Dynamic Programming

Dynamic Programming

- Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have **overlapping subproblems and optimal substructure** property.

- If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

- Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

- Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in mergesort and quick sort.

Dynamic Programming Solutions

- There are two ways of doing this.

- Top-Down

- Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as Memoization.

- Bottom-Up

- Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as Dynamic Programming.

Dynamic Programming: Memoization

Memoization is the top-down approach to solving a problem with dynamic programming.

It’s called memoization because we will create a memo, or a “note to self”, for the values returned from solving each problem.

Then, when we encounter the same problem again, we simply check the memo, and, rather than solving the problem a second (or third or fourth) time, we retrieve the solution from our memo.

https://jarednielsen.com/dynamic-programming-memoization-tabulation/

Top-Down Fibonacci

Here's our Fibonacci sequence, memoized:

```
const fibDown = (n, memo=[]) => {
if (n == 0 || n == 1) {
        return n;
    }
    if (memo[n] == undefined) {
        memo[n] = fibDown(n - 1, memo) + fibDown(n - 2, memo);
    }
    return memo[n];
}
```

Dynamic Programming: Tabulation

- With bottom-up, or tabulation, we start with the smallest problems and use the returned values to calculate larger values.
- We can think of it as entering values in a table, or spreadsheet, and then applying a formula to those values.

Bottom-Up Fibonacci

Here's our Fibonacci sequence, *tabulated*:

```
const fibottomUp = n => {
if (n === 0) {
            return 0;
     }
     let x = 0;
     let y = 1;
     for (let i = 2; i < n; i++) {
           let tmp = x + y;
           x = y;
           y = tmp;
     }
     return x + y;
}
```

Majority of the Dynamic Programming problems can be categorized into two types:

- 1. Optimization problems.
- 2. Combinatorial problems.

- The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized.

- Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Parameters of Comparison	Brute Force	Dynamic Programming
Methodology	It finds all the possible outcomes of a given problem.	It also finds all the possible outcomes, but avoids recomputation by storing solutions of the subproblems.
Time Complexity	It could be anything, sometimes even in exponential terms.	It helps us optimize the brute force approach, sometimes exponential terms are improved to polynomial terms(ex. factorial program).
Iterations	The number of iterations is more	The number of iterations is less(in terms of n)
Efficiency	It is less efficient	It is more efficient
Storage	Generally requires no extra space for storing results of sub-problems.	It requires extra space for storing the solutions to the sub-problems, which could be further used when required.

Divide and Conquer	Dynamic Programming
Subproblems are solved independently, and finally all solutions are collected to arrive at the final answers.	Dynamic programming considers a large number of decision sequences and all the overlapping substances.
The divide and conquer strategy is slower than the dynamic programming approach.	The dynamic programming strategy is slower than the divide and conquer approach.
Maximize time for execution.	Reduce the amount of time spent on execution by consuming less time.
Recursive techniques are used in Divide and Conquer.	Non-Recursive techniques are used in Dynamic programming.
A top-down approach is used in Divide and Conquer.	In a dynamic programming solution, the bottom-up approach is used.
The problems that are part of a Divide and Conquer strategy are independent of each other.	A dynamic programming subproblem is dependent upon other sub-problems.
One of the best examples of this strategy is a binary search.	One of the best examples of this strategy is the longest common subsequence.
No results are stored when completing sub-problems.	The solutions to sub-problems are saved in the table.
Repeating tasks.	There is no repeating task.
At a specified point, the split input splits big problems into smaller ones.	Every point in the split input is processed.
The divide and conquer strategy is simple to solve.	A dynamic programming solution can sometimes be complicated and challenging to solve.
Not more than one decision sequence is generated.	More than one decision sequence is generated.

Dynamic programming	Divide and Conquer
In dynamic programming, many decision sequences are generated, and all the overlapping sub instances are considered.	In this technique, the problem is divided into small subproblems. These subproblems are solved independently. Finally, all the solutions of subproblems are combined together to get the final solution to the given problem.
In dynamic programming, duplications in solutions are avoided totally.	In this method, duplications in sub solutions are neglected, i.e., duplicate sub solutions can be obtained.
Dynamic programming is more efficient than Divide and conquer technique.	Divide and conquer strategy is less efficient than the dynamic programming because we have to rework the solutions.
It is the non-recursive approach.	It is a recursive approach.
It uses the bottom-up approach of problem- solving.	It uses the top-down approach of problem- solving.

