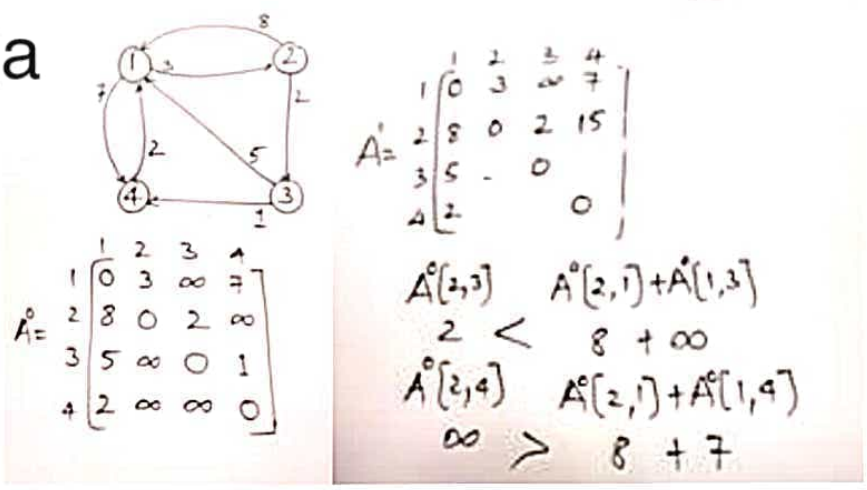
Ex:

1a



1
2
3
4
8
3
2
7
2
5
1

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

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

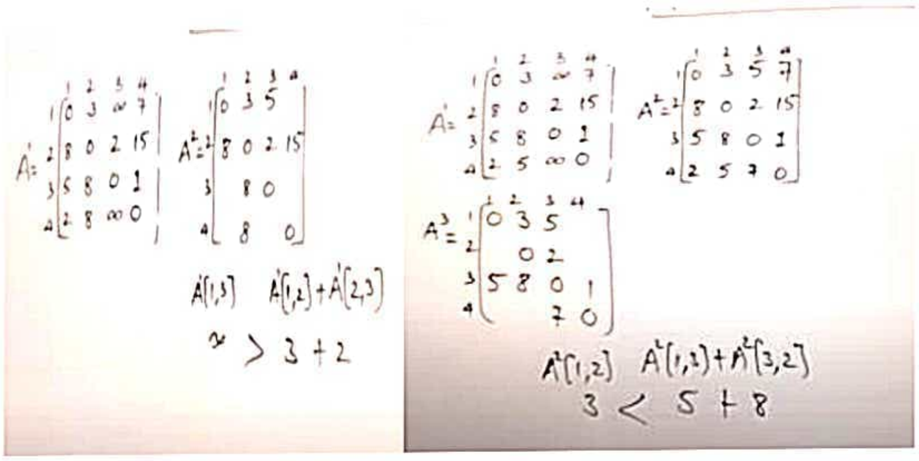
$A^0[2,3] \quad A^0[2,1]+A^0[1,3]$

$2 < 8 + \infty$

$A^0[2,4] \quad A^0[2,1]+A^0[1,4]$

$\infty > 8 + 7$

[redacted]



