

Techniques for the Design of Algorithms

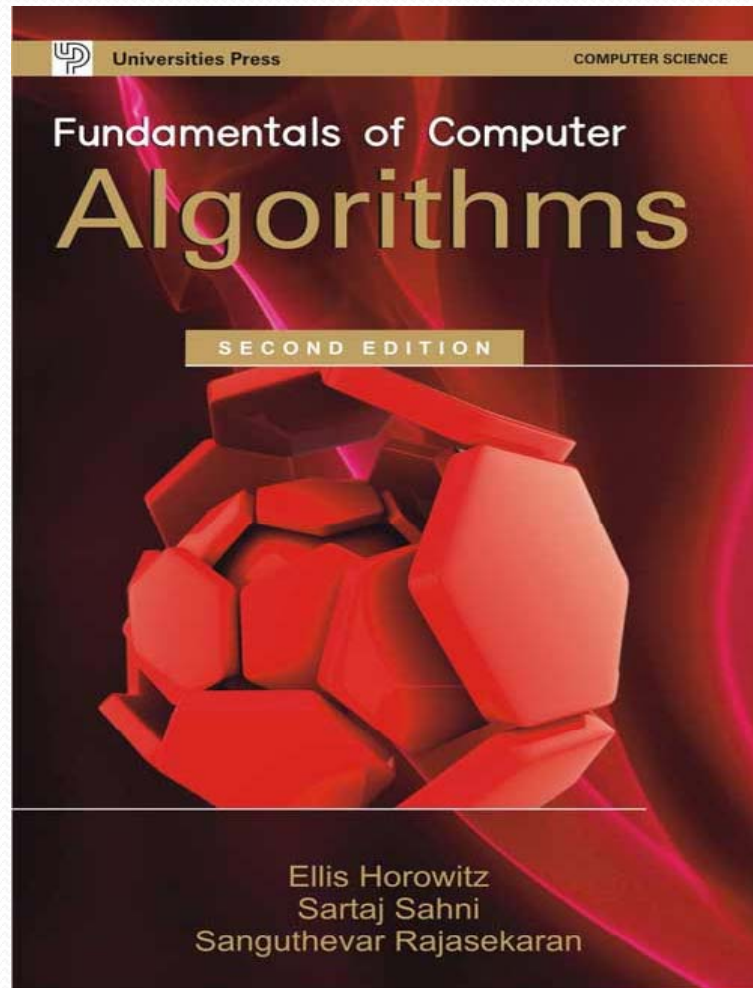
Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

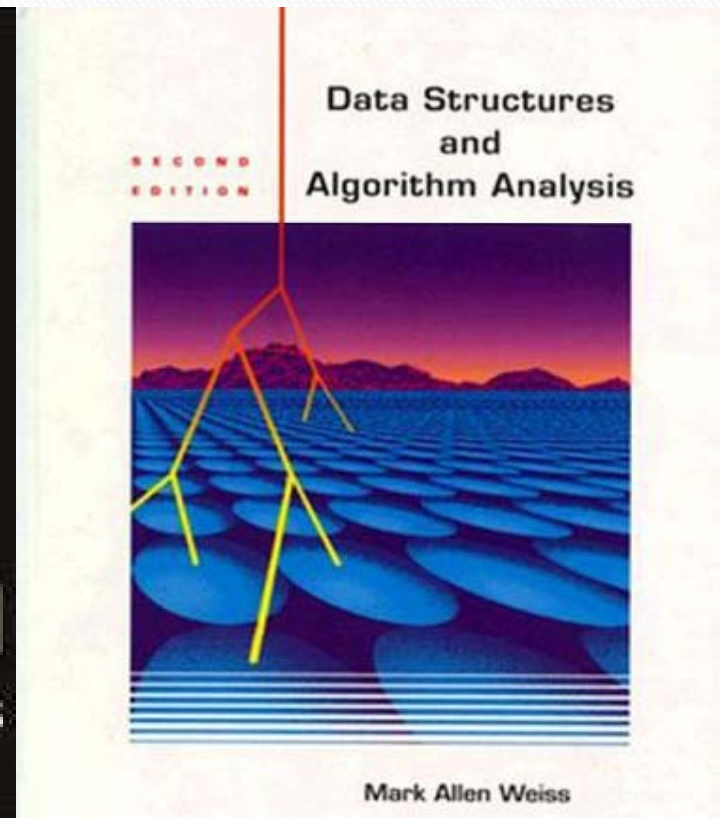
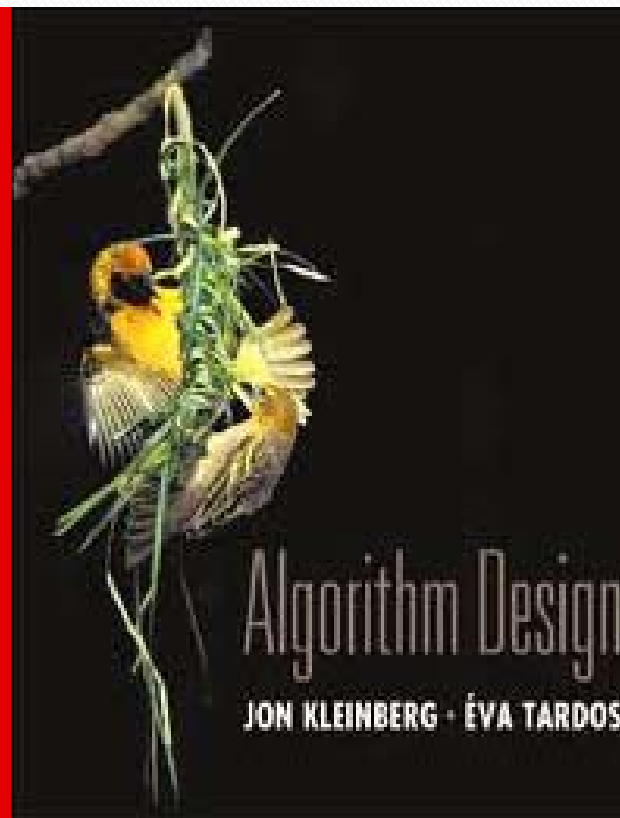
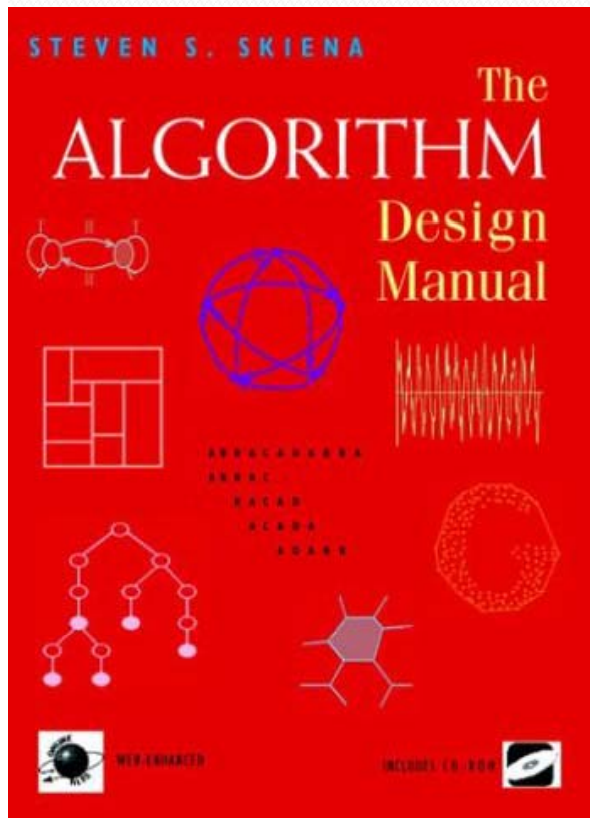
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Text

