Dynamic Programming

Dr. Priyadarshan Dhabe,

Ph.D (IIT Bombay)

Professor in Information Technology

What is Dynamic programming?

- It is **algorithm design technique** where the subproblems of the **same form** occurs many time.
- We **store the result** and use it to **avoid** re-computation of sub-problems **repeatedly** encountered in getting solution to **original problem**.
- It is mostly used for **optimization problems**, problems where multiple solutions are there and we have to opt for **best solution (define optimization problem, linear and non linear programming)**
- Storing the results of subproblems and using the tabulated/stored results of them is key feature of **dynamic programming.**

Dynamic programming-general strategy

- Dynamic programming, like the **divide-and-conquer** method, solves problems by **combining** the **solutions to subproblems**.
- "**Programming**" in this context refers to a **tabular** method.
- **divide-and-conquer** algorithms **partition** the problem into disjoint subproblems, solve the subproblems recursively, and then **combine** their solutions to solve the **original problem**.
- In contrast, dynamic programming applies when the subproblems **overlap**—that is, **when subproblems share subsubproblems.** In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem **just once** and then saves its answer in a **table**, thereby **avoiding** the work of recomputing the answer every time it solves each **subsubproblem**.

Dynamic programming-general strategy

- We typically 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

Fibonacci sequence algorithm-Recursive version

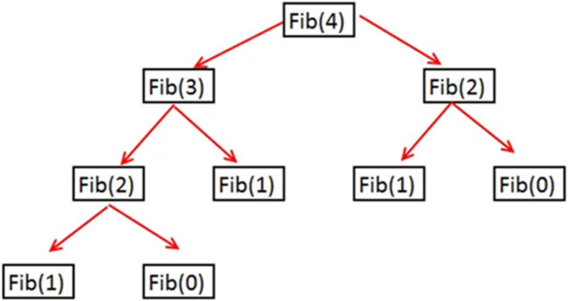
```
Int FibRec (int n)
{
    if (n==0)
        return 0;
    if(n==1)
        return 1;
    else
        f=FibRec(n-1)+FibRec(n-2);
            return f;
}
```

F={0,1,1,2,3,5,8,13,…….}

Given F(0)=0 and F(1)=1

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

Fibonacci sequence algorithm-Recursive version

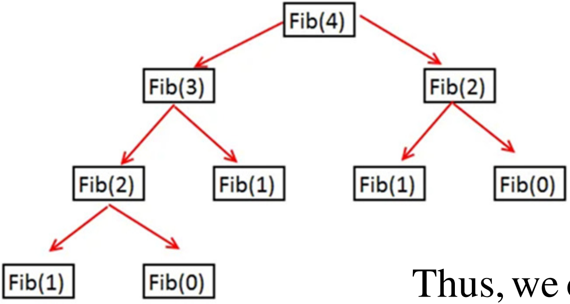


Fib(2) is called 2 times, thus recursive version is doing unnecessary work

$$T(n) = T(n-1) + T(n-2) + 1$$

Fibonacci sequence algorithm-Recursive version performance

For computing Fib(n) we need $(2^{n-1} + 1)$calls



T(n) = T(n-1) + T(n-2) + 1

Thus, we can write

$$T(n) = O(2^{n-1}) + O(2^{n-2}) + k$$

$$\Rightarrow O(2^n)$$

Fibonacci sequence algorithm- Dynamic prog.

```
Int FibDP(int n)
{
 int Fib[n+1]; //array of n+1
 Fib[0]=0; Fib[1]=1; //base
 for i=2 to n
    Fib[i]=Fib[i-1]+Fib[i-2];
 return Fib[n];
}
```

- **Fib[ ] is an array of size n+1**
- **At each iteration i ,previous 2 values (i-1) and (i-2) are added**
- **O(n) time and space complexity**

Recursive	Dynamic Programming
Time exponential O(2^n)	Time linear O(n)
Space O(2^n)	Space O(n)

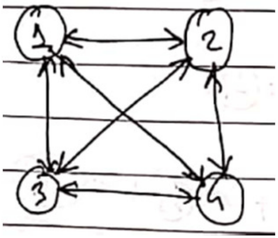
https://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/

Travelling salesman Problem- Dynamic prog.

- A **salesman** has to visit **set of n** cities. Each city has to be visited **only once** and has to return to the same starting city, such that the distance travelled should be **minimum**.

