Algorithms

Chapter 7: Dynamic Programming

GATE CS Lectures
by Monalisa

MonalisaCS

Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

MonalisaCS

- Chapter 7: Dynamic Programming:-
- The Bellman-Ford algorithm
- Warshall's and Floyd's Algorithm ,
- Rod cutting
- Matrix-chain multiplication
- Longest common subsequence
- Optimal binary search trees

MonalisaCS

Dynamic Programming

https://monalisacs.com/

- The word "programming" in the name of this technique stands for "planning" and does not refer to computer programming
- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- Dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- We apply dynamic programming to ***optimization problems***.
- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- When developing a dynamic-programming algorithm, we follow 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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

	Greedy Method	Dynamic Programming
1. Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
2. Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
3.Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
4.Space complexity	It is more efficient in terms of space as it never look back or revise previous choices	It requires Dynamic Programming table for scace and it increases it’s space complexity.
5.Time complexity	Greedy methods are generally faster. For example, Dijkstra’s shortest path algorithm takes O(ELogV)	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **The Bellman-Ford algorithm**
- The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in which edge weights may be negative.
- The Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
- If there is such a cycle, the algorithm indicates that no solution exists.
- If there is no such cycle, the algorithm produces the shortest paths and their weights.
- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(u, v)$.
- The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.
- **BELLMAN-FORD(*G* , *w* , *s*)**

```
1.  INITIALIZE-SINGLE-SOURCE(G,s)
2.  for i=1 to |G.V|-1
3.          for each edge (u , v) ∈ G.E
4.              RELAX(u ,v , w)
5.  for each edge (u , v) ∈ G.E
6.          if v. d > u . d+ w(u,v)
7.              return FALSE
8.  return TRUE
```

MonalisaCS

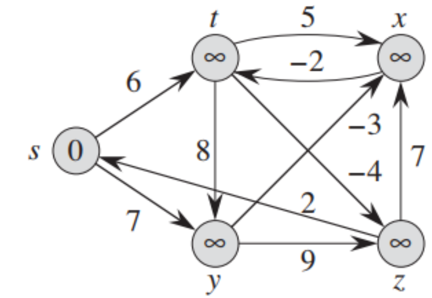
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

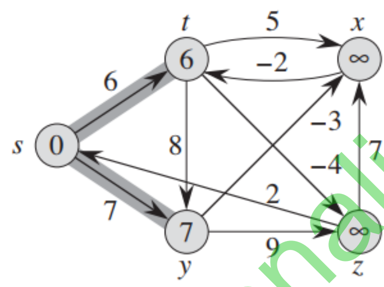
- The Bellman-Ford algorithm runs in time **O(VE),** since the initialization in line 1 takes O(V) time, each of the |V|- 1 passes over the edges in lines 2–4 takes O(E) time, and the for loop of lines 5–7 takes O(E) time.
- **Ex :**The source is vertex s. if edge (u,v) is shaded, then $v.\pi$ =u .
- **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values.
- The Bellman-Ford algorithm returns TRUE in this example.

- **BELLMAN-FORD(*G* , *w* , *s*)**

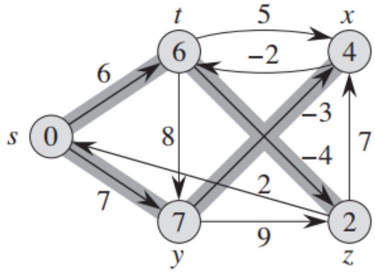
1. INITIALIZE-SINGLE-SOURCE(G,s)
2. **for** *i*=1 **to** |G.V|-1
3. **for** each edge (*u* , *v*) ∈ *G.E*
4. RELAX(u ,v , w)
5. **for** each edge (u , v) ∈ G.*E*
6. **if v.** *d* > u . *d*+ w(u,v)
7. **return** FALSE
8. **return** TRUE