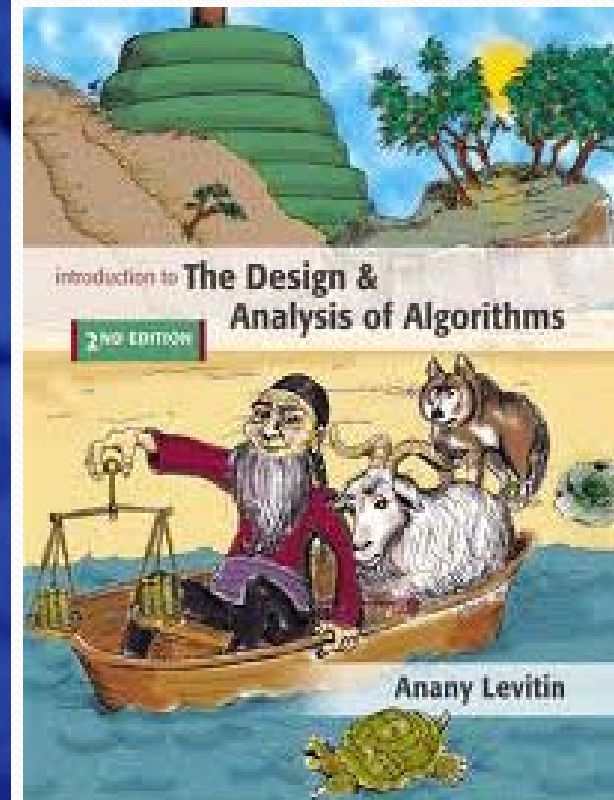
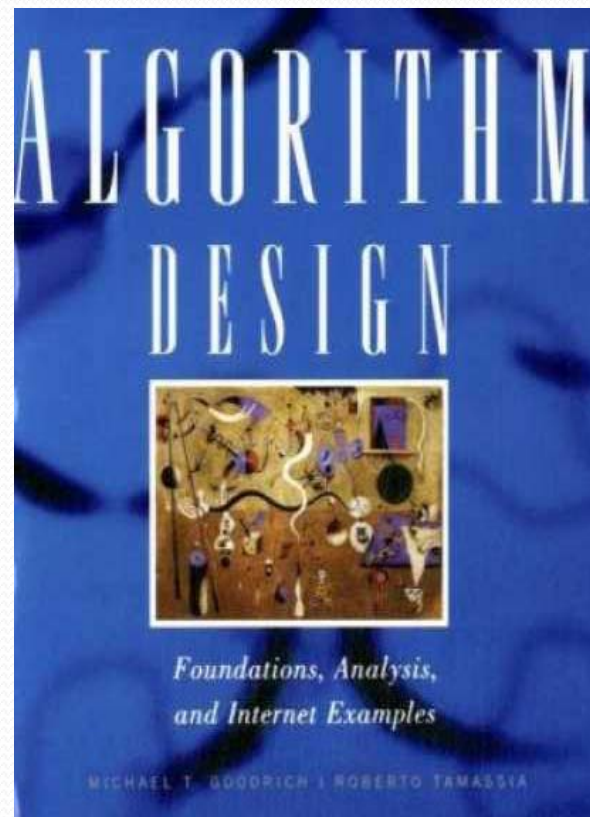
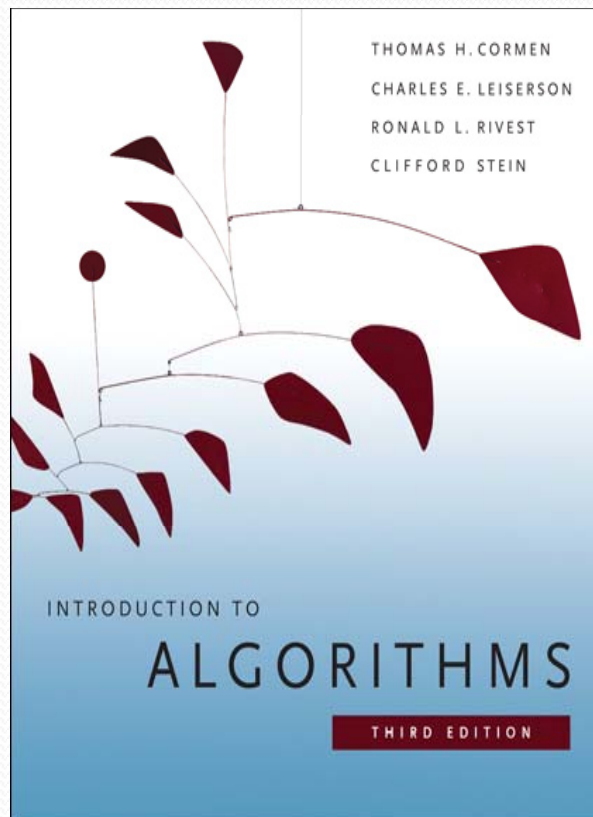


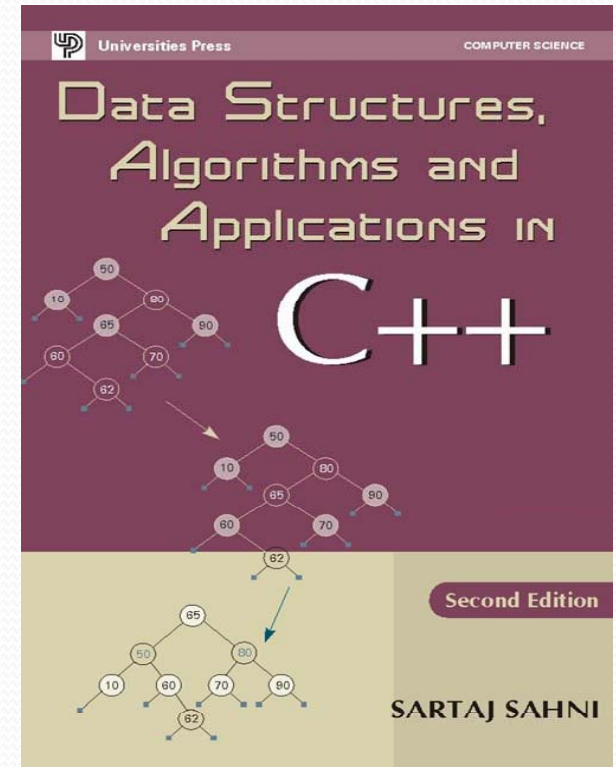
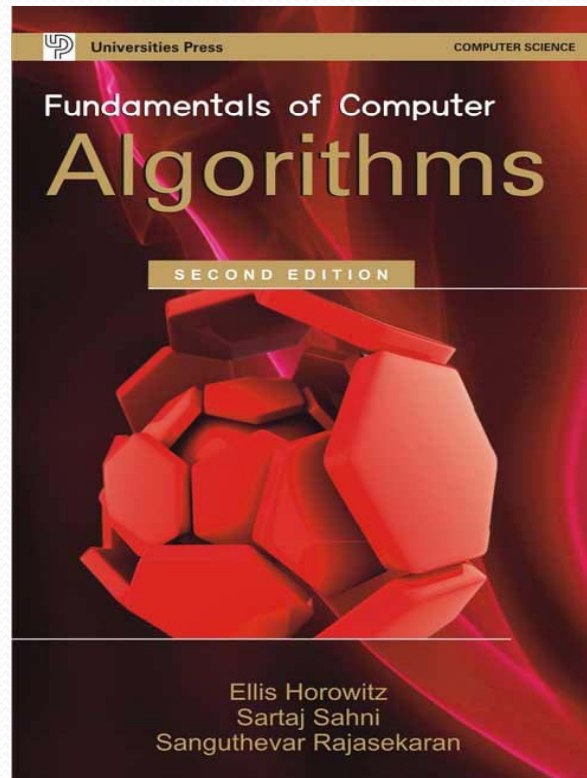
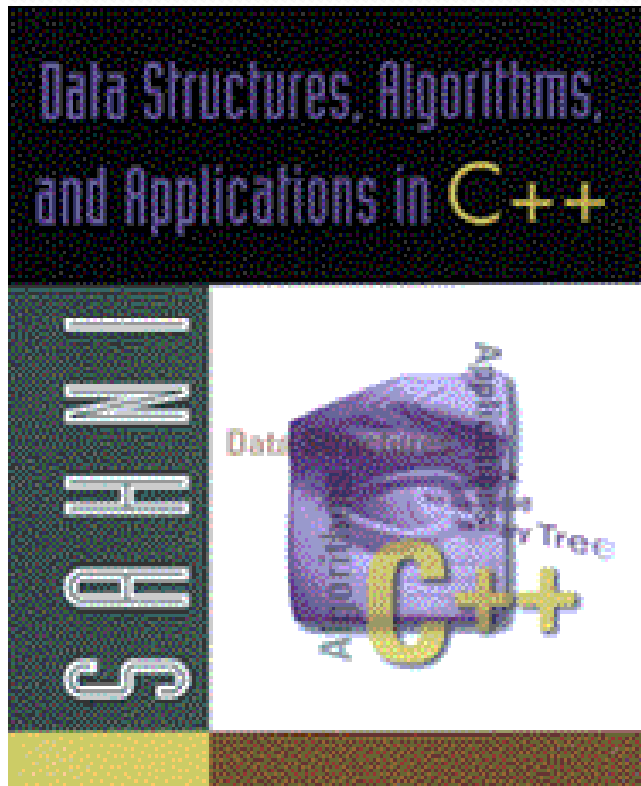
Reference:

Steven S. Skiena, The Algorithm Design Manual, Springer-Verlag London Limited, 2008.

