Choosing Best Algorithm Design Strategies For a Particular Problem

Thesis submitted in partial fulfillment of the requirements for the award of Degree of

Master of Engineering

in

Software Engineering

By:

Name: Shailendra Kumar Nigam

Roll No: 80731021

Under the supervision of:

Dr. Deepak Garg

Assistant Professor, CSED

&

Mr. Ravinder Kumar

Lecturer, CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

JUNE 2009

Certifica

I hereby certify that the work which is being presented in the thesis report entitl **"Choosing Best Algorithm Design Strategies for a Particular Problem"**, submitted me in partial fulfillment of the requirements for the award of degree of Master Engineering in Computer Science and Engineering submitted in Computer Science a Engineering Department of Thapar University, Patiala, is an authentic record of my ow work carried out under the supervision of **Dr. Deepak Garg** and **Mr. Ravinder Kum** and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degre of this or any other university.

Shailendra.

(Shailendra Kumar Nigam)

This is to certify that the above statement made by the candidate is correct and true to th best of my knowledge.

[illegible]

(Dr. Deepak Garg)

Assistant Professor, CSED

Thapar University

Patiala

And

[illegible]

(Mr. Ravinder Kumar)

Lecturer, CSED

Thapar University

Patiala

Countersigned by:

[illegible] 9/07/09.

Dr. Rajesh Kumar Bhatia

Assistant Professor & Head

Computer Science & Engineering. Department

Thapar University

Patiala.

[illegible] 13/7/09

(Dr. R.K.SHARMA)

Dean (Academic Affairs)

Thapar University,

Patiala.

Acknowledgement

No volume of words is enough to express my gratitude towards my guide, **Dr. Deepak Garg Assistant Professor.** Who has been very concerned and has aided for all the material essential for the preparation of this thesis work. I would also thankfull to **Mr. Ravinder Kumar** for his continual support, encouragement and invaluable suggestions towards the research area. They have helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am thankful to **Dr. Rajesh Kumar Bhatia, Head of Computer Science & Engineering Department Thapar University Patiala and Mrs. Inderveer Channa, P.G. Coordinator** for providing us adequate environment, facility for carrying thesis work.

I would like to thank to all staff members who were always there at the need of hour and provided with all the help and facilities, which I required for the completion of my thesis.

I would also like to express my appreciation to my co-worker and my friends Jitender, Nupur, Mandeep, Aman, Mahima, Balaji for motivation and providing interesting work environment. It was great pleasure in working with them during this thesis work.

At last but not the least I would like to thank God and my parents for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Shailendra.

Shailendra Kumar Nigam

Roll No. 80731021

M.E (Software Engineering)

ABSTRACT

Algorithms have come to be recognized as the cornerstone of computing. Algorithm design strategies are typically organized either by application area or by design technique. This report describes different designing algorithms such as Brute force, Greedy, Divide and Conquer, Dynamic programming, Backtracking, Branch and Bound and many more. It describes how a particular algorithm is used for a specific problem. This report also proposes how to choose the best algorithm design strategy for a particular problem to facilitate the development of best algorithms based upon algorithm design strategy techniques. It also describes how a particular algorithm is used for a specific problem. Taking various parameters does a comparison of various algorithms. This report advocates a wider use of different problems in teaching the best algorithm design strategies.

Table of Contents

List of Tables

List of Figures

CHAPTER 1
INTRODUCTION

1.1 Introduction

Our studies reveal how people design algorithms those that principal design methods are used based upon some parameters in the absence of specific knowledge and belief that these parameters will play an equally important role in the design of algorithms. A study of algorithms has come to be recognized as the cornerstone of computer science. The progress in this field to date, however, has been very uneven. While the framework for analysis of algorithms has been firmly established and successfully developed for quite some time, much less effort has been devoted to algorithm design techniques.

This comparative lack of interest is surprising and unfortunate in view of the two important payoffs in the study of algorithm design techniques: "First, it leads to an organized way to devise algorithms. Algorithm design techniques give guidance and direction on how to create a new algorithm. Though there are literally thousands of algorithms, there are very few design techniques. Second, the study of these techniques help us to categorize or organize the algorithms. Although some algorithms design strategies are better than others on average, there is rarely a best algorithm design strategies for a given problem. Instead, it is often the case that different algorithms design strategies perform well on different Problem instances [1]. Not surprisingly, this phenomenon is most pronounced among algorithms for solving hard problems, because runtimes for these algorithms design strategy are often highly variable from instance to instance. Choosing best algorithm design strategy is one of the most difficult decisions. Algorithm design is a specific method to create a mathematical process in solving problems. Applied algorithm design is algorithm engineering. Techniques for designing and implementing algorithm designs are algorithm design patterns, such as template method patterns, decorator patterns, uses of data structures, and name and sort lists. Some current uses of algorithm design can be found in Internet retrieval processes of web crawling packet routing and caching [2]. Use employ evolving design strategy to make algorithm. The initial design is a correct, if inefficient, solution to the problem but may not be highly efficient each subsequent design is an improvement or optimization of the prior design and the final design is an optimal, algorithm for solving the problem. At each stage optimized strategy is applied and the effect on algorithmic complexity is derived. As each transformation is considered, additional abstractions Necessary to express the design strategies are introduced.

The pedagogical advantages of the successive design strategy include:

- Students see design principles applied in a precise context.
- A succession of different principles is applied in the stages of the design for a single problem. This models the design process in the real world.
- At each stage, the demonstration of correctness requires showing only correctness relative to the preceding algorithm.

1.2 How to Design Algorithms

Creating an algorithm design is an art, which may never be fully automated. Various algorithm design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Algorithm is a method for solving a computational problem and algorithm design is identified in many solution theories of operation research, such as divide and conquer, dynamic programming and greedy algorithm.

The Techniques for designing and implementing algorithm design is based on template method patterns, data structures etc. A Design technique is often expressed in pseudocode as a template that can be particularized for concrete problems [3]. This template name is algorithm schemas. Algorithm schemas consist on identifying structural similarities among algorithms that solve different problems. Programming language such as algorithmic language, COBOL, FORTRAN, PASCAL, SAIL are computing tools to implement an algorithm design. But algorithm design is not a programming tool. Algorithm design basically mathematical process of writing a finite set of steps each of which may require one or more operations.

1.3 How to Express Algorithms

Algorithm design can be expressed in many kinds of notation like flowcharts, programming language, rational rose tool, computer aided design applications and pseudocode etc. flowcharts and rational rose are expressed in structured way and avoid the ambiguities in language statements. Computer-Aided Design (CAD), also known as Computer-Aided Drafting, is the use of computer software and systems to design and create 2D and 3D virtual models of goods and products for the purposes of testing. It is also sometimes referred to as computer assisted drafting. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. The algorithm need not as yet be expressed as program. It is sufficient to state it in any precise way. The purpose of assures that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in [4].

1.4 Fundamental Concepts of Algorithm

There are two fundamental concepts of algorithm

- Functional correctness
- Proof of correctness

1.4.1 Functional Correctness [5].

Functional correctness is depend on the following

- **Precondition:** Algorithm is correct if every data that satisfy some condition that is called precondition of the algorithm
- **Post condition:** The out put data satisfy a certain predefined condition that is called post condition of the algorithm.

1.4.2 Proofs of Correctness of Algorithms [6].

Correctness of algorithm is depends on the two issues

- Given an algorithm prove that it is correct. It is always achieves the intended result
- Design an algorithm with intended properties from scratch. This is even more difficult.

Proof of correctness also depends on the mathematical proof. Whenever algorithm is run on a set of inputs that satisfy the problems precondition is expected to hold before the method is executed. Post condition what holds after the method is executed. A proof that a program is correct often has two pieces (that can be developed separately)

- **Proof of partial correctness** [7]**:** This is a proof that, whenever an algorithm is run on a set of inputs satisfying the problem's precondition, either
    - The algorithm halts, and the outputs (and inputs) satisfy the problem's post condition, or
    - The algorithm does not halt at all.
- **Proof of termination** [7]**:** This is a proof that the algorithm always halts, whenever it is run on a set of inputs that satisfy the precondition.

CHAPTER 2

CONCEPT OF ALGORITHMS DESIGN STRATEGIES

2.1 Brute Force Algorithms

Definition: An algorithm that inefficiently solves a problem, often by trying every one of a wide range of possible solutions

Main Approach

- Generate and evaluate possible solutions until
    - Satisfactory solution is found.
    - Best solution is found.
    - All possible solutions found
        - Return best solution
        - Return failure if no satisfactory solution.
- Generally most expensive approach.

Description

A brute force algorithm simply tries all possibilities until a satisfactory solution is found such an algorithm can be:

- **Optimizing:** Find the best solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
- **Satisfying:** Stop as soon as a solution is found that is good enough

Brute force algorithm is require exponential time and used in various heuristics and optimizations can be used

Heuristic: A rule of thumb that helps you decide which possibilities to look at first.

Optimization: In this case, to eliminate certain possibilities without fully exploring them [8].