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

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

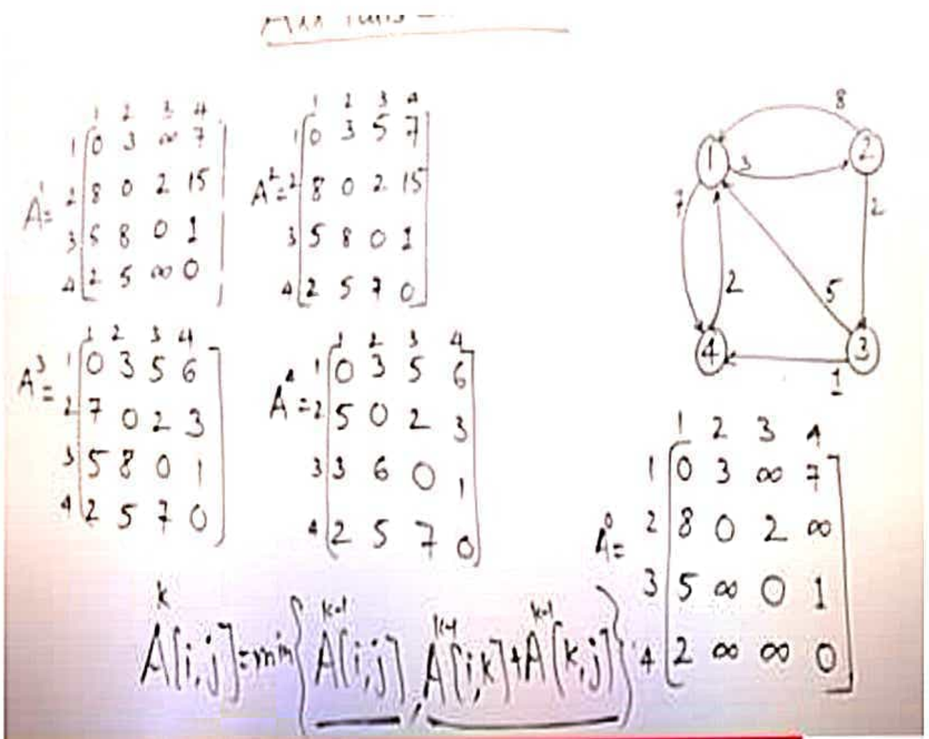
$\infty > 3 + 2$

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

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

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

$3 < 5 + 8$



1
2
3
4
8
3
2
7
2
5
1

All Pairs –

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

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

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

$$A^k[i,j] = \min\left\{ \underline{A^{k-1}[i,j]},\ \underline{A^{k-1}[i,k] + A^{k-1}[k,j]} \right\}$$

1b

In the 0/1 Knapsack Problem, merging and purging rules are heuristics or techniques used to optimize the selection of items to be included in or excluded from the knapsack in order to maximize the total value while staying within the knapsack's weight capacity.

1. Merging Rule:

   Merging rule is a technique that helps combine similar items or item groups into a single item. This simplifies the problem and reduces the number of items to be considered. Merging rule is generally applied when there are items with identical weights and values. By merging them into one item, the problem size is reduced, making it easier to solve.
   Example of Merging Rule:
   Let's say you have the following items with identical weights and values:
   - Item A: Weight = 3, Value = $10
   - Item B: Weight = 3, Value = $10
   - Item C: Weight = 5, Value = $15

   Instead of considering A and B as separate items, you can merge them into a single item with a weight of 6 (3 + 3) and a value of $20 (10 + 10). This simplifies the problem, making it easier to find the optimal solution.
2. Purging Rule:

   Purging rule is a technique used to eliminate items that are dominated by other items. An item is considered dominated if there is another item with both equal or smaller weight and a greater or equal value. By removing dominated items, you can reduce the problem size and improve the efficiency of the solution.
   Example of Purging Rule:
   Consider the following items:
   - Item X: Weight = 2, Value = $15
   - Item Y: Weight = 3, Value = $20
   - Item Z: Weight = 2, Value = $10

   In this case, Item Z is dominated by Item X because Item X has the same weight (2) but a greater value ($15 > $10). Therefore, you can eliminate Item Z from consideration as it will not be part of the optimal solution.

By applying merging and purging rules in the 0/1 Knapsack Problem, you can simplify the problem and potentially improve the efficiency of the algorithm used to find the optimal solution. However, it's important to note that these rules are not always applicable and may depend on the specific characteristics of the problem instance.