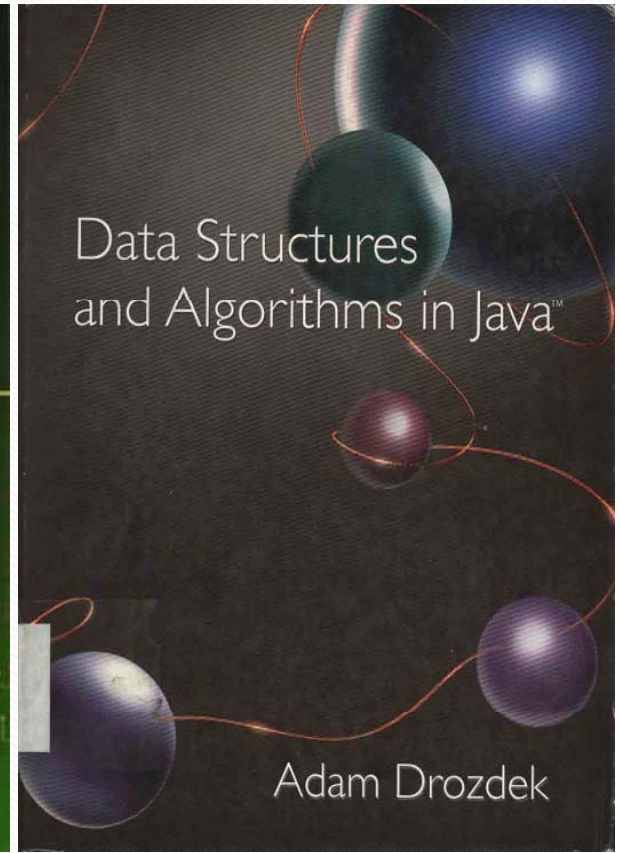
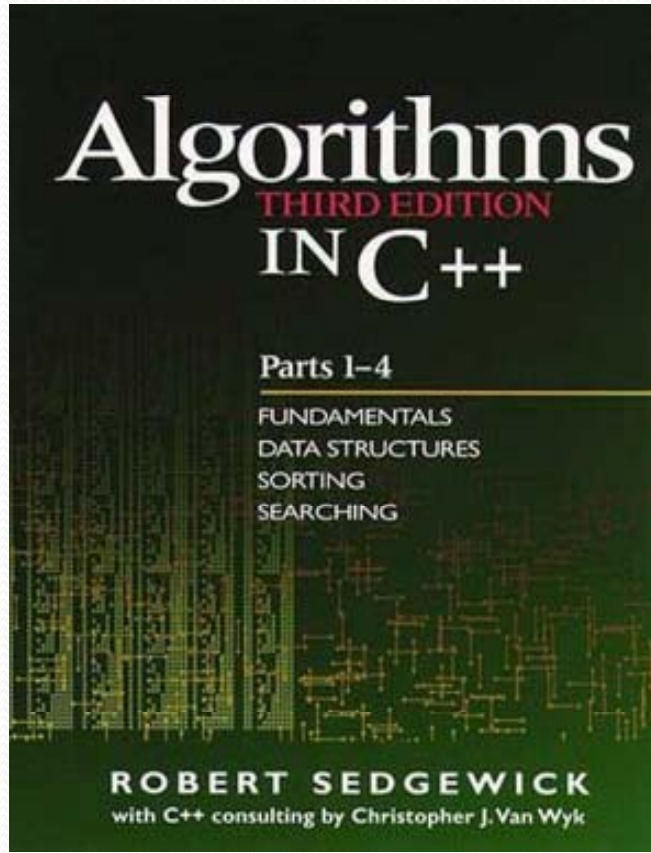
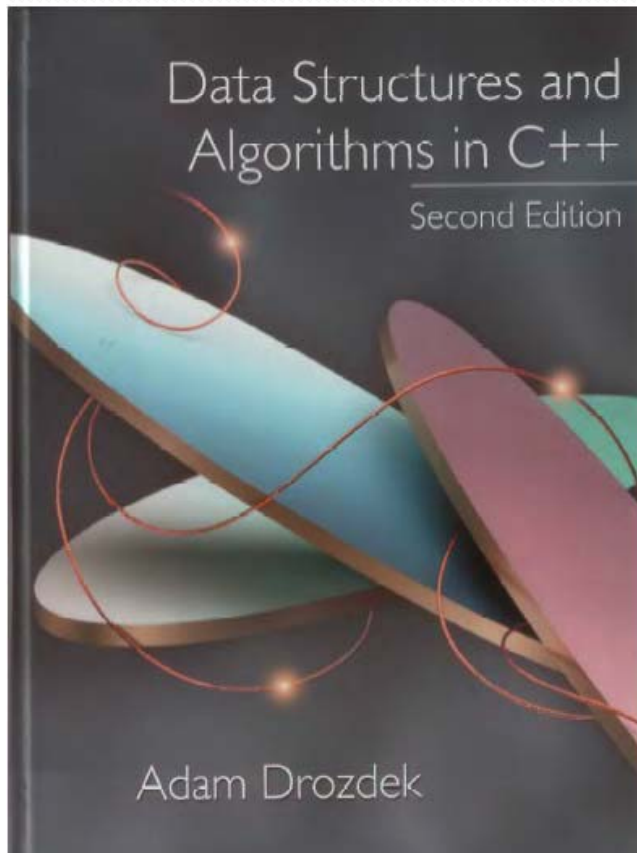


Reference:

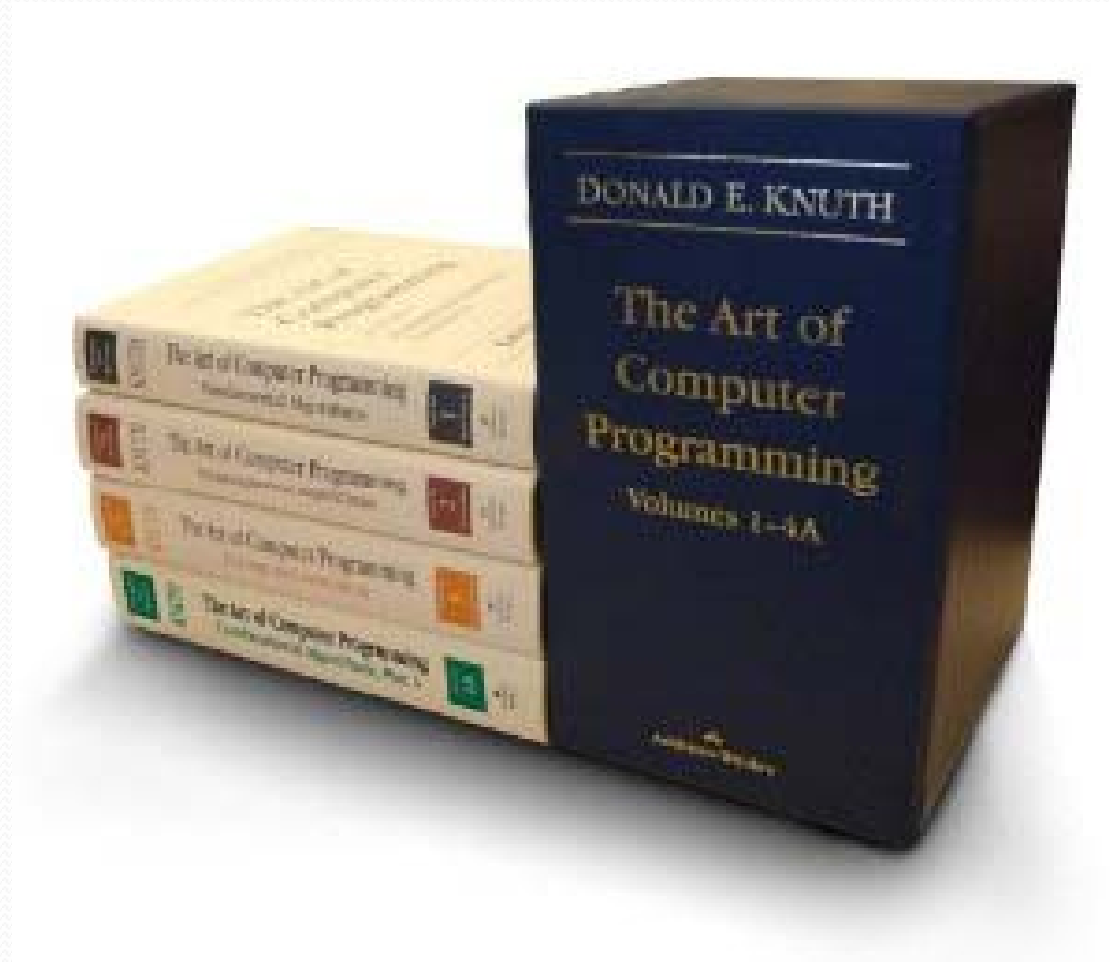




Reference:



Reference: The Art of Computer Programming (TAOCP)



The Art of Computer Programming (TAOCP)

- **Donald E. Knuth**

*Computer Science Department
Gates Building 4B
Stanford University
Stanford, CA 94305-9045 USA.*



- **Donald E. Knuth** is known throughout the world for his pioneering work on algorithms and programming techniques, for his invention of the Tex and Metafont systems for computer typesetting, and for his prolific and influential writing. Professor Emeritus of The Art of Computer Programming at Stanford University, he currently devotes full time to the completion of these fascicles and the seven volumes to which they belong.

