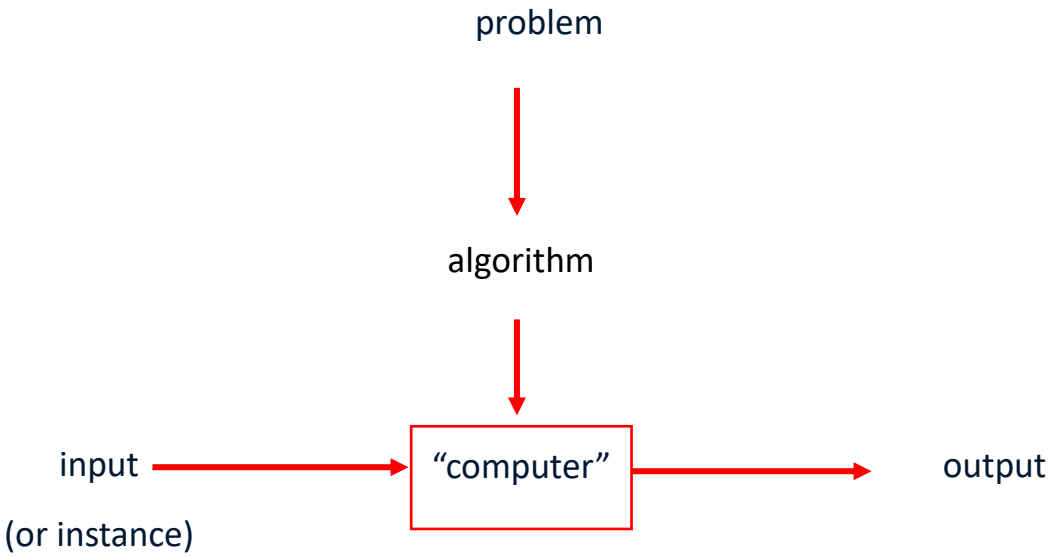
Unit 1

Introduction

- 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.
- Non ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm, works has to be specified carefully
- The same algorithm can be represented in several different ways
- There may exist several algorithms for solving same problem
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds
-



Notion of algorithm and problem

Requirements of an Algorithm

1. **Finiteness:** terminates after a finite number of steps
2. **Definiteness:** rigorously and unambiguously specified
3. **Clearly specified input:** valid inputs are clearly specified
4. **Clearly specified/expected output:** can be proved to produce the correct output given a valid input
5. **Effectiveness: steps** are sufficiently simple and basic

Why Study Algorithms?

- **Theoretical importance**
  - the core of computer science
- **Practical importance**
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Problem – To find GCD of two numbers

- Three Algorithms
  - Euclid's Algorithm
  - Consecutive integer checking
  - Middle school Procedure

Euclid's algorithm for computing gcd (m,n)

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Pseudocode:

```
ALGORITHM Euclid(m,n)
while n ≠ 0 do
  r ← m mod n
  m← n
  n ← r
return m
```

Other methods for computing gcd(m,n)

Consecutive integer checking algorithm

Step 1 Assign the value of min{ m,n} to t

Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

Other methods for gcd(m,n) [cont.]

Middle-school procedure

Step 1 Find the prime factorization of m

Step 2 Find the prime factorization of n

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(m,n)

Thus, for the numbers 60 and 24, we get

$60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$ $\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$

Both Consecutive Integer Checking and Middle School algorithm does not work properly if any one of the input is zero, but Euclid's algorithm works in the above case also.

Sieve of Eratosthenes

- Sieve of Eratosthenes is an algorithm for generating consecutive primes not exceeding any given integer n>1.
- The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n.
- Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on.
- Then it moves to the next item on the list, which is 3, and eliminates its multiples.
- No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass.
- The next remaining number on the list, which is used on the third pass, is 5.
- The algorithm continues in this fashion until no more numbers can be eliminated from the list.
- The remaining integers of the list are the primes needed.

Sieve of Eratosthenes

Algorithm Sieve(n)
//Input: Integer $n > 1$
//Output: Array L: List of primes less than or equal to n
for $p \leftarrow 2$ to n do
  $A[p] \leftarrow p$
for $p \leftarrow 2$ to $\sqrt{n}$ do
  if $A[p] \neq 0$ // p hasn't been previously eliminated from the list
    $j \leftarrow p * p$
    while $j \leq n$ do
      $A[j] \leftarrow 0$ //mark element as eliminated
      $j \leftarrow j + p$

```
//copy remaining elements of A to array L of the primes
i←0
for p→2 to n do
    if A[p]= 0
        L[i] ← A[p]
        i=i+1
return L
```

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

For this example, no more passes are needed because they would eliminate num-bers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

Fundamentals of Algorithmic problem solving

- Understanding the problem
- Ascertain the capabilities of the computational device
- Exact /approximate soln.
- Decide on the appropriate data structure
- Algorithm design techniques
- Methods of specifying an algorithm
- Proving an algorithms correctness
- Analysing an algorithm

- **Understanding the problem**: The problem given should be understood completely. Check if it is like some standard problems & if a known algorithm exists. Otherwise, a new algorithm must be devised. Creating an algorithm is an art which may never be fully automated. An important step in the design is to specify an in- stance of the problem.

- **Ascertain the capabilities of the computational device**: Once a problem is understood we need to know the capabilities of the computing device this can be done by knowing the type of the architecture, speed & memory availability.

- **Exact /approximate soln.:** Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. The solution is stated in two forms, Exact solution, or approximate solution. Examples of problems where an exact solution cannot be obtained are i) Finding a square root of number. ii)Solutions of nonlinear equations.

- **Decide on the appropriate data structure**: Some algorithms do not demand any in- genuity in representing their inputs. Some others are in fact are predicted on ingenious data structures. A data *type* is a well-defined collection of data with a well-defined set of operations on it. A data *structure* is an actual implementation of a particular abstract data type. The Elementary Data Structures are Arrays. These let you access lots of data fast. (good. Records let you organize non-homogeneous data into logical packages to keep every- thing together. These packages do not include operations, just data fields. Records do not help you process distinct items in loops (bad, which is why arrays of records are used). Sets let you represent subsets of a set with such operations as intersection, union, and equivalence.

- **Algorithm design techniques**: Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operation research and electrical engineering. Some important design techniques are linear, nonlinear and integer programming

- **Methods of specifying an algorithm**: There are mainly two options for specifying an algorithm: use of natural language or pseudocode & Flowcharts. A Pseudo code is a mixture of natural language & programming language like constructs. A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes.

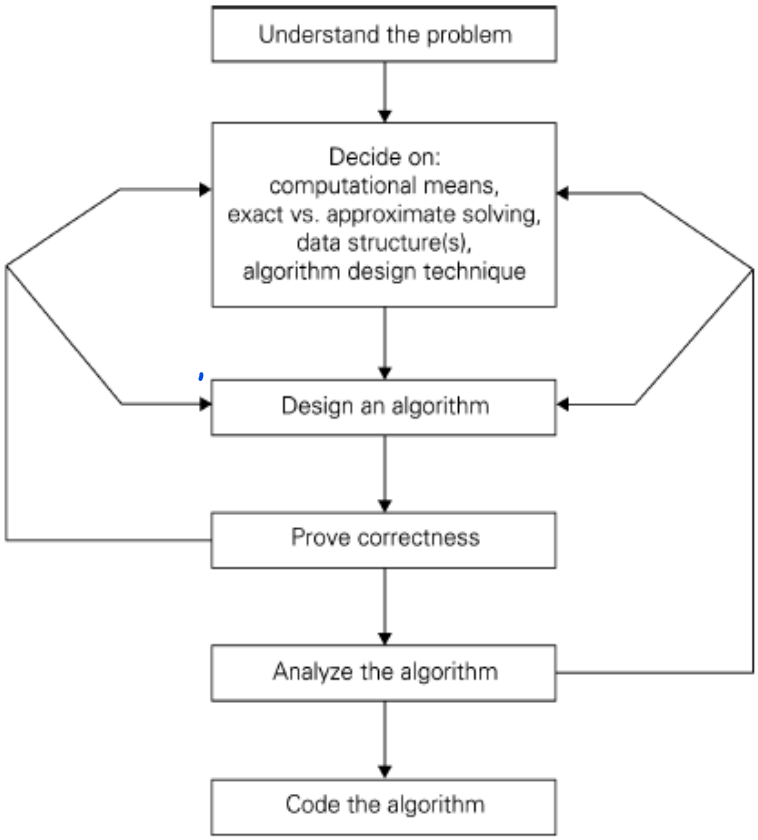
- **Proving an algorithms correctness**: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. We refer to this process as algorithm validation. The process of validation is to assure us that this algorithm will work correctly independent of issues concerning programming language it will be written in. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of a program. These assertions are often expressed in the predicate calculus. The second form is called a specification, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe same out- put. A complete proof of program correctness requires that each statement of programming language be precisely

defined, and all basic operations be proved correct. All these details may cause proof to be very much longer than the program.

- **Analysing algorithms**: As an algorithm is executed, it uses the computers central processing unit to perform operation and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms and performance analysis refers to the task of determining how much computing time and storage an algorithm requires. There are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. Yet another desirable characteristic of an algorithm is ***generality***. There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts.

- **Coding an Algorithm**

  Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a pro-gram either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. They have developed special techniques for doing such proofs but the power of these techniques of formal verification is limited so far to exceedingly small programs. As a practical matter, the validity of programs is still established by testing.



Algorithm Design and Analysis Process

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Sorting

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$
  - Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?

    - Help searching
    - Algorithms often use sorting as a key subroutine.
- Sorting key
    - A specially chosen piece of information used to guide sorting. E.g., sort student records by names.
- Examples of sorting algorithms
    - Selection sort
    - Bubble sort
    - Insertion sort
    - Merge sort
    - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
    - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
    - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Searching

- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
    - Sequential search
    - Binary search

String Processing

- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.

Examples:

(i) searching for a word or phrase on WWW or in a Word document

(ii) searching for a short read in the reference genomic sequence

Graph Problems

- Informal definition
    - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Modeling real-life problems
    - Modeling WWW
    - Communication networks
    - Project scheduling …
- Examples of graph algorithms
    - Graph traversal algorithms
    - Shortest-path algorithms
    - Topological sorting

Combinatorial Problems

- To find combinatorial object such as permutation, combination or a subset that satisfies certain constraints.
- Examples
    - Travelling Salesman
    - Graph coloring problem

Geometric Problems

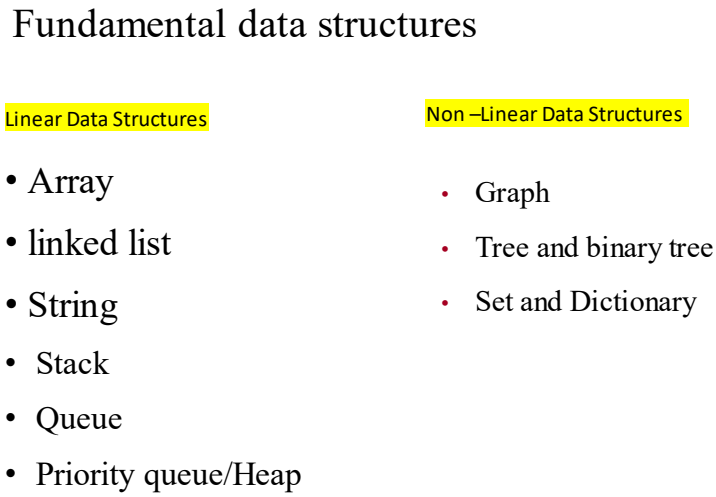
- Deals with geometric objects like points, line and polygons.
- Examples:
    - Closest pair problem
    - Convex hull problem

Numerical Problems

- Involve mathematical objects of continuous nature
- Solving equations, system of equations, computing definite integrals, evaluating functions

Fundamental Data Structures

- A **data structure** is a particular scheme of organizing related data items.



Linear Data Structures

- **Arrays**
  - A sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete operations performed on them

- **Linked List**
  - A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
  - Singly linked list (next pointer)
  - Doubly linked list (next + previous pointers)
  - dynamic length

- arbitrary memory locations
- access by following links
- Insert/delete operations performed on them

FIGURE 1.3 Array of n elements.

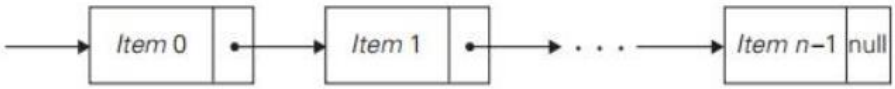


FIGURE 1.4 Singly linked list of n elements.

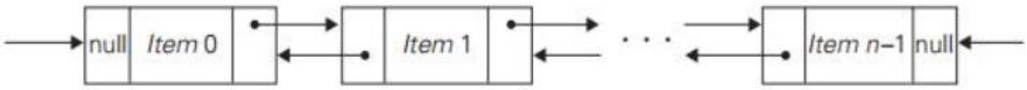


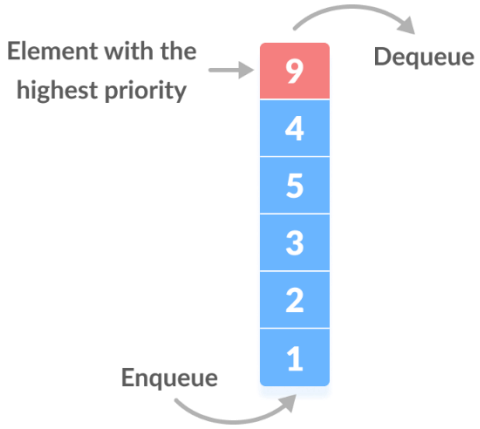
FIGURE 1.5 Doubly linked list of n elements.

- **Strings: sequence of characters**
  - Implemented using arrays
  - Binary or bit strings- sequence of 0's and 1's
  - Operations: concatenation, comparison, computing length of strings etc.

- **Stacks and Queues**
  - Stacks
    - A stack of plates
      - insertion/deletion can be done only at the top.
      - LIFO
    - Two operations (push and pop)
  - Queues
    - A queue of customers waiting for services
      - Insertion/enqueue from the rear and deletion/dequeue from the front.
      - FIFO

- Two operations (enqueue and dequeue)

- A **priority queue** is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.



Non -Linear Data Structures

- **Graphs**
  - A graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs."
  - Formally, a ***graph G = V, E*** is defined by a pair of two sets: a finite nonempty set ***V*** of items called ***vertices*** and a set ***E*** of pairs of these items called ***edges***.
  - If these pairs of vertices are unordered, i.e., a pair of vertices ***(u, v)*** is the same as the pair ***(v, u),*** we say that the vertices ***u*** and ***v*** are ***adjacent*** to each other and that they are connected by the ***undirected edge (u, v).***
  - A graph ***G*** is called ***undirected*** if every edge in it is undirected.
  - If a pair of vertices ***(u, v)*** is not the same as the pair ***(v, u),*** we say that the edge ***(u, v)*** is ***directed*** from the vertex ***u,*** called the edge's ***tail***, to the vertex ***v,*** called the edge's ***head***.
  - A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***.

- It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6).

$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$.

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$.

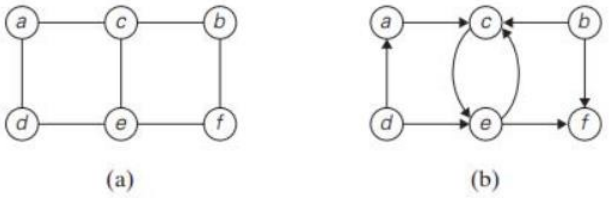
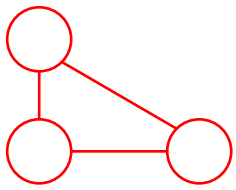


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Graph Properties

- Paths
  - A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.
  - Simple paths: All edges of a path are distinct.
  - Path lengths: the number of edges, or the number of vertices – 1.
- Connected graphs
  - A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.
- Connected component
  - The maximum connected subgraph of a given graph.



