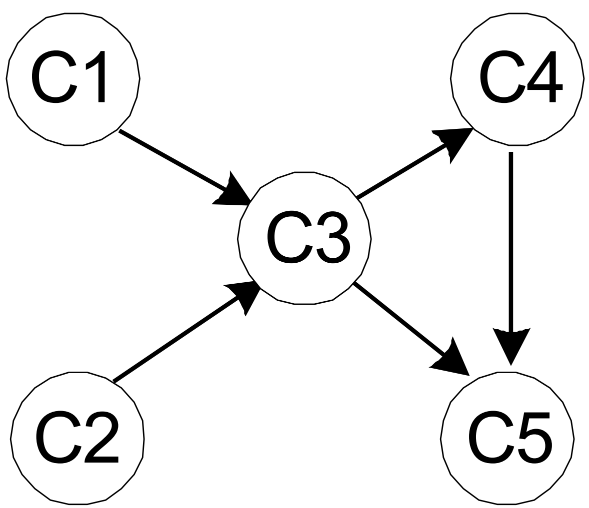


	$h_{5,2}$		
$f_{3,1}$	$g_{4,3}$		
$b_{2,4}$		$e_{6,5}$	$d_{8,7}$
$a_{1,6}$			$c_{7,8}$

Stack push/pop

Tree edges, backward edges, forward edges (directed graph), and cross edges (directed graph),

Topological Sorting



DFS-based algorithm:

DFS traversal: note the order with which the vertices are popped off stack (dead end)

Reverse order solves topological sorting

Back edges encountered?→ NOT a DAG!

Note: problem is solvable iff graph is DAG

Double check if a node is present earlier than its parent in your solution!