The Art of Computer Programming (TAOCP)

- www-cs-faculty.stanford.edu/~uno/books.html
- *Fundamental Algorithms*, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xx+650pp. ISBN 0-201-89683-4
- *Seminumerical Algorithms*, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xiv+762pp. ISBN 0-201-89684-2
- *Sorting and Searching*, Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), xiv+780pp.+foldout. ISBN 0-201-89685-0
- *Combinatorial Algorithms, Part 1* (Upper Saddle River, New Jersey: Addison-Wesley, 2011), xvi+883pp. ISBN 0-201-03804-8

Donald Knuth

- <http://www-cs-faculty.stanford.edu/~uno/>



- **Donald Ervin Knuth** (born 10 January 1938) is an American computer scientist, Professor Emeritus at Stanford University, and winner of the 1974 Turing Award.
- For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to the "art of computer programming" through his well-known books in a continuous series by this title

The Art of Computer Programming (TAOCP)

- **Volume 1:** *Fundamental Algorithms*
- **Volume 2:** *Seminumerical Algorithms*
- **Volume 3:** *Sorting and Searching*
- **Volume 4A:** *Combinatorial Algorithms, Part 1*
Upper Saddle River, New Jersey: Addison-Wesley, 2011
- *Combinatorial Algorithms, Part 2 ; ???*
- “I try to finish *The Art of Computer Programming*(TAOCP), a work that I began in 1962 and that I will need many years to complete”

The Remainder of Volume 4

The remaining subvolumes, currently in preparation, will have the following general outline:

- 7.2.2. Basic backtrack
- 7.2.3. Efficient backtracking
- 7.3. Shortest paths
- 7.4. Graph algorithms
 - 7.4.1. Components and traversal
 - 7.4.2. Special classes of graphs
 - 7.4.3. Expander graphs
 - 7.4.4. Random graphs

The Remainder of Volume 4

- 7.5. Network algorithms
 - 7.5.1. Distinct representatives
 - 7.5.2. The assignment problem
 - 7.5.3. Network flows
 - 7.5.4. Optimum subtrees
 - 7.5.5. Optimum matching
 - 7.5.6. Optimum orderings
- 7.6. Independence theory
 - 7.6.1. Independence structures
 - 7.6.2. Efficient matroid algorithms
- 7.7. Discrete dynamic programming
- 7.8. Branch-and-bound techniques
- 7.9. Herculean tasks (aka NP-hard problems)
- 7.10. Near-optimization
- 8. Recursion

Problem Solving

Problem

- Problem is a question to which we seek an answer?

Selection problem

- Suppose you have a group of N numbers and would like to determine the k^{th} largest. This is known as the *selection problem*.



● Problem SOLVING ?

If we are solving some problem, we are usually looking for some solution, which will be the best among others.

Problem Taxonomy

**Un-decidable
problem**

Do not have algorithms of any known complexity (polynomial or super-polynomial)

Decidable Problem




**Intractable
problem**

Have algorithms with Super polynomial time complexity

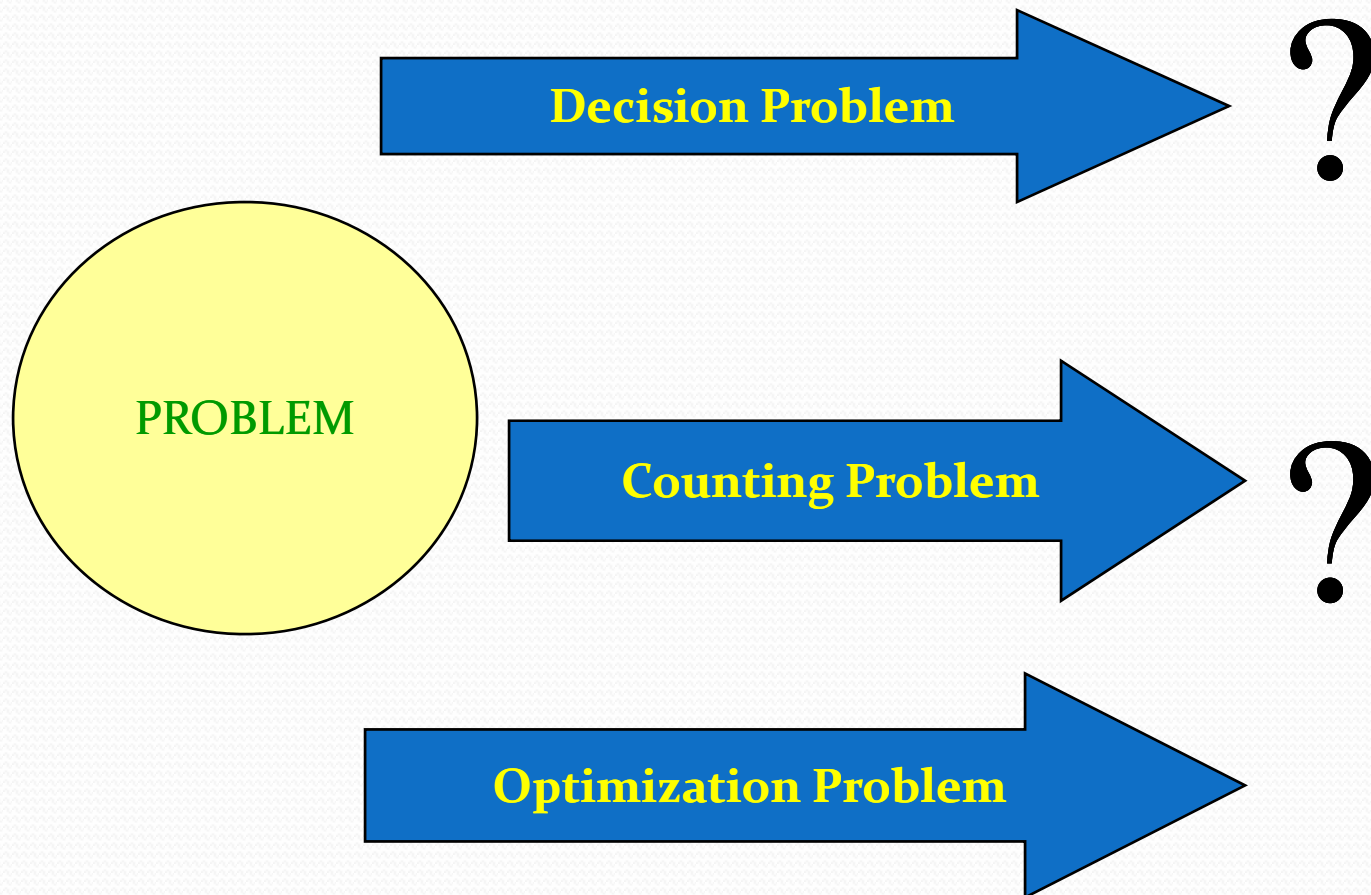
**Tractable
problem**

Have good algorithms with polynomial time complexity (irrespective of the degree)

Different type of decidable Problem

- **Decision Problems:** The class of problems, the output is either yes or no
-  Whether a given number is prime?
- **Counting Problem:** The class of problem, the output is a natural number
-  How many distinct factor are there for a given number
- **Optimization Problem:** The class of problem with some objective function based on the problem instance
-  Finding a minimal spanning tree for a weighted graph

Optimization Problem



Example : 0/1 Knapsack Problem

- Suppose there are n objects and a knapsack or bag with maximum capacity m .
- An object i has a weight w_i and p_i is the profit associated with the object i .
- If a fraction x_i , $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is m , we require the total weight of all chosen objects to be at most m .

The problem can be stated as

$$\text{maximize } \sum_{1 \leq i \leq n} p_i x_i \quad (4.1)$$

$$\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m \quad (4.2)$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \quad (4.3)$$

Problem Solving: Main Steps

1. Problem definition
2. Algorithm design / Algorithm specification
3. Algorithm analysis
4. Implementation
5. Testing
6. [Maintenance]

Problem Definition

- **What is the task to be accomplished?**
 - Calculate the average of the grades for a given student
 - Understand the talks given out by politicians and translate them in English
- **What are the time / space / speed / performance requirements ?**

Measure the difficulty of a problem

- NP-complete ?
- Undecidable ?
- Is the algorithm best ?
 - optimal (algorithm)
- We can also use the number of comparisons to measure a sorting problem.

Measure the goodness of an algorithm

- Time complexity of an algorithm
 - efficient (algorithm)
 - worst-case
 - average-case
 - Amortized
- We can use the number of comparisons to measure a sorting algorithm.

