B.E.

Fifth Semester Examination, May-2007

Analysis & Design of Algorithms (CSE-305-E)

Note : Attempt any *five* questions.

Q. 1. (a) What do you understand by Analysis of Algorithm? Justify its need.

Ans. An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's CPU to perform operations & its memory to hold the program & data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time & storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgements about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist. Questions such as how well does an algorithm perform is the best case, in the worst case, or on the average are typical. For each algorithm in the text, an analysis is also given.

Q. 1. (b) What is meant by searching & sorting? Discuss different methods (any two) used for searching and sorting. Also discuss their efficiency in terms of time and space.

Ans. Searching : Searching is the process to find out or retrieve a value or record from a table.

For searching we are having,

1. Linear search
2. Binary search

1. Linear Search : In this technique, we start at a beginning of a list or a table & search for the desired record by examining each subsequent record until either the desired record is found or the list is exhausted.

2. Binary Search : Linear search is a simple & easy method. It is efficient for small lists but highly inefficient for large list.

To search a particular item with a certain key value target the approxinate middle entry of the table is located, & its key value is examined. If its value is higher than the target, the key value of the middle entry of the first half of the list is examined & the procedure is repeated on the first half until the required item is found. If the value is lower than the target, the key value of the middle entry of second half of the table is taken & the procedure is repeated on the second half. This process continues until the required key is found or the search intervals became empty.

Sorting : Is the operation of arranging the records of a table according to the key value of each record.

Bubble Sort : The basic idea underlying the bubble sort is to pass through the table sequentially several times. Each pass puts the largest unsorted element in its correct place by compairing each element in the table with its successor & interchanging the two elements if they are not in proper order.

Q. 2. Discuss Merge sort and quick sort Algorithms. "Merge sort is good choice instead of quick sort." Do you agree with statement? Justify your answer.

Ans. Merge Sort : Given a sequence of n elements (also called keys) $a[1],\ldots,a[n]$, the general idea is to imagine them split into two sets $a[1],\ldots,a[n/2]$ & $a[[n/2]+1],\ldots,a[n]$. Each set is individually sorted, & the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Algorithm for merge sort is given below :

```
Algorithm mergesort (low, high)
        // a [low : high] is a global array to be sorted.
        // small (p) is true if there is only one element.
        // to sort. In this case the list is already
        //sorted.
  {
        if (low < high) then // If there are more than
        // one element.
  {     //Divide P into subproblems.
        // Find where to split the set
              mid : = [(low + high)/2];
        // Solve the subproblems:
              Mergesort (low, mid);
              Mergsort (mid + 1, high);
        // Combine the solutions.
              Merge (low, mid, high);
        }
  }.
```

Quicksort : In quicksort, the divison into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \le a[j]$ for all i between i & m & all j between m + 1 & n for some $m_1 i \le m \le n$. Thus, the elements in a [i : m] & $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of a [ ], say t = a[s], & then reordering the other elements. So that all elements appearing before t in a [1: n] are less than or equal to t & all elements appearing after t are greater than or equal to t. This rearranging is referred to as partitioning function partition accomplishes an in place partitioning of the elements of a [m : p – 1]. It is assumed that $a[p] \ge a[m]$ and that a [m] is the partitioning element. The function interchange (a, i, j) exchanges a [i] with a[j].

```
    Algorithm Quicksort (p, q)
        // sorts the elements a [p], ... a [q] which reside in the global array a [1 : n] into ascending order; a[n+1]
        is considered to be defined & must be ≥ all the elements in a [1 : n].
  {
   if (p < q) then // If there are more than one element .
   {     // divide p into two subproblems.
                J : = partition (a, p, q + 1);
         // J is the position of the partitioning element.
         // Solve the subproblems
                Quicksort (p, j – 1);
                Quicksort (j + 1, q);
         // There is no need for combining solutions
         }
   }.
```

/

Quicksort & Mergesort were evaluated on a SUN workstation 10/30. In both cases the recursive versions were used. For Quicksort the partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of a [m], a [[m + p–1]/[2]] and a [p–1]. Each data set consisted of random integers in the range (0. 1000). Tables (1) & (2) record the actual computing time in milliseconds. Table (1) displays the average computing times. For each n, 50 random data sets were used. Table (2) shows the worst case computing times for the 50 data sets.