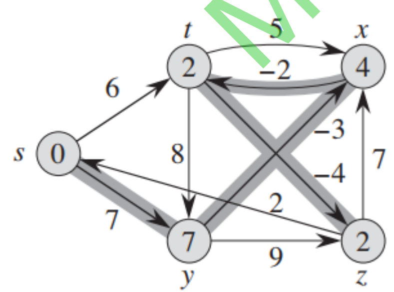
(a)



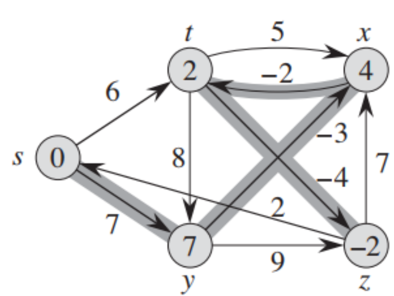
(b)



(c)



(d)



(e)

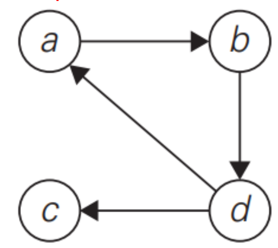
MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Transitive Closure

- Warshall ’s algorithm for computing the transitive closure of a directed graph and Floyd’s algorithm for the all-pairs shortest-paths problem.
- **Warshall ’s Algorithm**
- The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.
- Stephen Warshall, discovered it,so name is Warshall Algorithm
- Ex : (a) Digraph ,(b)Its adjacency matrix, (c) Its transitive closure
- Rule for changing zeros in Warshall’s Algorithm



(a)

$A =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

(b)

$R^{(k-1)} =$ (row k: column j = 1; row i: column j = 0 → column k = 1) $\Longrightarrow$ $R^{(k)} =$ (row k: column j = 1; row i: column j = 1, column k = 1)

$T =$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

(c)

MonalisaCS

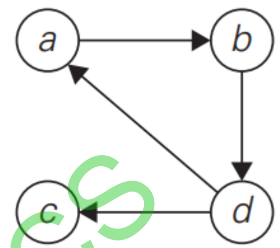
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **ALGORITHM** *Warshall(A[1..n, 1..n])*
- *//Input: The adjacency matrix A of a digraph with n vertices*
- *//Output: The transitive closure of the digraph*

1. $R^{(0)} \leftarrow A$
2. **for** $k \leftarrow 1$ **to** n **do**
3. **for** $i \leftarrow 1$ **to** n **do**
4. **for** $j \leftarrow 1$ **to** n **do**
5. $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
6. **return** $R^{(n)}$

- Its time efficiency is only $\Theta(n^3)$
- Application of Warshall's algorithm to the digraph



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)} =$

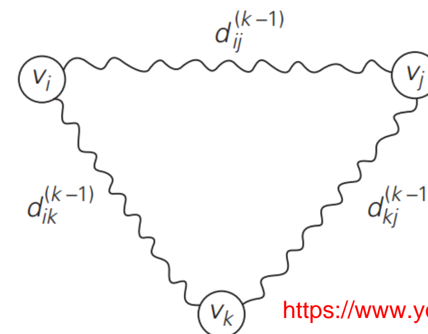
	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

MonalisaCS

https://www.youtube.com/@MonalisaCS

All-Pairs Shortest Paths

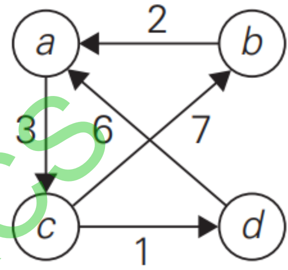
- The ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths from each vertex to all other vertices .
- We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm |V| times, once for each vertex as the source. If all edge weights are nonnegative, we can use Dijkstra's algorithm.
- If we use the linear-array implementation of the min-priority queue,the running time $O(|V|^3)$.
- The binary min-heap implementation of the min-priority queue,running time of $O(|V||E| \lg |V|)$
- In Dynamic Programming Floyd's Algorithm Used for all-pairs shortest paths problem.
- It is called ***Floyd's algorithm*** after its co-inventor Robert W.Floyd.
- It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length.
- It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix.***
- We define $d_{ij}^{(k)}$ recursively by
- $d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$ for $k \geq 1$, $d_{ij}^{(0)} = w_{ij}$.
- Underlying idea of Floyd's algorithm.



