

DYNAMIC PROGRAMMING & OTHER ALGORITHMS



DYNAMIC PROGRAMMING



Dynamic Programming is mainly an optimization over plain recursion.

The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

EX: FIBONACCI SEQUENCE

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

FIBONACCI IN CODE:

```
void fib ()  
  
{  
  
    fibresult[0] = 1;  
  
    fibresult[1] = 1,  
  
    for (int i = 2; i<n; i++)  
  
        fibresult[i] = fibresult[i-1] + fibresult[i-2];  
  
}
```

DYNAMIC PROGRAMMING VS GREEDY ALGORITHM

GREEDY ALGORITHM

In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.

DYNAMIC PROGRAMMING

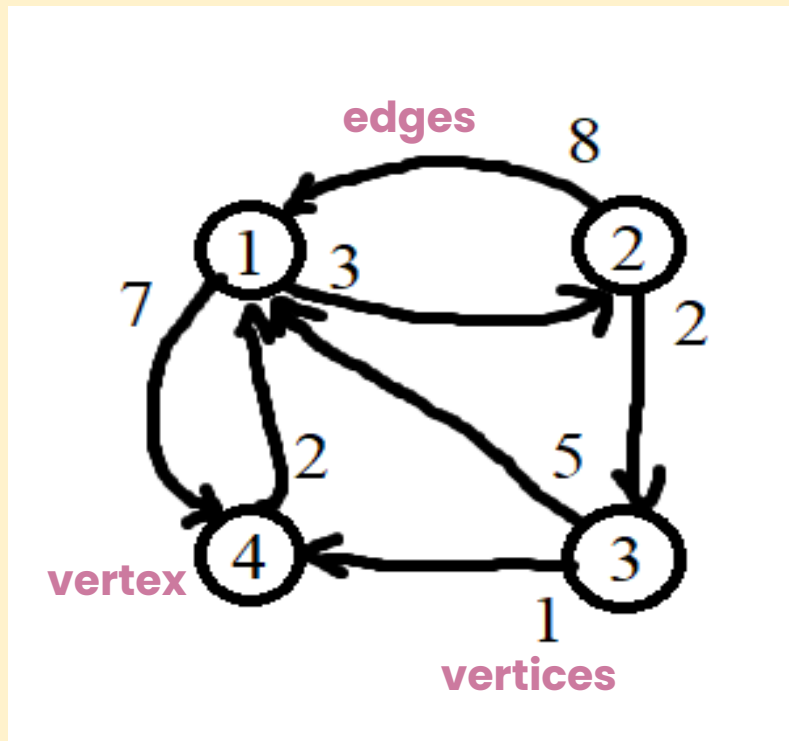
In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.

FLOYD-WARSHALL ALGORITHM

The Floyd-Warshall Algorithm is for solving all pairs shortest path problems. The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed Graph.

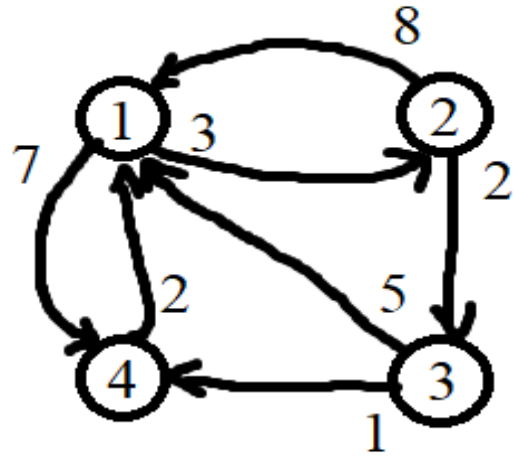
FLOYD-WARSHALL ALGORITHM

Given Graph:





HOW TO FIND THE SHORTEST PATH?



$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 4 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

Step 1: Create a matrix A^0 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i th vertex to the j th vertex. If there is no path from i th vertex to j th vertex, the cell is left as infinity.

Step 2: Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{matrix}$$

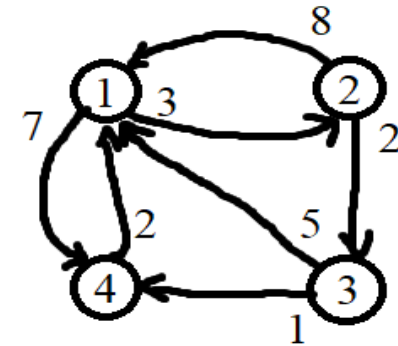
$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix} \end{matrix}$$

$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 4 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

$$(A[i][j] > A[i][k] + A[k][j])$$

$$\begin{array}{lcl} A^0[2,3] & A^0[2,1] + A^0[1,3] \\ 2 & < & 8 + \infty \end{array}$$

$$\begin{array}{lcl} A^0[2,4] & A^0[2,1] + A^0[1,4] \\ \infty & > & 8 + 7 \end{array}$$



$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{matrix}$$

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix} \end{matrix}$$

Step 3: For example: For $A1[2, 4]$, the direct distance from vertex 2 to 4 is 15 and the sum of the distance from vertex 2 to 1 and from vertex 1 to 4 is 15. Since $\infty > 15$, $A0[2, 4]$ is filled with ∞ .

