SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

UNIT - IV

Design and Analysis of Algorithm – SCSA1403

}

Greedy Approach and Dynamic Programming 9 Hrs.

Greedy Approach:- Optimal Merge Patterns- Huffman Code - Job Sequencing problem- -- Tree Vertex Splitting Dynamic Programming:– Dice Throw-- Optimal Binary Search Algorithms.

Greedy Approach

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on subsetparadigm.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
        solution := ∞ ;                // initialize the solution to empty for
        i:=1 to n do
        {
                x := select (a);
                if feasible (solution, x) then
                        solution := Union (Solution, x);
        }
        return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from ‘a’, removes it and assigns its value to ‘x’. Feasible is a Boolean valued function, which determines if ‘x’ can be included into the solution vector. The function Union combines ‘x’ with solution and updates the objective function.

Applications of the Greedy Strategy Optimal solutions:

- Change Making For “Normal” Coin Denominations
- Minimum Spanning Tree (Mst)
- Single-Source Shortest Paths
- Simple Scheduling Problems
- Huffman codes

Approximations/heuristics:

- Traveling salesman problem (TSP)
- Knapsack Problem

Optimal Merge Patterns Problem

Given n sorted files, find an optimal way (i.e., requiring the fewest comparisons or record moves) to pair wise merge them into one sorted file. It fits ordering paradigm.

Example

Three sorted files (x_1, x_2, x_3) with lengths (30, 20, 10)

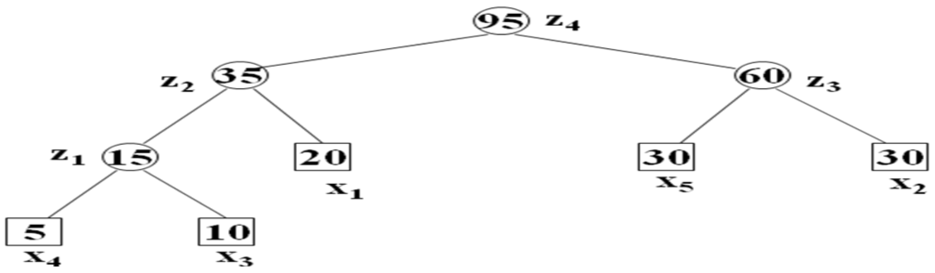
Solution 1: merging x_1 and x_2 (50 record moves), merging the result with x_3 (60 moves) à total 110 moves

Solution 2: merging x_2 and x_3 (30 moves), merging the result with x_1 (60 moves) à total 90 moves

The solution 2 is better.

A greedy method (for 2-way merge problem)

At each step, merge the two smallest files. e.g., five files with lengths (20, 30, 10, 5, 30).

