(A0510194) DESIGN AND ANALYSIS OF ALGORITHMS

UNIT IV

UNIT-IV:

Dynamic Programming: General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design, optimal binary search tree.

Dynamic Programming:

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.

0/1 knapsack problem:

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 xi are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n, we may be on 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 $x_n, x_{n-1}, \ldots, x_1$ will not be optimal.

The value of the optimal solution with n objects and with knapsack capacity m is

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

In general

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

If we use the ordered set $S^i = \{ (f_i(y_j),y_j) \mid 1 \le j \le k \}$ to represent $f_i(y)$. Each member of S^i is a pair (P,W), where $P = f_i(y_j)$ and $W = y_j$. Note that $S^0 = \{(0, 0)\}$.

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

$$S_1^i = \{ (P,W) \mid (P-p_i , W-w_i) \in S^i \}$$

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. This is called purging rule.

Eg: Consider the knapsack instance n = 3, (w_1, w_2, w_3) = (2, 3, 4), (P_1, P_2, P_3) = (1, 2, 5), and m = 6.

$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

Note that the pair (3, 5) has been eliminated from S^3 as a result of the Purging rule.

With m = 6, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3. The tuple (6, 6) $\notin$ S^2, and so we must set x_3 = 1. The pair (6, 6) came from the pair (6-P_3, 6-w_3) = (1,2). Hence(1,2) $\in S^2$. Since (1,2) $\in S^1$, we can set x_2 = 0. Since (1,2) $\notin S^0$, we obtain x_1=1. Hence an optimal solution is (x_1, x_2, x_3)= (1,0,1).

Informal Knapsack Algorithm:

```
Algorithm DKP(p, w, n, m)
{
    S^0 := {(0,0)};
    for i := 1 to n − 1 do
    {
        S_1^{i−1} := {(P,W)|(P − p_i, W − w_i) ∈ S^{i−1} and W ≤ m};
        S^i := MergePurge(S^{i−1}, S_1^{i−1});
    }
    (PX, WX) :=last pair in S^{n−1};
    (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
        any pair in S^{n−1} such that W + w_n ≤ m;
    // Trace back for x_n, x_{n−1}, ..., x_1.
    if (PX > PY) then x_n := 0;
    else x_n := 1;
    TraceBackFor(x_{n−1}, ..., x_1);
}
```

All pairs shortest path problem:

Let G = (V, E) be a directed graph with n vertices. Let cost be a cost Adjacency matrix for G such that cost(i,i)= 0, 1≤ i ≤ n. Then cost(i,j) is the length (or cost) of edge <i,j> if <i,j> ∈ E(G) and cost(i,j)= ∞ if i ≠ j and (i,j) ∉ E(G). The all-pairs shortest-path problem is to determine a matrix A such that A(i,j) is the length of a shortest path from i to j.

Let us examine a shortest i to j path in G, i ≠ j. This path originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. We can assume that this path does not contain any cycles. If k is an intermediate vertex on this shortest path, then the sub paths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length.

If k is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than k-1. Similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than k-1.

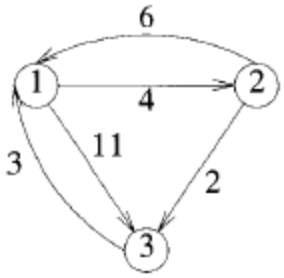
Using $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain

$$A(i,j) = \min \{ \min_{1 \le k \le n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\}$$

Clearly, $A^0(i,j)$ = cost(i,j), 1≤ i ≤ n, 1≤ j ≤ n. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$.

If it does not; then no intermediate vertex has index greater than k-1. Hence $A^k,(i,j)= A^{k-1}(i,j)$.Combining, we get

$$A^k(i,j) = \min \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\quad k \ge 1$$



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// n vertices; $A[i, j]$ is the cost of a shortest path from vertex
// i to vertex j. $cost[i, i] = 0.0$, for $1 \leq i \leq n$.
{
    for $i := 1$ **to** n **do**
        for $j := 1$ **to** n **do**
            $A[i, j] := cost[i, j]$; // Copy $cost$ into A.
    for $k := 1$ **to** n **do**
        for $i := 1$ **to** n **do**
            for $j := 1$ **to** n **do**
                $A[i, j] := \min(A[i, j], A[i, k] + A[k, j])$;
}

Travelling Sales person problem:

Let G = (V,E) be a directed graph with edge costs C_{ij}. The variable C_{ij} is defined such that $C_{ij} > 0$ for all i and j and $C_{ij} = \infty$ if $(i,j) \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost.