Algorithm

- A clearly specified set of instructions to solve a problem.
- Characteristics:
 - **Input:** Zero or more quantities are externally supplied
 - **Definiteness:** Each instruction is clear and unambiguous
 - **Finiteness:** The algorithm terminates in a finite number of steps.
 - **Effectiveness:** Each instruction must be primitive and feasible
 - **Output:** At least one quantity is produced

What is an algorithm?

- An algorithm is a procedure to accomplish a specific task.
- An algorithm is the idea behind any reasonable computer program.
- To be interesting, an algorithm must solve a general, well-specified *problem*.
- An algorithmic problem is specified by describing the complete set of *instances it must* work on and of its output after running on one of these instances.
- This distinction, between a problem and an instance of a problem, is fundamental.
- **An algorithm is a procedure that takes any of the possible input instances and transforms it to the desired output.**

Algorithms and Programs

- Algorithm: a method or a process followed to solve a problem.
 - A recipe.
- An algorithm takes the input to a problem (function) and transforms it to the output.
 - A mapping of input to output.
- A problem can have many algorithms.

Algorithm Properties

- An algorithm possesses the following properties:
 - It must be correct.
 - It must be composed of a series of concrete steps.
 - There can be no ambiguity as to which step will be performed next.
 - It must be composed of a finite number of steps.
 - It must terminate.
- A **computer program** is an instance, or concrete representation, for an algorithm in some programming language.

Defining Algorithm

- An algorithm is a finite set of instructions that, is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:
 1. Input: Zero or more quantities
 2. Output: At least one
 3. Definiteness : Each instruction is clear and unambiguous
 4. Finiteness: Algorithm terminates in finite number of steps
 5. Effectiveness: Every instruction must be very basic and feasible.

** Horowitz, Sahni @ Rajasekaran : Fundamentals of Algorithms*

Defining algorithm ??

- An algorithm is a *well-ordered* collection of *unambiguous* and *effectively computable* operations that, when executed, *produces a result* and *halts in a finite amount of time*.

Expressing Algorithm

- Reasoning about an algorithm is impossible without a careful description of the sequence of steps to be performed.
- The three most common forms of algorithmic notation are:
 - I. **English,**
 - II. **Pseudocode,**
 - III. **A real programming language.**

Pseudo code

- Structured like a programming language, but ignores many syntactical details (like \$ and ;)
- Complex operations can be written in natural language
- Still, we need to agree on some standard operations...
- We are following : Pseudo code as suggested in text “*Horowitz, Sahni @ Rajasekaran : Fundamentals of Algorithms*”
- Two good options for typesetting algorithms in LaTeX with packages, **algorithms** and **algorithm2e**
- www.ctan.org/tex-archive/macros/latex/contrib/algorithms/

Latex algorithm

Algorithm 1 General Task Allocation Algorithm

Require: Task Matrix

Ensure: Utilization Matrix

```
1:  $time, \tau \leftarrow 1$ 
2: Initialize Utilization Matrix,  $U^* \leftarrow \phi$ .
3:  $R^* \leftarrow \phi$ .
4: while  $taskq \neq \phi$  do
5:    $jq = \text{Get jobs from main queue}(taskq)$  where arrival
     time  $\leq \tau$ .
6:   while  $jq \neq \phi$  do
7:      $j \leftarrow \text{TaskChoosingPolicy}()$ 
8:      $i \leftarrow \text{ResourceChoosingPolicy}()$ 
9:     if  $i \neq \text{Null}$  then
10:      Assign task  $t_j$  to  $R_i$ 
11:       $U_{(\tau,i)} \leftarrow U_{(\tau,i)} + \text{utilization}(t_j, i)$ .
12:      Remove task  $t_j$  from  $taskq$  and  $jq$ .
13:     else
14:      Remove task  $t_j$  from  $jq$ .
15:     end if
16:   end while
17:    $\tau \leftarrow \tau + 1$ .
18: end while
19: return  $U$ .
```

Problem specifications

- Problem specifications have two parts:
 - (1) the set of allowed input instances,
 - (2) the required properties of the algorithm's output.
- It is impossible to prove the correctness of an algorithm for a fuzzily-stated problem. Put another way, ask the wrong problem and you will get the wrong answer.

Specifying the output requirements of a problem.

- There are two common traps in specifying the output requirements of a problem.
- **One is asking an ill-defined question.** Asking for the *best route between two places* on a map is a silly question unless you define what *best means*.
- Do you mean **the shortest route in total distance**, or the **fastest route**, or **the one minimizing the number of turns**?
- **The second trap is creating compound goals.** The three path-planning criteria mentioned above are all well-defined goals that lead to correct, efficient optimization algorithms.
- However, you must pick a single criteria. A goal like *Find the shortest path from a to b that doesn't use more than twice as many turns as necessary* is perfectly well defined, but complicated to reason and solve.

Robot Tour Optimization

Let's consider a problem that arises often in manufacturing, transportation, and testing applications. Suppose we are given a robot arm equipped with a tool, say a soldering iron. In **manufacturing circuit boards**, all the chips and other components must be fastened onto the substrate. More specifically, each chip has a set of contact points (or wires) that must be soldered to the board. To program the robot arm for this job, we must **first** construct an ordering of the contact points so the robot visits (and solders) the first contact point, then the second point, third, and so forth until the job is done. The robot arm then proceeds back to the first contact point to prepare for the next board, thus turning the tool-path into a closed tour, or cycle.



Robot Tour Optimization

Robots are expensive devices, so we want the tour that minimizes the time it takes to assemble the circuit board. A reasonable assumption is that the robot arm moves with fixed speed, so the time to travel between two points is proportional to their distance. In short, we must solve the following algorithm problem:

- **Problem:** *Robot Tour Optimization*
- **Input:** *A set S of n points in the plane.*
- **Output:** *What is the shortest cycle tour that visits each point in the set S ?*
- **You are given the job of programming the robot arm. Suggest an algorithm to solve this problem?**

Robot Tour Optimization

- Several algorithms might come to mind to solve this problem. Perhaps the most popular idea is the *nearest-neighbor heuristic*.
- *Starting from some point p_0 , we walk first to its nearest neighbor p_1 . From p_1 , we walk to its nearest unvisited neighbor, thus excluding only p_0 as a candidate.*
- *We now repeat this process until we run out of unvisited points, after which we return to p_0 to close off the tour.*
- *Written in pseudo-code, the nearest-neighbor heuristic looks like this: [next slide]*

Pseudo-code, the nearest-neighbor heuristic

1. **NearestNeighbor**(P)
2. Pick and visit an initial point p_0 from P
3. $p = p_0$
4. $i = 0$
5. While there are still unvisited points
6. $i = i + 1$
7. Select p_i to be the closest unvisited point to p_{i-1}
8. Visit p_i
9. Return to p_0 from p_{n-1}

The background of the slide is a solid blue color. At the top, there are several wavy, horizontal lines in shades of blue and cyan, creating a layered, water-like effect. The lines are smooth and flow from left to right, with some lines being slightly more prominent than others.

Algorithm Paradigm

Algorithm Paradigm

Algorithm Design Paradigms: General approaches to the construction of efficient solutions to problems

- **Interest of different paradigm**
 - They provide templates suited to solving a broad range of diverse problems.
 - They can be translated into common control and data structures provided by most high-level languages.
 - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Brute force

- **Brute force** is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem.
- **Examples of brute force algorithms are:**
 - ❑ Computing a^n ($a > 0$, n a nonnegative integer) by multiplying $a * a * ... * a$
 - ❑ Computing $n!$
 - ❑ Selection sort
 - ❑ Bubble sort
 - ❑ Sequential search
 - ❑ Exhaustive search: Traveling Salesman Problem, Knapsack problem.

Algorithm Paradigm

- Divide and Conquer
- Greedy
- Dynamic Programming
- Backtracking
- Branch and Bound
- Approximation algorithms
 - Randomized algorithms
- Soft Computing Techniques

Divide-and-Conquer

- Given an instance of the problem to be solved, split this into several smaller sub-instances (*of the same problem*), independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.
- With the divide-and-conquer method the size of the problem instance is reduced by a factor (e.g. half the input size), while with the decrease-and-conquer method the size is reduced by a constant.
- **Examples of divide-and-conquer algorithms:**
 - ❑ Computing a^n ($a > 0$, n a nonnegative integer) by recursion
 - ❑ Binary search in a sorted array (recursion)
 - ❑ Mergesort algorithm, Quicksort algorithm (recursion)
 - ❑ The algorithm for solving the fake coin problem (recursion)

Examples of decrease-and-conquer algorithms:

- ☐ Insertion sort
- ☐ Topological sorting
- ☐ Binary Tree traversals: inorder, preorder and postorder (recursion)
- ☐ Computing the length of the longest path in a binary tree (recursion)
- ☐ Computing Fibonacci numbers (recursion)
- ☐ Reversing a queue (recursion)
- ☐ Warshall's algorithm (recursion)

Issues in algorithm design (Divide and Conquer)

The issues here are two:

- ☐ How to solve the sub-instance
- ☐ How to combine the obtained solutions
- The answer to the second question depends on the nature of the problem.
- In most cases the answer to the first question is: using the same method. Here another very important issue arises: when to stop decreasing the problem instance, i.e. what is the minimal instance of the given problem and how to solve it.
- When we use recursion, the solution of the minimal instance is called “terminating condition”

Transform-and-Conquer

- These methods work as two-stage procedures. First, the problem is modified to be more amenable to solution. In the second stage the problem is solved.
- Types of problem modifications
 - Problem simplification e.g. presorting

Example: consider the problem of finding the two closest numbers in an array of numbers.