Travelling salesman Problem (TSP)- Dynamic prog.

- The dynamic programming approach is to solve the smaller subproblems first and use their stored solutions to get solutions for bigger problems, there by avoiding the repeated computations.



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

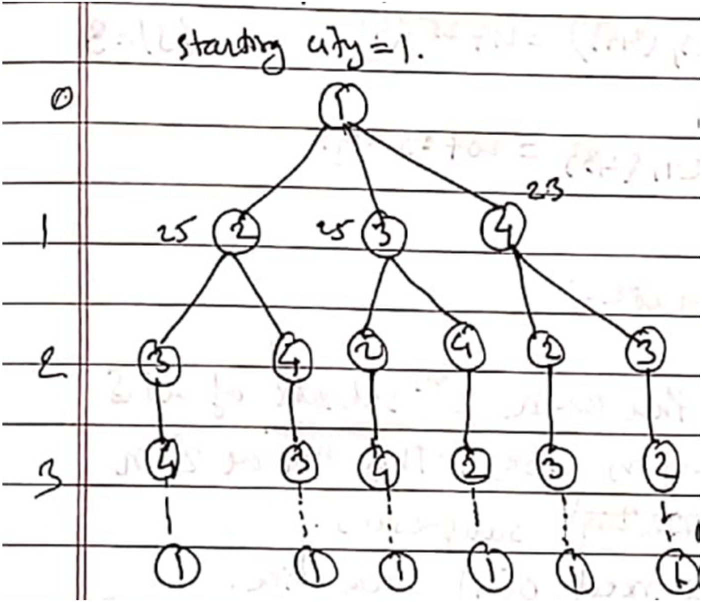
Cost adjacency matrix.

Matrix C

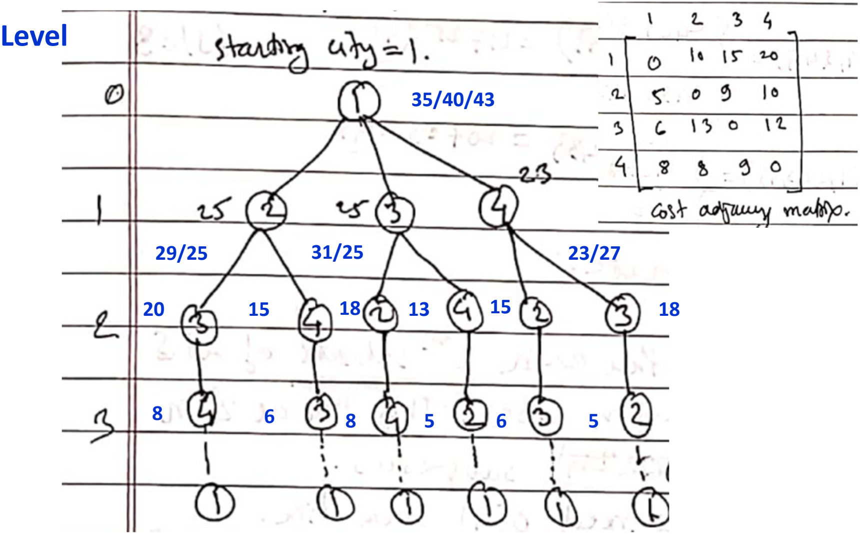
Travelling salesman Problem (TSP)- Dynamic prog.

- Way of working/understanding
  - Will solve by using brute-force
  - Derive the formula
  - Apply the formula

TSP- Using Dynamic prog.



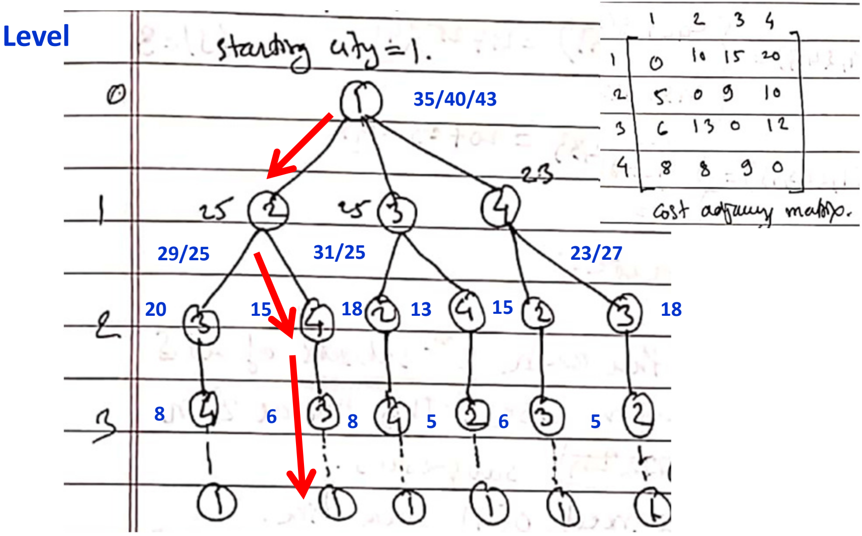
Understanding TSP- Using Dynamic prog.