Variable-size decrease

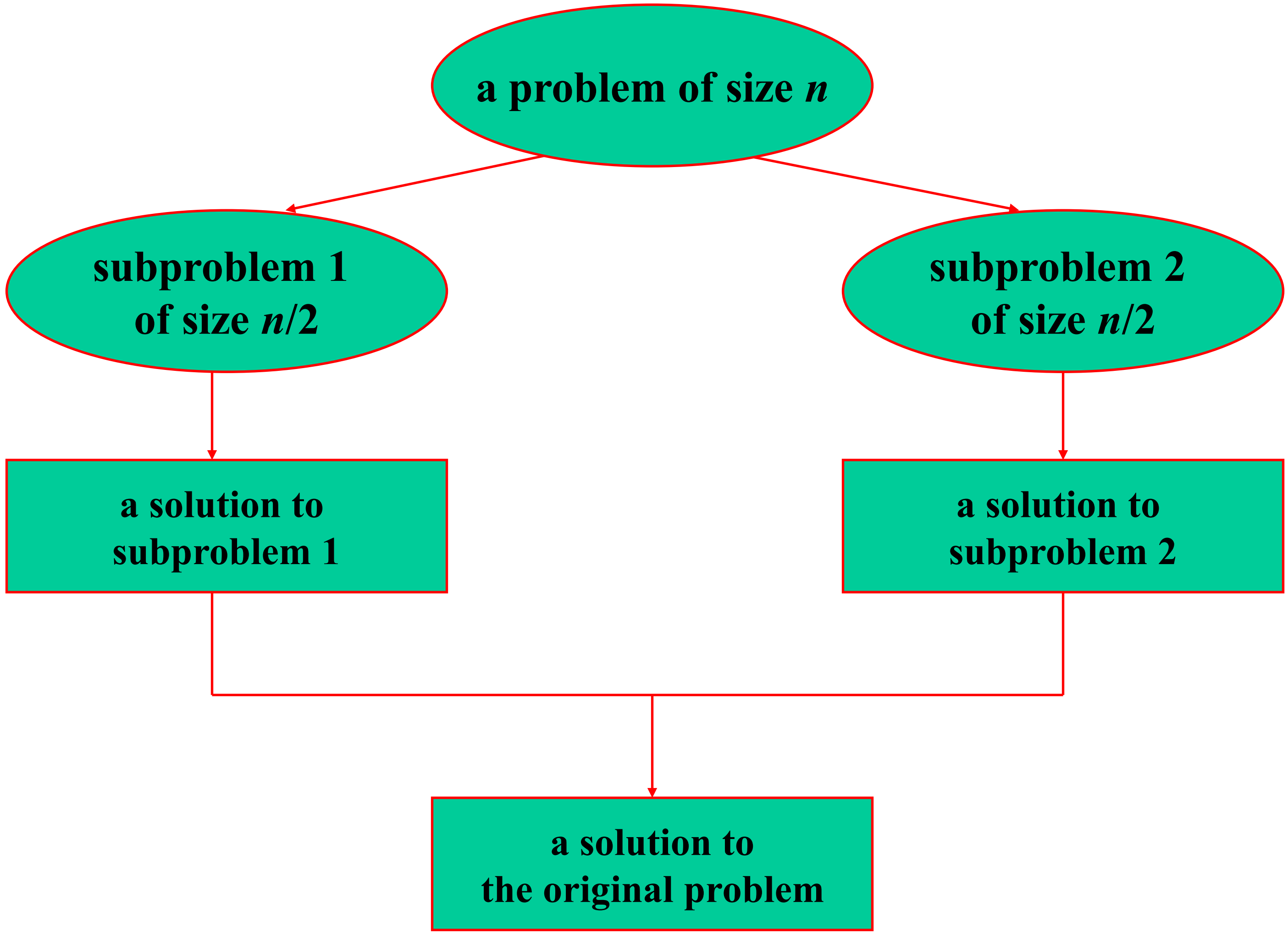
Binary search tree

- Searching and insertion

Selection by partition

- Find the median from a list

Design Strategy 3: Divide and Conquer



Mergesort

Recurrence

$C(n)=2C(n/2)+C_{merge}(n)$ for $n>1$, $C(1)=0$

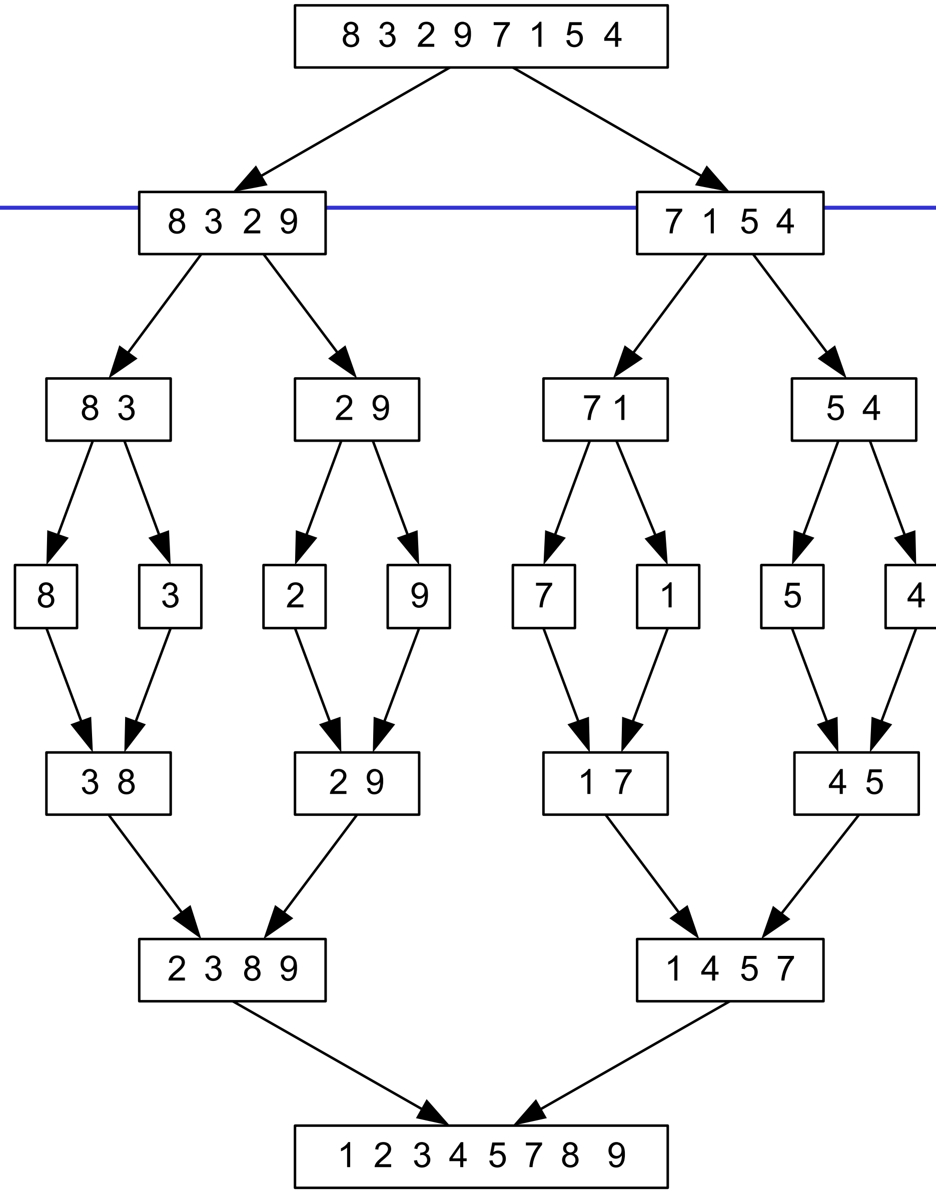
Basic operation is a comparison and we have

$C_{merge}(n)=n-1$ (worst case)

Using the Master Theorem, the complexity of mergesort algorithm is

$\Theta(n \log n)$

It is more efficient than SelectionSort, BubbleSort and InsertionSort, where the time complexity is $\Theta(n^2)$



Quicksort

Basic operation: **key comparison**

***Best case*: split in the middle — $\Theta(n \log n)$**

***Worst case*: sorted array! — $\Theta(n^2)$**

***Average case*: random arrays — $\Theta(n \log n)$**

$\rightarrow$ i $\qquad$ j $\leftarrow$

p	all are $\leq$ p	$\geq$ p	. . .	$\leq$ p	all are $\geq$ p

j $\leftarrow$ $\qquad$ $\rightarrow$ i

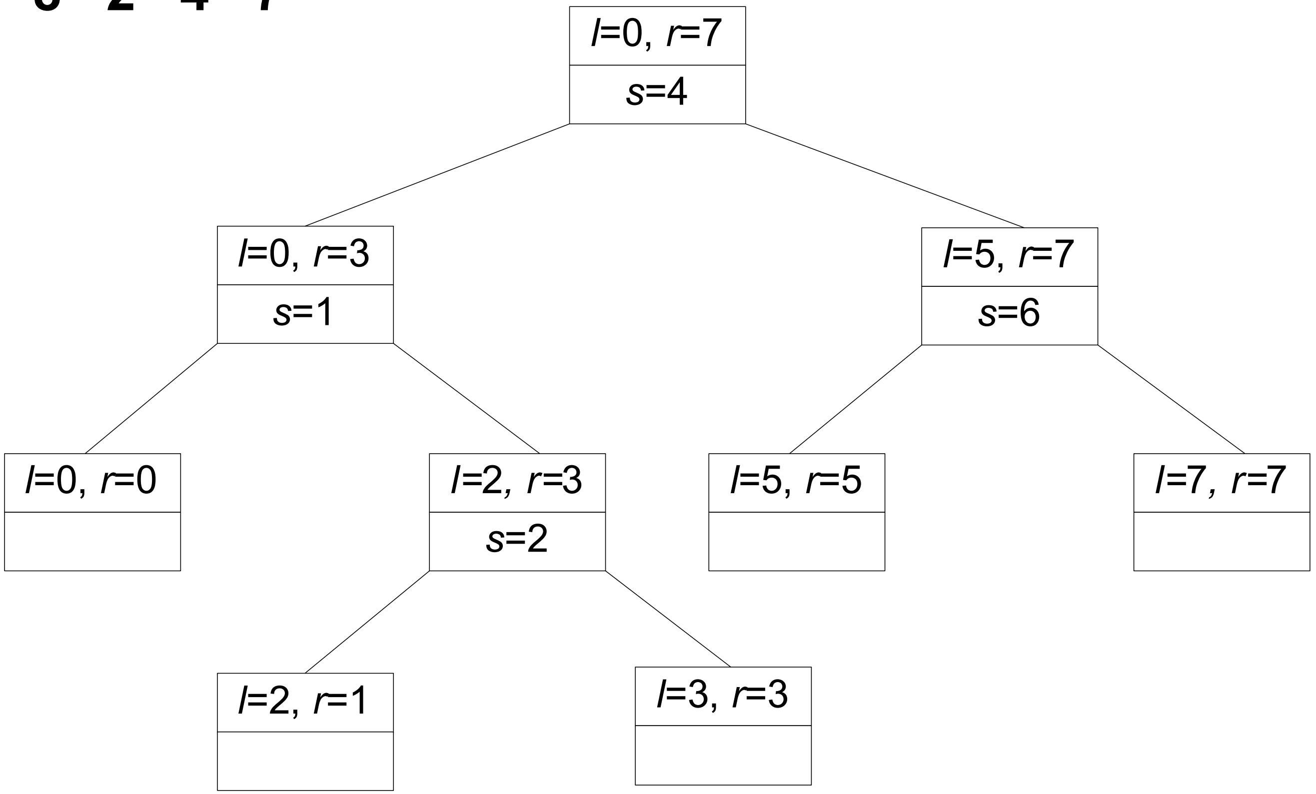
p	all are $\leq$ p	$\leq$ p	$\geq$ p	all are $\geq$ p

$\rightarrow$ i= j $\leftarrow$

p	all are $\leq$ p	= p	all are $\geq$ p

Quicksort Example

5 3 1 9 8 2 4 7



Other Divide-and-Conquer Applications

Binary-tree traversal

- Perform a preorder, inorder, and postorder traversal

How to develop divide-and-conquer algorithms for a given problem?

