UNIT - III

GREEDY METHOD.

* A greedy algorithm, always makes the choice that seems to be the best at that moment. It makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
* In greedy technique, the solution is constructed through a sequence of steps each expanding a partially constructed solution achieved until a complete solution is reached. At each step choice made should be
  - → Feasible - should satisfy the problem constraints
  - → Locally optimal - Among all feasible solutions the best choice is to be made
  - → Irrevocable - Once the particular choice is made then it should not get changed on subsequent steps
* In greedy method, the following activities are performed:
  - → First, select some solution from the input domain.
  - → Check whether the solution is feasible or not.
  - → From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function is referred as optimal solution.
* As greedy method works in stages. At each stage only one input is considered at each time. Based on this input it is decided whether particular input gives optimal solution or not.

EXAMPLE PROBLEM:

In each iteration, greedily select the works which will take the minimum amount of time to complete while maintaining two variables currentTime and numberofWorks.

* To complete the calculation:
  - Sort array A in non-decreasing order
  - select each to-do item one-by-one
  - Add the time that it will take to complete that to-do item into currentTime.
  - Add one to numberofWorks.
  - Repeat this as long as the CurrentTime is less than or equal to T.

* Let A = {5, 3, 4, 2, 1} and T = 6.

  After Sorting A = {1, 2, 3, 4, 5}

  After 1st iteration : Current Time = 1
  numberofWorks = 1

  After 2nd iteration . currentTime = 1+2 = 3
  numberofWorks = 2

  After 3rd iteration : CurrentTime = 3+3 = 6
  numberofWorks = 3

  After 4th iteration . currentTime = 6+4 = 10, which is greater than T(6)

  ∴ The answer is 3. (ie) three works can be completed within T(6).

HUFFMAN CODING USING GREEDY APPROACH.

* Codeword refers to assigning sequence of bits to text Symbols.
* There are 2 ways to assign codewords - 1. Fixed length Coding, 2. Variable length coding
* Fixed length coding assigns bit string of same length to each symbol.

  Example 1 : Symbols - A, B, C, D
  Codewords - 00 (A), 01 (B), 10 (C), 11 (D)

  Example 2 : Symbols - A, B, C, D, E, F, G, H
  Codewords - 000 (A), 001 (B), 010 (C), 011 (D), 100 (E), 101 (F), 110 (G), 111 (H)

Variable length Coding assigns shorter codewords to more frequent symbols and longer codewords to less frequent symbols.

* The variable length Codes assigned to input text are Prefix Codes, means the codes are assigned in such a way that the code assigned to one symbol is not the prefix of Code assigned to any other symbol.

HUFFMAN'S ALGORITHM:

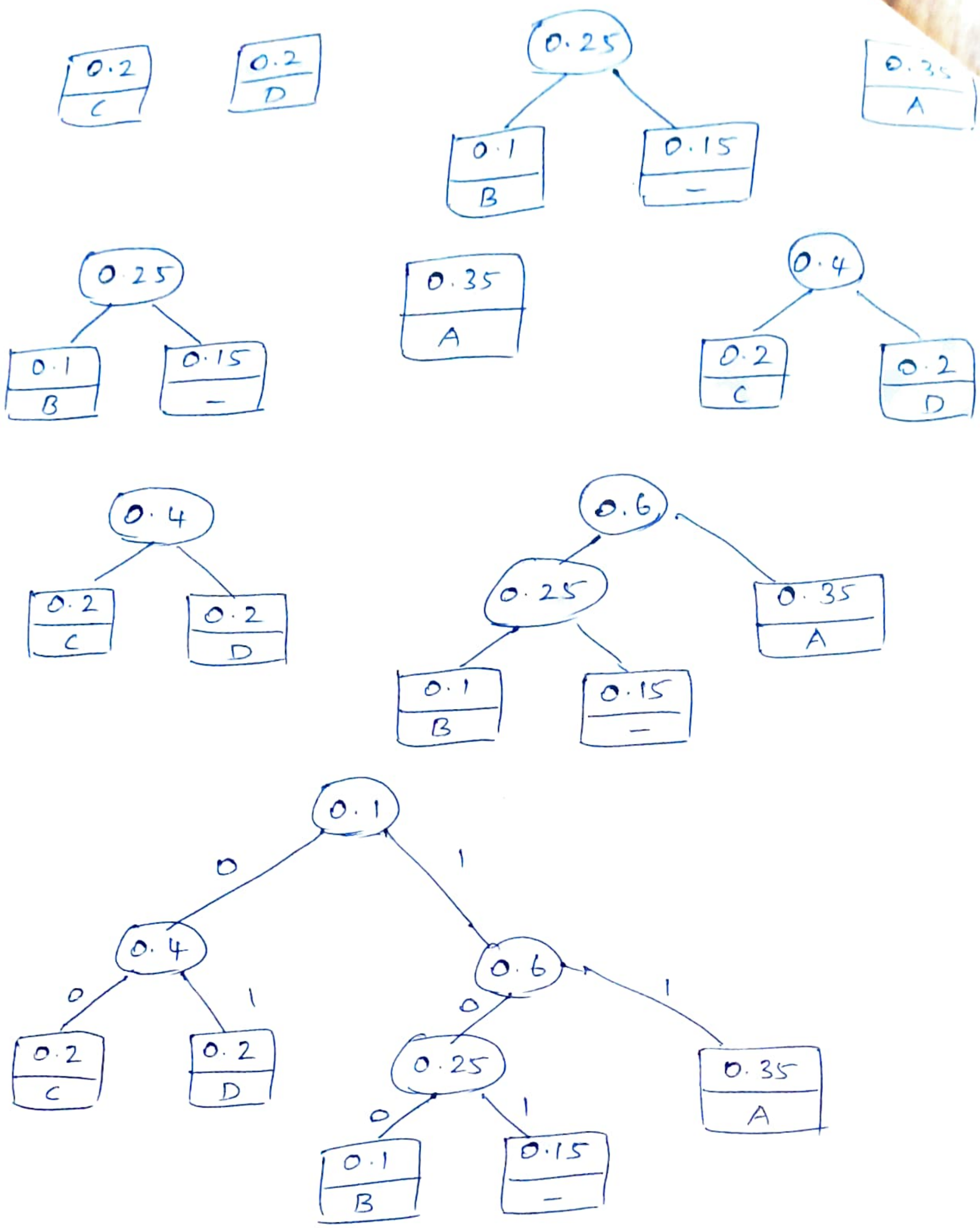
* Steps to construct Huffman Tree.
  1. Initialize n one-node trees and label them with the symbols of alphabet given.
  2. Repeat the following operation until a single tree is obtained. Find 2 trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

* Steps to generate Huffman Code.
  1. Label all the left edges with 0 and right edges with 1.
  2. Obtain codeword for a symbol by recording labels on the simple path from the root to symbol's leaf.

Example:
Consider the 5 symbol alphabet {A, B, C, D, –} with the following occurrence frequencies

Symbol	A	B	C	D	–
Frequency	0.35	0.1	0.2	0.2	0.15

0.1	0.15	0.2	0.2	0.35
B	–	C	D	A

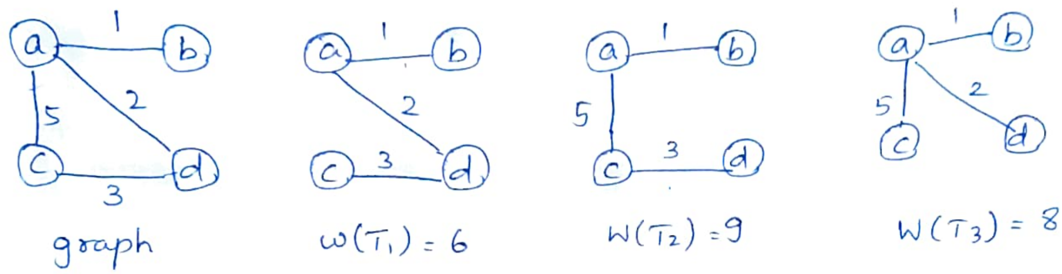


The resulting Codewords.