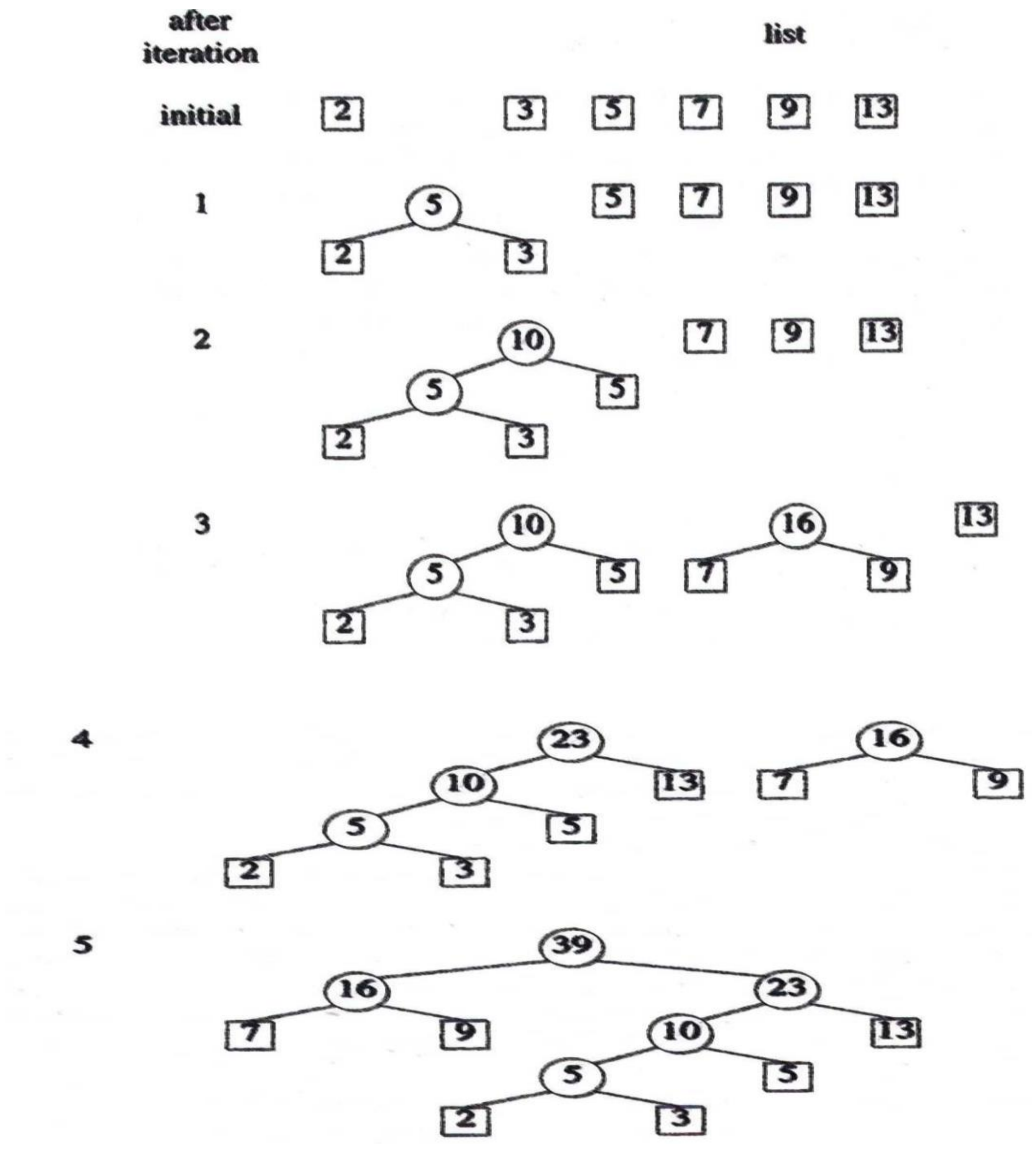


Total number of record moves = weighted external path length

The optimal 2-way merge pattern = binary merge tree with minimum weighted external path length

```
Algorithm struct treenode{
        struct     treenode     *lchild,
        *rchild;int weight;
  };
  typedef    struct    treenode
  Type;Type *Tree(int n)
  //    list is a global list of n single node
  //    binary trees as described above.
  {
        for (int i=1; i<n; i++) {
            Type    *pt    =    new
            Type;
            // Get a new tree node.
            pt -> lchild = Least(list); // Merge two trees
            with pt  ->  rchild  =  Least(list);  //  smallest
            lengths.
            pt -> weight = (pt->lchild)->weight
                    +          (pt->rchild)-
            >weight;Insert(list, *pt);
        }
        return (Least(list)); // Tree left in l is the merge tree.
 }
```

Example:



Time Complexity

If list is kept in non-decreasing order: O (n^2)

If list is represented as a min heap: O (n log n)

Optimal Storage on Tapes

There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length L_i. Assume the tape is initially positioned at the front. If the programs are stored in the order I = i_1, i_2… i_n, the time t_j needed to retrieve program i_j

k

If all programs are retrieved equally often, then the mean retrieval time (MRT) =this problemfits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$D(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} L_{ik}$$

Example

n=3 (l1, l2, l3) = (5, 10, 3) 3! =6 total combinations

L1 l2 l3 = l1+ (l1+l2) + (l1+l2+l3) = 5+15+18 = 38/3=12.6

n 3

L1 l3 l2 = l1 + (l1+l3) + (l1+l2+l3) = 5+8+18 = 31/3=10.3

n 3

L2 l1 l3 = l2 + (l2+l1) + (l2+l1+l3) = 10+15+18 = 43/3=14.3

n 3

L2 l3 l1 = 10+13+18 = 41/3=13.6

3

L3 l1 l2 = 3+8+18 = 29/3=9.6 min

3

L3 l2 l1 = 3+13+18 = 34/3=11.3 min

3 permutations at (3, 1, 2)

Example

n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 1 5, 27) (d_1, d_2, d_3, d_4) = (2, 1, 2, 1)

	Feasible solution	Processing sequence	value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3, 1	115
3	(1,4)	4, 1	127
4	(2,3)	2, 3	25
5	(3,4)	4,3	42

6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Example

Let n = 3, $(L_1, L_2, L_3) = (5, 10, 3)$. 6 possible orderings. The optimal is 3, 1, 2

Ordering I	d(I)
1,2,3	5+5+10+5+10+3 = 38
1,3,2	5+5+3+5+3+10 = 31
2,1,3	10+10+5+10+5+3 = 43
2,3,1	10+10+3+10+3+5 = 41
3,1,2	3+3+5+3+5+10 = 29
3,2,1,	3+3+10+3+10+5 = 34

Huffman Trees and Codes

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.

This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream. Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.
2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree:

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

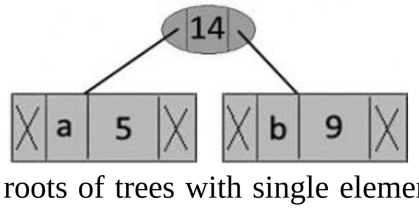
3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

Character	**Frequency**
a	5
b	9
c	12
d	13
e	16
f	45

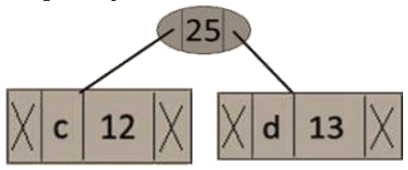
Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2: Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



roots of trees with single element each, and one heap node is root of tree with 3 elements Now min heap contains 5 nodes where 4 nodes are

Character	**Frequency**
c	12
d	13
Internal Node	14
e	16
f	45

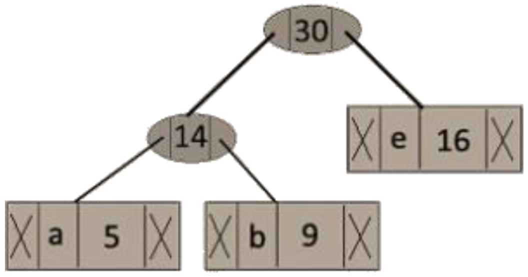
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	**Frequency**
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

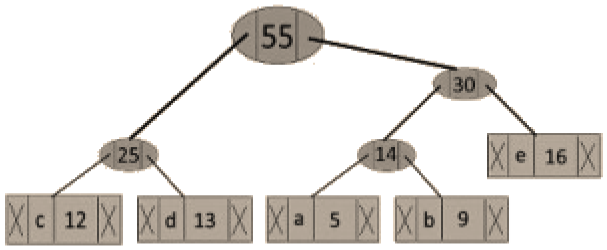


Now min heap contains 3 nodes.

