* Longest Common Subsequence *

w_1 = a b c d

$2^n = 2^4 = 16$ subsequences possible

write characters of string in increasing order of their positions.

= ab, cd, bd, ~~ca~~, ac, ~~db~~, bd, acd, bcd, abcd.

w_2 = bcd

= b, c, d, bc, cd, bd, bcd.

Q. X = a b a a b a

Y = b a b b a b

(i)

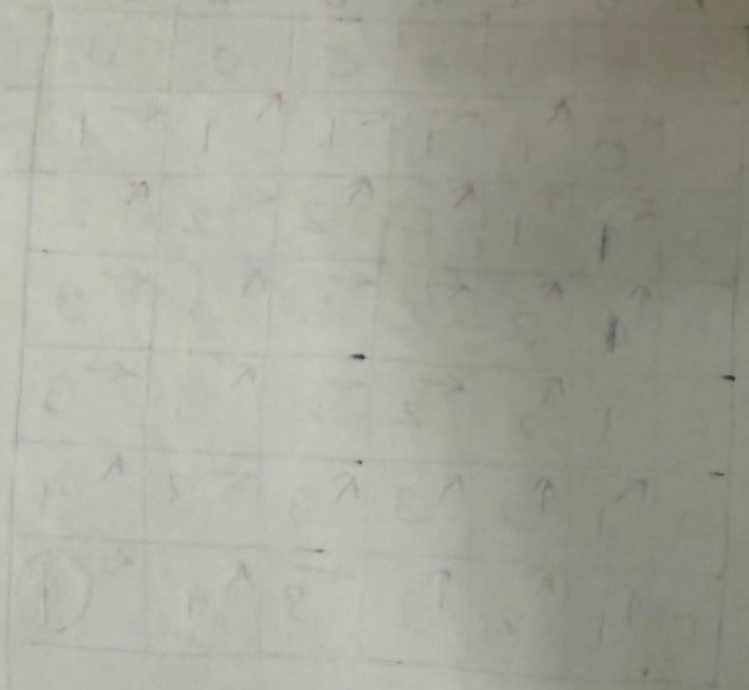
X ↓ \ Y → (j)	^	b	a	b	b	a	b
^	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1
b	0	0	1	2	2	2	2
a	0	0	2	2	2	3	3
a	0	1	2	2	2	3	3
b	0	1	2	3	3	3	4
a	0	1	2	3	3	4	4

i.e length of subsequence is 4.

baba

S1 { B, C, D, A, A, C, D }

S2 { A, C, D, B, A, C }



Longest Common Subsequence

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 S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a **strictly increasing sequence** of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

If

```
S1 = {B, C, D, A, A, C, D}
```

Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

If

```
S1 = {B, C, D, A, A, C, D}
S2 = {A, C, D, B, A, C}
```

Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...

Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Before proceeding further, if you do not already know about dynamic programming, please go through dynamic programming.

Using Dynamic Programming to find the LCS

Let us take two sequences:

X

A	C	A	D	B

The first sequence

Y

C	B	D	A

Second Sequence

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

Initialise a table

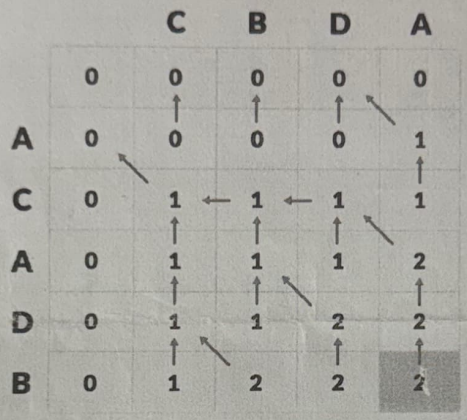
2. Fill each cell of the table using the following logic.
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

Fill the values

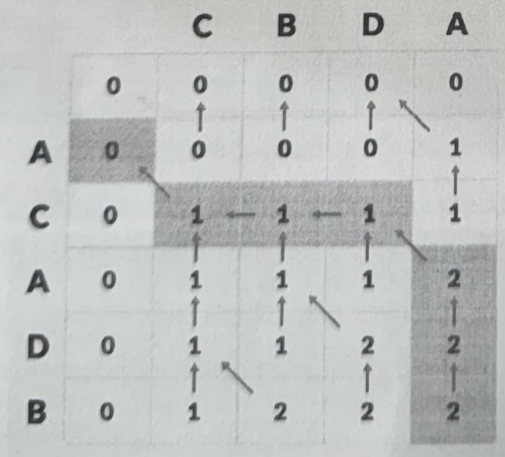
		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

5. **Step 2** is repeated until the table is filled. Fill all the values
6. The value in the last row and the last column is the length of the longest common

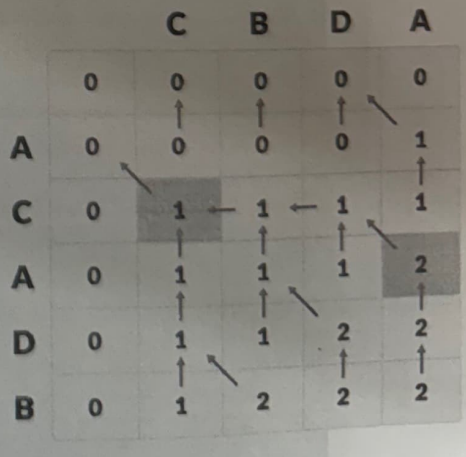


subsequence.

7. The bottom right corner is the length of the LCS
8. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Select the cells with diagonal arrows



Create a path according to the arrows

Thus, the longest common subsequence is CA.

C	A

LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)).

Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
  If X[i] = Y[j]
     LCS[i][j] = 1 + LCS[i-1, j-1]
     Point an arrow to LCS[i][j]
  Else
     LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
     Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Python, Java and C/C++ Examples

```
# The longest common subsequence in Python

# Function to find lcs_algo
def lcs_algo(S1, S2, m, n):
  L = [[0 for x in range(n+1)] for x in range(m+1)]

  # Building the mtrix in bottom-up way
  for i in range(m+1):
     for j in range(n+1):
```

```
        if i == 0 or j == 0:
            L[i][j] = 0
        elif S1[i-1] == S2[j-1]:
            L[i][j] = L[i-1][j-1] + 1
        else:
            L[i][j] = max(L[i-1][j], L[i][j-1])

    index = L[m][n]

    lcs_algo = [""] * (index+1)
    lcs_algo[index] = ""

    i = m
    j = n
    while i > 0 and j > 0:

        if S1[i-1] == S2[j-1]:
            lcs_algo[index-1] = S1[i-1]
            i -= 1
            j -= 1
            index -= 1

        elif L[i-1][j] > L[i][j-1]:
            i -= 1
        else:
            j -= 1

    # Printing the sub sequences
    print("S1 : " + S1 + "\nS2 : " + S2)
    print("LCS: " + "".join(lcs_algo))


S1 = "ACADB"
S2 = "CBDA"
m = len(S1)
n = len(S2)
lcs_algo(S1, S2, m, n)
```

Longest Common Subsequence Applications

1. in compressing genome resequencing data
2. to authenticate users within their mobile phone through in-air signatures

The Naive String Matching Algorithm

The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].
The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.....m] = T [s+1.....s+m] for each of the n - m +1 possible value of s.

NAIVE-STRING-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

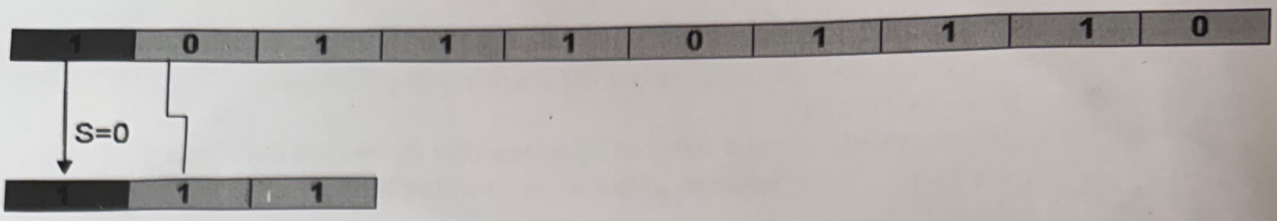
Analysis: This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

Example:
Suppose T = 1011101110

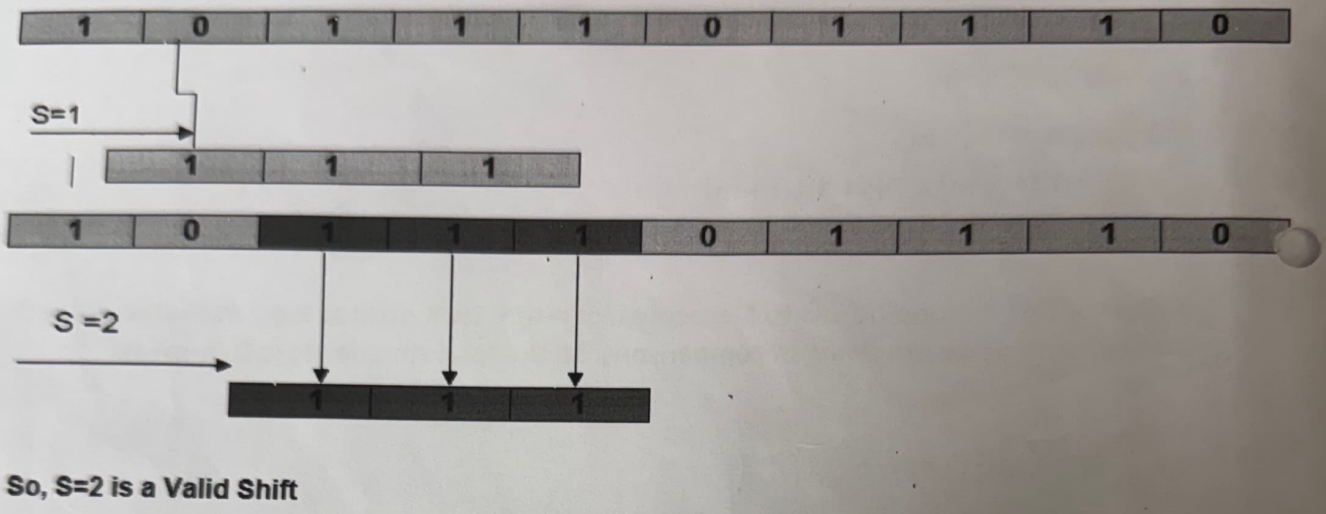
1. P = 111
2. Find all the Valid Shift

Solution:

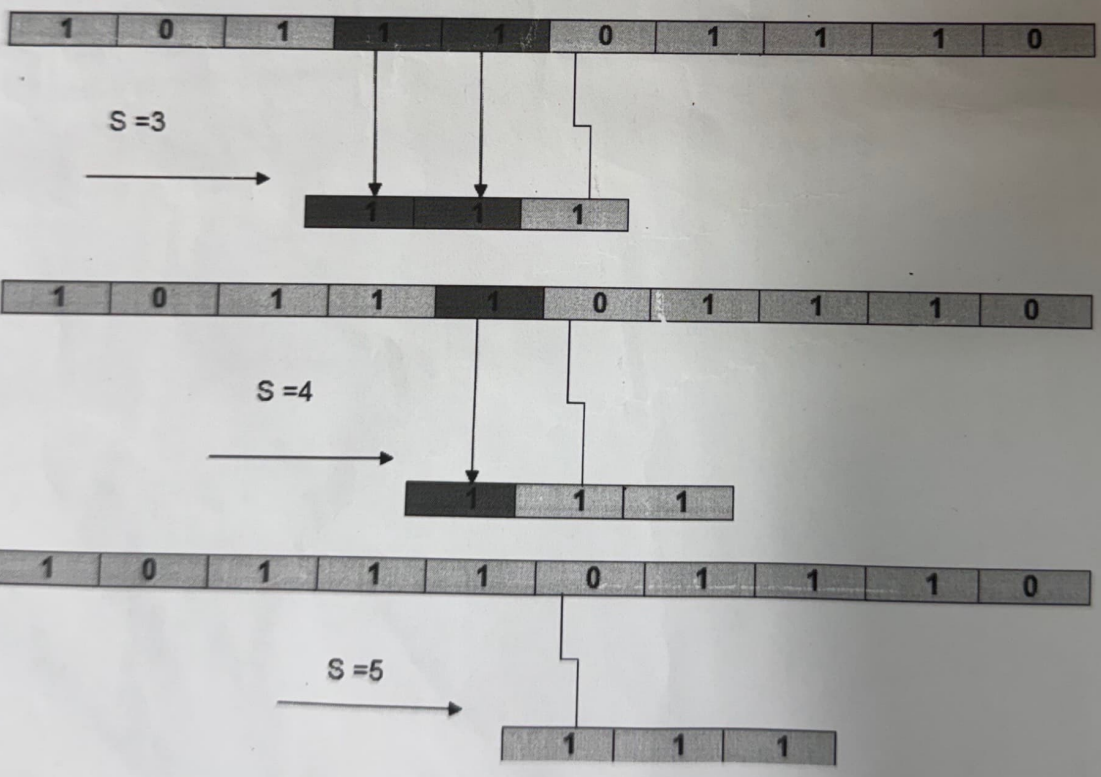
T = Text



P = Pattern



So, S=2 is a Valid Shift



- Naive pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern.
- Naive algorithm is exact string matching (means finding one or all exact occurrences of a pattern in a text) algorithm.
- This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.
- The naive approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].It returns all the valid shifts found.
-

Input: txt[] = "THIS IS STRING MATCHING ALGORITHM"
pat[] = "STRING"
Output: Pattern found at position 10

Example 3:

Input:

Main String: "ABAAABCDBBABCDDEBCABC"

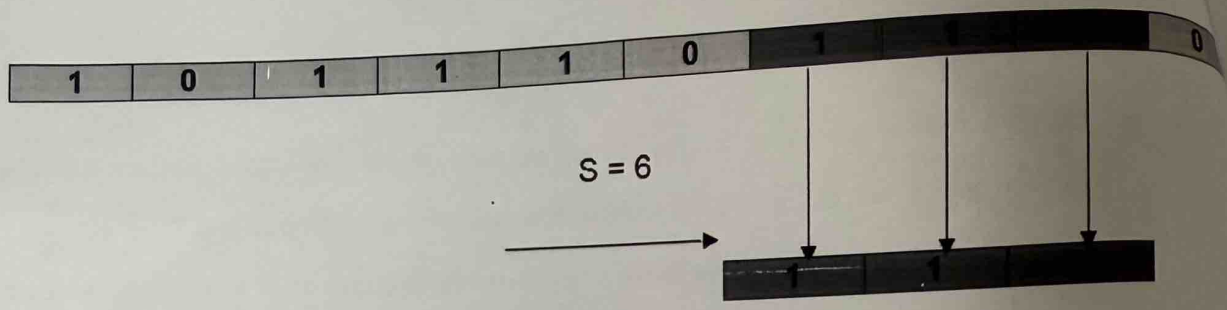
pattern: "ABC"

Output:

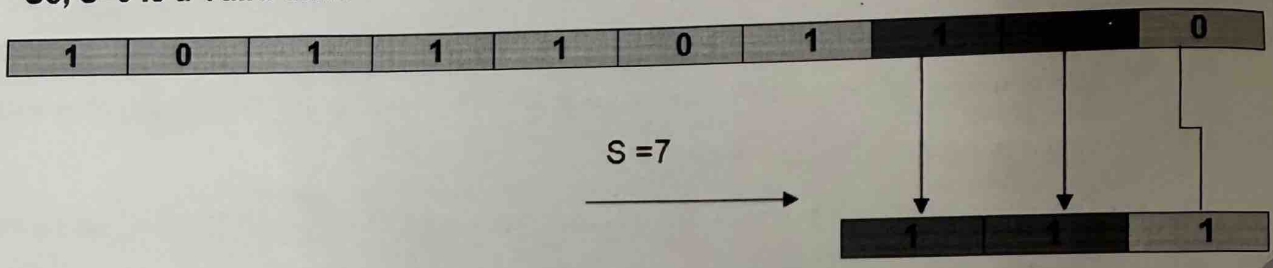
Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18



So, S=6 is a Valid Shift



Disadvantage:

Naive method is inefficient because information from a shift is not used again.

Problem with Naive Algorithm

Suppose T=cabababcd and P=ababc

T: c a b a b a b c d

P: a...

P: a b a b c

P: a...

P: a b a b c

- Whenever a character mismatch occurs after matching of several characters, the comparison begins by going back in from the character which follows the last beginning character.

* Rabin-Karp Algorithm *

Text : 1 2 3 4 5 6
n=6 a a a a a b

Pat : 1 2 3
m=3 a a b

1 + 1 + 2 = 4
h(p) ↑ Hash code

a - 1
b - 2
c - 3
d - 4
e - 5
f - 6
g - 7
h - 8
i - 9
j - 10

a a a a a b

1+1+1 = 3

Does not match with h(p) i.e 4.
So, slide to next set of 3 alphabets.

a a a a a b
1+1+1 = 3 No match.

a a a a a b No match.
1+1+1=3

a a a a a b Match.
1+1+2 = 4

After matching, perform comparison.

Example 2 :- a b c d a b c e

Pat : b c e
2+3+5 = 10

a b c d a b c e
1+2+3 = 6 No match.