Similarly, $A2$ is created using $A1$. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 8 & 0 & 2 & 15 \\ 3 & & 8 & 0 & \\ 4 & & 5 & & 0 \end{array}$$

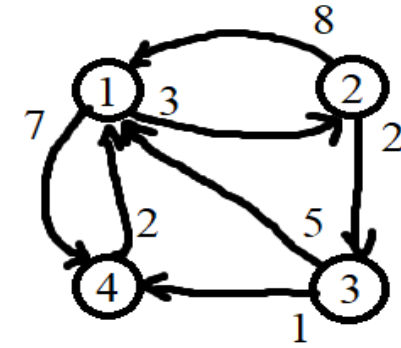
$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix} \end{matrix}$$

$$(A[i][j] > A[i][k] + A[k][j])$$

$$A^1[1,3] \quad A^1[1,2] + A^1[2,3] \\ \infty < 3 + 2$$

$$A^1[1,4] \quad A^1[1,2] + A^1[2,4] \\ 7 < 3 + 15$$



$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & 0 & \\ & 5 & & 0 \end{bmatrix} \end{matrix}$$

$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$

Step 4: Similarly, A3 and A4 is also created.

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & 5 & \\ 2 & & 0 & 2 & \\ 3 & 5 & 8 & 0 & 1 \\ 4 & & & 7 & 0 \end{array}$$

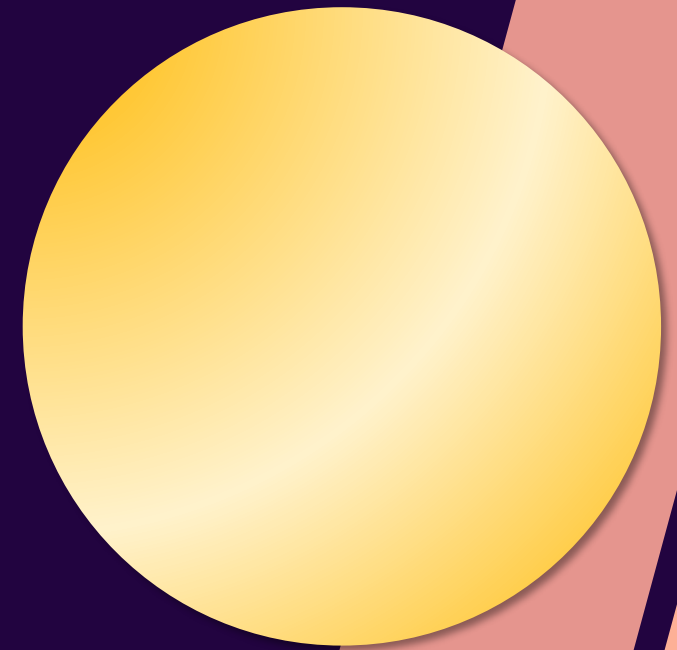
$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 6 \\ 2 & & 0 & & 3 \\ 3 & & & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 3 & 6 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 8 & \infty & 0 \end{array}$$

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$



$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 7 & 0 & 2 & 3 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 5 & 6 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 3 & 6 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$



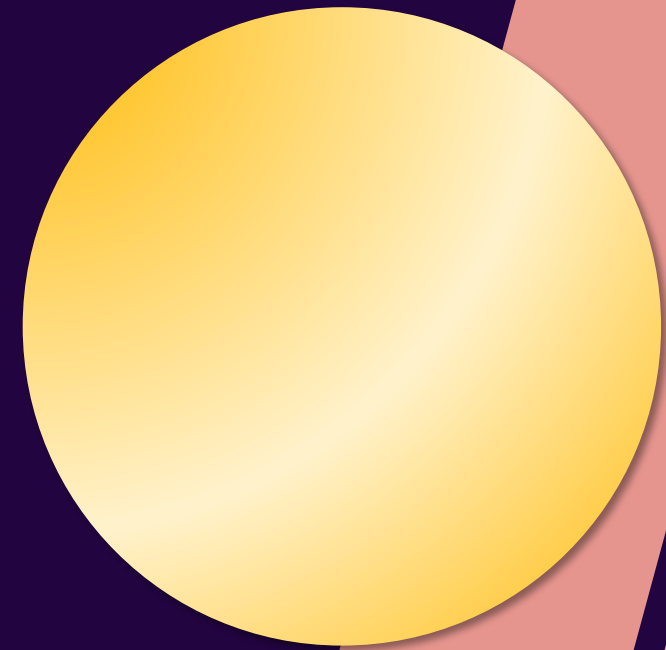
**Step 5: A4 GIVES THE
SHORTEST PATH BETWEEN
EACH PAIR OF VERTICES.**

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$

$$A^2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$

$$A^3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$

$$A^4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$



SOURCE CODE:

```
for(i=1; i<=n; i++)  
{  
    for (j=1; j<=n; j++)  
    {  
        A[i,j] =min(A[i,j] , A[i,k] + A[k,j]);  
    }  
}
```

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.

