Indian Institute of Information Technology Design and Manufacturing,
Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe:
Dhanalakshmi S

Dynamic Programming

In earlier lectures we have discussed paradigms such as incremental design (e.g., insertion sort), divide and conquer (e.g., binary search, merge sort, quick sort) which are the most sought after paradigms to design algorithms for many classical problems. In this lecture, we shall discuss another paradigm, 'Dynamic Programming' which is a popular paradigm to design algorithms for many optimization problems. This method has close resemblance to divide and conquer. Divide and conquer is a problem strategy if subproblems are independent whereas dynamic programming is preferred to divide and conquer if subproblems are overlapping. Due to overlapping subproblems, subproblems are solved only once and stored in a table which will be referred when it is encountered again during the course of the algorithm. Table helps to avoid recomputation of a subproblem and retrieves the solution to subproblem in constant time. Further, the solution to a problem is computed using the solutions to subproblems and hence it is clear that dynamic programming adopts a bottom-up strategy unlike divide and conquer which follows a top-down strategy.

When developing a dynamic programming algorithm, we follow a sequence of four steps [1] :

1. Characterize the structure of the optimal solution (Optimal Substructure).
2. Recursively define the value of an optimal solution (Recursive Solution).
3. Compute the value of an optimal solution, typically in a bottom-up fashion (The algorithm).
4. Construct an optimal solution from computed information.

We shall discuss dynamic programming in detail through the following case studies. We refer to [1] for Assembly line scheduling and matrix chain multiplication and [2, 3] for 0-1 knapsack, traveling salesman, and optimal binary search trees case studies.

1 Assembly Line Scheduling in Manufacturing Sector

Problem Statement: A manufacturing company has two assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i denotes the assembly line the station is on and j denotes the number of the station. The time taken per station is denoted by $a_{i,j}$. Each station is dedicated to do some sort of work in the manufacturing process. So, a chassis must pass through each of the n stations in order before exiting the company. The parallel stations of the two assembly lines perform the same task. After it passes through station $S_{i,j}$, it will continue to station $S_{i,j+1}$ unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station $j-1$ to station j on the other line takes time $t_{i,j}$. Each assembly line takes an entry time e_i and exit time x_i. Give an algorithm for computing the minimum time from start to exit.

Objective: To find the optimal scheduling i.e., the fastest way from start to exit.
Note: let $f_i[j]$ denotes the fastest way from start to station $S_{i,j}$.

Optimal Substructure: An optimal solution to a problem is determined using optimal solutions to subproblems (in turn, sub subproblems and so on). The immediate question is, how to break the problem in to smaller sub problems ? the answer is: if we know the minimum time taken by the chassis to leave station $S_{i,j-1}$, then the minimum time taken to leave station $S_{i,j}$ can be calculated quickly by combining $a_{i,j}$ and

$t_{i,j}$.
Final Solution: $f^{OPT} = \min\{f_1[n] + x_1, f_2[n] + x_2\}$.
Base Cases: $f_1[1] = e_1 + a_{1,1}$ and $f_2[1] = e_2 + a_{2,1}$.

Recursive Solution:
The chassis at station $S_{1,j}$ can come either from station $S_{1,j-1}$ or station $S_{2,j-1}$ (Since, the tasks done by $S_{1,j}$ and $S_{2,j}$ are same). But if the chassis comes from $S_{2,j-1}$, it additionally incurs the transfer cost to change the assembly line. Thus, the recursion to reach the station j in assembly line i are as follows:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min\{f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\} & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min\{f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\} & \text{if } j \geq 2 \end{cases}$$

Why Dynamic Programming ?
The given problem is an optimization problem and the above recursion exhibits the overlapping sub problems. Since, there are two ways to reach station $S_{1,j}$, the minimum time to leave station $S_{1,j}$ can be calculated by finding the minimum time to leave the previous two stations, $S_{1,j-1}$ and $S_{2,j-1}$. Since this problem exploits both optimal substructure property and overlapping subproblems property, it is a candidate problem for dynamic programming.

The algorithm: **Fastest-way**(a, t, e, x, n) Source: CLRS

where a denotes the assembly costs, t denotes the transfer costs, e denotes the entry costs, x denotes the exit costs and n denotes the number of assembly stages.

```
1. $f_1[1] = e_1 + a_{1,1}$
2. $f_2[1] = e_2 + a_{2,1}$
3.     for $j = 2$ to $n$
4.       if $((f_1[j-1] + a_{1,j}) \leq (f_2[j-1] + t_{2,j-1} + a_{1,j}))$ then
5.         $f_1[j] = f_1[j-1] + a_{1,j}$ and $l_1[j] = 1$          /* $l_p$ denotes the line $p$ */
6.       else
7.         $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ and $l_1[j] = 2$
8.       if $((f_2[j-1] + a_{2,j}) \leq (f_1[j-1] + t_{1,j-1} + a_{2,j}))$ then
9.         $f_2[j] = f_2[j-1] + a_{2,j}$ and $l_2[j] = 2$
10.      else
11.        $f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$ and $l_2[j] = 1$
12.    end for
13.    if $(f_1[n] + x_1 \leq f_2[n] + x_2)$ then
14.      $f^{OPT} = f_1[n] + x_1$ and $l^{OPT} = 1$
15.    else
16.      $f^{OPT} = f_2[n] + x_2$ and $l^{OPT} = 2$
```

1.1 Trace of the algorithm

Consider the below figure as the input.
Iteration 1: $j = 1$ **and** $j = 2$
$f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$ and start $= S_{1,1}$
$f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$ and start $= S_{2,1}$
$f_1[2] = \min\{f_1[1] + a_{1,2}, f_2[1] + t_{2,1} + a_{1,2}\} = \min\{9 + 9, 12 + 2 + 9\} = 18$, $l_1[2] = 1$
$f_2[2] = \min\{f_2[1] + a_{2,2}, f_1[1] + t_{1,1} + a_{2,2}\} = \min\{12 + 5, 9 + 2 + 5\} = 16$, $l_2[2] = 1$

