DESIGN OF ALGORITHMS

INTRODUCTION:

Algorithms are procedures that take in some input and produce some output. The reason why algorithms should be studied is because it does not rely specifically on technology. Algorithms do not require we to know java or c++ or whatever technology comes out tomorrow. It requires that we have an understanding of the concept of the steps that are involved to solve a problem for a given set of programming languages with common features.

The topic of algorithms is important in computer science because it allows for analysis on different ways to compute things and ultimately come up with the best way to solve a particular problem. By best we mean one that consumes the least amount of resources or has the fastest running time.

Algorithm-definition:

An algorithm is a finite step-by-step procedure to achieve a required result.

Characteristics of Algorithm:

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

Performance of Algorithms:

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the O (Big Oh)

notation. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation. The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the size n and is to be considered modulo a multiplicative constant.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

Θ-Notation (Same order):

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between c_1 $g(n)$ and c_2 $g(n)$ inclusive.In the set notation, we write as follows:$\Theta(g(n))$ = {$f(n)$: there exist positive constants c_1, c_1, and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$}

O-Notation (Upper Bound):

This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c\, g(n)$.In the set notation, we write as follows: For a given function $g(n)$, the set of functions$O(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$}.

Ω-Notation (Lower Bound):

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c\, g(n)$.In the set notation, we write as follows: For a given function $g(n)$, the set of functions $\Omega(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$}

The functions can be shown in the increasing order as below, where N is the number of input.

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

The following sections describe about the some of the techniques used in designing algorithms.

DIVIDE-AND-CONQUER METHOD:-

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to subproblems to create a solution to the original problem.

Example for Divide and Conquer : Binary Search

Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reached the required (given) element.

Problem Let $A[1 \ldots n]$ be an array of non-decreasing sorted order; that is $A[i] \leq A[j]$ whenever $1 \leq i \leq j \leq n$. Let *'q'* be the query point. The problem consist of finding *'q'* in the array *A*. If *q* is not in *A*, then find the position where *'q'* might be inserted. Formally, find the index *i* such that $1 \leq i \leq n+1$ and $A[i-1] < x \leq A[i]$.

Look for *'q'* either in the first half or in the second half of the array *A*. Compare *'q'* to an element in the middle, $\lceil n/2 \rceil$, of the array. Let $k = \lceil n/2 \rceil$. If $q \leq A[k]$, then search in the $A[1 \ldots k]$; otherwise search $T[k+1 \ldots n]$ for *'q'*. Binary search for *q* in subarray $A[i \ldots j]$ with the promise that

$A[i-1] < x \leq A[j]$

If $i = j$ then

return *i* (index)

$k = (i + j)/2$

if $q \leq A[k]$

then return Binary Search $[A[i-k], q]$

else return Binary Search $[A[k+1 \ldots j], q]$

Binary Search can be accomplished in logarithmic time in the worst case , i.e., $T(n) = \theta(\log n)$.

GREEDY METHOD:-

Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are used to solve optimization problems. Greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later.

Structure Greedy Algorithm

- Initially the set of chosen items is empty i.e., solution set.
- At each step
  - item will be added in a solution set by using selection function.
  - IF the set would no longer be feasible

- reject items under consideration (and is never consider again).

- ELSE IF set is still feasible THEN
  - add the current item.

Example for greedy method: Greedy Solution to the Fractional Knapsack Problem

There are n items in a store. For i =1,2, . . . , n, item i has weight $w_i > 0$ and worth $v_i > 0$. Thief can carry a maximum weight of W pounds in a knapsack. In this version of a problem the items can be broken into smaller piece, so the thief may decide to carry only a fraction x_i of object i, where $0 \leq x_i \leq 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.In Symbol, the fraction knapsack problem can be stated as follows. maximize $^{n}\mathbf{S}_{i=1}\, x_i v_i$ subject to constraint $^{n}\mathbf{S}_{i=1}\, x_i w_i \leq W$

It is clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load. Thus in an optimal solution $^{n}\mathbf{S}_{i=1}\, x_i w_i = W$.

Greedy-fractional-knapsack (*w, v, W*)

```
FOR i =1 to n
   do x[i] =0
weight = 0
while weight < W
   do i = best remaining item
      IF weight + w[i] ≤ W
         then x[i] = 1
              weight = weight + w[i]
         else
              x[i] = (w - weight) / w[i]
              weight = W
return x
```

If the items are already sorted into decreasing order of v_i / w_i, then the while-loop takes a time in $O(n)$; Therefore, the total time including the sort is in $O(n \log n)$.

DYNAMIC PROGRAMMING:

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (sub problems), then combine the solutions of the sub problems to reach an overall solution. Often, many of these sub problems are really the same. The dynamic programming approach seeks to solve each sub problem only once, thus reducing the number of computations: once the solution to a given sub problem has been computed, it is stored. The next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating sub problems grows exponentially as a function of the size of the input

Example for dynamic programming: Fibonacci sequence

In the bottom-up approach we calculate the smaller values of fib first, then build larger values from them.

```
function fib(n)
    if n = 0
        return 0
    var previousFib := 0, currentFib := 1
    else repeat n − 1 times  // loop is skipped if n=1
        var newFib := previousFib + currentFib
        previousFib := currentFib
        currentFib  := newFib
    return currentFib
```

For example, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

This method also uses O(*n*) time since it contains a loop that repeats n − 1 times, however it only takes constant (O(1)) space.