Scanning the tables we immediately see that Quicksort is faster than Mergesort for all values. Even though both algorithms require O(n log n) time on average. Quicksort usually performs well in practice.

n	1000	2000	3000	4000	5000
Mergesort	72.8	167.2	275.1	378.5	500.6
Quicksort	36.6	85.1	138.9	205–7	269.0
n	6000	7000	8000	9000	10000
Mergesort	607.6	723.4	811.5	949.2	1073.6
Quicksort	339.4	411.0	487.7	556.3	645.2

Fig. 1. Average computing times for two sorting algorithms on random inputs.

n	1000	2000	3000	4000	5000
Mergesort	105.7	206.4	335.2	422.1	589.9
Quicksort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
Mergesort	691.3	794.8	889.5	1067.2	1167.6
Quicksort	383.8	497.3	569.9	616.2	738.1

Fig. 2. Worst case computing times for two sorting algorithms an random inputs

Q. 3. State single source shortest path problem. Write an algorithm based on greedy method to obtain solution to shortest path problem. Also discuss How running time is affected by representation of Graph.

Ans. Single Source Shortest Path Problem: Graphs can be used to represent the highway structure of a state or country with vertices representing cities & edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions :

1. Is there a path from A to B.
2. If there is more than one path from A to B, which is the shortest path?

The problems defined by these questions are special cases of the path problems. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source & the last vertex the destination. The graphs are digraphs to allow for one way streets. In the problem we consider, we are given a directed graph G = (v, E), a weighting function cost for the edges of G & a source vertex V_0. The problem is to determine the shortest paths from V_0 to all the remaining vertices of G.

It is assumed that all the weights are positive. The shortest path between V_0 & some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Greedy algorithm to generate shortest path,

```
//Algorithm shortest paths (v, cost, dist, n)
//dist [3], $1 \le J \le n$, is set to the length of
```

```
        // the shortest path from vertex v to vertex}
        // in a digraph G with n vertices. dist [v]
        // is set to zero G is represented by tis cost
        // adjacency matrix cost [1:n, 1:n].
    {
            for i: = 1 to n do
    {   // Initialize S
            S [i]: = false; dist [i] : = cost [v, i];
    }
            S [v] : = true; dist [v]: = 0.0; // put v in s for num : = 2 to n–1 do
    {
        // Determine n–1 paths from v.
        Choose u from among those vertices not in S such that dist [u] is minimum;
            S [u]: = true; //put u in s
            for (each w adjacent to u with S[w] = false) do
        //update distances
        if (dist [w] > dist [u] + cost [u, w])) then
        dist [w] : = dist [u] + cost [u, w];
    }
  }.
```

The time taken by the algorithm on a graph with n vertices is $O\left(n^2\right)$. Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency materices were used to represent the graph, it takes $\left(O\left(n^2\right)\right)$ time just to determine which edges are in G, & so any shortest path algorithm using this representation must take $\Omega\left(n^2\right)$ time.

Q. 4. (a) Discuss the Dynamic programming technique. Also discuss the main difference between greedy method & dynamic programming with the help of example.

Ans. Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. A dynamic programming algorithm solves every sub subproblem just once and then saves its answer in a table, there by avoiding the work of recomputing the answer every time the subsubproblem is encountered. The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this

choice will lead to a globally optimal solution. This explores optimization problems that are soluble by greedy algorithm. Greedy algorithms do not always yield optimal solutions. but for many problems they do. The greedy method is quite powerful and works well for a wide range of problems.

Q. 4. (b) State O/I Knapsack problem. Apply Dynamic programming technique to solve this problem.

Ans O/I Knapsack : A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n, we may be in one of two possible states : the capacity remaining in the knapsack is m and no profit has accrued or the capacity remaining is $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principle of optimality holds.