Iteration 2: $j = 3$

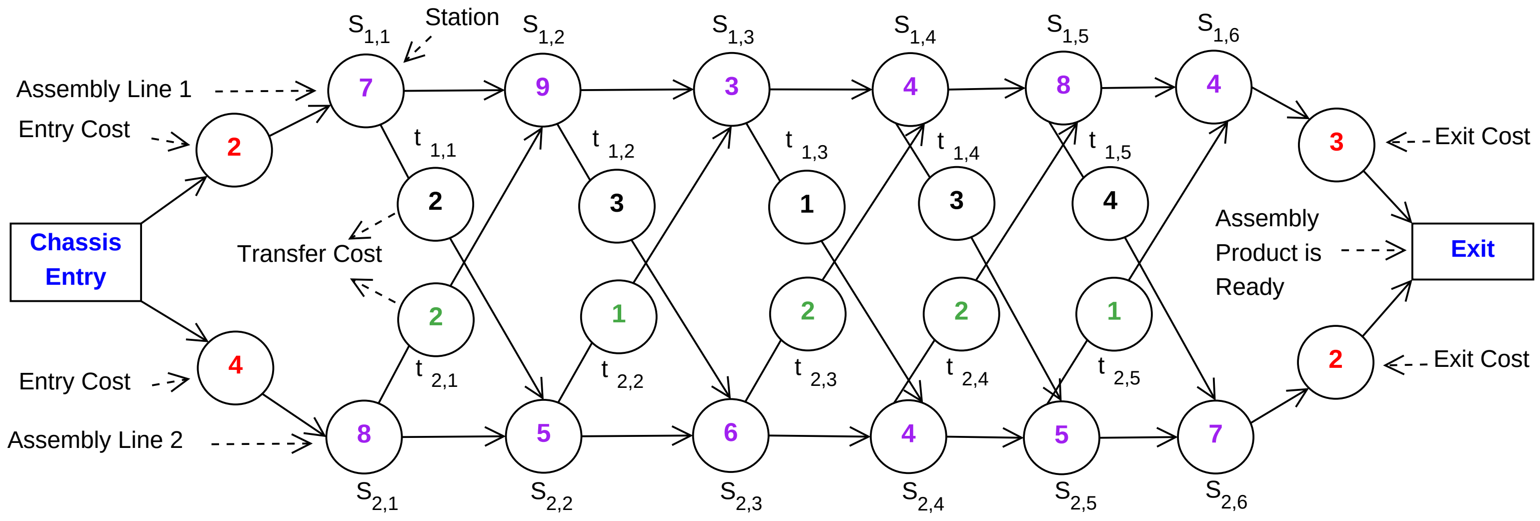


Figure 1: An illustration of assembly line scheduling problem with six stations

$f_1[3] = \min\{f_1[2] + a_{1,3}, f_2[2] + t_{2,2} + a_{1,3}\} = \min\{18 + 3, 16 + 1 + 3\} = 20,\ l_1[3] = 2$
$f_2[3] = \min\{f_2[2] + a_{2,3}, f_1[2] + t_{1,2} + a_{2,3}\} = \min\{16 + 6, 18 + 3 + 6\} = 22,\ l_2[3] = 2$

Iteration 3: $j = 4$
$f_1[4] = \min\{f_1[3] + a_{1,4}, f_2[3] + t_{2,3} + a_{1,4}\} = \min\{20 + 4, 22 + 2 + 4\} = 24,\ l_1[4] = 1$
$f_2[4] = \min\{f_2[3] + a_{2,4}, f_1[3] + t_{1,3} + a_{2,4}\} = \min\{22 + 4, 20 + 1 + 4\} = 25,\ l_2[4] = 1$

Iteration 4: $j = 5$
$f_1[5] = \min\{f_1[4] + a_{1,5}, f_2[4] + t_{2,4} + a_{1,5}\} = \min\{24 + 8, 25 + 2 + 8\} = 32,\ l_1[5] = 1$
$f_2[5] = \min\{f_2[4] + a_{2,5}, f_1[4] + t_{1,4} + a_{2,5}\} = \min\{25 + 5, 24 + 3 + 5\} = 30,\ l_2[5] = 2$

Iteration 5: $j = 6$
$f_1[6] = \min\{f_1[5] + a_{1,6}, f_2[5] + t_{2,5} + a_{1,6}\} = \min\{32 + 4, 30 + 1 + 4\} = 35,\ l_1[6] = 2$
$f_2[6] = \min\{f_2[5] + a_{2,6}, f_1[5] + t_{1,5} + a_{2,6}\} = \min\{30 + 7, 32 + 4 + 7\} = 37,\ l_2[6] = 2$
Optimum Schedule: $\min\{f_1[6] + x_1, f_2[6] + x_2\} = \{35 + 3, 37 + 2\} = 38$; $l^{OPT} = 1$.

Constructing an optimal solution: We now show how to construct an optimal solution using the above table values. Note that $l_i[j]$ stores the line number (1 or 2) that minimizes the total cost from start to station $j - 1$. The optimal solution from start to exit comes from the station $S_{1,6}$ (*ref. Iteration 5*). The fastest way to reach the station $S_{1,6}$ from start, $f_1[6]$, comes from $f_2[5]$ i.e., from the station $S_{2,5}$ (*ref. Iteration 4*). The fastest way to reach the station $S_{2,5}$ from start, $f_2[5]$, comes from $f_2[4]$ i.e., from the station $S_{2,4}$ (*ref. Iteration 3*). The fastest way to reach the station $S_{2,4}$ from start, $f_2[4]$, comes from $f_1[3]$ i.e., from the station $S_{1,3}$ (*ref. Iteration 2*). The fastest way to reach the station $S_{1,3}$ from start, $f_1[3]$, comes from $f_2[2]$ i.e., from the station $S_{2,2}$ (*ref. Iteration 1*). The fastest way to reach the station $S_{2,2}$ from start, $f_2[2]$, comes from $f_1[1]$ i.e., from the station $S_{1,1}$, which is one of the base cases (*ref. Iteration 1*). Thus, the optimal schedule is:
$Chassis\ \ Start \to e_1 \to S_{1,1} \to t_{1,1} \to S_{2,2} \to t_{2,2} \to S_{1,3} \to t_{1,3} \to S_{2,4} \to S_{2,5} \to t_{2,5} \to S_{1,6} \to x_1 \to exit.$

Run time of this algorithm is $\theta(n)$
Note: Brute force method of finding one optimal solution takes exponential time i.e., it takes 2^n possibilities for n scheduled sub tasks, as each station has two choices.

2 Optimal Order in Matrix Chain Multiplication

Our next case study for dynamic programming is to find the optimal order to solve matrix chain multiplication. Given a sequence $\{A_1, A_2, A_3, \ldots, A_n\}$ of n matrices, our **objective** is to find an optimal order to multiply a sequence of matrices minimizing the number of scalar multiplications. Note that the objective is not to multiply matrices, instead, to identify the order of multiplications. Since the matrix multiplication is associative, we have many options to multiply a chain of matrices. i.e., no matter how we parenthesize the product, the result will be same. For example, if the chain of matrices is $\{A_1, A_2, A_3, A_4\}$, then we can fully parenthesis (either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses) the product $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ in five distinct ways:

1. $((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$
2. $(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$
3. $(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$
4. $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$
5. $(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$

We have to choose the one among the above five ways which gives the minimum number of multiplications. For example, suppose if all A_1, A_2, A_3 and A_4 are a 10×10 matrix. Then, the number of multiplications in all the five ways is $10^3 + 10^3 + 10^3 = 3000$. Suppose if the order of each matrix is different, say, order of A_1 is 10×20, A_2 is 20×10, A_3 is 10×10 and A_4 is 10×20.
The number of multiplications in the first way is $10 \times 20 \times 10 + 10 \times 10 \times 20 + 10 \times 10 \times 20 = 6000$.
The number of multiplications in the second way is $10 \times 20 \times 10 + 10 \times 10 \times 10 + 10 \times 10 \times 20 = 5000$.
The number of multiplications in the third way is $10 \times 10 \times 20 + 20 \times 10 \times 20 + 10 \times 20 \times 20 = 10000$.
The number of multiplications in the fourth way is $20 \times 10 \times 10 + 10 \times 20 \times 10 + 10 \times 10 \times 20 = 6000$.
The number of multiplications in the fifth way is $20 \times 10 \times 10 + 20 \times 10 \times 20 + 10 \times 20 \times 20 = 10000$.
Clearly, the second parenthesization requires the least number of operations.

Optimal Substructure:
Optimal solution to $A_1 \cdot A_2 \cdot A_3 \cdots A_n$ contains within optimal solution to $(A_1 \cdot A_2 \cdots A_k)(A_{k+1} \cdot A_{k+2} \cdots A_n)$.

Recursive Solution:
Let $M[i,j]$ denotes the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$. The order of a matrix A_i be (p_{i-1}, p_i) and the order of $A_{i \ldots k}$ be (p_{i-1}, p_k). Now, let us define $M[i,j]$ recursively as follows:

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{M[i,k] + M[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

The $M[i,j]$ values give the costs of optimal solutions to sub problems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $S[i,j]$ to be a value of k at which we split the product $A_i \cdot A_{i+1} \cdots A_j$ in an optimal parenthesization.