MonalisaCS

https://monalisacs.com/

- **ALGORITHM** *Floyd(W [1..n, 1..n])*
- //Input:The weight matrix *W* of a graph with no negative-length cycle
- //Output: The distance matrix of the shortest paths lengths
- $D^{(0)} \leftarrow W$
- **for** $k \leftarrow 1$ **to** n **do**
- **for** $i \leftarrow 1$ **to** n **do**
- **for** $j \leftarrow 1$ **to** n **do**
- $D^{(k)}[i,j] \leftarrow \min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k]+D^{(k-1)}[k,j]\}$
- **return** $D^{(n)}$
- The running time is determined by the triply nested **for** loops, the algorithm runs in time $\Theta(n^3)$.
- Application of Floyd's algorithm



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

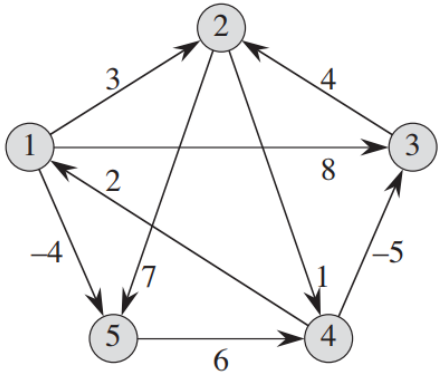
$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

MonalisaCS

https://www.youtube.com/@MonalisaCS

- Ex 2:



$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

Rod cutting

- The ***rod-cutting problem*** **:**Given a rod of length n inches and a table of prices p_i for i=1,2,3..n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i=1,2,\ldots n-1$.
- If an optimal solution cuts the rod into k pieces, for some $1\leq k\leq n$, then an optimal decomposition $n=i_1+i_2+\ldots.+i_k$ of the rod into pieces of lengths $i_1, i_2, \ldots., i_k$ provides maximum corresponding revenue
  $r_n = p_1 + p_2 + \ldots.. + p_k$.
- $r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots., r_{n-1}+r_1)$ or $r_n = \max_{1\leq i\leq n}(p_i+r_{n-i})$
- Example: A sample price table for rods. Each rod of length i inches earns p_i dollars of revenue.

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

- $r_1=1$, 1=1 no cuts — $r_2=5$, 2=2 no cuts
- $r_3=8$, 3=3 no cuts — $r_4=10$, 4=2+2
- $r_5=13$, 5=2+3 — $r_6=17$, 6=6 no cuts
- $r_7=18$, 7=1+6,2+2+3 — $r_8=22$, 8=2+6
- $r_9=25$, 9=3+6 — $r_{10}=30$, 10=10 no cuts

MonalisaCS

https://monalisacs.com/

- BOTTOM-UP-CUT-ROD(p , n)

```
1. let r[0…n] be a new array
2. r[0]= 0
3. for j=1 to n
4.       q=-∞
5.       for i =1 to j
6.            q = max(q, p[i]+r[j-i])
7.       r[j]= q
8. return r[n]
```

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

r_1=1 , 1=1 no cuts
r_2=5 , 2=2 no cuts
r_3=8 , 3=3 no cuts
r_4=10, 4=2+2
r_5=13, 5=2+3
r_6=17 , 6=6 no cuts
r_7=18, 7=1+6,2+2+3
r_8=22 , 8=2+6
r_9=25, 9=3+6
r_{10}=30 , 10=10 no cuts

- Line 1 creates a new array r[0 …n] in which to save the results of the subproblems,
- Line 2 initializes r[0] to 0, since a rod of length 0 earns no revenue.
- Lines 3–6 solve each subproblem of size j , for j= 1, 2,….. n, in order of increasing size.
- Line 7 saves in r[j] the solution to the subproblem of size j .
- Finally, line 8 returns r[n], which equals the optimal value r_n.
- The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure.

