Presentation Topic

Design and Analysis of Algorithms

Department of Computer Engineering

BRACT'S, Vishwakarma Institute of Information Technology, Pune-48

(An Autonomous Institute affiliated to Savitribai Phule Pune University)

(

Objective/s of this session

- To study the **analysis** of algorithms
- To study the **greedy and dynamic programming** algorithmic strategies
- To study the **backtracking and branch and bound algorithmic** strategies
- To study the concept of **hard problems** through understanding of **intractability and NP-Completeness**
- To study some **advance techniques to solve intractable problems**
- To study **multithreaded** and **distributed algorithms**

Learning Outcome/Course Outcome

- Analyze algorithms for their time and space complexities in terms of asymptotic performance.
- Apply greedy and dynamic programming algorithmic strategies to solve a given problem
- Apply backtracking and branch and bound algorithmic strategies to solve a given problem
- Identify intractable problems using concept of NP-Completeness
- Use advance algorithms to solve intractable problems
- Solve problems in parallel and distributed scenarios

Contents

- **Greedy Strategy**
  - Control Abstraction (C.A.) and time analysis of C.A.
  - Knapsack problem
  - Job Sequencing with deadlines
  - Huffman coding
- **Dynamic Programming**:
  - Principle of Optimality
  - General Strategy
  - 0/1 Knapsack
  - Optimal Binary Search Tree
  - Multistage graphs

GREEDY STRATEGY

Greedy Strategy

- Used to solve an optimization problem.
- An Optimization problem is one in which the aim is to either **maximize** or **minimize** a given **objective function** w. r. t. some **constraints or conditions,** given a set of input values
- **Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function.**
- **It makes a locally optimal choice in the hope that this choice will lead to an overall globally optimal solution.**
- **The greedy algorithm does not always guarantee the optimal solution but it generally produces solutions that are very close in value to the optimal, (**for the selected problems it gives **optimal solution)**

Knapsack Problem

- *Given a set of items, each with a* ***weight*** *and a* ***value****, determine which items to include in the collection (in Knapsack) so that the total weight is* ***less than or equal to*** *a given limit and the* ***total value is as large as possible****.*

Knapsack Problem

Given *n objects and a* knapsack.
Object *i* *has a* ***weight*** $\mathbf{w_i}$*; and the knapsack has a* ***capacity M****.*
If a fraction *x;, 0 <=* x_i *<= 1, of object i is placed into the knapsack then a* profit of P_iX_i *is earned.*
The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
Since the knapsack capacity is *M,* we require the total weight of all chosen objects to be at most *M.*

Formally, problem may be stated as:

- maximize $\sum_{1<=i<=n} P_iX_i$ (1)

  subject to

  $\sum_{1\le i\le n} w_iX_i <= M$ *(2)*

  $0\le x_i\le 1$, $p_i \ge 0$, $w_i\ge 0$ ………(3)

- The profits and weights are positive numbers.

Knapsack problem

Consider the following instance of the knapsack problem: $n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
i)	(1/2, 1/3, 1/4)	16.5	24.25
ii)	(1, 2/15, 0)	20	28.2
iii)	(0, 2/3, 1)	20	31
iv)	(0, 1, 1/2)	20	31.5

Greedy Method example

No. of objects = 3 , M =20

P1 ,P2, P3 = (25,24,15) and W1, W2, W3 = (18, 15 ,10)

As per the algorithm take p_i/w_i ratio (i = 1 to n)

P1/W1 = 25/18 = **1.3**, P2/W2 = 24/15 = **1.6**, P3/W3 = 15/10 = **1.5**

Arrange p_i/w_i ratio in **descending order** :

P2/W2 = **1.6** , P3/W3 = **1.5**, P1/W1 = **1.3**

Then select the objects, provided object weight <= M

here W2 = 15 < 20 ; Profit = 24 W =15

Remaining capacity is 5

Take 5 units of third object i.e. 10/2 = 5 , Therefore W = 15 + 5 = **20** = M

Take same fraction of profit of third object profit = ½ * 15 = 7.5

Therefore Profit is 24 + 7.5 = **31.5**

Thus as capacity is full no more object can be accommodated .

Hence Answer = Total capacity = 20 Profit =31.5

Objects fraction selected = (0 ,1, 0.5)

Q)

- If an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points.
- However, on an aptitude tests with different point values, and test taker needs to answer as many as possible, it is more difficult to make choices.
  - Ex. In a test of 12 questions and 125 possible points , and Max 60 minutes

Ques	Q1)	Q2)	Q3)	Q4)	Q5)	Q6)	Q7)	Q8)	Q9)	Q10)	Q11)	Q12)
Marks	5	5	6	7	7	8	9	10	10	14	20	24
Specific Student Time	*3*	*5*	*2*	*8*	*12*	*4*	*10*	*12*	*16*	*17*	*16*	*25*

Soln

Ques	Q1)	Q2)	Q3)	Q4)	Q5)	Q6)	Q7)	Q8)	Q9)	Q10)	Q11)	Q12)
Marks	5	5	6	7	7	8	9	10	10	14	20	24
Specific Student Time	*3*	*5*	*2*	*8*	*12*	*4*	*10*	*12*	*16*	*17*	*16*	*25*
	1.66	*1*	*3*	*.875*	*.583*	*2*	*.9*	*.833*	*.625*	*.8235*	*1.25*	*.96*

Solve Examples using greedy method

consider the following instances of knapsack problem, find optimal solution.

1. No. of objects = 7, m = 15
(p1, p2, p3, p4, p5, p6, p7) = (10, 5, 15, 7, 6, 18, 3)
(w1, w2, w3, w4, w5, w6, w7) = (2, 3, 5, 7, 1, 4, 1)

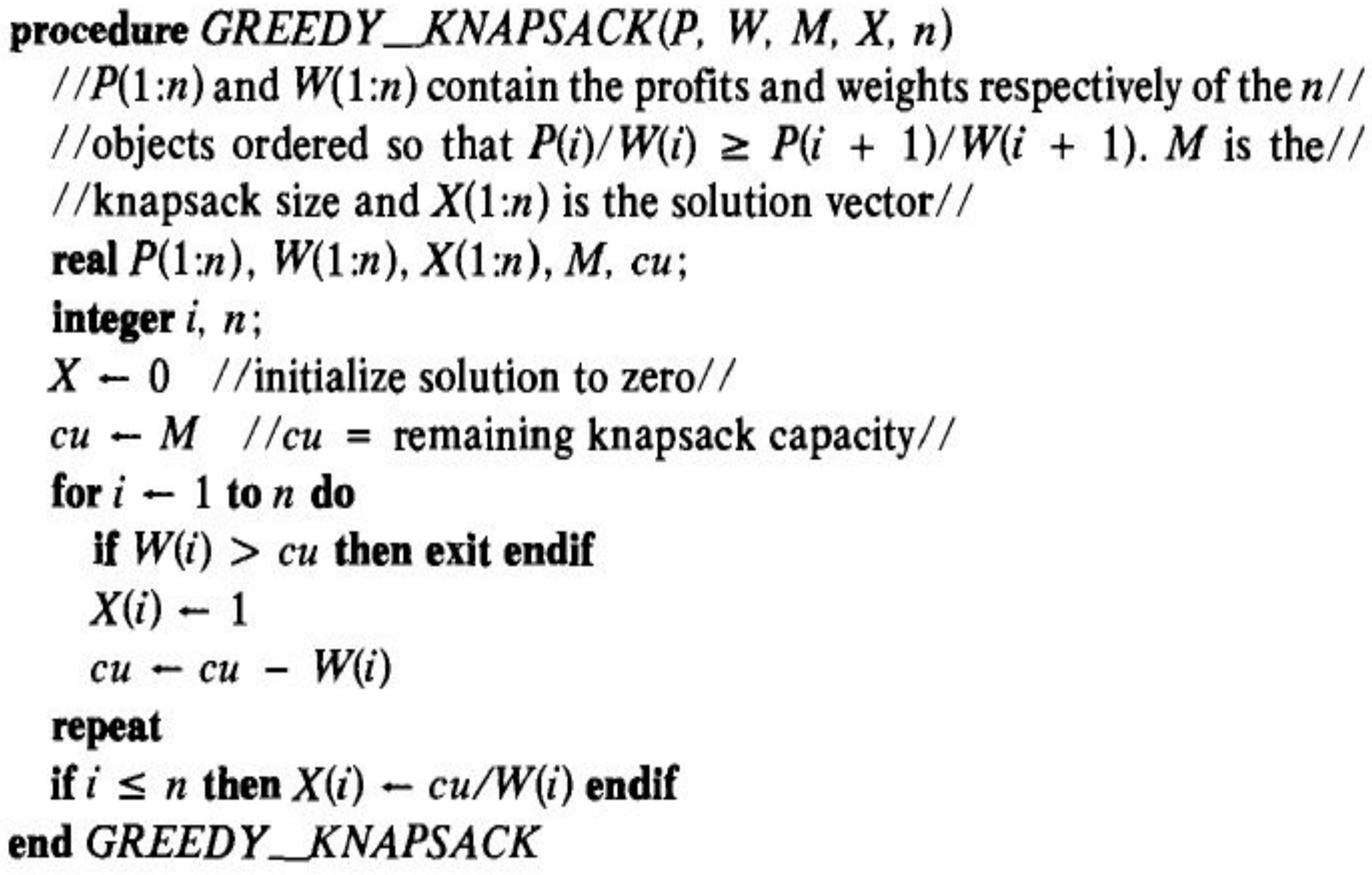
2. No. of objects =5, m=100,
(p1, p2, p3, p4, p5) = (20, 30, 66, 40, 60)
(w1, w2, w3, w4, w5) = (10, 20, 30, 40, 50)

Submit your answers at following link :

Real life applications of knapsack problems (*Multiple Knapsack problem, Stochastic Knapsack problem, 0/1 Knapsack, Bounded Knapsack problem etc*)

- Financial modeling
- Production and inventory management systems
- Stratified sampling
- Design of queuing network models in manufacturing, and control of traffic overload in telecommunication systems
- Other areas of applications
  - Yield management for airlines
  - Hotels and rental agencies
  - College admissions
  - Quality adaptation and admission control for interactive multimedia systems
  - Cargo loading, capital budgeting, cutting stock problems
  - Computer processing allocations in huge distributed systems