Why Dynamic Programming ?
The given problem is an optimization problem and the above recursion exhibits the overlapping sub problems. Since, for each choice of i and j satisfying $1 \le i \le j \le n$, computing $M[i,j]$ requires $M[i,k]$ and $M[k+1,n]$, $i \le k < j$. A recursive algorithm may encounter each sub problem many times in different branches of its recursion tree. As this case study satisfies, overlapping sub problems and optimal substructure properties, candidate problem for applying dynamic programming.

The algorithm: **Matrix-chain-order(p)** Source: CLRS [1]

Input: Sequence of matrices $A_1, A_2, \ldots, A_n$ with order $< p_0, p_1 >, < p_1, p_2 >, \ldots, < p_{n-1}, p_n >$ respectively.

```
1.$n = length(p) - 1$
2.      for $i = 1$ to $n$
3.        $M[i,i] = 0$
4.      for $l = 2$ to $n$                 /* $l$ denotes the length of the chain */
5.        for $i = 1$ to $n - l + 1$
6.          $j = i + l - 1$
7.          $M[i,j] = \infty$
8.          for $k = i$ to $j - 1$
9.            $q = M[i,k] + M[k+1,j] + p_{i-1}p_kp_j$
10.           $q < M[i,j]$
11.             $M[i,j] = q$
12.             $S[i,j] = k$
13.Return $M$ and $S$
```

2.1 Trace of the algorithm

Consider an example for matrix chain order where $n = 4$ (thus, the length of the chain is 3) and the matrix dimensions are as follows:

matrix	A_1	A_2	A_3	A_4
dimensions	2×3	3×4	4×4	4×1

Iteration 1:
$n = length(p) - 1 = 5 - 1 = 4$ $(length(p) = length(p_0p_1p_2p_3p_4) = 5)$
$M[1,1] = M[2,2] = M[3,3] = M[4,4] = 0$

$$\left.\begin{array}{r} l = 2 \\ i = 1 \\ j = 2 \end{array}\right\} (k = 1 \;\; to \;\; 1) \Rightarrow q = M[1,1] + M[2,2] + 2 \cdot 3 \cdot 4 = 24$$

Since, $q < M[1,2] = \infty$, $M[1,2] = 24$ and $S[1,2] = 1$

Iteration 2:

$$\left.\begin{array}{r} l = 2 \\ i = 2 \\ j = 3 \end{array}\right\} (k = 2 \;\; to \;\; 2) \Rightarrow q = M[2,2] + M[3,3] + 3 \cdot 4 \cdot 4 = 48$$

Since, $q < M[2,3] = \infty$, $M[2,3] = 48$ and $S[2,3] = 2$

Iteration 3:

$$\left.\begin{array}{r} l = 2 \\ i = 3 \\ j = 4 \end{array}\right\} (k = 3 \;\; to \;\; 3) \Rightarrow q = M[3,3] + M[4,4] + 4 \cdot 4 \cdot 1 = 16$$

Since, $q < M[3,4] = \infty$, $M[3,4] = 16$ and $S[3,4] = 3$

Iteration 4:

$$\left.\begin{array}{r} l = 3 \\ i = 1 \\ j = 3 \end{array}\right\} \begin{array}{l} M[1,3] = \infty \\ (k = 1 \;\; to \;\; 2) \end{array} \begin{array}{l} k = 1 \Rightarrow q = M[1,1] + M[2,3] + 2 \cdot 3 \cdot 4 = 0 + 48 + 24 = 72; 72 < M[1,3]; M[1,3] = 72; S[1,3] = 1 \\ k = 2 \Rightarrow q = M[1,2] + M[3,3] + 2 \cdot 4 \cdot 4 = 24 + 0 + 32 = 56; 56 < M[1,3]; M[1,3] = 56; S[1,3] = 2 \end{array}$$

Thus, $M[1,3] = 56$ and $S[1,3] = 2$

Iteration 5:

$$\left.\begin{array}{r} l = 3 \\ i = 2 \\ j = 4 \end{array}\right\} \begin{array}{l} M[2,4] = \infty \\ (k = 2 \;\; to \;\; 3) \end{array} \begin{array}{l} k = 2 \Rightarrow q = M[2,2] + M[3,4] + 3 \cdot 4 \cdot 1 = 0 + 16 + 12 = 28; 28 < M[2,4]; M[2,4] = 28; S[2,4] = 2 \\ k = 3 \Rightarrow q = M[2,3] + M[4,4] + 3 \cdot 4 \cdot 1 = 48 + 0 + 12 = 60; 60 \nless M[2,4] \end{array}$$

Thus, $M[2,4] = 28$ and $S[2,4] = 2$

Iteration 6:

$$\left.\begin{array}{c} l=4\\ i=1\\ j=4 \end{array}\right\} \begin{array}{c} M[1,4]=\infty\\ (k=1\ \ to\ \ 3) \end{array} \begin{array}{l} k=1 \Rightarrow q = M[1,1]+M[2,4]+2\cdot 3\cdot 1 = 0+28+6=34; 34 < M[1,4]; M[1,4]=34; S[1,4]=1\\ k=2 \Rightarrow q = M[1,2]+M[3,4]+2\cdot 4\cdot 1 = 24+16+8=48; 48 \not< M[1,4]\\ k=3 \Rightarrow q = M[1,3]+M[4,4]+2\cdot 4\cdot 1 = 56+0+8=74; 74 \not< M[1,4] \end{array}$$

Thus, $M[1,4] = 34$ and $S[1,4] = 1$

$i \mid j$	1	2	3	4
1	0	24	56	34
2		0	48	28
3			0	16
4				0

Constructing an optimal solution: The minimum number of scalar multiplications needed to compute the product $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ with the given order is 34. The corresponding parenthesization is $(A_1 \cdot (A_2 \cdot A_3 \cdot A_4))$ (Since, the minimum of $M[1,4]$ comes from $M[1,1]$ and $M[2,4]$) $= (A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$ (Since, the minimum of $M[2,4]$ comes from $M[2,2]$ and $M[3,4]$). Hence, the optimal order for the given matrix chain multiplication is $(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$.

Run time of this algorithm is $O(n^3)$, whereas, the trivial algorithm runs in $\Omega(x)$ time, where x is the number of ways of parenthesizing a sequence of matrices. The number of ways of parenthesizing a chain of matrices is catalan number, i.e., x is $\frac{1}{n+1}\binom{2n}{n}$.

3 Knapsack Problem

Given a set of objects $x_1, x_2, \ldots, x_n$ with weights $w_1, w_2, \ldots, w_n$, profits $p_1, p_2, \ldots, p_n$, and the capacity of the knapsack W. Our **objective** is to find a subset $S \subseteq \{x_1, x_2, \ldots, x_n\}$ of maximum profit, subject to the constraints,

$$\sum_{i \in S} w_i \leq W$$

.

Optimal Substructure:
Let OPT denotes the optimum solution. If n^{th} object is not part of OPT, then $OPT(n, W) = OPT(n-1, W)$. Otherwise, $OPT(n, W) = p_n + OPT(n-1, W - w_n)$. i.e., if n is not a part of the optimum solution, then finding the optimum solution for n objects is equivalent to finding the optimum solution for $n-1$ objects (excluding the n^{th} object). Suppose, if n is a part of the optimum solution, then optimum solution for n objects is equivalent to the optimum solution for $n-1$ objects with $W = W - w_n$ + profit of the n^{th} object.