The C code [9]

```
Void BF (char *x, int m, char *y, int n) {
```

```
int i, j;
/* Searching */
for (j = 0; j <= n - m; ++j)
{
        for (i = 0; i < m && x[i] == y[i + j]; ++i);
                if (i >= m)
                        OUTPUT(j);
}
```

Strengths:

- Wide applicability
- Simplicity
- Yields reasonable algorithms for some important problems
    - Searching, string matching, matrix multiplication
- Yields standard algorithms for simple computational tasks
    - Sum or product of *n* numbers, finding max or min in a list

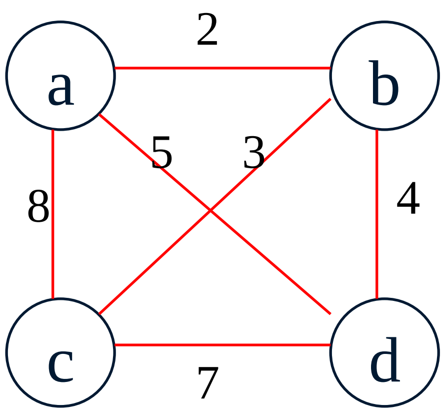
Weaknesses:

- Rarely yields efficient algorithms
- Some brute force algorithms unacceptably slow
- Not as constructive/creative as some other design techniques

Example 1: Traveling salesman problem

Question: 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.

Example:



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

2.2 Greedy Algorithm [10]

The greedy algorithm is perhaps the most straightforward design technique. It can be applied to a wide variety of problem. Most though not all of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

Note: greedy algorithm avoid backtracking and exponential time $O(2^n)$

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences E.g. Kruskal’s MST algorithm, Dijkstra’s algorithm [11].

Definition

Greedy algorithm work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. Greedy algorithm to find minimum spanning tree. Want to find set of edges [12].

Note: Prim’s algorithm and Kruskal’s algorithm are greedy algorithms that find the globally optimal solution, a minimum spanning tree. In contrast, any known greedy algorithm to find a Hamiltonian cycle might not find the shortest path, that is, a solution to the traveling salesman problem. If there is no greedy algorithm that always finds the optimal solution for a problem, one may have to search (exponentially) many possible solutions to find the optimum. Greedy algorithms are usually quicker, since they don’t consider the details of possible alternatives

Type of Greedy Algorithm

There are three type of greedy algorithms

- Pure Greedy Algorithms
- Orthogonal Greedy Algorithms
- Relaxed Greedy Algorithms

General Characteristics of Greedy Algorithms [13]

Commonly, greedy algorithms and the problems they can solve are characterized by most or all of the following features.

- To construct the solution of our problem, a set (or list) of candidates is required: the coins that are available, the edges of a graph that may be used to build a path, the set of jobs to be Scheduled, or whatever.
- As the algorithm proceeds, two other sets are accumulated. One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.
- There is a function that checks whether a particular set of candidates provides a solution to our problem, ignoring questions of optimality for the time being. For instance, do the coins add up to the amount to be paid? Do the selected edges provide a path to the node to reach? Have all the jobs been scheduled?
- A second function checks whether a set of candidates is feasible, that is, whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution to our problem. Here too, the time being concerned is not with optimal1ty.
- Yet another function, the selection function, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.
- Finally an objective function gives the value of a solution found: the number of coins used to make change, the length of the path constructed, the time needed to process all the jobs in the schedule, or whatever other values are trying to optimize. Unlike the three functions mentioned previously, the objective function does not appear explicitly in the greedy algorithm.

Example: - Scheduling

Given jobs j1, j2, j3, ..., jn with known running times t1, t2, t3, ..., tn. what is the best way to schedule the jobs to minimize average completion time?

Job	**Time**
J1	16
J2	8
J3	3
J4	14

Scheduling

J1	J2	J3	J4
16	24	27	41

Average completion time = (16+24+27+41)/4 = 27

J3	J2	J4	J1
3	11	25	41

Average completion time = (3+11+25+41)/4 = 20

Description

- Greedy-choice property: if shortest job does not go first, the *y* jobs before it will complete 3 time units faster, but j3 will be postponed by time to complete all jobs before it
- Optimal substructure: if shortest job is removed from optimal solution, remaining solution for n-1 jobs is optimal

Optimality Proof

•Total cost of a schedule is

$$\sum_{k=1}^{N}(N-k+1)t_{ik}$$

t1 + (t1+t2) + (t1+t2+t3) ... (t1+t2+...+tn)

$$(N+1)\sum^{N} t_{ik} - \sum k*t_{ik}$$

$k=1$

•First term independent of ordering, as second term increases, total cost becomes smaller

Suppose there is a job ordering such that $x > y$ and $t_{ix} < t_{iy}$ Swapping jobs (smaller first) increases second term decreasing total cost

Show: $xt_{ix} + yt_{iy} < yt_{ix} + xt_{iy}$

$xt_{ix} + yt_{iy} = xt_{ix} + yt_{ix} + y(t_{iy} - t_{ix})$

$= yt_{ix} + xt_{ix} + y(t_{iy} - t_{ix})$

$< yt_{ix} + xt_{ix} + x(t_{iy} - t_{ix})$

$= yt_{ix} + xt_{ix} + xt_{iy} - xt_{ix} \qquad = yt_{ix} + xt_{iy}$

2.3 Divide and Conquer

Divide and conquer algorithm suggests splitting the inputs into distinct subsets. These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide and conquer strategy can be possibly be reapplied. Often the sub problems resulting from a divide and conquer design are of the same type as the original problem. For those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive algorithm. This algorithm technique is the basis of efficient algorithms for all kinds of the problems, such as quick sort, merge sort and discrete Fourier transform. Its application to numerical algorithms is commonly known as binary splitting [4].

Divide-and-conquer algorithm works as follows:

- Divide and conquer algorithm are divided into several smaller instances of the same problem and same size.
- The smaller instances are solved by recursively.
    - Sometimes, a different algorithm is applied when instances become small enough.
- The smaller instances are combined and to get a solution to the original problem.
    - No necessary to combine in some cases.

- Divide-and-conquer technique is ideally suited for parallel computers, in which each sub problem can be solved simultaneously by its own processors.
- Common case: Dividing a problem into two smaller problems

Algorithm of Divide and Conquer

1. Algorithm D-and-C (n: input size)
2. If n ≤ n0 /* small size problem*/
3. Solve problem without further sub-division;
4. Else
5. Divide into m sub-problems;
6. Conquer the sub-problems by solving them
7. Independently and recursively; /* D-and-C (n/k) */
8. Combine the solutions;

EXAMPLES:

- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Strassen’s algorithm
- VLSI tree layout

Divide and Conquer

Divide: P => P1,…………….Pk

Conquer: S (P1),……………...S (Pk)

Merge: S (P1),………………., S (Pk)=> S(P)

Examples: Sorting (merge sort and quick sort), searching (binary search), closest pair (the *O* (*n* log *n*) algorithm), and selection (the linear-time algorithm).

Algorithm template:

- Function P(n)
- if n <=c
- Solve P directly
- Return its solution

- Else P => P1, ..., Pk //divide
- For i = 1 to k
- Si = P(ni) //conquer
- S1, ..., Sk => S //merge
- return S

Time complexity:

$$T(n) = \begin{cases} 1 & n \le c \\ \sum_{i=1}^{k} T(ni) + D(n) + M(n), & n < c \end{cases}$$

Strassen's algorithm

-Given A=(a_{ij}) $_{nxn}$ and B=(b_{ij}) $_{nxn}$.