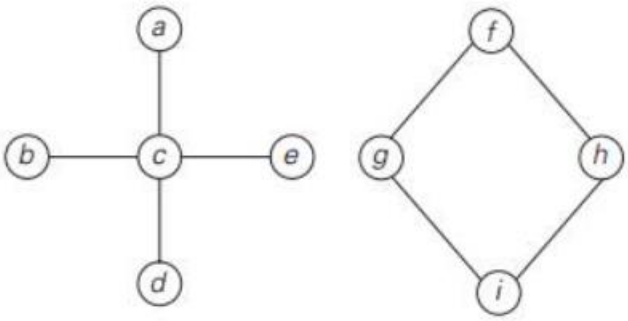
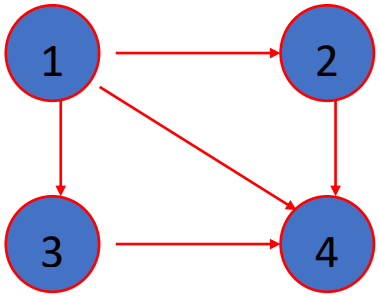


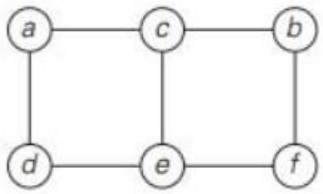
FIGURE 1.9 Graph that is not connected.

- Cycle
  - A simple path of a positive length that starts and ends the same vertex.
- Acyclic graph
  - A graph without cycles
  - DAG (Directed Acyclic Graph)



Graph Representation

- Adjacency matrix
  - n x n boolean matrix if |V| is n.
  - The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.
  - The adjacency matrix of an undirected graph is symmetric.
- Adjacency linked lists
  - A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.



(a)

FIGURE 1.6 (a) Undirected graph

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a	→	c	→	d			
b	→	c	→	f			
c	→	a	→	b	→	e	
d	→	a	→	e			
e	→	c	→	d	→	f	
f	→	b	→	e			

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.

Weighted Graphs

- A ***weighted graph*** (or weighted digraph) is a graph (or di-graph) with numbers assigned to its edges.
- These numbers are called ***weights*** or ***costs***.
- If a weighted graph is represented by its adjacency matrix, then its element ***A*[*i, j*] will simply contain the weight of the edge from the ***i***th to the ***j*** th vertex if there is such an edge and a special symbol, e.g., ∞**,** if there is no such edge. Such a matrix is called the ***weight matrix*** or ***cost matrix***.
- Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge (Figure 1.8c).

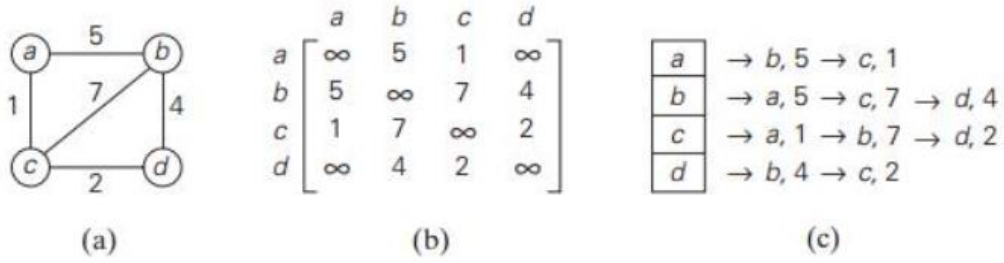
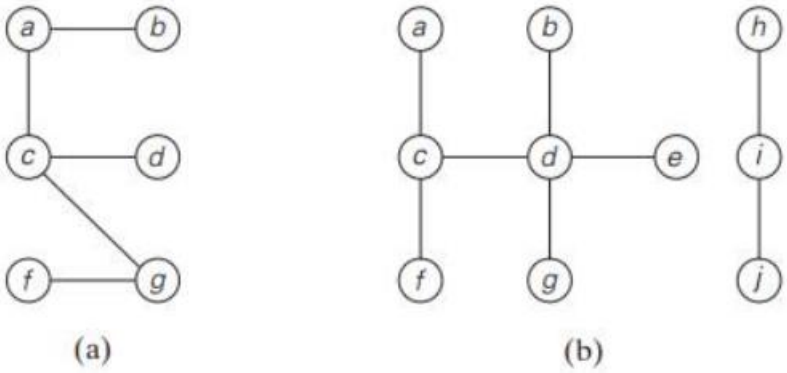


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Trees

- A tree (or free tree) is a connected acyclic graph.
- Forest: a graph that has no cycles but is not necessarily connected.



(a) Tree. (b) Forest.

- Important property of a tree

Number of edges in a tree: |E| = |V| - 1

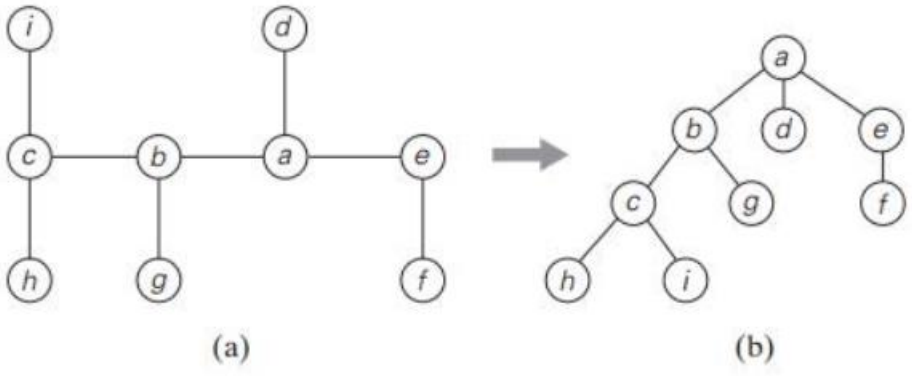
Rooted Trees

- Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the ***root*** of the so-called ***rooted tree***.
- A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

- Ancestors

- For any vertex *v* in a tree *T*, all the vertices on the simple path from the root to that vertex are called ancestors.

- Descendants
  - All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.
- Parent, child and siblings
  - If *(u, v)* is the last edge of the simple path from the root to vertex *v*, *u* is said to be the parent of *v* and *v* is called a child of *u*.
  - Vertices that have the same parent are called siblings.
- Leaves
  - A vertex without children is called a leaf.
- Subtree
  - A vertex *v* with all its descendants is called the subtree of *T* rooted at *v*.



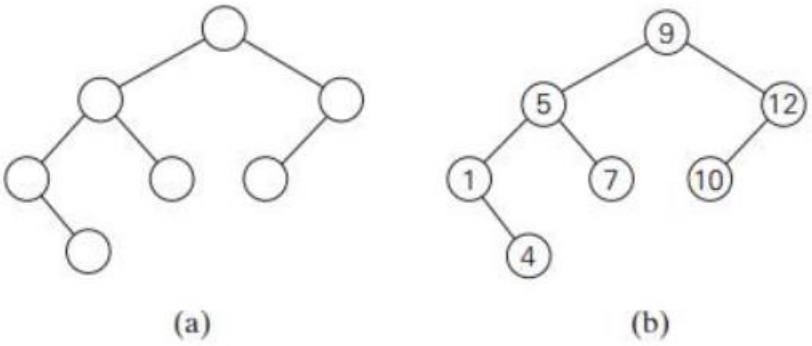
(a) Free tree. (b) Its transformation into a rooted tree.

- Depth of a vertex
  - The length of the simple path from the root to the vertex.
- Height of a tree
  - The length of the longest simple path from the root to a leaf.

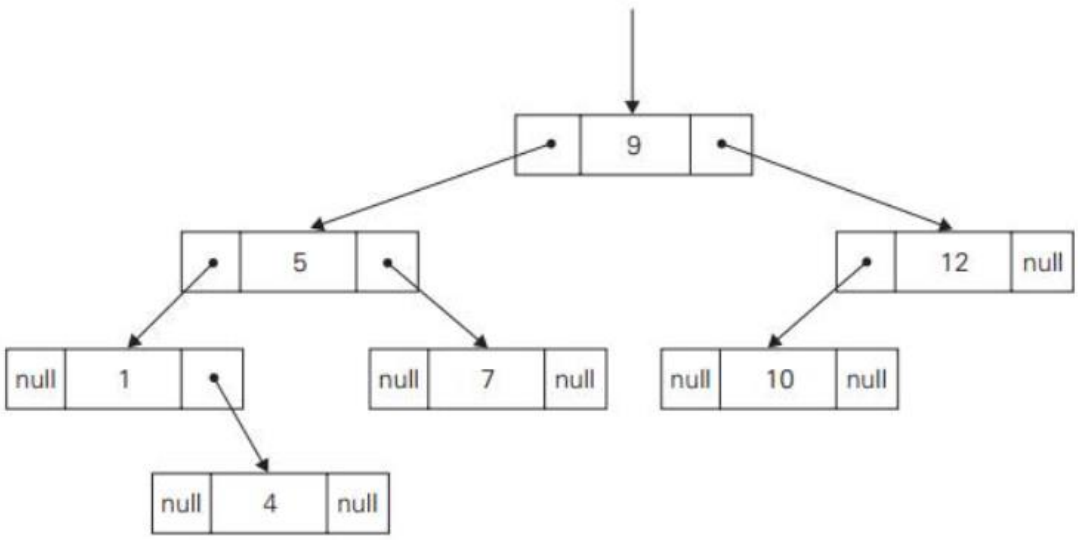
Ordered Trees

- Ordered trees

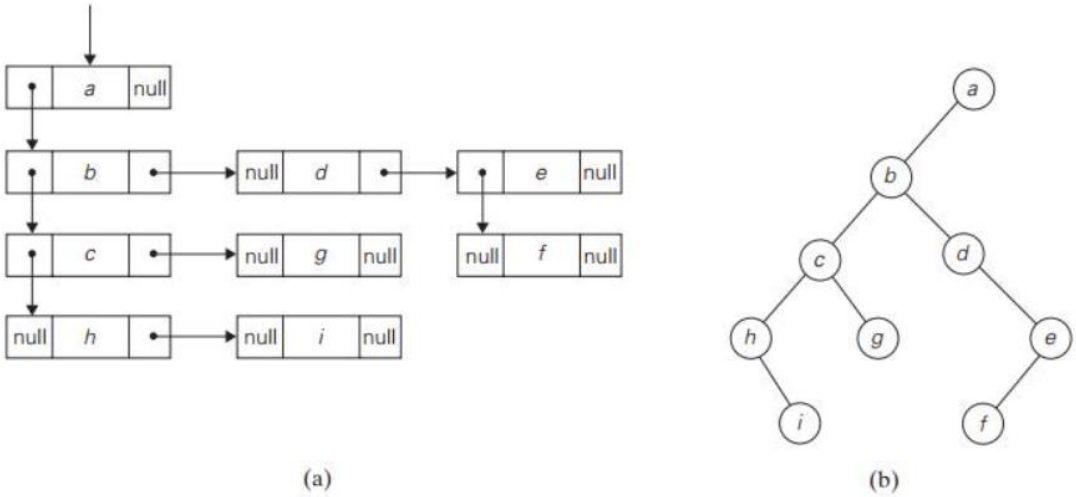
    - An ordered tree is a rooted tree in which all the children of each vertex are ordered.
- Binary trees
    - A binary tree is an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent.
- Binary search trees
    - Each vertex is assigned a number.
    - A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.
- $\lfloor \log_2 n \rfloor \leq h \leq n - 1$, where h is the height of a binary tree and n the size.



(a) Binary tree. (b) Binary search tree.



Standard implementation of the binary search tree in Figure 1.12b.



(a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.

The above representation is first child–next sibling representation. Thus, all the siblings of a vertex are linked via the nodes' right pointers in a singly linked list, with the first element of the list pointed to by the left pointer of their parent.

Sets and Dictionaries

- **Set**
  - A ***set*** can be described as an unordered collection (possibly empty) of distinct items called ***elements*** of the set.
  - ***multiset***, or ***bag***, is an unordered collection of items that are not necessarily distinct.
  - A specific set is defined either by an explicit listing of its elements (e.g., S = {2, 3, 5, 7}) or by specifying a property that all the set's elements and only they must satisfy (e.g., S = {n: n is a prime number smaller than 10}).
  - The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.
  - Implementation of sets in computer applications:
    - If set Universal set U has n elements, then any subset S of U can be represented by a bit string of size n, called a bit vector, in which the ith element is 1 if and only if the ith element of U is included in set S. Thus, to continue with our example, if U = {1, 2, 3, 4, 5, 6, 7, 8, 9}, then S = {2, 3, 5, 7} is represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast, but at the expense of potentially using a large amount of storage.
    - The second and more common way to represent a set for computing purposes is to use the list structure to indicate the set's elements. Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need.
    - Difference between sets and list.

- First, a set cannot contain identical elements; a list can.
- Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite.

- In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection.
- A data structure that implements these three operations is called the dictionary.

Fundamentals of the Analysis of Algorithm Efficiency

The Analysis Framework

1. Measuring an Input's Size

2. Units for Measuring Running Time

3. Orders of Growth

4. Worst-Case, Best-Case, and Average-Case Efficiencies

There are two kinds of efficiency: time efficiency and space efficiency. ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs. ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Measuring an Input's Size

Almost all algorithms run longer on larger inputs. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

- For problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists it will be the size of the list.
- For the problem of evaluating a polynomial $p(x) = a_n x^n + \cdots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- For computing the product of two $n \times n$ matrices two measures are needed like order of matrix 'n' and total number of elements 'N' in the matrices multiplied.
- For a spell-checking problem, size metric may be influenced by the operations. That is if the algorithm examines individual characters of its input, the measure the size by the number of characters; if it works by processing words, we should count their number in the input.

- For problems such as checking primality of a positive integer ***n***, the input is just one number, and it is this number's magnitude that determines the input size. In such situations, it is preferable to measure size by the number ***b*** of bits in the ***n***'s binary representation:

$$b = \lfloor \log_2 n \rfloor + 1. \qquad \textbf{(2.1)}$$

Units for Measuring Running Time

- To measure the running time , we can measure the execution time of a program implementing the algorithm. But this approach depends on the speed of a particular computer, the quality of a program implementing the algorithm and type of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the program. Since we are after a measure of an algorithm's efficiency, we would like to have a metric that does not depend on these extraneous factors.
- One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- The basic operation of an algorithm is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.
- Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n.

Let cop be the execution time of an algorithm's basic operation on a particular computer, and let C(n) be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time T (n) of a program implementing this algorithm on that computer by the formula

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

Orders of Growth

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. For large values of ***n***, it is the function's order of growth that counts.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes. Also note that although specific values of such a count depend, of course, on the logarithm's base, the formula

$$\log_a n = \log_a b \log_b n$$

makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant. Therefore we omit a logarithm's base and write simply $\log n$ in situations where we are interested just in a function's order of growth to within a multiplicative constant.

On the other end of the spectrum are the exponential function 2^n and the factorial function $n!$. Both these functions grow so fast that their values become astronomically large even for rather small values of ***n.*** Hence, algorithms that require an exponential number of operations are practical for solving only problems of small sizes.

Worst-Case, Best-Case, and Average-Case Efficiencies

For a given input size, how does algorithm perform on different datasets of that size

For datasets of size **n** identify different datasets that give:

Worst case: $C_{worst}(n)$ – maximum over all inputs of size n

Best case: $C_{best}(n)$ – minimum over all inputs of size n

Average case: $C_{avg}(n)$ – "average" over inputs of size n

Example:

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Worst case: In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: C_{worst} (n) = n.

Best-case: The best-case inputs for sequential search are lists of size **n** with their first element equal to a search key; accordingly, C_{best} (n) = 1 for this algorithm.

Average Case:

Let's consider again sequential search. The standard assumptions are that

(a) the probability of a successful search is equal to p ($0 \le p \le 1$)

(b) the probability of the first match occurring in the ith position of the list is the same for every i.

In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being (1 − p). Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

Asymptotic Notations and Basic Efficiency Classes

1. ***O* notation**
2. **Ω notation**
3. **Θ notation**

O-notation

Definition: A function ***t (n)*** is said to be in ***O(g(n))***, denoted ***t (n)*** ∈ ***O(g(n)),*** if ***t (n)*** is bounded above by some constant multiple of ***g(n)*** for all large ***n,*** i.e., if there exist some positive constant ***c*** and some nonnegative integer n_0 such that

$t(n) \leq cg(n)$ for all $n \geq n_0$.