- Brute force solution: $O(n^2)$
 - Transform and conquer solution: $O(n \log n)$
 - Presort the array – $O(n \log n)$
 - Scan the array comparing the differences - $O(n)$
 - Change in the representation

Example: AVL trees guarantee $O(n \log n)$ search time
 - Problem reduction

Example: least common multiple
- $\text{lcm}(m,n) = (m*n) / \text{gcd}(m,n)$

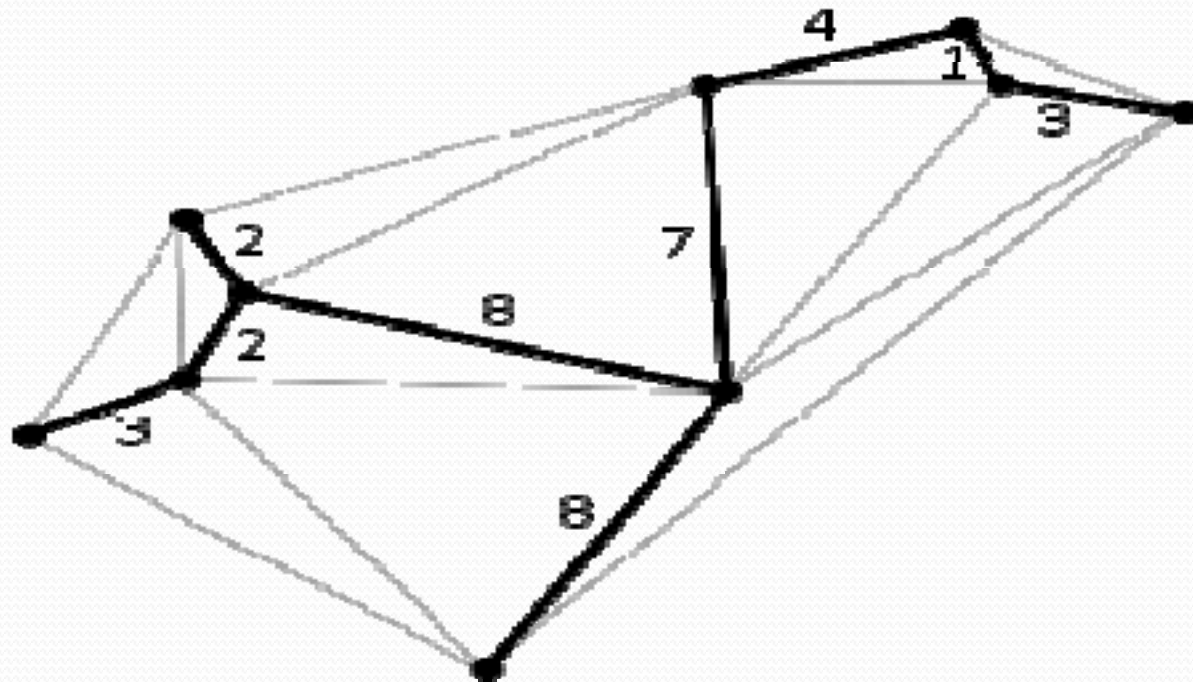
Greedy Algorithms "take what you can get now" strategy

- The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far.
- At each step the choice must be locally optimal – this is the central point of this technique.
- **Examples:**
 - ❑ Minimal spanning tree
 - ❑ Shortest distance in graphs
 - ❑ Greedy algorithm for the Knapsack problem
 - ❑ The coin exchange problem
 - ❑ Huffman trees for optimal encoding
- Greedy techniques are mainly used to solve optimization problems. They do not always give the best solution.
 1. **Sub-set sum paradigm**
 2. **Ordering Paradigm**

Spanning tree

Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.



Dynamic Programming


- One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the **same computations being performed repeatedly** since *identical* sub-instances may arise.
- The idea behind *dynamic programming* is to avoid this pathology by obviating the requirement to calculate the same quantity twice.
- The method usually accomplishes this by maintaining a *table of sub-instance results*.
- Dynamic Programming is a **Bottom-Up Technique** in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.

Dynamic Programming

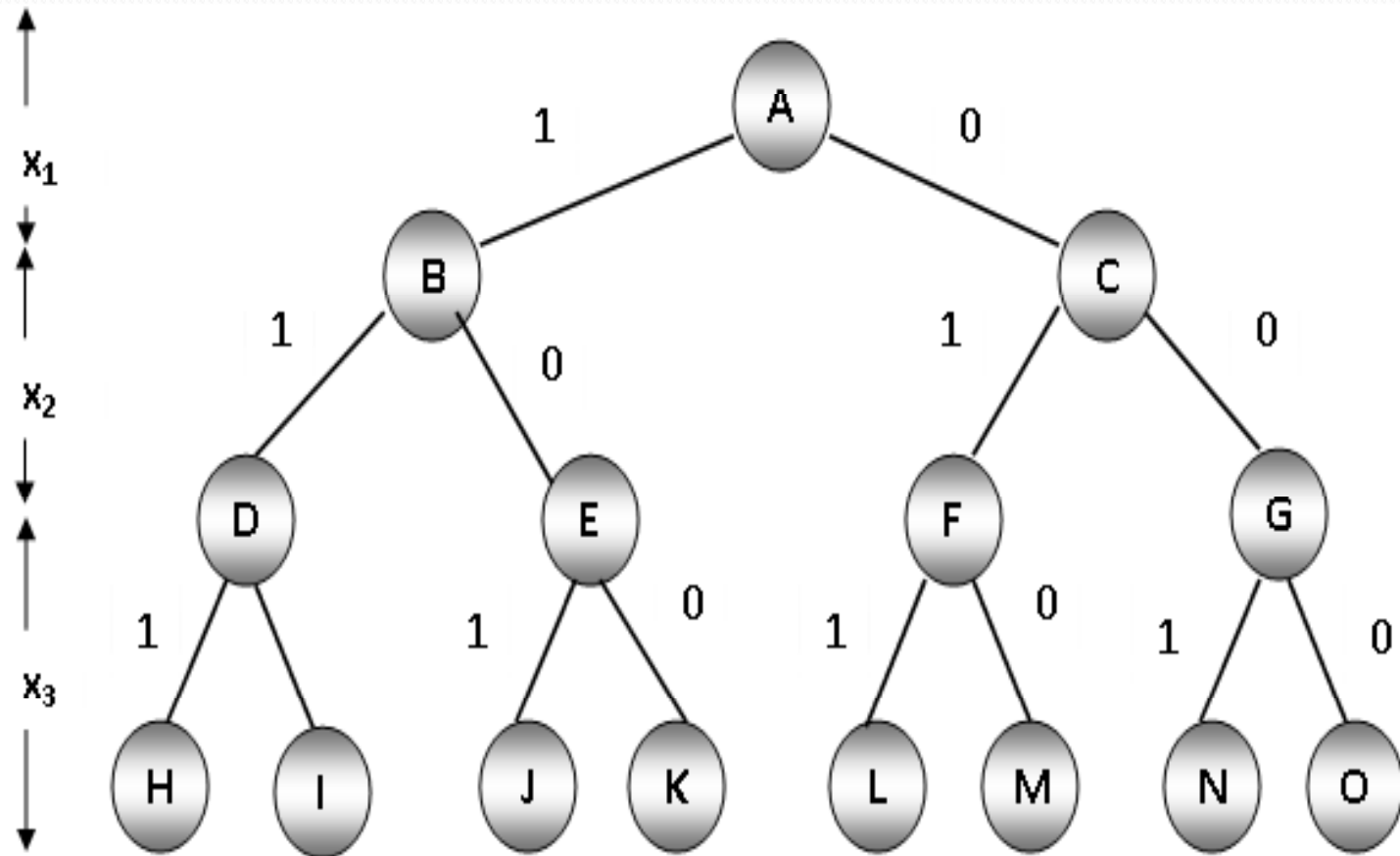
- In contrast, Divide-and-Conquer is a **Top-Down Technique** which *logically* progresses from the initial instance down to the smallest sub-instance via intermediate sub-instances.
- **Examples:**
 - ❑ Fibonacci numbers computed by iteration.
 - ❑ Warshall's algorithm implemented by iterations

Backtracking and branch-and-bound: generate and test methods

- The method is used for state-space search problems. State-space search problems are problems, where the problem representation consists of:
 - initial state
 - goal state(s)
 - a set of intermediate states
 - a set of operators that transform one state into another. Each operator has preconditions and postconditions.
 - a cost function – evaluates the cost of the operations (optional)
 - a utility function – evaluates how close is a given state to the goal state (optional)

- 
- The solving process solution is based on the construction of a **state-space tree**, whose nodes represent states, the root represents the initial state, and one or more leaves are goal states. Each edge is labeled with some operator.
 - If a node **b** is obtained from a node **a** as a result of applying the operator **O**, then **b** is a child of **a** and the edge from **a** to **b** is labeled with **O**.
 - The solution is obtained by searching the tree until a goal state is found.

