Module 5

THE LIMITATIONS OF ALGORITHM POWER and COPYING: Lower-Bound Arguments, Decision Trees, P, NP, and NP Complete Problems.

BACKTRACKING & BRANCH-AND-BOUND: N-queens problem, sum of subset, assignment problem. Approximation Algorithms for NP-Hard Problems.

PRAM ALGORITHMS: Introduction, Computational Model, Parallel Algorithms for Prefix Computation.

COMPUTATIONAL MODELS

The most popular computational model is the **P**arallel **R**andom **A**ccess **M**achine (PRAM) which is a natural generalization of the **R**andom **A**ccess **M**achine (RAM).

The PRAM model consists of p synchronized processors $P1, P2, \ldots, Pp$, a shared memory with memory cells $M[1], M[2], \ldots, M[m]$ and memories of the processors.

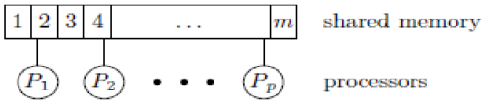


Figure 15.9 Parallel random access machine.

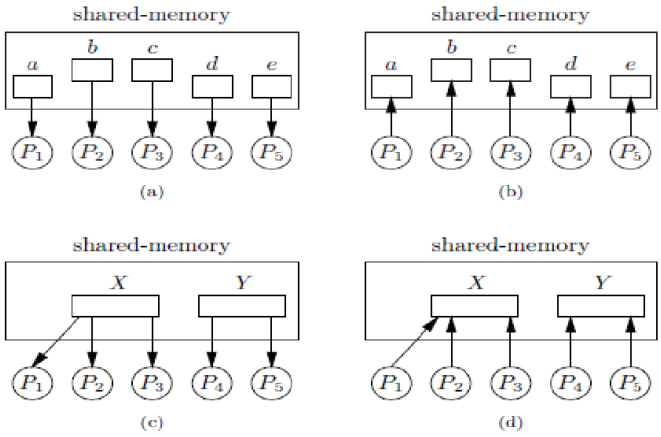


Figure 15.10 Types of parallel random access machines.

Figure 15.9. shows processors and the shared random access memory There are variants of this model. They differ in whether multiple processors are

allowed to access the same memory cell in a step, and in how the resulting conflicts are resolved. In particular the following variants are distinguished:
Types on the base of the properties of read/write operations are

1. **EREW (Exclusive-Read Exclusive-Write) PRAM,**
2. **ERCW (Exclusive-Read Concurrent-Write) PRAM,**
3. **CREW (Concurrent-Read Exclusive-Write) PRAM,**
4. **CRCW (Concurrent-Read Concurrent-Write) PRAM.**

Figure 15.10(a) shows the case when at most one processor has access a memory cell (ER), and Figure 15.10(d) shows, when multiple processors have access the same cell (CW).
Types of concurrent writing are *common, priority, arbitrary, combined.*

MESH, HYPERCUBE AND BUTTERFLY

Mesh is a popular computational model. A d-dimensional mesh is an $a1 \times a2 \times \cdots \times ad$ sized grid having a processor in each grid point. The edges are the communication lines, working in two directions. Processors are labelled by d-tuples, as $Pi1,i2,...,id$. Each processor is a RAM, having a local memory. The local memory of the processor $Pi1,i2,...,id$ is $M[i1, \ldots, id, 1], \ldots, M[i1, \ldots, id,m]$. Each processor can execute in one step such basic operations as adding, subtraction, multiplication, division, comparison, read and write from/into the local memory, etc. Processors work in synchronised way, according to a global clock.
The simplest mesh is the ***chain,*** belonging to the value $d = 1$. Figure 15.11 shows a chain consisting of 6 processors.

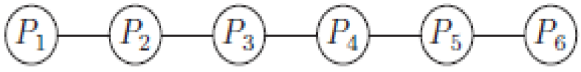


Figure 15.11 A chain consisting of six processors.