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Need to be solved bottom up manner-brute force

Understanding TSP- Using Dynamic prog.

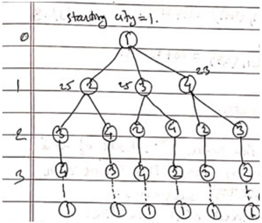


	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Need to be solved bottom up manner-brute force

Understanding TSP- Using Dynamic prog.

- Lets generate formula for the above problem and then we will generalize it to any starting node.



$$g(1,\{2,3,4\}) = \min\begin{cases} C_{1,2} + g(2,\{3,4\}) \\ C_{1,3} + g(3,\{2,4\}) \\ C_{1,4} + g(4,\{2,3\}) \end{cases}$$

g(1,{2,3,4})- cost of going from node 1 and visiting each of the node from set s={2,3,4} once and returning back to node 1.

$$g(2,\{3,4\}) = \min\begin{cases} C_{2,3} + g(3,\{4\}) \\ C_{2,4} + g(4,\{3\}) \end{cases}$$

$$g(3,\{2,4\}) = \min\begin{cases} C_{3,2} + g(2,\{4\}) \\ C_{3,4} + g(4,\{2\}) \end{cases}$$

$$g(4,\{2,3\}) = \min\begin{cases} C_{4,2} + g(2,\{3\}) \\ C_{4,3} + g(3,\{2\}) \end{cases}$$

$$g(3,\{4\}) = C_{3,4} + g(4,\phi)$$

$g(4,\phi)$ – cost of going from 4 through each node in empty set and returning to node 1, which is same as $\mathrm{C}_{4,1}$

$$g(4,\{3\}) = C_{4,3} + g(3,\phi)$$

Understanding TSP- Using Dynamic prog.

$$g(i,S) = \min_{j \in S} \{ c_{ij} + g(j, S - \{j\}) \}$$

where

$g(i,S)$ - is the shortest path starting from i and going through all the vertices in set S and terminate at node i.

https://www.youtube.com/watch?v=XaXsJJh-Q5Y

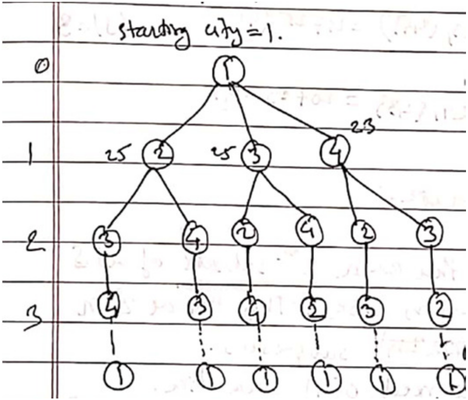
Understanding TSP- Using Dynamic prog.

Starting from level 3 compute the costs, there are links between each of the node 2,3,4 to the city 1.

$$g(2,\phi) = c_{21} = 5$$

$$g(3,\phi) = c_{31} = 6$$

$$g(4,\phi) = c_{41} = 8$$

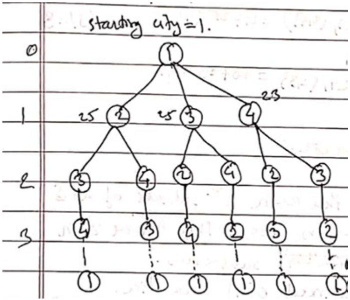


$$g(i,S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\}$$

$g(2,\phi)$ - means there are no intermediate nodes bet 2 and 1

Understanding TSP- Using Dynamic prog.