Recursive sub problem:
If $W < w_i$, then $OPT(i, W) = OPT(i-1, W)$.
Otherwise, $OPT(i, W) = \max(OPT(i-1, W), p_i + OPT(i-1, W - w_i))$.
Why Dynamic Programming ?
The given problem is an optimization problem and the above recursion exhibits the overlapping sub problems. Since, for each choice of i, computing $OPT(i, W)$ requires $OPT(i-1, W)$. A recursive algorithm may encounter each sub problem many times in different branches of its recursion tree. Due to the presence of optimal substructure and overlapping sub problems properties, this is a candidate problem for dynamic programming.

```
The algorithm: Profit-Knapsack($n, W$)

1.Construct a two-dimensional array $M[0, \dots, n; 0, \dots, W]$
2.Initialize $M[0, w] = 0, \forall \ w = 0, \dots, W$
3.for $i = 1$ to $n$
4.    for $w = 0$ to $W$
5.        if $(w < w_i)$ then
6.            $M[i, w] = M[i-1, w]$
7.        else
8.            $M[i, w] = \max\{M[i-1, w], p_i + M[i-1, w-w_i]\}$
```

3.1 Trace of the algorithm

Consider the following example: Number of objects, $n = 3$. Capacity of knapsack, $W = 5$. Weights of the objects (x_1, x_2, x_3) are $(w_1, w_2, w_3) = (1, 2, 3)$. Profits of the objects (x_1, x_2, x_3) are $(p_1, p_2, p_3) = (3, 2, 1)$. Now let us trace the algorithm:

Iteration 1:
Construct a two-dimensional array M of size 4×6
Initialize $M[0, 0] = M[0, 1] = M[0, 2] = M[0, 3] = M[0, 4] = M[0, 5] = 0$.

$$\left.\begin{array}{l} i = 1 \\ w = 0 \end{array}\right\} 0 < w_1 = 1 \Rightarrow M[1, 0] = M[0, 0] = 0$$

Iteration 2:

$$\left.\begin{array}{l} i = 1 \\ w = 1 \end{array}\right\} 1 \not< w_1 = 1 \Rightarrow M[1, 1] = \max\{M[0, 1], p_1 + M[0, 0]\} = \max\{0, 3 + 0\} = 3$$

Iteration 3:

$$\left.\begin{array}{l} i = 1 \\ w = 2 \end{array}\right\} 2 \not< w_1 = 1 \Rightarrow M[1, 2] = \max\{M[0, 2], p_1 + M[0, 2-1]\} = \max\{0, 3 + 0\} = 3$$

Iteration 4:

$$\left.\begin{array}{l} i = 1 \\ w = 3 \end{array}\right\} 3 \not< w_1 = 1 \Rightarrow M[1, 3] = \max\{M[0, 3], p_1 + M[0, 3-1]\} = \max\{0, 3 + 0\} = 3$$

Iteration 5:

$$\left.\begin{array}{l} i = 1 \\ w = 4 \end{array}\right\} 4 \not< w_1 = 1 \Rightarrow M[1, 4] = \max\{M[0, 4], p_1 + M[0, 4-1]\} = \max\{0, 3 + 0\} = 3$$

Iteration 6:

$$\left.\begin{array}{l} i = 1 \\ w = 5 \end{array}\right\} 5 \not< w_1 = 1 \Rightarrow M[1, 5] = \max\{M[0, 5], p_1 + M[0, 5-1]\} = \max\{0, 3 + 0\} = 3$$

Iteration 7:

$$\left.\begin{array}{l} i = 2 \\ w = 0 \end{array}\right\} 0 < w_2 = 2 \Rightarrow M[2, 0] = M[1, 0] = 0$$

Iteration 8:

$$\left.\begin{array}{l} i = 2 \\ w = 1 \end{array}\right\} 1 < w_2 = 2 \Rightarrow M[2, 1] = M[1, 1] = 3$$

Iteration 9:

$$\left.\begin{array}{l} i = 2 \\ w = 2 \end{array}\right\} 2 \not< w_2 = 2 \Rightarrow M[2, 2] = \max\{M[1, 2], p_2 + M[1, 2-2]\} = \max\{3, 2 + 0\} = 3$$

Iteration 10:

$$\left.\begin{array}{l} i = 2 \\ w = 3 \end{array}\right\} 3 \not< w_2 = 2 \Rightarrow M[2, 3] = \max\{M[1, 3], p_2 + M[1, 3-2]\} = \max\{3, 2 + 3\} = 5$$

Iteration 11:

$$\left.\begin{array}{l} i=2 \\ w=4 \end{array}\right\} 4 \not< w_2 = 2 \Rightarrow M[2,4] = \max\{M[1,4], p_2 + M[1,4-2]\} = \max\{3, 2+3\} = 5$$

Iteration 12:
$$\left.\begin{array}{l} i=2 \\ w=5 \end{array}\right\} 5 \not< w_2 = 2 \Rightarrow M[2,5] = \max\{M[1,5], p_2 + M[1,5-2]\} = \max\{3, 2+3\} = 5$$

Iteration 13:
$$\left.\begin{array}{l} i=3 \\ w=0 \end{array}\right\} 0 < w_3 = 3 \Rightarrow M[3,0] = M[2,0] = 0$$

Iteration 14:
$$\left.\begin{array}{l} i=3 \\ w=1 \end{array}\right\} 1 < w_3 = 3 \Rightarrow M[3,1] = M[2,1] = 3$$

Iteration 15:
$$\left.\begin{array}{l} i=3 \\ w=2 \end{array}\right\} 2 < w_3 = 3 \Rightarrow M[3,2] = M[2,2] = 3$$

Iteration 16:
$$\left.\begin{array}{l} i=3 \\ w=3 \end{array}\right\} 3 \not< w_3 = 3 \Rightarrow M[3,3] = \max\{M[2,3], p_3 + M[2,3-3]\} = \max\{5, 1+0\} = 5$$

Iteration 17:
$$\left.\begin{array}{l} i=3 \\ w=4 \end{array}\right\} 4 \not< w_3 = 3 \Rightarrow M[3,4] = \max\{M[2,4], p_3 + M[2,4-3]\} = \max\{5, 1+3\} = 5$$

Iteration 18:
$$\left.\begin{array}{l} i=3 \\ w=5 \end{array}\right\} 5 \not< w_3 = 3 \Rightarrow M[3,5] = \max\{M[2,5], p_3 + M[2,5-3]\} = \max\{5, 1+3\} = 5$$

The corresponding table is as follows:

$n=3$	0	3	3	5	5	5
$n=2$	0	3	3	5	5	5
$n=1$	0	3	3	3	3	3
$n=0$	0	0	0	0	0	0
	$w=0$	$w=1$	$w=2$	$w=3$	$w=4$	$w=5$

Constructing an optimal solution: The value of $M[3,5]$ in the above table denotes how much maximum profit can the thief get with the capacity of his knapsack, W, i.e., $OPT(3,5) = M[3,5]$. $M[3,5]$ is obtained by $M[2,5]$, which says us not to include the object x_3 but still we are able to get the profit 5 (*ref Iteration 18*). $M[2,5]$ is obtained by $p_2 + M[1,3]$, which says us to include the object x_2 to get the profit 5 (*ref Iteration 12*). $M[1,3]$ is obtained by $p_1 + M[0,2]$, which says us to include the object x_1 to get the profit 5 (*ref Iteration 4*). Thus, the thief has to take the objects x_1 and x_2 such that the profit is $p_1 + p_2 = 3 + 2 = 5$. Hence, the solution set is $(x_1, x_2, x_3) = (1, 1, 0)$, where 1 denotes the presence of the object in the knapsack and 0 denotes the absence of the object in the knapsack.

Run time of this algorithm is $O(nW)$, where n denotes the number of objects and W denotes the maximum weight of objects that the thief can put in his knapsack. Though, the run time appears like polynomial time, it purely depends on the value of W. For example, if W is $O(n)$ then the run time is polynomial in n and if W is $O(2^n)$ then the run time is exponential in n, as we have to construct a matrix of size $(n+1) \times (W+1)$. Thus, the run-time of this algorithm depends on W and algorithms of this type whose run-time includes a variable (in this case W) are known as pseudo-polynomial time algorithms.