Symbol	A	B	C	D	—
frequency	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101

MINIMUM SPANNING TREE

* A weighted graph has weights assigned to its edges. A minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights of all its edges.
* The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.



PRIM'S ALGORITHM

ALGORITHM Prim(G)

// Input : A weighted connected graph $G = (V, E)$

// Output : E_T, the set of edges composing a MST of G.

$V_T \leftarrow \{V_0\}$

$E_T \leftarrow \phi$

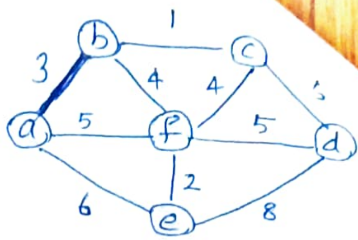
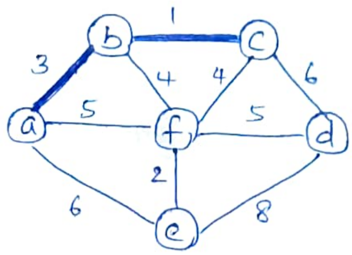
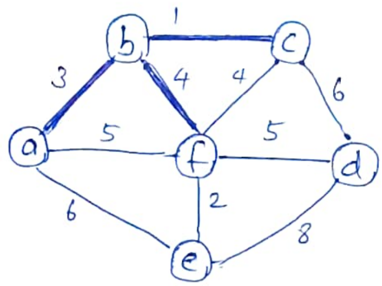
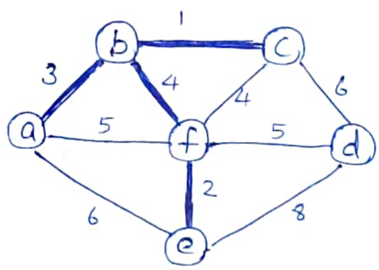
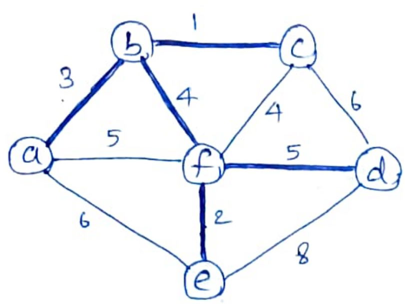
for $i \leftarrow 1$ to $|V| - 1$ do

  find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$.

  $V_T \leftarrow V_T \cup \{u^*\}$

  $E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Tree Vertices	Remaining Vertices	Illustration
a(–, –)	b(a,3), c(–, ∞), d(–, ∞) e(a,6), f(a,5)	
b(a,3)	c(b,1) d(–, ∞) e(a,6) f(b,4)	
c(b,1)	d(c,6) e(a,6) f(b,4)	
f(b,4)	d(f,5) e(f,2)	
e(f,2)	d(f,5)	
d(f,5)		

USKAL'S ALGORITHM

ALGORITHM Kruskal (G)

// Input : A weighted connected graph $G = (V, E)$

// Output : E_T, the set of edges composing a MST of G

Sort E in nondecreasing order of the edges weights $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \phi$; ecounter $\leftarrow 0$ // initialize the set of tree edges and its size.

$k \leftarrow 0$; // initialize the no. of processed edges

while (ecounter $< |V| - 1$) do

  $k \leftarrow k + 1$

  if $E_T \cup \{e_{i_k}\}$ is acyclic

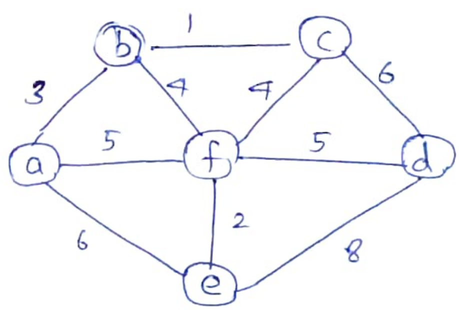
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$;

    ecounter $\leftarrow$ ecounter $+ 1$;

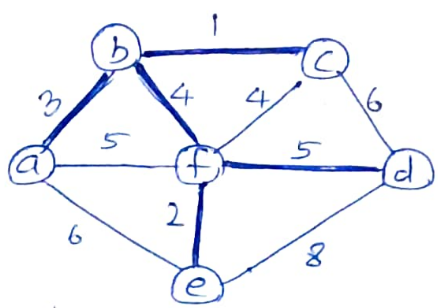
return E_T.