Let $f_j(y)$ be the value of an optimal solution to KNAP (i, j, y). Since the principle of optimality holds, we obtain.

$$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \qquad \ldots(1)$$

For arbitrary $f_i(y)$, $i > 0$, equation (1) generalizes to

$$f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \qquad \ldots(2)$$

Equation (2) can be solved for $fn(m)$ by beginning with the knowledge $fo(y) = 0$ for all y and $fi(y) = -\infty, y < 0$.

Then $f_1, f_2, \ldots, f_n$ can be successively computed using (2).

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \le y \le m$. Since $f_i(y) = -\infty$ for y < 0, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\theta(m)$ time, it takes $\theta(mn)$ time to compute f_n. When the w_i's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \le y \le m$. So f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore an alternative method for both cases. Notice that $f_i(y)$ is an ascending step function, i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \ge y_k$; and $f_i(y) = f_i(y_j)$, $y_i \le y < y_j + 1$.

So, we need to compute only $f_i(y_j)$, $1 \le j \le k$. We use the ordered set $S^i = \{(f(y_j), y_i) / 1 \le j \le k\}$ to represent $f_i(y)$. Each member of S^i is a pair (p, w), where $P = f_i(y_j)$ and $w = y_j$. Notice that $S^0 = \{(0,0)\}$.

We can compute S^{i+1} from S^i by first computing.

$$S_1^i = \{(p, w) / (p - p_i, w - w_i) \varepsilon S^i\} \quad ...(3)$$

Now S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (p_j, w_j) and (p_k, w_k) with the property that $p_j \le p_k$ and $w_j \ge w_k$, then the pair (p_j, w_j) can be discarded because of (2). Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above (p_k, w_k) dominates (p_j, w_j). Interestingly, the strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^n possibilities for $x_1, x_2, ..., x_n$. Let S^i represent the possible states resulting from the 2^i decision sequences for $x_1 .... x_i$. A state refers to a pair (p_j, w_j), w_j being the total weight of objects included in the knapsack and p_j being the corresponding profit. To obtain S^{i+1}, we note that the possibilities for x_{i+1}, are $x_{i+1=0}$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i. When $x_{i+1} = 1$ the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i call the set of these additional states S_1^i. The S_1^i is the same as in equation (3). Now S^{i+1} can be computed by merging the states in S^i and S_1^i together.

Q. 5. (a) What do you understand by Back tracking? Discuss.

Ans. Backtracking represents one of the most general technique. In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, ..., x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes a criterion function $p(x_1, ..., x_n)$. Sometimes it seeks all vectors that satisfy P. For example, sorting the array of integers in a [1 : n [is a problem whose solution is expressible by an n-tuple where x_i is the index in a of the ith smallest element. The criterion function P is the inequality $a[x_i] \le a[x_{i+1}]$ for $1 \le i < n$. The set S_i is finite and includes the integers 1 through n. Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n-tuple.

Q. 5. (b) What are Hamiltonian cycles? Determine the order of Magnitude of the worst-case running time for the back tracking procedure that finds all Hamiltonian cycles.

Ans. Hamiltonian Cycles : Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at same vertex $V_1 \in G$ and the vertices of G are visited in the order $V_1, V_2, ..., V_{n+1}$ then the edges (V_i, V_{i+1}) are in E, $1 \le i \le n$, and the V_i are distinct except for V_1 and V_{n+1}, which are equal.

```
Algorithm Hamiltonian (K).
// This algorithm uses the recursive formulation of.
//Back-tracking to find all the Hamiltonian cycles.
```

```
// of a graph. The graph is stored as an adjacency.
//Matrix G [1 : n, 1 : n]. All cycles begin at node 1.
{
    Repeat
    {//Generate values for x [k].
        Next value (k); // Assign a legal next value to x [k].
        If (x [k] = 0) then return;
        If (k = n) then write (x [1:n]);
        Else Hamiltonian (k + 1);
    } Until (false);
}
```

Q. 6. (a) Explain Branch & Bound Design strategy with the help of example.

Ans. The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. We have already seen two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First-In-First-Out) search as the list of live nodes is a first-in-first-out list (or queue). A D-search like state space search will be called.