The processors of a chain are $P1, \ldots, Pp$. $P1$ is connected with $Pp-1$, Pp is connected with $Pp-1$, the remaining processors Pi are connected with $Pi-1$ and

Pi+1. If $d = 2$, then we get a rectangle. If now $a1 = a2 = pp$, then we get a ***square.*** Fig15.12 shows a square of size 4 × 4.

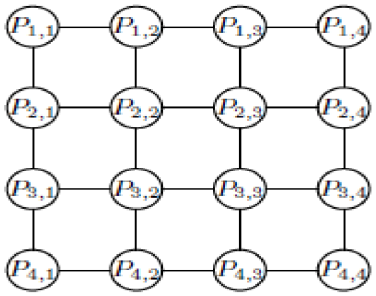


Figure 15.12 A square of size 4 × 4.

A square contains several chains consisting of a processors. The processors having identical first index, form a ***row of processors,*** and the processors having the same second index form a ***column of processors.*** Algorithms running on a square often consists of such operations, executed only by processors of some rows or columns.

If $d = 3$, then the corresponding mesh is a brick. In the special case $a1 = a2 = a3 = 3pp$ the mesh is called ***cube.***

Figure 15.13 shows a cube of size 2 × 2 × 2.

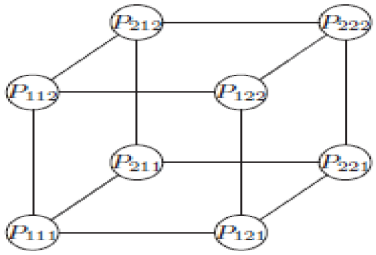


Figure 15.13 A 3-dimensional cube of size 2 × 2 × 2.

The next model of computation is the ***d-dimensional hypercube*** Hd. This model can be considered as the generalisation of the square and cube: the square represented on Figure 15.12 is a 2-dimensional, and the cube, represented on Figure 15.13 is a 3-dimensional hypercube. The processors of Hd can be labelled by a binary number consisting of d bits. Two processors of Hd are connected iff the Hamming-distance of their labels equals to 1.

Therefore each processors of Hd has d neighbours, and the of Hd is d. Figure 15.14 represents H4.

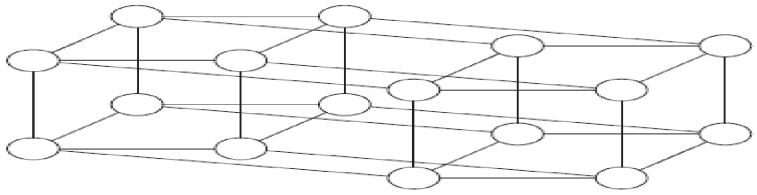


Figure 15.14 A 4-dimensional hypercube $\mathcal{H}_4$.

The butterfly model Bd consists of $p = (d + 1)2^d$ processors and $2dd+1$ edges. The processors can be labelled by a pair hr, li, where r is the ***columnindex*** and l is the ***level*** of the given processor. Figure 15.15 shows a ***butterfly*** model B3 containing 32 processors in 8 columns and in 4 levels. Finally Figure 15.16 shows a ***ring*** containing 6 processors.

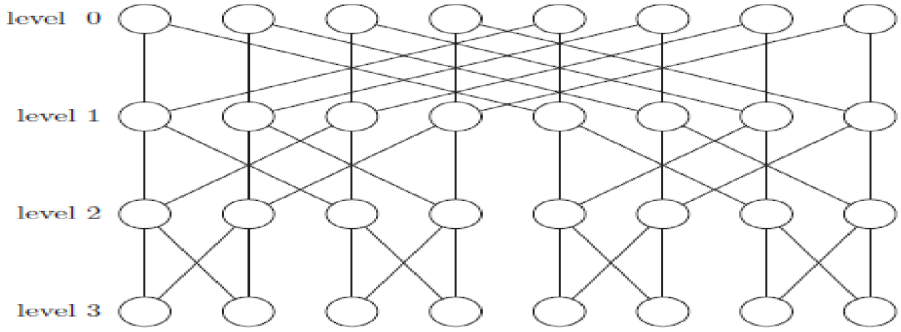


Figure 15.15 A butterfly model.