- Large integer multiplication

Design Strategy 4: Transform-and Conquer

Solve problem by transforming into:

a more convenient instance of the same problem (*instance simplification*)

- Presorting
  - Searching, computing the mode, finding repeated elements, etc
  - Computing the median (selection problem)

a different representation of the same instance (*representation change*)

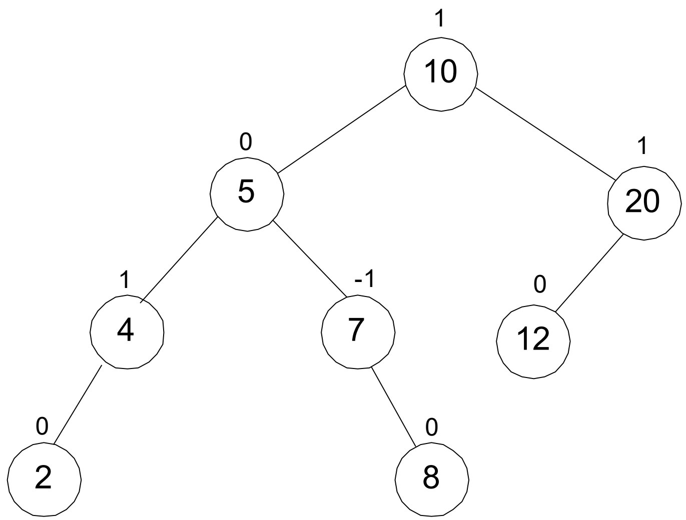
- balanced search trees
- heaps and heapsort

a different problem altogether (*problem reduction*)

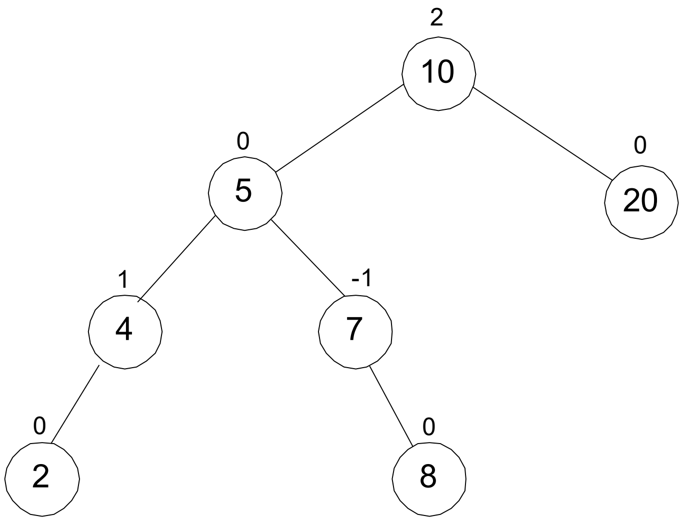
- reductions to graph problems, e.g., Hamiltonian Circuit to decision version TSP

Balanced Trees: AVL trees

For every node, difference in height between left and right subtree is at most 1



An AVL tree



Not an AVL tree

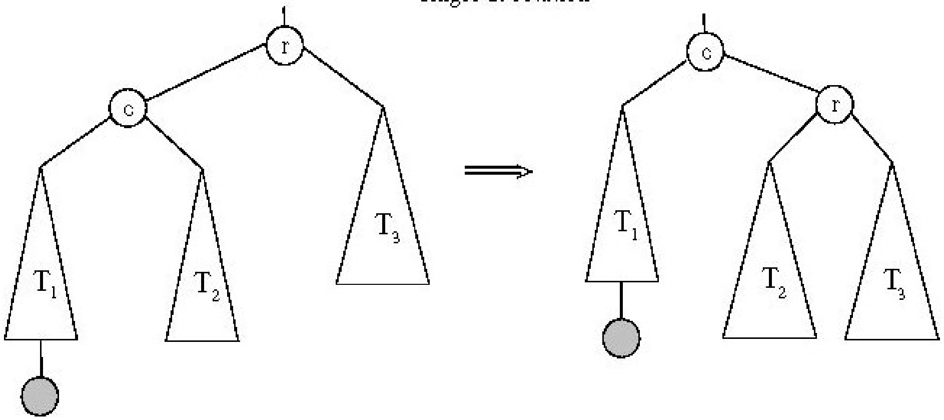
The number shown above the node is its *balance factor*, the difference between the heights of the node's left and right subtrees.

Maintain the Balance of An AVL Tree

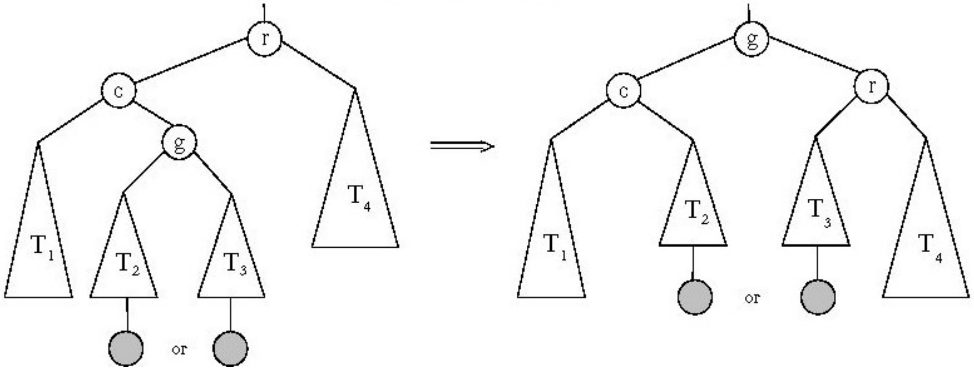
Insert a new node to a AVL binary search tree may make it unbalanced. The new node is always inserted as a leaf

We transform it into a balanced one by *rotation* operations

single R-rotation



double LR-rotation

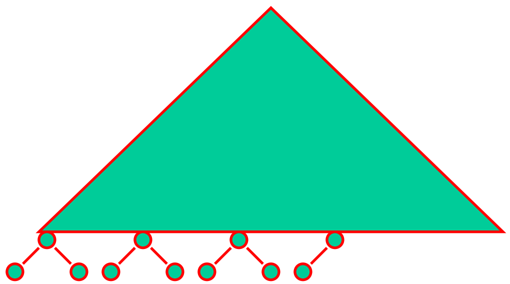


Heap and Heapsort

Definition:

A *heap* is a binary tree with the following conditions:

(1) it is essentially complete: all its levels are full except possibly the last level, where only some rightmost leaves may be missing



(2) The key at each node is ≥ keys at its children

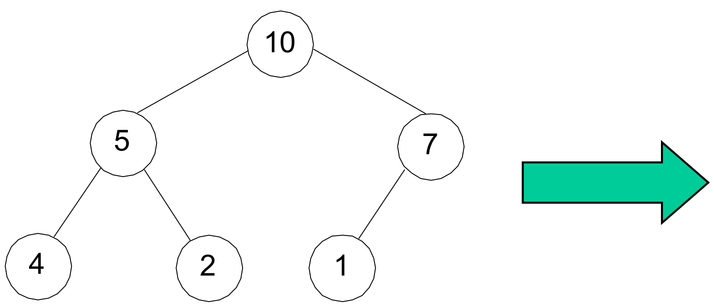
Heap Implementation

A heap can be implemented as an array *H*[1..*n*] by recording its elements in the top-down left-to-right fashion.