Now, slide & get next set

so, 6-1 = 5+4 i.e a b c d a b c e
= 9 2+3+4 = 9 No match.

match alphabets now
3+4+1 = 8 No match.
4+1+2 = 7 No match.
1+2+3 = 6 No match.
2+3+5 = 10 match.

Drawback of :-

Text :- c c a c c a a e d b a

Pat :- d b a

4 + 2 + 1 = 7

c c a c c a a e d b a

3 + 3 + 1 = 7 match, but alphabets are different.

3 + 1 + 3 = 7, match - " -

1 + 3 + 3 = 7 match - " -

many matching, but patter not found, so these are called as spurious hits.

maximum time taken by algo is $\theta(mn)$.

Rabin-Karp Algorithm

In this tutorial, you will learn what rabin-karp algoroithm is. Also, you will find working examples of rabin-karp algorithm in C, C++, Java and Python.

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison.

How Rabin-Karp Algorithm Works?

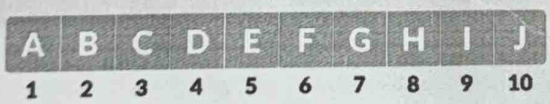
A sequence of characters is taken and checked for the possibility of the presence of the required string. If the possibility is found then, character matching is performed.

Let us understand the algorithm with the following steps:

1. Let the text be:

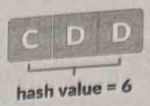
2. And the string to be searched in the above text be: Pattern

3. Let us assign a `numerical value(v)/weight` for the characters we will be using in the problem. Here, we have taken first ten alphabets only (i.e. A to J).



4. `n` be the length of the pattern and `m` be the length of the text. Here, `m = 10` and `n = 3`.

   Let `d` be the number of characters in the input set. Here, we have taken input set {A, B, C, ..., J}. So, `d = 10`. You can assume any suitable value for `d`.



5. Let us calculate the hash value of the pattern:

```
hash value for pattern(p) = Σ(v * dm-1) mod 13
                          = ((3 * 10²) + (4 * 10¹) + (4 * 10⁰)) mod 13
                          = 344 mod 13
                          = 6
```

In the calculation above, choose a prime number (here, 13) in such a way that we can perform all the calculations with single-precision arithmetic.

The reason for calculating the modulus is given below.

5. Calculate the hash value for the text-window of size `m`.

```
For the first window ABC,

hash value for text(t) = Σ(v * dm-1) mod 13

                       = ((1 * 10²) + (2 * 10¹) + (3 * 10⁰)) mod 13
                       = 123 mod 13
                       = 6
```

6. Compare the hash value of the pattern with the hash value of the text. If they match then, character-matching is performed.
   In the above examples, the hash value of the first window (i.e. `t`) matches with `p` so, go for character matching between ABC and CDD. Since they do not match so, go for the next window.
7. We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.

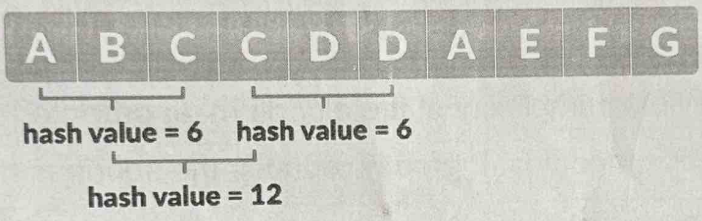
```
t = ((1 * 10²) + ((2 * 10¹) + (3 * 10⁰)) * 10 + (3 * 10⁰)) mod 13
  = 233 mod 13
  = 12
```

In order to optimize this process, we make use of the previous hash value in the following way.

```
t = ((d * (t - v[character to be removed] * h) + v[character to be added].)
mod 13

  = ((10 * (6 - 1 * 9) + 3 )mod 13

  = 12

Where, h = d^(m-1) = 10^(3-1) = 100.
```

8. For BCC, t = 12 (≠6). Therefore, go for the next window.

   After a few searches, we will get the match for the window CDA in the text.



Hash value of different windows

Algorithm

```
n = t.length
m = p.length
h = dm-1 mod q
p = 0
t0 = 0
for i = 1 to m
    p = (dp + p[i]) mod q
    t0 = (dt0 + t[i]) mod q
for s = 0 to n - m
    if p = ts
```

```
        if p[1.....m] = t[s + 1..... s + m]
            print "pattern found at position" s
    If s < n-m
        ts + 1 = (d (ts - t[s + 1]h) + t[s + m + 1]) mod q
```

Limitations of Rabin-Karp Algorithm

Spurious Hit

When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit.

Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use modulus. It greatly reduces the spurious hit.

Rabin-Karp Algorithm Complexity

The average case and best case complexity of Rabin-Karp algorithm is `O(m + n)` and the worst case complexity is `O(mn)`.
The worst-case complexity occurs when spurious hits occur a number for all the windows.

Rabin-Karp Algorithm Applications

- For pattern matching
- For searching string in a bigger text

text → a b c d e f g h

pat → d e f

* Knuth-Morris-Pratt algorithm *

text → a b c d a b c a b c d f
i

pat → a b c d f
j

worst case of Basic Search :-

text → a a a a a a a b

pat → a a a b

$O(nm)$.

Pattern a b c d a b c
1 2 3 4 5 6 7

prefix : a, ab, abc, abcd ....

suffix : c, bc, abc, dabc ....
start from RHS of string

Is there any prefix same as suffix?

abc is the prefix which is also there in suffix.

Example :-

P_1 = a b c d a b e a b f
1 2 3 4 5 6 7 8 9 10
0 0 0 0 1 2 0 1 2 0

P_2 = a b c d e a b f a b c
0 0 0 0 0 1 2 0 1 2 3

P_3 = a a b c a d a a b e
0 1 0 0 1 0 1 1 2 0

P_4 = a a a a b a a c d
0 1 2 3 0 1 2 0 0

Π table or LPS table.