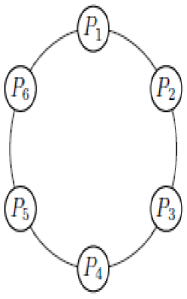


Figure 15.16 A ring consisting of 6 processors.

Backtracking

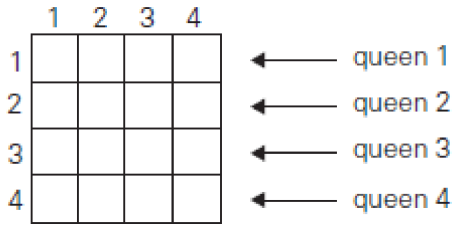
Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

A **state-space tree** is the tree of the construction of the solution from partial solution starting with the root with no component solution.

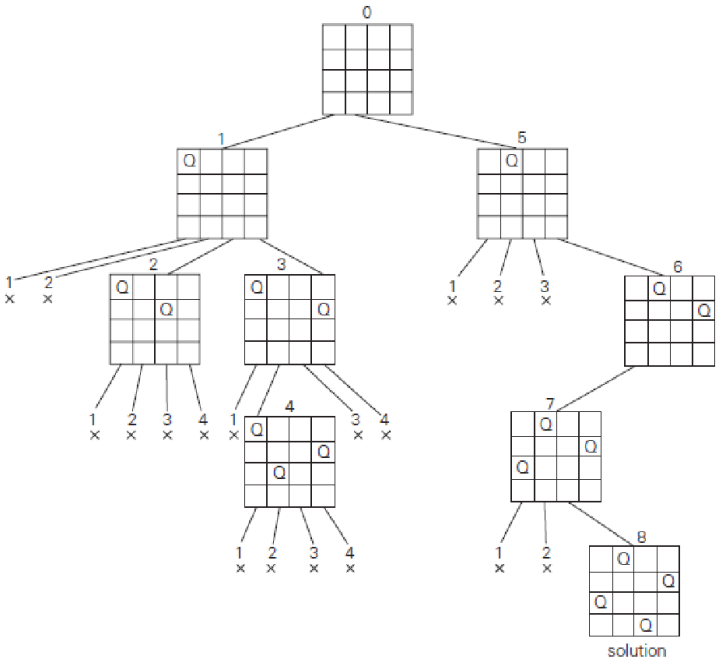
n-Queens Problem

Problem Statement: The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the four-queen's problem and solve it by the backtracking technique



State Space Tree for n-queens problem

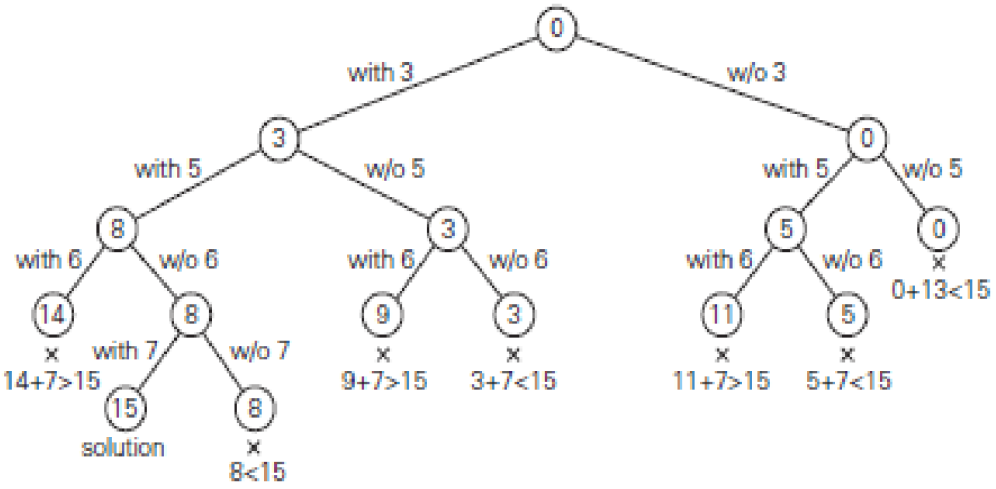


Subset-Sum Problem

Find a subset of a given set $A = \{a1, \ldots, an\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}.