Leave *H*[0] empty

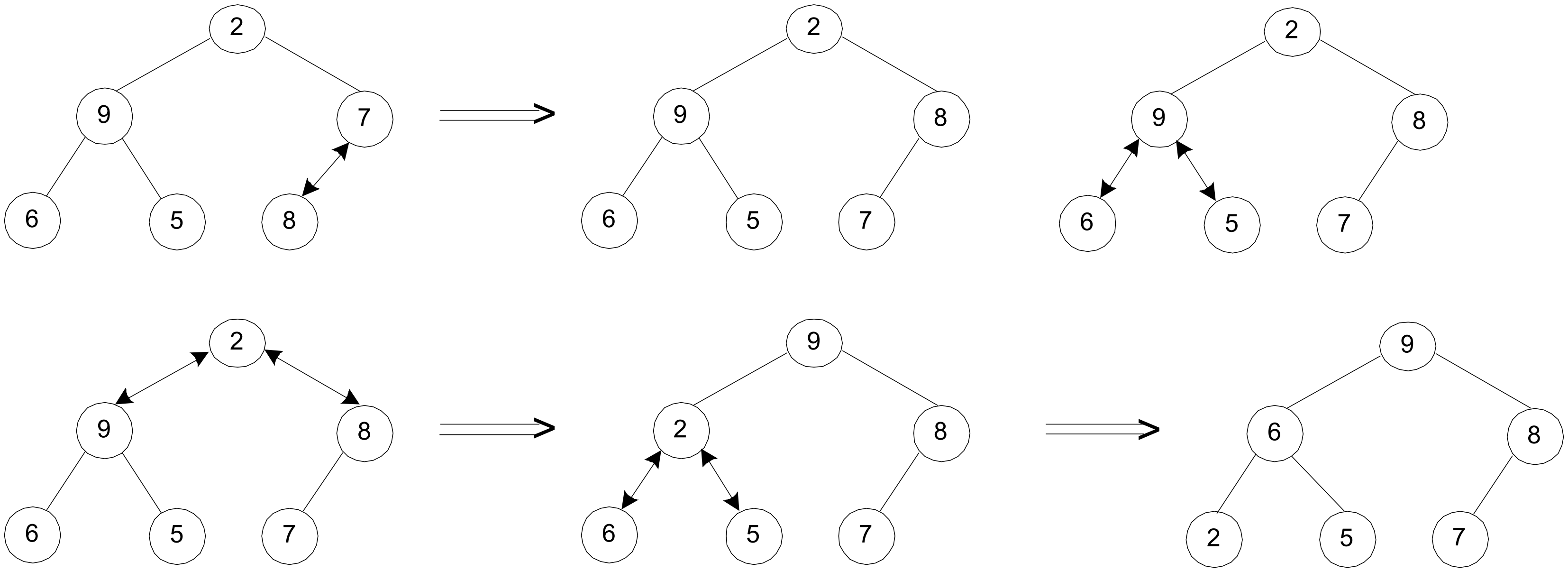
First $\lfloor n/2 \rfloor$ elements are parental node keys and the last $\lceil n/2 \rceil$ elements are leaf keys

***i*-th element's children are located in positions 2*i* and 2*i*+1**



Index	0	1	2	3	4	5	6
value		10	5	7	4	2	1

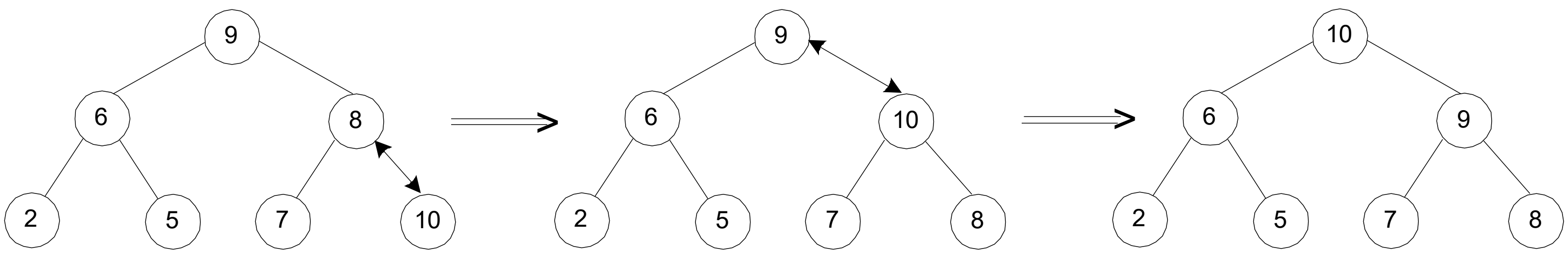
Heap Construction -- Bottom-up Approach



$$BU_{worst}(n) \in \Theta(n)$$

Heap Construction – Top-down Approach

Insert a new key 10 into the heap with 6 keys [9 6 8 2 5 7]



$$TD_{worst} = \sum_{i=1}^{n} \log i \in \Theta(n \log n)$$

Heapsort

Two-Stage algorithm to sort a list of *n* keys

First, heap construction $\mathrm{O}(n)$

Second, sequential root deletion (the largest is deleted first, and the second largest one is deleted second, etc …)

$$C(n) \le 2\sum_{i=1}^{n-1} \log_2 i \in \mathrm{O}(n \log n)$$

Therefore, the time efficiency of heapsort is $\mathrm{O}(n \log n)$ **in the worst case, which is the same as mergesort**

Note: Average case efficiency is also $\mathrm{O}(n \log n)$

Design Strategy 5: Space-Time Tradeoffs

For many problems some extra space really pays off:

extra space in tables

- hashing

input enhancement

- auxiliary tables (shift tables for pattern matching)

tables of information that do all the work

- dynamic programming

Horspool's Algorithm

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \\ \text{characters of the pattern to its last character, otherwise} \end{cases}$$

Shift Table for the pattern "BARBER"

c	A	B	C	D	E	F	…	R	…	Z	_
t(*c*)	4	2	6	6	1	6	6	3	6	6	6

Example

See Section 7.2 for the pseudocode of the shift-table construction algorithm and Horspool's algorithm