Greedy Algorithm for Knapsack Problem



```
procedure GREEDY_KNAPSACK(P, W, M, X, n)
   //P(1:n) and W(1:n) contain the profits and weights respectively of the n//
   //objects ordered so that $P(i)/W(i) \geq P(i + 1)/W(i + 1)$. M is the//
   //knapsack size and X(1:n) is the solution vector//
   real P(1:n), W(1:n), X(1:n), M, cu;
   integer i, n;
   $X \leftarrow 0$   //initialize solution to zero//
   $cu \leftarrow M$   //cu = remaining knapsack capacity//
   for $i \leftarrow 1$ to n do
      if $W(i) > cu$ then exit endif
      $X(i) \leftarrow 1$
      $cu \leftarrow cu - W(i)$
   repeat
   if $i \leq n$ then $X(i) \leftarrow cu/W(i)$ endif
end GREEDY_KNAPSACK
```

Analysis of Algorithm knapsack

1. To sort the objects in non-increasing order of p_i / w_i, computation time = O(nlogn)
2. While loop will be executed 'n' times in worst case, so the computation time = O(n)
3. **Time complexity = O(nlogn)**
4. **Space complexity =** c + space required for x[1:n] = **O(n)**

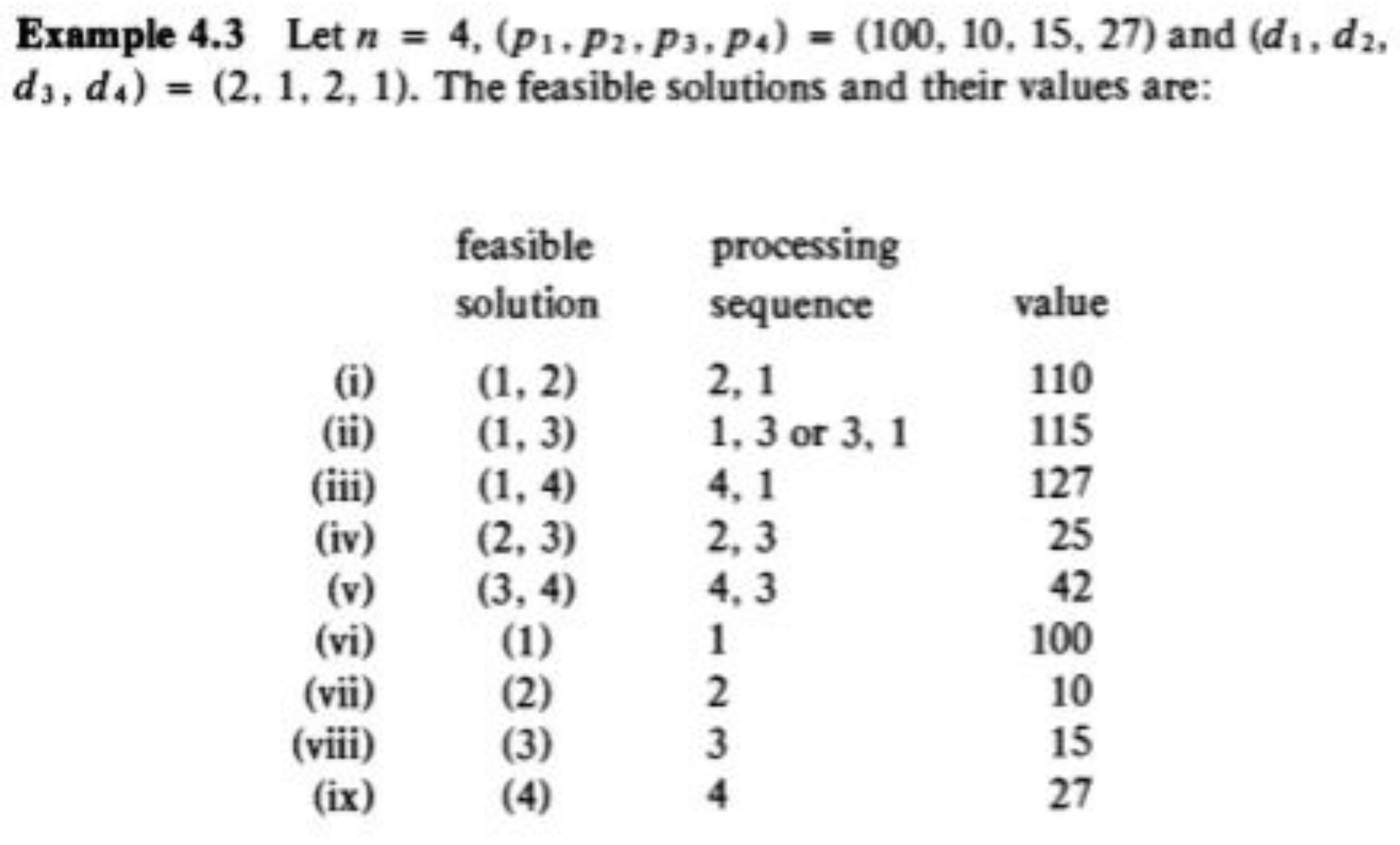
Scheduling Algorithms : Job scheduling

Scheduling Algorithms : Job scheduling

Given a set of *n jobs.*

- *d_i is an integer deadline associated with job i*
- *d_i >= 0 and a profit p_i >= 0.*
- *For any job i the profit p_i is earned iff the job* is completed by its deadline.
- In order to complete a job one has to process the job on a machine for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset, *J, of jobs* such that each job in this subset can be completed by its deadline.
- The value of a feasible solution *J is the sum of the profits of the jobs in J or $\sum_{i \in j} p_i$.*
- *An optimal solution is **a feasible solution** with maximum value.*