$$Let\ C = A\ x\ B = (cij)n\ xn\ , for\ Cij = \sum_{k=1}^{n} aik\ bkj$$

– First algorithm:

```
for i = 1 to n
for j = 1 to n
c[i,j] = 0
for k = 1 to n
c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

Time complexity: O (n^3)

- Second algorithm:

$$\mathbf{Anxn} = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$$

$$\mathbf{Bnxn} = \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

$$\mathbf{Cnxn} = \begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix}$$

$C_{11} = A_{11}\ B_{11} + A_{12}\ B_{21}$

$C_{12} = A_{11}\ B_{12} + A_{12}\ B_{22}$

$C21 = A_{21}\ B_{11} + A_{22}\ B_{21}$

$C22 = A_{21}\ B_{12} + A_{22}\ B_{22}$

So the multiplication of two n x n matrices becomes eight multiplications of two n/2 x n/2 matrices, giving us $T(n)=8T(n/2)+O(n^2)$. By iterating, we have $T(n)=O(n^3)$. No improvement!

M_1	$(A_{12} - A_{22})(B_{21} + B_{22})$
M_2	$(A_{11} + A_{22})(B_{11} + B_{22})$
M_3	$(A_{11} - A_{21})(B_{11} + B_{12})$
M_4	$(A_{11} + A_{12})B_{22}$
M_5	$A_{11}(B_{12} - B_{22})$
M_6	$A_{22}(B_{21} - B_{11})$
M_7	$(A_{21} + A_{22})B_{11}$

C_{11}	$M_1 - M_2 - M_4 + M_6$
C_{12}	$M_4 + M_5$
C_{21}	$M_6 + M_7$
C_{22}	$M_2 - M_3 + M_5 - M7$

Using the above idea in the algorithm, we get

$T(n) = 7T(n/2)+O(n^2)$, thus $T(n) = O(n^{\log 7}) = O(n^{2.81})$

By iterating

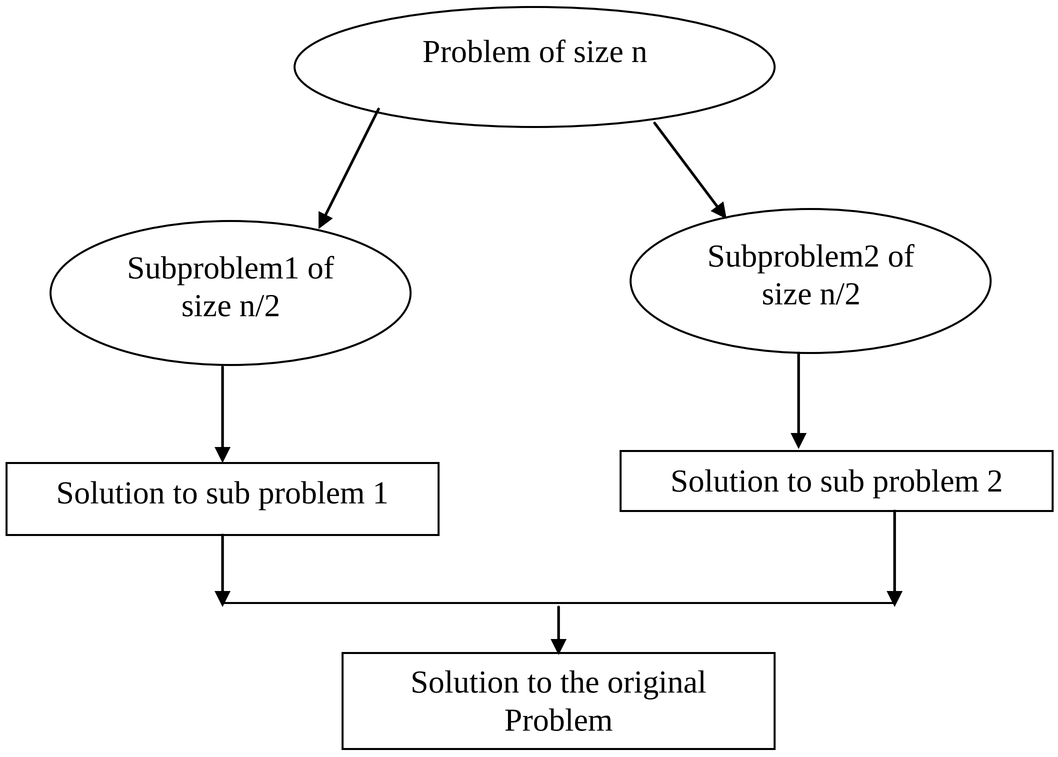

Problem of size n
Subproblem1 of size n/2
Subproblem2 of size n/2
Solution to sub problem 1
Solution to sub problem 2
Solution to the original Problem

Figure 2.1 Divide and Conquer

2.4 Dynamic programming [4][14]

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. Dynamic programming is a similar to divide and conquer algorithm. It is express solution of a problem in terms of solutions to sub problems. The Key difference is between dynamic programming and divide and conquer is that while sub problems in divide and conquer are independent, sub problems in dynamic programming may them selves share sub problems. This means that if these were treated as independent sub problems, the complexity would be higher. Dynamic programming is typically used to solve optimization problems. In bioinformatics, the most common use of dynamic programming is in sequence matching and alignment.

- To begin, the word programming is used by mathematicians to describe a set of rules, which must be followed to solve a problem.
- Thus, linear programming describes sets of rules which must be solved a linear problem.
- In our context, the adjective dynamic describes how the set of rules works.
- In this course, a number of examples of recursive algorithms are seen.

- The run time of these algorithms may be found by solving the recurrence relation itself.
- The first example of a dynamic program is a technique for solving the following recurrence relation [9].

$$\mathrm{F}(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ \mathrm{F}(n-1)+\mathrm{F}(n-2) & n>1\end{cases}$$

- You will recall that this defines the Fibonacci sequence of integers:

  1, 1, 2, 3, 5, 8, 13, 21, 33, 54,...

Example

```
// Calculate the nth Fibonacci number
Double F( double n ) {
   if ( n <= 1 ) {
      return 1.0;
   } else {
      return F( n – 1 ) + F( n – 2 );
   }
}
```

- Recall definition of Fibonacci numbers: f(0) = 0

  f(1) = 1

  f(n) = f(n-1) + f(n-2)
- Compute the nth Fibonacci number recursively (top-down)

```
                     f(n)
          f(n-1)       +       f(n-2)
   f(n-2)  +  f(n-3)      f(n-3)  +  f(n-4)
```

Example: Fibonacci numbers (2)

Compute the *n*th Fibonacci number using bottom-up iteration:

1. F (0) = 0
2. F (1) = 1
3. F (2) = 0+1 = 1

4. F (3) = 1+1 = 2
5. F (4) = 1+2 = 3
6. F (n-2) =
7. F (n-1) =
8. F (n) = f (n-1) + f (n-2)

Example: Computing binomial coefficients

Algorithm Based On Identity

- Algorithm Binomial (n,k)
  for i <- 0 to n do
    1. for j <-0 to min(j,k) do
    2. if j=0 or j=i then C[i,j] ß 1
    3. else C[i,j]ßC[i-1,j-1]+C[i-1,j]
    4. return C[n,k]
- Pascal’s Triangle

2.5 Backtracking Algorithm [4][15]

Backtracking algorithm represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. Many of the problems being solved using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories explicit and implicit.

- View the problem as a sequence of decisions
- Systematically considers all possible outcomes for each decision
- Backtracking algorithms are like the brute-force algorithms
- However, they are distinguished by the way in which the space of possible solutions is explored
- Sometimes a backtracking algorithm can detect that an exhaustive search is not needed

Example: - Solving a maze

- Given a maze, find a path from start to finish

- At each intersection, you have to decide between three or fewer choices:
    - Go straight
    - Go left
    - Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Solving a puzzle

- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed
- The object is to remove all but the last peg
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking

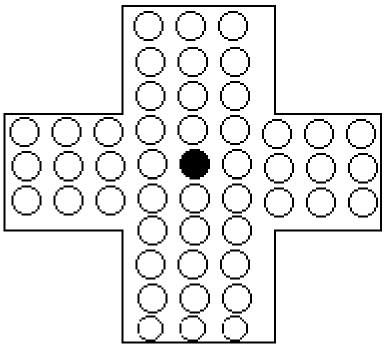


Figure 2.2 Puzzle Problem

2.6 Branch and Bound Algorithms [4][16]

Branch and Bound Algorithm based on limiting search using current solution. It means this is a general search method. This method considering the root problem and lower bounding and upper bounding procedures are applied to the root problem. Branch and bound algorithm is applied recursively to the sub problem. If an optimal solution is found to a sub problem, it is a feasible solution to the full problem, but not necessarily globally optimal.

Branch and Bound Algorithm Approach

- Firstly try to track best current solution found
- The partial current solutions that can't be improved that should be eliminated.
- Reduces amount of backtracking

Note: Not guaranteed to avoid exponential time O (2^n)

Basic features of Branch and Bound Algorithm

Best solution is only compared with a nodes bound values only if the bound value us not better then the best solution so far there are following reasons

- The value of the node bound is not better than the other
- Node does not represent the feasible solutions
- The node consists of a single point represent the subset of feasible solutions.

Example: Assignment Problem [17]

Assigning n people to n jobs so that the total cost is minimized. Each person does one job and each job is assigned to one person.

Read the assignments as <Job 1, Job 2, Job 3, Job 4>:

<c,b,a,d> assigns Person *c* Job 1, Person *b* Job 2, *etc.*

$$C = \begin{array}{cccc|l} \textbf{Job 1} & \textbf{Job2} & \textbf{Job3} & \textbf{Job4} & \\ 3 & 2 & 7 & 8 & \text{Person a} \\ 6 & 4 & 3 & 7 & \text{Person b} \\ 5 & 8 & 1 & 8 & \text{Person c} \\ 7 & 6 & 9 & 4 & \text{Person d} \end{array}$$

<a,b,c,d> cost =3+4+1+4=12

<a,b,d,c> cost=3+4+9+8=24

<a,d,b,c> cost=3+6+4+8=21

<d,a,b,c> cost=7+2+3+8=20

<d,c,b,a> cost =7+8+3+8=26

Etc. totaling 4! Permutations.

Permutations: Generate n! Permutations. The following prints all the costs of the n! Job assignments

All permutations algorithm - this is a simple algorithm just to generate all n! Permutations

Assumes: person <- a + 1 => person = b

Initially, X[a..d] is unassigned any Job.

Permutations(X[a..d], person)