4 Optimal Binary Search Tree

The first task in any compilation process of a program is parsing, which scans the program and identifies the set of keywords and non-keywords. For example, if you mistype *int* a, b as *ant* a, b, compiler has to show an error message "Syntax error". The idea is, for the given set of keywords such as while, do, if and so on, we should maintain a nice data structure so that when parsing is done it should display the keywords and non-keywords as quickly as possible. In this section, we shall present a data structure called "*Binary Search Tree (BST)*"which helps to minimize the parsing time. For the given set of keywords there are many BST's, the objective is to choose a BST that minimizes the search cost. For example, the set of keywords are $\{do, if, while\}$ and the possible BST's are as follows:

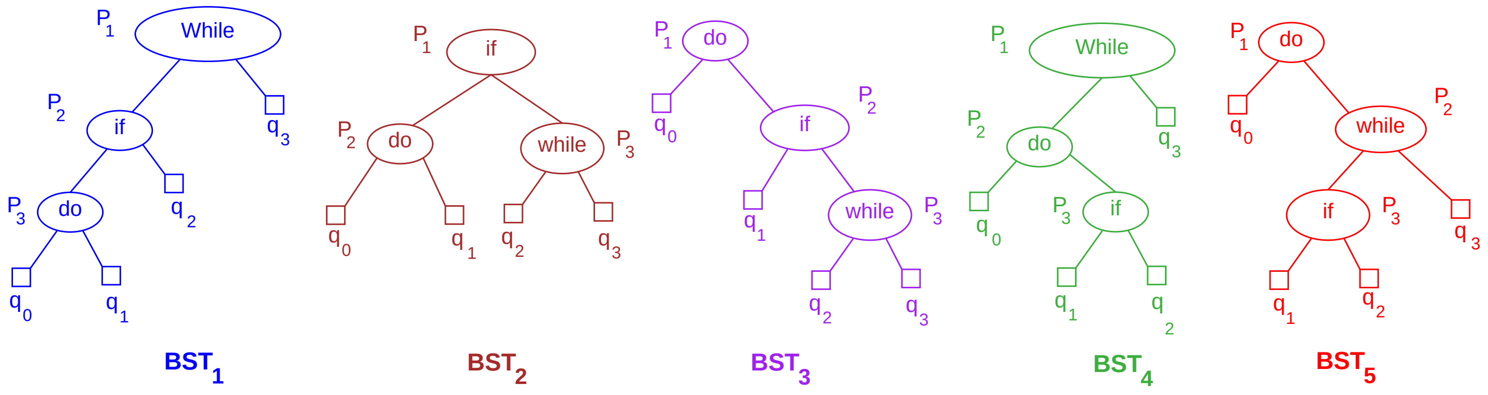


Figure 2: All possible BST's for a set of keywords $\{do, if, while\}$

Suppose if $p(i) = \frac{1}{7}$ *for all* $i =$ *1 to 3* and $q(i) = \frac{1}{7}$ *for all* $i =$ *0 to 3*, where $p(i)$ denotes the probability of occurrence of the keyword a_i and $q(i)$ denotes the probability of the non-keyword x such that $a_i < x < a_{i+1}$. Note that $q(i)$ denotes how many times (frequency or probability) syntax error is thrown by the compiler with respect to x. $p(i)$ denotes how many times (frequency or probability) the key word a_i is referred in a program.
Cost of search for each BST is as follows:

$$Cost(BST_1) = \sum_{i=1}^{3} p_i \times \text{height of } p_i + \sum_{i=0}^{3} q_i \times (\text{height of } q_i - 1) = \frac{1}{7}\cdot 1 + \frac{1}{7}\cdot 2 + \frac{1}{7}\cdot 3 + \frac{1}{7}\cdot 3 + \frac{1}{7}\cdot 3 + \frac{1}{7}\cdot 2 + \frac{1}{7}\cdot 1 = \frac{15}{7}$$

$$Cost(BST_2) = \sum_{i=1}^{3} p_i \times \text{height of } p_i + \sum_{i=0}^{3} q_i \times (\text{height of } q_i - 1) = \frac{1}{7}\cdot(1+2+2) + \frac{1}{7}\cdot(2+2+2+2) = \frac{13}{7}$$

$$Cost(BST_3) = \sum_{i=1}^{3} p_i \times \text{height of } p_i + \sum_{i=0}^{3} q_i \times (\text{height of } q_i - 1) = \frac{1}{7}\cdot(1+2+3) + \frac{1}{7}\cdot(1+2+3+3) = \frac{15}{7}$$

$$Cost(BST_4) = \sum_{i=1}^{3} p_i \times \text{height of } p_i + \sum_{i=0}^{3} q_i \times (\text{height of } q_i - 1) = \frac{1}{7}\cdot(1+2+3) + \frac{1}{7}\cdot(1+2+3+3) = \frac{15}{7}$$

$$Cost(BST_5) = \sum_{i=1}^{3} p_i \times \text{height of } p_i + \sum_{i=0}^{3} q_i \times (\text{height of } q_i - 1) = \frac{1}{7}\cdot(1+2+3) + \frac{1}{7}\cdot(1+2+3+3) = \frac{15}{7}$$

Clearly, BST_2 is optimal, as the search cost is less compared to other BST's.
Objective: Given a set of identifiers (keywords) $\{a_1, a_2, \ldots, a_n\}$ with the constraint $a_1 < a_2 < \ldots < a_n$. Our objective is to construct an Optimal Binary Search Tree (OPT BST), which minimizes the cost of search.

Optimal Substructure:
Given a set of identifiers, how to identify the right BST ?

- $\rightarrow$ To apply dynamic programming, we need to identify the right root.
- $\rightarrow$ Among $\{a_1, \dots, a_n\}$, which a_i would be the root.
- $\rightarrow$ Say a_k is the root of an optimal BST, then $a_1, a_2, \dots, a_{k-1}$ lie in the left sub tree of the root and $a_{k+1}, \dots, a_n$ lie in the right sub tree of the root.
- $\rightarrow$ Let $w(i,j) = q(i) + \sum\limits_{l=i+1}^{j} q(l) + p(l)$
- $\rightarrow$ The cost of the BST with a_k as the root is calculated using the cost of left subtree and right subtree plus the increase in cost due to a_k. The total cost with a_k as the root is:
  $Cost(l) + Cost(r) + p_k + w(0, k-1) + w(k, n)$,
  which is the cost of left sub tree + cost of right sub tree + $p_k \times 1$ + increased cost with respect to the left sub tree (i.e., $w(0, k-1)$ since the height increases by one) + increased cost with respect to the right sub tree i.e., $w(k, n)$ since the height increases by one).
- $\rightarrow$ If the tree is optimal, then the above expression must be minimum. Hence, $Cost(l)$ must be minimum over all BST containing $a_1, a_2, \dots, a_{k-1}$ and $q_0, q_1, \dots, q_{k-1}$. Similarly, $Cost(r)$ must be minimum. Thus, for the minimum k, the expected cost is
  $p(k) + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n)$.
  $p(k)$: Cost of the root
  $c(0, k-1)$: cost of left subtree $Cost(l)$, for simplicity we use 'c' instead of 'cost'
  $c(k, n)$: $Cost(r)$
  $w(0, k-1) + w(k, n)$: Increase in cost due to the increase in height.