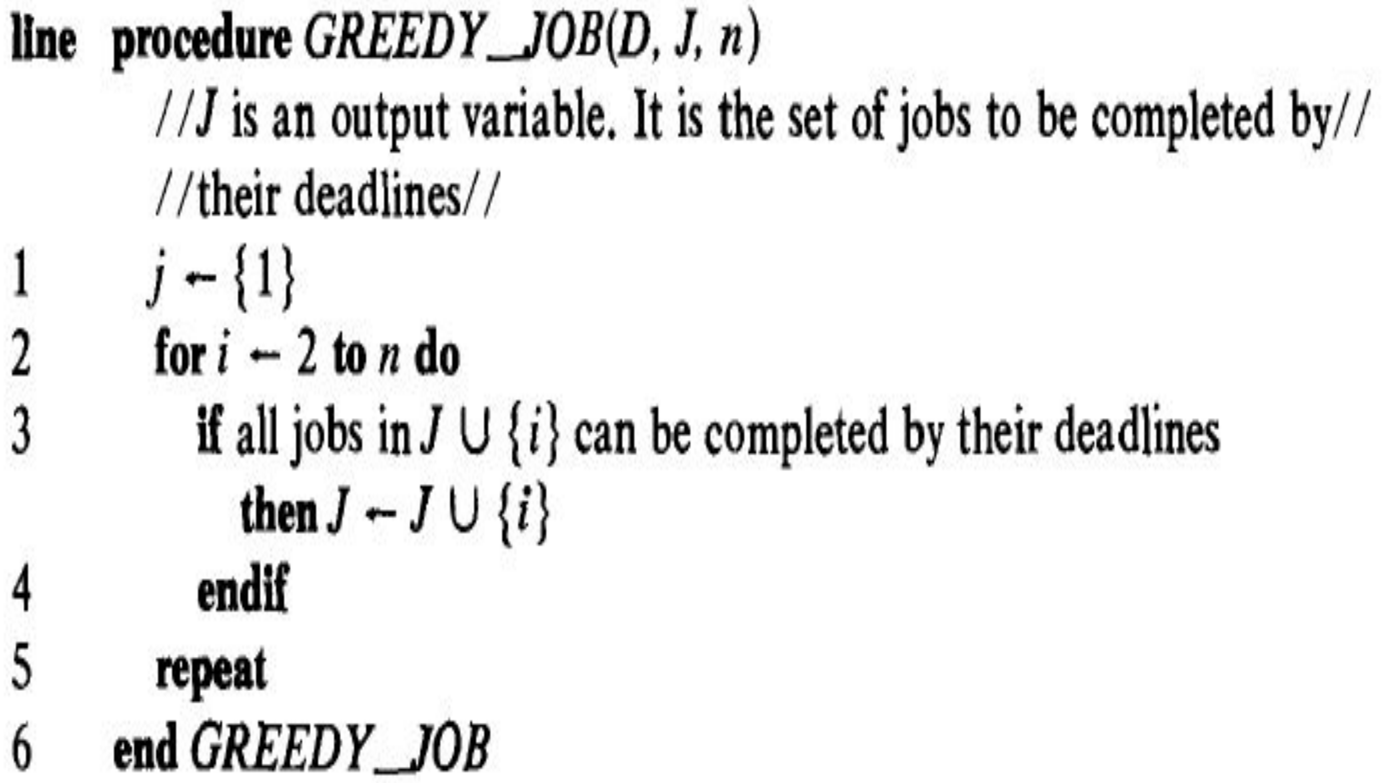
Example of Job Sequencing



Example 4.3 Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

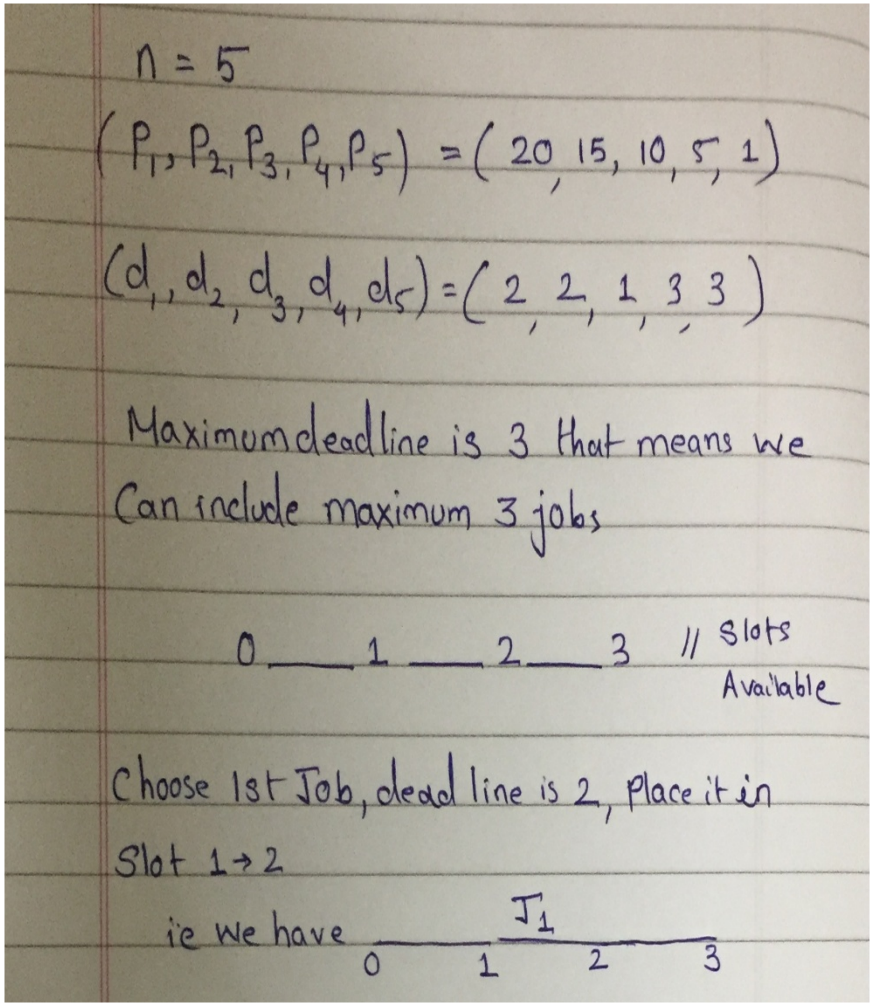
	feasible solution	processing sequence	value
(i)	(1, 2)	2, 1	110
(ii)	(1, 3)	1, 3 or 3, 1	115
(iii)	(1, 4)	4, 1	127
(iv)	(2, 3)	2, 3	25
(v)	(3, 4)	4, 3	42
(vi)	(1)	1	100
(vii)	(2)	2	10
(viii)	(3)	3	15
(ix)	(4)	4	27

Greedy Job sequencing



```
line  procedure GREEDY_JOB(D, J, n)
        //J is an output variable. It is the set of jobs to be completed by//
        //their deadlines//
1       $j \leftarrow \{1\}$
2       for $i \leftarrow 2$ to $n$ do
3         if all jobs in $J \cup \{i\}$ can be completed by their deadlines
            then $J \leftarrow J \cup \{i\}$
4         endif
5       repeat
6     end GREEDY_JOB
```

Job Sequencing Example (Not efficient for large number of jobs/ deadlines)



$n = 5$

$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

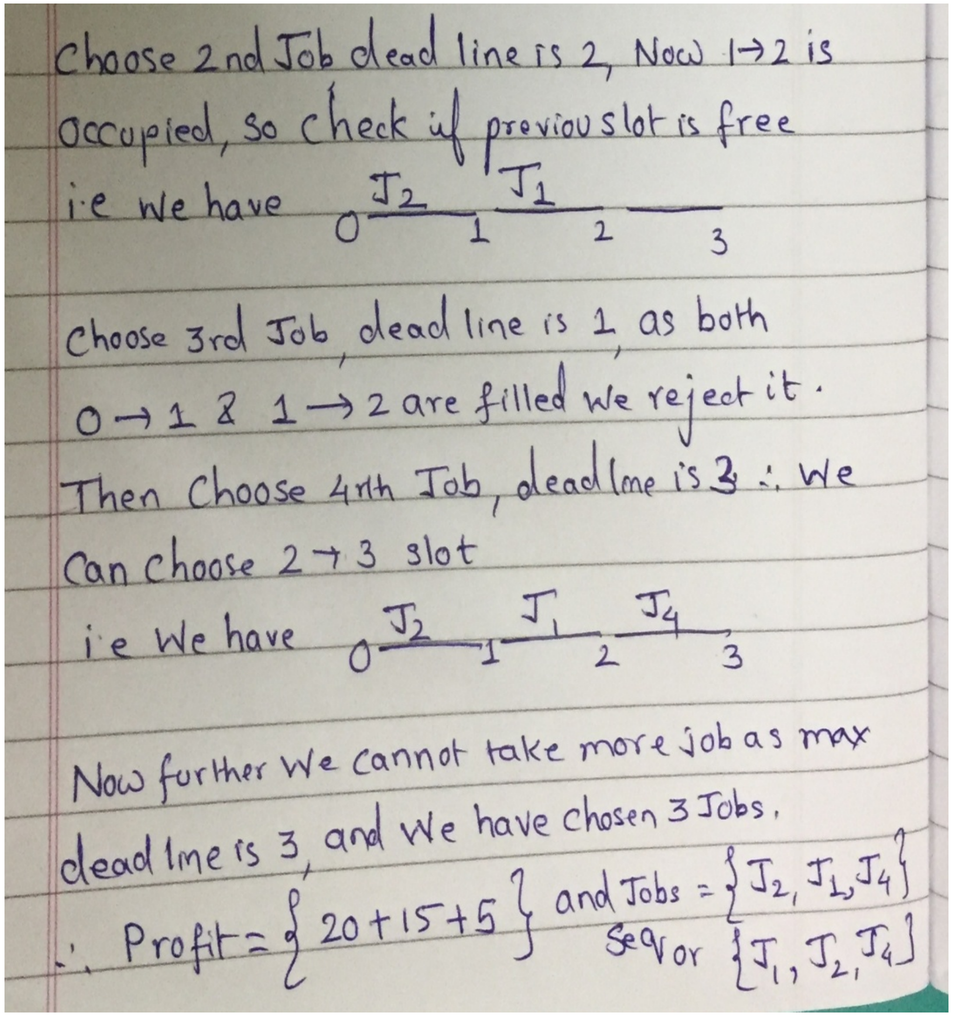
$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

Maximum deadline is 3 that means we Can include maximum 3 jobs

0 ___ 1 ___ 2 ___ 3 // Slots Available

Choose 1st Job, dead line is 2, Place it in Slot 1→2

i.e we have 0 ___ 1 $\underline{J_1}$ 2 ___ 3



Choose 2nd Job dead line is 2, Now 1→2 is occupied, so check if previou slot is free

i.e we have 0 $\underline{J_2}$ 1 $\underline{J_1}$ 2 ___ 3

Choose 3rd Job, dead line is 1, as both 0→1 & 1→2 are filled we reject it.

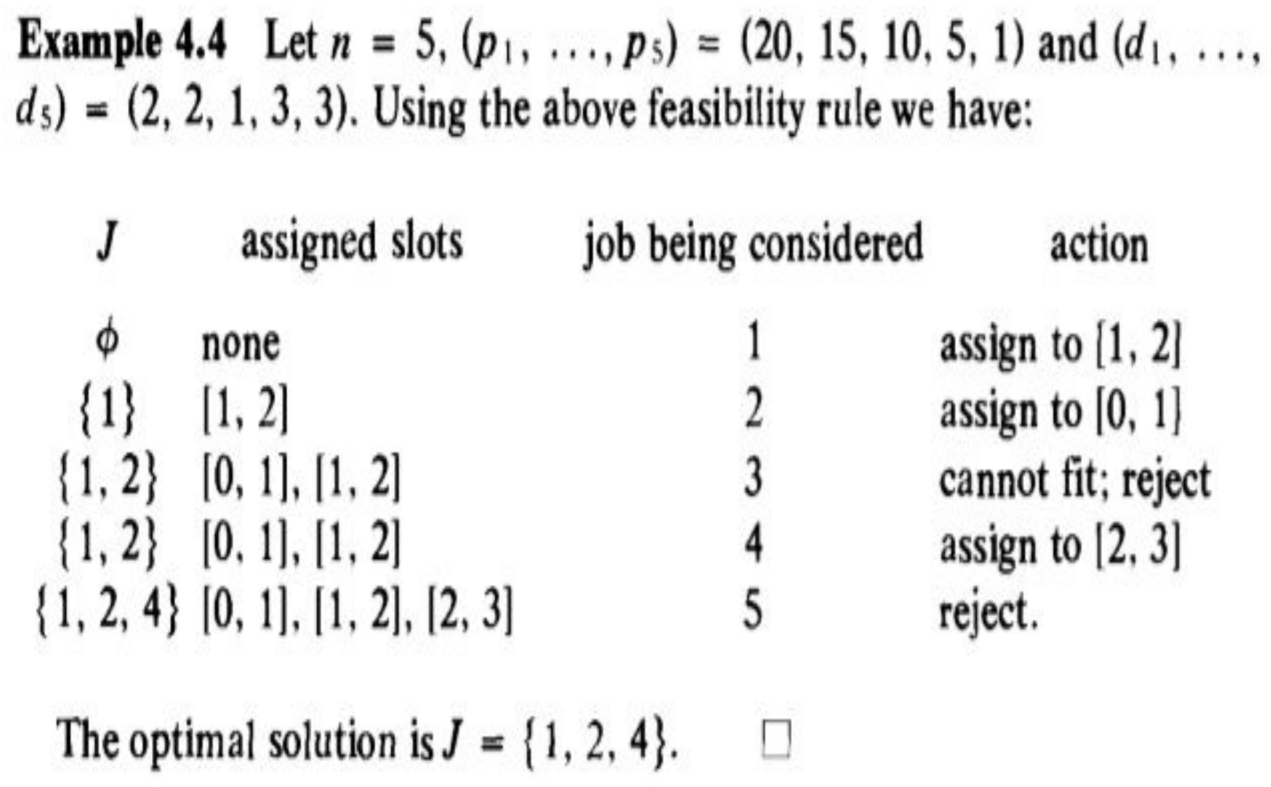
Then Choose 4rth Job, dead line is 3 ∴ we Can choose 2→3 slot

i.e We have 0 $\underline{J_2}$ 1 $\underline{J_1}$ 2 $\underline{J_4}$ 3

Now further we cannot take more job as max dead line is 3, and we have chosen 3 Jobs.