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.

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

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.

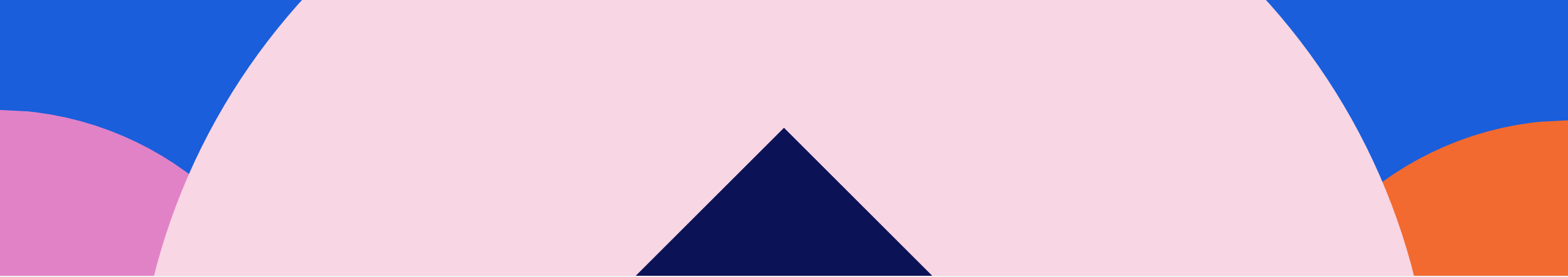
1. Create a table of dimension $n+1*m+1$ where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

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

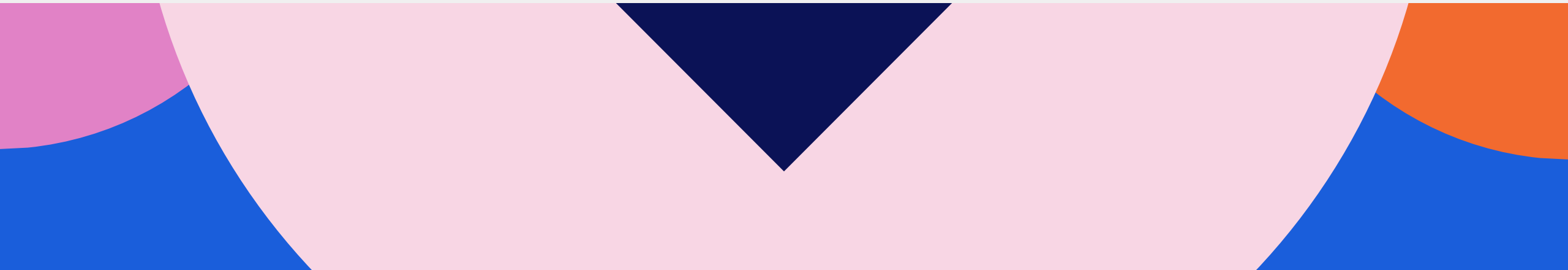
Initialise a table

A decorative graphic centered on a solid blue background. It consists of two concentric circles. The outer circle is divided horizontally: the top half is pink and the bottom half is orange. The inner circle is a solid light pink color. Overlaid on the center of these circles is a horizontal, rounded rectangular box with a dark pink background and a thin white border. Inside this box, the text "2. Fill each cell of the table using the following logic." is written in white, bold, sans-serif font.