LIFO (Last-In-First-Out) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

Q. 6. (b) Consider the following traveling sales person instance defined by the cost Matrix :

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

Obtain the reduced cost matrix.

Ans.

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

/

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Reduced upto

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Q. 7. Discuss following :

(a) NP Word Problems, **(b) NP Complete Problems,**

(c) Cook's theorem, **(d) Satisfiability.**

Ans. (a) & (b) -NP Word Problems and NP complete Problems : The theory of NP-completeness which we present here does not provide a method of obtaining polynomial time algorithms for problems in the second group. Nor does it say that algorithms of this complexity do not exist. Instead, what we do is show that many of the problems for which there are no known polynomial time algorithms are computationally related. In fact, we establish two classes of problems. These are given the names NP-hard and NP-complete. A problem that is NP-complete has the property that it can be solved in polynomial time if and only if all other NP-complete problems can also be solved in polynomial time. If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. All NP-complete problems are NP-hard, but some NP-hard problems are not known to be NP-complete. Although one can define many distinct problem classes having the properties stated above for the NP-hard and NP-complete classes, the classes we study are related to nondeterministic computations (to be defined later). The relationship of these classes to nondeterministic computations together with the apparent power of nondeterminism leads to the intuitive (though as yet unproved) conclusion that no NP-complete or NP-hard problem is polynomially solvable.

(c) Cook's Theorem : Cook's theorem states that satisfiability is in P if and only if P = NP. We now prove this important theorem. We have already seen that satistiability is in NP. Hence if P= NP, then satisfiability is in P. It remains to be shown that if satisfiability is in P, then P = NP. To do this, we show how to obtain from any polynomial time nondeterministic decision algorithm A and input I a formula Q (A, I) such that Q is satisfiable iff A has a successful termination with input I. If the length & I is n and the time complexity of A is p(n) for some polynomial P ( ), then the length of Q is $O\left(P^3(n)\log n\right) = O\left(p^4(n)\right)$. The time needed to construct Q is also $= O\left(p^3(n)\log n\right)$. A deterministic algorithm 2 to determine the outcome of A on any input I can be easily obtained. Algorithm z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether Q is satisfiable. If $O\left(q(m)\right)$ is the time needed to determine whether a formula of length m is satisfiable, then the complexity of z is $O\left(p^3(n)\log n + q\left(p^3(n)\log n\right)\right)$. If satisfiability is in P, then q (m) is satisfiability is in p, then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P. So, the above construction shows that if satisfiability is in P, then P = NP.

(d) Satisfiability : Let $x_1, x_2 .....$ denote boolean variables (their value is either true or false). Let $\overline{x}_i$ denote the negation of x_i. A literal is either variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations and or. Example of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x}_y)$ and $(x_3 \vee \overline{x}_4) \wedge (x_1 \vee \overline{x}_2)$. The symbol $\vee$ denote or and $\wedge$ denotes and A formula is in conjunctive normal form (CNF) if and only if it is represented as $\wedge_{i=1}^{k} e_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in disjunctive normal form (DNF) if and only if it is represented as $\vee_{i=1}^{k} C_i$ and each clause C_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x}_4)$ is in DNF whereas $(x_3 \vee \overline{x}_4) \wedge (x_1 \vee \overline{x}_2)$ is in CNF. The satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables. CNF-Satisfiability is the satisfiability problem for CNF formulas. It is easy to obtain a polynomial time nondeterministic algorithm that terminates successfully if and only if a given propositional formula $E(x_1, ..., x_n)$ is satisfiable. Such an algorithm could proceed by simply choosing (nondeterministically) one of the 2^n possible assignment of truth values to $(x_1, ..., x_n)$ and verifying that $E(x_1, ..., x_n)$ is true for that assignment.