Character	**Frequency**
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency

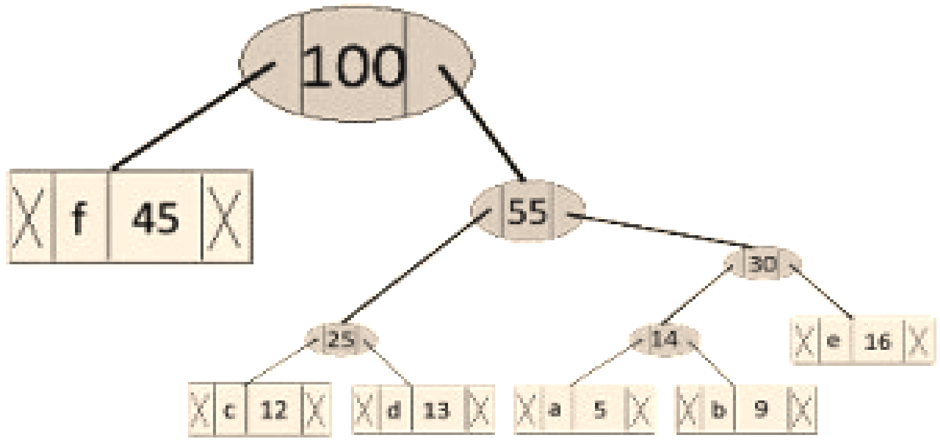
25 + 30 = 55



Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



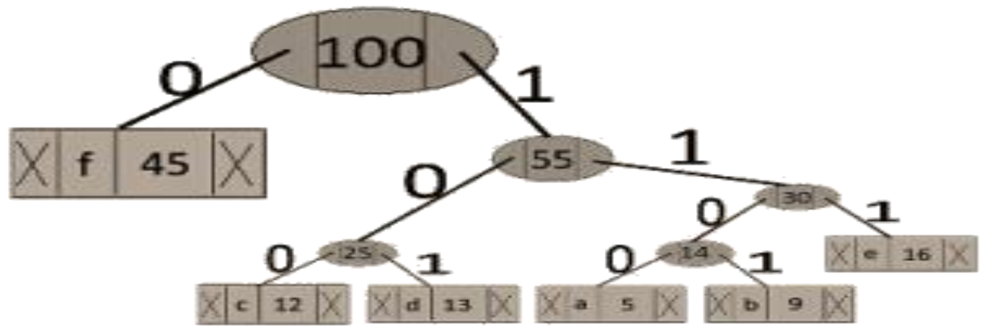
Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Algorithm:

Huffman(n, f)

Huffman Coding Input: Array of numerical frequencies or probabilities.

Output: Binary coding tree with n leaves that has minimum expected code length

```
1. for i := 1 to n do
2. H[i] := i, f(i)
3. create a leaf node labeled i (both chilren are Nil)
4. BuildHeap(H)
5. for i := n + 1 to 2n − 1 do
6. x := ExtractMin(H); y := ExtractMin(H)
7. create a node labeled i with children the nodes labeled x.label and y.label
8. Insert H,(i, x.freq + y.freq)
```

Analysis:

This algorithm runs in O(n log n) time:

Putting the first n pairs into H and creating the n leaves takes O(n) time and turning H into a heap using BuildHeap also takes O(n) time (line 4). The for loop in lines 5–8 is repeated n

− 1 times. In each iteration we perform two ExtractMin operations and one Insert operation, each of which takes O(log n) time.

So the loop takes O(n log n) time, and the entire algorithm takes O(n) + O(n) + O(n log n) = O(n log n) time.

The time complexity of the Huffman algorithm is **O(nlogn)**. Using a heap to store the weight of each tree, each iteration requires **O(logn)** time to determine the cheapest weight and insert the new weight. There are **O(n)** iterations, one for each item.

Job Sequencing Problem

Job sequencing with deadlines the problem is stated as below. There are n jobs to be processed on a machine. Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$. Pi is earned iff the job is completed by its deadline. The job is completed if it is processed on a machine for unit time. Only one machine is available for processing jobs. Only one job is processed at a time on the machine.A given Input set of jobs 1,2,3,4 have sub sets 2^n so 2^4 $^{=16}$ It can be written as {1},{2},{3},{4},{Ø},{1,2},{1,3},{1,4},{2,3},{2,4},{3,4},{1,2,3},

{1,2,4},{2,3,4},{1,2,3,4},{1,3,4} total of 16 subsets

Problem:

n=4 , P=(70,12,18,35) , d=(2,1,2,1)

Feasible Solution	Processing Sequence	Profit value	Time Line 0 1 2
1	1	70	
2	2	12	
3	3	18	
4	4	35	
1,2	2,1	82	
1,3	1,3 /3,1	88	
1,4	4,1	105	
2,3	3,2 /2,3	30	
3,4	4,3/3,4	53	

We should consider the pair i,j where di <=dj if di>dj we should not consider pair then reverse the order. We discard pair (2, 4) because both having same dead line(1,1) and cannot process same. Time and discarded pairs (1,2,3), (2,3,4), (1,2,4)…etc since processes are not completed within their deadlines. A feasible solution is a subset of jobs J such that each job iscompleted by its deadline. An optimal solution is a feasible solution with maximum profit value.