∴ Profit = $\{20 + 15 + 5\}$ and Jobs = $\{J_2, J_1, J_4\}$ Seq or $\{J_1, J_2, J_4\}$

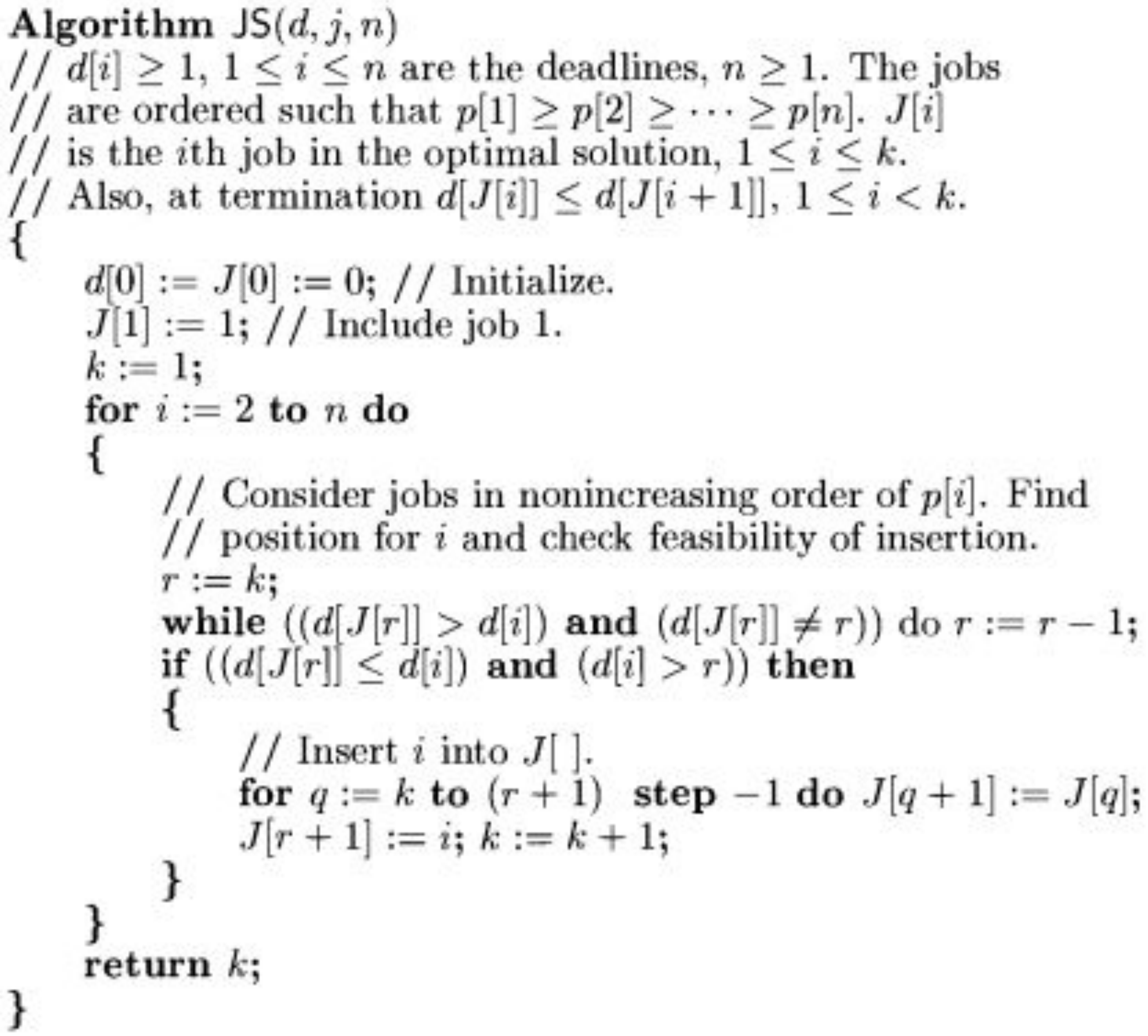
Solved Example



Example 4.4 Let $n = 5$, $(p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule we have:

J	assigned slots	job being considered	action
ϕ	none	1	assign to [1, 2]
{1}	[1, 2]	2	assign to [0, 1]
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject.

The optimal solution is $J = \{1, 2, 4\}$. □



```
Algorithm JS(d, j, n)
// $d[i] \geq 1$, $1 \leq i \leq n$ are the deadlines, $n \geq 1$. The jobs
// are ordered such that $p[1] \geq p[2] \geq \cdots \geq p[n]$. $J[i]$
// is the $i$th job in the optimal solution, $1 \leq i \leq k$.
// Also, at termination $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$.
{
    $d[0] := J[0] := 0$; // Initialize.
    $J[1] := 1$; // Include job 1.
    $k := 1$;
    for $i := 2$ to $n$ do
    {
        // Consider jobs in nonincreasing order of $p[i]$. Find
        // position for $i$ and check feasibility of insertion.
        $r := k$;
        while (($d[J[r]] > d[i]$) and ($d[J[r]] \neq r$)) do $r := r - 1$;
        if (($d[J[r]] \leq d[i]$) and ($d[i] > r$)) then
        {
            // Insert $i$ into $J[\,]$.
            for $q := k$ to $(r + 1)$ step $-1$ do $J[q + 1] := J[q]$;
            $J[r + 1] := i$; $k := k + 1$;
        }
    }
    return $k$;
}
```

Solve the given examples

Example 1 :

Let n = 5 , (p1, ….. P5) = (60,100, 20, 40, 20)

(d1,….d5) =(2, 1, 3, 2, 1)

Example 2 :

Let n= 7, (p1, ……p7) = (35,30,25,20,15,12,5)

(d1,……d7) = (3,4,4,2,3,1,2)

Example 3:

Let n= 6, (p1, ……p6) = (50,45,40,30,25,10)(d1,……d6) = (4,1,3,1,2,2)

Solution

1. Let n = 5 , (p1, ….. P5) = (60,100, 20, 40, 20)
(d1,….d5) =(2, 1, 3, 2, 1)
Answer : { J1 , J2, J3 } Profit : 180

2. Let n= 7, (p1, ……p7) = (35,30,25,20,15,12,5)
(d1,……d7) = (3,4,4,2,3,1,2)
Answer : { J4 , J3, J1 , J2 } Profit :110

3. Let n= 6, (p1, ……p6) = (50,45,40,30,25,10)
(d1,……d6) = (4,1,3,1,2,2)
Answer : {J2, J5, J3, J1 } , Profit : 160

Word problems

(1) An appliance repair situation: Let n = 5 , (p1, ….. P5) = (60,100, 20, 40, 20)

(d1,….d5) =(2, 1, 3, 2, 1)

Select which appliance to repair

(2) A tailor shop situation : Let n= 7, (p1, ……p7) = (35,30,25,20,15,12,5)

(d1,……d7) = (3,4,4,2,3,1,2)

Select which appliance to repair

Huffman Coding

Huffman Coding

- Proposed by Dr. David A. Huffman in 1952
  - *"A Method for the Construction of Minimum Redundancy Codes"*
- Applicable to many forms of data transmission
  - Our example: text files

Huffman Coding

- Huffman coding is a form of **statistical coding**
- Not all characters occur with the same frequency!
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, be it e or X

Huffman Coding

- Any savings in **tailoring codes** to **frequency** of character?
- Code word lengths are no longer fixed like ASCII.
- Code word lengths vary and will be shorter for the more frequently used characters.

Basic Algorithm: Using Huffman Coding

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Fixed Length VS Variable length encoding

Fixed Length Coding

- Consider the following text:

  ABBCDBCCDAABBEEEBEAB – 20 characters

If the above string has to be sent over network then each character will be coded in ASCII form.

Thus ASCII code for A is 65, B is 66, C is 67 and so on..

While transmission ASCII code is converted to binary form each ASCII code has 8 bits of coding for example

A = 65 =01000001 , B = 66 = 01000010, and so on…

Thus to transfer the above message it will require 20 * 8 = 160 bits

This is called Fixed length encoding as each character requires 8 bits i. e we do not apply any compression technique.

Fixed Length Coding Continued..

Consider the following text:

ABBCDBCCDAABBEEEBEAB – 20 characters

Now we know that the characters (A B C D E) are repeated in the message. If there are 4 characters we can represent them with fixed 2 bits as $2^2 = 4$ codes.

But we have 5 different characters , thus we will require 3 bit code to represent them as $2^3 = 8$.

Thus we can have code as below:

A - 000 B – 001 C – 010 D – 011 E – 100

So now how many bits will be required 20 * 3 = **60 bits**

But now along with this 60 bits of encoding the sender will also have to send characters and their codes for receiver to decode the message.