It is convenient to sort the set's elements in increasing order. So, we will assume that The state-space tree can be constructed as a binary tree like that in Figure 12.4 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$.

State Space Tree for Sum of subsets



$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

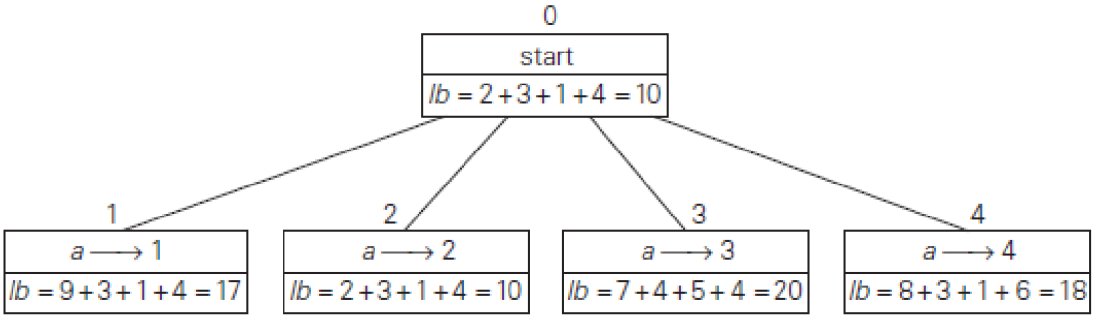
Branch and Bound Technique

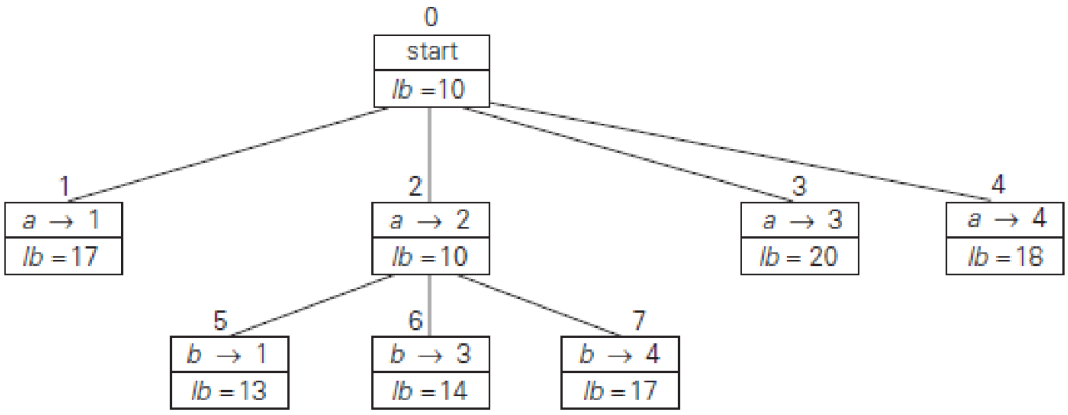
Branch and bound is a systematic method for solving optimization problems