Tree edges   Sorted list of edges   Illustration

bc	ef	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8



Include edges bc, ef, ab, bf and df. cf and af are not included since they form cycle.



Time Complexity:

* The running time of Prim's Algorithm is in $(|V| - 1 + |E|)\, O(\log |V|) = O(|E| \log |V|)$, because in a connected graph, $|V| - 1 \leq |E|$.

* The running time of Kruskal's Algorithm is in $(|V| - 1 + |E|)\, O(\log |E|) = O(|E| \log |E|)$, because in a connected graph, $|V| - 1 \leq |E|$.

KNAPSACK PROBLEM:

* Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Ex:

Objects	1	2	3	4	5	6	7
Profit (P)	5	10	15	7	8	9	4
Weight (W)	1	3	5	4	1	3	2

$W = 15$

(i) Max. Profit Method:

Objects	Profit (P)	Weight (W)	Remaining weight
3	15	5	$15 - 5 = 10$
2	10	3	$10 - 3 = 7$
6	9	3	$7 - 3 = 4$
5	8	1	$4 - 1 = 3$
4	$7 * 3/4 = 5.25$	3	$3 - 3 = 0$
Total Profit	$= 47.25$		

...eight Method:

Objects	Profit (P)	Weight (W)	Remaining weight
1	5	1	15 - 1 = 14
5	8	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	15 * 1/5 = 3	1	1 - 1 = 0
Total Profit	= 46.		

(iii) Max. Profit / Weight Ratio:

Objects	Profit (P)	Weight (W)	Remaining Weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0
Total profit	51.		

Hence, out of the above 3 methods maximum Profit / Weight Ratio method yields maximum profit

DYNAMIC PROGRAMMING:

* Dynamic Programming approach is similar to div[ide] and conquer in breaking down the problem into smaller and yet smaller possible sub-problems.

* But, unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

* It is used, where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization.

* Before solving the in-hand subproblem, dynamic alg. will try to examine the results of previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

Eg:

* The Fibonacci numbers are the elements of the sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n>1$$

and 2 initial conditions, $F(0) = 0$ & $F(1) = 1$.

* The problem of computing $F(n)$ is expressed in terms of its smaller and overlapping subproblems of computing $F(n-1)$ and $F(n-2)$.

...sack Problem:

...ven set of items, each with a weight and a value, determine subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible

Formula:

$$M[i,W] = \max\{M[i-1,W],\ M[i-1,\ w-w[i]] + P[i]\}$$

Example:

weights = {3, 4, 6, 5} and Profit = {2, 3, 1, 4}.

W = 8 and n = 4.

M[i,W]		i\W	0	1	2	3	4	5	6	7	8
P_i	W_i	0	0	0	0	0	0	0	0	0	0
2	3 ✓	1	0	0	0	2 ←	2	2	2	2	2
3	× 4	2	0	0	0	2 ←	3	3	3	5	5
4	✓ 5	3	0	0	0	2	3	4	4	5	6 ←
1	× 6	4	0	0	0	2	3	4	4	5	6 ←

$$M[4,7] = \max\{M[4-1,7],\ M[4-1,\ 7-W[4]] + P[4]\}$$
$$= \max\{M[3,7],\ M[3,7-6]+1\}$$
$$= \max\{5,\ M[3,1]+1\}$$
$$= \max\{5,\ 0+1\}$$
$$= \max\{5,1\}$$
$$= 5.$$

max(3+0, 2) = 3

max(3+0, 2) = 3

max(3+2, 2)

5, 2 = 5

Output = {1, 0, 0, 1}. Item 1 and 4 are included into the knapsack to yield maximum profit.

```
Algorithm knapsack (i, w)
{
  if w < Wi
      value ← knapsack (i-1, w)
  else
      value ← max({M[i-1, W], M[i-1, w-w[i]] + P[i]})
  M[i, W] ← value
  return M[i, W]
}
```

Time Complexity of 0/1 knapsack problem is $O(nW)$, where n is the number of items and W is the capacity of knapsack.