Therefore 8 bits ASCII code for 5 characters = 40

+ Total No. of bits required for encoding (5 * 3) =15

Total = 40 + 15 = **55 bits**

Hence the total number of bits that will be sent is 60 + 55 = **115 bits**

Huffman Coding – Variable Length Coding

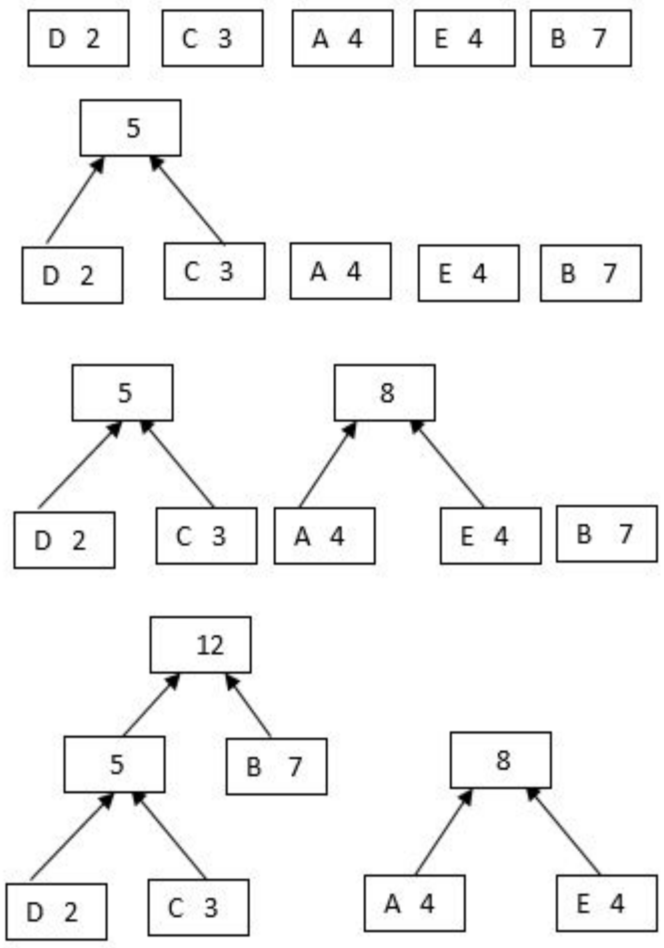
Consider the following text:

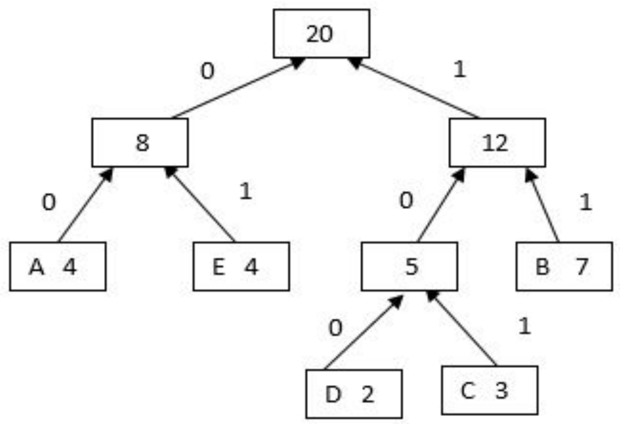
ABBCDBCCDAABBEEEBEAB – 20 characters

Character	Frequency
A	4
B	7
C	3
D	2
E	4

Arrange the characters in increasing order of frequency and generate Huffman tree.

Huffman Tree





Character	Frequency	Code
A	4	0 0
B	7	1 1
C	3	1 0 1
D	2	1 0 0
E	4	0 1

Huffman Code – Variable Length Code

Character	Frequency	Code
A	4	0 0
B	7	1 1
C	3	1 0 1
D	2	1 0 0
E	4	0 1

We can observe every character has variable length code. So now how will we calculate number of bits that will be transferred as follows :

Length of message = freq of char * no of bits

Length = 4*2 + 7*2 + 3*3 + 2*3 + 4*2 =
(8 + 14 + 9 + 6 + 8) = 45 bits

For decoding the message at receiver end we will also have to send char and code table which requires following no. of bits :

No. of characters * 8 bits (ASCII code) + Total no. of code bits

5 * 8 + (2 + 2 + 3 + 3 + 2) = 40 + 12 = 52 bits

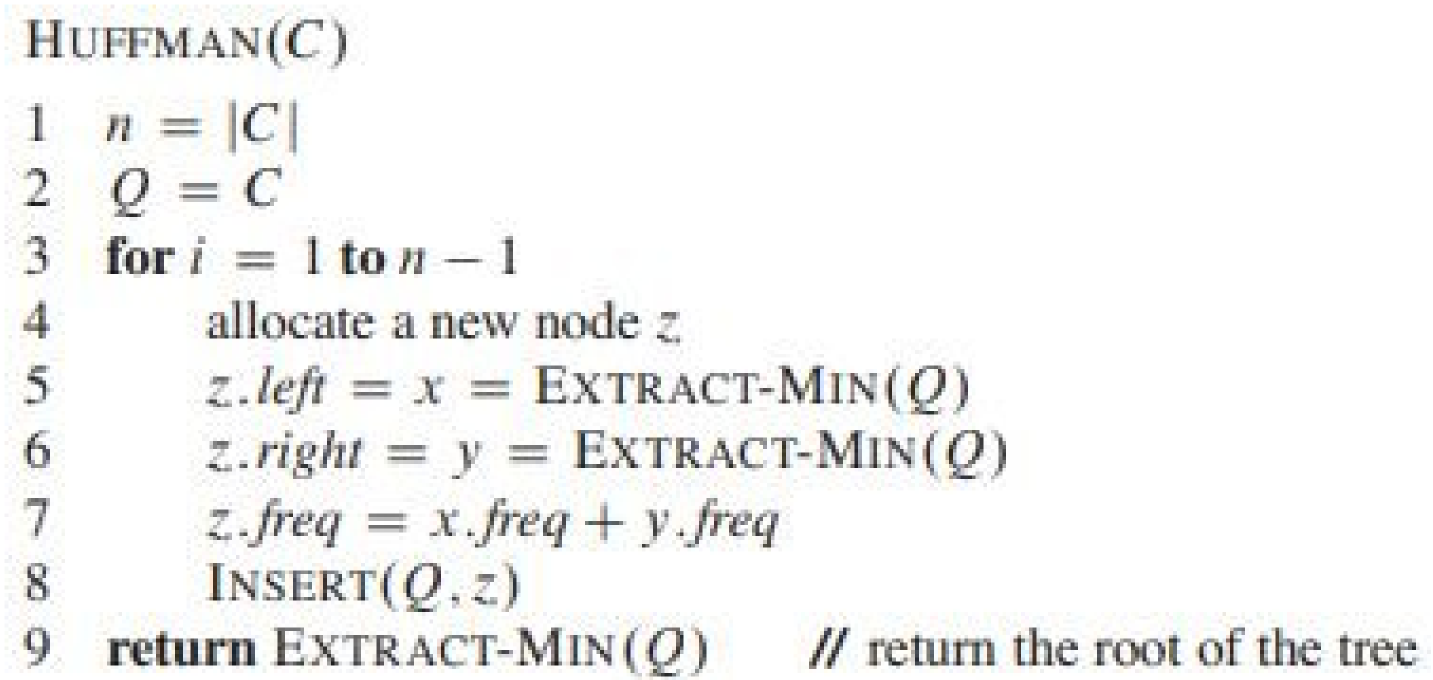
Therefore total number of bits transferred are :
45 + 52 = 97 bits

Thus using variable length code the number of bits that should be transferred while sending the message are minimized

Prefix codes

- Prefix codes – Are codes in which no codeword is also a prefix of some other codeword
- Prefix code can always achieve the optimal data compression among any character code (can be proved)

C is a set of n characters and that each character c belonging to C is an object with an attribute c.freq giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| - 1 “merging” operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the freq attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