MonalisaCS

https://www.youtube.com/@MonalisaCS

Matrix-chain multiplication

- We are given a sequence (chain) $\{A_1, A_2, \dots A_n\}$ of n matrices to be multiplied,
- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- MATRIX-MULTIPLY(A, B)

```
1. if A . columns ≠ B . Rows
2.        error "incompatible dimensions"
3. else let C be a new A.rows ×B.columns matrix
4.     for i= 1 to A.rows
5.        for j = 1 to B.columns
6.            c_ij = 0
7.            for k =1 to A.columns
8.              c_ij = c_ij + a_ik . b_kj
9. return C
```

- We can multiply two matrices A and B only if they are compatible:The number of columns of **A** must equal the number of rows of **B**.
- If A=p× q matrix and B=q× r matrix, the resulting matrix C =p× r matrix.
- A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.
- If n=Number of Matrices-1
- Number of distinct parenthesizations possible $= \frac{{}^{2n}C_n}{n+1}$ [Catalan number]
- For ex , $\{A_1, A_2, A_3, A_4\}$, then five distinct ways:

MonalisaCS

https://monalisacs.com/

- $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.
- Ex: consider the problem of a chain $\{A_1, A_2, A_3\}$ of three matrices.the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively.
- $((A_1,A_2)A_3)$:10*100*5=5000 scalar multiplications to compute 10×5 matrix product A_1A_2, plus another 10*5*50= 2500 scalar multiplications to multiply this matrix by A_3, total of 7500.
- $(A_1(A_2A_3)$:100*5*50+10*100*50=75000 scalar multiplications .
- We shall implement the tabular, bottom-up method in the procedure MATRIXCHAIN-ORDER.
- This procedure assumes that matrix A_i has dimensions $p_{i-1}\times p_i$ for i=1,2,….n. Its input is a sequence $p=\{p_0, p_1, \ldots . p_n\}$, where p.*length*=n+1.
- The procedure uses an auxiliary table m[1..n,1…n] for storing the m[i,j] costs and another auxiliary table s[1…n-1,2…n] that records which index of k achieved the optimal cost in computing m[i,j] . We shall use the table s to construct an optimal solution.
- The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle.
- $m[i,j]=0$ if i=j,
- $= \min_{i\le k<j} \{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$ if $i<j$.

MonalisaCS

https://www.youtube.com/@MonalisaCS

Matrix	A_1	A_2	A_3	A_4
Dimension	3×2	2 ×4	4×2	2×5

- MATRIX-CHAIN-ORDER(p)

1. $n = p.length - 1$
2. let $m[1..n,1..n]$ and $s[1..n-1,2..n]$ be new tables
3. **for** $i = 1$ **to** n
4. $m[i,i] = 0$
5. **for** $l = 2$ **to** n
6. **for** $i = 1$ **to** $n-l+1$
7. $j = i + l - 1$, $m[i, j]=\infty$
8. **for** $k = i$ **to** $j-1$
9. $q = m[i,k]+m[k+1,j]+p_{i-1}\,p_k\,p_j$
10. **if** $q < m[i, j]$
11. $m[i, j] = q$, $s[i, j] = k$
12. **return** m and s

m (rows i, columns j):

i \ j	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

s (rows i, columns j):