2a **All pairs shortest path:**

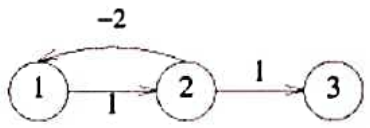
It aims to figure out the shortest path from each vertex v to every other u. Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u's row and v's column should be the weight of the shortest path from u to v.
Three approaches for improvement:

Algorithm	**Cost**
Matrix Multiplication	O (V^3 logV)
Floyd-Warshall	O (V^3)

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. (Johnson's Algorithm for sparse graphs uses adjacency lists.) The input is a n x n matrix W representing the edge weights of an n-vertex directed graph G = (V, E). That is, W = (w_{ij}), where

$$w_{ij} = \begin{cases} 0 & \text{if} \quad i = j, \\ w(i,j) & \text{if} \quad i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if} \quad i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

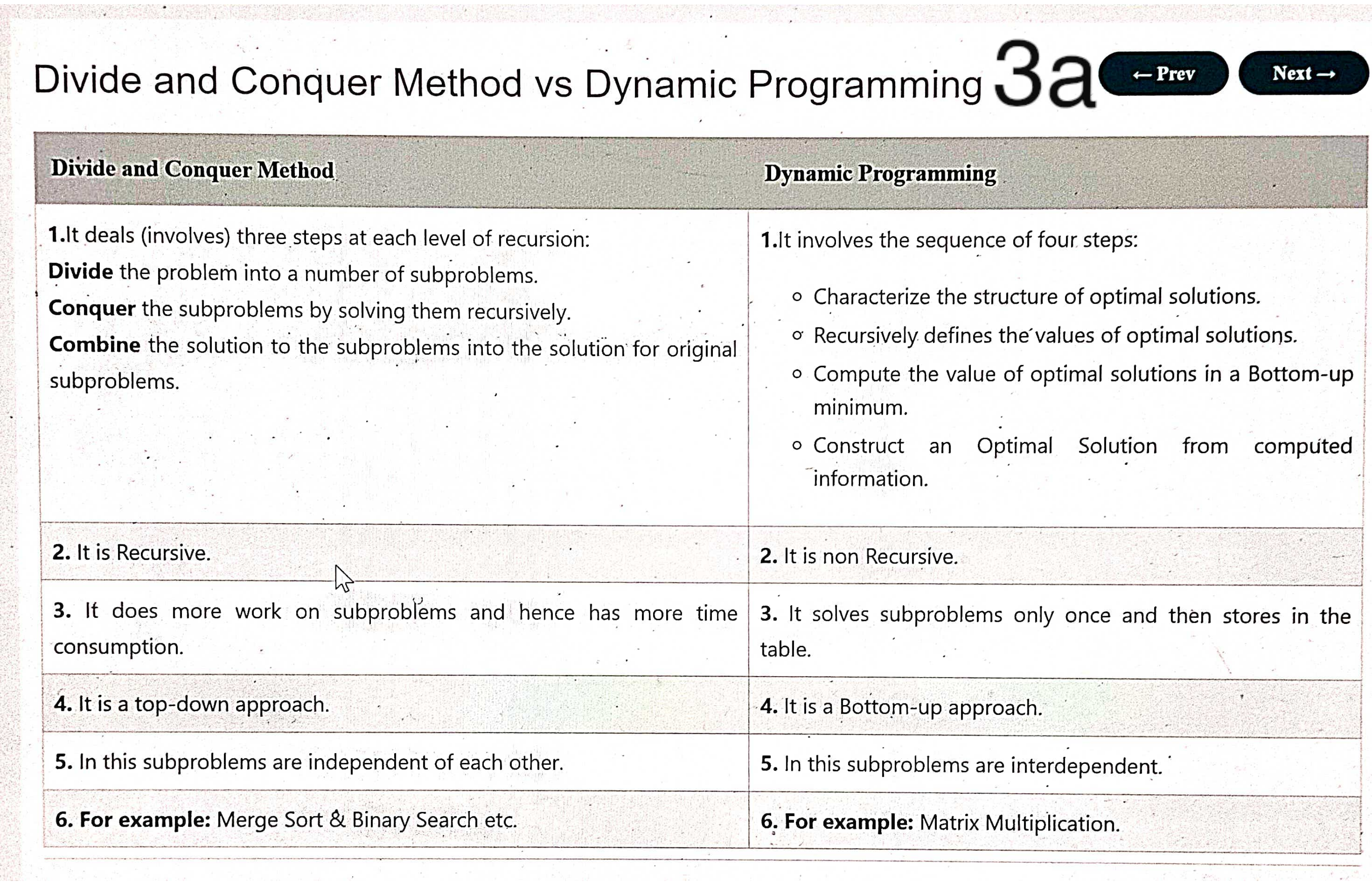
Ex:



$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

Algorithm:

```
Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// $n$ vertices; $A[i,j]$ is the cost of a shortest path from vertex
// $i$ to vertex $j$. $cost[i,i] = 0.0$, for $1 \le i \le n$.
{
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $A[i,j] := cost[i,j]$; // Copy cost into $A$.
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $A[i,j] := \min(A[i,j], A[i,k] + A[k,j])$;
}
```



Divide and Conquer Method vs Dynamic Programming 3a

Divide and Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: ○ Characterize the structure of optimal solutions. ○ Recursively defines the values of optimal solutions. ○ Compute the value of optimal solutions in a Bottom-up minimum. ○ Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

3b Reliability Design

Q: Design a 3 stage system with Device types D1, D2, D3. The costs are 30, 15, 20 \$ resp. The cost of the system is to be no more than 105 \$. The reliability of each device type is 0.9, 0.8 and 0.5 resp.

$c_1 = 30 \quad c_2 = 15 \quad c_3 = 20 \qquad C = 105$

$r_1 = 0.9 \quad r_2 = 0.8 \quad r_3 = 0.5$

$u_i = \left(C + c_i - \sum_{j=1}^{n} c_j\right) / c_i$

$U_1 = \left(105 + 30 - (30+15+20)\right) / 30 = 2$

$U_2 = \left(105 + 15 - (30+15+20)\right) / \cancel{15} = 3$

$U_3 = \left(105 + 20 - (30+15+20)\right) / 20 = 3$

```
Algorithm BellmanFord($v, cost, dist, n$)                    4a
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for $i := 1$ to $n$ do // Initialize $dist$.
        $dist[i] := cost[v, i]$;
    for $k := 2$ to $n - 1$ do
        for each $u$ such that $u \neq v$ and $u$ has
                at least one incoming edge do
            for each $\langle i, u \rangle$ in the graph do
                if $dist[u] > dist[i] + cost[i, u]$ then
                    $dist[u] := dist[i] + cost[i, u]$;
}
```

Analysis :

- A graph can be represented using adjacency matrix or adjacency list.
- If adjacency matrix is used
  - Each iteration of the for loop takes *$O(n^2)$* time. Here n is the number of vertices in the graph.
  - Overall complexity is *$O(n^3)$*
- If adjacency list is used
  - Each iteration of the for loop takes *$O(e)$* time. Here is the number of edges in the graph.
  - Overall complexity is *$O(ne)$*

String Editing: 5a, 8a

Given two strings: Source String and Destination String.

- The Source String will never modify.
- It is required to convert the destination string into source string using three possible operations: INSERT/DELETE/UPDATE.
- The cost of operation: INSERT=1, DELETE=1, UPDATE=2.
- The objective is to perform string editing in minimum cost.
- The process of translation/mapping is carried out phase wise.