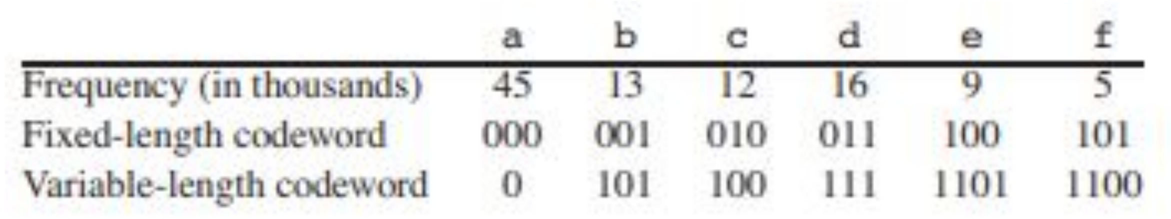


```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Summary

- Huffman coding is a technique used to compress files for transmission
- Uses statistical coding
  - more frequently used symbols have shorter code words
- Encoding satisfies Prefix Code
- Time Complexity for Huffman coding algorithm is O(nlogn).

Q)



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Fixed length requires - > 100,000 *3 = 300,000 bits

(45 * 1 + 13 * 3 + 12 * 3 + 16 * 3 + 9 * 4 + 5 * 4)* 1,000 = 224,000 bits

Solve Example

Message : ABABCABCDABFFFDEAABBCCDDEEA

1. Huffman code of each character.
2. No bits that will be required for transferring the message.
3. Total no. of bits for transferring message along with table.

Submit your answers at following link :

Dynamic Programming

Recap

1. Greedy Method
2. Control Abstraction of Greedy Method
3. Examples in greedy Method

Objective/s of this session

- To study the **analysis** of algorithms
- <u>To study the **greedy and dynamic programming** algorithmic strategies</u>
- To study the **backtracking and branch and bound algorithmic** strategies
- To study the concept of **hard problems** through understanding of **intractability and NP-Completeness**
- To study some **advance techniques to solve intractable problems**
- To study **multithreaded** and **distributed algorithms**

Learning Outcome/Course Outcome

After completion of the course, student will be able to

- Analyze algorithms for their time and space complexities in terms of asymptotic performance.
- <u>Apply greedy and dynamic programming algorithmic strategies to solve a given problem</u>
- Apply backtracking and branch and bound algorithmic strategies to solve a given problem
- Identify intractable problems using concept of NP-Completeness
- Use advance algorithms to solve intractable problems
- Solve problems in parallel and distributed scenarios

Unit II	Greedy Method & Dynamic Programming

Greedy Method: General strategy,
the principle of optimality,
Knapsack problem, Job Sequencing with Deadlines,
Huffman coding.

Dynamic Programming: General Strategy, 0/1 Knapsack, OBST, multistage graphs

Unit II : Dynamic Programming - Syllabus

- Introduction to Dynamic Programming
- General Strategy – Principle of Optimality
- Implementation of 0/1 Knapsack Problem
- Implementation of Optimal Binary Search Tree (OBST)
- Multistage Graphs

Unit II : Dynamic Programming

The Principle of Optimality :

- The Principle of Optimality states that in an optimal sequence of decisions or choices, each sub-sequence must also be optimal.
- The Principle of Optimality 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 decisions.

Properties of Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the **two main properties of a problem** that suggests that the given problem can be solved using Dynamic :

1) Overlapping Sub problems
2) Optimal Substructure

1. Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

2. Optimal Substructure:

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example, the Shortest Path problem has following optimal substructure property:
If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.

Properties of Dynamic Programming

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \        /    \       /    \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

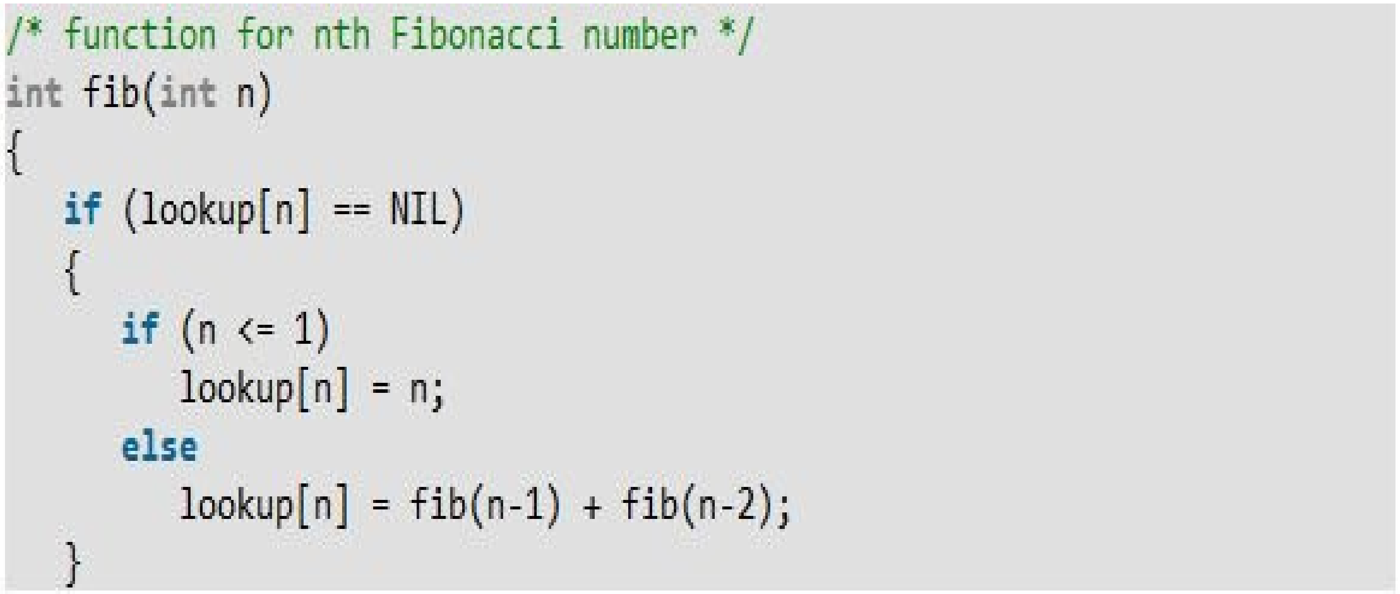
We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

a) Memoization (Top Down)

b) Tabulation (Bottom Up)

Properties of Dynamic Programming

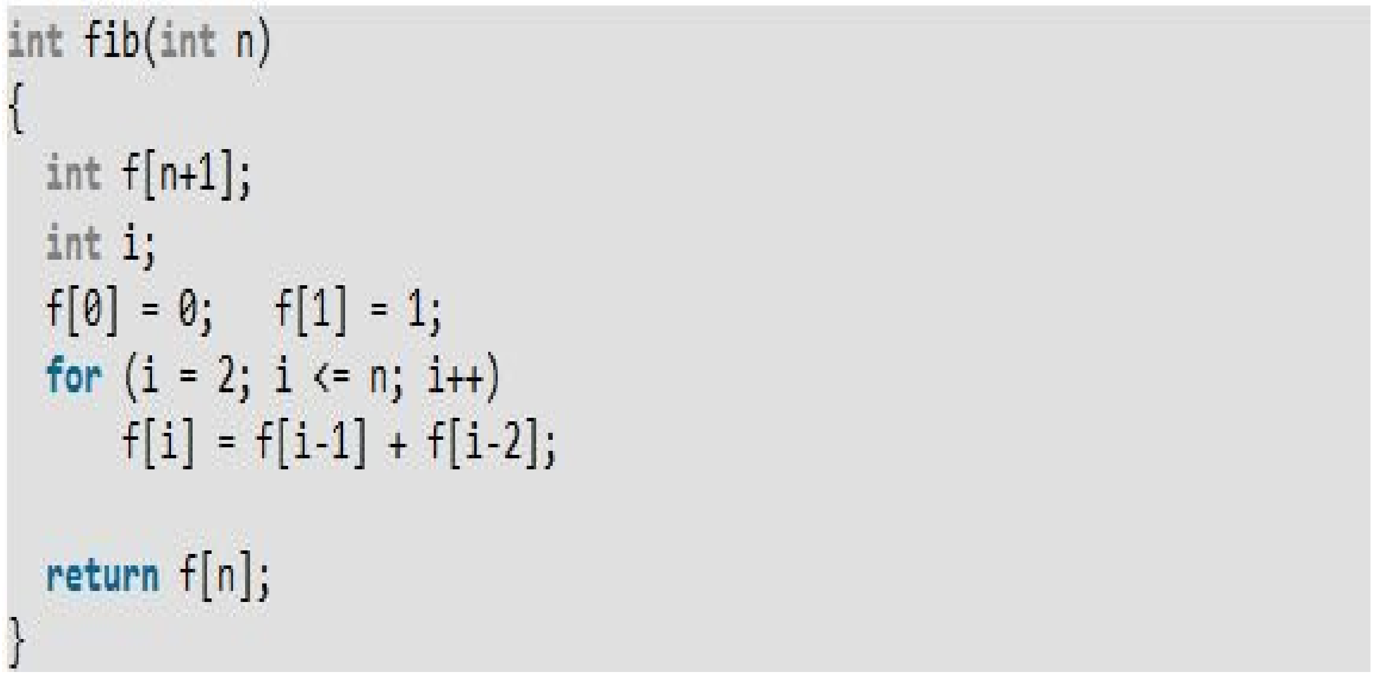
a) **Memoization (Top Down):** The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.



```
/* function for nth Fibonacci number */
int fib(int n)
{
   if (lookup[n] == NIL)
   {
      if (n <= 1)
         lookup[n] = n;
      else
         lookup[n] = fib(n-1) + fib(n-2);
   }
```

Properties of Dynamic Programming

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.