Hence, the optimal substructure is $c(0,n) = \min\limits_{1 \le k < n} \{p(k) + c(0,k-1) + c(k,n) + w(0,k-1) + w(k,n)\}$

Recursive sub problem:
In general for any $c(i,j)$,

$$c(i,j) = min_{i<k\le j}\{c(i,k-1) + c(k,j) + p(k) + w(i,k-1) + w(k,j)\} = \min_{i<k\le j}\{c(i,k-1) + c(k,j)\} + w(i,j)$$

$c(i,j)$ can be solved for $c(0,n)$ by first computing all $c(i,j)$ such that $j - i = 1$, next we can compute all $c(i,j)$ such that $j - i = 2$ and so on.
Note: $c(i,i) = 0$ and $w(i,i) = q(i)$ for all $0 \le i \le n$. Also, $w(i,j) = p(j) + q(j) + w(i,j-1)$

Why Dynamic Programming ? The given problem is an optimization problem and it is clear from the above discussion that this problem exhibits the overlapping sub problem and optimal substructure properties. Since, for each choice of i and j, computing $c(i,j)$ requires $c(i,k-1)$ and $c(k,j)$, $i < k \le j$.

The algorithm: **OPT BST**(p, q, n)

Given n distinct identifiers $a_1 < a_2 < \ldots < a_n$ and probabilities $p[i]$ and $q[i]$ this algorithm computes the cost $c[i, j]$ of optimal binary search trees t_{ij} for identifiers $a_{i+1}, \ldots, a_j$. It also computes $r[i, j]$, the root of t_{ij} and $w[i, j]$, the weight of t_{ij}.

```
1.for $i = 0$ to $n-1$ do
2.    $w[i,i] = q[i]$; $r[i,i] = 0$; $c[i,i] = 0$      /* Optimal BST with one node */
3.    $w[i,i+1] = q[i] + q[i+1] + p[i+1]$
4.    $r[i,i+1] = i+1$
5.    $c[i,i+1] = q[i] + q[i+1] + p[i+1]$
6.$w[n,n] = q[n]$; $r[n,n] = c[n,n] = 0$

6.5 Run two for loops and update $w(i,j) = p(j) + q(j) + w(i,j-1)$

7.for $m = 2$ to $n$ do
8.    for $i = 0$ to $n-m$ do
9.        $j = i + m$
10.         Find a $k$ in the range $i < k \leq j$ that minimizes $c[i,k-1] + c[k,j]$
11.         Compute $c[i,j] = w[i,j] + c[i,k-1] + c[k,j]$
12.         $r[i,j] = k$
```

4.1 Trace of the algorithm

Consider the following example:
$(a_1, a_2, a_3, a_4) = (cout, float, if, while)$ with the following probabilities:

i	0	1	2	3	4
$p(i)$		$\frac{1}{20}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{20}$
$q(i)$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{5}$	$\frac{1}{20}$	$\frac{1}{20}$

Now, let us trace the algorithm for the above example:

Iteration 1: (first loop)
$w[0,0] = q[0] = \frac{1}{5}$; $r[0,0] = c[0,0] = 0$
$w[0,1] = p[1] + q[1] + w[0,0] = \frac{1}{20} + \frac{1}{10} + \frac{1}{5} = 0.05 + 0.1 + 0.2 = 0.35$
$c[0,1] = \min\{c[0,0] + c[1,1]\} + w[0,1] = 0.35$ and $r[0,1] = 1$

Iteration 2: (first loop)
$w[1,1] = q[1] = \frac{1}{10} = 0.1$; $r[1,1] = c[1,1] = 0$
$w[1,2] = p[2] + q[2] + w[1,1] = 0.2 + 0.2 + 0.1 = 0.5$
$c[1,2] = \min\{c[1,1] + c[2,2]\} + w[1,2] = 0.5$ and $r[1,2] = 2$

Iteration 3: (first loop)
$w[2,2] = q[2] = \frac{1}{5} = 0.2$; $r[2,2] = c[2,2] = 0$
$w[2,3] = p[3] + q[3] + w[2,2] = 0.1 + 0.05 + 0.2 = 0.35$
$c[2,3] = \min\{c[2,2] + c[3,3]\} + w[2,3] = 0.35$ and $r[2,3] = 3$

Iteration 4: (first loop)
$w[3,3] = q[3] = \frac{1}{20} = 0.05$; $r[3,3] = c[3,3] = 0$
$w[3,4] = p[4] + q[4] + w[3,3] = 0.05 + 0.05 + 0.05 = 0.15$
$c[3,4] = \min\{c[3,3] + c[4,4]\} + w[3,4] = 0.15$ and $r[3,4] = 4$

$w[4,4] = q[4] = \frac{1}{20} = 0.05$; $r[4,4] = c[4,4] = 0$

Iteration 5: (second loop when $j - i = 2$)
$m = 2$; $i = 0$; $j = 0 + 2 = 2$:
$w[0,2] = p[2] + q[2] + w[0,1] = 0.2 + 0.2 + 0.35 = 0.75$
$c[0,2] = \min\{c[0,0] + c[1,2], c[0,1] + c[2,2]\} + w[0,2] = \min\{0.5, 0.35\} + 0.75 = 1.1$ and $r[0,2] = 2$

Iteration 6: (second loop when $j - i = 2$)
$m = 2$; $i = 1$; $j = 1 + 2 = 3$:
$w[1,3] = p[3] + q[3] + w[1,2] = 0.05 + 0.1 + 0.5 = 0.65$
$c[1,3] = \min\{c[1,1] + c[2,3], c[1,2] + c[2,3]\} + w[1,3] = \min\{0.35, 0.5\} + 0.65 = 1.0$ and $r[1,3] = 2$

Iteration 7: (second loop when $j - i = 2$)
$m = 2$; $i = 2$; $j = 2 + 2 = 4$:
$w[2,4] = p[4] + q[4] + w[2,3] = 0.05 + 0.05 + 0.35 = 0.45$
$c[2,4] = \min\{c[2,2] + c[3,4], c[2,3] + c[4,4]\} + w[2,4] = \min\{0.15, 0.35\} + 0.45 = 0.6$ and $r[2,4] = 3$

Iteration 8: (second loop when $j - i = 3$)
$m = 3$; $i = 0$; $j = 0 + 3 = 3$:
$w[0,3] = p[3] + q[3] + w[0,2] = 0.1 + 0.05 + 0.75 = 0.9$
$c[0,3] = \min\{c[0,0] + c[1,3], c[0,1] + c[2,3], c[0,2] + c[3,3]\} + w[0,3] = \min\{1.0, 0.35 + 0.35, 1.1\} + 0.9 = 1.6$ and $r[0,3] = 2$

Iteration 9: (second loop when $j - i = 3$)
$m = 3$; $i = 1$; $j = 1 + 3 = 4$:
$w[1,4] = p[4] + q[4] + w[1,3] = 0.05 + 0.05 + 0.65 = 0.75$
$c[1,4] = \min\{c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4]\} + w[1,4] = \min\{0.6, 0.5 + 0.15, 1.0\} + 0.75 = 1.35$ and $r[1,4] = 2$

Iteration 10: (second loop when $j - i = 4$)
$m = 4$; $i = 0$; $j = 0 + 4 = 4$:
$w[0,4] = p[4] + q[4] + w[0,3] = 0.05 + 0.05 + 0.9 = 1.0$
$c[0,4] = \min\{c[0,0]+c[1,4], c[0,1]+c[2,4], c[0,2]+c[3,4], c[0,3]+c[4,4]\}+w[0,4] = \min\{1.35, 0.35+0.6, 1.1+0.15, 1.6\} + 1.0 = 1.95$ and $r[0,4] = 2$

Constructing an optimal solution:
The computation of $c[0,4]$ says that the root of the OPT BST is a_2 i.e., *float* (since the value of $r[0,4]$ is 2). Hence, the left sub tree for a_2 contains a_1 and the right sub tree contains a_3 and a_4. The left sub tree of a_2 has only one node, so the left node of a_2 is a_1 i.e., *cout*. The right sub tree of a_2 has two nodes, so to fix the right node of a_2 (which is either a_3 or a_4), we look at $c[2,4]$, which says the right node of a_2 is a_3 i.e., *if* (since, $r[2,4] = 3$). Thus, the left sub tree for the node *if* is NULL and the right sub tree contains a_4 i.e., *while* (*ref Figure 3*).