String :-

	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

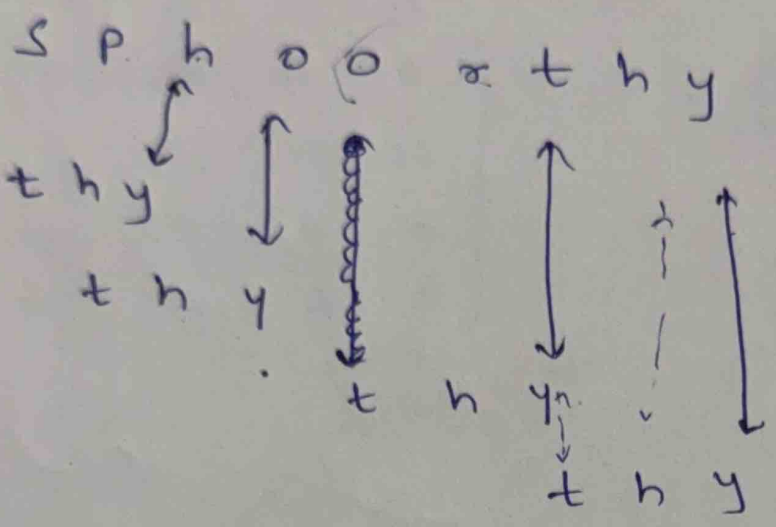
Pattern :-

	1	2	3	4	5
	a	b	a	b	d
0	0	0	1	2	0

Compare string of i with $j+1$, if it matches, move $i+1$ & $j+1$

1, 6, 4,
15, 13, 20, 12, 14
24, 25, 27, 22,
31, 37, 38, 32, 36, 39
44, 47, 45,
56, 57, 59
65, 64, 63, 68, 67, 62,
73

* Boyce Moore Algorithm *.



0	1	2	
t	h	y	*
2	1	1	3

max (1, PI-index-1)

T h i s i s a t e s t t

T e s t

0	1	2	3	
t	e	s	t	*
3 1	2	1	1	3

max(1, 4-0-1)

max (1,3) = 3

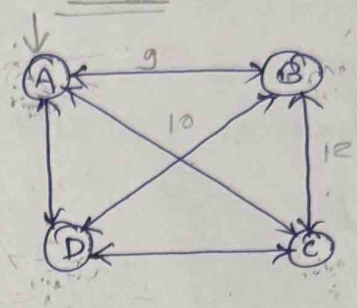
* Travelling Salesman Problem *

DP

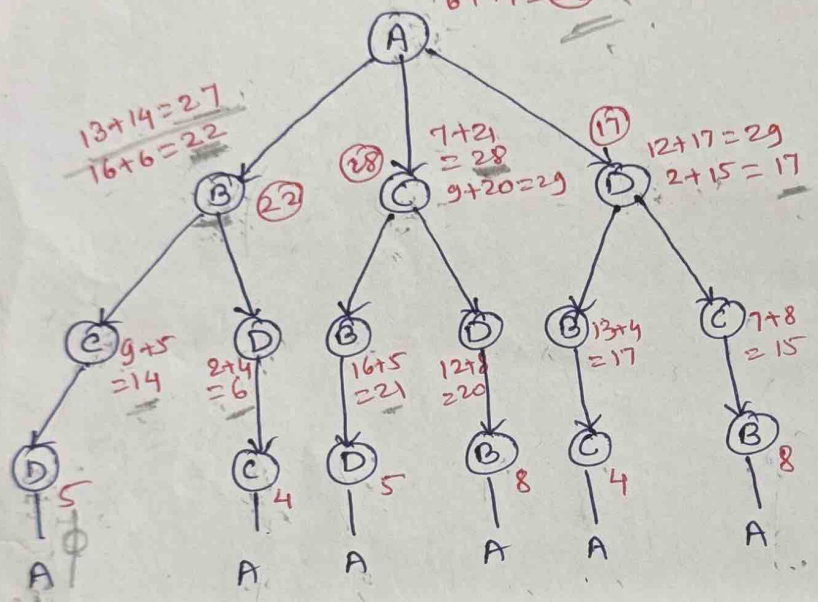
Buy

origin

	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0



11+28 = 39
16+22 = 38
6+17 = 23



i = starting vertex.

S =

$g(x, \phi) = w(x, i)$ ← starting vertex.

A → D → C → B → A

$$g(i,S) = \min_{j \in S} \left[w(i,j) + g(j, \{S-j\}) \right].$$

$$g(A, \{B,C,D\}) = \min \left[w(A,B) + g(B, \{C,D\}) \right] =$$
$$w(A,C) + g(C, \{B,D\})$$
$$w(A,D) + g(D, \{B,C\}) \Big]$$

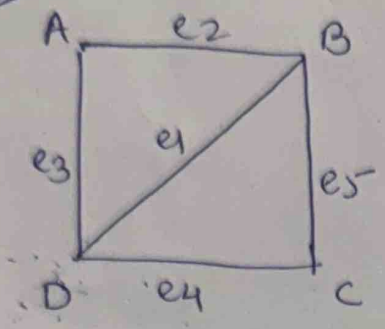
$$g(B, \{C,D\}) = \min \left[w(B,C) + g(C, \{D\}) = 13 + 14 = 27 \right.$$
$$\left. w(B,D) + g(D,C) \right].$$
$$16 \qquad 2$$

$$g(C,D) = \min \left[w(C,D) + g(D, \phi) \right]$$
$$= w(C,D) + w(D,A)$$
$$g(C,D) = 9 + 5 = 14.$$

Vertex Covering

A subset of V is called a vertex covering of G if every edge of G is incident with or covered by a vertex in subset of V.

Eg.1.