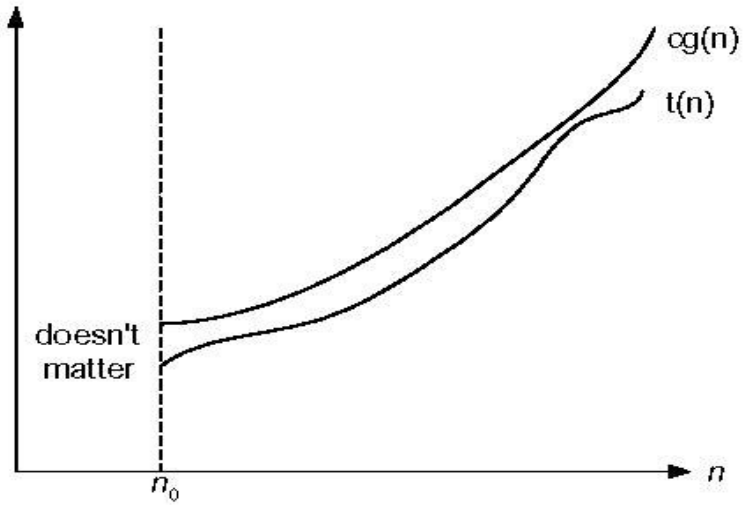


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Ω-notation

Definition: A function ***t (n)*** is said to be in ***(g(n)),*** denoted ***t (n)*** ∈ ***(g(n)),*** if ***t (n)*** is bounded below by some positive constant multiple of ***g(n)*** for all large ***n,*** i.e., if there exist some positive constant ***c*** and some nonnegative integer n_0 such that

$t(n) \geq cg(n)$ for all $n \geq n_0$.

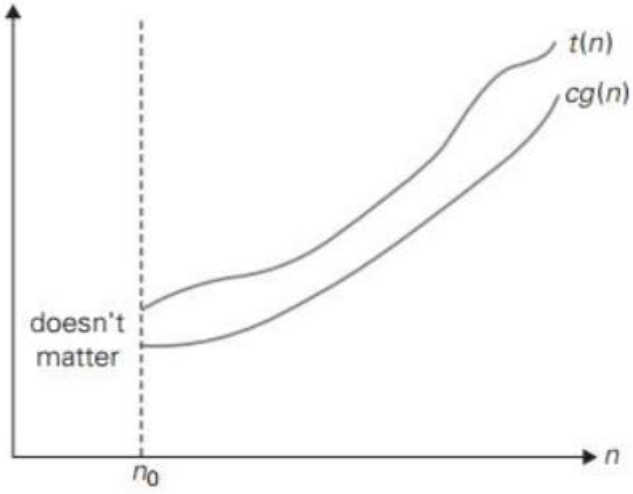


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Θ notation

Definition: A function ***t (n)*** is said to be in ***(g(n))***, denoted ***t (n)*** ∈ ***(g(n)),*** if ***t (n)*** is bounded both above and below by some positive constant multiples of ***g(n)*** for all large ***n,*** i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$c_2 g(n) \le t(n) \le c_1 g(n)$ for all $n \ge n_0$.

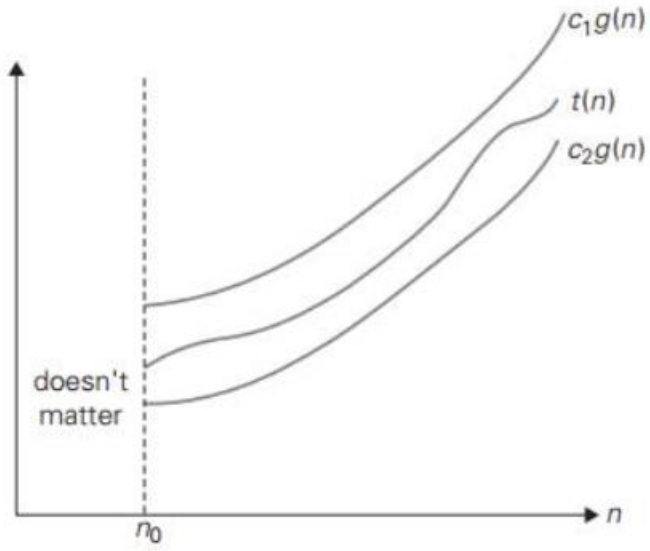


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Note: Kindly refer to your notes for problems on asymptotic notations

Basic Efficiency Classes

1	**constant**	**Best case**
log *n*	**logarithmic**	**Divide Ignore part**
n	**linear**	**Examine each Online/Stream Algs**
n* log *n	***n*-log-*n* or linearithmic**	**Divide Use all parts**
n^2	**quadratic**	**Nested loops**
n^3 n^k	**cubic**	**Nested loops Examine all *k*-tuples**
2^n	**exponential**	**All subsets**
***n*!**	**factorial**	**All permutations**

Unit 1- Part 2

Ms Nikitha Saurabh

Assistant Professor

Department of ISE

NMAMIT,Nitte

Useful summation formulas and rules

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Mathematical Analysis of Non recursive Algorithms

Example 1: Maximum element

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

- Let us denote C(n) the number of times this comparison is executed and try to find a formula expressing it as a function of size n. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, we get the following sum for C(n):

$$C(n) = \sum_{i=1}^{n-1} 1.$$

- Applying (S1) formula we get,

C(n)=u-l+1= n-1-1+1=n-1

Efficiency class=Θ (n)

Time efficiency of non-recursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- The natural measure of the input's size here is again n, the number of elements in the array.
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- Note, however, that the number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy.
- We will limit our investigation to the worst case only.

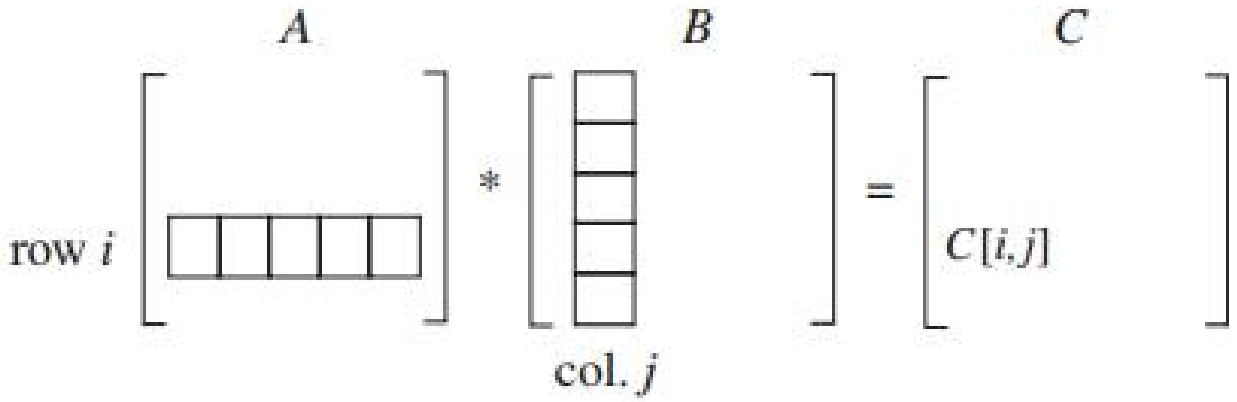
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in O(n^2).$$

Matrix Multiplication

Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable ***k*** ranging from the lower bound 0 to the upper bound ***n*** − 1. Therefore, the number of multiplications made for every pair of specific values of variables ***i*** and ***j*** is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Efficiency class = $\Theta(n^3)$

Example 4: Counting binary digits

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

- First, notice that the most frequently executed operation here is not inside the while loop but rather the comparison n > 1 that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

$$C(n)=\log_2(n)+1$$

$$\in \Theta(\log(n))$$

Mathematical Analysis of Recursive Algorithms

Example 1: Recursive evaluation of $n!$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

:

Identify Input size and Basic operation

- Consider ‘ n ‘ itself as an algorithm’s input size.
- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n).
- Since the function F (n) is computed according to the formula

  F (n) = F (n − 1) * n for n > 0,

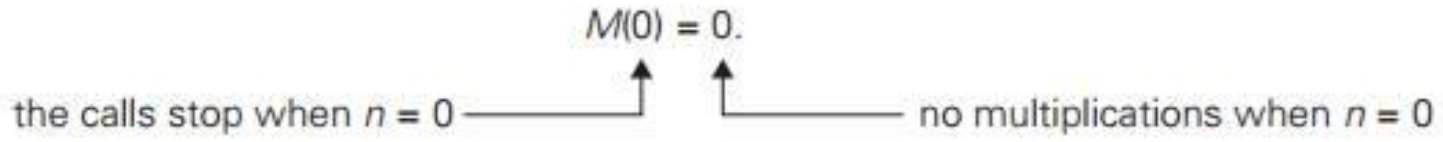
- The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

Set up recurrence relation and initial condition

if n = 0 return 1

- This tells us two things.
- First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0.
- Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.
- Therefore, the initial condition we are after is



Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad \textbf{(2.2)}$$
$$M(0) = 0.$$

Solve Recurrence using Backward Substitutions

$$
\begin{aligned}
M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\
&= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\
&= [M(n-3) + 1] + 2 = M(n-3) + 3.
\end{aligned}
$$

- In general, M(n) = M(n − i) + i.

- What remains to be done is to take advantage of the initial condition given.
- Since it is specified for n = 0, we have to substitute i = n in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

$= \Theta(n)$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (ie find a closed form or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Tower of Hanoi

In this puzzle, we have ***n*** disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

- The problem has an elegant recursive solution, which is illustrated in Fig-ure 2.4. To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

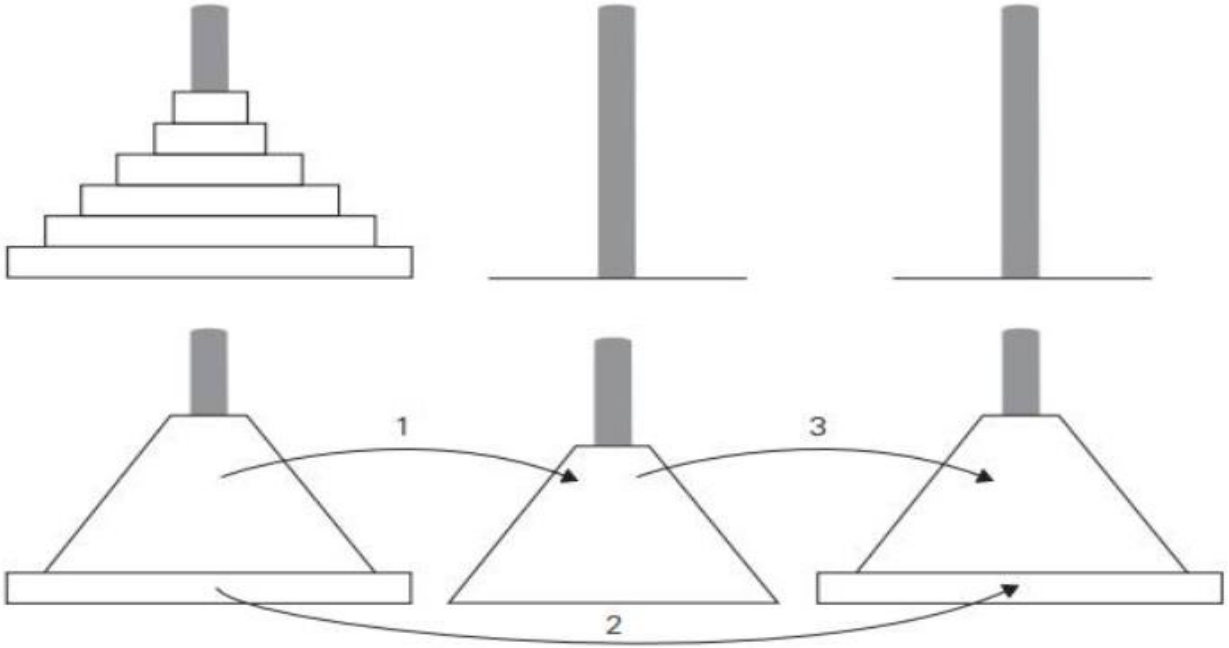


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \qquad \textbf{(2.3)}$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

$\Theta(2^n)$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Efficiency=$\Theta(2^n)$

Unit 2

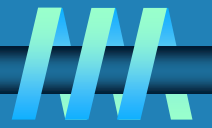
BRUTE FORCE(Chapter 3)

- **Selection Sort**
- **Bubble Sort**
- **Sequential Search**
- **Brute-Force String Matching**
- **Exhaustive Search**
- **Depth-First Search**
- **Breadth-First Search.**

DIVIDE AND CONQUER (Chapter 5)

- **Merge sort**
- **Quick sort**
- **Multiplication of large integers**
- **Strassen's Matrix Multiplication**

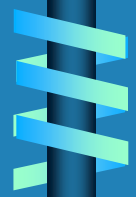
Brute Force



- A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved
- It involves checking every possible solutions to a problem.

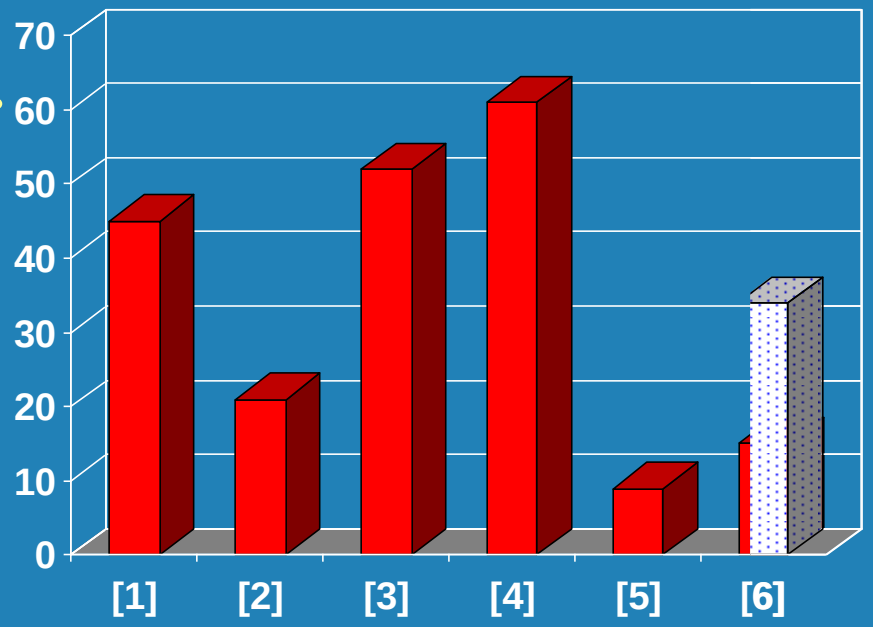
Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list



The Selectionsort Algorithm

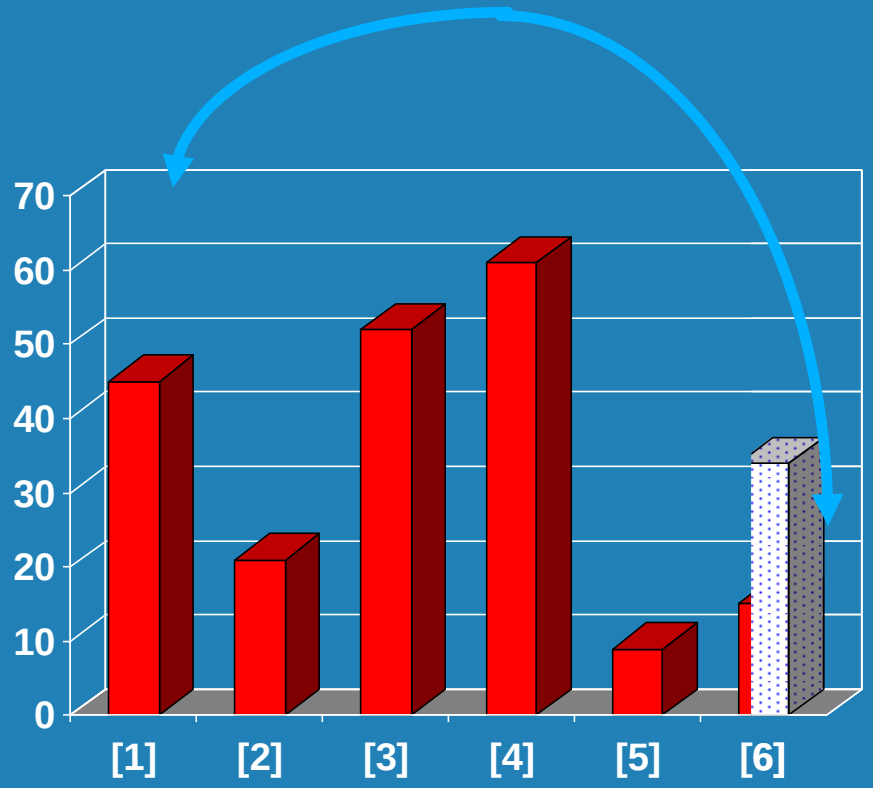
- Start by finding the smallest entry.