Example: find the pattern BARBER from the following text

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O
B A R B E R
        B A R B E R
          B A R B E R
                      B A R B E R
                          B A R B E R
                                B A R B E R
```

Open Hashing

Store student record into 10 bucket using hashing function

$$h(SSN) = SSN \bmod 10$$

xxx-xx-6453

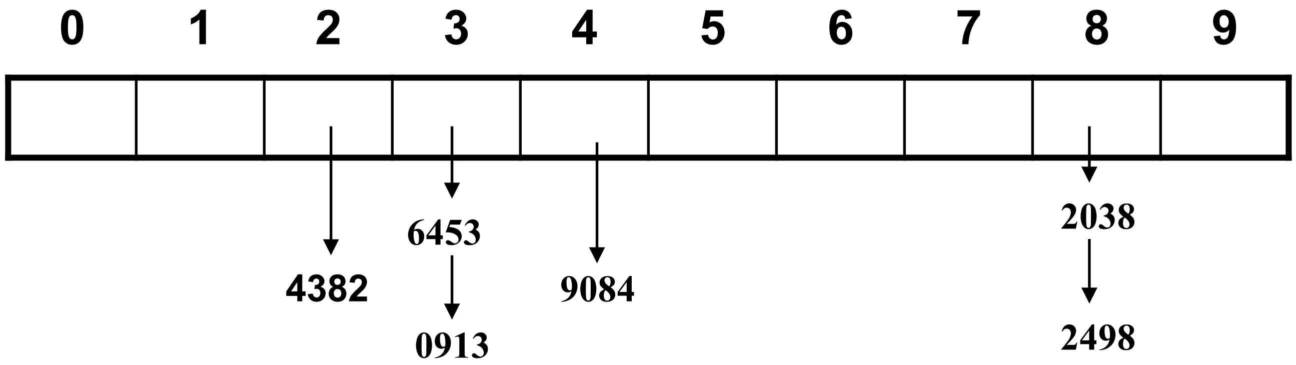
xxx-xx-2038

xxx-xx-0913

xxx-xx-4382

xxx-xx-9084

xxx-xx-2498



Average comparisons: 1/6+1/6+2*1/6+1/6+1/6+2*1/6=4/3

Largest comparisons: 2

Closed Hashing (Linear Probing)

xxx-xx-6453

xxx-xx-2038

xxx-xx-0913

xxx-xx-4382

xxx-xx-9084

xxx-xx-2498

0	1	2	3	4	5	6	7	8	9
		4382	6453	0913	9084			2038	2498

Design Strategy 6: Dynamic Programming

Dynamic Programming is a general algorithm design technique

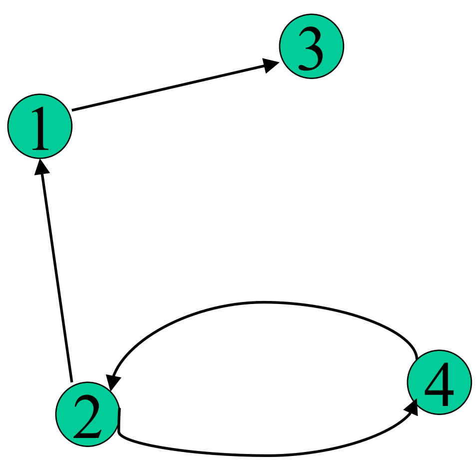
"Programming" here means "planning"

Main idea:

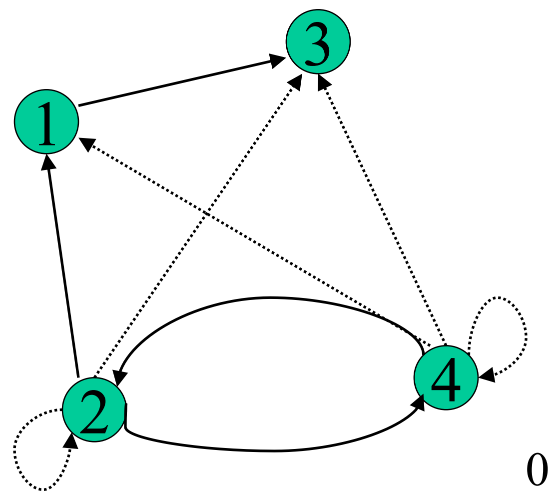
- solve several smaller (overlapping) subproblems
- record solutions in a table so that each subproblem is only solved once
- final state of the table will be (or contain) solution

Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a graph
- (Alternatively: all paths in a directed graph)
- Example of transitive closure:



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

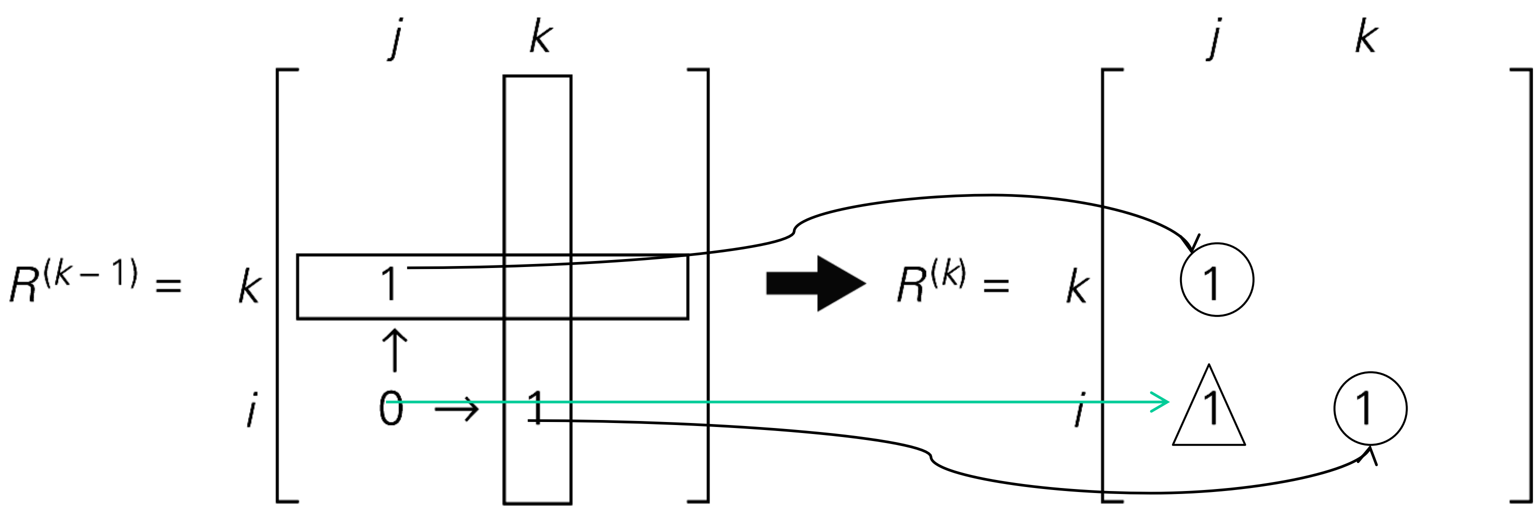


$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

Warshall's Algorithm

In the k^{th} stage determine if a path exists between two vertices i, j using just vertices among 1,…,k

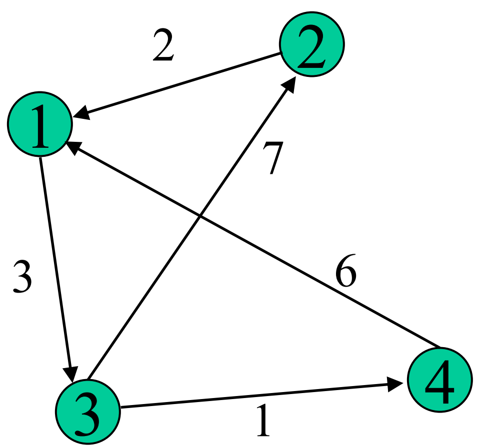