$V = \{A, B, C, D\}$

$K1 = \{B, D\} \rightarrow$ m.v.c (minimum vertex covering)

$K2 = \{A, B, C\} \rightarrow$ m.v.c

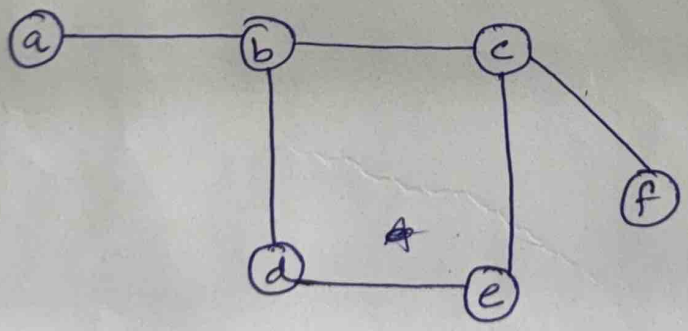
$K3 = \{A, D, C\} \rightarrow$ m.v.c

$K4 = \{A, C\} \rightarrow$ No vertex covering

$K5 = \{A, B, C, D\} \rightarrow$ NO m.v.c

minimum vertex covering → K1 = 2 vertex.

Eg.2.



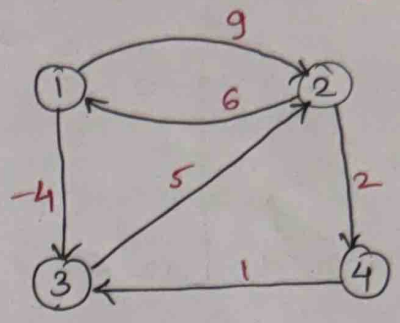
minimum Vertex cover :- {B, C, D} or {B, C, E}.

All pair shortest path (Floyd Warshall Algo.)

Distance matrix

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 9 & -4 & \infty \\ 6 & 0 & \infty & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

Base matrix



Working: Active row & column.

middle vertex

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

Calculating D(2,3) via 1:

So, $D^0[2,3]$ $D^0[2,1] + D^0[1,3]$

$\infty > 6 + (-4) = 2$

$D^0[2,4]$ $D^0[2,1] + D^0[1,4]$

$2 < 6 + \infty$

$D^0[3,2]$ $D^0[3,1] + D^0[1,2]$

$\infty + 9$

Find 5 $5 < \infty$

$D^0[3,4]$, $D^0[4,2]$, $D^0[4,3]$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$D^1[1,3]$ $D^1[1,4]$, $D^1[3,1]$ $D^1[4,3]$, $D^1[3,4]$ $D^1[4,3]$

Calculating D(1,3) via D2.

$D^1[1,3]$: $D^1[1,2] + D^1[2,3]$

-4 $9 + 2$

$-4 < 11$

$D^1[1,4]$: $D^1[1,2] + D^1[2,4]$

$\infty > 9 + 2 = 11$

$D^1[3,1]$ $D^1[3,2] + D^1[2,1]$

$\infty > 5 + 6 = 11$

$D^1[3,4]$ $D^1[3,2] + D^1[2,4]$

$\infty > 5 + 2 = 7$

$D^1[4,1]$ $D^1[4,2] + D^1[2,1]$

∞ $\infty + 6$

$\infty = \infty$

$D^1[4,3]$ $D^1[4,2] + D^1[2,3]$

$1 < \infty + 2$

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix}$$

$D^2[1,2]$ $D^2[1,4]$, $D^2[2,1]$ $D^2[2,4]$, $D^2[4,1]$ $D^2[4,4]$

$D^2[1,2]$ $D^2[1,3] + D^2[3,2]$

$9 > -4 + 5$

$9 > 1$

$D^2[1,4]$ $D^2[1,3] + D^2[3,4]$

11 $-4 + 7$

$11 > 3$

$D^2[2,1]$ $D^2[2,3] + D^2[3,1]$

$6 < 2 + 11$

$D^2[2,4]$ $D^2[2,3] + D^2[3,4]$

$2 < 2 + 7$

$D^2[4,1]$ $D^2[4,3] + D^2[3,1]$

$\infty > 1 + 11$

$D^2[4,2]$ $D^2[4,3] + D^2[3,2]$

$\infty > 1+5$

$D^4 =$

	1	2	3	4
1	0	1	-4	3
2	6	0	2	2
3	11	5	0	7
4	12	6	1	0

$D^4[1,2]\ D^4[1,3]$
$D^4[2,1]\ D^4[2,3]$
$D^4[3,1]\ D^4[3,2]$

$D^3[1,2]$ $D^3[1,4] + D^3[4,2]$

$1 < 3+6$

$D^3[1,3]$ $D^3[1,4] + D^3[4,3]$

$-4 < 3+1$

$D^3[$

$D^k[i,j] = \min[D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j]]$

~~$D^3[1,2] = \min[D^3[1,4], D^3[$~~

$D^4[1,2] = D^3[1,2]$, $D^3[1,4] + D^3[4,2]$

$1 < 3+6 = 9$

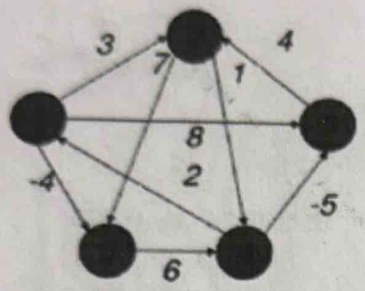
$D^4[1,3] = D^3[1,3]$ $D^3[1,4] + D^3[4,3]$

-4 $3+1$

$-4 < 4$

$D^4[1,3] = \underline{-4}$

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{vmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{vmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

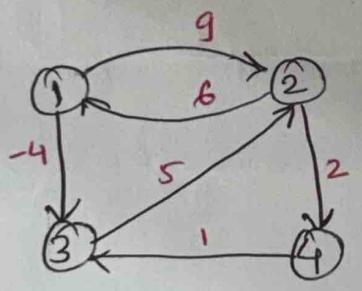
Algorithm

floydWarshal(cost)

Input – The cost matrix of given Graph.

Output – Matrix to for shortest path between any vertex to any vertex.