[0]	[1]	[2]	[3]	[4]	[5]

The Selectionsort Algorithm

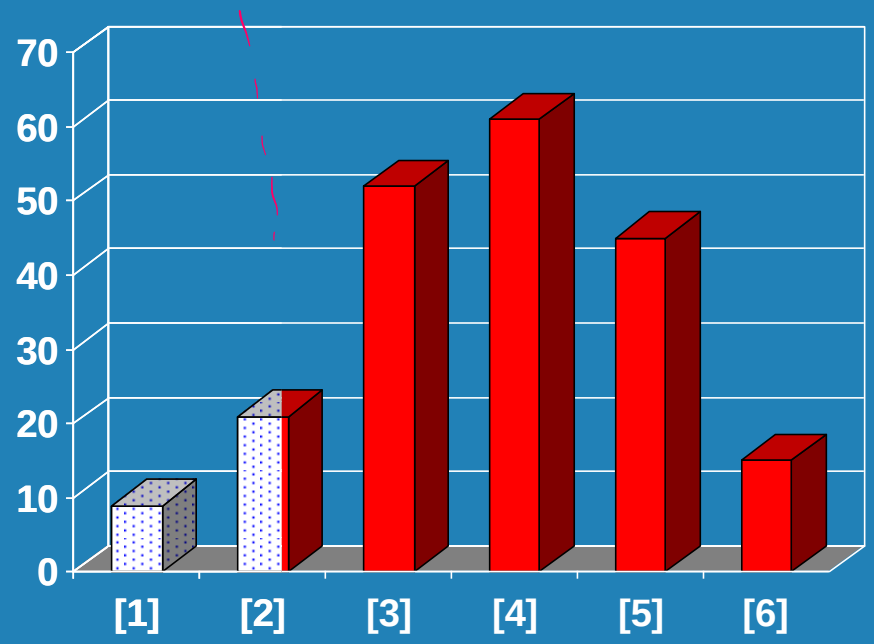
- Start by finding the smallest entry.
- Swap the smallest entry with the first entry.



[0]	[1]	[2]	[3]	[4]	[5]

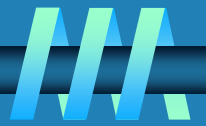
The Selectionsort Algorithm

- Start by finding the smallest entry.
- Swap the smallest entry with the first entry.

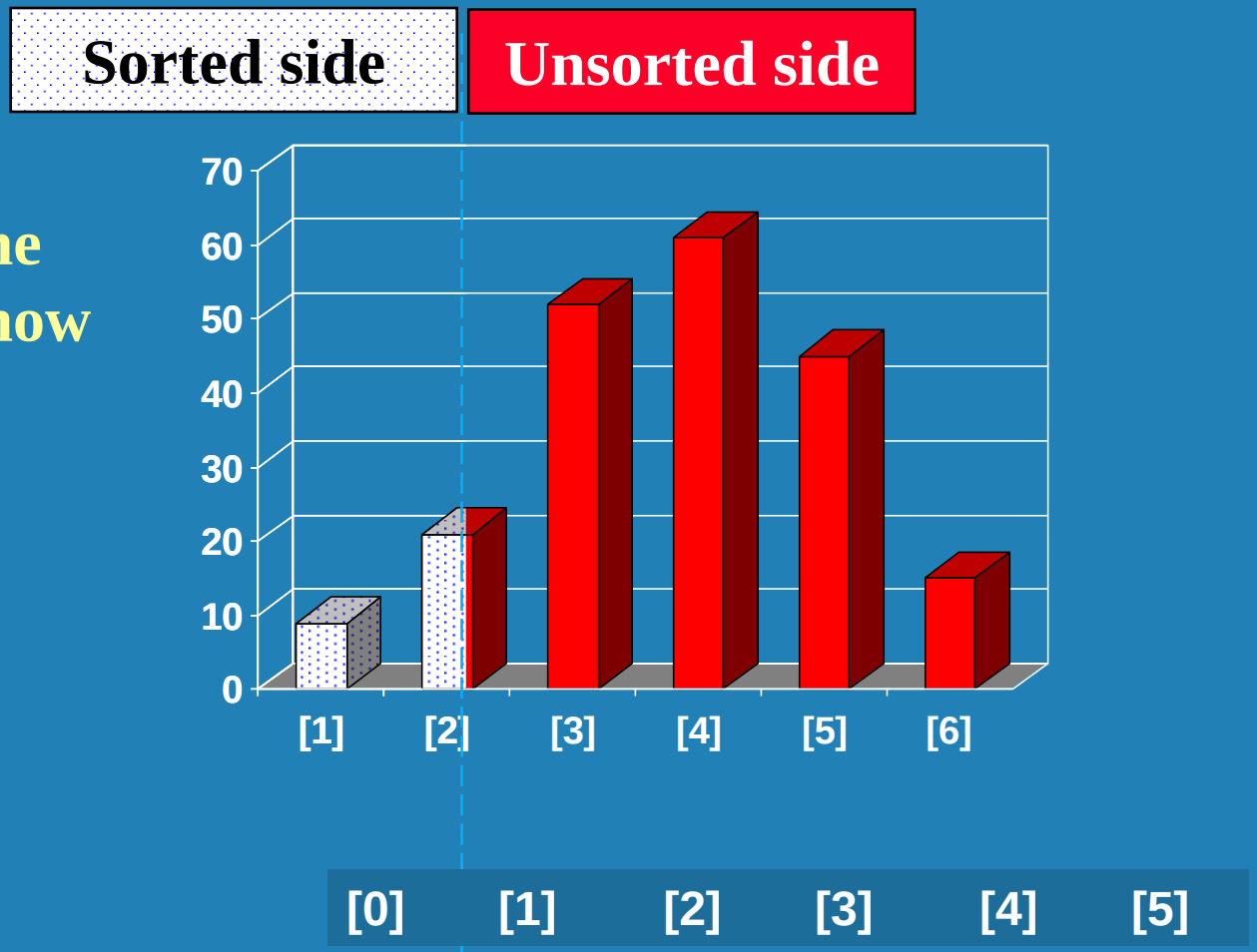


[0]	[1]	[2]	[3]	[4]	[5]

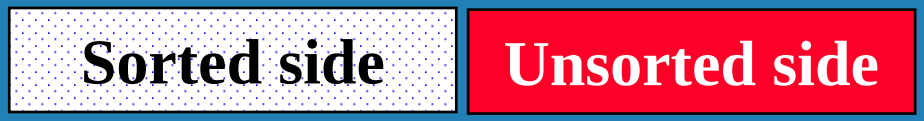
The Selectionsort Algorithm



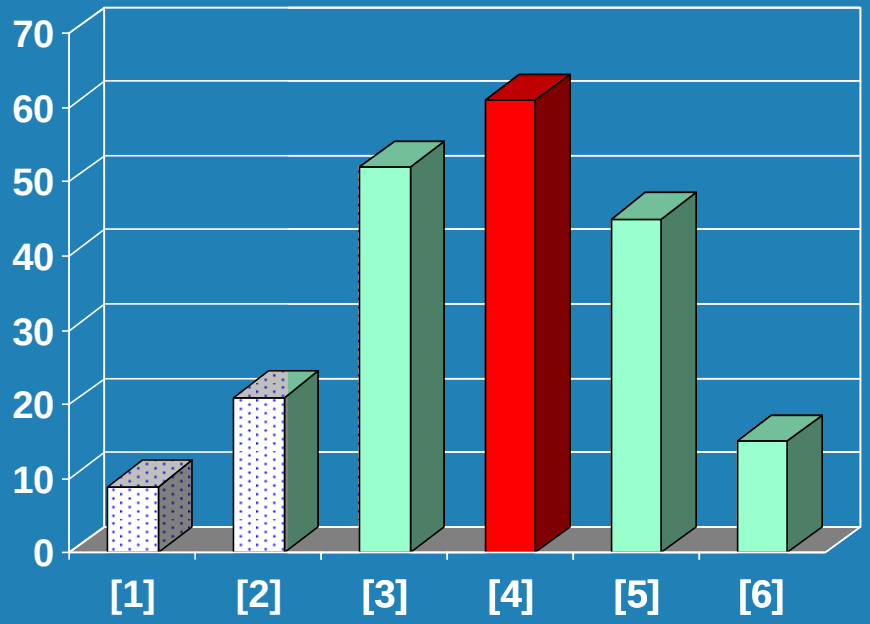
- Part of the array is now sorted.



The Selectionsort Algorithm

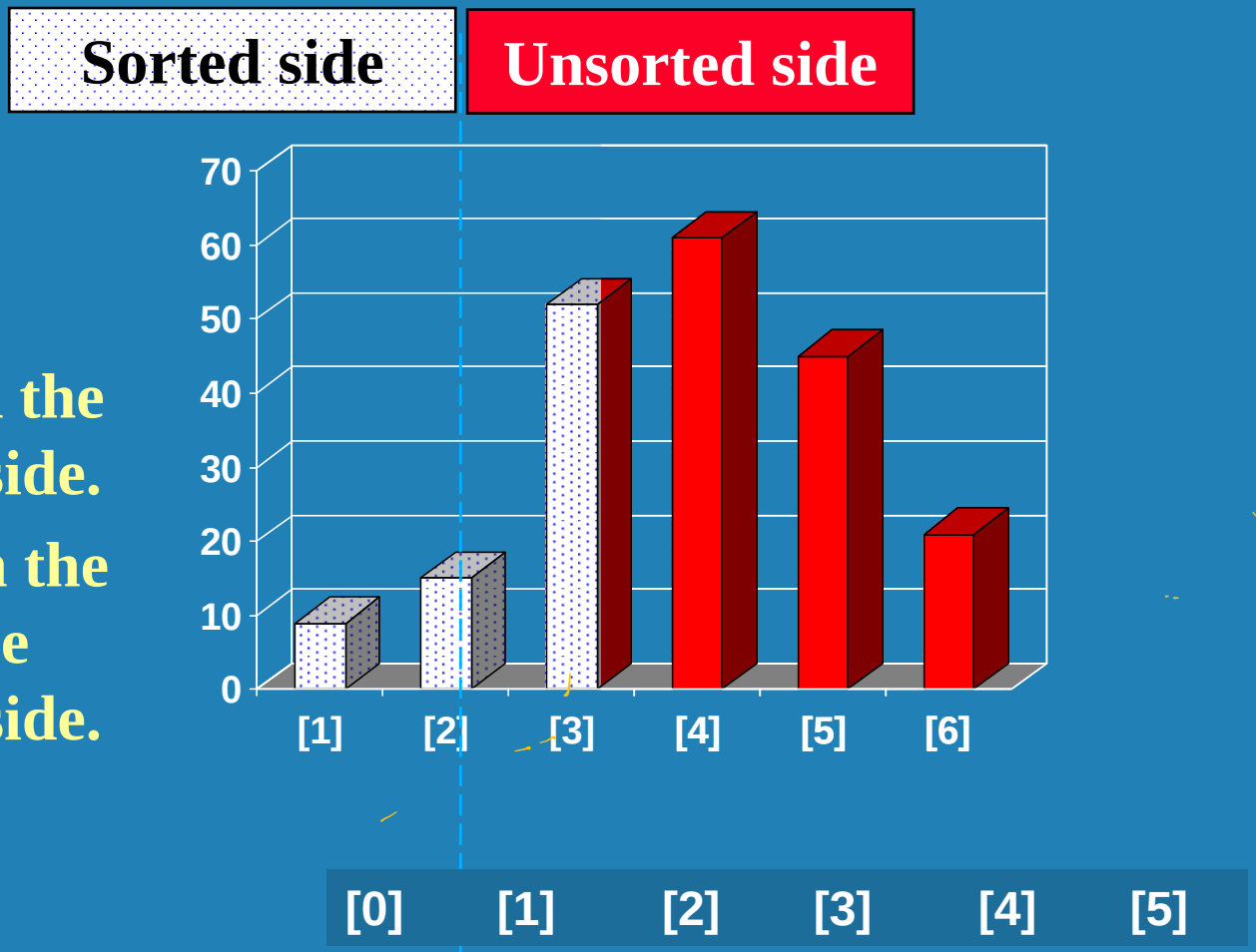


- Find the smallest element in the unsorted side.



[0]	[1]	[2]	[3]	[4]	[5]

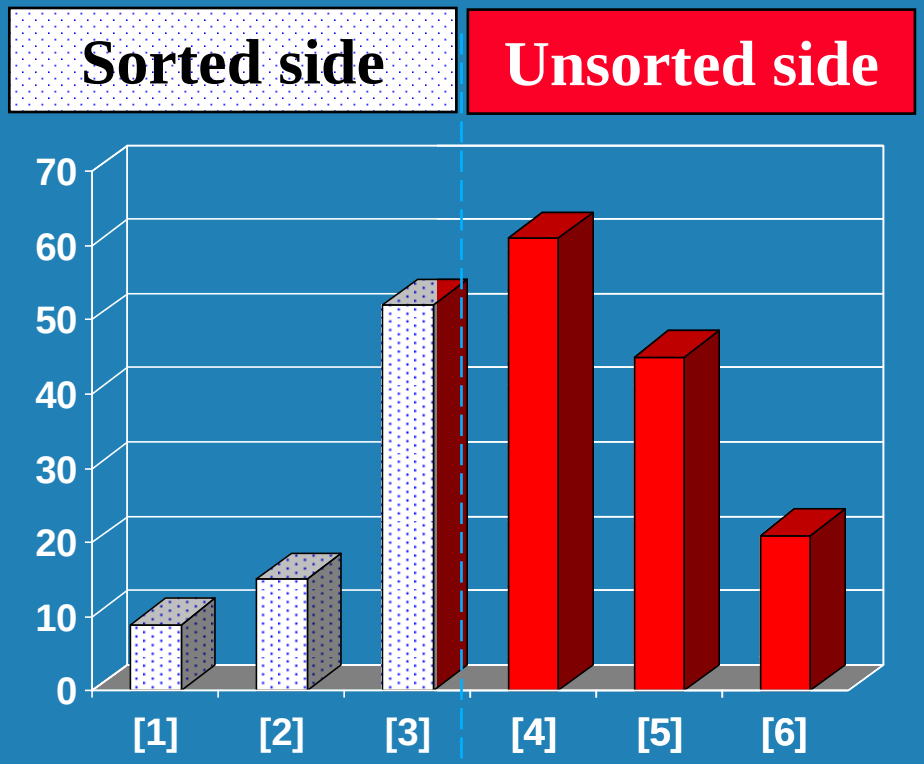
The Selectionsort Algorithm



- Find the smallest element in the unsorted side.
- Swap with the front of the unsorted side.