```
1.      if person = d then print cost(X)        -- Bottom of space
2.      else
3.          for Job ∈{1, 2, 3, 4 } do
4.            if not assigned(X, Job)
5.               X[person+1] <-Job             -- Assign person a job
6.               Permutations( X[a..d], person+1 )
7.               X[person+1] <- Φ              -- Unassign job
```

Cost (X) returns cost of assigning Job 1..4 to person a..d

Assigned (X, Job) returns true if Job is assigned person a..d

The resulting state-space for assigning Jobs {1, 2, 3, 4} to each person {a, b, c, d} is:

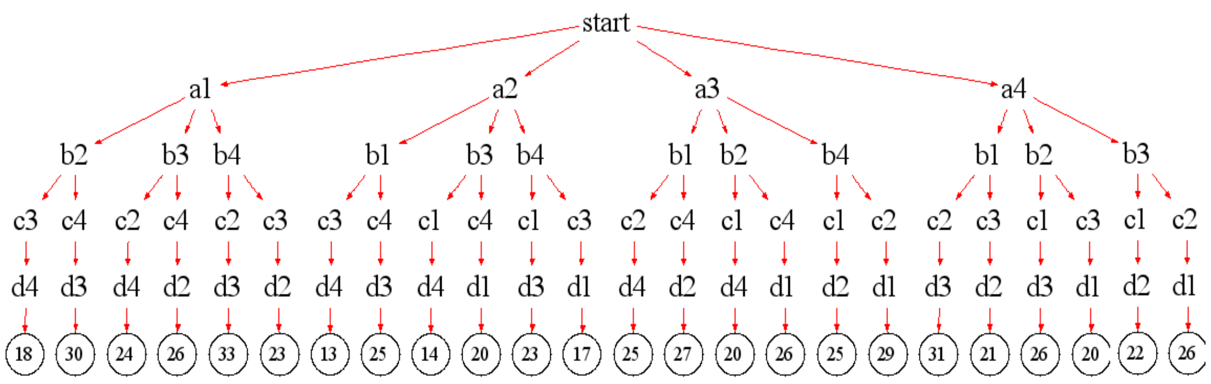

start
a1
a2
a3
a4
b2 b3 b4 b1 b3 b4 b1 b2 b4 b1 b2 b3
c3 c4 c2 c4 c2 c3 c3 c4 c1 c4 c1 c3 c2 c4 c1 c4 c1 c2 c2 c3 c1 c3 c1 c2
d4 d3 d4 d2 d3 d2 d4 d3 d4 d1 d3 d1 d4 d2 d4 d1 d2 d1 d3 d2 d3 d1 d2 d1
18 30 24 26 33 23 13 25 14 20 23 17 25 27 20 26 25 29 31 21 26 20 22 26

Figure 2.3 Branch and bound Assignment problem

$$C = \begin{array}{c|cccc|c} & \textbf{Job 1} & \textbf{Job2} & \textbf{Job3} & \textbf{Job4} & \\ \hline & 3 & 2 & 7 & 8 & \text{Person a} \\ & 6 & 4 & 3 & 7 & \text{Person b} \\ & 5 & 8 & 1 & 8 & \text{Person c} \\ & 7 & 6 & 9 & 4 & \text{Person d} \end{array}$$

From the table above, the rightmost branch <d, c, b, a>, cost=$7^{d1} + 8^{c2} + 3^{b3} + 8^{a4}$=26

EXAMPLE: 4-queens problem

EXAMPLES: a) Longest Common Subsequence (LCS)

Given two sequences *x*[1 . . *m*]and *y*[1 . . *n*], find a longest subsequence common to them both.

b) Optimal Substructure

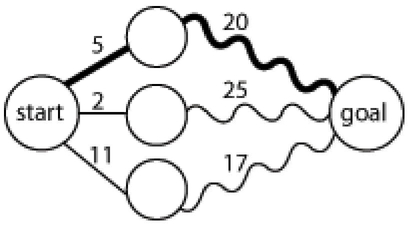


Figure 2.4 Optimal Substructure

2.7 Decrease-And-Conquer algorithm [18][19]

Decrease-and-conquer is an approach to solving a problem by:

- Change an instance into one smaller instance of the problem.
- Solve the small instance.
- Convert the solution of the small instance into a solution for the large instance.

Decrease by a Constant

Decrease-by-a constant decreases the instance size by 1 (or some other constant), e.g., 210 = 2 * 29

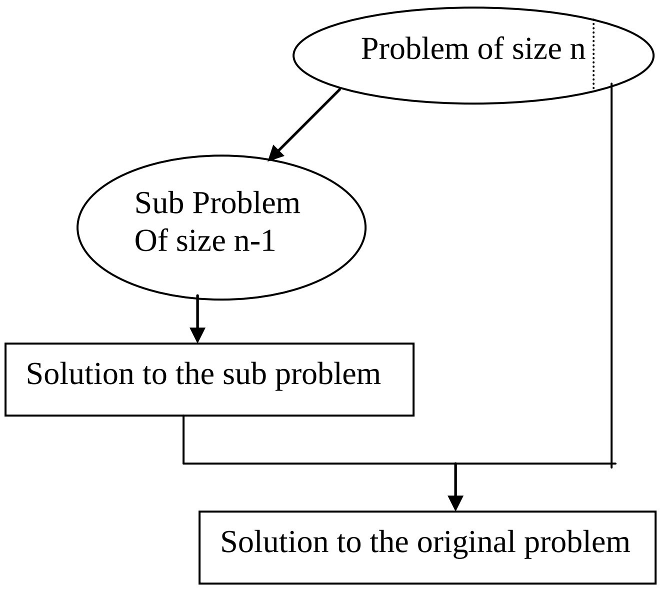


Figure 2.5 Decrease-And-Conquer algorithm

Decrease by a Constant Factor

Decrease-by-a constant-factor decreases the instance size by half (or some other fraction), e.g., 210 = 25 ∗ 25.

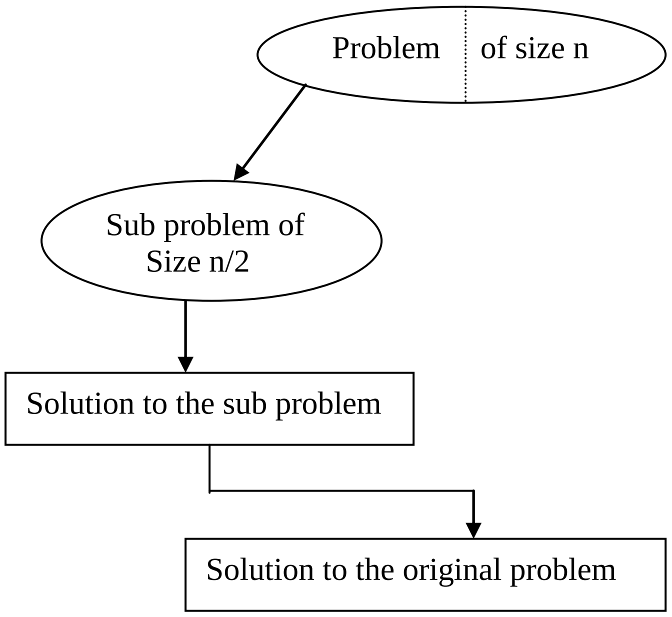


Figure 2.6 Decrease by a constant factor

Comments on Insertion Sort

- Insertion sort ensures A[0] ≤ A[1] ≤ . . . ≤ A[i − 1].
- Insertion sort looks for correct position for A[i].

- Insertion sort shifts values at and above correct position.
- Worst Case: The number of comparisons

$$\sum_{i=1}^{n-1} i = n(n-1)/2 \in O(n^2).$$

- Best Case: $n - 1 \in \Omega(n)$ comparisons if array is already sorted.
- Average Case ≈ n2/4 comparisons.

2.7.1 Depth-First Search

Graph Traversal

Graph traversal algorithms process all the vertices of a graph in a systematic fashion.

- They are useful for many graph problems such as checking connectivity, checking a cyclicity, connected components, finding articulation points, and topological sorting.
- First all the vertices are marked as unvisited.
- Then an unvisited vertex is selected, marked as visited, and all unvisited vertices reachable from that vertex are marked as visited.
- Repeat above step until all vertices are visited.

Depth-First Search Algorithm

```
1. Algorithm DFS (v)
2. // Recursively visits unvisited vertices from v
3. // Input: Vertex v
4. // Output: Unvisited vertices from v are marked
5. Count ← count + 1
6. Mark v with count
7. For each vertex u adjacent from v do
8. If v is marked with 0
9. DFS (u)
```

2.7.2 Breadth-First Search Algorithm

Breadth-first search is a graph-searching algorithm that begins at the root node and explores all the remaining nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Algorithm (informal)

1. Enqueue the root node.
2. Dequeue a node and examine it.
3. If the element sought is found in this node, quit the search and return a result.
4. Otherwise enqueue any successors (the direct child nodes) that have not yet been examined.
5. If the queue is empty, every node on the graph has been examined -- quit the search and return, "not found".
6. Repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

Algorithm BFS (v)