Solution space 0/1 knapsack



Backtracking

- **Backtracking uses depth-first search** usually without cost function. The main algorithm is as follows:
 1. Store the initial state in a stack
 2. While the stack is not empty, do:
 - Read a node from the stack.
 - While there are available operators do:
 - Apply an operator to generate a child
 - If the child is a goal state – stop
 - If it is a new state, push the child into the stack
- The utility function is used to tell how close is a given state to the goal state and whether a given state may be considered a goal state.
- If no children can be generated from a given node, then we **backtrack** – read the next node from the stack.

Problem solving using state-space search techniques:

- **Example:** The following problems can be solved using state-space search techniques:
 - A farmer has to move a goat, a cabbage and a wolf from one side of a river to the other side using a small boat. The boat can carry only the farmer and one more object (either the goat, or the cabbage, or the wolf). If the farmer leaves the goat with the wolf alone, the wolf would kill the goat. If the goat is alone with the cabbage, it will eat the cabbage. How can the farmer move all his property safely to the other side of the river?"
 - You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

Problem solving using state-space search techniques:

- We have to decide:
 - representation of the problem state, initial and final states
 - representation of the actions available in the problem, in terms of how they change the problem state.
- **Example:**
 - Problem state: pair of numbers (X,Y): X - water in jar 1 called A, Y - water in jar 2, called B.
Initial state: (0,0),
Final state: (2,_) here "_" means "any quantity"
 - Available actions (operators): *next slide*

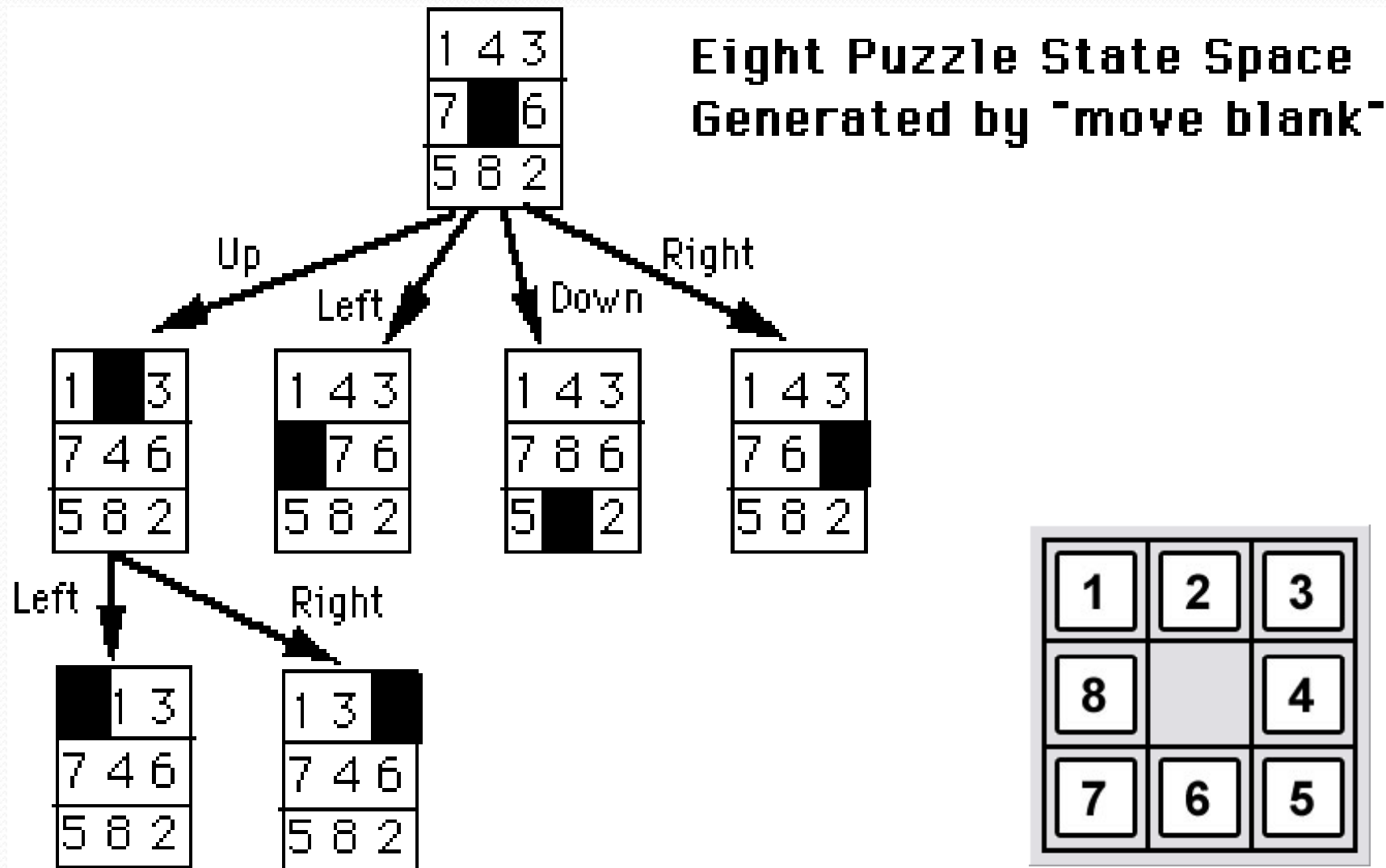
Problem solving using state-space search techniques:

Description	Pre-conditions on (X,Y)	Action (Post-conditions)
o1. Fill A	$X < 4$	(4, Y)
o2. Fill B	$Y < 3$	(X, 3)
o3. Empty A	$X > 0$	(0, Y)
o4. Empty B	$Y > 0$	(X, 0)
o5. Pour A into B	<ul style="list-style-type: none">• $X > 3 - Y$• $X \leq 3 - Y$	<ul style="list-style-type: none">(X + Y - 3 , 3)(0, X + Y)
o6. Pour B into A	<ul style="list-style-type: none">• $Y > 4 - X$• $Y \leq 4 - X$	<ul style="list-style-type: none">(4, X + Y - 4)(X + Y, 0)

Branch-and-bound

- Branch and bound is used when we can evaluate each node using the cost and utility functions. At each step we choose the best node to proceed further. Branch-and bound algorithms are implemented using a priority queue. The state-space tree is built in a **breadth-first** manner.
- **Example:** The 8-puzzle problem. The cost function is the number of moves. The utility function evaluates how close is a given state of the puzzle to the goal state, e.g. counting how many tiles are not in place.

Eight puzzle problem



Conclusion

- The basic question here is: How to choose the approach?
- First, by understanding the problem, and second, by knowing various problems and how they are solved using different approaches.

The background of the slide is a solid blue color. At the top, there are several wavy, horizontal lines in shades of blue and cyan, creating a layered, wave-like effect. The text is centered in the middle of the slide.