i \ j	2	3	4
1	1	1	3
2		2	3
3			3

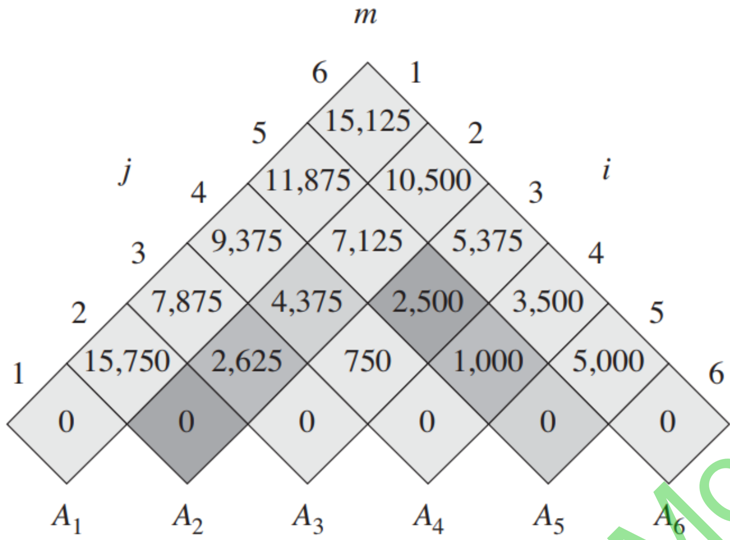
- $m[1,2]=3*2*4=24$
- $m[2,3]=2*4*2=16$
- $m[3,4]=4*2*5=40$
- $m[1,3]=\text{Min}$
- $\{k=1: m[1,1]+m[2,3] + p_0 p_1 p_3 = 0+16+3*2*2=28,$
- $k=2: m[1,2]+m[3,3]+p_0 p_2 p_3 = 24+0+3*4*2=48\} = 28$

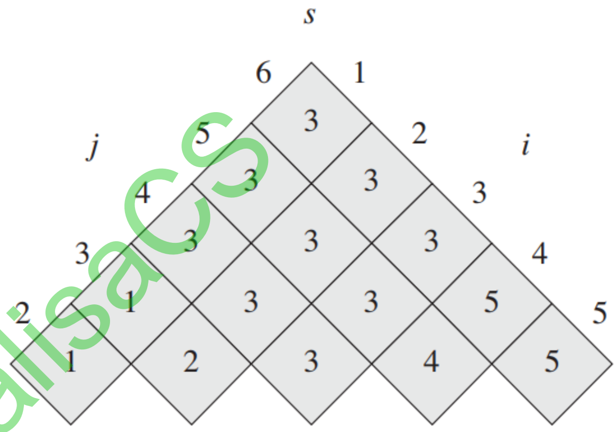
- The nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm .
- The algorithm requires $\Theta(n^2)$ space to store the m and s tables.
- $m[2,4]=\text{Min}\{k=2: m[2,2]+m[3,4] + p_1 p_2 p_4 = 0+40+2*4*5=80,$
- $k=3: m[2,3]+m[4,4]+p_1 p_3 p_4 = 16+0+2*2*5=36\} = 36$
- $m[1,4]=\text{Min}\{k=1: m[1,1]+m[2,4] + p_0 p_1 p_4 = 0+36+3*2*5=66,$
- $k=2: m[1,2]+m[3,4]+p_0 p_2 p_4 = 24+40+3*4*5=124$
- $k=3: m[1,3]+m[4,4] + p_0 p_3 p_4 = 28+0+3*2*5=58\} = 58$
- $(A_1A_2A_3A_4)=((A_1A_2A_3)A_4)=((A_1(A_2\,A_3))\,A_4)$

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Matrix A_1 A_2 A_3 A_4 A_5 A_6
- Dimension 30×35 35×15 15×5 5×10 10×20 20×25
- The m and s tables computed by MATRIX-CHAIN-ORDER for n = 6
-