**2. Fill each cell of the table
using the following logic.**



3. If the character corresponding 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

5. Step 2 is repeated until the table is filled.

		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

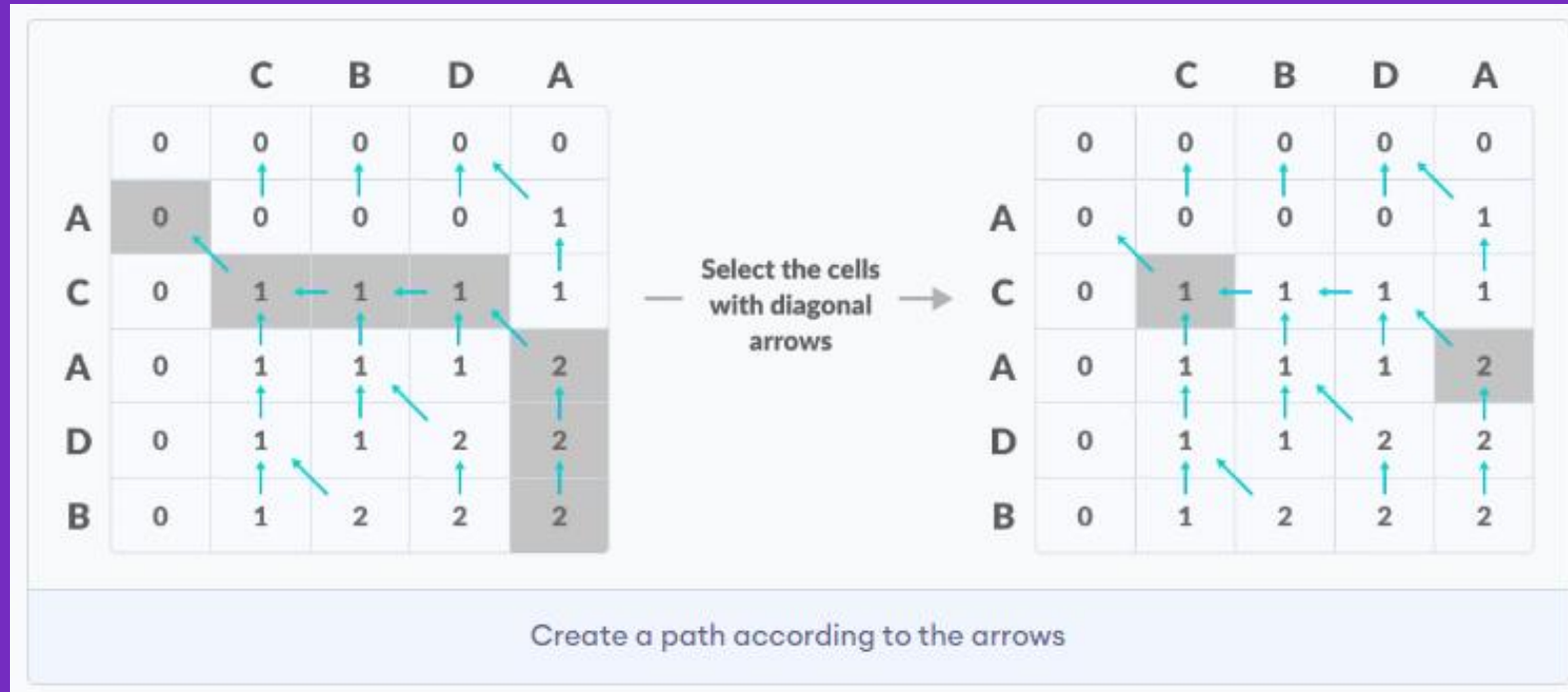
Fill all the values

6. The value in the last row and the last column is the length of the longest common subsequence.

		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

The bottom right corner is the length of the LCS

7. 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.



Thus, the longest
common
subsequence is CA.