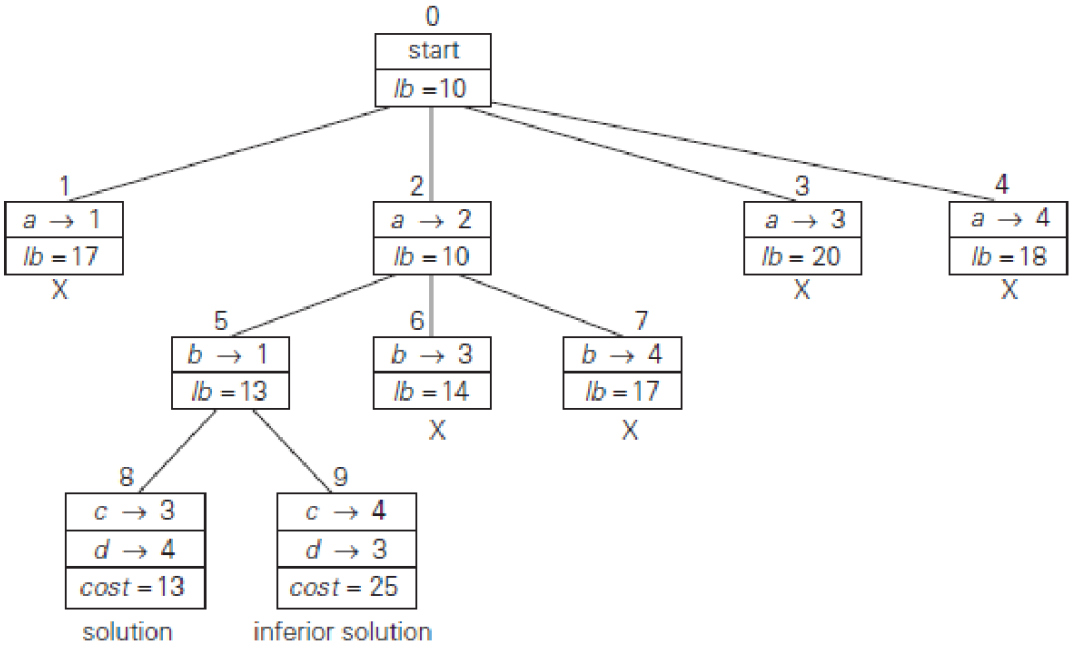
Assignment Problem

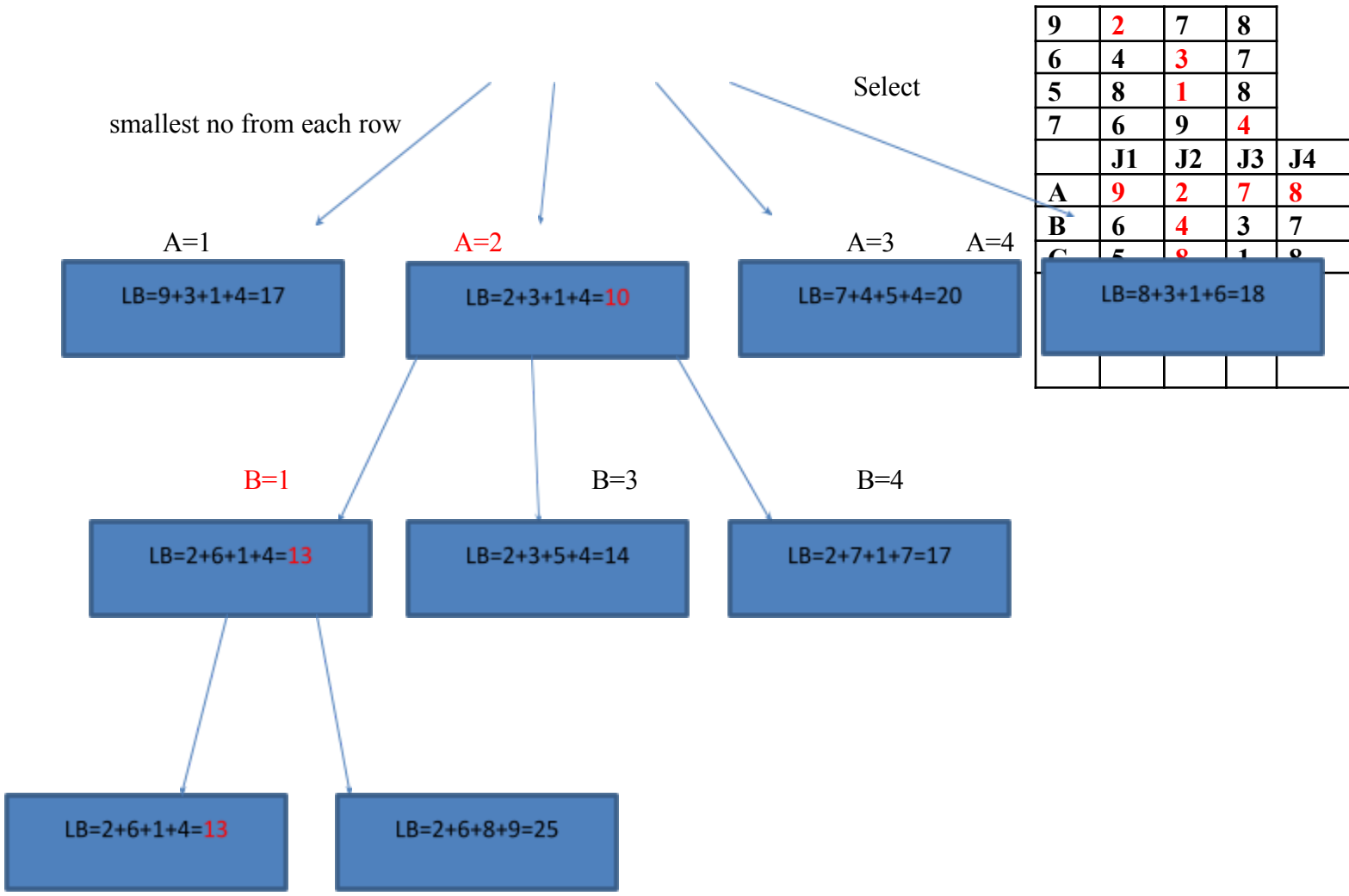
Assigning n people to n jobs so that the total cost of the assignment is as small as possible.