MonalisaCS

- $$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
  $$= 7125 .$$
- $(A_1A_2A_3A_4A_5A_6) = ((A_1(A_2A_3))A_4A_5A_6) = ((A_1(A_2A_3))((A_4A_5)A_6))$

https://www.youtube.com/@MonalisaCS

Longest common subsequence

- A ***substring*** is a contiguous sequence of characters within a string.
- A ***subsequence*** is a sequence that can be derived from the given sequence by deleting some or no elements without changing the order of the remaining elements.
- A string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0.
- Biological applications often need to compare the DNA of two (or more) different organisms.
- Given two sequences X and Y , we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y .
- For example, if X=(A,B,C,B,D,A,B) and Y=(B,D,C,A,B,A), the sequence (B,C,A) is a common subsequence of both X and Y .
- The sequence (B,C,A) is not a *longest* common subsequence (LCS) of X and Y , since it has length 3 and the sequence (B,C,B,A),(B,D,A,B) which are also common to both X and Y , has length 4.
- In the ***longest-common-subsequence problem***, we are given two sequences $X=(x_1,x_2,\ldots.x_m)$ and $Y=(y_1,y_2,\ldots.y_n)$ and wish to find a maximum length common subsequence of X and Y .
- ***Optimal substructure of an LCS***
- Let $X=(x_1,x_2\ldots x_m)$ and $Y=(y_1,y_2\ldots y_n)$ be sequences, and let $Z=(z_1,z_2,\ldots.z_k)$ be any LCS of X and Y.

1. If $x_m=y_n$, then $z_k=x_m=y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m\neq y_n$, then $z_k\neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m\neq y_n$, then, $z_k\neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

MonalisaCS

- Ex: Lets take X=(ABBAB) and Y=(ACBAB)
- LCS ("ABBAB" , "ACBAB")
- If $x_5=y_5$, then $z_4=x_5=y_5$=B and Z_3 is an LCS of X_4 and Y_4.
- LCS ("ABBA" , "ACBA")
- If $x_4=y_4$, then $z_3=x_4=y_4$=A and Z_2 is an LCS of X_3 and Y_3.
- LCS ("ABB" , "ACB")
- If $x_3=y_3$, then $z_2=x_3=y_3$=B and Z_1 is an LCS of X_2 and Y_2.
- LCS ("AB" , "AC")
- If $x_2 \neq y_2$, then $z_1 \neq x_2$ implies that Z is an LCS of X_1 and Y .
- LCS ("A" , "AC")
- If $x_1 \neq y_2$, then, $z_1 \neq y_2$ implies that Z is an LCS of X and Y_1.
- LCS ("A" , "A")
- If $x_1=y_1$, then $z_1=x_1=y_1$ =A.
- LCS="ABAB"
- **A recursive solution**
- Let us define c[*i, j*] to be the length of an LCS of the sequences X_i and Y_j .
- c[*i,j*]= {0 if i=0 or j=0,
- {c[*i*-1,*j*-1]+1 if *i,j* >0 and $x_i=y_j$,
- {max(c[*i,j*-1],c[*i*-1,*j*]) if *i,j* >0 and $x_i \neq y_j$.
- **Computing the length of an LCS**
- It stores the c[i,j] values in a table c[0..m,0..n] and the table b[1…m,1..n] help us construct an optimal solution.
- It computes the entries in ***row-major*** order.

MonalisaCS

https://monalisacs.com/

- LCS-LENGTH(X,Y)

```
1.  m=X.length ,n=Y.length
2.  let b[1..m,1..n] and c[0..m,0..n] be new tables
3.  for i = 1 to m
4.          c[i,0]=0
5.  for j =0 to n
6.          c[0, j]=0
7.  for i=1 to m
8.      for j=1 to n
9.          if x_i = = y_j
10.             c[i,j]= c[i-1,j-1]+1
11.             b[i,j]= "↖"
12.         elseif c[i-1,j]≥ c[i,j-1]
13.             c[i,j]= c[i-1,j]
14.             b[i,j]= "↑"
15.         else c[i,j]= c[i,j-1]
16.             b[i,j]= "←"
17. return c and b
```