Computing costs at level 2



$$g(i,S)=\min_{j\in S}\{c_{ij}+g(j,S-\{j\})\}$$

$$g(2,\{3\})=c_{23}+g(3,\phi)=9+6=15$$

$$g(2,\{4\})=c_{24}+g(4,\phi)=10+8=18$$

$$g(3,\{2\})=c_{32}+g(2,\phi)=13+5=18$$

$$g(3,\{4\})=c_{34}+g(4,\phi)=12+8=20$$

$$g(4,\{3\})=c_{43}+g(3,\phi)=9+6=15$$

$$g(4,\{2\})=c_{42}+g(2,\phi)=8+5=13$$

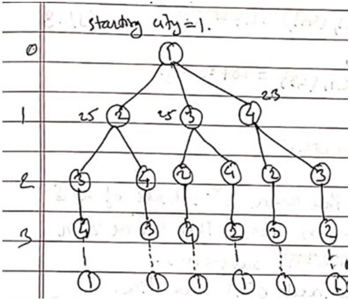
	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Cost adjacency matrix.

Understanding TSP- Using Dynamic prog.

Computing costs at level 1

$$g(i,S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\}$$



$$g(2,\{3,4\}) = \min\begin{cases} j=3 & c_{23} + g(3,\{4\}) = 9+20 = 29 \\ j=4 & c_{24} + g(4,\{3\}) = 10+15 = 25 \end{cases}$$

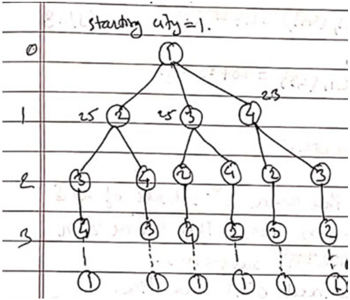
$$g(2,\{3,4\}) = 25$$

$$g(3,\{2,4\}) = \min\begin{cases} j=2 & c_{32} + g(2,\{4\}) = 13+18 = 31 \\ j=4 & c_{34} + g(4,\{2\}) = 12+13 = 25 \end{cases}$$

$$g(3,\{2,4\}) = 25$$

Understanding TSP- Using Dynamic prog.

Computing costs at level 1



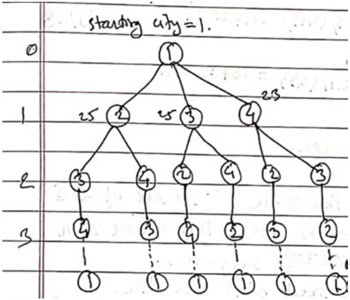
$$g(i,S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

$$g(4,\{2,3\}) = \min \begin{cases} j=2 & c_{42} + g(2,\{3\}) = 8 + 15 = 23 \\ j=3 & c_{43} + g(3,\{2\}) = 9 + 18 = 27 \end{cases}$$

$$g(4,\{2,3\}) = 23$$

Understanding TSP- Using Dynamic prog.

Computing costs at level 0



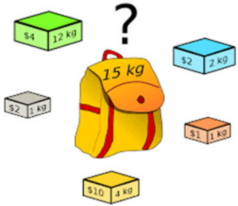
$$g(i,S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

$$g(1,\{2,3,4\}) = \min \begin{cases} j=2 \quad c_{12} + g(2,\{3,4\}) = 10 + 25 = 35 \\ j=3 \quad c_{13} + g(3,\{2,4\}) = 15 + 25 = 40 \\ j=4 \quad c_{14} + g(4,\{2,3\}) = 20 + 23 = 43 \end{cases}$$

$$g(1,\{2,3,4\}) = 35$$

Final answer

0/1 knapsack problem



- We have a knapsack with a capacity of weight W and a set of objects with their corresponding profits. We have to choose the set of objects with weights <=W and has to **maximize the profit.**
- 0/1 means we can either pick an object (1) or not pick(0) it. That is **partial object** is not allowed.
- An object can be **picked only once**.

0/1 knapsack problem

Object	ob1	ob2	ob3
weight	2	4	8
profit	20	25	60

Given knapsack capacity M=12, find the objects to be put in the sack to get **maximum profit** with weight<=12

For each object two possibilities are there 1 or 0. Thus, for n objects we have 2^n possibilities.

0/1 knapsack problem-brute force approach

Object	ob1	ob2	ob3
weight	2	4	8
profit	20	25	60

In this case $2^3=8$ possibilities are there.

objects	Weight	profit
000	0	0
001	8	60
010	4	25
011	12	85
100	2	20
101	10	80
110	6	45
111	14 >12 (not possible)	

0/1 knapsack problem- Dynamic Prog.