Example

Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1, 2)	(2, 1)	110
(ii)	(1, 3)	(1, 3) or (3, 1)	115
(iii)	(1, 4)	(4, 1)	127 is the optimal one
(iv)	(2, 3)	(2, 3)	25
(v)	(3, 4)	(4, 3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

Problem: n jobs, S = {1, 2… n}, each job i has a deadline d_i □ 0 and a profit p_i □ 0. We needone unit of time to process each job and we can do at most one job each time. We can earnthe profit p_i if job i is completed by its deadline.

The optimal solution = {1, 2, 4}.
The total profit = 20 + 15 + 5 = 40.

i	1	2	3	4	5
p_i	20	15	10	5	1
d_i	2	2	1	3	3

Algorithm

Step 1: Sort p_i into non-increasing order.
After sorting p_1 □ p_2 □ p_3 □ … □ p_i.
Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i totime slot [r-1, r], where r is the largest integer such that 1 □ r □ d_i and [r-1, r] is free.
Step 3: Stop if all jobs are examined. Otherwise, go to step
2.Time complexity: O (n^2)

Example

I	p_i	d_i
1	20	2
2	15	2
3	10	1
4	5	3
5	1	3

assign to [1, 2] assign to [0, 1] Reject assign to [2, 3]Reject
solution = {1, 2, 4}
total profit = 20 + 15 + 5 = 40

Greedy Algorithm to Obtain an Optimal Solution

Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

In the example considered before, the non-increasing profit vector

is(100 27 15 10) (2 1 2 1)

p_1 p_4 p_3 p_2 d_1 d4 d3

d2J = {1} is a feasible one
J = {1, 4} is a feasible one with processing sequence
J = {1, 3, 4} is not feasible
J = {1, 2, 4} is not feasible
J = {1, 4} is optimal
High level description of job sequencing algorithmProcedure greedy job (D, J, n)

```
// J is the set of n jobs to be completed by their deadlines
{
J:={
1};
    for i:=2 to n do
    {
       if (all jobs in J U{i} can be completed by their
      deadlines)then J:= ß J U {i};
    }
}
```

Greedy Algorithm for Sequencing unit time jobs

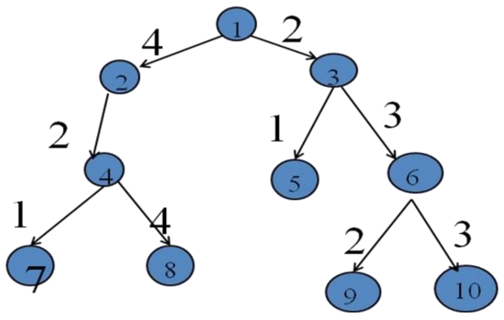
```
Procedure JS(d,j,n)
{
d[0]:=J[0]:=0; //initialize and J(0) is a fictious job with d(0) =
0 //J[1]:=1; //include job 1
K:=1; // job one is inserted into J //
for i :=2 to n do // consider jobs in non increasing order of pi //
r:=k;
While ((d[J[r]]>d[i]) and (d[J[r]]#r)) do r:=r-1;
If ((d[J[r]  d[i]) and d[i]>r)) then { //insert i into
J[]For q:=k to (r+1) step-1 do j[q+1]:=j[q];
J[r+1]:=i; k:=k+1;
} } return k;
}
```

TVSP (Tree Vertex Splitting Problem)

Let T= (V, E, W) be a directed tree. A weighted tree can be used to model a distribution network in which electrical signals are transmitted. Nodes in the tree correspond to receivingstations & edges correspond to transmission lines. In the process of transmission some loss is occurred. Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network model may not able tolerate losses beyond a certain level. In places where the loss exceeds the tolerance value boosters have to be placed. Given a networks and tolerance value, the TVSP problem is to determine an optimal placement of boosters. The boosters can only placed at the nodes of the tree.

d (u) = Max { d(v) + w(Parent(u), u)}

d(u) – delay of node v-set of all edges & v belongs to child(u)δ tolerance value



If d (u)>= δ than place the booster.d (7)= max{0+w(4,7)}=1
d (8)=max{0+w(4,8)}=4
d (9)= max{0+ w(6,9)}=2
d (10)= max{0+w(6,10)}=3 d(5)=max{0+e(3.3)}=1
d (4)= max{1+w(2,4), 4+w(2,4)}=max{1+2,4+3}=6> δ ->booster d (6)=max{2+w(3,6),3+w(3,6)}=max{2+3,3+3}=6> δ->booster
d (2)=max{6+w(1,2)}=max{6+4)=10> δ->booster
d (3)=max{1+w(1,3), 6+w(1,3)}=max{3,8}=8> δ ->booster
Note: No need to find tolerance value for node 1 because from source only power is transmitting.

Dynamic Programming

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by a mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub problems.

There are two ways of doing this.

1.) Top-down: Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitively. This is referred to as Memorization.

2.) Bottom-up: Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial sub problem, up towards the given problem. In this process, it is guaranteed that the sub problems are solved before solving the problem. This is referred to as Dynamic Programming.

Dice Throw

Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.

The **Naive approach** is to find all the possible combinations of values from n dice and keep on counting the results that sum to X.