Matrix Chain Multiplication:

* Assume 3 matrices A_1, A_2 and A_3 of dimensions 2×3, 3×4 and 4×2. These 3 matrices can be multiplied in any one of the 2 ways - (i) $(A_1 * A_2) * A_3$ and (ii) $A_1 * (A_2 * A_3)$

* The no. of multiplications required in case (i).

  To multiply $A_1 * A_2 = 2*3*4 = 24$.

  To multiply $(A_1 * A_2) * A_3 = 2*4*2 = 16$.

  Total no. of multiplications $= 24 + 16 = 40$.

* The no. of multiplications required in case (ii).

  To multiply $(A_2 * A_3) = 3*4*2 = 24$.

  To multiply $A_1 * (A_2 * A_3) = 2*3*2 = 12$.

  Total no. of multiplications $= 24 + 12 = 36$.

Therefore, when case (ii) is used it leads to least cost matrix multiplication of $A_1 * A_2 * A_3$

...ber of possible parenthesization is given by, $^nC_r = \frac{n!}{r!(n-r)!}$

$2(n-1)C_{n-1} / n$

$6C_3 = \frac{6\times5\times4\times3\times2\times1}{3\times2\times1\times3\times2\times1}$

* To multiply 4 matrices $(n=4) \Rightarrow A_1 * A_2 * A_3 * A_4$.

$2\times3 C_3 / 4 = 6C_3/4 = \frac{6*5*4}{1\times2*3\times4} = 5$. $\frac{5\times4}{4}$ $6C_3 = \frac{6\times5\times4\times3\times2\times1}{1\times2\times3}$

$= 6\times5\times4$

* To find out the no. of multiplications required and hence to get optimal solution,

$$C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} * d_k * d_j \}.$$

* Example: Assume 4 matrices $A_1 * A_2 * A_3 * A_4$ of dimensions, 3×2, 2×4, 4×2, 2×5. where $d_0 = 3$, $d_1 = 2$, $d_2 = 4$, $d_3 = 2$ & $d_4 = 5$. Construct 2 matrices cost matrix (table) and k table of size $n\times n$ (4×4).

C	1	2	3	4
1	0			
2		0		
3			0	
4				0

k	1	2	3	4
1				
2				
3				
4				

$$C[1,2] = \min_{\substack{1 \le k < 2 \\ k=1}} \{ C[1,1] + C[2,2] + d_0 * d_1 * d_2 \}$$

$$= 0 + 0 + 3*2*4 = 24.$$

$$C[2,3] = \min_{\substack{2 \le k < 3 \\ k=2}} \{ C[2,2] + C[3,3] + d_1 * d_2 * d_3 \}$$

$$= 0 + 0 + 2*4*2 = 16.$$

$$C[3,4] = \min_{\substack{3 \le k < 4 \\ k=3}} \{ C[3,3] + C[4,4] + d_2 * d_3 * d_4 \}$$

$$= 0 + 0 + 4*2*5 = 40.$$

$$C[1,3] = \min_{1 \le k < 3} \begin{cases} k=1 & C[1,1] + C[2,3] + d_0 \cdot d_1 \cdot d_3 \\ k=2 & C[1,2] + C[3,3] + d_0 \cdot d_2 \cdot d_3 \end{cases}$$

$$= \min \{ 0 + 16 + 3*2*2,\ 24 + 0 + 3*4*2 \}$$

$$= \min \{ 28, 48 \}$$

$$= 28.$$

$$C[\overset{i}{2},\overset{j}{4}] = \min_{2 \le k < 4} \begin{cases} k=2 & C[2,2] + C[3,4] + d_1 \cdot d_2 \cdot d_4 \\ k=3 & C[2,3] + C[4,4] + d_1 \cdot d_3 \cdot d_4 \end{cases}$$

