Computers are general purpose machines for solving problems and so algorithms are important in computer programming. To make a computer useful in problem solving, we must give the problem as well as the technique to solve the problem to it. So by programming the computer with various algorithms to solve problems, the computers can be made "intelligent". Computers are well-suited for solving tedious problems because of their speed and accuracy.

Much of the study of computer science is dedicated to finding efficient algorithms and representing them so that computers will understand that. In our study about algorithms, we will learn what defines an algorithm, Algorithm design techniques, well-known Algorithms and their advantages.

**Algorithms - Definition**

An algorithm is a well-ordered sequence of unambiguous and effectively computable instructions that when executed produces a result and halts in a finite amount of time

**Characteristics of Algorithms**  
a. Algorithms are well-ordered  
b. Algorithms have unambiguous instructions  
c. Algorithms have effectively computable instructions  
d. Algorithms produce a result  
e. Algorithms halt in a finite amount of time

**Algorithm Design Techniques**

Algorithm design techniques are common approaches to the construction of efficient solutions to problems. Such methods are of interest because:

1. They provide templates suited to solve a wide range of diverse problems.  
2. They can be converted into common control and data structures provided by most of the high-level languages.  
3. The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

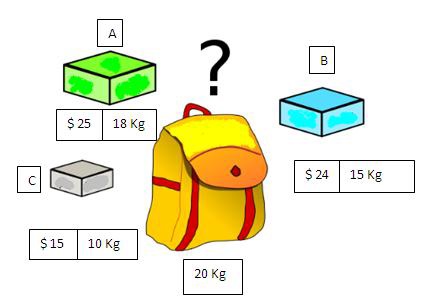
Following are the most important design techniques to solve different types of the problems:  
1. Greedy algorithms  
2. Divide-and-conquer  
3. Dynamic programming  
4. Backtracking and branch-and-bound

**Greedy Algorithm**  
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 flawless and ideal, and this flawlessness is achieved by selection criteria of inputs.

Example: Greedy algorithm for the kit bag problem

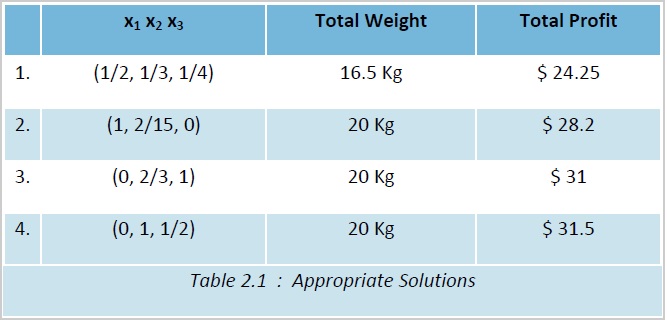
**Kit bag problem**  
We are given n objects and a kit bag. Object i has a weight wi and the kit bag has a capacity m. If a fraction of xi, 0 <= xi <= 1, of object i is placed into the kit bag, then a profit of pixi is earned. The objective is to fill the kit bag in a way that maximizes the total profit earned and the total weight of all chosen objects to be at most m (capacity of the kit bag).

Let us consider an instance of the above problem.  
There are three objects A, B, and C and the capacity of the kit bag is 20. Profits and weights of the objects A, B, and C are given like this, (p1, p2, p3) = (25, 24, 15), (w1, w2, w3)= (18, 15, 10).



Now we need to fill the kit bag with these objects in such a way that we will gain maximum profit.  
Let us try to find the solution, without applying Greedy algorithm:  
First, we need to find all appropriate solutions (Total weight of the objects <= capacity of the kit bag i.e., 20)

**Appropriate Solutions**

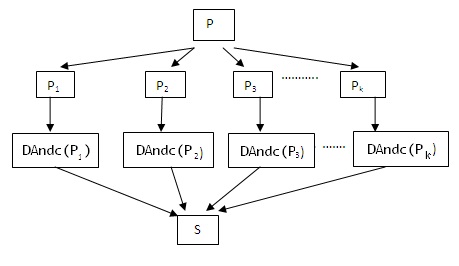
****

We got totally four appropriate solutions. In the first appropriate solution, we have taken fraction i.e., half of the object A, one third of B and one fourth of C.