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```



Source vertex 1 ← 2, 3, 4

Source Vertex 2 ← 1, 3, 4

Source vertex 3 ← 1, 2, 4

Source vertex 4 ← 1, 2, 3

*Matrix chain multiplication :-

A_1	A_2	A_3	A_4
5×4	4×6	6×2	2×7

A × B = C

[ ] [ ] [ ]

5×4 4×3 5×3

= 15 entries

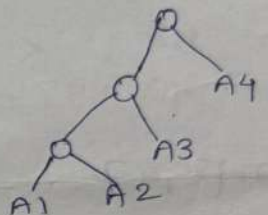
total multiplication

= 5×3×4 = 60.

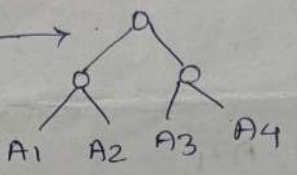
$T(n) = \frac{2n\,C_n}{n+1}$

* How parenthesis should be given?

- $((A_1 \cdot A_2) \cdot A_3) \cdot A_4$ ⟶



- $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$ ⟶



- $A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))$

A_1 A_2 A_3 A_4

↓ ↓ ↓ ↓

m[1,1] m[2,2] m[3,3] m[4,4]

Nothing to multiply. So cost is:

0 0 0 0

m[1,2]

$A_1 \cdot A_2$

5×4 4×6

Total cost is → 5×4×6 = 120

m[2,3]

$A_2 \cdot A_3$

4×6 6×2

Total cost = 4×6×2

= 48

m[3,4]

$A_3 \cdot A_4$

6×2 2×7

Total cost = 6×2×7

= 84

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Here only 2-2 matrices are selected.

Now select 3 matrix.

$m[1,3]$

$A1 \cdot A2 \cdot A3$

There are 2 possibilities of multiplication:-

$A1 \cdot (A2 \cdot A3)$ $\quad$ $(A1 \cdot A2) \cdot A3$

5x4 4x6 6x2 $\quad$ 5x4 4x6 6x2

Now, find the cost

$m[1,1] + m[2,3] + 5x4x2$ $\quad$ $m[1,2] + m[3,3] + 5x6x2$

$0 + 48 + 40$ $\quad$ $120 + 60$

$88 \quad < \quad 180$

$\therefore m[1,3] = 88$

$m[2,4]$

~~m[2~~ $A2 \cdot A3 \cdot A4$.

There are 2 possibilities of multiplication.

$A2 \cdot (A3 \cdot A4)$ $\quad$ $(A2 \cdot A3) \cdot A4$

4x6 6x2 2x7 $\quad$ 4x6 6x2 2x7

$m[2,2] + m[3,4] + 4x6x7$ $\quad$ $m[2,3] + m[4,4] + 4x2x7$

$0 + 84 + 168 \quad > \quad 48 + 0 + 56$

$\therefore m[2,4] = 104$

$m[1,4]$

$A1 \cdot A2 \cdot A3 \cdot A4$

5x4 4x6 6x2 2x7

$m[1,4] = \min \{ m[1,1] + m[2,4] + 5x4x7, \quad = 0 + 104 + 140 =$

$m[1,2] + m[3,4] + 5x6x7, \quad = 120 + 84 + 210 =$

$m[1,3] + m[4,4] + 5x2x7 \} \quad = 88 + 0 + 70 = 158$

Giving Small value.

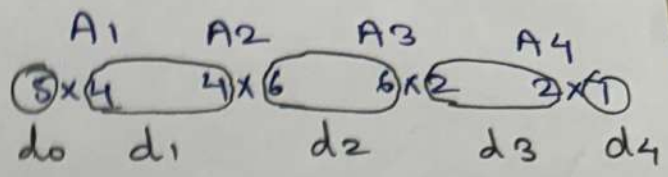
i.e $A1 \cdot (A2 \cdot A3 \cdot A4)$

$(A1 \cdot A2) \cdot (A3 \cdot A4)$

$(A1 \cdot A2 \cdot A3) \cdot A4$

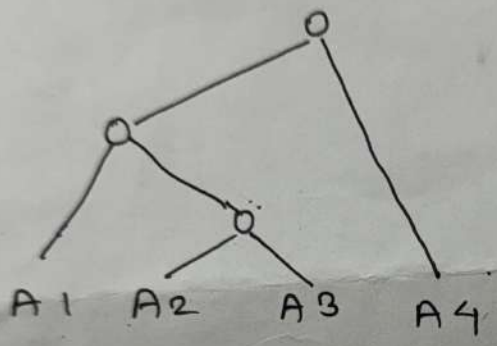
i.e. $m[1,4] = 158$

$$[i,j] = \min \{ m[i,K] + m[K+1, j] + d_{i-1} * d_K * d_j \}$$



$(A1 \cdot A2 \cdot A3) \cdot A4$

$((Ai) \cdot (A2 \cdot A3)) \cdot A4$



* 0/1 knapsack Problem *

weights = {3, 4, 6, 5}
Values (Profits) = {2, 3, 1, 4}

$w = 8$
$n = 4$

$\sum w_i x_i \leq W$

$x = \{1, 0, 0, 1\}$
$x = \{0, 0, 0, 1\}$
$x = \{0, 0, 1, 1\}$

P_i	w_i	i \ w	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	0	2 ←	2	2	2	2	2
	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 ←