```
1. // Visits unvisited vertices from v
2. // Input: Vertex v
3. // Output: Unvisited vertices from v are marked
4. Count ← count + 1; mark v with count
5. Initialize a queue with v
6. While the queue is not empty do
7. U ← remove vertex from the queue
8. For each vertex w adjacent from u do
9. If w is marked with 0
10. Count ← count + 1
11. Mark w with count
12. Add w to the queue
```

2.8 Transfer and Conquer Algorithm [1][20]

Transform-and-conquer is an approach to solving a problem by changing an instance to:

- A simpler instance of the same problem, or
- A different representation of the same problem, or
- An instance of a different problem.

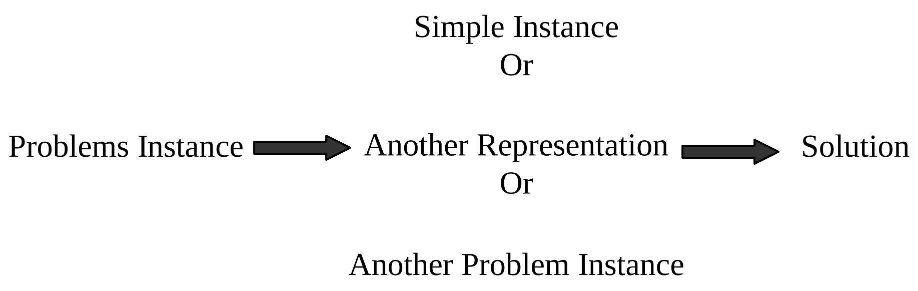


Figure 2.7 Step of Transfer and conquer algorithm

Three kinds of transformation:

1. Instance simplification:

- A more convenient instance of the same problem
- Presorting, uniqueness checking, searching

2. Representational change:

- A different representation of the same instance
- Balanced search trees
- Algorithms in Action; Dr Linda Stern
- Heaps and heap sort

3. Problem reduction:

- A different problem altogether
- Lcm, counting paths, linear programming
- Reductions to graph problems

CHAPTER 3
COMPARISON OF ALGORITHM DESIGN STRATEGIES

Various algorithm design strategies has been compared on the basis of various factors like complexity, memory required, stability etc. This is very important to know about what is complexity of my algorithm in term of time and space. It would be vary harmful to blindly use sorting without considering complexity of algorithm. Comparison of various algorithm design strategy is also depends upon CPU, Memory disk usage and network usage etc. This resources is defined the efficiency of algorithm and performance is depends on the machine, compiler as well as the code. Suppose size of the problem is larger then complexity then it will affect the performance. The time required by a method is proportional to the number of basic operations that it performs

Here are some examples of basic operations:

- Arithmetic operation
- Assignment
- Test
- Read
- Write

Some methods perform the same number of operations every time they are called. For example, consider the size method, of the Sequence class always performs just one operation: return numItems; the number of operations is independent of the size of the sequence, methods like this (that always perform a fixed number of basic operations) require constant time. Other methods may perform different numbers of operations, depending on the value of a parameter or a field.

3.1 Different Notation for Calculating Complexity

To show the complexity of the sorting algorithm in time and space, some asymptotic notations are used. These notations help us predict the best, average and poor behavior of the sorting algorithm. The various notations are as follow:

• Worst Case Running Time

• Best Case Running Time

•Best and Worst case are the same

3.1.1 Big-O Notation

• **Definition:** A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f (n) = O (g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

• **Formal Definition:** f (n) = O (g (n)) means there are positive constants c and k, such that $0 \le f(n) \le cg(n)$ for all $n \ge k$. The values of c and k must be fixed for the function f-and-must-not-depend-on-n [21] [24].

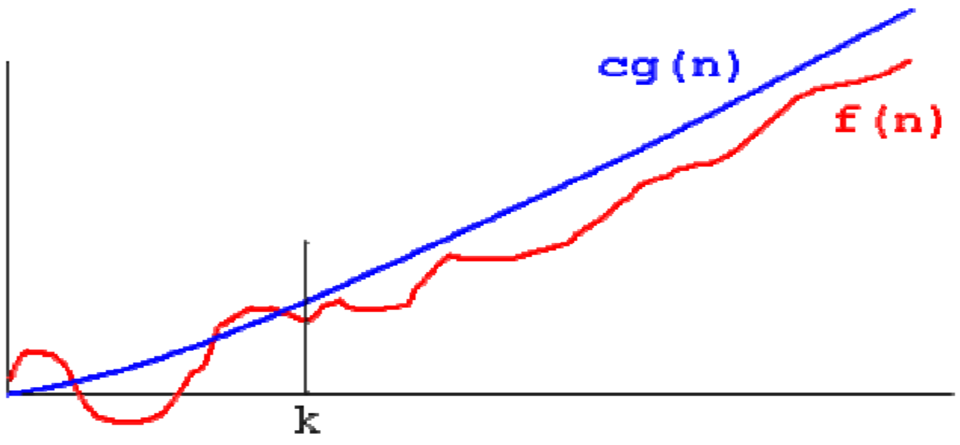


Figure 3.1: Big O Notation Graph

3.1.2 Theta Notation (θ)

• **Definition:** A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f (n) = θ (g(n)) means it is within a constant multiple of g(n). The equation is read, "f of n is theta g of n".

• **Formal Definition:** f(n) = θ (g(n)) means there are positive constants c1, c2, and k, such that 0 _ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ k. The values of c1, c2, and k must be fixed for the function f and must not depend on n [22] [24].

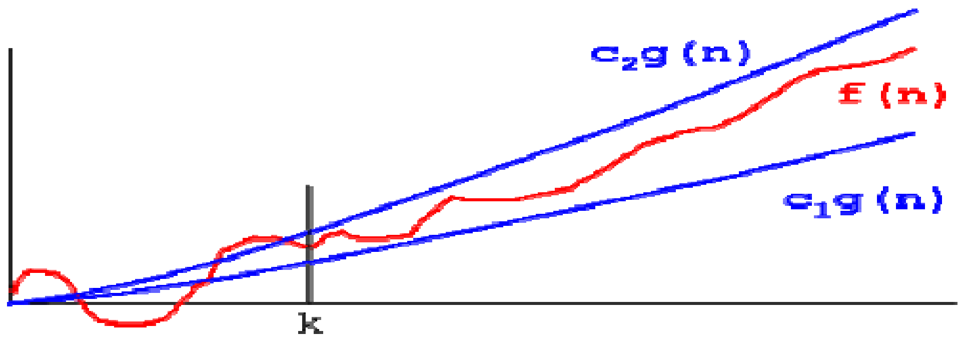


Figure 3.2: Theta Notation Graph
This notation is medium bound indicate what average can happen

3.1.3 Omega Notation (ω)

• **Definition:** A theoretical measure of the execution of algorithms usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f (n) = ω (g (n)) means g (n) becomes insignificant relative to f (n) as n goes to infinity.

• **Formal Definition:** f(n) = ω (g(n)) means that for any positive constant c, there exists a constant k, such that $0 \leq cg(n) < f(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c.
This notation is lower bound indicate what best can happen [23] [24].

3.2 How to Determine Complexities [24][25]

In general, how can you determine the run time of a piece of code? The answer is that it depends on what types of statements are used.

1. Sequence of statements
2. statement 1;
3. statement 2;
4. ...
5. statement k;

Total time = time (statement 1) + time (statement 2) + ... + time (statement k) if each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O (1). In the following examples, assume the statements are simple unless noted otherwise.

6. if-then-else statements

7. if (condition) {

8. sequence of statements 1

9. }

10. else {

11. sequence of statements 2

12. }

Worst-case time is the slowest of the two possibilities: max (time (sequence 1) time (sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

13. for loops

14. for (i = 0; i < N; i++) {

15. sequence of statements

16. }

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

17. Nested loops

18. for (i = 0; i < N; i++) {

19. for (j = 0; j < M; j++) {

20. sequence of statements

21. }

22. }

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O (N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is O (N^2). When a loop is involved, the same rule applies.

For example:

for (j = 0; j < N; j++) g(N);

Has complexity (N^2). The loop executes N times and each time call g (N).

3.3 Best-case and Average-case Complexity [24]

Some methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, if add before is called with a sequence of length N, it may require time proportional to N (to move all of the items and/or to expand the array). This is what happens in the worst case. However, when the current item is the last item in the sequence, and the array is not full, add Before will only have to move one item, so in that case its time is independent of the length of the sequence; i.e., constant time. In general, there is a need to consider the best and average time requirements of a method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a method is part of a time-critical system like One that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average case times for that computation are not relevant - the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain) [25].

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases. For add Before, for a sequence of length N, the worst-case time is O (N), the best-case time is O (1), and the average-case time (assuming that each item is equally likely to be the current item) is O (N), because on average, N/2 items will need to be moved.

Note that calculating the average-case time for a method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

3.4 Advantage and Disadvantage of Design strategy

Design Strategy	**Use**	**Advantage**	**Disadvantage**	**Example**
Brute force	Defence methods (Strong passwords) And Game methods (Chess Game)	Speeding up searches	1. Does not uses any tactics or short cut 2. Enhaustically checks for all notation space	Selection sort, String matching, Exhaustive search
Greedy algorithm	Used for Solving meta heuristic Problem (A meta heuristic is a heuristic method (Heuristics are "rules of thumb", educated guesses, intuitive judgments or simply common sense.) for solving a very general class of computational problems by combining user-given black-box procedures	1. Very large Number of feasible solutions. 2.Easy to implement	1. It is much slower 2. Does not give optimum result for all problems 3. May be receiving	1. Traveling salesman problem 2. Scheduling problem
Divide and conquer	D&C algorithm that was specifically developed for computer and properly analyzed is the merge sort algorithm, invented by John von Neumann in 1945.	1.Solving difficult problems 2. Algorithm efficiency 3. Parallelism 4. Memory access	1. Conceptual difficulty 2. Recursion overhead 3. Repeated sub problems	1. Tower of Hanoi 2. Merge Sort
Dynamic Programming	Used for Solving problems	Does not required repeated calculation	1. Recursive formulation is difficult to make	1. Fibonacci sequence 2. Word wrap

	exhibiting the properties of overlapping sub problems and optimal substructure Multidimensional optimization problem		2. Only for overlapping sub problems	3. Interval scheduling 4. Matrix-chain multiplication problem
Backtracking Algorithm	Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test[26].	1. quick test 2. Pair matching 3. Following real life concept	1. Not widely implemented. 2. Cannot express left-recursive rules 3. More time & complexity	Eight queens puzzle.
Branch and bound	1. Used for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. 2. Branch and bound is a systematic method for solving optimization problems	1. Very large Number of feasible solutions. 2. Tightens the solution space often every step 3. Proming possible	1. Finding proming strategies require clever thinking technologies	1. The Graph Partitioning Problem. 2. The Quadratic Assignment Problem 3. The Symmetric Traveling Salesman problem
Decrease-and-conquer	It is used for Change an instance into one smaller instance of the problem	Solve smaller instance	Depends on Efficiency of sorting.	Binary search Fake-coin puzzle