Assume that the tour starts and ends at vertex 1. Every tour consists of an edge (l,k) for some $k \in V- \{1\}$ and a path from vertex k to vertex 1.Thepath from vertex k to vertex1goes through each vertex in V- {1} exactly once. It is easy to see that if the tour is optimal, then the path from k to 1must be a shortest k to 1path going through all vertices in V- {l,k}. Hence, the principle of optimality holds.

Let g(i,S) be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1.The function g(l,V-{1}) is the length of an optimal sales person tour.

From the principle of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

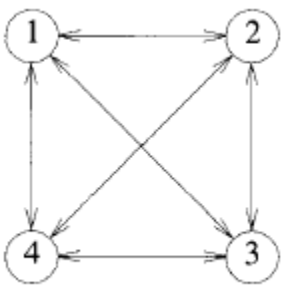
In general for $i \notin S$,

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

$g(i, \Phi) = C_{i1}$, $1 \le i \le n$.

Then we can obtain g(i, S) for all S of size1.Then we can obtain g(i, S) for S with $|S| = 2$, and so on.

Eg:



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

$$g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6, \text{ and } g(4,\phi) = c_{41} = 8.$$

$$\begin{array}{lllll} g(2,\{3\}) & = & c_{23} + g(3,\phi) & = & 15 \\ g(3,\{2\}) & = & 18 & & \\ g(4,\{2\}) & = & 13 & & \end{array} \qquad \begin{array}{lll} g(2,\{4\}) & = & 18 \\ g(3,\{4\}) & = & 20 \\ g(4,\{3\}) & = & 15 \end{array}$$

$$\begin{array}{lllll} g(2,\{3,4\}) & = & \min\ \{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} & = & 25 \\ g(3,\{2,4\}) & = & \min\ \{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} & = & 25 \\ g(4,\{2,3\}) & = & \min\ \{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} & = & 23 \end{array}$$

$$\begin{aligned} g(1,\{2,3,4\}) &= \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

An optimal tour of the graph has length 35. Let J(i,S) be the value of j that minimizes g(i,S).

J(1, {2,3,4}) = 2

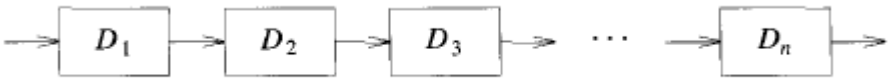
J(2, {3,4}) = 4

J(4,{3}) =3

So, the optimal tour is 1, 2, 4, 3, 1.

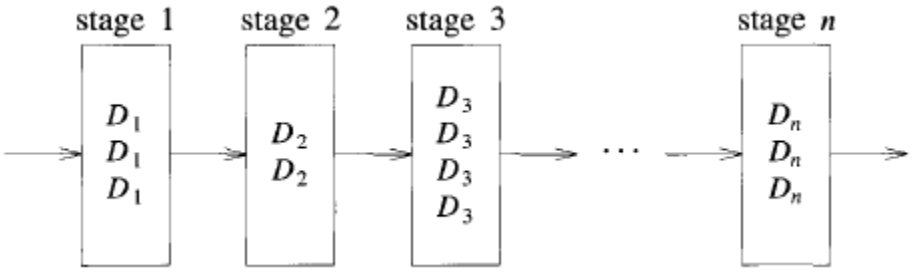
Reliability Design Problem:

The problem is to design a system that is composed of several devices connected in series.



Let r_i be the reliability of device D_i (the probability that device i will function properly). Then, the reliability of the entire system is πr_i. Even if the individual devices are very reliable, the reliability of the system may not be very good. For example, if n = 10 and r_i = 0.99,$1 \le i \le 10$,then πr_i = 0.904.

Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel through the use of switching circuits.



The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1- r_i)^{mi}$. Hence the reliability of stage i becomes $1-(1- r_i)^{mi}$. Thus, if r_i = 0.99 and m_i = 2, the stage reliability becomes 0.9999.

Let us assume that the reliability of stage i is given by a function $\Phi_i(m_i)$ $1 \le i \le n$, The reliability of the system of stages is $\pi_{1 \le i \le n} \Phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c, be the cost of each unit of device i and let C be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem

$$\text{maximize } \Pi_{1 \le i \le n}\ \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer, } 1 \le i \le n$$

Assume that each $C_i > 0$, so each m_i must be in the range $1 < m_i < u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$$

Let $f_i(x)$ represent the maximum value of $\pi_{1 \le j \le i} \Phi(m_j)$ subject to the constraints $\sum_{1 \le j \le i} C_j m_j \le x$ ad $1 < m_j < u_j$, $1 \le j \le i$.