Q. 8. (a) Discuss different techniques for algebraic problems.

Ans. A system that allows for the manipulation of mathematical expressions (usually including arbitrary precision integers, polynomials and rational functions) is called a mathematical symbol manipulation system. These system have been truthfully used to solve a variety of scientific problems for many years. The techniques we study here have often led to efficient ways to implement the operations offered by these systems. The first design technique we present is called algebraic transformation. Assume we have an input I that is a memebr of set S_1 and a function F (I) that describes what must be computed. Usually the output f (I) is also a members of S_1. Though a method may exist for computing f (I) using operations on elements in S_1, this method may be inefficient. The algebraic transformation technique suggests that we alter the input into another form to produce a member of set S_2. The set S_2 contains exactly the same elements as S_1 except it assumes a different representation for them. Why would be transform the input into another form? Because it may be easier to compute the function f for elements of S_2 then for elements of S_1. Once the answer in S_2 is computed, an inverse transformation is performed to yield the result in set S_1.

Q. 8. (b) What do you understand by optimal Binary Search Tree? Justify significance of OBST with the help of suitable example.

Ans. To apply dynamic programming to the problem of obtaining an optimal binary search tree, we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i's should be assigned to the root node of the tree. If we choose a_k, then it is clear that the internal nods for $a_1, a_2, ..., a_{k-1}$ as well as the external nodes for the classes, $E_0, E_1, \ldots E_{k-1}$ will be lie in the left subtree L of the root. The remaining nodes will be in the right subtree r. Define,

$$\mathrm{cost}(l) = \sum_{1\le i<k} p(i)*\mathrm{level}(a_i) + \sum_{0\le i<k} q(i)*(\mathrm{level}(E_i)-1)$$

And

$$\mathrm{cost}(r) = \sum_{k<i\le} p(i)*\mathrm{level}(a_i) + \sum_{k<i\le n} q(i)*(\mathrm{level}(E_i)-1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

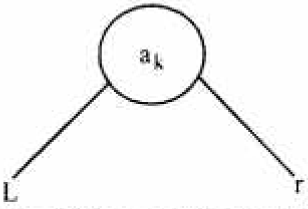

Fig. An optimal binary search tree with root a_k

Using w (i, j) to represent the sum $q(i) + \sum_{l=i+1}^{j} (q(i) + p(l))$, we obtain the following as the expected cost of the search tree (Figure 1.)

$$p(k) + \mathrm{cost}(l) + \mathrm{cost}(r) + w(0, k-1) + w(k, n) \qquad \ldots(1)$$

If the tree is optimal, then (1) must be minimum. Hence cos t (l) must be minimum over all binary search trees containing $a_1, a_2, \ldots, a_{k-1}$ and $E_0, E_1, \ldots E_{k-1}$. Similarly cost (r) must be minimum. If we use C (i, j) to represent the cost of an optimal binary search tree $t_{i,j}$ containing $a_{i+1}, \ldots, a_j$ and $E_i, \ldots, E_j$, then for the tree to be optimal. We must have $\mathrm{cost}(l) = c(0, k-1)$ and $\mathrm{cost}(r) = c(k, n)$. In addition, k must be chosen such that,

$p(k) + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n)$ is minimum. Hence for $c(0, n)$ we obtain

$$c(0, n) = \min_{1\le k<n} \{c(0, k-1) + c[k, n]\} + p(k) + w(0, k-1) + w(k, n) \qquad \ldots(2)$$

We can generalize (2) to obtain for any c (i, j)

$$c(i, j) = \min_{i<k\le j} \{c(i, k-1) + c[k, j]\} + p(k) + w(i, k-1) + w(k, j)$$

$$c(i, j) = \min_{i<k\le j} \{c(i, k-1) + c(k, j) + w(i, j)\} \qquad \ldots(3)$$

Equation (3) can be solved for c (0, n) by first computing all c (i, j) such that $j - i = 1$ (note $c(i, i) = 0$ and w (i, i) = q(i), $0 \le i \le n$). Next we can compute all c (i, j) such that $j - i = 2$, then all $c(i, j)$ with $j - i = 3$ and so on. If during this computation we record the root $r(i, j)$ of each tree t_{ij}, then an optimal binary search tree can be constructed from these $r(i, j)$. Note that r (i, j) is the value of k that minimizes.