$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$









9	2	7	8	
6	4	3	7	
5	8	1	8	
7	6	9	4	
	J1	J2	J3	J4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8

C=3 C=4

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

LOWER-BOUND ARGUMENTS

We can look at the efficiency of an algorithm two ways. We can establish its **asymptotic efficiency class** (say, for the worst case) and see where this class stands with respect to the **hierarchy of efficiency classes**.

For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm, whereas the algorithm for the Tower of Hanoi problem is very slow because its efficiency is exponential.

Lower bounds means estimating the minimum amount of work needed to solve the problem. We present several methods for establishing lower bounds and illustrate them with specific examples.

1. Trivial Lower Bounds
2. Information-Theoretic Arguments
3. Adversary Arguments
4. Problem Reduction

In analyzing the efficiency of specific algorithms in the preceding, we should distinguish between a lower-bound class and a minimum number of times a particular operation needs to be executed.

Trivial Lower Bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's **input** that must be **processed** and the number of **output** items that need to be **produced**.

Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a **trivial lower bound**. For example, any algorithm for generating all permutations of n distinct items must be in $\Omega(n!)$ because the size of the output is n!. And this

bound is **tight** because good algorithms for generating permutations spend a constant time on each of them except the initial one.

Information-Theoretic Arguments

The information-theoretical approach seeks to establish a lower bound based on **the amount of information it has to produce** by algorithm.

Consider an example "**Game of guessing number**", the well-known game of deducing a positive integer between 1 and n selected by somebody by asking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by $[\log_2 n]$.

The number of bits needed to specify a particular number among the n possibilities. Each answer to the question gives information about each bit.

1. Is the first bit zero? → No→ first bit is 1
2. Is the second bit zero? → Yes→ second bit is 0
3. Is the third bit zero? → Yes→ third bit is 0
4. Is the forth bit zero? → Yes→ forth bit is 0

The number in binary is 1000, i.e.8 in decimal value.

The above approach is called the ***information-theoretic argument*** because of its connection to information theory. This is useful for finding ***information-theoretic lower bounds*** for many problems involving comparisons, including sorting and searching.