Then, the value of an optimal solution is $f_n(C)$. The last decision made requires to choose m_n from {1, 2,3,... , u_n}. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $C-c_nm_n$ in an optimal way.

$$f_n(c) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\}$$

In general

$$f_i(x) = \max_{1 \le m_i \le u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\}$$

Clearly, $f_0(x) = 1$ for all x, $0 \le x \le c$.

Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$. The dominance rule (f_1,x_1) dominates (f_2,x_2) iff $f_1 \ge f_2$ and $x_1 \le x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i.

Eg: $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $C = 105$, $r_1 = 0.9$, $r_2 = 0.8$, $r_3 = 0.5$

Find the maximum possible devices in each stage using $u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$

u1=2, u2= 3, and u3 = 3.

$S^0 = \{(1,0)\}$

We can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together.

We can use S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$.

$S_1^1 = \{(0.9, 30)\}$

$S_2^1 = \{(0.99, 60)\}$ $S^1 = \{(0.9, 30), (0.99, 60)\}$

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$

$S_2^2 = \{(0.864, 60)\}$

Note that the tuple (0.9504, 90) which comes from (0.99,60) has been eliminated from S_2^2 as this leaves only 15. This is not enough to allow m_3=1.

$S_3^2 = \{(0.8928, 75)\}$

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$ as the tuple (0.792, 75) is dominated by (0.864, 60).

$S_1^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.54, 85), (0.648, 100)\}$

$S_3^3 = \{(0.63, 105)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$.

The best design has a reliability of 0.648and a cost of 100.

Tracing back through S^i's, we determine that m_1=1, m_2=2 and m_3=2.

OBST (Optimal Binary Search Tree):

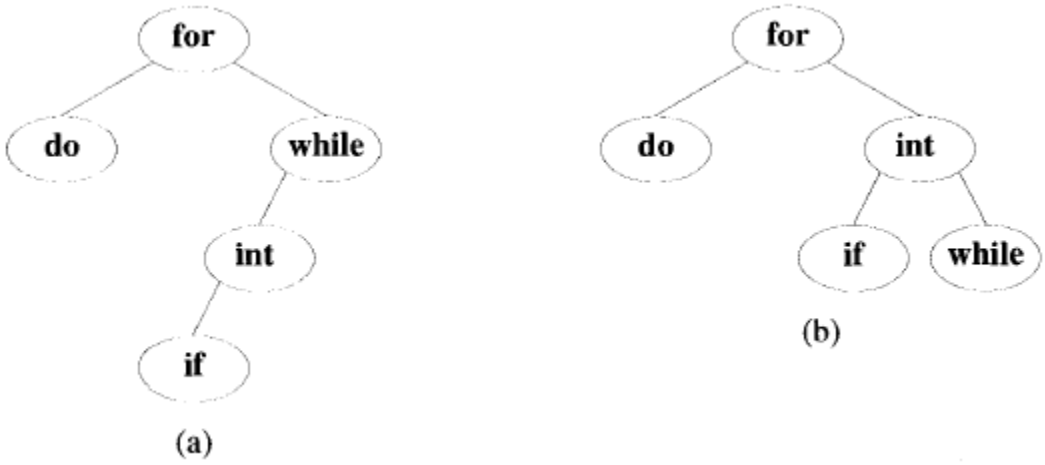
Binary Search Tree:

A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier, and

i. All identifiers in the left sub tree of T are less than the identifiers in the root node T
ii. All identifiers in the right sub tree of T are greater than the identifiers in the root node T
iii. The left and right sub trees of T are also binary search trees.

To determine whether an identifier X is present in a binary search tree, X is compared with the root. If X is less than the identifier in the root, then the search continues in the left sub tree. If X equals the identifier in the root, the search terminates successfully, Otherwise the search continues in the right sub tree.

Consider the following two binary search trees:



The tree of Figure (a) in the worst case requires four comparisons to find an identifier, whereas the tree of Figure (b) requires only three. On the average the two trees need 12/5 and 11/5 comparisons respectively.

This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches are made.

In a general situation, we can expect different identifiers to be searched for with different frequencies (or probabilities). In addition, we can expect unsuccessful searches also to be made.