BACKTRACKING:

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

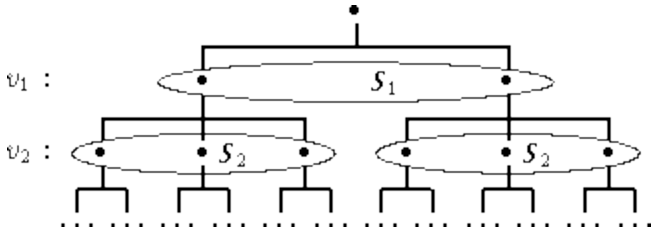
Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (v_1, ..., v_i) which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

```
ALGORITHM try(v1,...,vi)
  IF (v1,...,vi) is a solution THEN RETURN (v1,...,vi)
  FOR each v DO
    IF (v1,...,vi,v) is acceptable vector  THEN
      sol = try(v1,...,vi,v)
      IF sol != () THEN RETURN sol
    END
  END
  RETURN ()
```

If S_i is the domain of v_i, then $S_1 \times ... \times S_m$ is the solution space of the problem. The validity criteria used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.



Example for backtracking: eight queen problem:

The classic example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution.

BRANCH AND BOUND:-

Branch and bound is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. A branch-and-bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded, by using upper and lower estimated bounds of the quantity being optimized.

General description:

In order to facilitate a concrete description, we assume that the goal is to find the minimum value of a function , where ranges over some set of admissible or candidate solutions (the search space or feasible region). A branch-and-bound procedure requires two tools. The first one is a splitting procedure that, given a set of candidates, returns two or more smaller sets whose union covers . This step is called branching, since its recursive application defines a tree structure (the search tree) whose nodes are the subsets of .

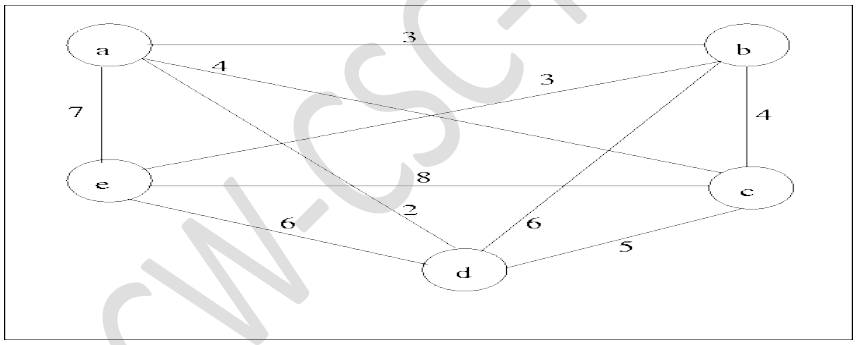
The second tool is a procedure that computes upper and lower bounds for the minimum value of within a given subset of . This step is called bounding. The key idea of the BB algorithm is: if the lower bound for some tree node (set of candidates) is greater than the upper bound for some other node , then may be safely discarded from the search. This step is called pruning, and is usually implemented by maintaining a global variable (shared among all nodes of the tree) that records the minimum upper bound seen among all sub regions examined so far. Any node whose lower bound is greater than can be discarded.

The recursion stops when the current candidate set is reduced to a single element, or when the upper bound for set matches the lower bound. Either way, any element of will be a minimum of the function within it

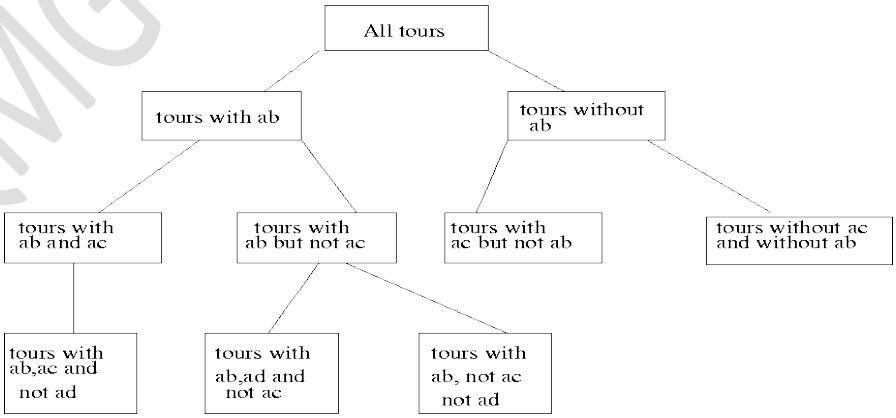
Example for Branch and Bound: The traveling salesman problem

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Example of a complete graph with five vertices



A solution tree for a TSP instance



In the above solution tree, each node represents tours defined by a set of edges that must be in the tour and a set of edges that may not be in the tour. These constraints alter our choices for the two lowest cost edges at each node.

Each time we branch, by considering the two children of a node, we try to infer additional decisions regarding which edges must be included or excluded from tours represented by those nodes. The rules we use for these inferences are:
If excluding (x, y) would make it impossible for x or y to have as many as two adjacent edges in the tour, then (x, y) must be included.

If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (x, y) must be excluded.

When we branch, after making what inferences we can, we compute lower bounds for both children. If the lower bound for a child is as high or higher than the lowest cost found so far, we can ``prune" that child and need not consider or construct its descendants.

If neither child can be pruned, we shall, as a heuristic, consider first the child with the smaller lower bound. After considering one child, we must consider again whether its sibling can be pruned, since a new best solution may have been found.

REFERENCES:

1.http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/intro.htm
2.http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/divide.htm
3.http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm
4.http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/knapscakFrac.htm
5.http://en.wikipedia.org/wiki/Dynamic_programming#Fibonacci_sequence

6.http://wiki.answers.com/Q/What_are_the_characteristics_of_an_algorithm_describe_with_an_example

7.http://wiki.answers.com/Q/Why_are_algorithms_so_important

8.http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html

9.http://en.wikipedia.org/wiki/Branch_and_bound

10.http://lcm.csa.iisc.ernet.in/dsa/node187.html

.