$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1, \ldots, k\text{-}1) \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ AND } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1, \ldots, k\text{-}1) \end{cases}$$



Floyd's Algorithm: All pairs shortest paths

In a weighted graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}$, $D^{(1)}$, … using an initial subset of the vertices as intermediaries.

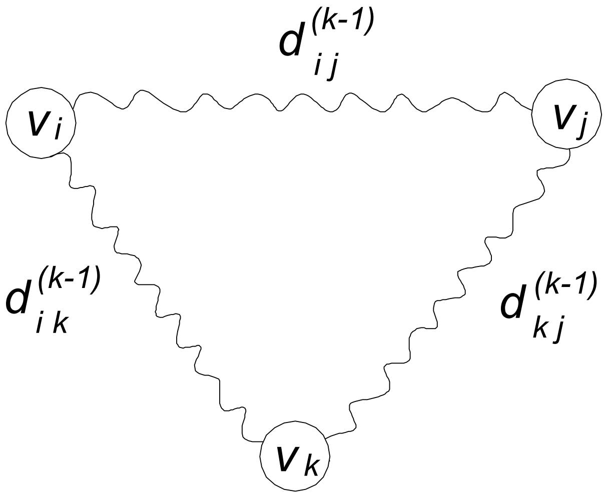


	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	7	7	0	1
4	6	16	9	0

Similar to Warshall's Algorithm

$d_{ij}^{(k)}$ in $D^{(k)}$ is equal to the length of shortest path among all paths from the *i*th vertex to *j*th vertex with each intermediate vertex, if any, numbered not higher than *k*



$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \textbf{ for } k \geq 1, d_{ij}^{(0)} = w_{ij}$$

Design Strategy 7: Greedy algorithms

The greedy approach constructs a solution through a sequence of steps until a complete solution is reached, On each step, the choice made must be

- *Feasible:* Satisfy the problem's constraints
- *locally optimal:* the best choice
- *Irrevocable:* Once made, it cannot be changed later

Optimal solutions:

- Minimum Spanning Tree (MST)
- Single-source shortest paths
- Huffman codes

Approximations:

- Traveling Salesman Problem (TSP)
- Knapsack problem
- other combinatorial optimization problems

Prim's MST algorithm

Start with tree consisting of one vertex

"grow" tree one vertex/edge at a time to produce MST

- Construct a series of expanding subtrees T_1, T_2, …

at each stage construct T_{i+1} from T_i: add minimum weight edge connecting a vertex in tree (T_i) to one not yet in tree