Adversary Arguments

Adversary Argument is a method of proving by **playing a role of adversary (opponent)** in which algorithm has to work more for **adjusting input** consistently. Consider the Game of guessing number between positive integer 1 and n by asking a person (Adversary) with yes/no type answers for questions. After each question at least one-half of the numbers reduced. If an algorithm stops before the size of the set is reduced to 1, the adversary can exhibit a number.

Any algorithm needs $[\log_2 n]$ iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set. Hence, at least $[\log_2 n]$ questions need to be asked by any algorithm in the worst case. This example illustrates the ***adversary method*** for establishing lower bounds.

Problem Reduction

Problem reduction is a method in which a difficult unsolvable problem P is reduced to another solvable problem B which can be solved by a known algorithm. A similar reduction idea can be used for finding a lower bound. To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q!). In other words, we should show that an arbitrary instance of problem Q can be transformed to an instance of problem P, so any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P.

DECISION TREES

Important algorithms like sorting and searching are based on comparing items of their inputs. The study of the performance of such algorithm is called a **decision tree**. As an example, Figure

5.1 presents a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node.

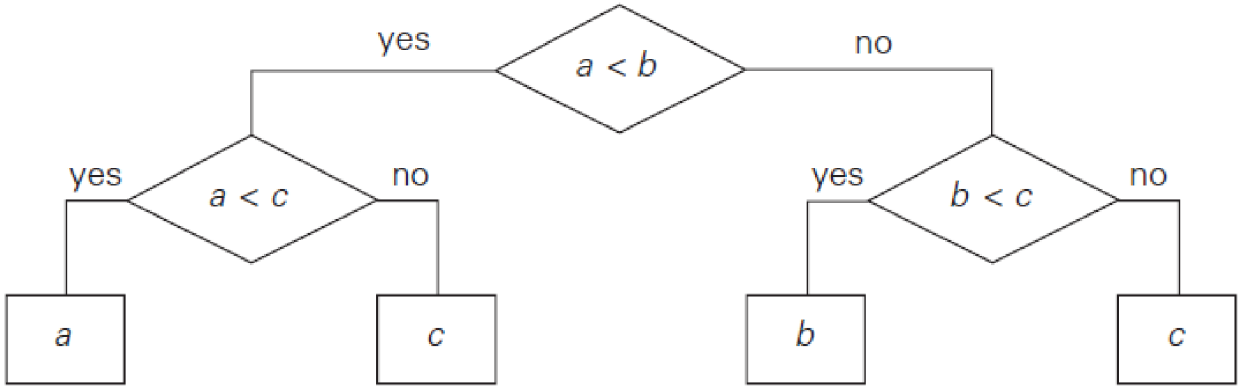


FIGURE 5.1 Decision tree for finding a minimum of three numbers.