This problem can be efficiently solved using **Dynamic Programming (DP)**.

Let the function to find X from n dice is: Sum(m, n, X)

The function can be represented as:

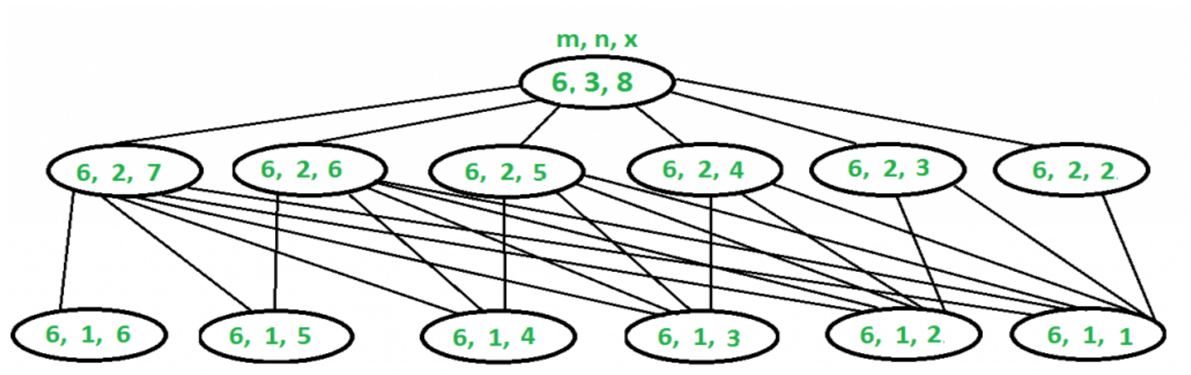
```
Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth dice
               + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth dice
               + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth dice
                  ...................................................
                  ...................................................
                  ...................................................
              + Finding Sum (X - m) from (n - 1) dice plus m from nth dice
```

So we can recursively write Sum(m, n, x) as following

```
Sum(m, n, X) = Sum(m, n - 1, X - 1) +
               Sum(m, n - 1, X - 2) +
               .................... +
               Sum(m, n - 1, X - m)
```

Why DP approach?

The above problem exhibits overlapping subproblems. See the below diagram. Also, see this recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:



Sum(6, 3, 8) = Sum(6, 2, 7) + Sum(6, 2, 6) + Sum(6, 2, 5) +
Sum(6, 2, 4) + Sum(6, 2, 3) + Sum(6, 2, 2)

To evaluate Sum(6, 3, 8), we need to evaluate Sum(6, 2, 7) which can recursively written as following: Sum(6, 2, 7) = Sum(6, 1, 6) + Sum(6, 1, 5) + Sum(6, 1, 4) +
Sum(6, 1, 3) + Sum(6, 1, 2) + Sum(6, 1, 1)

We also need to evaluate Sum(6, 2, 6) which can recursively written as following:
Sum(6, 2, 6) = Sum(6, 1, 5) + Sum(6, 1, 4) + Sum(6, 1, 3) +
Sum(6, 1, 2) + Sum(6, 1, 1)..
....
Sum(6, 2, 2) = Sum(6, 1, 1)

Algorithm:

```
int findWays(int m, int n, int x)
{
   // Create a table to store results of subproblems.  One extra
   // row and column are used for simpilicity (Number of dice
   // is directly used as row index and sum is directly used
   // as column index).  The entries in 0th row and 0th column
   // are never used.
   int table[n + 1][x + 1];
   memset(table, 0, sizeof(table)); // Initialize all entries as 0
   // Table entries for only one dice
   for (int j = 1; j <= m && j <= x; j++)
      table[1][j] = 1;
   // Fill rest of the entries in table using recursive relation
```

```
    // i: number of dice, j: sum
    for (int i = 2; i <= n; i++)
        for (int j = 1; j <= x; j++)
            for (int k = 1; k <= m && k < j; k++)
                table[i][j] += table[i-1][j-k];
    /* Uncomment these lines to see content of table
    for (int i = 0; i <= n; i++)
    {
      for (int j = 0; j <= x; j++)
        cout << table[i][j] << " ";
      cout << endl;
    } */
    return table[n][x];
}
```

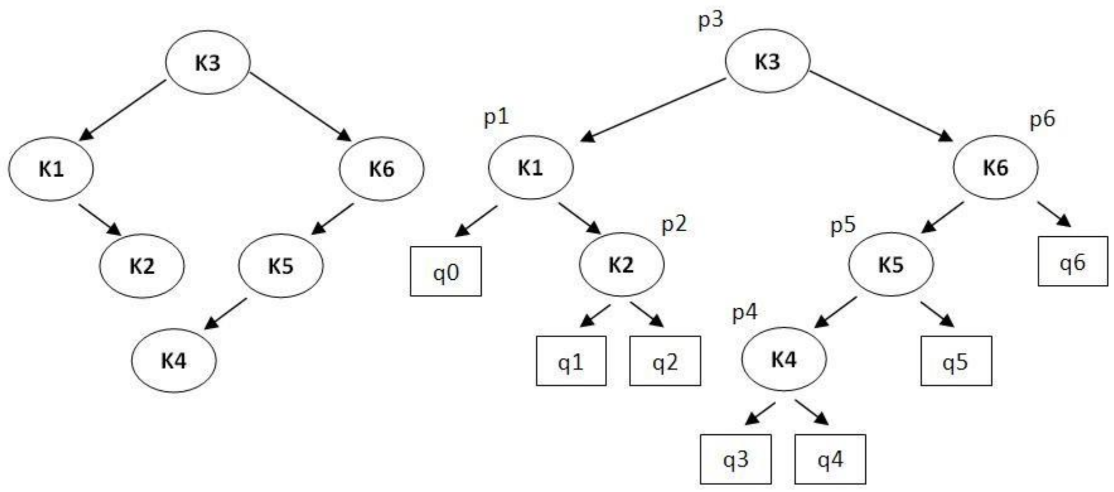
Time Complexity: O(m * n * x) where m is number of faces, n is number of dice and x is given sum.