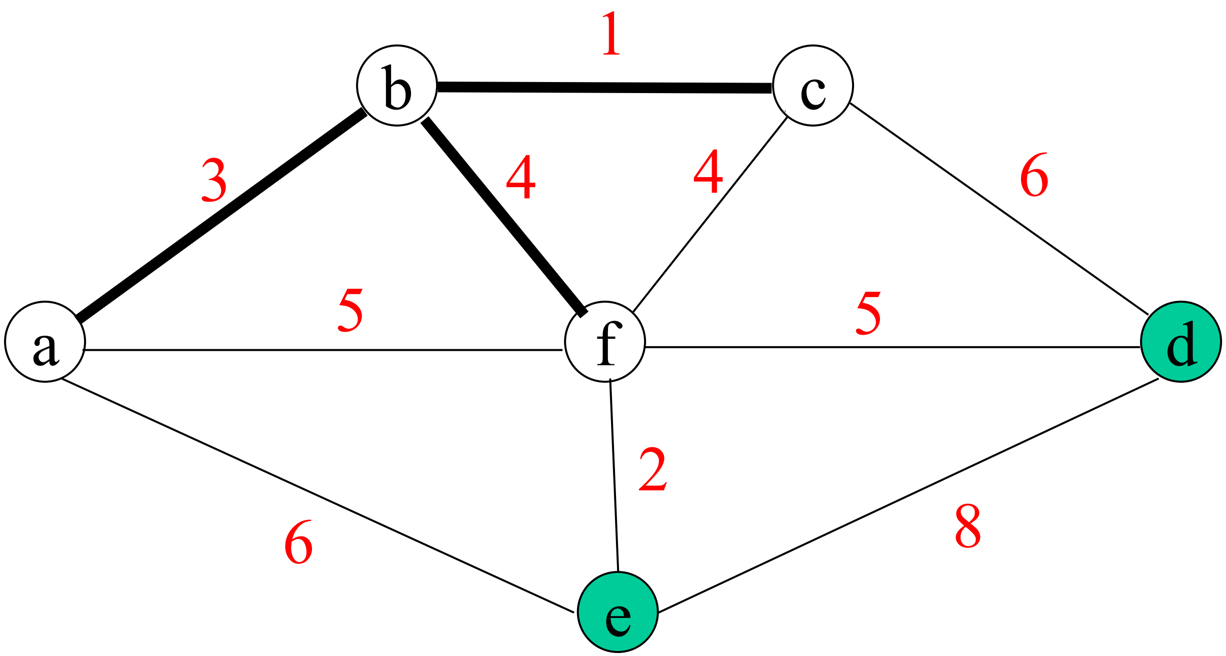
- choose from "fringe" edges
- (this is the "greedy" step!)

algorithm stops when all vertices are included

Step 4:

Add the minimum-weight fringe edge f(b,4) into *T*

Priority queue: d(f,5), e(f,2)



Single-Source Shortest Path: Dijkstra's Algorithm

Similar to Prim's MST algorithm, with the following difference:

- Start with tree consisting of one vertex
- "grow" tree one vertex/edge at a time to produce spanning tree
  - Construct a series of expanding subtrees T_1, T_2, …
- Keep track of shortest path from source to each of the vertices in T_i
- at each stage construct T_{i+1} from T_i: add ~~minimum weight edge~~ connecting a vertex in tree (T_i) to one not yet in tree
  - choose from "fringe" edges
  - (this is the "greedy" step!)

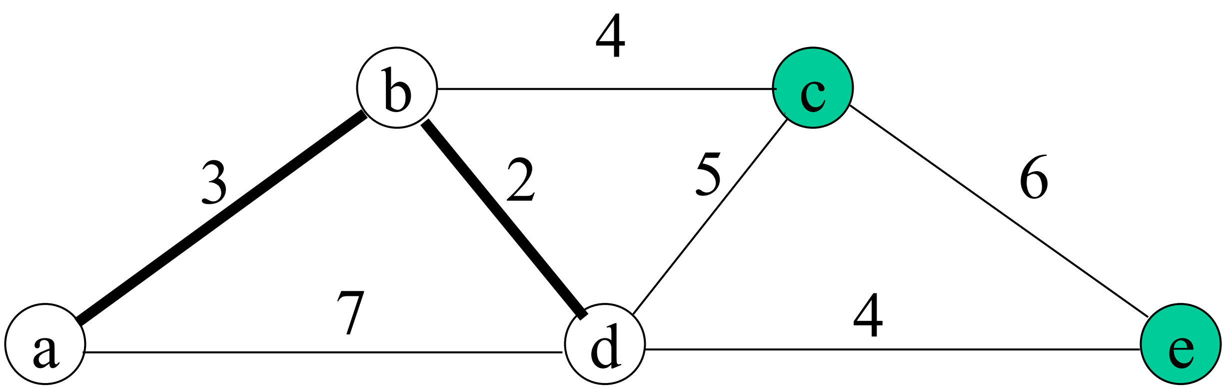
edge (*v,w*) with lowest *d*(*s,v*) + *d*(*v,w*)

- algorithm stops when all vertices are included

Step 3:

Tree vertices: a(-,0), b(a,3), d(b,5)

Priority queue: c(b,3+4), e (-,∞)→e(d,5+4)



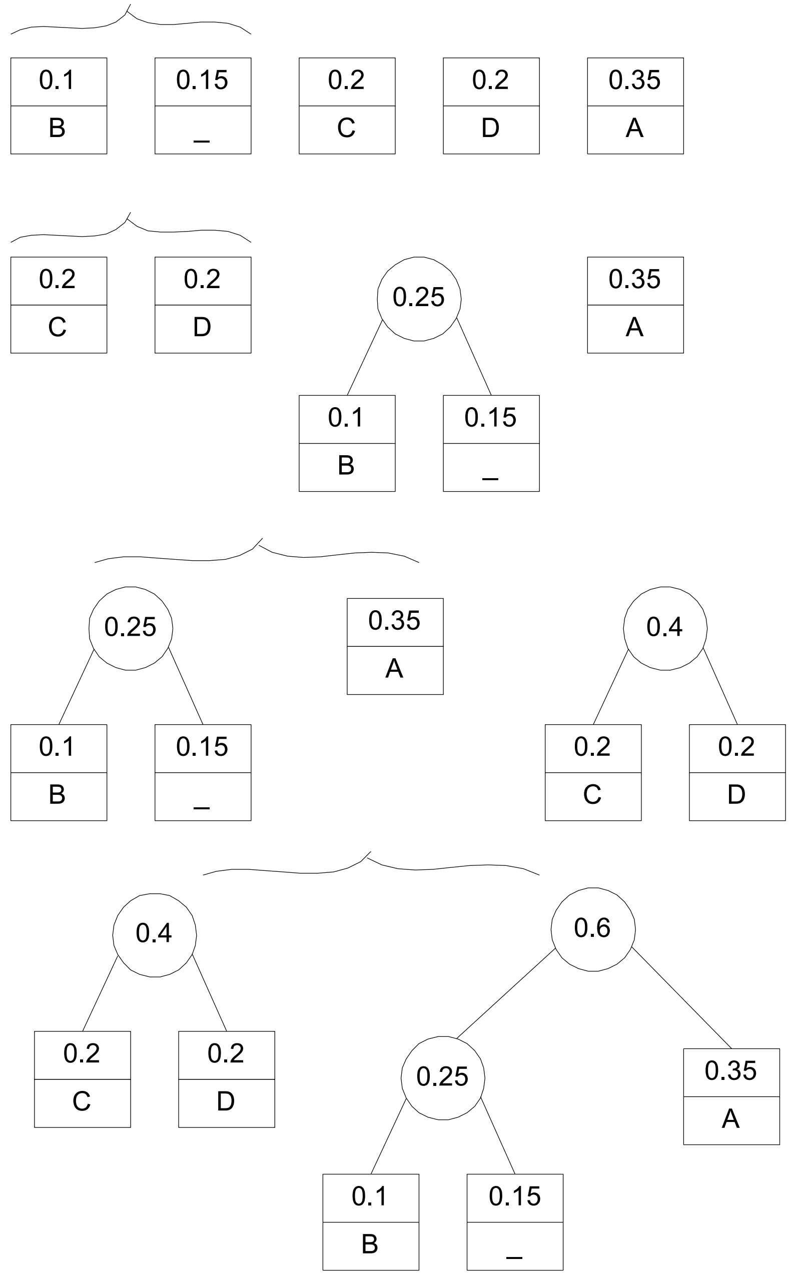
Huffman Coding Algorithm

Step 1: Initialize *n* one node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's weight

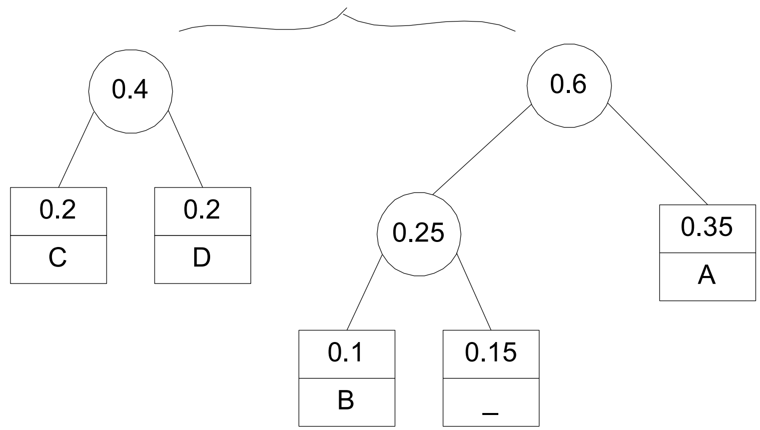
Step 2: Repeat the following operation until a single tree is obtained. Find two trees with the smallest weights. Make them the left and right subtrees of a new tree and record the sum of their weights in the root of the new tree as its weight

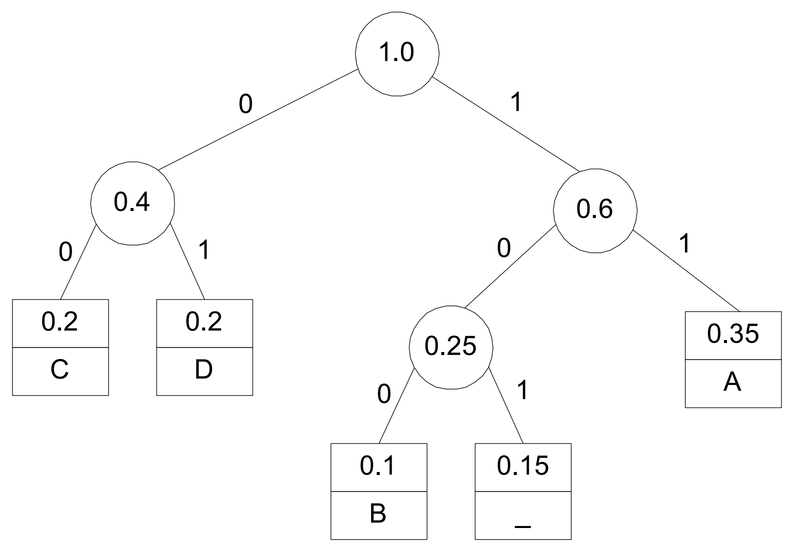
Example: alphabet {A, B, C, D, _} with frequency

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15



0.1
B
0.15
_
0.2