Transform-and-conquer	Solve a problem's instance by transforming it into another	1. Fast 2. Algorithm efficiency	Not widely implemented	Searching and sorting (Telephone directories in sorted order)

	simpler/easier instance of the same problem			

Table 3.1 Advantage and Disadvantage of Design strategy [4][20][26].

3.5 Type of problems

There are many different problems, being discussed through different algorithm design strategies. Some problems are related to dynamic programming, optimization problem, hard problem etc. some problems are based on one or more type of algorithm design strategies need to find which algorithm strategies is best for a particular problem as well as suitable examples for a each problem. Some useful guidelines are available regarding the suitability of a particular technique to a problem, then a lot of time can be saved and algorithms may be developed only in that technique method**.**

Type of Problem	**Algorithm Strategies**	**Example**
• Multi-branched recursion. • Hard Problems • Sharing repeated sub problems • Overlapping sub problems • Optimal substructure • Memorization	-Divide-and-conquer algorithms are naturally implemented as recursive procedures. It is solve the conceptual and optimization problem by caching sub problem solutions (memorization) rather than recomputing them - It is provide a natural way to design efficient algorithm. -The dynamic programming algorithm is suitable for the observe the dependency of the sub problem	Fibonacci numbers, Towers of Hanoi, The Halting Problem, geometric curves, Closest-Points Merge sort
• Optimization problems • Heuristic problem • Interval Scheduling	-Brute force Is a straightforward approach. -This is directly based on The problem's statement and definitions of the concepts.	Selection sort, String matching, Convex-hull problem, and Exhaustive

	-Greedy algorithms can run faster than brute force ones. - It is not always greedy strategy tell the correct solution.	search, Traveling salesman problem
• Combinatorial optimization problems	- Backtracking depends on user-given black box procedures. - Backtracking is a better approach than brute force (Independently evaluating all possible solutions)[27].	Calculate the path (route)(Example the Traveling Salesman Problem, Minimum Spanning Tree Problem, N Queens, Time and space complexity - Useful when problem size is small - Integer linear programs (ILPs) problems
• Representation problem	Transform and Conquer algorithm basically change one instance to another instance of the problems so this type of the problem basically suitable for the transform and conquer algorithm.	Heap sort, gaussian elimination, hashing, search trees
• Global optimization problem • Test-Cover Problem	- The branch and bound strategy divides a problem to be solved into a number of sub problems, similar to the strategy backtracking. - Branch and bound algorithm is Sometimes we can tell that a particular branch will not lead to an optimal solution:	Travelling salesman problem

	- The partial solution may already be infeasible - Already have another solution that is guaranteed to be better than any descendant of the given solution	

Table 3.2 This table shown the types of the problems and define the algorithms and Example of algorithms [1][4][20].

3.6 Characteristic of problems

Before choosing a best algorithm design strategies for a problem there is a need to know the characteristics of that problem. Those characteristics will be the basis to choose its strategy. To solve any problem the problem’s characteristics must be defined and what is purpose of choosing such algorithm strategy for that particular problem. i.e. (objective of selection).moreover what is use of this strategy for a problem. This is a first step to solve any problems. Some useful guidelines are available regarding the suitability of a particular technique to a problem, then a lot of time can be saved and algorithms may be developed only in that technique method.

Type of problem	**Characteristic**	**Purpose of use (Objective of Selection)**
Multi-branched recursion problem	• Complete task solve by combining solutions to sub-tasks. • Decompose a complete task into smaller, simpler sub-tasks that are similar • Thus, each sub-task can be solved by applying a similar technique • The base case is the smallest problem that the routine solves and the value is returned to the calling method	It's better when you can guarantee this things: 1) each recursive step breaks down the problem into a smaller problem of the same type. 2) Each recursive step reduces the problem significantly. 3) Less memory required 4) Distinct sub-problems can be executed on different processors (Parallelism)

	[28]. • Calling a method involves certain overhead in transferring the control to the beginning of the method and in storing the information of the return point [28]. • Memory is used to store all the intermediate arguments and return values on the internal stack [28].	5) Recursive algorithm, there is considerable freedom in the choice of the base cases, the small sub problems that are solved directly in order to terminate the recursion.
Memorization problem	• Memorization is a technique used to speed up computer programs by storing the results of functions • Memorization is a reduce the power consumption and increase the performance. • Memorization is a characteristic of dynamic programming. • Functions can only be memorized if they are referentially transparent that is, if they will always return the same result given the same arguments. • Memorization does not change the values returned by a function. It only changes the performance characteristics of the function.	1) Memorizing is a technique that can come in handy in programming situations where you're performing a calculation that has input, and the same input always yields the same result. 2) Memorize - Make functions faster by trading space for time
Optimization	• We are working on real-	1) Optimization is the

problems	world, large-scale, hard optimization problems • Mostly Optimization problem is handle the mixed-integer and nonlinear programming problems. • Optimization problem is responsible for a bounded ness, linearity, convexity and monotonicity • Optimization problem is providing the feasible domain (Convexity) using the solving the problem. • Optimization problem is use of mathematical strategies to search for a optimum combinations. • Design optimization as systematic design improvement.	collective process of finding the set of conditions required to achieve the best result from a given situation for a certain objective 2) It is a very powerful technique for solving allocation problems 3) It is solve a brainstorming problem and lateral thinking problem. 4) Optimization techniques in PROC CALIS will find the correct solution.
Heuristic problem	• These methods in most cases employ experimentation and trial-and-error techniques • Heuristics are rules of thumb. • Heuristics are a way to improve time for determining an exact or approximate solution for NP-problems. • Heuristics are a way to improve time for determining an exact or approximate	1) A heuristic method is particularly used to rapidly come to a solution that is reasonably close to the best possible answer, or optimal solution. 2) Heuristic is control information according to the problem solving in human beings and machines 3) Heuristics are a way to improve time for

	solution for NP-problems.	determining an exact or approximate solution for problems.
Interval scheduling problems	• Interval scheduling problems, also known as fixed job scheduling or k-track assignment problems [29]. • Interval scheduling problems is that each job has a finite number of fixed processing intervals [29]. • These problems arise naturally in different real-life operations planning situations, including the assignment of transports to loading/unloading terminals, work planning for personnel, computer wiring, bandwidth allocation of communication channels, printed circuit board manufacturing, gene identification and examining computer memory structures. • Show its relations to cognate problems in graph theory, and survey existing models, results on computational complexity and solution algorithms.	1) A simple flow problem formulation permits minimizing maximum lateness for the more general multimachine case. 2) Performance measures here can focus on the individual jobs; for instance, one may wish to maximize the total weight of the accepted jobs. 3) Interval scheduling allow taking into account the cost of rejecting (or the profit of accepting) an individual job. 4) It is used in real-time operating systems.
Activity Selection Problem	• The main problem for action selection is complexity. • all computation takes both	1) Find Optimal scheduling of unit time jobs with deadlines and penalties for

	time and memory, agents cannot possibly consider every option available to them at every instant in time • The action selection mechanism determines not only the agent's actions in terms of impact on the world, but also directs its perceptual attention, and updates its memory.	missing the deadline. 2) The activity-selection problem is to select the Maximum number of mutually compatible activities.

Table 3.3 Characteristic of the problems [28][29][30]

3.7 Details of Applicable Algorithms

This table represents the list of problems and which algorithm strategies are applicable for it. There are some examples which are related to multi recursion problems, optimization problems etc. some problems are solved by one or more algorithm strategies.