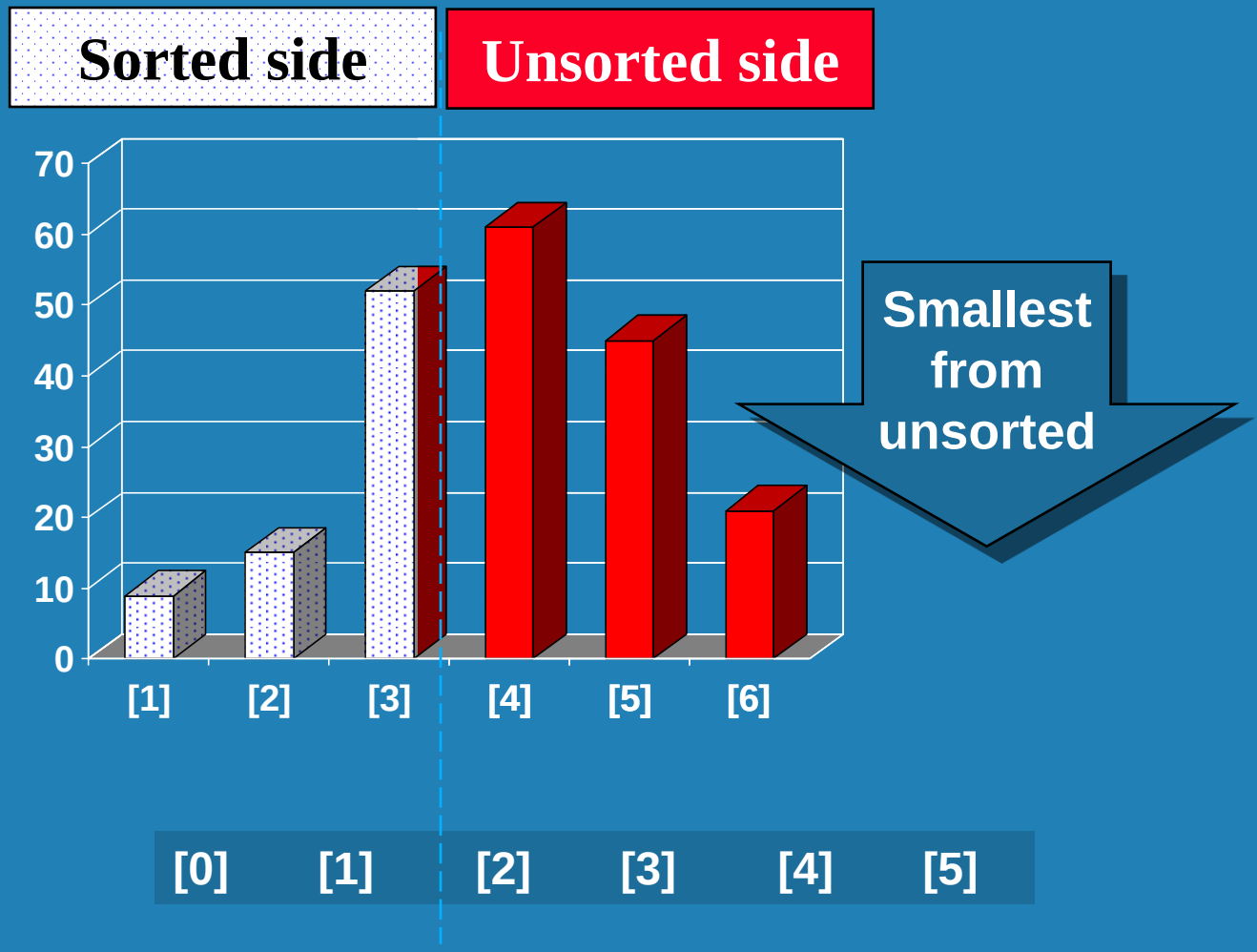
The Selectionsort Algorithm

- We have increased the size of the sorted side by one element.



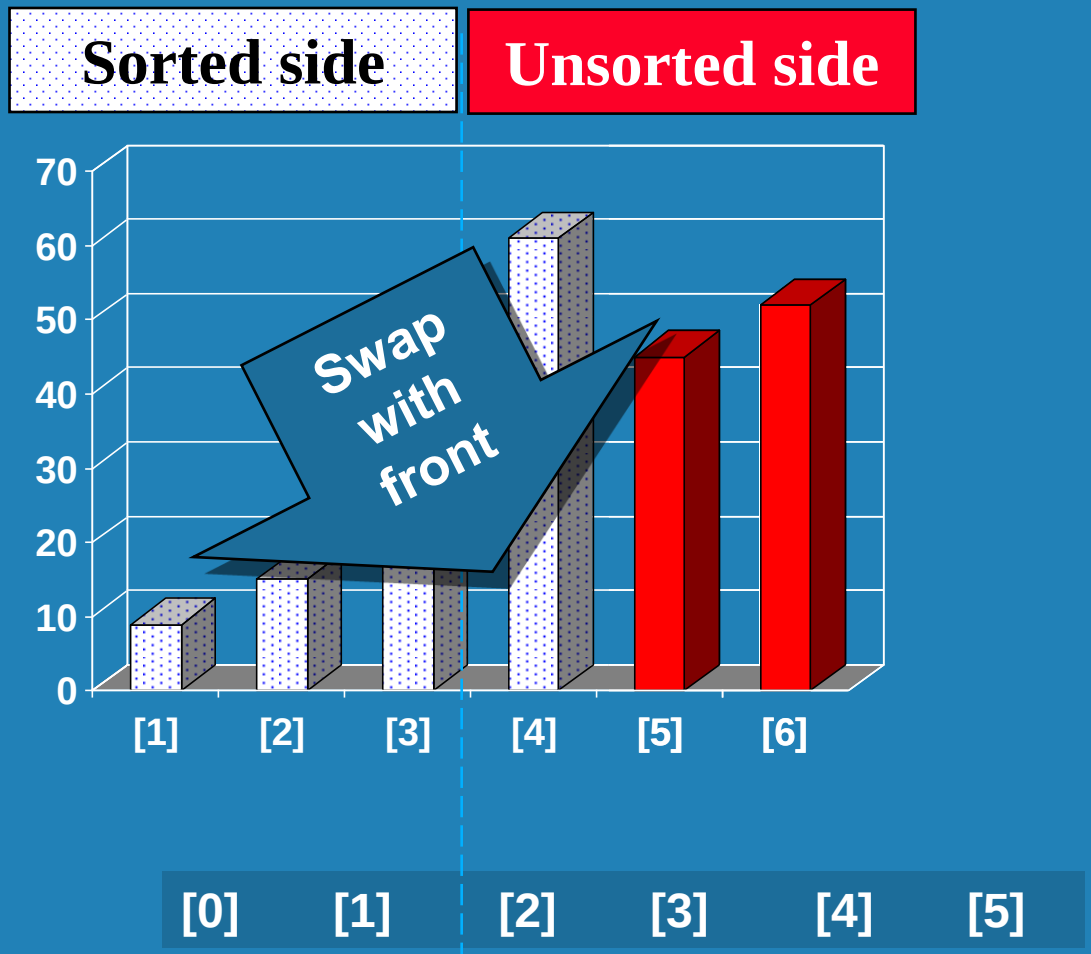
The Selectionsort Algorithm

- The process continues...

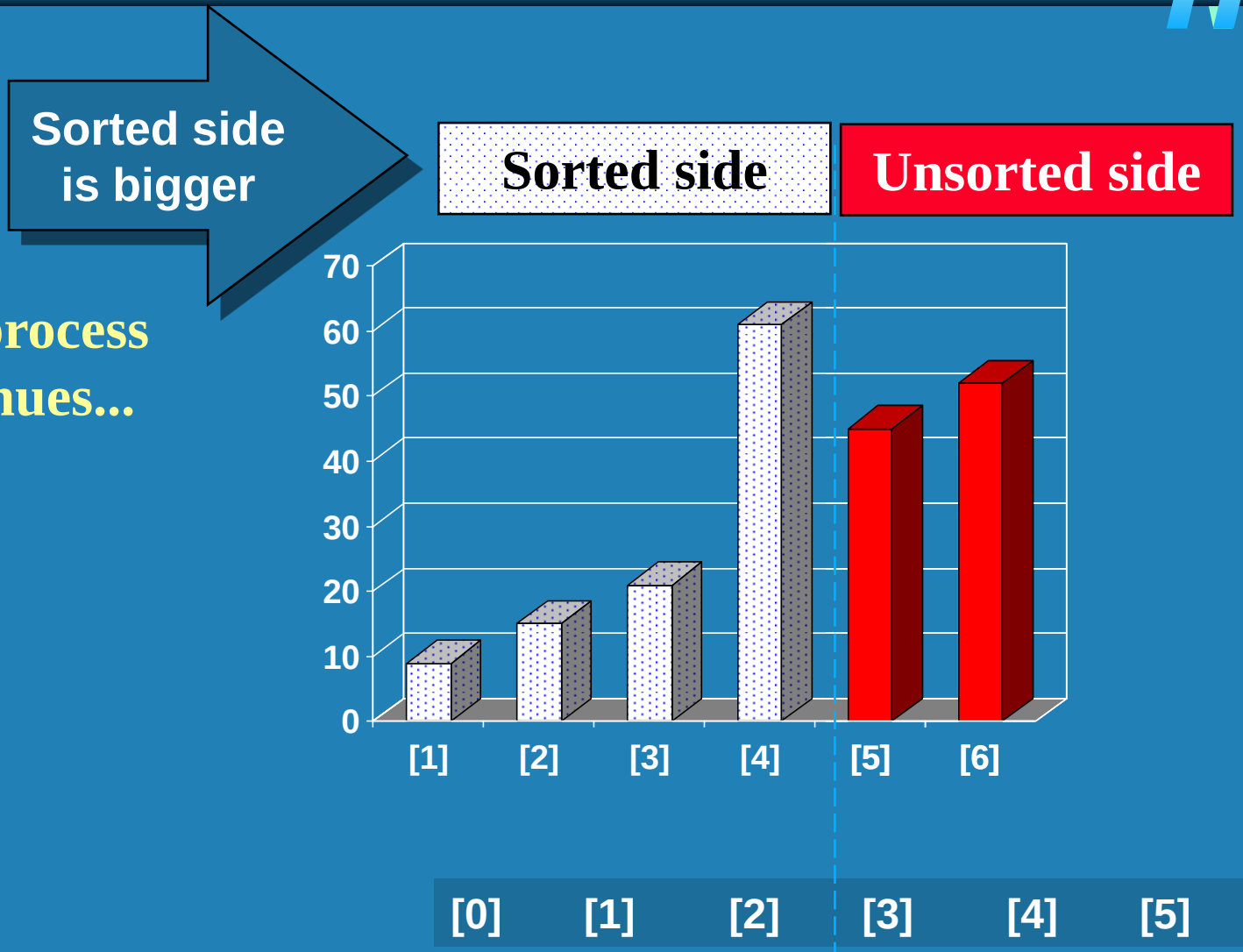


The Selectionsort Algorithm

- The process continues...



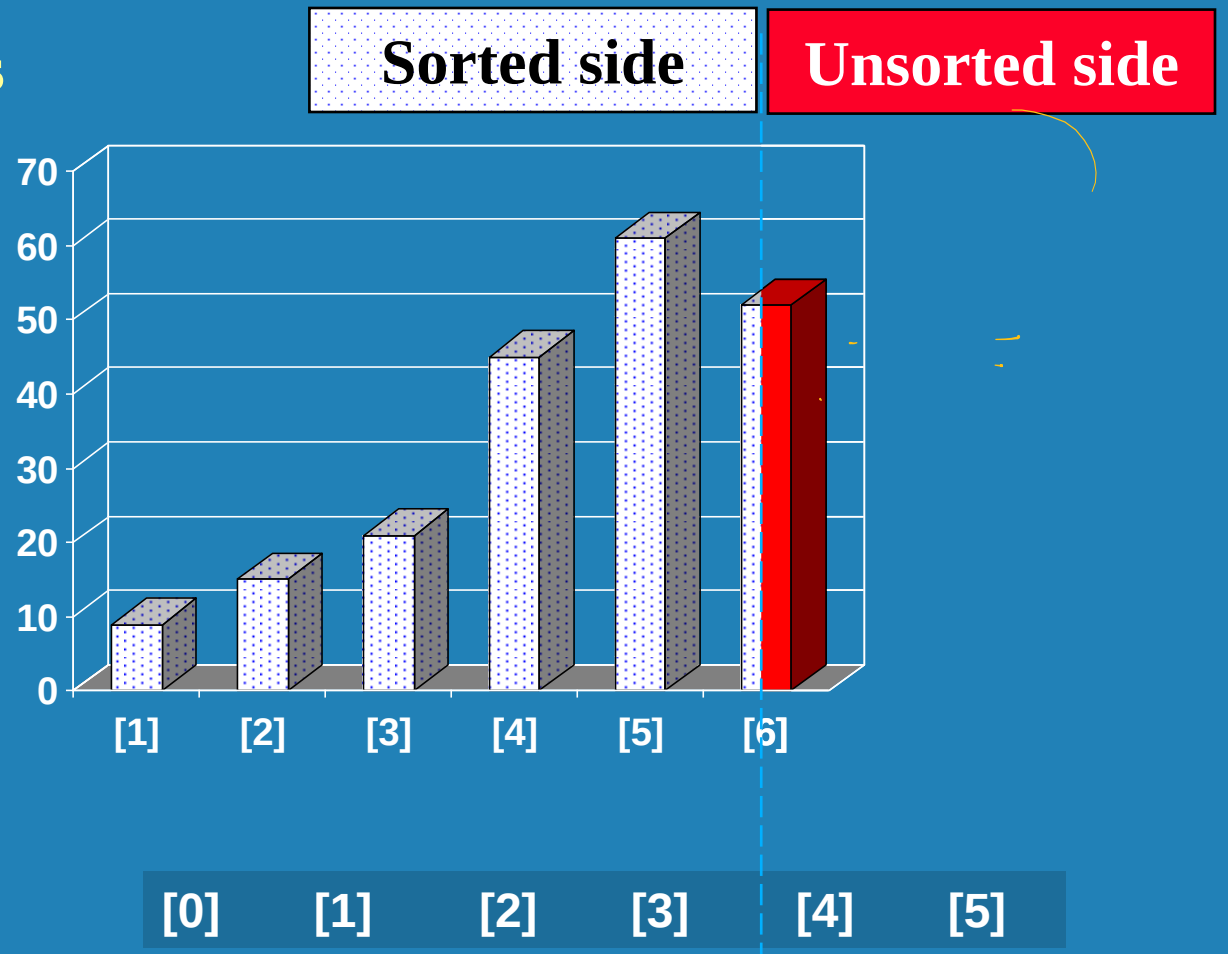
The Selectionsort Algorithm



- The process continues...

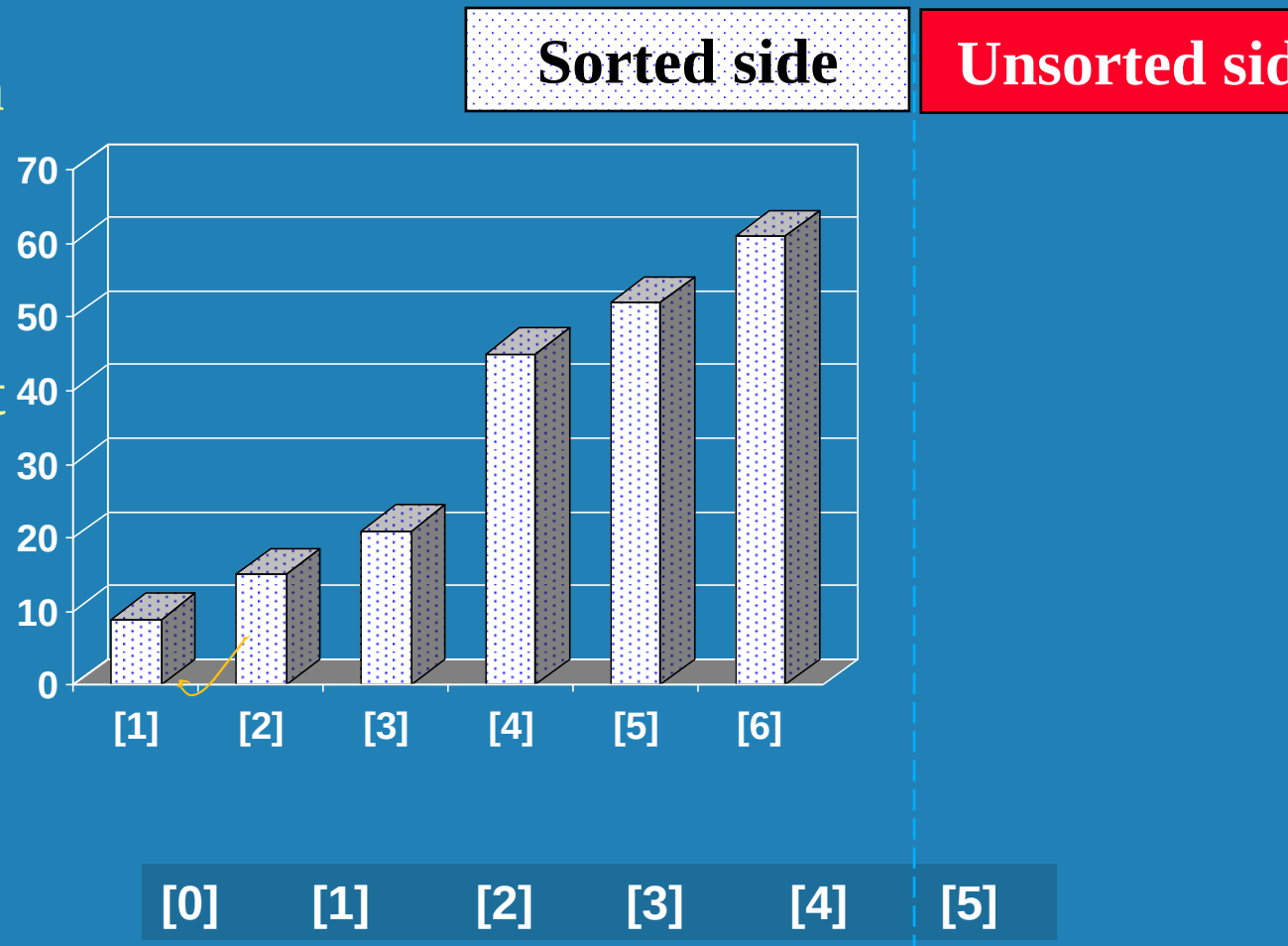
The Selectionsort Algorithm

- The process keeps adding one more number to the sorted side.
- The sorted side has the smallest numbers, arranged from small to large.