We can add the following two conditions at the beginning of findWays() to improve performance of the program for extreme cases (x is too high or x is too low)

```
// When x is so high that sum can not go beyond x even when we

// get maximum value in every dice throw.

if (m*n <= x)

            return (m*n == x);

// When x is too low

if (n >= x)

            return (n == x);
```

Optimal Binary Search Algorithms

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum. For the purpose of a better presentation of optimal binary search trees, we will consider “extended binary search trees”, which have the keys stored at their internal nodes. Suppose “n” keys k1, k2, … , k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn. An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



a) b)

a) Binary Search Tree b) Extended Binary Search Tree

In the extended tree: The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;

The round nodes represent internal nodes; these are the actual keys stored in the tree;

Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.

If the user searches a particular key in the tree, 2 cases can occur:

Case 1 – the key is found, so the corresponding weight ‘p’ is incremented;

Case 2 – the key is not found, so the corresponding ‘q’ value is incremented.

Where optimal binary search trees may be used

In general, word prediction is the problem of guessing the next word in a sentence as the sentence is being entered, and updates this prediction as the word is typed. Currently “word prediction” implies both “word completion and word prediction”. Word completion is defined as offering the user a list of words after a letter has been typed, while word prediction is defined as offering the user a list of probable words after a word has been typed or selected, based on previous words rather than on the basis of the letter. Word completion problem is easier to solve since the knowledge of some letter(s) provides the predictor a chance to eliminate many of irrelevant words.

Online dictionaries rely heavily on the facilities provided by optimal search trees. As the dictionary has more and more users, it is able to assign weights to the corresponding words, according to the frequency of their search. This way, it will be able to provide a much faster answer, as search time dramatically decreases when storing words into a binary search tree.

Word prediction applications are becoming increasingly popular. For example, when you start typing a query in google search, a list of possible entries almost instantly appears.

The terminal node in the extended tree that is the left successor of k_1 can be interpreted as representing all key values that are not stored and are less than k_1. Similarly, the terminal node in the extended tree that is the right successor of k_n, represents all key values not stored in the tree that are greater than k_n. The terminal node that is successed between k_i and k_{i-1} in an inorder traversal represents all key values not stored that lie between k_i and k_{i-1}.

In the extended tree in the above figure if the possible key values are 0, 1, 2, 3, …, 100 then the terminal node labeled q_0 represents the missing key values 0, 1 and 2 if $k_1=3$. The terminal node labeled q_3 represents the key values between k_3 and k_4. If $k_3=17$ and $k_4=21$ then the terminal node labeled q_3 represents the missing key values 18, 19 and 20. If k_6 is 90 then the terminal node q_6 represents the missing key values 91 through 100.

An obvious way to find an optimal binary search tree is to generate each possible binary search tree for the keys, calculate the weighted path length, and keep that tree with the smallest weighted path length. This search through all possible solutions is not feasible, since the number of such trees grows exponentially with “n”.

An alternative would be a recursive algorithm. Consider the characteristics of any optimal tree. Of course it has a root and two subtrees. Both subtrees must themselves be optimal binary search trees with respect to their keys and weights. First, any subtree of any binary search tree must be a binary search tree. Second, the subtrees must also be optimal.

Since there are "n" possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys less than that key must appear in its left subtree while all keys greater than it must appear in its right subtree. Stating the recursive algorithm based on these observations requires some notations:

OBST(i, j) denotes the optimal binary search tree containing the keys ki, ki+1, …, kj;

Wi, j denotes the weight matrix for OBST(i, j)

Wi, j can be defined using the following formula:

$$W_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

Ci, j, $0 \le i \le j \le n$ denotes the cost matrix for OBST(i, j)

Ci, j can be defined recursively, in the following manner: Ci, i = Wi, j

Ci, j = Wi, j + mini<k≤j(Ci, k - 1 + Ck, j)

Ri, j, $0 \le i \le j \le n$ denotes the root matrix for OBST(i, j)

Assigning the notation Ri, j to the value of k for which we obtain a minimum in the above relations, the optimal binary search tree is OBST(0, n) and each subtree OBST(i, j) has the root kRij and as subtrees the trees denoted by OBST(i, k-1) and OBST(k, j).

OBST(i, j) will involve the weights qi-1, pi, qi, …, pj, qj.

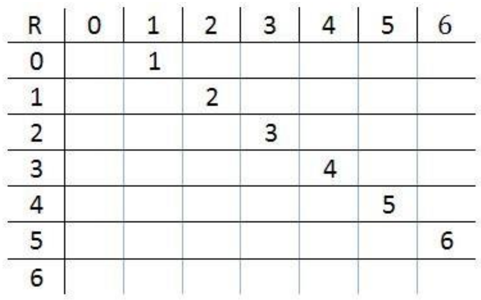
All possible optimal subtrees are not required. Those that are consist of sequences of keys that are immediate successors of the smallest key in the subtree, successors in the sorted order for the keys.