```
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}
```

Unit II : Dynamic Programming

Introduction to Dynamic Programming :

- In D&C strategy we **divide** an instance into smaller sub-instances, the sub-instances are further divided into smaller sub-instances and this is continued till the solution of the sub-instances is **trivial**. Then we **combine** these solutions to get the solution to the original instance.

- It sometimes happens that the natural way of dividing an instance suggested by the structure of the problem leads us to consider several **overlapping sub-instances**. If we solve these sub-instances independently, they will in turn create a host of identical sub-instances. If we do not pay attention to this duplication, we are likely to end up with **inefficient algorithm**, but on the other hand we take the advantage of the duplication and arrange to **solve each sub-instance only once** and **save the solution for later use**, then more efficient algorithm will result. In **Dynamic Programming** exactly this is done.

Unit II : Dynamic Programming

Introduction to Dynamic Programming … contd. :

- The underlying idea of Dynamic Programming (**DP**) is thus quite simple. **It avoids calculating same thing more than once** and it saves known results in a **table** which gets populated as sub-instances are solved.

- **D&C** is a **top-down** method of dividing the instance into sub-instances and solving them independently. **DP** on the other hand is a **bottom-up** technique. We usually start with the **smallest**, and hence the simplest, sub-instances. By combining their solutions (usually solutions of **previous sub-instances**) we obtain solutions to sub-instances of **increasing size**. This process is continued till we arrive at the solution of the original instance.

- In **DP** we get the optimal solutions to sub-instances **(Optimal Sub-structure)** which are used into sub-instances of increasing size . **Optimal Sub-structure** is one of the **element of DP.**

- In **Greedy approach,** at a time one element from the input is included in the solution **which always finds a place in final solution** whereas in DP, **solutions to earlier sub-instances may not find a place in final solution**.

0/1 Knapsack Problem : Problem Definition

Problem Statement :
We are given ‘n’ objects and a knapsack or a bag.
Object ‘i’ has a weight wi, 1<=i<=n, and the knapsack has a capacity ‘m’ (maximum weight the knapsack can hold).

If an object **xi € {0, 1}** is placed into the knapsack then a profit of pixi is earned.
The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is ‘m’, total weight of chosen objects should not exceed ‘m’. Profits and weights are positive numbers.

This problem is Subset Selection problem.

Knapsack problem: brute-force approach

- Since there are **n** items, there are **2^n** possible combinations of items (0/1)

- We go through all combinations and find the one with the most total **value** and with **total weight less or equal to W**

- Running time will be $O(2^n)$

Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the sub-problems

Mathematical Model of 0/1 Knapsack Problem :

maximize $\sum_{1 \le i \le n} p_i x_i$ …….. (1)

subject to $\sum_{1 \le i \le n} w_i x_i \le m$ …….. (2)

$x_i \in \{0, 1\}$, $1 \le i \le n$ ………(3)

$p_i \ge 0$, $w_i \ge 0$ ……...(4)

Example 5.21 Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

Example : n = 4 , (w1, w2, w3, w4) = (2, 3, 4, 5), (p1, p2, p3, p4) = (2, 3, 5,7), m=7

Example : n = 6 , (w1, w2, w3, w4, w5, w6) =(p1, p2, p3, p4, p5,p6) = (100, 50, 20, 10, 7, 3) , m=165

Solving Knapsack using Sets method

Example : n = 3 , (w1, w2, w3) = (2, 3, 4) , (p1, p2, p3) = (1, 2, 5) , m=6

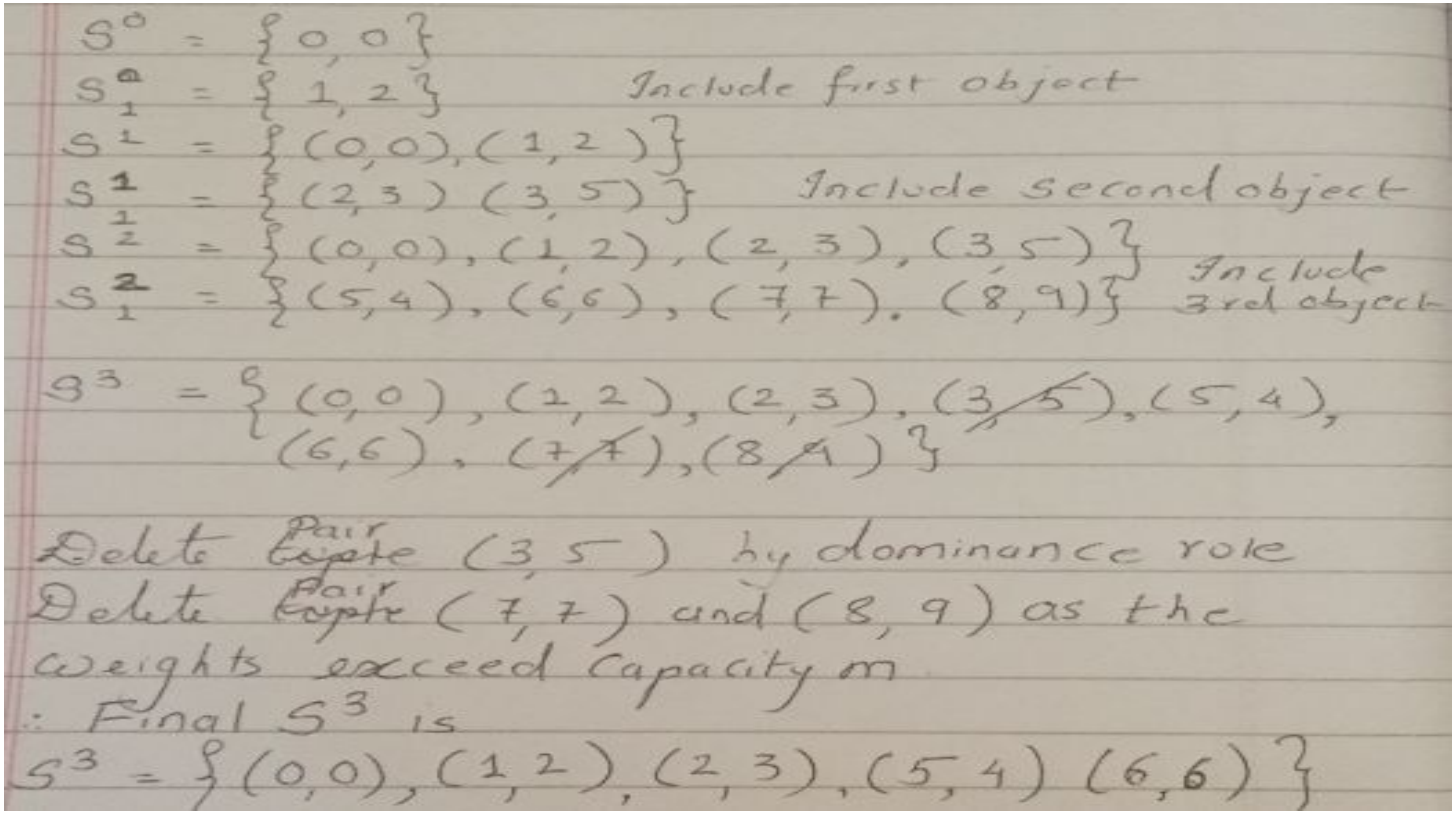
$S^0 = \{0, 0\}$ next compute $\boxed{S_1^0}$ to compute S^1 $\boxed{S_1^i = \{(P, W) \mid (P - p_i, W - w_i) \in S^{i-1}\}}$

We compute S^{i+1} (S^1) from S^i (S^0) by first computing S_1^i

S^i may now be obtained by merging together S^{i-1} and S_1^i.

$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

Example 5.21 Consider the knapsack instance $n = 3, (w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have



$S^0 = \{0, 0\}$

$S^0_1 = \{1, 2\}$ Include first object

$S^1 = \{(0,0), (1,2)\}$

$S^1_1 = \{(2,3)\ (3,5)\}$ Include second object

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$

$S^2_1 = \{(5,4), (6,6), (7,7), (8,9)\}$ Include 3rd object

$S^3 = \{(0,0), (1,2), (2,3), (\cancel{3,5}), (5,4), (6,6), (\cancel{7,7}), (\cancel{8,9})\}$

Delete ~~tuple~~ Pair (3,5) by dominance role

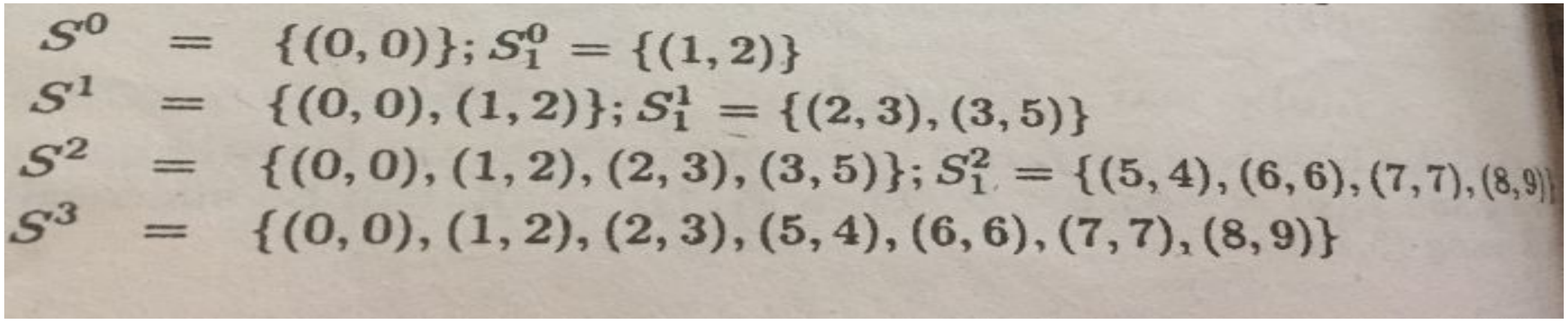
Delete ~~tuple~~ Pair (7,7) and (8,9) as the weights exceed capacity m

∴ Final S^3 is

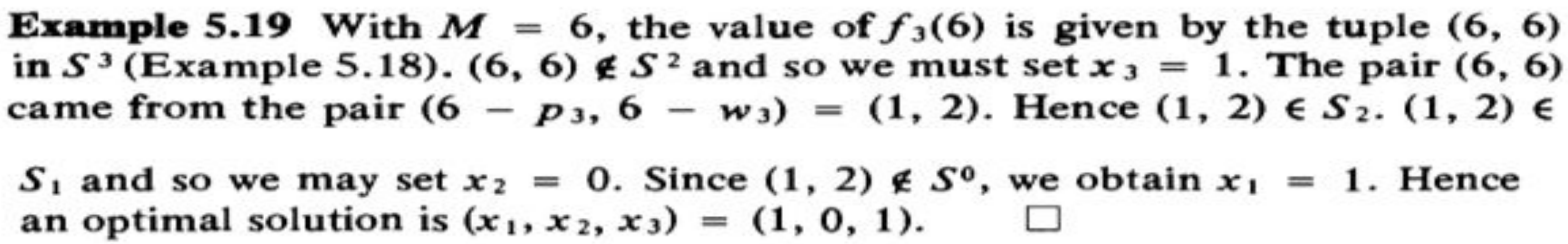
$S^3 = \{(0,0), (1,2), (2,3), (5,4)\ (6,6)\}$

Solving Knapsack using Sets method

Example : n = 3 , (w1, w2, w3) = (2, 3, 4) , (p1, p2, p3) = (1, 2, 5) , m=6



$$S^0 = \{(0,0)\};\ S_1^0 = \{(1,2)\}$$
$$S^1 = \{(0,0),(1,2)\};\ S_1^1 = \{(2,3),(3,5)\}$$
$$S^2 = \{(0,0),(1,2),(2,3),(3,5)\};\ S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\}$$
$$S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}$$



Example 5.19 With $M = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.18). (6, 6) $\notin S^2$ and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence $(1, 2) \in S_2$. $(1, 2) \in$

S_1 and so we may set $x_2 = 0$. Since $(1, 2) \notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. □

Solving Knapsack using Sets method

Example : n = 4 , (w1, w2, w3, w4) = (2, 3, 4, 5) ,
(p1, p2, p3, p4) = (2, 3, 5,7) , m=7

Solve above example of 0/1 knapsack using sets method

Informal Algorithm