So total weight of the three objects taken into the kit bag is  
18 \* 1/2 + 15 \* 1/3 + 10 \* 1/4 = 9 + 5 + 2.5 = 16.5Kg,  
which is less than the capacity of the kit bag (20), where 18, 15 and 10 are the weights of the objects A,B and C respectively.  
Total profit gained is 25 \* 1/2 + 24 \* 1/3 + 15 \* 1/4 = 12.5 + 8 + 3.75 = $24.25, where 25, 24 and 15 are the profits of the objects A, B and C respectively.

Similarly, the profits earned in the remaining appropriate solutions are obtained like this.  
2nd solution: 25 \* 1 + 24 \* 2/15 + 15 \* 0 (object C is not taken) = 25 + 3.2 + 0 = $28.2  
3rd solution: 25 \* 0(object A is not taken) + 24 \* 2/3 + 15 \* 1 = 0 + 16 + 15 = $31  
4th solution: 25 \* 0(object A is not taken) + 24 \* 1 + 15 \* 1/2 = 0 + 24 + 7.5 = $31.5  
It is clear that the 4th one is a best solution among all these solutions, since we are attaining maximum profit using this solution. Using this approach we can get the best solution (without applying a Greedy algorithm), but it is time consuming. The same can be achieved very easily, using the Greedy technique.

**Divide and Conquer**  
The strategy of D and C (Divide and Conquer) technique to solve a problem is as follows.  
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 by recursive applications of D and C and then combine the sub-instance solutions so as to get a solution for the original instance



**Example**  
Binary search problem: Let A={ a1, a2, a3, ..., an} be a list of elements sorted in increasing order i.e., a1 <= a2 <= ... <= an. We need to determine whether a given element x is present in the list or not. If it exists, return it's position, otherwise return 0.

**Dynamic programming**  
This technique is applied over problems, whose solutions can be viewed as the result of a sequence of decisions.

We can achieve an optimal sequence of decisions using Greedy methods also. You have observed how we have taken decisions over the objects in kit bag problem, one at a time without making any erroneous decision. But Greedy method is applicable only to those problems where there is a scope of using local optimality in taking step-wise decisions. There are other problems for which, it is not possible to take step-wise decisions (based on only local information).

For example, we need to find a shortest path from vertex ai to aj. Let Ai be the set of vertices from vertex i. Which of the vertices in Ai should be the next on the path? We cannot take a perfect decision now, because there is no guarantee that future decisions will lead to an optimal sequence of decisions. So we can not apply Greedy technique for this problem.

Let us see one more similar example, where Greedy technique fails to obtain optimality.

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

This method is used for state-space search problems. They are problems, where the problem presentation 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 post conditions.
* A cost function - evaluates the cost of the operations (optional).
* A utility function - evaluates how close a given state to the goal state is (optional).
* A set of intermediate states.

The solving process is based on the construction of a state-space tree, the nodes in it 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.

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:

I. Read a node from the stack.  
II. While there are available operators do:

a. Apply an operator to generate a child  
b. If the child is a goal state - stop  
c. If it is a new state, push the child into the stack

The utility function is used to tell how close a given state is to the goal state and whether a given state may be considered as a goal state or not.  
If there can be no children generated from a given node, then we backtrack - read the next node from the stack.

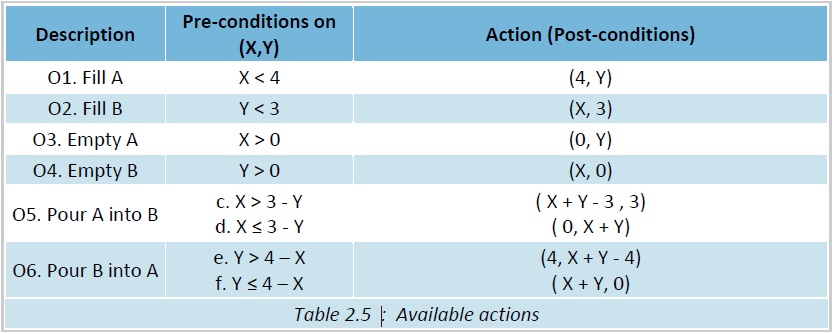
For example:  
1. Problem state- pair of numbers (X,Y) where X - water in jar 1 called A,

 Y - water in jar 2 called B.

Initial state: (0,0)  
Final state: (2,\_ ) here "\_" means "any quantity"

2. Available actions (operators):

**Available actions (operators)**

****

**Algorithm Analysis**

In computer science, the analysis of algorithms is the determination of the number of resources (such as time and storage) necessary to execute them. A good number of algorithms are designed to work with inputs of arbitrary length. Generally, the running and efficiency time of an algorithm is confirmed as a function relating to storage locations (space complexity) or the input length to the number of steps (time complexity).