S.No	Problem	Applicable Algorithms						
		A	B	C	D	E	F	G
1.	Fibonacci numbers	√	√					
2.	Towers of Hanoi,	√	√					
3.	The Halting Problem	√	√					
4.	Merge sort	√	√					
5.	Selection sort,			√	√			
6.	String matching			√	√			
7.	Exhaustive search			√	√			
8.	Traveling salesman problem			√	√			√
9.	Minimum Spanning Tree Problem					√		
10.	Integer linear programs (ILPs) problems					√		
11.	Heap sort,						√	

12.	Hashing, search trees						√	
13.	Gaussian elimination						√	

Table 3.4 Details of Applicable Algorithms.

A= Divide and Conquer Algorithm

B= Dynamic programming Algorithm

C=Brute force Algorithm

D=Greedy algorithm

E=Backtracking Algorithm

F= Transform and Conquer algorithm

G=Branch and bound algorithm

CHAPTER 4
PROBLEM STATEMENT

There exist a number of algorithms, every algorithm is problem specific. The choice of an algorithm may not just depend on computational complexity; it also depends upon the characteristics, advantages and disadvantages. This report shows how an algorithm is best for a particular situation, based upon their advantages and comparison with others. The problem of choosing the best algorithm design strategy arises frequently in a computer programming. How one can predict an algorithm is best for a particular problem? What makes a good design strategy for an algorithm? Speed is probably the top consideration, but other factors of interest includes versatility in handling various data types, consistency of performance, memory requirements, length and complexity of code, and the property of stability.

There are some advantages and disadvantages in every algorithm design strategy, which are known and this disadvantage leads to various algorithm design strategy to solve a particular problem. Some algorithm design strategies are problem specific means they are well suited for some specific problem and have disadvantage against another problem. One objective is that, after applying different strategies for a particular problem, a set of guidelines can be given that how a particular category of algorithm design strategy is better for a particular set of problems.

CHAPTER 5
RESULT & CONCLUSION

5.1 Different Algorithm Design Strategies to solve the Problems

Several design technique are applied to a single problem. These design technique is Brute Force, Dynamic Programming, Branch and Bound, Greedy Algorithms, divide and conquer, backtracking, decrease and conquer and transfer and conquer algorithm. This design technique to solve the different Problem. The main goal of this report is to compare the results of these algorithms and find the best one.

5.1.1. The Knapsack Problem

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a best solution from among many other solutions. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items .

Different Design Strategies

Algorithm Design Strategies	
Brute Force	• It will be 2^n possible combinations of items for the knapsack. • It is used for small instance of the knapsack problem. • It does not require much programming effort. • It can be represented as tree format.
Dynamic Programming	• Dynamic programming algorithm to derive a recurrence relation that expresses a solution to an instance of the problem in terms of solutions to its smaller instances [18]. • It does not require any additional structures.

Greedy Algorithm	• Greedy programming techniques are used in optimization problems. • Possible greedy strategies to the 0/1 Knapsack problem: o First of all choose maximum value from the remaining items and increases the value of the knapsack. o Select the lightest item from the remaining items, which uses up capacity as slowly as possible allowing more items to be stuffed in the knapsack. o Select the items with as high a value per weight as possible. • We implement and test all strategies. We got the best results is select the items with as high value-to-weight ratios as possible.
Branch and Bound	• This approach solves some large instances of difficult combinatorial problems. • Branch and bound is based on the state space tree. • In the worst case, the branch and bound algorithm will generate all intermediate stages and all leaves • The tree will be complete and will have $2^{n-1} – 1$ nodes, i.e. will have an exponential complexity. • It is better than the brute force algorithm because on average it will not generate all possible nodes. • The required memory based on the length of the priority queue.
Backtracking	• It is based on item weights and values, find the combination of items to include in the knapsack that will maximize the value, subject to a weight limitation.

	• The current value of the partial knapsack probably cannot be used. • The development without a full branch-and-bound implementation. • Backtracking would be much more effective if we had even more items or a smaller knapsack capacity [31].

Table 5.1.1 Different Algorithm Design Strategies to solve the 0/1 Knapsack Problem

For the comparison of the different algorithm design technique, files of different sizes are generated. There are two type of comparison.

- Increasing the number of items to the knapsack
- Increasing the capacity of the knapsack

These constraints related to number of item and capacity.

5.1.2. The Traveling Sales Man problem

The traveling salesman problem is considered the most prominent unsolved combinatorial optimization problems and to be sure, the best that existing solution methods can do is to handle relatively small traveling sales man problem or large problems with special methods.

Different Design Strategies

Algorithm Design Strategies	
Brute Force	• Seems to be the obvious solution. • Computationally expensive- turns out to be O (n!). • The brute-force method is to simply generate all possible routes and compare the distances. • The time required to come up with a solution is n!
Dynamic Programming	• Dynamic Programming Algorithm solves the respective problem in only O ($n^2 2^n$). • Dynamic-programming algorithm for solving

	Traveling Sales man Problem with a special type of precedence constraints. • We have applied our procedure to solving Traveling Sales man Problem with time, scheduling problems, release and delivery times, in delivery problems, and in routing.
Greedy Algorithm	• It is based on Kruskal's algorithm. It only gives a sub optimal solution in general [32]. • Works for complete graphs. May not work for a graph that is not complete. • As in Kruskal's algorithm, first sort the edges in the increasing order of weights. • Starting with the least cost edge, look at the edges one by one and select an edge only if the edge, together with already selected edges, 1. Does not cause a vertex to have degree three or more. 2. Does not form a cycle, unless the number of selected edges equals the number of vertices in the graph.
Branch and Bound	• An enhancement of backtracking. • The branch-and-bound algorithm does not limit us to any particular way of traversing the tree. • It is used only for optimization problems. • The backtracking algorithm requires the using of DFS traversal and is used for non-optimization problems
Backtracking	• Backtracking is a general technique for organizing the exhaustive search for a solution to a combinatorial problem. • The backtracking technique can be applied to those problems that exhibit the domino principle.

	• If a constraint is not satisfied by a partial solution, the constraint will not be satisfied by any extension of the partial solution to a global solution.
Heuristic Algorithm	• It is often called as a difficult problem. • Traveling cost is the minimum. • We are not aware of any other quick algorithm that finds a best solution we will use a heuristic algorithm. • Heuristic Algorithm solves the respective problem in only N^2

Table 5.1.2 Different Algorithm Design Strategies to solve The Traveling Sales Man problem**.**

Comparison of dynamic-programming algorithm, heuristic algorithm, brute force, greedy algorithm branch and bound algorithm and backtracking for solving TSPs with a precedence constraint. These constraints related to delivery time, scheduling, routing.

5.1.3 The Closest pair of point’s problem

The closest pair of point’s problem or closest pair problem is a problem of computational geometry. Find a pair of points with the smallest distance between them. Algorithm of finding distances between all pairs of points and selecting the minimum requires $O(dn^2)$ time. It turns out that the problem may be solved in O(n log n) time. The optimality follows from the observation that the element uniqueness problem (with the lower bound of Ω(n log n) for time complexity) is reducible to the closest pair problem: checking whether the minimal distance is 0 after the solving of the closest pair problem answers the question whether there are two coinciding points.

Different Design Strategies

Algorithm Design Strategies	
Divide & Conquer	• Divide the problem into two equal sized sub problems

	• Solve those sub problems recursively • Merge the sub problem solutions into an overall solution and hence takes O (nlogn) time. • Divide: Sort the points by x- coordinate; draw vertical line to have roughly n/2 points on each side. • Conquer: find closest pair in each side recursively. • Combine: Find closest pair with one point in each side.
Brute Force	• The closest pair of points can easily be computed in $O(n^2)$ time • To do that, one could compute the distances between all the n(n-1)/2 pairs of points, then pick the pair with the smallest distance.