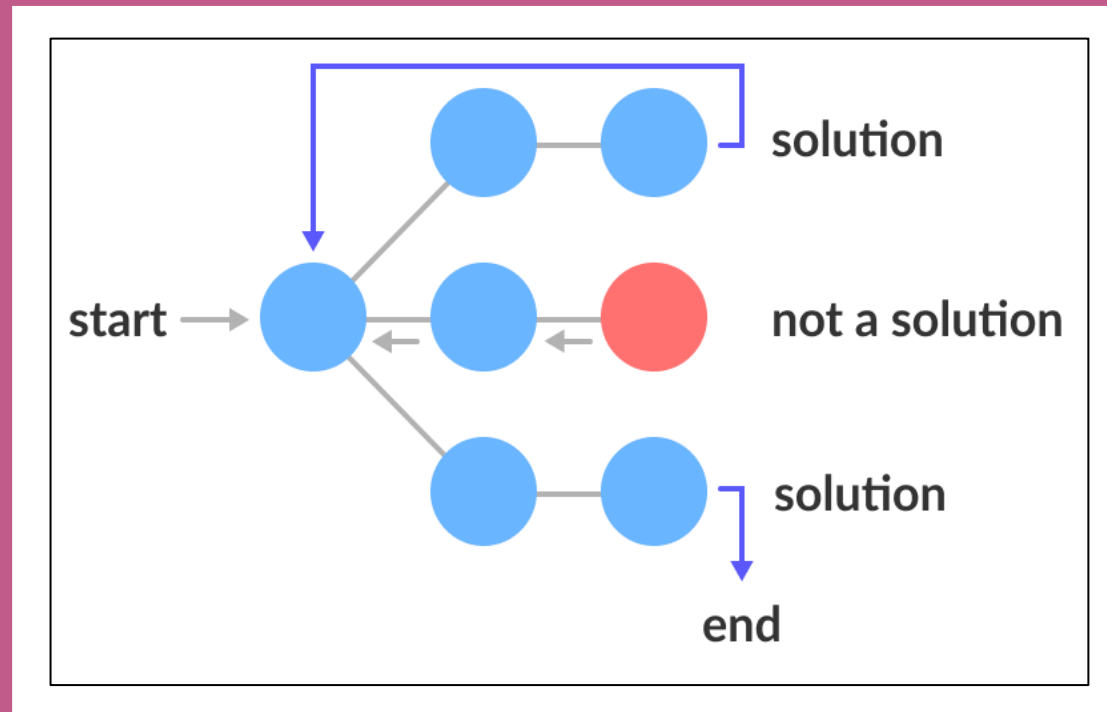
C	A
---	---

LCS

BACKTRACKING ALGORITHM

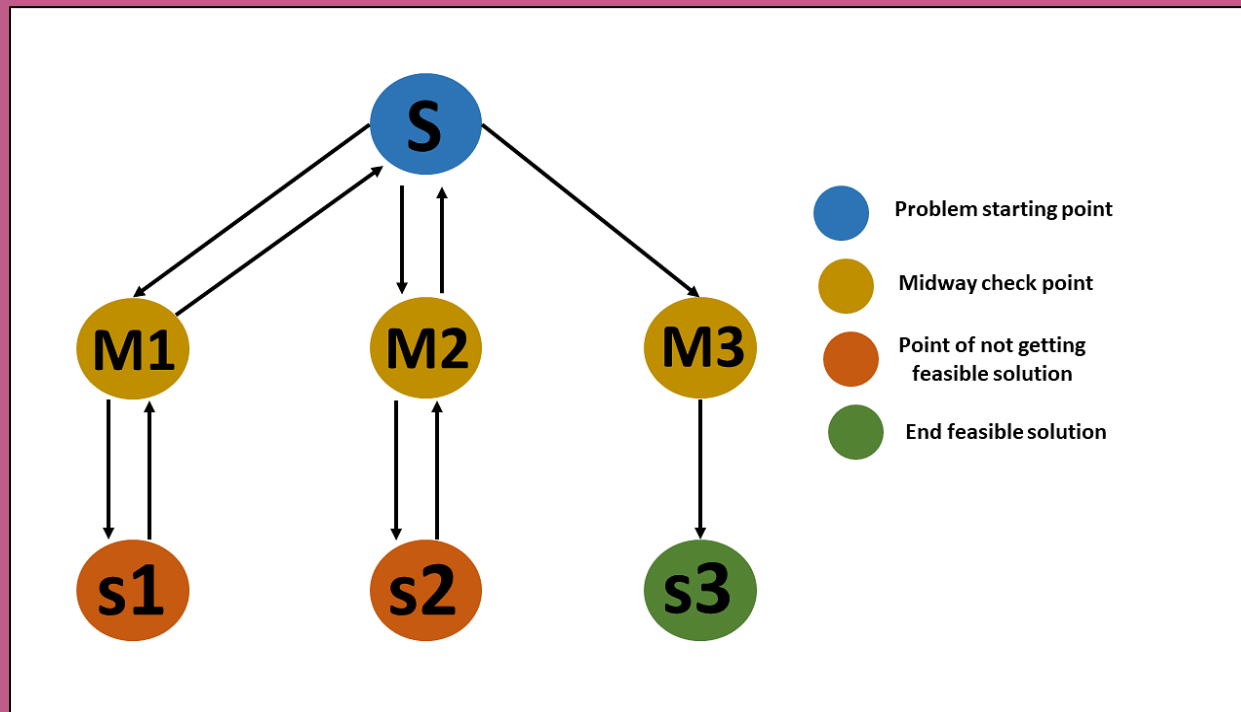
- A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- A Brute Force Algorithm is the straightforward approach to a problem. i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

BACKTRACKING ALGORITHM



BACKTRACKING ALGORITHM

Example:



1

CONSTRAINTS

2

APPLICATIONS OF BACKTRACKING

3

SUM OF SUBSET PROBLEM

4

GRAPH COLORING

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint**– It is a rule in which how each element in a tuple is related.
- **Explicit constraint** – It is a rule that restrict each element to be chosen from the given set.

1

CONSTRAINTS

2

**APPLICATIONS OF
BACKTRACKING**

3

**SUM OF SUBSET
PROBLEM**

4

GRAPH COLORING

Applications of Backtracking
N-queen problem – The N-Queens problem is commonly used to teach the programming technique of backtrack search. N Queen as another example problem that can be solved using backtracking.

1

CONSTRAINTS

2

APPLICATIONS OF BACKTRACKING

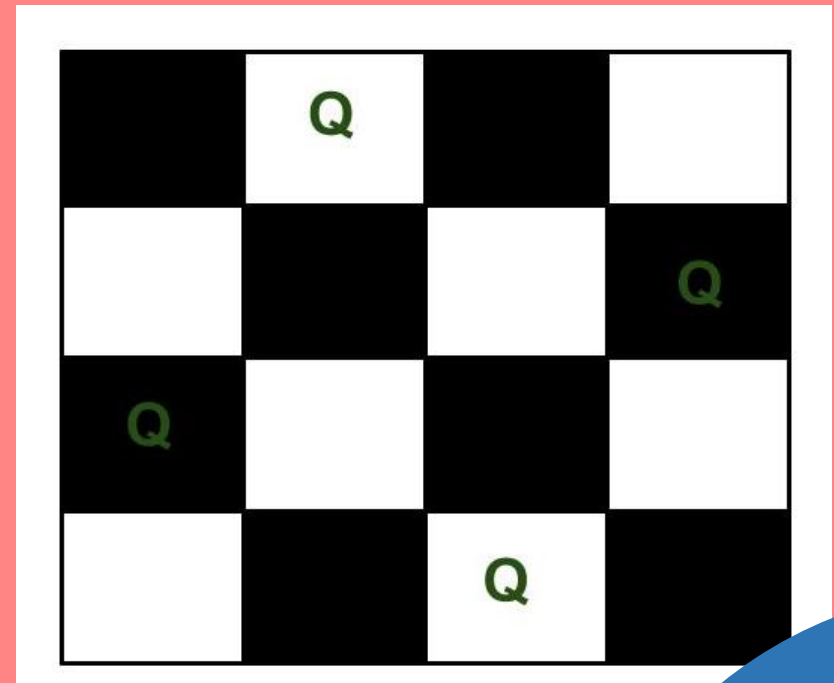
3

SUM OF SUBSET PROBLEM

4

GRAPH COLORING

The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other.