$$= \min \{ 0 + 40 + 2*4*5,\ 16 + 0 + 2*2*5 \}$$

$$= \min \{ 80, 36 \}$$

$$= 36.$$

$$C[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & C[1,1] + C[2,4] + d_0 \cdot d_1 \cdot d_4 \\ k=2 & C[1,2] + C[3,4] + d_0 \cdot d_2 \cdot d_4 \\ k=3 & C[1,3] + C[4,4] + d_0 \cdot d_3 \cdot d_4 \end{cases}$$

$$= \min \{ 0 + 36 + 3*2*5,\ 24 + 40 + 3*4*5,\ 28 + 0 + 3*2*5 \}$$

$$= \min \{ 66, 124, 58 \}$$

$$= 58$$

C	1	2	3	4
1	0	24	28	$\boxed{58}$
2		0	16	36
3			0	40
4				0

K	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Paranthesization:

$A_1 A_2 A_3 A_4 = (A_1 A_2 A_3) A_4$

$= \cancel{((A_1) A_2 A_3) A_4}$

$A_1 (A_2 A_3) A_4$

$A_1 (A_2 A_3) A_4$

Time Complexity of Matrix Chain Multiplication is $O(n^3)$ and space complexity is $O(n^2)$.

Common Subsequence (LCS)

The longest Common Subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

* If S_1 and S_2 are the 2 given sequences, then, Z is the common subsequence of S_1 and S_2 if Z is a sub sequence of both S_1 and S_2. Furthermore, Z must be a strictly increasing sequence of the indices of both S_1 and S_2.

* Eg 1: Str 1 : a b c d e f g h i j k l

  Str 2 : f c e h k

  Common Subsequences : fhk, cehk, ehk, hk

  Longest common subsequence : cehk.

```
Algorithm LCS (x, y)
  m ← length (x)
  n ← length (y)
  for i ← 1 to m do
     c [i, 0] ← 0
  for j ← 0 to m do
     c [0, j] ← 0
  for i ← 1 to m do
     for j ← 1 to n do
        if (xi == yj) then
           c [i, j] ← c [i-1, j-1] + 1
           b [i, j] ← '↖'
```

else if c[i-1, j] ≥ c[i, j-1] then

c[i, j] ← c[i-1, j]

b[i, j] ← "↑"

else

c[i, j] ← c[i, j-1]

b[i, j] ← "←"

return c and b.

Eg. x : A B C B D A B

y : B D C A B A.

	j	0	1	2	3	4	5	6
i		y	B	D	C	A	B	A
0	x	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

The longest common subsequence is BCBA and its length is 4.

Time Complexity of LCS is O(m * n) and Space Complexity of LCS is O(m * n)

al Binary Search Tree (OBST).

Consider 4 keys A, B, C and D to be searched for with probabilities 0.1, 0.2, 0.4 and 0.3 respectively.

* Figure depicts 2 out of 14 possible binary search tree containing these keys.

* The average no. of comparisons in a successful search in the first tree is $0.1 * 1 + 0.2 * 2 + 0.4 * 3 + 0.3 * 4 = 2.9$ and for second tree it is $0.1 * 2 + 0.2 * 1 + 0.4 * 2 + 0.3 * 3 = 2.1$

* Let $a_1, \ldots a_n$ be distinct keys ordered from smallest to largest and let $P_1, \ldots P_n$ be the probabilities for searching them. Let $C(i,j)$ be the smallest average no. of comparisons made in a successful search in a binary search tree.

* Construct 2 tables - cost table and root table of size $(n+1) * (n+1)$

* Let $C(i, i-1) = 0$ and $C(i,i) = P_i$ and $R(i,i) = i$.

$$C(i,j) = \min_{i \le k \le j} \{ C(i, k-1) + C(k+1, j) \} + \sum_{s=i}^{j} P_s \quad \text{for } 1 \le i \le j \le n$$

Example: Consider the following keys and probabilities. Construct an optimal binary search tree.