- Ex: The tables produced sequences X=(A,B,C,B,D,A,B) and Y=(B,D,C,A,B,A)

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	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

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j. \end{cases}$$

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- The running time of the procedure is Θ(mn), each table entry takes Θ (1) time to compute.
- **Constructing an LCS**
- We simply begin at b[m,n] and trace through the table by following the arrows.
- Whenever we encounter a " ↖" in entry b[*i*,*j*] , it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found.
- With this method, we encounter the elements of this LCS in reverse order.
- The following recursive procedure prints out an LCS of X and Y in the proper , forward order.
- The procedure takes time O(m+n), since it decrements at least one of i and j in each recursive call.
- PRINT-LCS(*b, X,i, j*)

```
1. if i == 0 or j == 0
2.         return
3. if b[i, j] == "↖"
4.         PRINT-LCS(b, X, i-1, j-1)
5.         print x_i
6. elseif b[i,j]= ="↑"
7.         PRINT-LCS(b, X, i-1, j)
8. else PRINT-LCS(b, X, i, j-1)
```

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	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

- This procedure prints
- BCBA,BCAB,BDAB

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Determine an LCS of ⟨1, 0, 0, 1, 0, 1, 0, 1⟩ and ⟨0, 1, 0, 1, 1, 0, 1, 1, 0⟩

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j. \end{cases}$$

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	0	1	0	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ 1	← 1	↖ 1	↖ 1	← 1	↖ 1	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ 4	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ 5	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ 6
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

- LCS= 100110
- 101011
- 101101
- 010101
- 101010

MonalisaCS

https://www.youtube.com/@MonalisaCS

Optimal Binary Search Tree

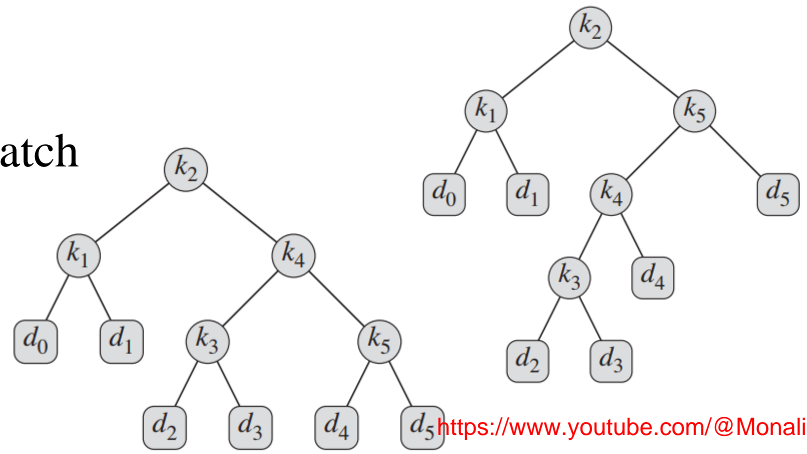
- We are given a sequence $K=\{k_1,k_2...k_n\}$ of n distinct keys in sorted order, and we wish to build a binary search tree from these keys.
- For each key k_i, we have a probability p_i . Some searches may be for values not in K, and so we also have n+1 “dummy keys” $d_0,d_1,d_2...d_n$ representing values not in K.
- In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for i=1,2…,n-1, the dummy key d_i represents all values between k_i and k_{i+1}.
- For each dummy key d_i, we have a probability q_i that a search will correspond to d_i.
- Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have $\sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i=1$.
- For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an **optimal binary search tree**
- The total number of binary search trees with n keys is equal to the nth ***Catalan number***,
- $C(n)=\frac{1}{n+1}\, {}^{2n}C_n$ for n>0.
- Two binary search trees for a set of n = 5 keys with the following probabilities:
-