1

CONSTRAINTS

2

**APPLICATIONS OF
BACKTRACKING**

3

**SUM OF SUBSET
PROBLEM**

4

GRAPH COLORING

The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.

{ 0, 1, 0, 0 }

{ 0, 0, 0, 1 }

{ 1, 0, 0, 0 }

{ 0, 0, 1, 0 }

1

CONSTRAINTS

2

APPLICATIONS OF BACKTRACKING

3

SUM OF SUBSET PROBLEM

4

GRAPH COLORING

Subset Sum Problem Solution using Backtracking Algorithm

The main idea is to add the number to the stack and track the sum of stack values.

Given a set of elements and a sum value. We need to find all possible subsets of the elements with a sum equal to the sum value.

Example:

- Set: {10, 7, 5, 18, 12, 20, 15}
- Sum: 30
- Output: [{10, 5, 15}, {10, 20}, {7, 5, 18}, {18, 12}]

1

CONSTRAINTS

2

APPLICATIONS OF BACKTRACKING

3

SUM OF SUBSET PROBLEM

4

GRAPH COLORING

A graph coloring is an assignment of labels traditionally called "colors" to each vertex.

Example:

In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color. After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored.

1

CONSTRAINTS

2

**APPLICATIONS OF
BACKTRACKING**

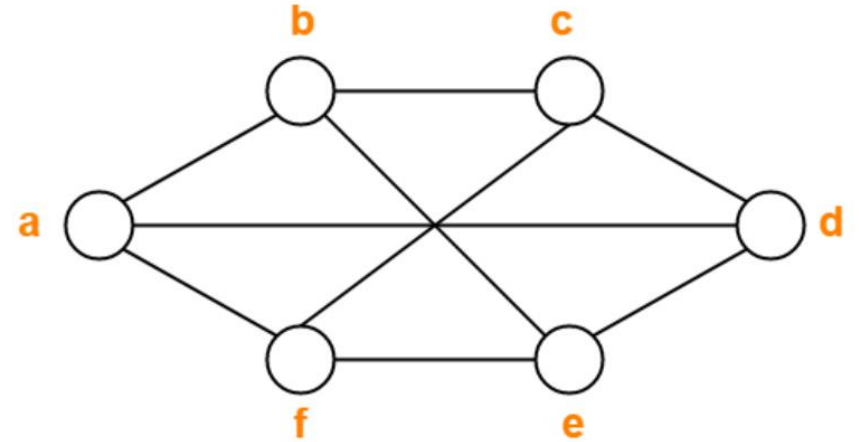
3

**SUM OF SUBSET
PROBLEM**

4

GRAPH COLORING

Find chromatic number of the following graph-



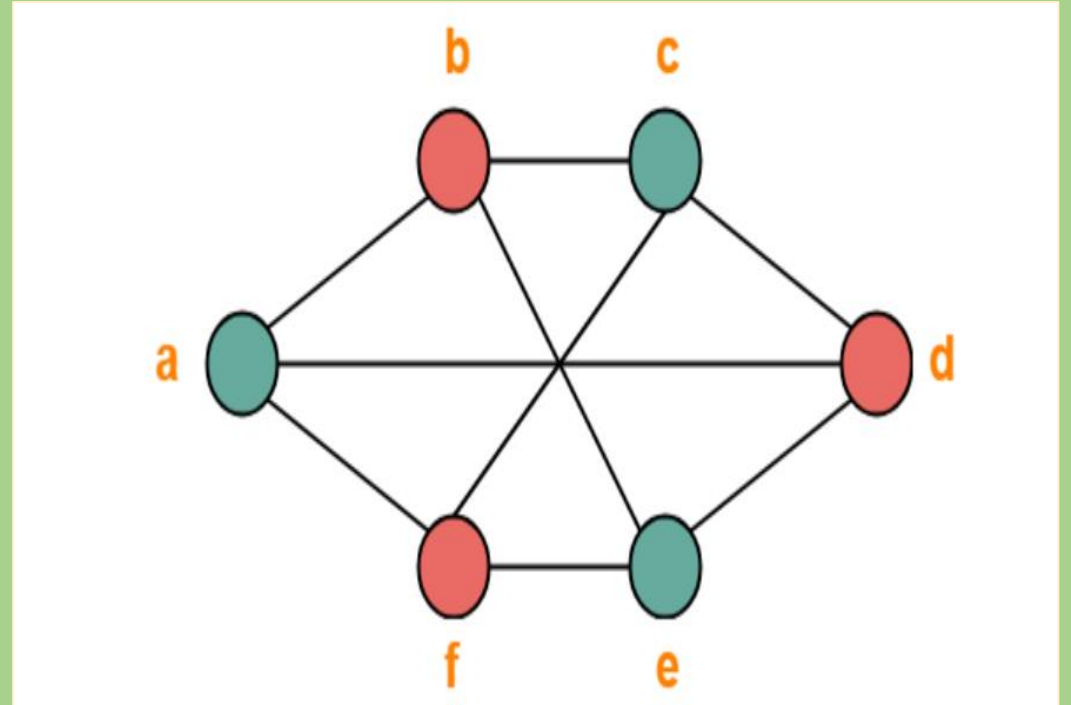
Vertex	a	b	c	d	e	f
Color	c1	c2	c1	c2	c1	c2