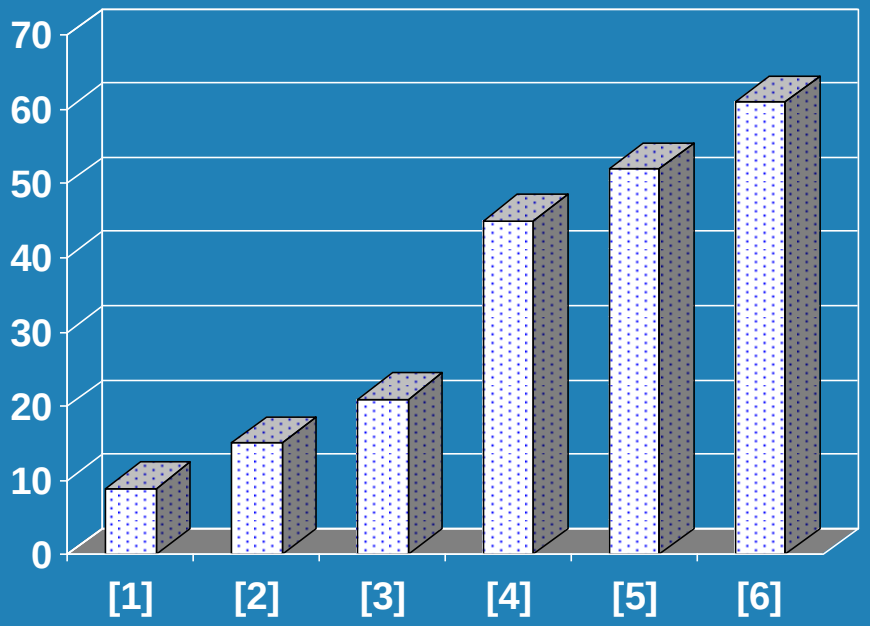
The Selectionsort Algorithm

- We can stop when the unsorted side has just one number, since that number must be the largest number.



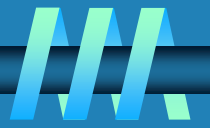
The Selectionsort Algorithm

- The array is now sorted.
- We repeatedly selected the smallest element, and moved this element to the front of the unsorted side.



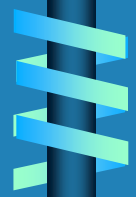
[0]	[1]	[2]	[3]	[4]	[5]

Brute-Force Sorting Algorithm

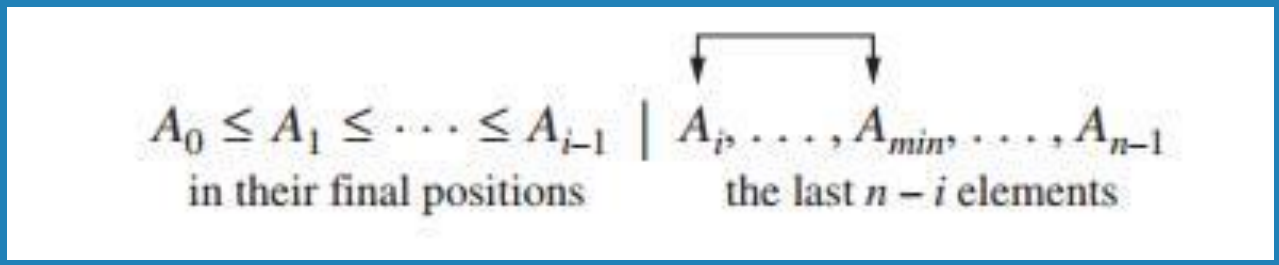


Selection Sort

- *Scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.*
- *Then we scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position.*

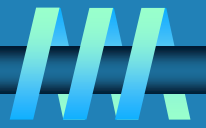


Generally, on the i^{th} pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with Ai:



After ***n*** − 1 passes, the list is sorted.

Selection Sort



ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Example: 89, 45, 68, 90, 29, 34, 17

89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Analysis of Selection Sort

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

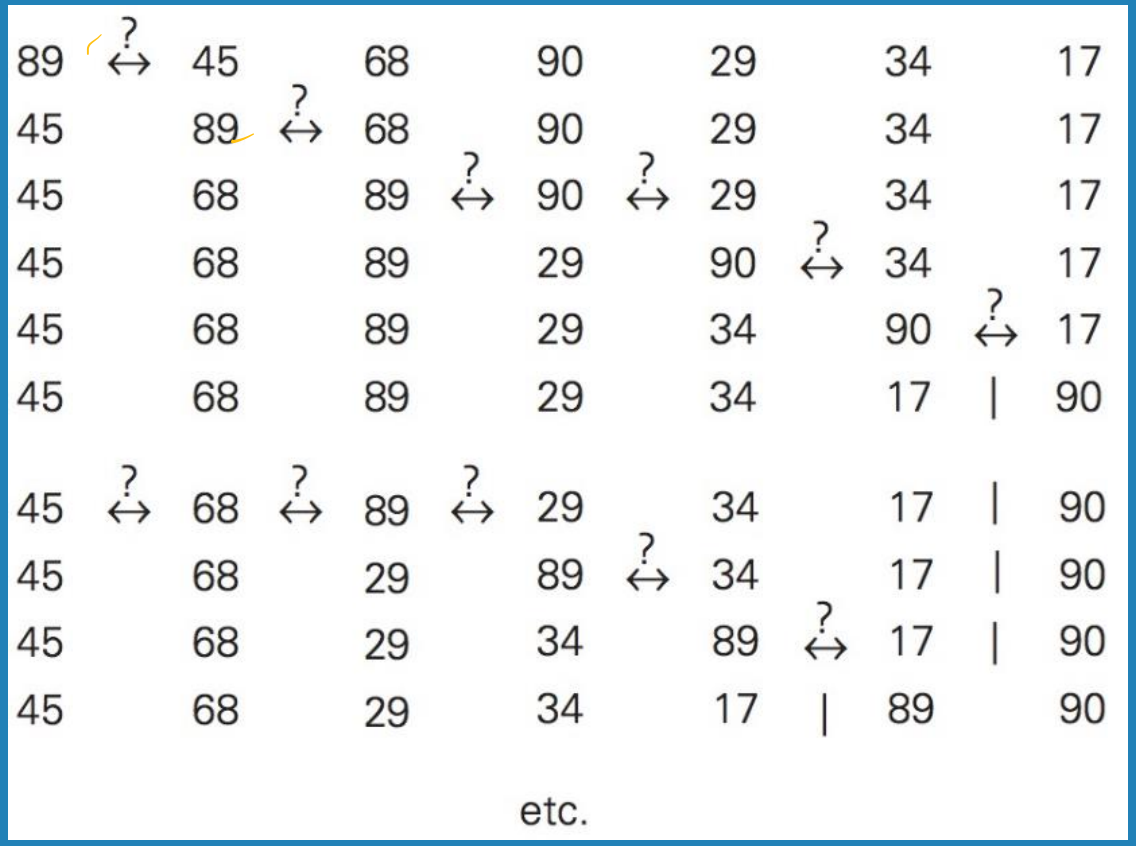
Efficiency: $\Theta(n^2)$

Number of key swaps= n ,specifically it is n-1

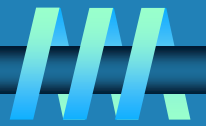
Brute-Force Sorting Algorithm

Bubble Sort **compare adjacent elements of the list and exchange them if they are out of order.**

Example:



Bubble Sort Algorithm



ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 0$ **to** $n-2$ **do**

  for $j \leftarrow 0$ **to** $n-2-i$ **do**

    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Analysis of Bubble Sort

Efficiency:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Key swapping

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in O(n^2).$$

Assignment

Consider the following list of integers.
Sort them in ascending order using the below mentioned Algorithms.

- Selection Sort
- Bubble Sort

Kindly Provide step wise Handwritten solution

90 , 20, 40, 15,70,45,39

Brute-Force Sequential Search

- **Brute-force algorithm: compare successive elements of a given list with a given search key until either a match is encounter or the list is exhausted without finding a match**

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Note: Already studied under Algorithm Analysis framework- best case, worst case, average case

Brute-Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until
  - all characters are found to match (successful search); or
  - a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Pseudocode

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

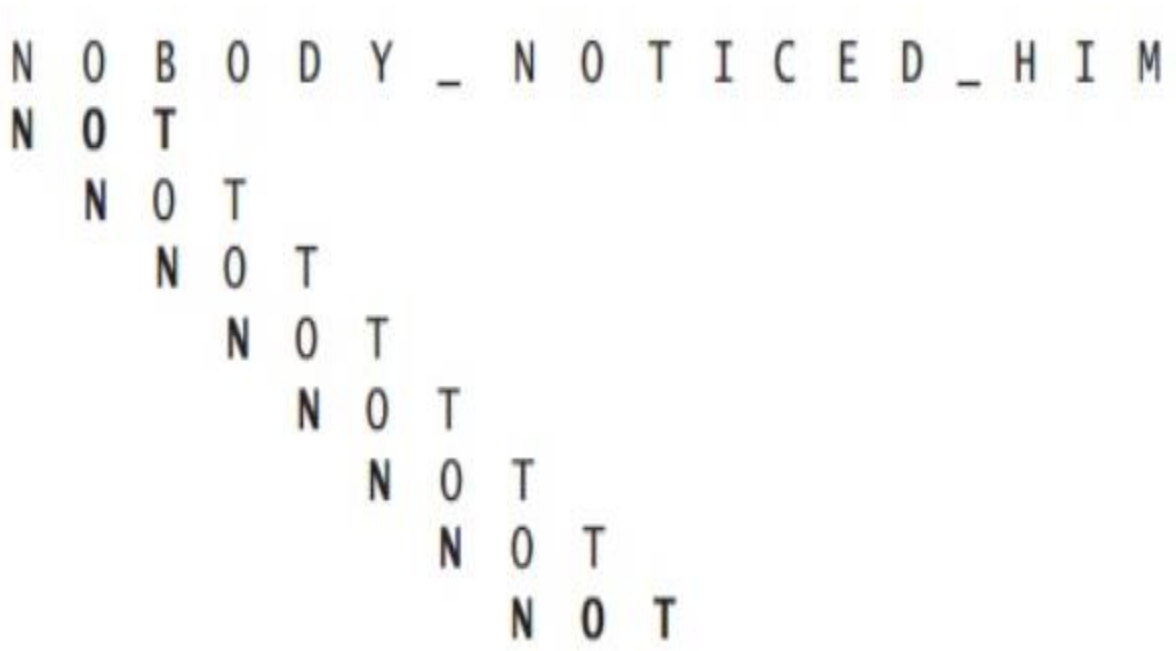
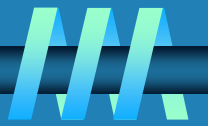


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

Efficiency



Basic Operation: Character comparison

- **Worst Case: Pattern found at the end of the string(success)**
- **Pattern not found(unsuccess)**

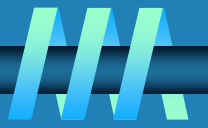
$$C(n) = m(m-n+1) = O(nm)$$

Brute-Force Strengths and Weaknesses

- **Strengths**
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

- **Weaknesses**
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

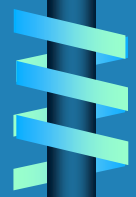
Exhaustive Search



An exhaustive search solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found



Example 1: Traveling Salesman Problem

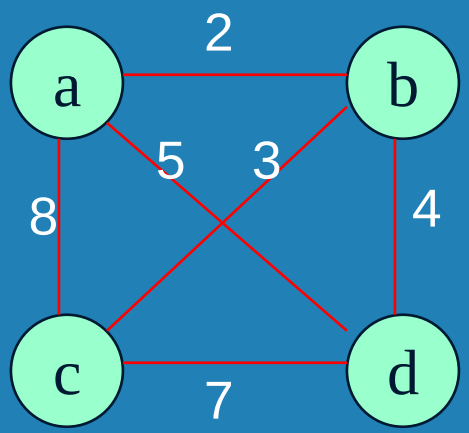
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph

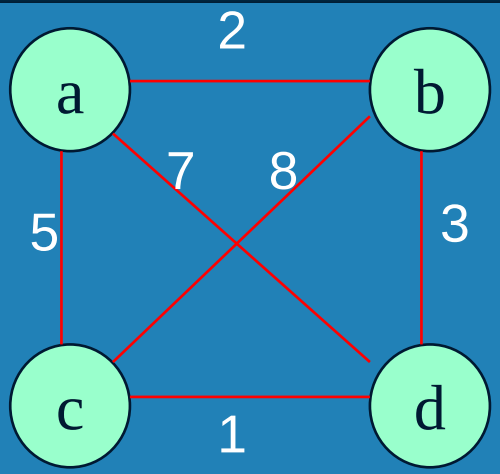
- *Hamiltonian circuit*: a sequence of *n + 1* adjacent vertices where the first vertex of the sequence is the same as the last one and all the other *n – 1* vertices are distinct.

TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

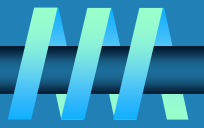


- **Basic operation: Finding (n-1)! hamiltonion circuits because we need to generate n-1 permutations of intermediate cities. Hence the problem belongs to efficiency class n!**
- **Efficiency: $\Theta(n!)$**



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Example 2: Knapsack Problem



Given *n* items:

- **weights:** w_1 w_2 … w_n
- **values:** v_1 v_2 … v_n
- **a knapsack of capacity *W***

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

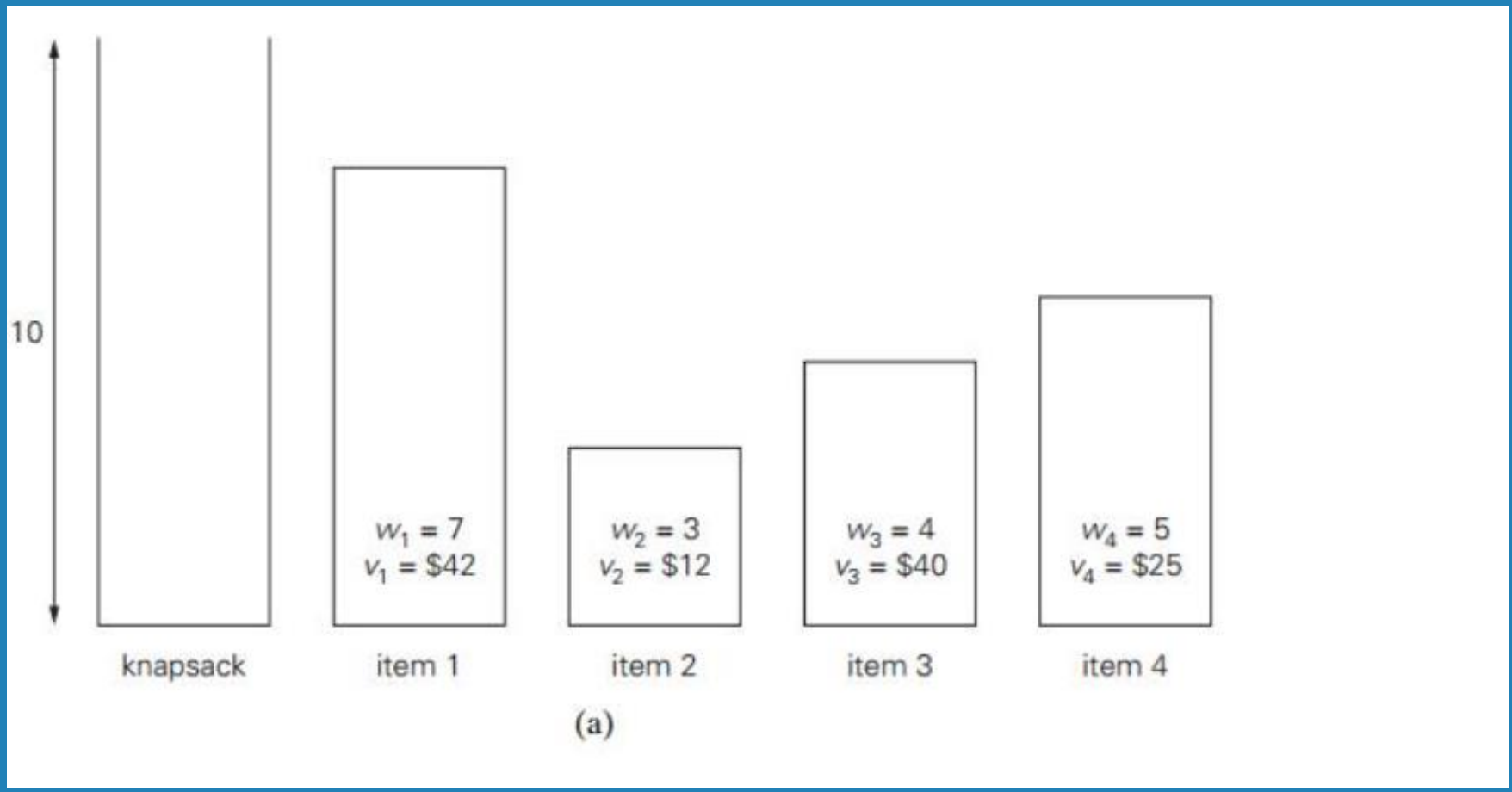
item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{0}	0	$0
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency: $\Theta(2^n)$

Example


10
$w_1 = 7$
$v_1 = \$42$
$w_2 = 3$
$v_2 = \$12$
$w_3 = 4$
$v_3 = \$40$
$w_4 = 5$
$v_4 = \$25$
knapsack
item 1
item 2
item 3
item 4
(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

Example 3: The Assignment Problem

- **There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost.**

	Job 1	Job2
Person 1	3	2
Person 2	8	4

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

Solution:

P1=1 p2=2 = 3+4=7 → Cheapest solution

P1=2 p2=1 = 2 +8=10

A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Assignment Problem by Exhaustive Search

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

etc.