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

* Apply $C(i, i-1) = 0$.

$C(1, 1-0) = C(1,0) = 0$

$C(2, 2-1) = C(2,1) = 0$

$C(3, 2) = C(4,3) = C(5,4) = 0$.

cost table

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

* Apply $C(i,i) = P_i$ and $R(i,i) = i$

$C(1,1) = P_1 = 0.1$ and $R(1,1) = 1$

$C(2,2) = P_2 = 0.2$ and $R(2,2) = 2$

$C(3,3) = P_3 = 0.4$ and $R(3,3) = 3$

$C(4,4) = P_4 = 0.3$ and $R(4,4) = 4$.

C	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

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

$$C(1,2) = \min_{1 \le k \le 2} \begin{cases} k=1 & C(1,0) + C(2,2) + P_1 + P_2 \\ k=2 & C(1,1) + C(3,2) + P_1 + P_2 \end{cases}$$

$= \min \{ 0 + 0.2 + 0.3 \ , \ 0.1 + 0 + 0.3 \}$

$= \min \{ 0.5, 0.4 \}$

$= 0.4 \ (k = 2)$

$$= \min_{2 \le k \le 3} \begin{cases} k=2 & C(2,1) + C(3,3) + P_2 + P_3 \\ k=3 & C(2,2) + C(4,3) + P_2 + P_3 \end{cases}$$

$$= \min \{0 + 0.4 + 0.6,\ 0.2 + 0 + 0.6\}$$

$$= \min \{1.0,\ 0.8\}$$

$$= 0.8 \ (k=3).$$

$$C(3,4) = \min_{3 \le k \le 4} \begin{cases} k=3 & C(3,2) + C(4,4) + P_3 + P_4 \\ k=4 & C(3,3) + C(5,4) + P_3 + P_4 \end{cases}$$

$$= \min \{0 + 0.3 + 0.7,\ 0.4 + 0 + 0.7\}$$

$$= \min \{1.0,\ 1.1\}$$

$$= 1.0 \ (k=3).$$

$$C(1,3) = \min_{1 \le k \le 3} \begin{cases} k=1 & C(1,0) + C(2,3) + P_1 + P_2 + P_3, \\ k=2 & C(1,1) + C(3,3) + P_1 + P_2 + P_3, \\ k=3 & C(1,2) + C(4,3) + P_1 + P_2 + P_3 \end{cases}$$

$$= \min \{0 + 0.8 + 0.7,\ 0.1 + 0.4 + 0.7,\ 0.4 + 0 + 0.7\}$$

$$= \min \{1.5,\ 1.2,\ 1.1\} = 1.1 \ (k=3)$$

$$C(2,4) = \min_{2 \le k \le 4} \begin{cases} k=2 & C(2,1) + C(3,4) + P_2 + P_3 + P_4, \\ k=3 & C(2,2) + C(4,4) + P_2 + P_3 + P_4, \\ k=4 & C(2,3) + C(5,4) + P_2 + P_3 + P_4 \end{cases}$$

$$= \min \{0 + 1 + 0.9,\ 0.2 + 0.3 + 0.9,\ 0.8 + 0 + 0.9\}$$

$$= \min \{1.9,\ 1.4,\ 1.7\} = 1.4 \ (k=3)$$

$$C(1,4) = \min_{1 \le k \le 4} \begin{cases} k=1 & C(1,0) + C(2,4) + 1 \\ k=2 & C(1,1) + C(3,4) + 1 \\ k=3 & C(1,2) + C(4,4) + 1 \\ k=4 & C(1,3) + C(5,4) + 1 \end{cases}$$

$$= \min \{0 + 1.4 + 1,\ 0.1 + 1 + 1,\ 0.4 + 0.3 + 1,\ 1.1 + 0 + 1\}$$

$$= \min \{2.4,\ 2.1,\ 1.7,\ 2.1\}$$

$$= 1.7 \ (k=3)$$

C	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

R	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4.
5