Let us assume that the given set of identifiers is $\{a_1, a_2, a_3 \ldots a_n\}$ with $a_1 < a_2 < a_3 < \ldots < a_n$. Let p(i) be the probability with which we search for a_i. Let q(i) be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \le i \le n$. (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$).

Then, $\sum_{0 \le i \le n} q(i)$ the probability of an unsuccessful search.

$$\sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} q(i) = 1.$$

Given this data we wish to construct an optimal binary search tree for $\{a1, a_2, a_3 \ldots a_n\}$.

In obtaining a cost function for binary search trees, it is useful to add an external node in place of every empty sub tree in the search tree. All other nodes are internal nodes. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and n +1external nodes. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

If a successful search terminates at an internal node at level l, then the expected cost contribution from the internal node for a_i is p(i) * level(a_i).

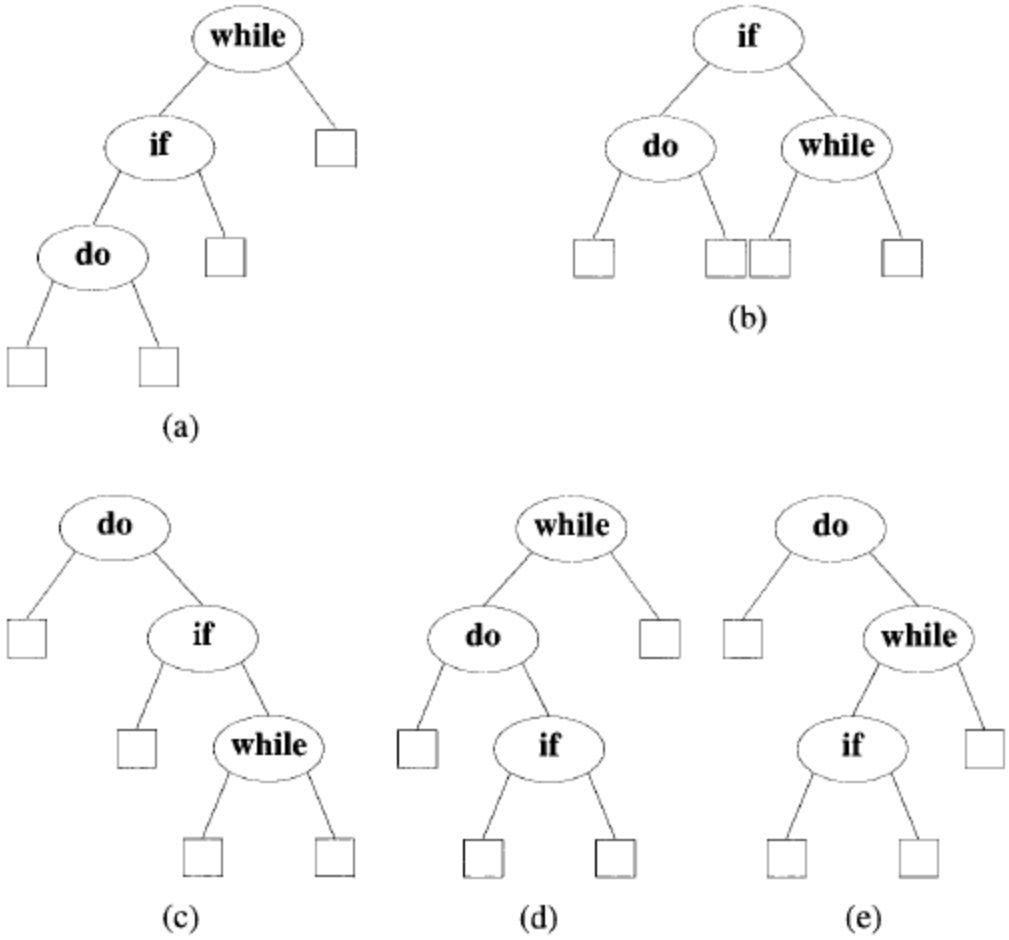
The identifiers not in the binary search tree can be partitioned into n + 1 equivalence classes E_i, $0 \le i \le n$. The class E_0 contains all identifiers x such that $x < a_1$. The class E1 contains all identifiers x such that ai < x < ai+1, $1 \le i < n$. The class E_n contains all identifiers x, $x > a_n$.

If the failure node for E_i is at level l, then the cost contribution of this node is q(i) * (level(E_i)-1).

So the expected cost of a binary search tree is

$\sum_{1 \le i \le n} p(i)$* level($a_i$) $+\sum_{0 \le i \le n} q(i)$ * (level(E_i)-1).