Weight W ->

M

Items considered-i

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

Increasing order

M[i, w]=profit

For i=0, no item is considered and thus, first row of M will be zero, since no profit from 0 objects

Weights={3,4,6,5}, Profit={2,3,1,4}, capacity W=8 and N=4

0/1 knapsack problem- Dynamic Prog.

i

W ->

M

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2									
3									
4									

Increasing order

Considering first object and ignore remaining objects

M[i, w]=profit

Weights={3,4,6,5}, Profit={2,3,1,4}, capacity W=8 and N=4

0/1 knapsack problem- Dynamic Prog.

i

W ->

M

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0								
4	0								

Increasing order

Considering 1 and 2 nd object

M[i, w]=profit

Weights={3,4,6,5}, Profit={2,3,1,4}, capacity W=8 and N=4

0/1 knapsack problem- Dynamic Prog.

i

W ->

M

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	**2**	**2**	**2**	**2**	**2**	**2**
2	0	0	0	2	**3**	**3**	**3**	**5**	**5**
3	0	0	0	2	3	**4**	**4**	**5**	**6**
4	0								

Increasing order

Considering 1,2 and 3rd object

M[i, w]=profit

Weights={3,4,6,5}, Profit={2,3,1,4}, capacity W=8 and N=4

0/1 knapsack problem- Dynamic Prog.

i

W ->

M

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	**2**	**2**	**2**	**2**	**2**	**2**
2	0	0	0	2	**3**	**3**	**3**	**5**	**5**
3	0	0	0	2	3	**4**	**4**	**5**	**6**
4	0	0	0	2	3	4	**4**	**5**	**6**

Increasing order

Final answer

Considering 1,2, 3 and 4th object

M[i, w]=profit

Weights={3,4,6,5}, Profit={2,3,1,4}, capacity W=8 and N=4

0/1 knapsack problem- Dynamic Prog.-Philosophy

- The optimal solution to the overall problem depends upon the optimal solution to its subproblems. This simple optimization reduces time complexities from exponential to polynomial.

0/1 knapsack problem- Dynamic Prog.

W ->

i

M

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	4	4	5	6
4	0	0	0	2	3	4	4	5	6

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

Where **p[i]-** is **profit** from **i th object**

$$M[i,w] = \max\left(M[i-1,w], M[i-1,w-w[i]] + p[i]\right)$$
$$M[2,4] = \max\left(M[1,4], M[1,4-4] + p[2]\right)$$
$$M[2,4] = \max\left(M[1,4], M[1,0] + 3\right)$$
$$M[2,4] = \max\left(2, 0 + 3\right)$$
$$M[2,4] = 3$$

$$M[i,w] = \max\left(M[i-1,w], M[i-1,w-w[i]] + p[i]\right)$$
$$M[2,7] = \max\left(M[1,7], M[1,7-4] + p[2]\right)$$
$$M[2,7] = \max\left(M[1,7], M[1,3] + 3\right)$$
$$M[2,7] = \max\left(2, 2 + 3\right)$$
$$M[2,7] = 5$$

0/1 knapsack problem- Dynamic Prog.

W ->

i

pi	wi
2	3
3	4
4	5
1	6

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	4	4	5	6
4	0	0	0	2	3	4	4	5	6

Deciding the objects in the sack

X={0,0,0,0}

X={0,0,1,0}

Profit from 3rd object is 4.
6-4=2 profit, check 2 in 2nd row

X={1,0,1,0}

https://www.youtube.com/watch?v=nLmhmB6NzcM

Time complexity of Dynamic programming is
O(n*w) (polynomial time), where
n- number of objects and W is the knapsack capacity. Using
Brute-force approach is O(2^n) (exponential time)

Solve 0/1 knapsack problem using Dynamic Prog.

Weights={2,3,4,5}, Profit={1,2,5,6}, capacity W=8 and N=4

TSP-Time complexity

For a pnroblem of size n cities, there can be 2^{n} subsets of set S.

Each subset contains max n cities. Thus, there are $2^{n}.n$ subproblems.

To solve each problem we need O(n) linear time, thus

$$\mathrm{T(n)} = O(2^{n}.n.n) = O(2^{n}.n^{2})$$