1 CONSTRAINTS

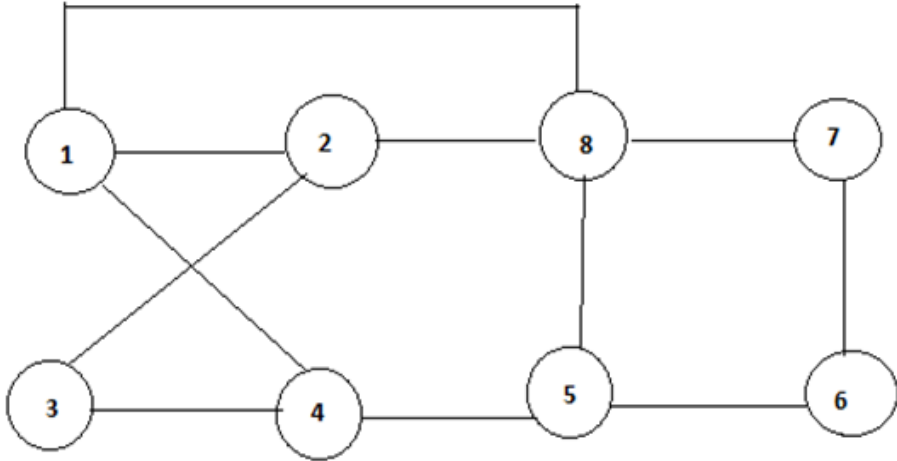
2 APPLICATIONS OF
BACKTRACKING

3 SUM OF SUBSET
PROBLEM

4 GRAPH COLORING



HAMILTON CYCLE



hamiltonian path is = 1 -- 2 -- 3 -- 4 -- 5 -- 6 -- 7 -- 8 -- 1

Hamilton Path is a path that goes through every Vertex of a graph exactly once.

A Hamilton Circuit is a Hamilton Path that begins and ends at the same vertex.

The background features a white canvas with several dark blue geometric shapes. There are four large, rounded rectangular blocks positioned at the top, bottom, left, and right. Interspersed between these larger blocks are smaller, solid dark blue squares. A central horizontal dark blue rectangle contains the text.

RABIN KARP ALGORITHM

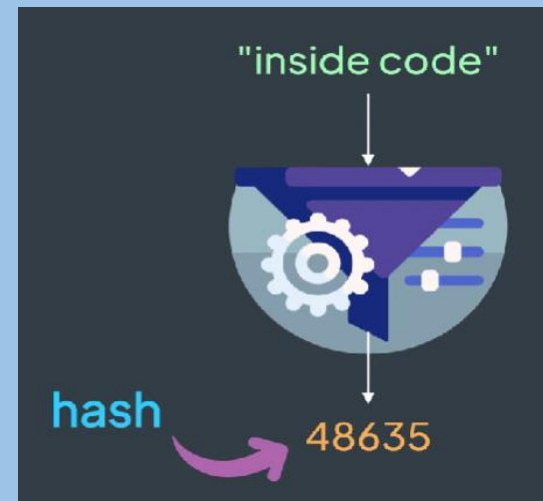
Remember the ff definitions:

Hashing – The process of converting a key to a fixed-size value

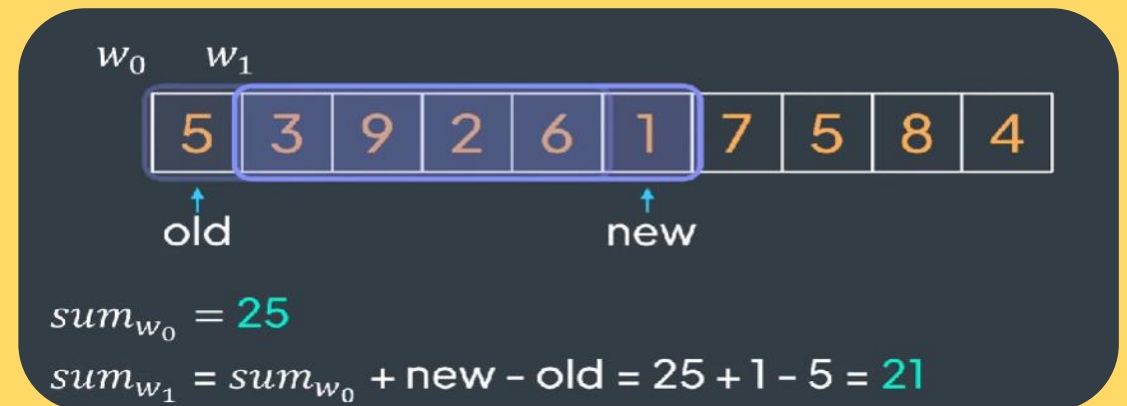
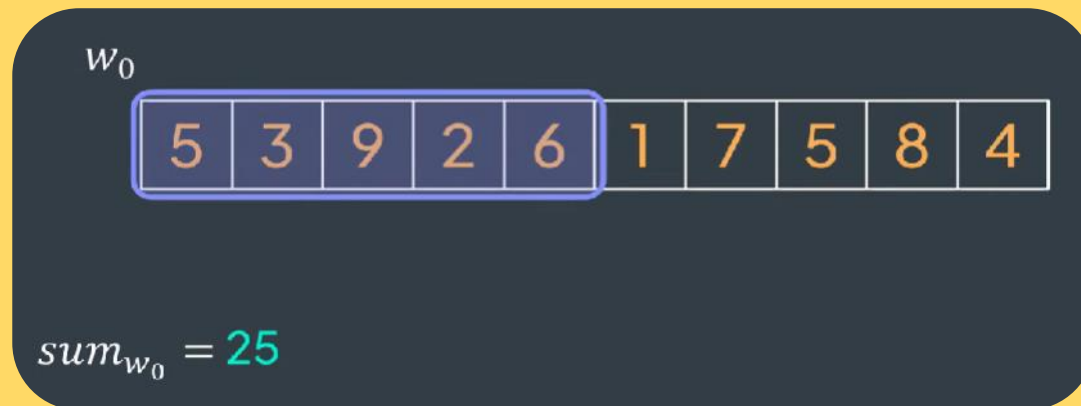
Example:

How does hashing work:

- It uses functions or algorithms to map object data to a representative integer value.
- A hash can then be used to narrow down searches when locating these items on that object data map.



A problem-solving technique that works by maintaining a window, a contagious part of the data. Its disadvantage is that if we want to calculate something about the next window, we don't need to do it from scratch, we just take the value of the actual window and update in in some way, which is faster.



SLIDING WINDOW

STRING-MATCHING

is the process of searching for the locations of one or multiple strings (also called patterns) inside a larger string.

```
s = "abcabfdeabacccabebf"  
p = "cab"
```

What is Rabin-Karp Algorithm?

Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the text using a hash function. (

A hash function is a tool to map a larger input value to a smaller output value. This output value is called the hash value.

Hashing the process of converting a key to a fixed-size value)

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.

Example:

1. Let the text be:

A B C C D D A E F G

Text

And the string to be searched in the above text be:

C D D

Pattern

How Rabin-Karp Algorithm Works?

2. 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).

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	6	7	8	9	10
Text Weights									

3. 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 $\{A, B, C, \dots, J\}$. So, $d = 10$. You can assume any suitable value for d .

4. Let us calculate the hash value of the pattern.

```
hash value for pattern(p) =  $\sum(v * d_{m-1}) \bmod 13$   
=  $((3 * 10^2) + (4 * 10^1) + (4 * 10^0)) \bmod 13$   
=  $344 \bmod 13$   
= 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) =  $\sum(v * d_{n-1}) \bmod 13$   
=  $((1 * 10^2) + (2 * 10^1) + (3 * 10^0)) \bmod 13$   
=  $123 \bmod 13$   
= 6
```

6. Compare the hash value of the pattern with the hash value of the text. If they match then, character-matching is performed.

7. We calculate the hash value of the next window by subtracting the first term and adding the next term as shown below.

$$\begin{aligned} t &= ((1 * 10^2) + ((2 * 10^1) + (3 * 10^0)) * 10 + (3 * 10^0)) \bmod 13 \\ &= 233 \bmod 13 \\ &= 12 \end{aligned}$$

$$\begin{aligned} t &= ((d * (t - v[\text{character to be removed}] * h) + v[\text{character to be added}]) \bmod 13 \\ &= ((10 * (6 - 1 * 9) + 3) \bmod 13 \\ &= 12 \end{aligned}$$

Where, $h = d^{m-1} = 10^{3-1} = 100$.

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

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



BRUTE FORCE:

Given a string s of size n and a pattern p of size m characters of s with p

We have "string s " of n characters, and "string p " of m characters.

Two equal 'string' must have the same number of characters, m in this case because we are searching for p , So, the idea is to compare all substring of s of m characters with p .



We have "string s" of n characters, and "string p" of m characters.

Two equal 'string' must have the same number of characters, m in this case because we are searching for p,
So, the idea is to compare all substring of s of m characters with p.

```
s = "abcabfdeabaccabebf"  
p = "cab"
```

Example. Values of 's' and 'p', we are finding the value 'cab' in 'abcabfdeabaccabebf'.

```
s = "abcabfdeabaccabebf"  
p = "cab"
```

abc = cab? No

```
s = "abcabfdeabaccabebf"  
p = "cab"
```

bca = cab? No

```
s = "abcabfdeabaccabebf"  
p = "cab"
```

cab = cab? Yes