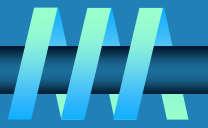
FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search.

- **How many assignments are there?**
  - **Permutation of *n* jobs**
- **Efficiency? *Θ(n!)***

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known way to get exact solution

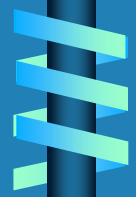
Graph Traversal Algorithms



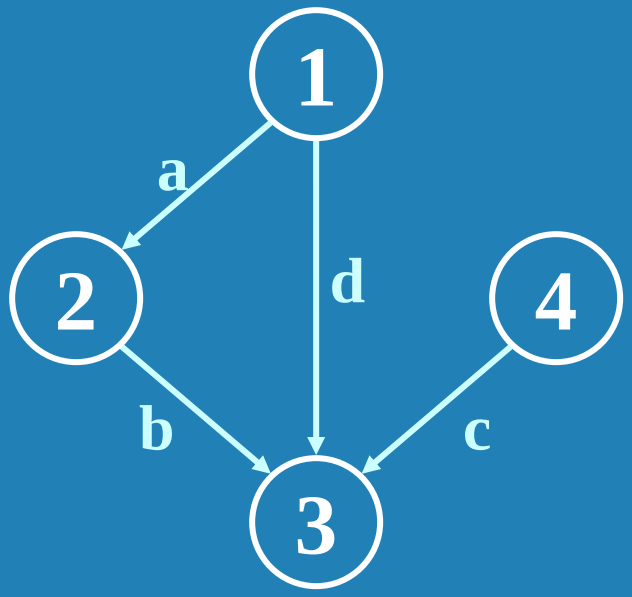
Many problems require processing all graph vertices (and edges) in systematic fashion

<u>Graph traversal algorithms</u>:

- **Depth-first search (DFS)**
- **Breadth-first search (BFS)**



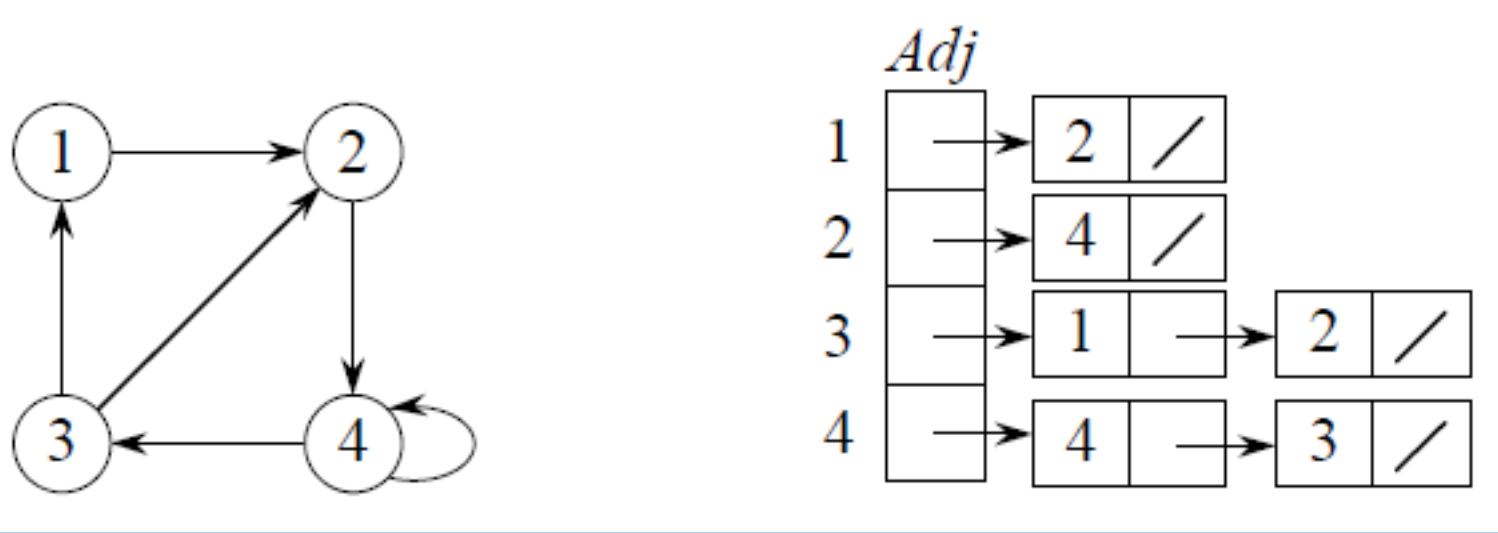
Graphs: Adjacency Matrix



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example: For a directed graph



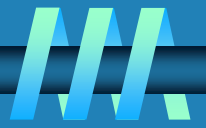
Depth-First Search (DFS)

- Visits graph's vertices by always moving "deeper" from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.

- Uses a stack
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, *i.e.*, when there is no adjacent unvisited vertex

- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

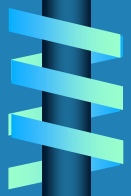
Pseudocode of DFS



ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)



Example: DFS traversal of undirected graph

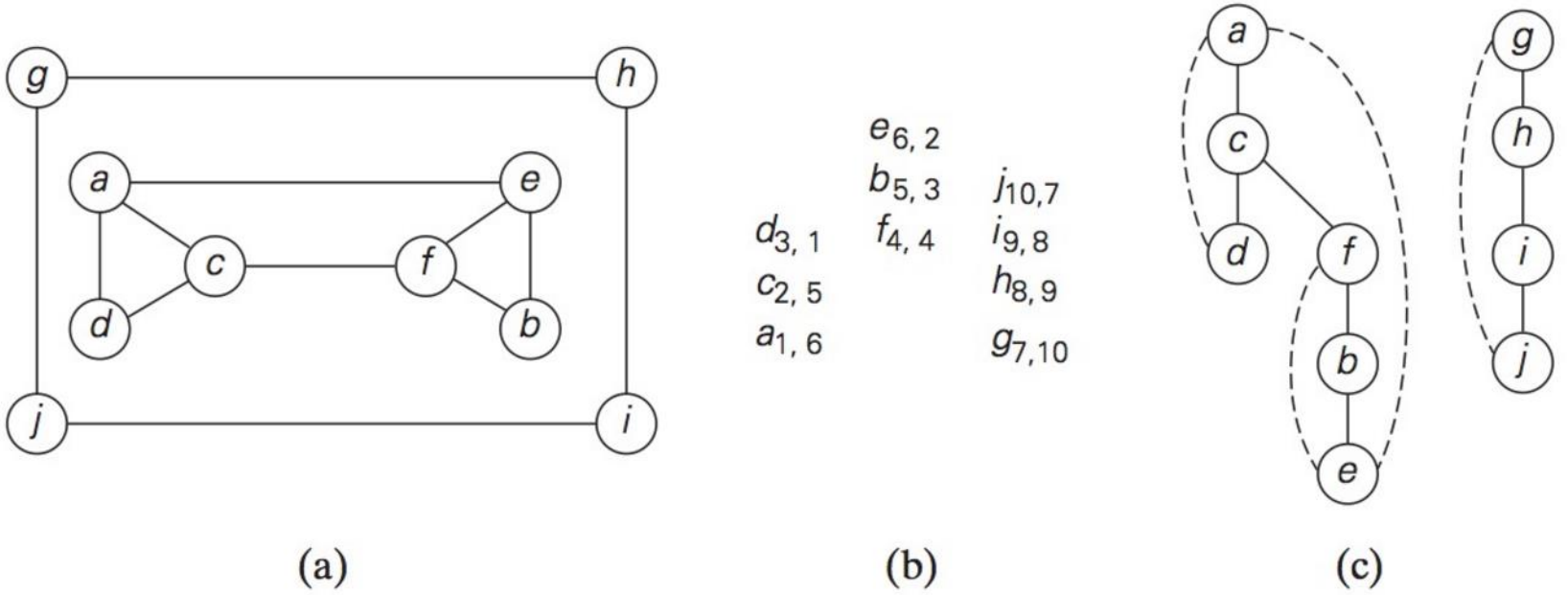
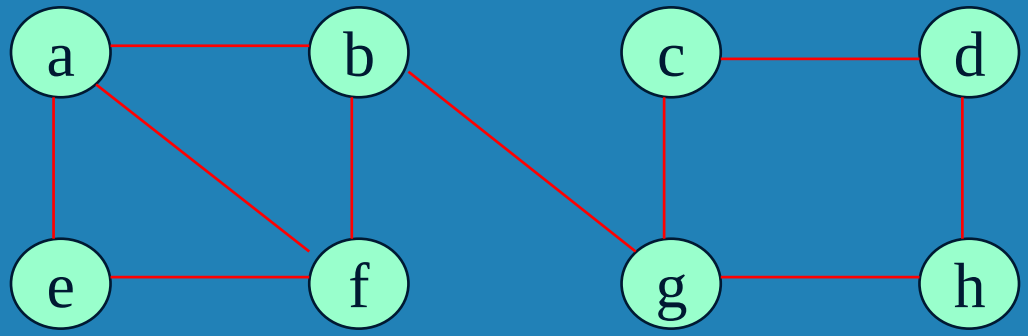
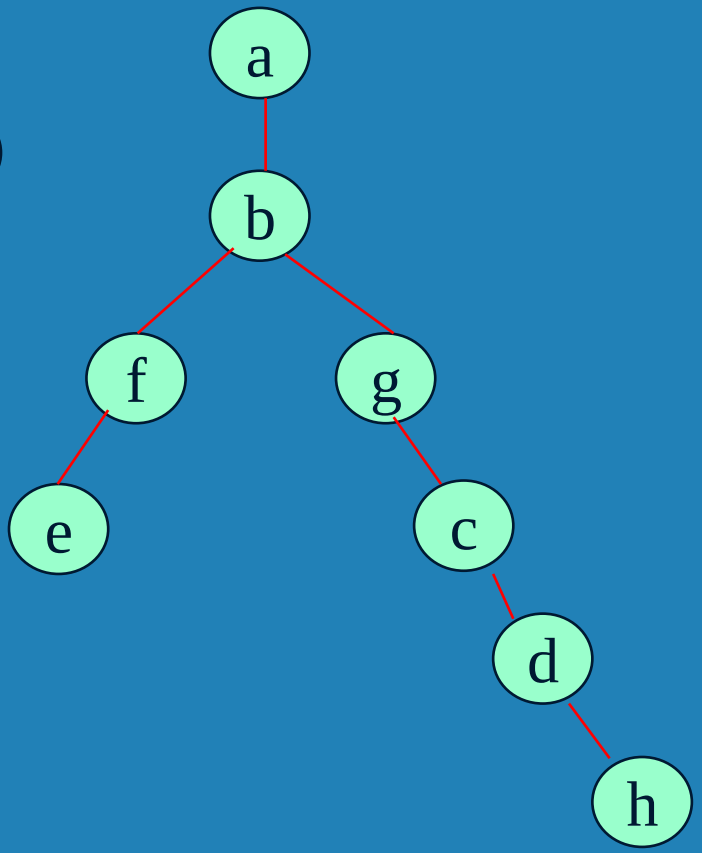


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Example: DFS traversal of undirected graph



DFS tree:



DFS traversal stack:

$h_{8,3}$
$d_{7,4}$
$c_{6,5}$
$g_{5,6}$

$e_{4,1}$
$f_{3,2}$
$b_{2,7}$
$a_{1,8}$

Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)
- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity
  - finding articulation points and biconnected components
  - searching state-space of problems for solution (AI)

Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- "Redraws" graph in tree-like fashion (with tree edges and cross edges for undirected graph)

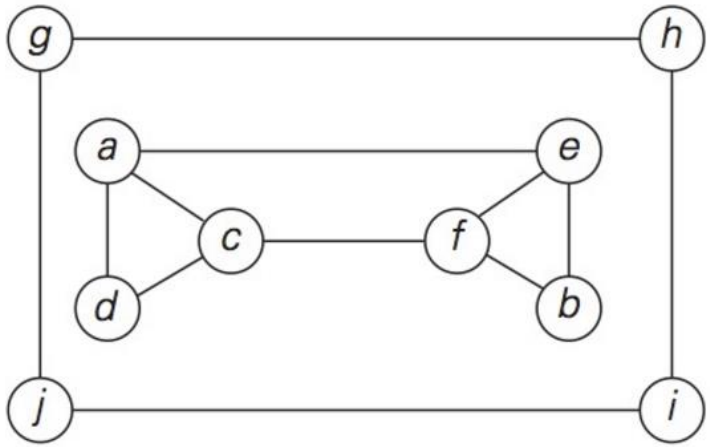
Pseudocode of BFS

ALGORITHM *BFS*(G)

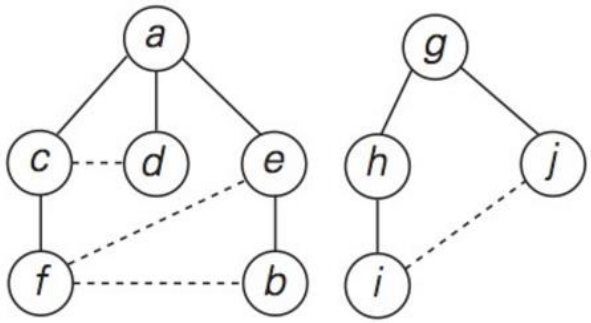
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Example of BFS traversal of undirected graph



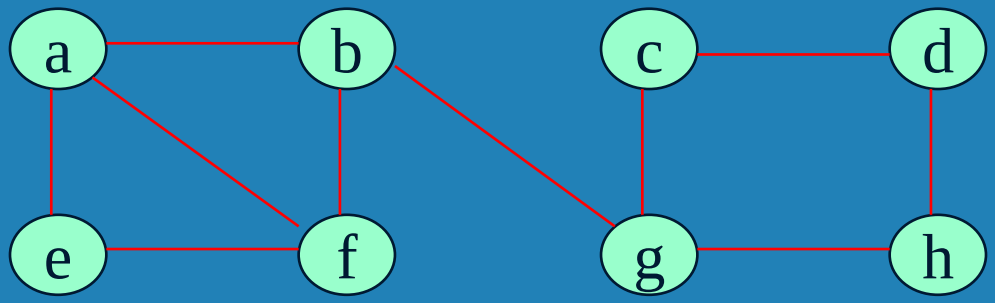
$a_1\ c_2\ d_3\ e_4\ f_5\ b_6$
$g_7\ h_8\ j_9\ i_{10}$