String Editing: Cost of conversion

- Let the source string be Y [y1... yj]
- Let the destination string be X [x1...xi]
- Cost of transformation: depends on character at Xi and Yj

Insert/Delete/Update operation

- (i) If no string in Y and character present in X, to make Y=X: Perform?
- (ii) If string is in Y and no string is present in X, to make Y=X: Perform?
- (iii) If string contents present in Y and X, then cost of transformation depends upon

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases}$$

$$\text{where } cost'(i,j) = \min \{ \begin{array}{l} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i,y_j), \\ cost(i,j-1) + I(y_j) \} \end{array}$$

Alternatively, you can implement the edit distance algorithm yourself using a ***dynamic programming approach***, and can be implemented as follows:

Example:

```
def edit_distance(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])

    return dp[m][n]
str1 = "kitten"
str2 = "sitting"
print(edit_distance(str1, str2))
```

Output:

```
This code returns the edit distance between str1 and str2 as 3.
```

The ***dynamic programming implementation*** can be described as follows:

The edit distance can be calculated using a ***dynamic programming approach***. It involves creating a ***2D table dp*** with ***m + 1 rows*** and ***n + 1 columns***, where ***m*** and ***n*** are the lengths of the two strings ***str1*** and ***str2***. After that, the table is filled in using a series of rules, based on whether the characters in ***str1*** and ***str2*** match or not. The final result is the edit distance between ***str1*** and ***str2*** which is stored in ***dp[m][n]***.

The idea behind the dynamic programming approach is to use a ***2D table dp*** to store the intermediate results and avoid redundant calculations. The ***dp[i][j]*** field stores the edit distance between the first ***i*** characters of string ***str1*** and the first ***j*** characters of string ***str2***.

The first ***for loop*** initializes the first row and first column of the ***dp*** table with the number of operations required to transform an empty string into ***str1*** or ***str2***. It is just the same as the relevant string's length.

The edit distance between two strings ***str1*** and ***str2*** can be calculated using the second for loop. If the current characters in both strings match, the corresponding ***dp*** entry is equal to the previous ***dp entry (dp[i - 1][j - 1])***. It means no operations are required to transform ***str1*** into ***str2***.

If the characters do not match, the corresponding ***dp entry*** is equal to the minimum of the three possible operations: ***deletion, insertion***, and ***substitution***. The ***dp[i][j - 1]*** entry corresponds to ***deletion***, the ***dp[i - 1][j]*** entry corresponds to ***insertion***, and the ***dp[i - 1][j - 1]*** entry corresponds to ***substitution***. The ***1*** in the formula ***1 + min(...)*** represents the ***cost*** of the current operation. After the for loop, the ***dp[m][n]*** entry will contain the final result, which is the minimum number of operations required to transform ***str1*** into ***str2***.

The edit distance between two strings is determined using a similar methodology by ***Levenshtein***. The distance functions from the ***Pylev package***. The difference is that the ***pylev library*** is implemented in C, which makes it much faster than a pure Python implementation.

Consider the two strings "kitten" and "sitting". The dynamic programming table dp would look like this:

```
  0 1 2 3 4 5 6 7
--------------------------
0|0 1 2 3 4 5 6 7
1|1 1 2 3 4 5 5 6
2|2 2 2 3 4 4 5 6
3|3 3 3 3 4 5 5 6
4|4 4 4 4 4 5 6 7
5|5 5 5 5 5 5 6 7
```

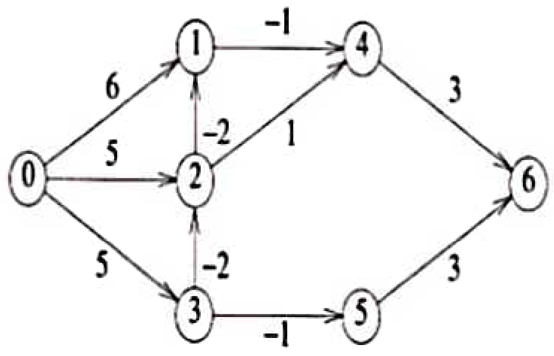
Greedy vs. Dynamic Programming :

6a

- Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no a priori litmus test by which one can tell if the Greedy method will lead to an optimal solution. By contrast, there is a litmus test for Dynamic Programming, called The Principle of Optimality

Example : Following graph has 7 vertices

6b



	$dist^k[7]$						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

The table has arrays $dist^k$, for k = 1, 2, …. 6
These values are computing by following the above equation.