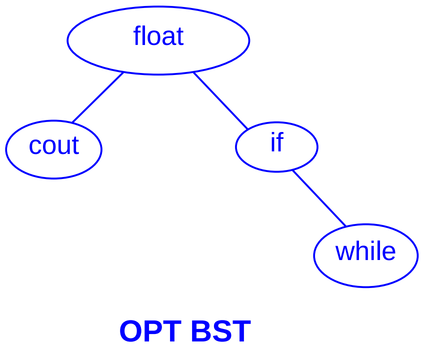


Figure 3: Optimal BST for the given example

Run time analysis
Let n denotes the the number of keywords. The time taken in first loop is $n-2+1+1 = O(n)$. The time taken in second loop is as follows:

$$\left.\begin{array}{l} m = 2 \Rightarrow n-2+1+1 = n \\ m = 3 \Rightarrow n-3+1+1 = n-1 \\ m = 4 \Rightarrow n-2+1+1 = n-2 \\ \vdots \\ m = n \Rightarrow n-n+1 = 1 \end{array}\right\} 1+2+\ldots+n = O(n^2).$$

To find minimum k we incur $O(m)$ time (because, $j-i=m$). i.e., $O(n)$ time. So, the total time complexity is $O(n^2)\cdot O(n) = O(n^3)$.
A simple brute force (trivial) algorithm incurs $\Omega(x)$, where x denotes the number of BST's which is again close to catalan number.

5 Traveling Salesman Problem

Input: A directed graph $G=(V,E)$ with edge costs.
Objective: To find a directed tour of minimum cost by visiting each vertex exactly once (except the start node)

Note:

1. Every tour starting at vertex 'i'consists of an edge $<i,k>$ for some $k \in V\backslash\{1\}$ and a path from vertex k to vertex 1. The path from k to 1 goes through each vertex in $V\backslash\{1,k\}$ exactly once.
2. If the path (tour) is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V\backslash\{1,k\}$
3. Hence, the principle of optimality holds.

Notation: $g(i,S)$ denotes the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1. Let C_{ij} denotes the edge cost between i and j. $C_{ij} > 0$ for all i,j and $C_{ij} = \infty$ if $(i,j) \notin E(G)$.

Optimal Substructure:
Our goal is to find $g(1,V\backslash\{1\})$. From the principle of optimality it follows that,
$g(1,V\backslash\{1\}) = \min_{2\le k\le n}\{C_{1k} + g(k, V\backslash\{1,k\})\}$

Recursive sub problem:
The principle of optimality helps in identifying sub problems, sub subproblems, etc., To solve $g(1,V\backslash\{1\})$ we need to know $g(k,V\backslash\{1,k\})$ for all choices of k. This can be calculated using,

$$g(i,S) = \min_{j \in S; i \neq j} \{C_{ij} + g(i, S \backslash \{j\})\}$$

Note: $g(i,\emptyset) = C_{i1}$, $1 \leq i \leq n$, which says from i without going through any set reach vertex 1.

Why Dynamic Programming ? The given problem is an optimization problem and the above recursion exhibits the overlapping sub problems. Since, for each choice of i and S, computing $g(i,S)$ requires $g(i, S\backslash\{j\})$, $j \in S$. A recursive algorithm may encounter each sub problem many times in different branches of its recursion tree. This property of overlapping sub problems is the reason for applying dynamic programming.

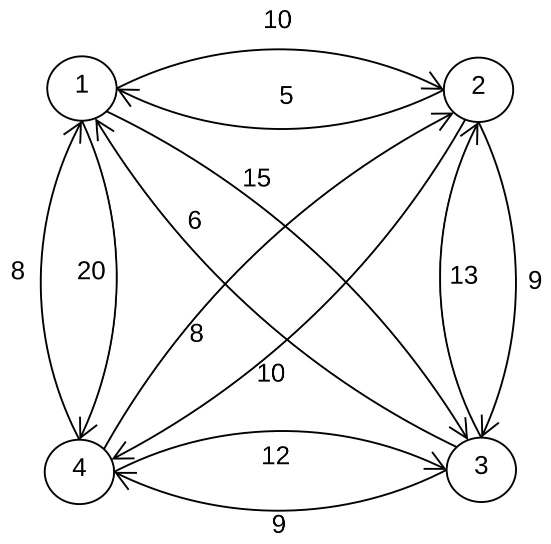
The algorithm: **TSP**(n, C)

Given a directed graph with n vertices, directed edges between every two vertices and costs for each edge C_{ij}. Let S denotes the set of all subsets of vertices $\{2, 3, \ldots, n\}$. This algorithm computes $g(i,S)$ the shortest path starting at vertex i, going through all vertices in S and terminates at vertex i. It also computes $J(i,S)$, which gives the next vertex in the shortest path from i .

```
1.for $i = 1$ to $n$ do
2.    $g(i,\emptyset) = C_{i1}$; $J(i,\emptyset) = 0$;
3.for $m = 1$ to $n-1$ do
4.    for $i = 2$ to $n$ do
5.        Choose all $j \in S$ such that $i \neq j$ and $| S\backslash\{j\} | = m$, from all such $j$, find a $j$ that minimizes $C_{ij} + g(i, S\backslash\{j\})$
6.        Compute $g(i,S) = C_{ij} + g(j, S\backslash\{j\})$
8.        $J(i,S) = j$
9.Find a $k$, $2 \leq k \leq n$, that minimizes $C_{1k} + g(k, V\backslash\{1,k\})$. i.e., find $g(1, V\backslash\{1\})$.
10.$J(1, V\backslash\{1\}) = k$
```

5.1 Trace of the algorithm

Consider the following directed graph:



The corresponding cost matrix is $C_{ij} = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Now, let us trace the algorithm for the above graph:

Iteration 1: $\mid S \mid= 0$
Now, $S = \emptyset$:
$g(1, \emptyset) = C_{11} = 0$
$g(2, \emptyset) = C_{21} = 5$
$g(3, \emptyset) = C_{31} = 6$
$g(4, \emptyset) = C_{41} = 8$

Iteration 2: $\mid S \mid= 1$
Now, increase the size of S by one.
$g(2, \{3\}) = C_{23} + g(3, \emptyset) = 9 + 6 = 15;\ J(2, \{3\}) = 3$
$g(2, \{4\}) = C_{24} + g(4, \emptyset) = 10 + 8 = 18;\ J(2, \{4\}) = 4$
$g(3, \{2\}) = C_{32} + g(2, \emptyset) = 13 + 5 = 18;\ J(3, \{2\}) = 2$
$g(3, \{4\}) = C_{34} + g(4, \emptyset) = 12 + 8 = 20;\ J(3, \{4\}) = 4$
$g(4, \{2\}) = C_{42} + g(2, \emptyset) = 8 + 5 = 13;\ J(4, \{2\}) = 2$
$g(4, \{3\}) = C_{43} + g(3, \emptyset) = 9 + 6 = 15;\ J(4, \{3\}) = 3$