The bottom-up approach generates all the smallest required optimal subtrees first, then all next smallest, and so on until the final solution involving all the weights is found. Since the algorithm requires access to each subtree's weighted path length, these weighted path lengths must also be retained to avoid their recalculation. They will be stored in the weight matrix ‘W’. Finally, the root of each subtree must also be stored for reference in the root matrix ‘R’.

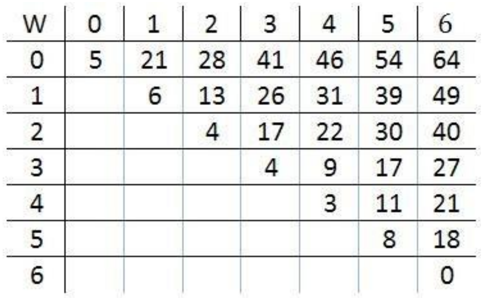
Example of Optimal Binary Search Tree (OBST)

Find the optimal binary search tree for N = 6, having keys k1 … k6 and weights p1 = 10, p2 = 3, p3 = 9, p4 = 2, p5 = 0, p6 = 10; q0 = 5, q1 = 6, q2 = 4, q3 = 4, q4 = 3, q5 = 8, q6 = 0. The following figure shows the arrays as they would appear after the initialization and their final disposition.

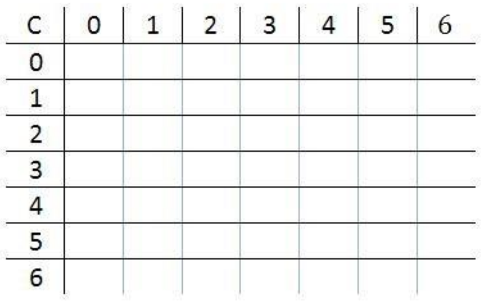
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
k		3	7	10	15	20	25
p	-	10	3	9	2	0	10
q	5	6	4	4	3	8	0



R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							



W	0	1	2	3	4	5	6
0	5	21	28	41	46	54	64
1		6	13	26	31	39	49
2			4	17	22	30	40
3				4	9	17	27
4					3	11	21
5						8	18
6							0



C	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							

Initial array values

The values of the weight matrix have been computed according to the formulas previously stated, as follows:

W (0, 0) = q0 = 5	W (0, 1) = q0 + q1 + p1 = 5 + 6 + 10 = 21
W (1, 1) = q1 = 6	W (0, 2) = W (0, 1) + q2 + p2 = 21 + 4 + 3 = 28
W (2, 2) = q2 = 4	W (0, 3) = W (0, 2) + q3 + p3 = 28 + 4 + 9 = 41

W (3, 3) = q3 = 4
W (4, 4) = q4 = 3
W (5, 5) = q5 = 8
W (6, 6) = q6 = 0
--- and so on until we reach
W (5, 6) = q5 + q6 + p6 = 18

W (0, 4) = W (0, 3) + q4 + p4 = 41 + 3 + 2 = 46
W (0, 5) = W (0, 4) + q5 + p5 = 46 + 8 + 0 = 54
W (0, 6) = W (0, 5) + q6 + p6 = 54 + 0 + 10 = 64
W (1, 2) = W (1, 1) + q2 + p2 = 6 + 4 + 3 = 13

The elements of the cost matrix are afterwards computed following a pattern of lines that are parallel with the main diagonal.

C (0, 0) = W (0, 0) = 5
C (1, 1) = W (1, 1) = 6
C (2, 2) = W (2, 2) = 4
C (3, 3) = W (3, 3) = 4
C (4, 4) = W (4, 4) = 3
C (5, 5) = W (5, 5) = 8
C (6, 6) = W (6, 6) = 0

C	0	1	2	3	4	5	6
0	5						
1		6					
2			4				
3				4			
4					3		
5						8	
6							0

Figure 3. Cost Matrix after first
step C (0, 1) = W (0, 1) + (C (0, 0) + C (**1**, 1)) = 21 + 5 + 6 = 32
C (1, 2) = W (0, 1) + (C (1, 1) + C (**2**, 2)) = 13 + 6 + 4 = 23
C (2, 3) = W (0, 1) + (C (2, 2) + C (**3**, 3)) = 17 + 4 + 4 = 25
C (3, 4) = W (0, 1) + (C (3, 3) + C (**4**, 4)) = 9 + 4 + 3 = 16
C (4, 5) = W (0, 1) + (C (4, 4) + C (**5**, 5)) = 11 + 3 + 8 = 22
C (5, 6) = W (0, 1) + (C (5, 5) + C (**6**, 6)) = 18 + 8 + 0 = 26

**The bolded numbers represent the elements added in the root matrix.*

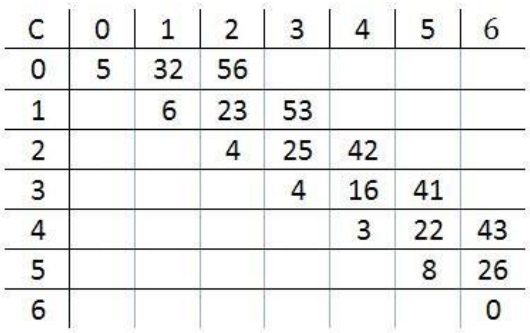
C	0	1	2	3	4	5	6
0	5	32					
1		6	23				
2			4	25			
3				4	16		
4					3	22	
5						8	26
6							0