Eg: (a1, a2, a3) = (do, if, while)



With equal probabilities p(i) = q(i) = 1/7 for all i, we have

cost(tree a) = 15/7
cost(tree b) = 13/7
cost(tree c) = 15/7
cost(tree d) = 15/7
cost(tree e) = 15/7

As expected, tree b is optimal.

With p(l)= 0.5, p(2) = 0.1, p(3) = 0.05, q(0)= 0.15, q(1) = 0.1, q(2)= 0.05 and g(3)= 0.05 we have
cost(tree a) = 2.65
cost(tree b) = 1.9
cost(tree c) = 1.5
cost(tree d) = 2.15
cost(tree e) = 1.6 Tree c is optimal.

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 nodes for a_1, a_2, a_3 … a_{k-1} as well as the external nodes for the classes $E_0,E_1,\ldots,E_{k-1}$ will lie in the left sub tree 'l' of the root. The remaining nodes will be in the right sub tree 'r'. So we can define

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

and

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

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

Consider W(i,j) represent the sum

$$q(i) + \sum_{l=i+1}^{j} (q(l) + p(l))$$

we obtain the following as the expected cost of the search tree

$$p(k) + cost(l) + cost(r) + w(0, k-1) + w(k, n) \quad \ldots\ldots (1)$$

If the tree is optimal, then eq(1) must be minimum. Hence, cost(l) must be minimum over all binary search trees containing a_1, a_2, a_3 … a_{k-1} and $E_0,E_1,\ldots,E_{k-1}$. Similarly cost(r) also must be minimum.

If we use c(i,j) to represent the cost of an optimal binary search tree t_{ij} containing a_{i+1} to a_j and $E_i,\ldots,E_j$, then for the tree to be optimal, we must have cost(l) = c(0,k-1) and 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,

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

In general,

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

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

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 ≤ i ≤ 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 eq (3).

Eg: Let n = 4 and (a_1, a_2, a_3, a_4) = (do,if, int,while). Let p(l:4) = (3,3,1,1) and q(0 :4) = (2,3,1,1,1). The p's and q's have been multiplied by 16 for convenience. Initially, we have w(i,i)= q(i), c(i,i)=0 and r(i,i)= 0, 0 ≤ i ≤ 4.

Using W(i,j)=p(j)+<q(j)+w(i,j-1),we get

$$\begin{aligned}
w(0,1) &= p(1) + q(1) + w(0,0) = 8 \\
c(0,1) &= w(0,1) + \min\{c(0,0) + c(1,1)\} = 8 \\
r(0,1) &= 1 \\
w(1,2) &= p(2) + q(2) + w(1,1) = 7 \\
c(1,2) &= w(1,2) + \min\{c(1,1) + c(2,2)\} = 7 \\
r(0,2) &= 2 \\
w(2,3) &= p(3) + q(3) + w(2,2) = 3 \\
c(2,3) &= w(2,3) + \min\{c(2,2) + c(3,3)\} = 3 \\
r(2,3) &= 3 \\
w(3,4) &= p(4) + q(4) + w(3,3) = 3 \\
c(3,4) &= w(3,4) + \min\{c(3,3) + c(4,4)\} = 3 \\
r(3,4) &= 4
\end{aligned}$$

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

The box in row i and column j shows the values of w(j,j+i), c(j,j+i) and r(j,j+i) respectively.

```
Algorithm OBST(p, q, n)
// Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities
// p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
// the cost c[i, j] of optimal binary search trees t_ij for identifiers
// a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
// w[i, j] is the weight of t_ij.
{
    for i := 0 to n − 1 do
    {
        // Initialize.
        w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
        // Optimal trees with one node
        w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
        r[i, i + 1] := i + 1;
        c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
    }
    w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
    for m := 2 to n do  // Find optimal trees with m nodes.
        for i := 0 to n − m do
        {
            j := i + m;
            w[i, j] := w[i, j − 1] + p[j] + q[j];

            k := Find(c, r, i, j);
                // A value of l in the range r[i, j − 1] ≤ l
                // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
            c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
            r[i, j] := k;
        }
    write (c[0, n], w[0, n], r[0, n]);
}

Algorithm Find(c, r, i, j)
{
    min := ∞;
    for m := r[i, j − 1] to r[i + 1, j] do
        if (c[i, m − 1] + c[m, j]) < min then
        {
            min := c[i, m − 1] + c[m, j]; l := m;
        }
    return l;
}
```

The time complexity is $O(n^3)$.