Dynamic Programming Schema

- Every Dynamic Programming problem has a schema to be followed:
- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Floyd-Warshall Algorithm

- Floyd Warshall is a dynamic programming-based algorithm so it breaks the problem into smaller subproblems and then solves each unitary subproblem after that it combines the result to solve the bigger problem while storing the result of each subproblem for future reference.
- It is also known as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, and WF algorithm.
- Floyd Warshall is all pair shortest path algorithm used to find the shortest path or shortest distance between every pair of vertices in the given graph.
- The algorithm is applicable on both directed and undirected graphs, but it fails with the graph having negative cycles; a negative cycle means the sum of the edges in a cycle is negative.

Algorithm

- Step 1 – Construct an adjacency matrix A with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

- Step 2 – Derive another adjacency matrix A1 from A keeping the first row and first column of the original adjacency matrix intact in A1. And for the remaining values, say A1[i,j], if A[i,j]>A[i,k]+A[k,j] then replace A1[i,j] with A[i,k]+A[k,j]. Otherwise, do not change the values. Here, in this step, k = 1 (first vertex acting as pivot).

- Step 3 – Repeat Step 2 for all the vertices in the graph by changing the k value for every pivot vertex until the final matrix is achieved.

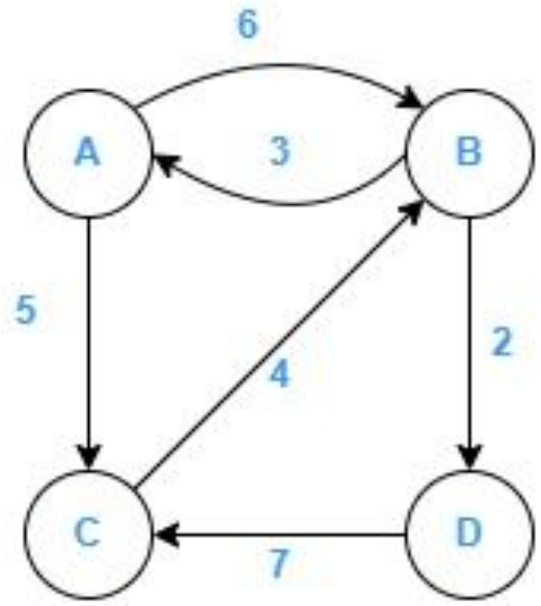
- Step 4 – The final adjacency matrix obtained is the final solution with all the shortest paths.

Psuedo Code

```
n = no of vertices

D = matrix of dimension n*n

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Dk[i, j] = min (Dk-1[i, j], Dk-1[i, k] + Dk-1[k, j])
return D
```



D0 =

	A	B	C	D
A	0	6	5	∞
B	3	0	∞	2
C	∞	4	0	∞
D	∞	∞	7	0

DA =

	A	B	C	D
A	0	6	5	∞
B	3	0	**8**	**2**
C	∞	**4**	0	∞
D	∞	**∞**	**7**	0

DB =

	A	B	C	D
A	0	6	**5**	**8**
B	3	0	8	2
C	**7**	4	0	**6**
D	∞	∞	7	0

DC =

	A	B	C	D
A	0	6	5	**8**
B	3	0	8	2
C	7	4	0	6
D	**14**	**11**	7	0

DD = :

	A	B	C	D
A	0	6	5	**8**
B	3	0	8	2
C	7	4	0	6
D	**14**	**11**	7	0

https://www.shiksha.com/online-courses/articles/about-floyd-warshall-algorithm
/

Complexity

Following are the complexities in the algorithm:

Time Complexity: There are three for loops in the pseudo-code of the algorithm, so the time complexity will be O (n^3).

Space Complexity: The space complexity of Floyd's Warshall algorithm is O (n^2).

Application

In networking devices

In Routing data packets

Calculate the inversion of the real matrix

Calculating transitive closure of directed graphs

To check whether an undirected graph is bipartite

To find the shortest path in a directed graph