R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							

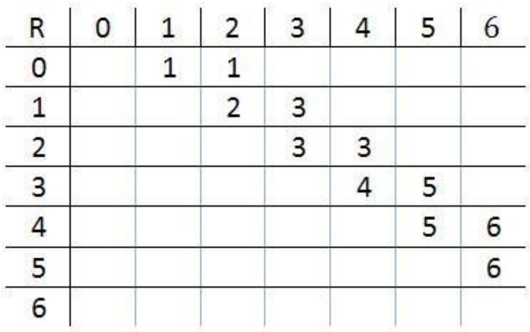
Cost and Root Matrices after second step

C (0, 2) = W (0, 2) + min (C (0, 0) + C (**1**, 2), C (0, 1) + C (2, 2)) = 28 + min (**28**, 36) = 56
C (1, 3) = W (1, 3) + min (C (1, 1) + C (2, 3), C (1, 2) + C (**3**, 3)) = 26 + min (31, **27**) = 53
C (2, 4) = W (2, 4) + min (C (2, 2) + C (**3**, 4), C (2, 3) + C (4, 4)) = 22 + min (**20**,

28) = 42
C (3, 5) = W (3, 5) + min (C (3, 3) + C (4, 5), C (3, 4) + C (**5**, 5)) = 17 + min (26, **24**) = 41
C (4, 6) = W (4, 6) + min (C (4, 4) + C (5, 6), C (4, 5) + C (**6**, 6)) = 21 + min (29, **22**) = 43



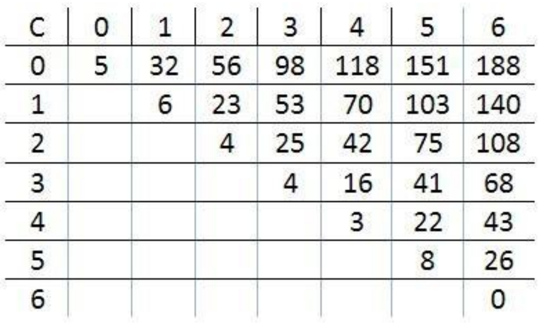
C	0	1	2	3	4	5	6
0	5	32	56				
1		6	23	53			
2			4	25	42		
3				4	16	41	
4					3	22	43
5						8	26
6							0



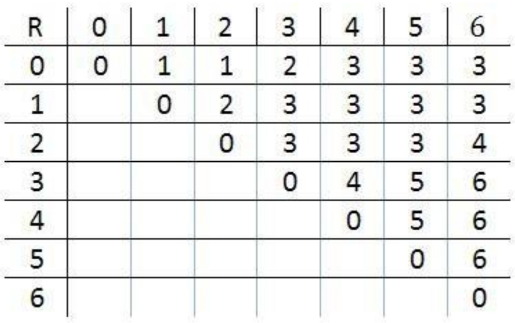
R	0	1	2	3	4	5	6
0		1	1				
1			2	3			
2				3	3		
3					4	5	
4						5	6
5							6
6							

Cost and Root matrices after third step

And so on
…
C(1, 5) = W(1, 5) + min(C(1, 1) + C(2, 5), C(1, 2) + C(**3**, 5), C(1, 3) + C(4, 5), C(1, 4) + C(5, 5)) =
= 39 + min(81, **64**, 75, 78) = 103



C	0	1	2	3	4	5	6
0	5	32	56	98	118	151	188
1		6	23	53	70	103	140
2			4	25	42	75	108
3				4	16	41	68
4					3	22	43
5						8	26
6							0



R	0	1	2	3	4	5	6
0	0	1	1	2	3	3	3
1		0	2	3	3	3	3
2			0	3	3	3	4
3				0	4	5	6
4					0	5	6
5						0	6
6							0

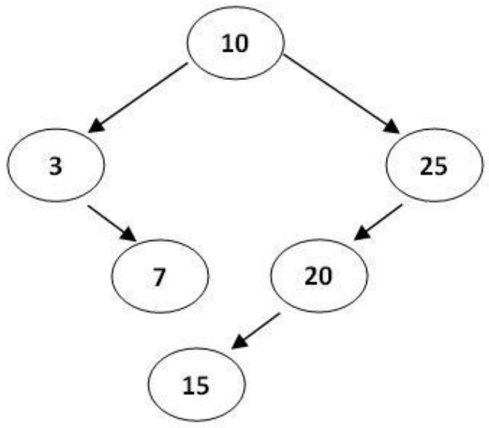
Final array values

The resulting optimal tree is shown in the bellow figure and has a weighted path length of 188. Computing the node positions in the tree is performed in the following manner:

- The root of the optimal tree is R(0, 6) = k3;
- The root of the left subtree is R(0, 2) = k1;
- The root of the right subtree is R(3, 6) = k6;
- The root of the right subtree of k1 is R(1, 2) = k2

- The root of the left subtree of k6 is R(3, 5) = k5
- The root of the left subtree of k5 is R(3, 4) = k4

Thus, the optimal binary search tree obtained will have the following structure:



The Obtained Optimal Binary Search Tree

Analysis

$T_{i,\,j}$ consists of a root containing a_k, for some k and left and right subtrees of the root, with the left subtree being an **optimal** (min cost) **tree** $T_{i,\,k-1}$ and the right subtree being $T_{k,\,j}$. The **optimal** $T_{i,\,j}$ will have root a_k that minimizes the sum $c_{i,\,k-1} + c_{k,\,j}$. The **time complexity** of this algorithm is $O(n^3)$.