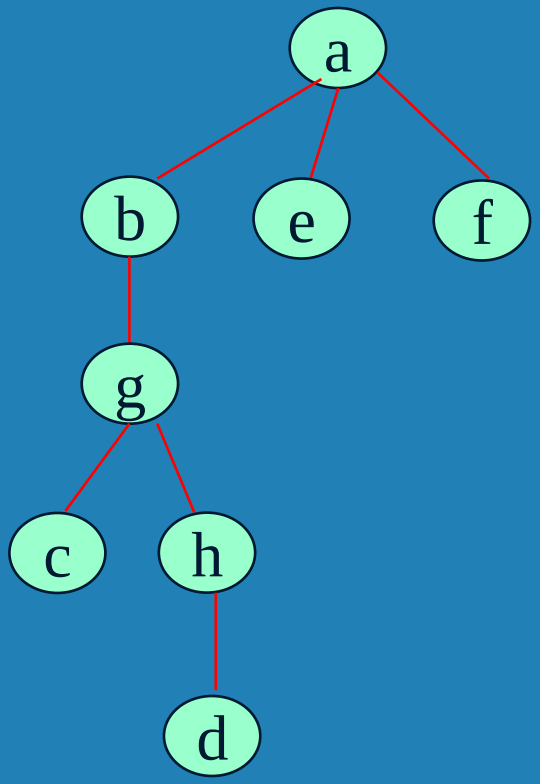
(a) (b) (c)

FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

Example of BFS traversal of undirected graph



BFS tree:



BFS traversal queue:

$a_1\ b_2\ e_3\ f_4\ g_5\ c_6\ h_7\ d_8$

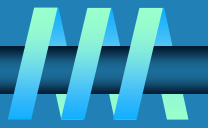
Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- Yields single ordering of vertices (order added/deleted from queue is the same)
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges

Unit 2- Chapter 5

Ms Nikitha Saurabh
Assistant Professor
Dept. of ISE
NMAMIT,Nitte

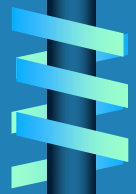
Divide-and-Conquer



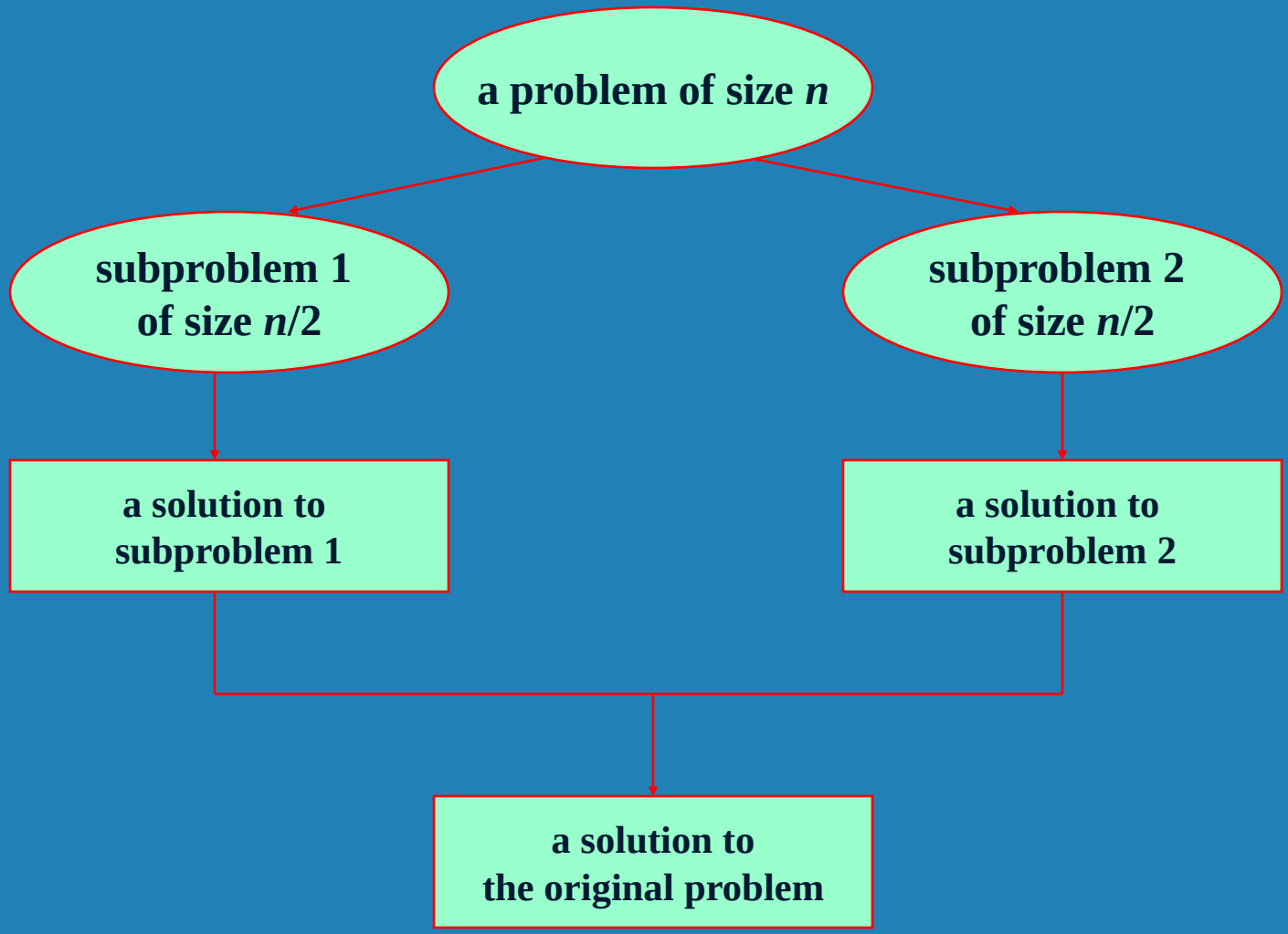
The most-well known algorithm design strategy:

1. **Divide instance of problem into two or more smaller instances**
2. **Solve smaller instances recursively**
3. **Obtain solution to original (larger) instance by combining these solutions**

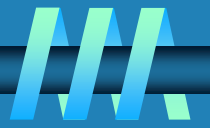
.



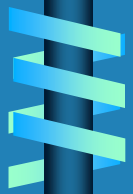
Divide-and-Conquer Technique (cont.)



Divide-and-Conquer Examples



- Sorting: mergesort and quicksort
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm



General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

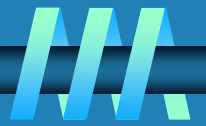
Note: The same results hold with O instead of Θ.

For example, the recurrence for the number of additions A(n) made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

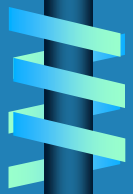


Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

- **Note : Refer notes for the solutions for above equations**



Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis of Mergesort

- **Number of key comparisons – recurrence:**

$$C(n) = 2C(n/2) + C_{merge}(n) = 2C(n/2) + n\text{-}1$$

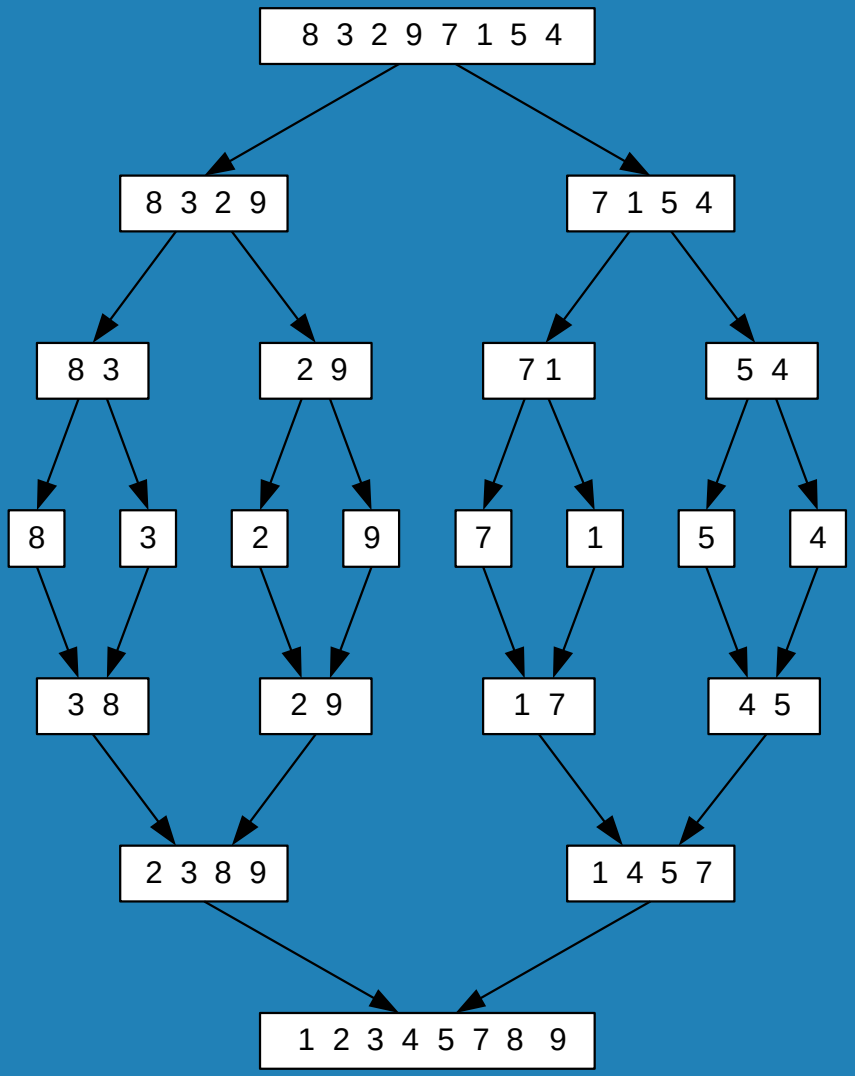
For merge step – worst case:

$$C_{merge}(n) = n\text{-}1$$

Therefore ,according to master's theorem,

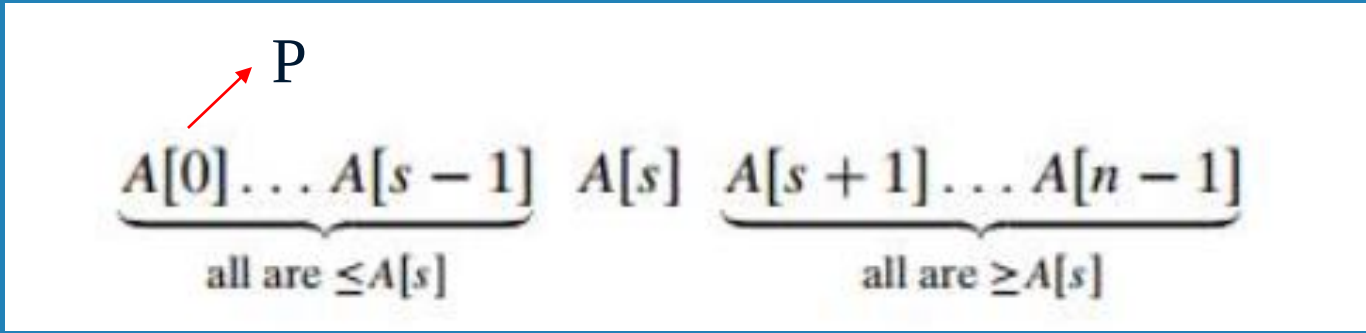
All cases have same efficiency: $\Theta(n \log n)$

Mergesort Example



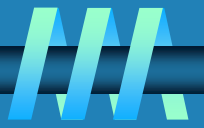
Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



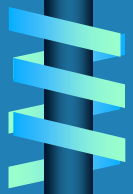
- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quicksort Algorithm



ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order

if l < r
s ←Partition(A[l..r]) //s is a split position
Quicksort(A[l..s − 1])
Quicksort(A[s + 1..r])

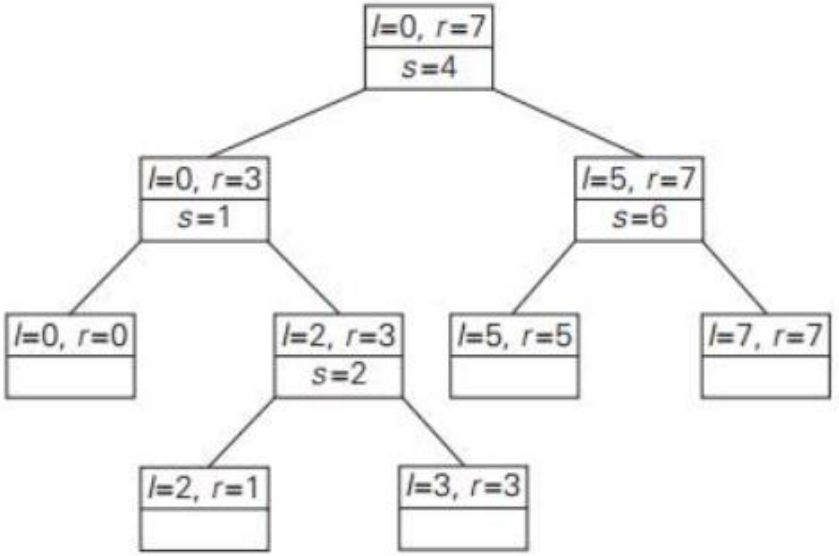


Hoare's (Two-way) Partition Algorithm

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//          as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//          indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//          this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])    //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			ij				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9



(b)

Analysis of Quicksort

- **Best case:**

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

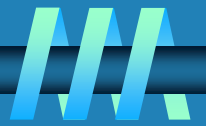
- **Worst case:**

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3$$

- **Average case:**

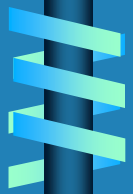
$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$



Performance:

- Best case: split in the middle — $\Omega(n \log n)$
- Worst case: sorted array! — $O(n^2)$
- Average case: random arrays — $\Theta(n \log n)$



Multiplication of Large Integers

- Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

  A = 12345678901357986429 B = 87654321284820912836

  Obviously, if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.
- Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, this turns out not to be the case.
- The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

- To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$23 = 2 \cdot 10^1 + 3 \cdot 10^0$ and $14 = 1 \cdot 10^1 + 4 \cdot 10^0$.

Now let us multiply them:

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)\ 10^1 + (3 * 4)\ 10^0.$$

- **Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit numbers a = a1a0 and b = b1b0, their product c can be computed by the formula**

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

Where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the ***a***'s digits and the sum of the ***b***'s digits minus the sum of c_2 and c_0.

- **Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:**

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

For 4-digit numbers: A * B where A = 2135 and B = 4014

A = $(21 \cdot 10^2 + 35)$, B = $(40 \cdot 10^2 + 14)$

So, A * B = $(21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if A = A_1A_0 and B = B_1B_0 (where A and B are *n*-digit, and A_1, A_0, B_1, B_0 are *n* / 2-digit numbers),

i.E A= $A_1 10^{n/2} + A_0$, B = $B_1 10^{n/2} + B_0$

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_0 + A_2 * B_0) \cdot 10^{n/2} + A_0 * B_0$

Recurrence of Large integer multiplication

- **Since multiplication of n-digit numbers requires three multiplications of n/2-digit numbers, the recurrence for the number of multiplications M(n) is**

- $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

- **Solving it by backward substitutions for $n = 2^k$ yields**

- $M(2^k) = 3M(2^{k-1})$
  $= 3[3M(2^{k-2})]$
  $= 3^2M(2^{k-2})$
  $= \ldots = 3^iM(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k.$

- Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Let A(n) be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides 3A(n/2) of these operations needed to compute the three products of n/2-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \qquad A(1) = 1.$$

- Applying the Master Theorem, we obtain $A(n) \in (n^{\log_2 3})$, which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

- **Example:** $M_2 + M_4 = (A_{10} + A_{11}) * B_{00} + A_{11} * (B_{10} - B_{00})$
  $= A_{10}B_{00} + A_{11}B_{10}$

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Analysis of Strassen's Algorithm

Let us evaluate the asymptotic efficiency of this algorithm. If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n × n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

which is smaller than n^3 required by the brute-force algorithm.

- **Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions A(n) made by Strassen's algorithm. To multiply two matrices of order n > 1, the algorithm needs to multiply seven matrices of order n/2 and make 18 additions/subtractions of matrices of size n/2; when n = 1, no additions are made since two numbers are simply multiplied.**
- $T(n) = 7T(n/2) + 18(n/2)^2$ for $n > 1$, $A(1) = 0$.
- According to the Master Theorem, $T(n) \in (n^{\log_2 7})$.