C
0.2
D
0.35
A
0.2
C
0.2
D
0.25
0.35
A
0.1
B
0.15
_
0.25
0.35
A
0.4
0.1
B
0.15
_
0.2
C
0.2
D
0.4
0.6
0.2
C
0.2
D
0.25
0.35
A
0.1
B
0.15
_


0.4
0.2
C
0.2
D
0.6
0.25
0.1
B
0.15
_
0.35
A

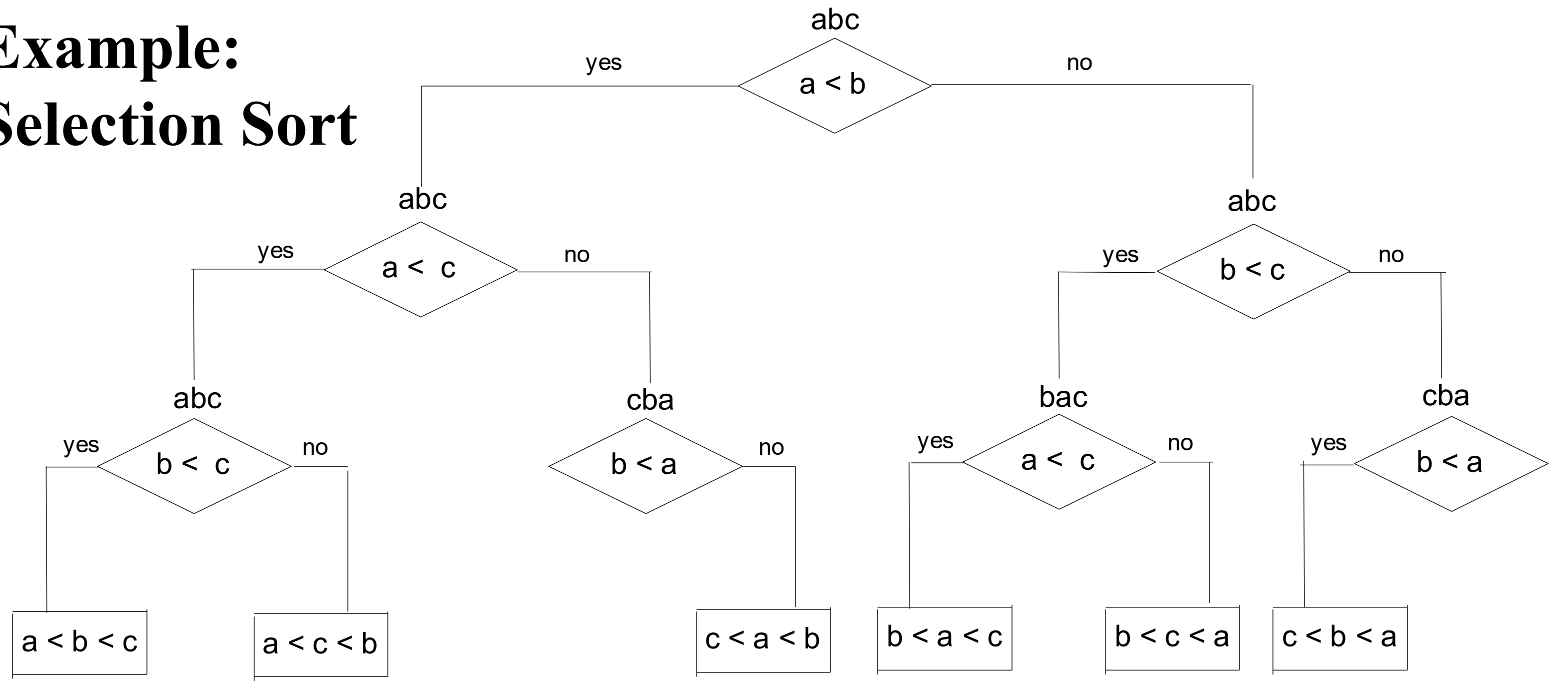

1.0
0
1
0.4
0
1
0.2
C
0.2
D
0.6
0
1
0.25
0
1
0.1
B
0.15
_
0.35
A

Limitations of Algorithm Power: Information-Theoretic Arguments

The number of leaves: *L*

Information-theoretic lower bound: worst case $\lceil \log_2 L \rceil$

Example: Selection Sort



P, *NP*, and *NP*-Complete Problems

As we discussed, problems that can be solved in polynomial time are usually called tractable and the problems that cannot be solved in polynomial time are called intractable, now

Is there a polynomial-time algorithm that solves the problem?

_P_: the class of decision problems that are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n

_NP_: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine

A decision problem *D* is _NP-complete (NPC)_ iff

1. $D \in NP$
2. every problem in *NP* is polynomial-time reducible to *D*

Design Strategies for NP-hard Problems

exhaustive search **(brute force)**

- useful only for small instances

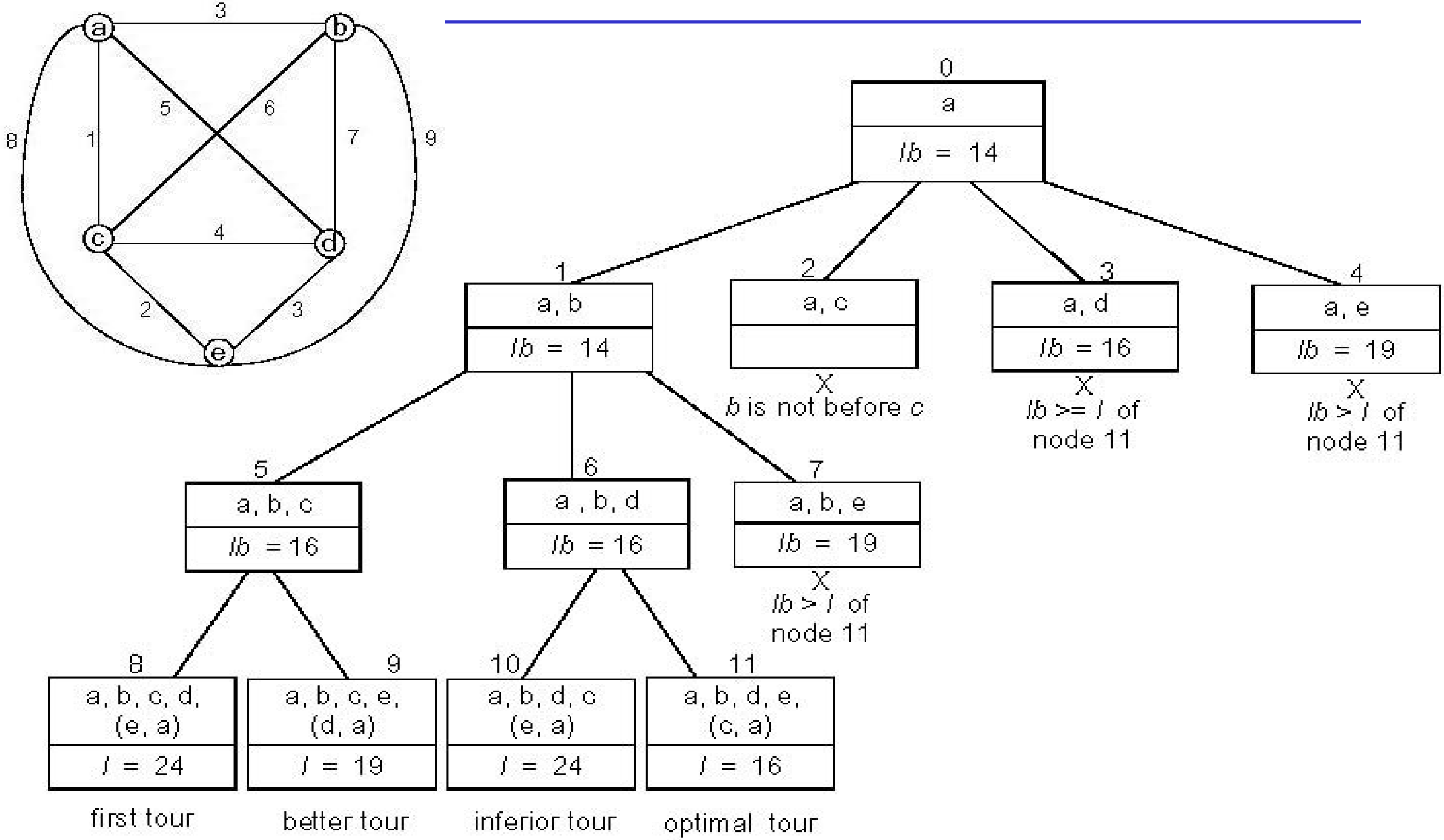
backtracking

- eliminates some cases from consideration

branch-and-bound

- An enhancement of backtracking.
- Applicable to optimization problems
- Uses a lower bound for the value of the objective function for each node (partial solution) so as to:
  - guide the search through state-space
  - rule out certain branches as “unpromising”

Traveling salesman example:



Thank you!

Good luck in your final exam!

Don’t forget to bring your cheat sheet!