value (2,4)
max (3+0, 2)
3

max (3+0, 2)
3

value (2,7)
max (3+2, 2)
5

value (2,8)
max (3+2, 2)
5

value (3,5)
max (4+0, 3)
4

value (3,6)
max (4+0, 3)
4

value (3,8)
max (4+2, 5)
6

$$m[i,w] = \max\left(m[i-1,w],\ m[i-1, w-w[i]] + P[i]\right)$$

$m(4,7) = \max(m(3,7),\ m[3, 7-6] + 1)$
$= \max[(5, 0+1)(5,1)] = \underline{5}$

$x_i = \{1, 0, 0, 1\}$

$6 - 4 = 2$

Q. knapsack of capacity W = 5.

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

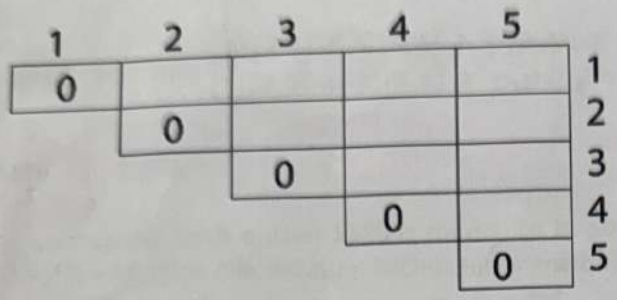
→

P. \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

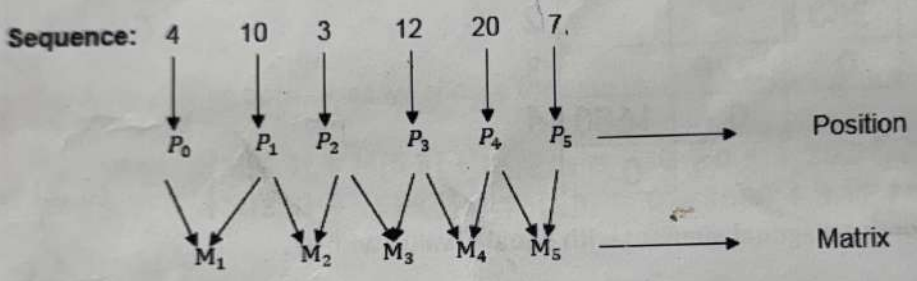
max (

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.



Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})

On the basis of sequence, we make a formula

For M_i ⟶ p [i] as column

p [i-1] as row

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. $m(1,2) = m_1 \times m_2$

$= 4 \times 10 \times 10 \times 3$

$= 4 \times 10 \times 3 = 120$

2. m (2, 3) = $m_2 \times m_3$
   = 10 x 3 x 3 x 12
   = 10 x 3 x 12 = 360

3. m (3, 4) = $m_3 \times m_4$
   = 3 x 12 x 12 x 20
   = 3 x 12 x 20 = 720

4. m (4,5) = $m_4 \times m_5$
   = 12 x 20 x 20 x 7
   = 12 x 20 x 7 = 1680

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

M [1, 3] = $M_1 M_2 M_3$

1. There are two cases by which we can solve this multiplication: ($M_1 \times M_2$) + M_3, M_1+ ($M_2 \times M_3$)
2. After solving both cases we choose the case in which minimum output is there.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and ($M_1 \times M_2$) + M_3 this combination is chosen for the output making.

3. $(M_1 \times M_2) \times (M_3 \times M_4)$

After solving these cases we choose the case in which minimum output is there

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and $(M_1 \times M_2) \times (M_3 \times M_4)$ combination is taken out in output making,

$M[2, 5] = M_2 M_3 M_4 M_5$

There are three cases by which we can solve this multiplication:

1. $(M_2 \times M_3 \times M_4) \times M_5$
2. $M_2 \times (M_3 \times M_4 \times M_5)$
3. $(M_2 \times M_3) \times (M_4 \times M_5)$

After solving these cases we choose the case in which minimum output is there

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and $M_2 \times (M_3 \times M_4 \times M_5)$combination is taken out in output making.

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 5 matrices:

$M[1, 5] = M_1 M_2 M_3 M_4 M_5$

There are five cases by which we can solve this multiplication:

1. $(M_1 \times M_2 \times M_3 \times M_4) \times M_5$
2. $M_1 \times (M_2 \times M_3 \times M_4 \times M_5)$
3. $(M_1 \times M_2 \times M_3) \times M_4 \times M_5$
4. $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

M [1, 5] = 1344

As comparing the output of different cases then '**1344**' is minimum output, so we insert 1344 in the table and $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Step 3: Computing Optimal Costs: let us assume that matrix A_i has dimension $p_{i-1} \times p_i$ for i=1, 2, 3....n. The input is a sequence $(p_0, p_1, \ldots\ldots p_n)$ where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

The algorithm first computes m [i, j] ← 0 for i=1, 2, 3.....n, the minimum costs for the chain of length 1.

$M[2, 4] = M_2 M_3 M_4$

1. There are two cases by which we can solve this multiplication: $(M_2 \times M_3)+M_4$, $M_2+(M_3 \times M_4)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[2, 4] = \min \begin{Bmatrix} M[2,3] + M[4,4] + p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10.3.20 = 1320 \end{Bmatrix}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and $M_2+(M_3 \times M_4)$ this combination is chosen for the output making.

$M[3, 5] = M_3 \ M_4 \ M_5$

1. There are two cases by which we can solve this multiplication: $(M_3 \times M_4) + M_5$, $M_3+ (M_4 \times M_5)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[3, 5] = \min \begin{Bmatrix} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{Bmatrix}$$

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and $(M_3 \times M_4) + M_5$ this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:

$M[1, 4] = M_1 \ M_2 M_3 M_4$

There are three cases by which we can solve this multiplication:

1. $(M_1 \times M_2 \times M_3) M_4$
2. $M_1 \times (M_2 \times M_3 \times M_4)$

$A_1 \times A_2 \times A_3$

2 3 3 4 4 2

d_1 d_2 d_3 d_4 d_5 d_6.

$(A_1 * A_2) \times A_3$

2 3 3 4 4 2

$A_1 \times (A_2 \times A_3)$

2 3 3 4 4 2

0

$3 \times 4 \times 2 = 24$

Total multiplications to be performed $2 \times 3 \times 4 = 24 + 0$

2 4 4 2

$24 + 0 + 2 \times 4 \times 2 = 40$

2 3 2

$0 + 24 + 2 \times 3 \times 2 = 36$.