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

MonalisaCS

- **(a)** A binary search tree with expected search cost 2.80.
- **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.
- **A recursive solution:**
- Let us define e[i,j] as the expected cost of searching an optimal binary search tree containing the keys $k_i \ldots k_j$. Ultimately, we wish to compute e[1,n].
- $e[i,j] = \{ q_{i-1}$ if $j = i-1$
- $\{ \min_{i \le r \le j} \{ e[i,r-1] + e[r+1,j] + w(i,j) \}$ if $i \le j$.
- We also use a table *root*[i,j] , for recording the root of the subtree.
- We will need one other table for efficiency.
- Rather than compute the value of $w[i, j]$ from scratch every time we store these values in a table w[1…n+1,0..n].
- For the base case, $w[i,i-1] = q_{i-1}$ for $1 \le i \le n+1$.
- $w[i,j] = w[i,j-1] + p_j + q_j$ For $j \ge$ I

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



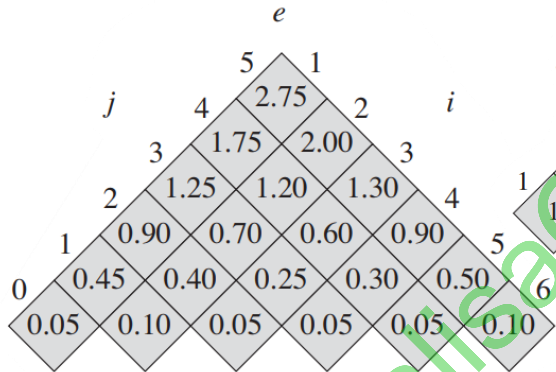
MonalisaCS
https://monalisacs.com/
https://www.youtube.com/@MonalisaCS

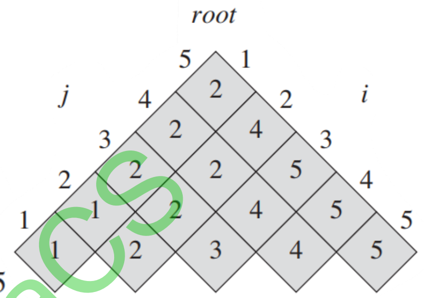
- **OPTIMAL-BST** (p,q,n)

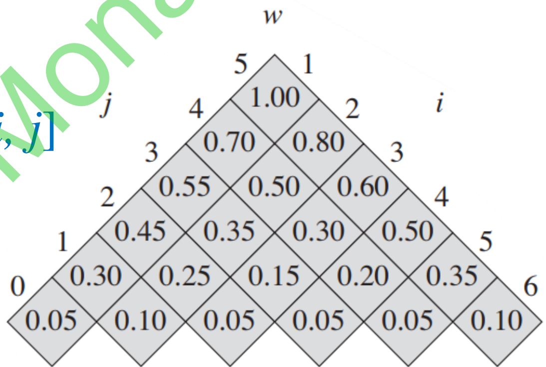
1. let e[1…n+1,0..n],w[1…n+1,0…n],and *root*[1…n,1…n] be new tables
2. **for** i=1 **to** n + 1
3. $e[i,i-1]=q_{i-1}$
4. $w[i,i-1]=q_{i-1}$
5. **for** l =1 **to** n
6. **for** i =1 **to** n- l + 1
7. $j = i + l - 1$
8. $e[i,j]=\infty$
9. $w[i,j]=w[i,j-1]+p_j+q_j$
10. **for** r = i **to** j
11. t= $e[i,r-1]+e[r+1,j]+ w[i,j]$
12. **if** t < e[i , j]
13. e[i,j]= t
14. *root*[i,j]=r
15. **return** e and *root*

- The OPTIMAL-BST procedure takes $\Theta(n^3)$ time

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10







MonalisaCS
https://monalisacs.com/
https://www.youtube.com/@MonalisaCS