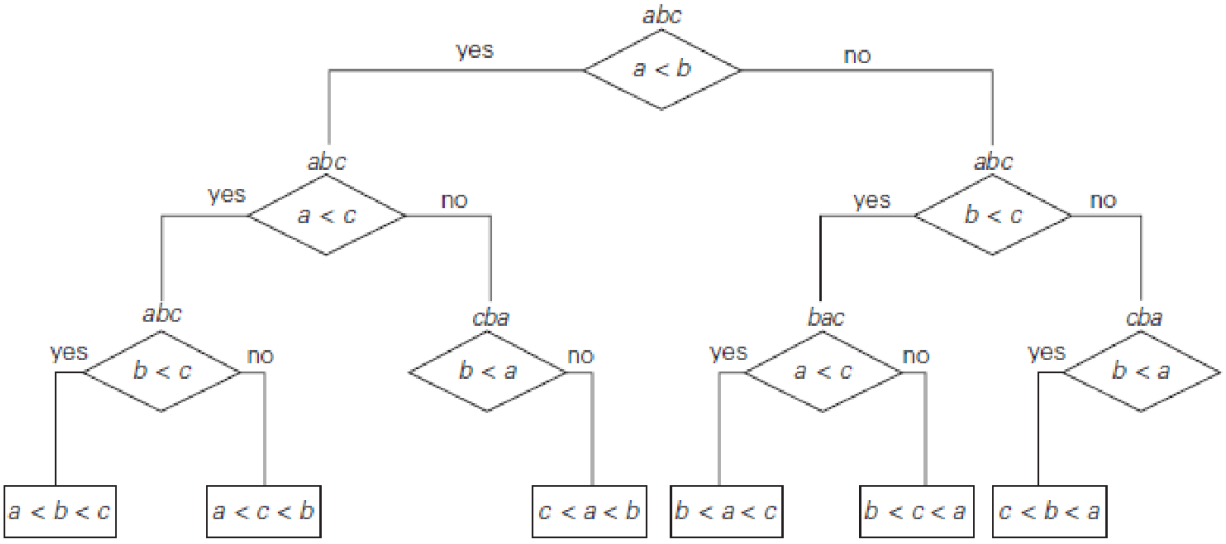


FIGURE 5.2 Decision tree for the tree-element selection sort

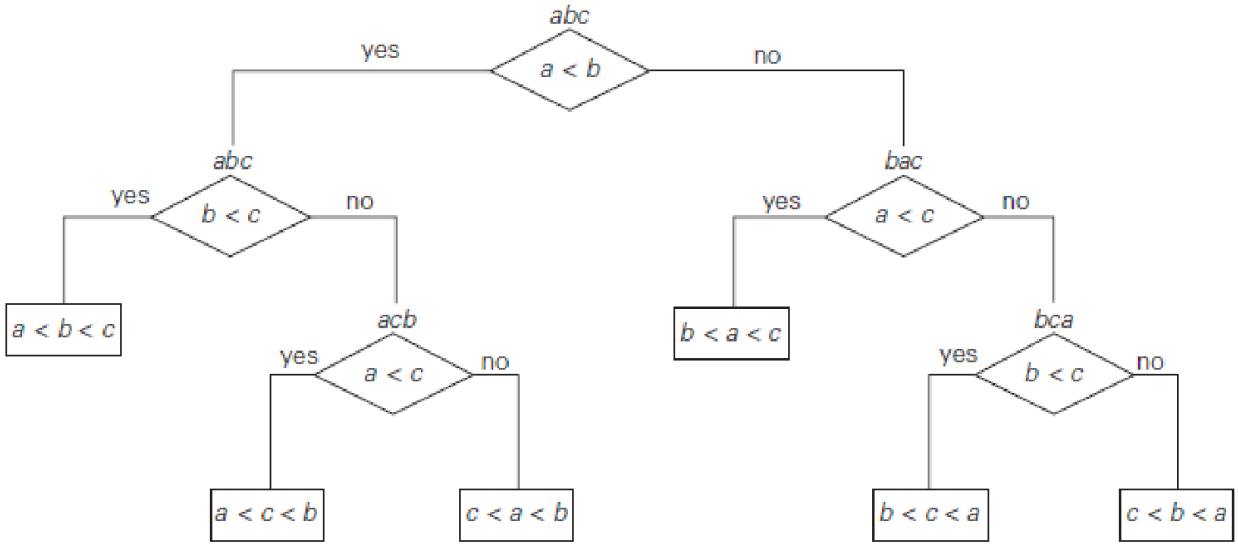


FIGURE 5.3 Decision tree for the three-element insertion sort.

P, NP AND NP-COMPLETE PROBLEMS

Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

Class *P* is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial class***.

Examples:

- Searching
- Element uniqueness
- Graph connectivity
- Graph acyclicity
- Primality testing

Non polynomial-time algorithm: There are many important problems, however, for which no polynomial-time algorithm has been found.

***Hamiltonian circuit problem*:** Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

- ***Traveling salesman problem*:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

***Knapsack problem*:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

- ***Partition problem*:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- ***Bin-packing problem*:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.
- ***Graph-coloring problem*:** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.
- ***Integer linear programming problem*:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

1. **Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the given instance.
2. **Deterministic ("verification") stage:** A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to instance I , the algorithm either returns no or is allowed not to halt at all.)

Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.