Iteration 3: $\mid S \mid= 2$
Now, the size of S is two.
$g(2, \{3, 4\}) = \min\{C_{23} + g(3, \{4\}), C_{24} + g(4, \{3\})\} = \min\{9 + 20, 10 + 15\} = 25;\ J(2, \{3, 4\}) = 4$
$g(3, \{2, 4\}) = \min\{C_{32} + g(2, \{4\}), C_{34} + g(4, \{2\})\} = \min\{13 + 18, 12 + 13\} = 25;\ J(3, \{2, 4\}) = 4$
$g(4, \{2, 3\}) = \min\{C_{42} + g(2, \{3\}), C_{43} + g(3, \{2\})\} = \min\{8 + 15, 9 + 18\} = 23;\ J(4, \{2, 3\}) = 2$

Iteration 4: $\mid S \mid= 3$
Now, the size of S is three.
$g(1, \{2, 3, 4\}) = \min\{C_{12}+g(2, \{3, 4\}), C_{13}+g(3, \{2, 4\}), C_{14}+g(4, \{1, 3\})\} = \min\{10+25, 15+25, 20+23\} = 35;\ J(1, \{2, 3, 4\}) = 2$

Constructing an optimal solution:
The computation of $g(1, \{2, 3, 4\})$ says that we have to reach the vertex 2 from 1 (since the value of $J(1, \{2, 3, 4\})$ is 2). The computation of $g(2, \{3, 4\})$ says that we have to reach the vertex 4 from 2 (since the value of $J(2, \{3, 4\})$ is 4). Similarly, the computation of $g(4, \{3\})$ says that we have to reach the vertex 3 from 4 (since the value of $J(4, \{3\})$ is 3). Thus, the required tour with minimum cost is $1 \to 2 \to 4 \to 3 \to 1$.

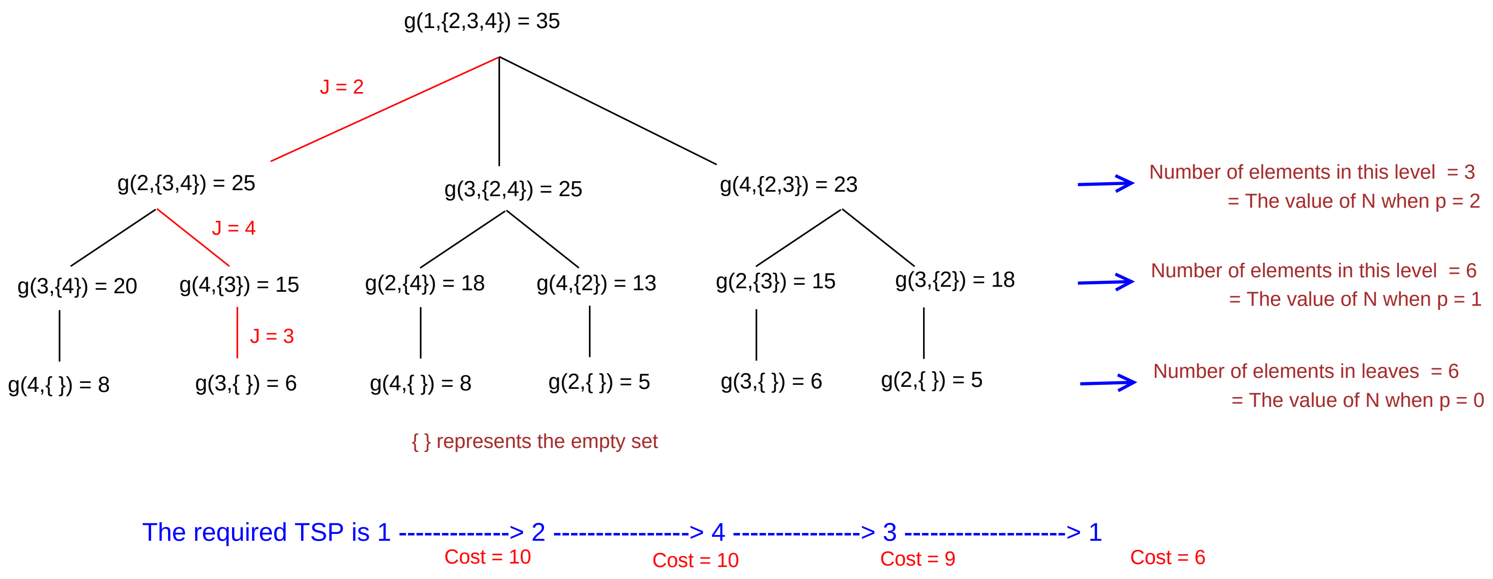
Run time analysis
Let N denotes the number of $g(i, S)$ that have to be computed before $g(1, V \backslash \{1\})$. The number of distinct sets S of size p not including 1 and i is $\left(\begin{array}{c} n-2 \\ p \end{array} \right)$. For each value of S, there are $(n - 1)$ choices for i.

Therefore, $N = \sum_{p=0}^{n-2} (n-1) \left(\begin{array}{c} n-2 \\ p \end{array} \right) = (n-1) \left[\left(\begin{array}{c} n-2 \\ 0 \end{array} \right) + \left(\begin{array}{c} n-2 \\ 1 \end{array} \right) + \ldots + \left(\begin{array}{c} n-2 \\ n-3 \end{array} \right) + \left(\begin{array}{c} n-2 \\ n-2 \end{array} \right) \right]$
$= (n - 1)2^{n-2}$

To compute $g(i, S)$, for each choice of k, we make $(k - 1)$ comparisons to identify the minimum j at each $g(i, S)$. i.e., $O(k)$ comparisons. So, the total time complexity is:

$$\left[(n - 1) \cdot 2^{n-2} + 1\right] \cdot O(k) \text{ (1 denotes the effort spend in level one)}$$
$$= (n - 1) \cdot 2^{n-2} \cdot O(n) = O(2^n \cdot n^2).$$

Note that, the trivial algorithm for this problem takes $O(n!)$ (Since, the number of different TSP tours = number of branches = $O((n - 1)!)$).



g(1,{2,3,4}) = 35
J = 2
g(2,{3,4}) = 25
g(3,{2,4}) = 25
g(4,{2,3}) = 23
Number of elements in this level = 3
= The value of N when p = 2
J = 4
g(3,{4}) = 20
g(4,{3}) = 15
g(2,{4}) = 18
g(4,{2}) = 13
g(2,{3}) = 15
g(3,{2}) = 18
Number of elements in this level = 6
= The value of N when p = 1
J = 3
g(4,{ }) = 8
g(3,{ }) = 6
g(4,{ }) = 8
g(2,{ }) = 5
g(3,{ }) = 6
g(2,{ }) = 5
Number of elements in leaves = 6
= The value of N when p = 0
{ } represents the empty set
The required TSP is 1 -------------> 2 ----------------> 4 ---------------> 3 -------------------> 1
Cost = 10
Cost = 10
Cost = 9
Cost = 6

Figure 4: A tree representation for the example given in trace, with the values of N at each level.

6 Conclusion

In this lecture, we have witnessed the power of dynamic programming through five classical problems. Due to the presence of optimal substructure and overlapping subproblem properties, a considerable improvement in the run time for all the case studies discussed which we summarize below by comparing with its trivial algorithm.

Case Study	**Run time of trivial algorithm**	**Run time of dynamic programming**
Assembly line scheduling	$O(2^n)$	$O(n)$
OPT Matrix Chain Multiplication	$O(4^n)$	$O(n^3)$
Knapsack problem	$O(2^n)$	$O(nW)$
Optimal BST	$O(4^n)$	$O(n^3)$
TSP	$O(n^n)$	$O(n^2 \cdot 2^n)$

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thank all of them.

References

[1] H.Cormen, T., C. E.Leiserson, R. L.Rivest and C. Stein: Introduction to Algorithms 3rd Edition. McGraw-Hill Higher Education, (2001).

[2] E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.

[3] Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.