For instance, distk[1] = 0 for all k since 1 is the source node. Also, dist1[2]=6, dist1[3] = 5 and dist1[4]=5, since there are edges from 1 to these nodes. Distance to remaining vertices is infinity since there are no edges to theses from 1.

General Method:

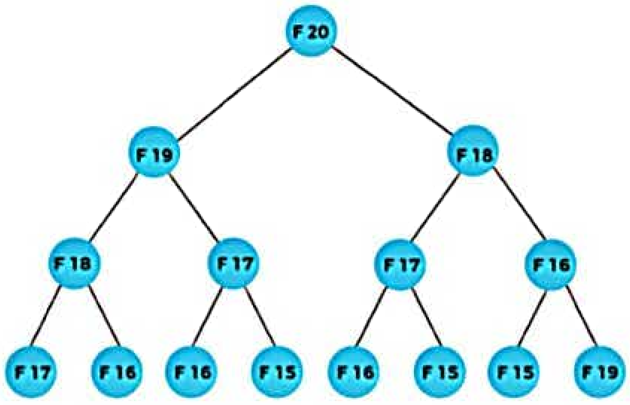
7a Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The sub problems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.
The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each sub problem just once, and then storing their solutions to avoid repetitive computations.

Ex:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,.....
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

$$F(n) = F(n-1) + F(n-2),$$

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.
The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



- This can be done using Dynamic Programming methodology
    - If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k - 1$ edges, then $dist^k[u] = dist^{k-1}[u]$
    - If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $<j, u>$. The path from v to j has $k - 1$ edges, and its length is $dist^{k-1}[j]$. all vertices i such that the edges $<i, u>$ is in the graph are candidates for j. since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the correct value for j.

This will result in the following recurrence

$$dist^k[u] = \min\left\{\begin{matrix} dist^{k-1}[u] \\ \min\limits_i\{dist^{k-1}[i] + cost[i,u]\} \end{matrix}\right\}$$

This recurrence can be used to compute distk from distk-1, for k = 2, 3, …, n-1

7b.

Single Source Shortest Path Problem

- **Problem** : Given a directed graph G = (V, E), a weighting function *cost* for the edges of G, and a source vertex v_0, The problem is to determine the shortest paths from v_0 to all the remaining vertices of G.
- This problem can be solved using greedy method (Dijkstra's algorithm) if the edge weights are positive. This algorithm does not necessarily give the correct answers on the graphs which have negative weights.

- Dynamic Programming suggests a technique to find solution to this problem.
  - Here negative edge weights are permitted, but graph should not have cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges.
  - In an n-vertex graph with no cycles of negative length, the shortest path between any two perties has at most (n – 1) edges on it.

- Solution:
  - Let $dist^l[u]$ be the length of a shortest path from the source vertex *v* to vertex *u* under the constraint that the shortest path contains at most *l* edges.
  - Then $dist^1[u] = cost[v,u],\ 1 \leq u \leq n$
  - When there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n - 1$ edges. Hence $dist^{n-1}[u]$ is the length of an unrestricted shortest path from *v* to *u*.
  - Our goal is then to compute $dist^{n-1}[u]$ for all *u*.

8b

- Dynamic Programming is an algorithm design method and it can be used when the solution to a problem can be viewed as the result of sequence of decisions.
- Dynamic programming (DP) solves every subsubproblem exactly once, and is therefore more efficient in those cases where the subsubproblems are not indepndent.
- An optimal sequence of decisions is obtained by making use of **Principle Of Optimality**
- The <u>principle of optimality</u> states that an optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- Principle of Optimality can also be referred as <u>Optimal substructure</u>. A problem has optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its sub-problems. In other words, we can solve larger problems given the solutions to its smaller sub problems.
- The idea of dynamic programming is thus quite simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up when sub instances are solved.
- Dynamic programming is a method for solving optimization problems. The idea: Compute the solutions to the subsub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.