Branch and Bound	• Select good branching. • Store the information in a stack format. • Not effective, because data is stored in different location. • It is very difficult monitor of the data • User facing the Leakage memory problem.
Backtracking	• The closest pair of points problem asks for the minimal number of tests needed to uniquely identify a disease infection.

Heuristic Algorithm	• It is more flexible design systems but not guarantee that the solution found is optimal. • It is a efficient and flexible • It is able to produce an acceptable solution to a problem in many practical scenarios but for which there is no formal proof of its correctness. • In practical problems, a heuristic algorithm may be the only way to get good solutions in a reasonable amount of time.

Table 5.1.3 Different Algorithm Design Strategies to solve The Closest pair of point's problem.

5.1.4 The N-Puzzles Problem

The N-puzzle problem provides a good framework for describing a concept of AI. This concept is related to the various uninformed and informed search algorithms. This is usually applied in this setting and their performance is evaluated.

Different Design Strategies

Algorithm Design Strategies	
Brute Force	• Brute-force approach to solving problems in • Explicitly • Implicitly • Combinatorial objects such as permutations, combinations, and subsets of a given set. It suggests generating all the elements of the problem's domain and then finding a desired element (e.g., the one that optimizes a given objective function). • In fact, many puzzles can provide good examples of problems that either cannot be solved by brute force at all, or for which this strategy yields a very clumsy and unsatisfactory solution.

	• Puzzles that can be solved by brute force, one can suggest, for example, getting the 3-by-3 magic square by exhaustive search. It provides a good illustration of the limitations of exhaustive search and the usefulness of knowing an algorithm's efficiency class.
Divide-and-conquer	• Few puzzles solvable by the divide-and-conquer approach. Here are two examples that are rather well known. o The first one is the triomino puzzle. o The other problem is the nuts-and-bolts problem. • Divide-and-conquer is based on partitioning. • Solving each of them recursively, and then combining their solutions to get a solution to the original problem.

Decrease-and-conquer	• The decrease-by-a-constant variety suggests decreasing a problem's size by a constant.
	• This approach is considered by some to be a special case of divide-and conquer, it is better to consider them as distinct design strategies.
	• The crucial difference between the two lies in the number of smaller sub problems that need to be solved: several (usually, two) in divide-and-conquer algorithms and just one in decrease-and-conquer algorithms. It is further useful, both from the design and the analysis perspectives, to distinguish three varieties of this strategy
	  o Decrease-by-a-constant variety
	  o Decrease-by-a-constant-factor,
	  o Variable-size

Transform-and-conquer	The last most general technique is based on the idea of transformation.

	• Its first variety called instance simplification. • The Second variety called Representation change. • The third variety of the transformation strategy is problem reduction.

Table 5.1.4 Different Algorithm Design Strategies to solve The N-Puzzles Problem.

N-Puzzles can be very helpful for different algorithm design technique. Two type of algorithm design techniques are considered

- The Most general algorithm design techniques like: brute force, divide and-conquer, decrease-and-conquer, and transform-and conquer and
- Less general techniques like: greedy, dynamic programming, backtracking, and branch-and-bound.

According to my research for future more puzzles suitable for algorithm design technique and analysis of algorithms will be found in will be found in existing collections or specifically designed for this worthy purpose

The objective of the analysis in these tables is that if a new problem arises then based on the inherent characteristics of the problem, it can be categorized in to particular category and then right algorithm can be written. Using the given strategies some odd problems were taken from different sources and the result was, ability to figure out exact strategies to be used for 67% of the problems in the first instance. For 9% of the problems, two different strategies were tried to get the efficient algorithm; because the characteristics of these problem does not exactly points to a particular strategy. The remaining 24% could not be categorized into any of the above categories or they were looking similar to multiple categories. So initially it is a good to start and the research will continues further to improve these results so that more problems can be categorized and solved in first instance.

ANNEXURE I
REFERENCES

[1] Annay Levitin, “Do We Teach the Right Algorithm Design Techniques?” Technical Symposium on Computer Science Education. The proceedings of the thirtieth SIGCSE technical symposium on Computer science education, ISBN: 1-58113-085-6 pp 179- 183 (1999).

[2] Michael T. Goodrich,Roberto Tamassia, “Algorithm Design Foundations,Analysis and Internet Examples”.
http://ww3.algorithmdesign.net/ch00-front.html

[3] Javier Galve Frances, Julio Garcia Martin, Jose M. Burgos Ortiz, Miguel Sutil Martin, “An Approach to Algorithm Design by Patterns”
http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP1998/1998_Galve-FrancesEtAl_AlgorithmDesignByPatterns.pdf.

[4] Fundamentals of computer algorithms by Ellis Horowitz Sartaj Sahni Sanguthevar Rajasekaran 2005.

[5] Algorithm Correctness www.cs.wm.edu/~coppit/csci243-fall2002/AlgCor1.pdf

[6] Frank M. Carrano, Janet J. Prichard. Data Abstraction and problem solving with java. Addison Wesley Longman, 2001
http://jaireggeton.googlepages.com/correct.pdf

[7] Michael J. Jacobson, “Correctness of Algorithms” (CPSC 331, Winter 2007) University of Calgary Canada.
http://pages.cpsc.ucalgary.ca/~jacobs/Courses/cpsc331/W07/topics/correct ness.html

[8] Types of Algorithms http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/ 33-algorithm-types.ppt

[9] Christian Charras, ”Brute Force Algorithm”, 1997.
http://www-igm.univ-mlv.fr/%7Elecroq/string/node3.html#SECTION0030

[10] Fawzi Emad, Chau-Wen Tseng ,“Algorithm Strategies” University of Maryland.
http://www.cs.umd.edu/class/spring2005/cmsc132/lecs/lec34.ppt

[11] algorithmics.comp.nus.edu.sg/wiki/_.../algorithm_design.ppt?id.

[12] Data structures and algorithm analysis in C by Mark Allen Weiss.

[13] www.rocw.raifoundation.org/computing/BCA/.../lecture-15.pdf

[14] Algorithm Design Strategies
www.cs.purdue.edu/homes/ayg/CS490B/lec1.pdf

[15] Anastas Misev, "Algorithmic Patterns Data Structures and Algorithms in Java".
http://perun.im.ns.ac.yu/java/workshops/Algorithmic-patterns.pdf

[16] Fawzi Emad, Chau-Wen Tseng, " Algorithm Strategies"
www.cs.umd.edu/class/spring2005/cmsc132/lecs/lec34.ppt

[17] http://homepages.ius.edu/rwisman/C455/html/notes/Backtracking/
BranchandBound.htm

[18] Decrease and Conquer Concept Asst. Prof. Dr. Bunyarit Uyyanonvara
Thammasat University.
www.siit.tu.ac.th/bunyarit/.../ITS033x06xDecreasexConquer.ppt

[19] B.B. Karki, LSU, " Decrease- and – Conquer"
www.csc.lsu.edu/~karki/DA-08/DA16.pdf

[20] Cormen, Leiserson And Rivest, Introduction To Algorithms, McGraw Hill And Mit Press, 1990, 329-333.

[21] http://www.itl.nist.gov/div897/sqg/dads/HTML/bigOnotation.html big-o-notation 29 March 2009.

[22] http://www.nist.gov/dads/HTML/theta.html theta, accessed on 29 March 2009.

[23] http://www.nist.gov/dads/HTML/omegaCapital.html, accessed on 29 March 09

[24] Pandey, Ramesh Chand, Goel, Shivani, "Study and Comparison of various sorting", Algorithms, Thapar University July 2008.

[25] http://pages.cs.wisc.edu/~hasti/cs367-common/notes/COMPLEXITY.html,
Accessed on 29 April 2009.

[26] Gurari, Eitan, Backtracking algorithms CIS 680: DATA STRUCTURES (1999).

[27] Backtracking Algorithms
www.cs.rpi.edu/~hollingd/psics/notes/backtracking.pdf

[28] Recursion (Winter 2004-5), http://www2.latech.edu/~box/ds/chap6.ppt

[29] Interval Scheduling, Frits C.R. Spieksma, Katholieke Universiteit Leuven, Naamsestraat 69, B-3000 Leuven, Belgium, frits.
www.mistaconference.org/2007/papers/Interval%20Scheduling.pdf

[30] De Sevin, E. Thalmann, D.A motivational Model of Action Selection for Virtual Humans. In: Computer Graphics International (CGI), IEEE Computer Society Press, New York (2005).

[31] Brute Force Approach
www.cse.msu.edu/~torng/Classes/Archives/cse830.../Lecture11.ppt

[32] A Greedy Algorithm for Traveling Sales Man Problem.
http://lcm.csa.iisc.ernet.in/dsa/node186.html

[33] Hristakeva, Maya and Dipti Shrestha. “Solving the 0/1 Knapsack Problem” MICS Proceedings 2004.

[34] Cengiz Erbas, Seyed sarkeshik, Murat M. Tanik, “Different perspectives of the N-queens problem**”,** ACM Annual Computer Science Conference Proceedings of the 1992 ACM annual conference on Communications,
ISBN:0-89791-472-4 pp: 99 – 108.

[35] Anany Levitin, Mary-Angela Papalaskari, “Using Puzzles in Teaching Algorithms” ACM SIGCSE Bulletin**,** Volume 34, Issue 1 (March 2002) ISSN:0097-8418, pp 292 – 296.

ANNEXURE II
LIST OF PUBLICATIONS

[1] Shailendra Nigam, Dr. Deepak Garg "Choosing Best Algorithm Design Strategies For a Particular Problem", In Proceedings of the IEEE International Advance Computing Conference (IACC 09), Thapar University Patiala, India (6-7 March 2009).