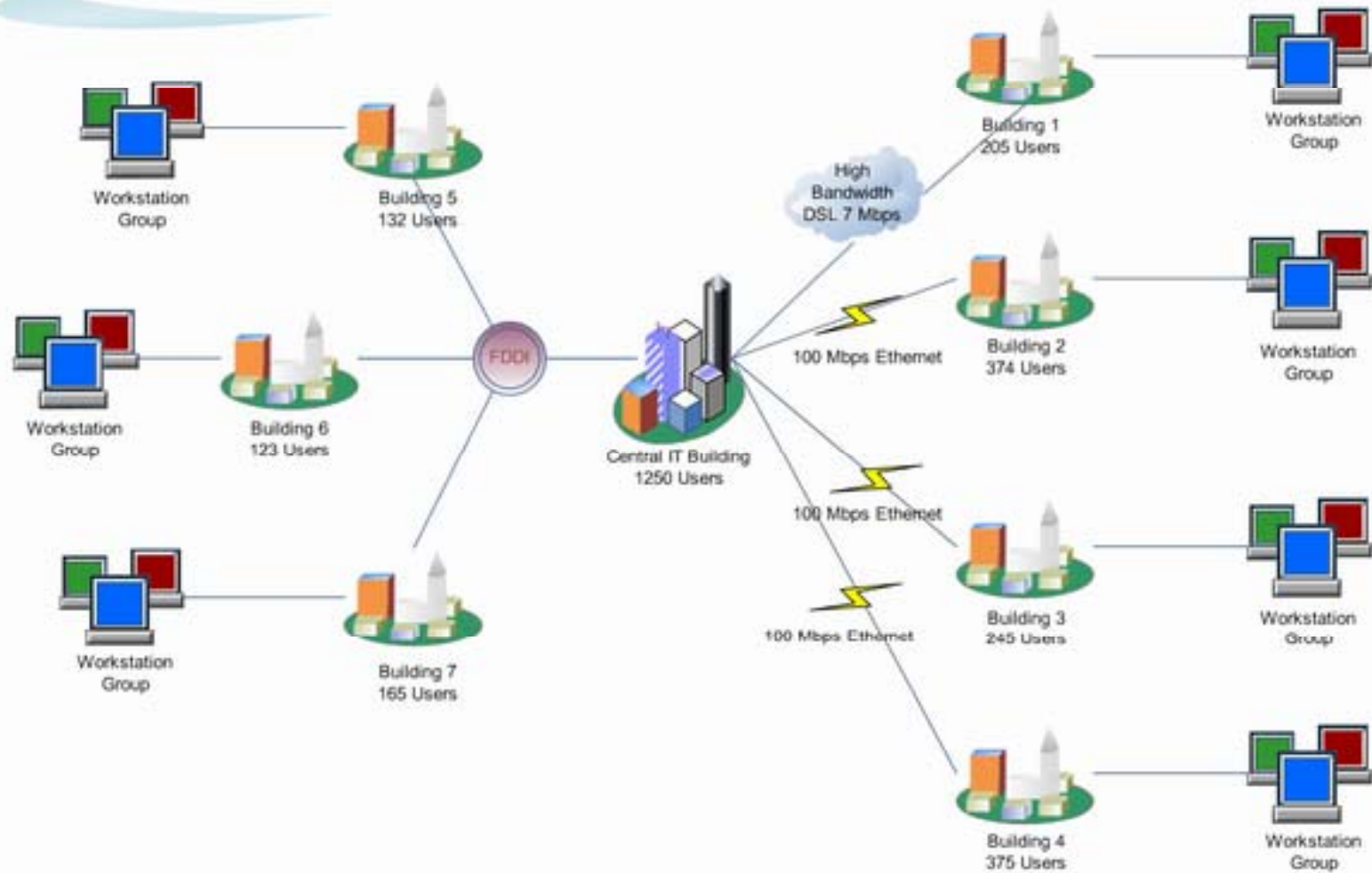
MODELING : Algorithm Design

Modeling

- **Modeling** is the art of formulating your application in terms of precisely described, well-understood problems.
- Proper modeling is the key to applying **algorithmic design techniques** to any **real-world problem**.
- Real-world applications involve real-world objects.
- Most algorithms, however, are designed to work on rigorously defined abstract structures such as permutations, graphs, and sets.
- You must first describe your problem abstractly, in terms of fundamental structures and properties.

Campus Executive Overview Guideline

Sunday, Jan. 1, 2006



Rules for Algorithm design

- The secret to successful algorithm design, and problem solving in general, is to make sure *you ask the right questions*.
- Below, I give a possible series of questions for you to ask yourself as you try to solve difficult algorithm design problems

1. Do I really understand the problem?

- (a) What exactly does the input consist of?
- (b) What exactly are the desired results or output?
- (c) Can I construct some examples small enough to solve by hand? What happens when I solve them?
- (d) Are you trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Might your problem be formulated in more than one way? Which formulation seems easiest?



2. Can I find a simple algorithm for the problem?

(a) Can I find the solve my problem exactly by searching all subsets or arrangements and picking the best one?

- I. If so, why am I sure that this algorithm always gives the correct answer?
- II. How do I measure the quality of a solution once I construct it?
- III. Does this simple, slow solution run in polynomial or exponential time?
- IV. If I can't find a slow, *guaranteed* correct algorithm, am I sure that my problem is well defined enough to permit a solution?



(b) Can I solve my problem by repeatedly trying some heuristic rule, like picking the biggest item first? The smallest item first? A random item first?

- I.** If so, on what types of inputs does this heuristic rule work well? Do these correspond to the types of inputs that might arise in the application?
- II.** On what types of inputs does this heuristic rule work badly? If no such examples can be found, can I show that in fact it always works well?
- III.** How fast does my heuristic rule come up with an answer?



3. Are there special cases of this problem I know how to solve exactly?

- (a) Can I solve it efficiently when I ignore some of the input parameters?
- (b) What happens when I set some of the input parameters to trivial values, such as 0 or 1?
- (c) Can I simplify the problem to create a problem I can solve efficiently? How simple do I have to make it?
- (d) If I can solve a certain special case, why can't this be generalized to a wider class of inputs?



4. Which of the standard algorithm design paradigms seem most relevant to the problem?

- (a) Is there a set of items which can be sorted by size or some key? Does this sorted order make it easier to find what might be the answer?
- (b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search or a partition of the elements into big and small, or left and right? If so, does this suggest a divide-and-conquer algorithm?
- (c) Are there certain operations being repeatedly done on the same data, such as searching it for some element, or finding the largest/smallest remaining element? If so, can I use a data structure of speed up these queries, like hash tables or a heap/priority queue?

Millennium Problems ...

The Clay Mathematics Institute (CMI)

<http://www.claymath.org/>

Millennium Problems ...

- **Yang–Mills and Mass Gap**
- Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.
- **Riemann Hypothesis**
- The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part $1/2$.
- **P vs NP Problem**
- If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Millennium Problems

- **Navier–Stokes Equation**
- This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.
- **Hodge Conjecture**
- The answer to this conjecture determines how much of the topology of the solution set of a system of algebraic equations can be defined in terms of further algebraic equations. The Hodge conjecture is known in certain special cases, e.g., when the solution set has dimension less than four. But in dimension four it is unknown.

Millennium Problems ...

- **Poincaré Conjecture**
- In 1904 the French mathematician Henri Poincaré asked if the three dimensional sphere is characterized as the unique simply connected three manifold. This question, the Poincaré conjecture, was a special case of Thurston's geometrization conjecture. Perelman's proof tells us that every three manifold is built from a set of standard pieces, each with one of eight well-understood geometries.
- **Birch and Swinnerton-Dyer Conjecture**
- Supported by much experimental evidence, this conjecture relates the number of points on an elliptic curve mod p to the rank of the group of rational points. Elliptic curves, defined by cubic equations in two variables, are fundamental mathematical objects that arise in many areas: Wiles' proof of the Fermat Conjecture, factorization of numbers into primes, and cryptography, to name three.



Assignment

- Suppose you need to search a given ordered array of n integers for a specific value. The entries are sorted in non-decreasing order.
 1. Give a simple algorithm that has a worst-case complexity of $\Theta(n)$
 2. We can do better by using the “binary search” strategy. Explain what it is and give an algorithm based on it. What is the worst-case complexity of this algorithm?
- **Write algorithms in pseudocode notation**

Sample problems

- Write the most efficient algorithm you can think of for the following: Find the k -th item in an n -node doubly-linked list. What is the running time in terms of big- θ ?
- Write the most efficient algorithm you can think of for the following: Given a set of p points, find the pair closest to each other. Be sure and try a divide-and-conquer approach, as well as others. What is the running time in terms of big- θ ?
- Design and implement an algorithm that finds all the duplicates in a random sequence of integers.
- Reconsider the three algorithms you just designed given the following change: the input has increased by 1,000,000,000 times.

Sample problems

- You have two arrays of integers. Within each array, there are no duplicate values but there may be values in common with the other array. Assume the arrays represent two different sets. Define an $O(n \log n)$ algorithm that outputs a third array representing the union of the two sets. The value " n " in $O(n \log n)$ is the sum of the sizes of the two input arrays.
- You are given an increasing sequence of numbers u_1, u_2, \dots, u_m , and a decreasing series of numbers d_1, d_2, \dots, d_n . You are given one more number C and asked to determine if C can be written as the sum of one u_i and one d_j . There is an obvious brute force method, just comparing all the sums to C , but there is a much more efficient solution. Define an algorithm that works in linear time.



Sample problems

- Design and implement a dynamic programming algorithm to solve the **change counting problem**. Your algorithm should always find the optimal solution--even when the greedy algorithm fails.
- Suppose a certain algorithm has been empirically shown to sort 100,000 integers or 100,000 floating-point numbers in two seconds. Would we expect the same algorithm to sort 100,000 strings in two seconds? Explain.

Thanks for Your Attention!