Algorithm analysis is a vital part of a broader computational complexity theory. The broader computational complexity theory provides theoretical estimates for the resources needed by any algorithm that solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms, it is common to estimate the complexity function for arbitrarily large input in the asymptotic sense. Big Oh ("Add "h" to "O") notation, Big-theta notation and Big-omega notation are used to this end..

Following are the most important analysis methods  
a. Running Time Complexities  
b. Asymptotic notations  
c. Recurrence Relations

**Running Time complexities**

The complexity of an algorithm is measured by the operations needed to solve the corresponding problem. We are concerned with estimating the complexity of algorithms, where the number of operations depends on the size of the input.

Examples:

1. Reading a file: the number of read operations depends on the number of records in a file.
2. Finding a name in a list of names: the number of operations depends on the number of the names in the list.
3. Finding greatest element in an array of elements: number of operations depends on length of the array.

If N (N is the size of the input) is the number of the elements to be processed by an algorithm, then the number of operations can be represented as a function of N: f (N) (sometimes we use lower case n).

We can compare the complexity of two algorithms by comparing the corresponding functions. Moreover, we are interested what happens with the functions for large N, i.e. we are interested in the asymptotic growth of these functions.

**Classification of functions by their asymptotic growth**  
Each particular growing function has its own speed of growing. Some functions grow slower, others grow faster.

The speed of growth of a function is called asymptotic growth. We can compare functions by studying their asymptotic growth

**Asymptotic notations**  
Given a function f(n), all other functions fall into three classes:

a. Growing with the same speed as f(n)  
b. Growing slower than f(n)  
c. Growing faster than f(n)

f(n) and g(n) have the same speed of growing, if  
lim (f(n)/g(n)) = c, 0 < c < ∞, n -> ∞

**Notation:** f(n) = Θ(g(n)) , pronounced "theta"

**Discussion**  
f(n) and g(n) have the same speed of growth if  
lim ( f(n)/g(n) ) = c, 0 < c < ∞, n → ∞  
Let Θ( f(n) ) be the set of all functions, that grow with the speed of f(n).  
If g(n) has the same speed of growth, then g(n) є Θ( f(n) ).

**The Big-Oh notation**

The Big-Oh notation is used to shorten our reasoning about growth speeds.  
 f(n) = O( g(n) ) if f(n) grows with the same speed or slower than g(n).

i.e. if f(n) = Θ(g(n)) or f(n) = o(g(n)), then we write f(n) = O(g(n))  
Thus n+5 = Θ(n) = O(n) = O(n2) = O(n3) = O(n5)

While all the equalities are technically correct, we would like to have the closest estimation:  
n+5 = Θ(n). However, general practice is to use the Big-Oh notation and to write:  
n+5 = O(n)

**The Big-Omega notation**

Big-Omega notation is the inverse of the Big-Oh: If g(n) = O(f(n)), then f(n) = Ω (g(n))  
Here we say that f(n) grows faster or with the same speed as g(n), and write f(n) = Ω(g(n))  
We shall use mainly the Big-Oh estimate for analysis of algorithm.

**Recurrence relations**

The recurrence relations can be solved in three ways

a. Substitution method  
b. Recursion tree method  
c. Master's method

**Well-known Fundamental Algorithms**  
In this section, we present the most popular algorithms. The Fundamental Algorithm provides sound theoretical basis for testing existing algorithms or generating any new algorithms to meet specific needs. In the coming sections, we will discuss about the well-known fundamental algorithms, their development, description and their performance analysis.

We will have a closer look on the topics  
1. Exchanging the Values of Two Variables  
2. Counting  
3. Summation of a Set of Numbers  
4. Factorial Computation  
5. Reversing the Digits of an Integer

**Sorting Algorithms**  
**Introduction**  
In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly. It is also often useful for canonicalizing data and for producing human-readable output. We will discuss

1. Sorting by Selection  
2. Sorting by Exchange  
3. Sorting by Insertion  
4. Sorting by Partitioning  
5. The Two-way Merge Sorting

**Searching Algorithms**  
**Introduction**

Search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables.

Search Algorithm Types,  
1. Linear Search  
2. Binary Search  
3. Hash Algorithm