```
Algorithm DKP($p, w, n, m$)
{
    $S^0 := \{(0,0)\}$;
    for $i := 1$ to $n-1$ do
    {
        $S_1^{i-1} := \{(P,W) | (P - p_i, W - w_i) \in S^{i-1}$ and $W \leq m\}$;
        $S^i :=$ MergePurge($S^{i-1}, S_1^{i-1}$);
    }
    $(PX, WX) :=$last pair in $S^{n-1}$;
    $(PY, WY) := (P' + p_n, W' + w_n)$ where $W'$ is the largest $W$ in
        any pair in $S^{n-1}$ such that $W + w_n \leq m$;
    // Trace back for $x_n, x_{n-1}, \ldots, x_1$.
    if $(PX > PY)$ then $x_n := 0$;
    else $x_n := 1$;
    TraceBackFor($x_{n-1}, \ldots, x_1$);
}
```

Unit II : Dynamic Programming

0/1 Knapsack Problem : Solution by DP strategy (Recursion)

- Let **gi(y)** denote the value of optimal solution to **knap(i+1, n, y),** where (i+1) to n are the objects and y is the capacity of knapsack.
- Clearly **g0(m)** is the value of an optimal solution to **knap(1, n, m)**
- The possible decisions for x1 are 0 or 1. From the principle of optimality it follows that ,

$$g_0(m) = \max \{ \underset{\text{if } x_1 = 0}{g_1(m)},\ \underset{\text{if } x_1 = 1}{(g_1(m - w_1) + p_1)} \}$$

- The above equation can be generalized as,

$$g_i(y) = \max \{ \underset{\text{if } x_{i+1} = 0}{g_{i+1}(y)},\ \underset{\text{if } x_{i+1} = 1}{(g_{i+1}(y - w_{i+1}) + p_{i+1})} \}$$

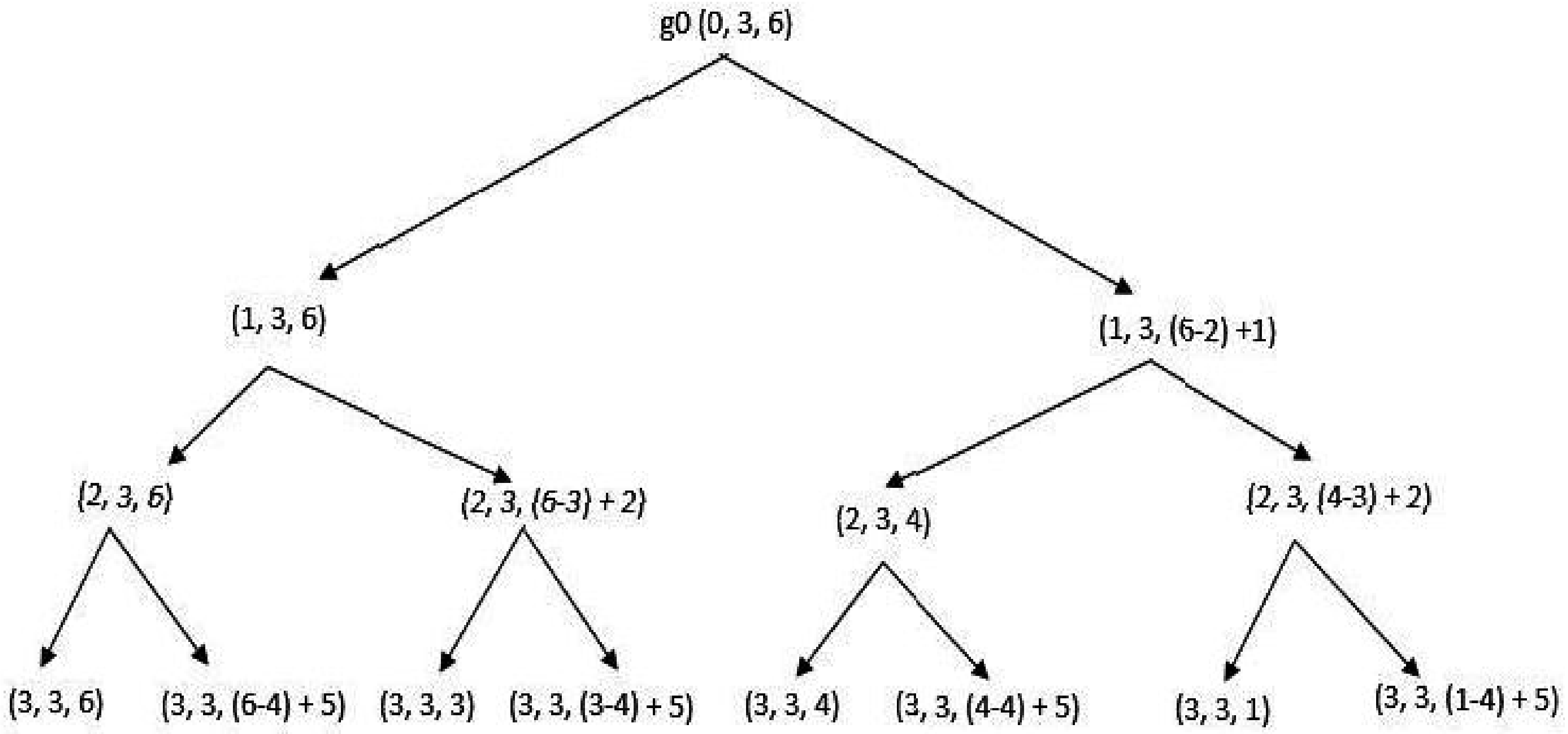
- This equation can be used to obtain gn-1(y) from gn(y) and can be further used recursively to obtain **optimal solution g0(y)** with the knowledge that $g_n(y) = 0$ for all $y >= 0$ and

$$g_n(y) = -\infty \qquad \text{for all } y < 0$$

Solving Knapsack using Sets method

Example n = 3 , m=6

(w1, w2, w3) = (2, 3, 4) , (p1, p2, p3) = (1, 2, 5)



Unit II : Dynamic Programming

0/1 Knapsack Problem : Solution by DP strategy (Recursion)

Now we can write down recursive algorithm to solve 0/1 knapsack problem using above equations.

```
Algorithm    0-1-knapsack-rec( i, j, m)
// global array arr[1:n] contains weights and corresponding profits for // objects 1 to
n
{    if    (i = j)
          {    if    m >= 0
                     return (0)
               else  return (- ∞ )
          }
     else  return (max((01-knapsack(i+1, j, m)),
                        (01-knapsack(i+1, j, m-w(i+1))+ p(i+1)))
}
```

Example : Solve the following instance using algorithm 0-1-knapsack-rec.

$n = 3$, $(w1, w2, w3) = (2, 3, 4)$ and

$m = 6$, $(p1, p2, p3) = (1, 2, 5)$

Unit II : Dynamic Programming

0/1 Knapsack Problem : Solution by DP strategy (Recursion)

Analysis of Algorithm 0-1-knapsack-rec :

The recurrence equations will be,

$$t(n) = \begin{cases} 0 & \text{if } n=0 \\ 2t(n-1) + 1 & \text{otherwise} \end{cases}$$

After solving this recurrence we get,

Time Complexity = **$O(2^n)$**

Space Complexity = **$O(n)$** … space required for stack

Is the TC using Dynamic Programming ?

Unit II : Dynamic Programming

0/1 Knapsack Problem : Solution by DP strategy (DP)

Algorithm 0-1-knapsack-dp : In this method table of size (1:m, 1:n] is populated either row wise or column wise. In row wise method objects are considered one at a time starting from 1st object to nth object. Following Rule is used,

$$P[i, j] = \max (P[i-1, j],\ P[i-1, j-w[i]] + p[i])$$

Example : Solve the following instance using Dynamic Programming.

$n = 3$, $(w1, w2, w3) = (2, 3, 4)$ and
$m = 6$, $(p1, p2, p3) = (1, 2, 5)$

… j →

	Weight Limit 1 to n ->	**00**	**1**	**2**	**3**	**4**	**5**	**6**
i ↓	**w1 = 2, p1 = 1**	0	0	1	1	1	1	1
	w2 = 3, p2 = 2	0	0	1	2	2	3	3
	w3 = 4, p3 = 5	0	0	1	2	5	5	6

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)]
 for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(capacity + 1):
            if weights[i - 1] <= j:
                dp[i][j] = max(dp[i - 1][j],profit[i - 1]][j - weights[i - 1]] + dp[i -
            else:
                dp[i][j] = dp[i - 1][j]

    return dp[n][capacity]

# Example usage
weights = [3, 2, 4, 5]
profit = [6, 8, 7, 10]
capacity = 5
print("Maximum value:", knapsack(weights, values, capacity))
```

- weights is a list of weights of items.
- profit is a list of corresponding profits of items.
- capacity is the maximum weight the knapsack can hold
- dp array is a 2D array where dp[i][j] represents the maximum value that can be achieved using the first i items, with a knapsack of capacity j.
- dp[n][capacity] will give you the maximum value that can be obtained using all the items and the given knapsack capacity.

Unit II : Dynamic Programming

Example : Solve the following instance using Dynamic Programming.

n = 4, (w1, w2, w3,w3) = (1, 3, 4, 5) and

m = 7, (p1, p2, p3, p3) = (1, 4, 5, 7)

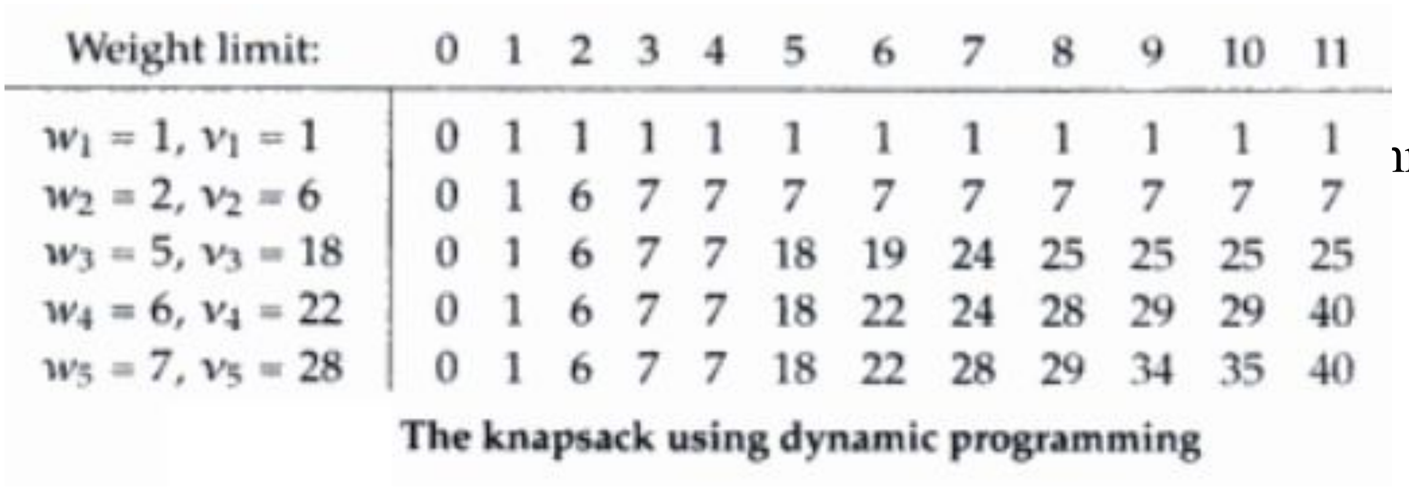
... j →

i ↓ Weight Limit 1 to n ->	00	1	2	3	4	5	6	7
w1 = 1, p1 = 1	0	1	1	1	1	1	1	1
w2 = 3, p2 = 4	0	1	1	4	5	5	5	5
w3 = 4, p3 = 5	0	1	1	4	5	6	6	9
w3 = 5, p4 = 7	0	1	1	4	5	7	8	9

Example : Solve the following instance using Dynamic Programming.

n = 5, (w1, w2, w3,w4,w5) = (1, 2,5,6,7) and

m = 11, (p1, p2, p3, p4,p5) = (1, 6, 18, 22, 28) ming.

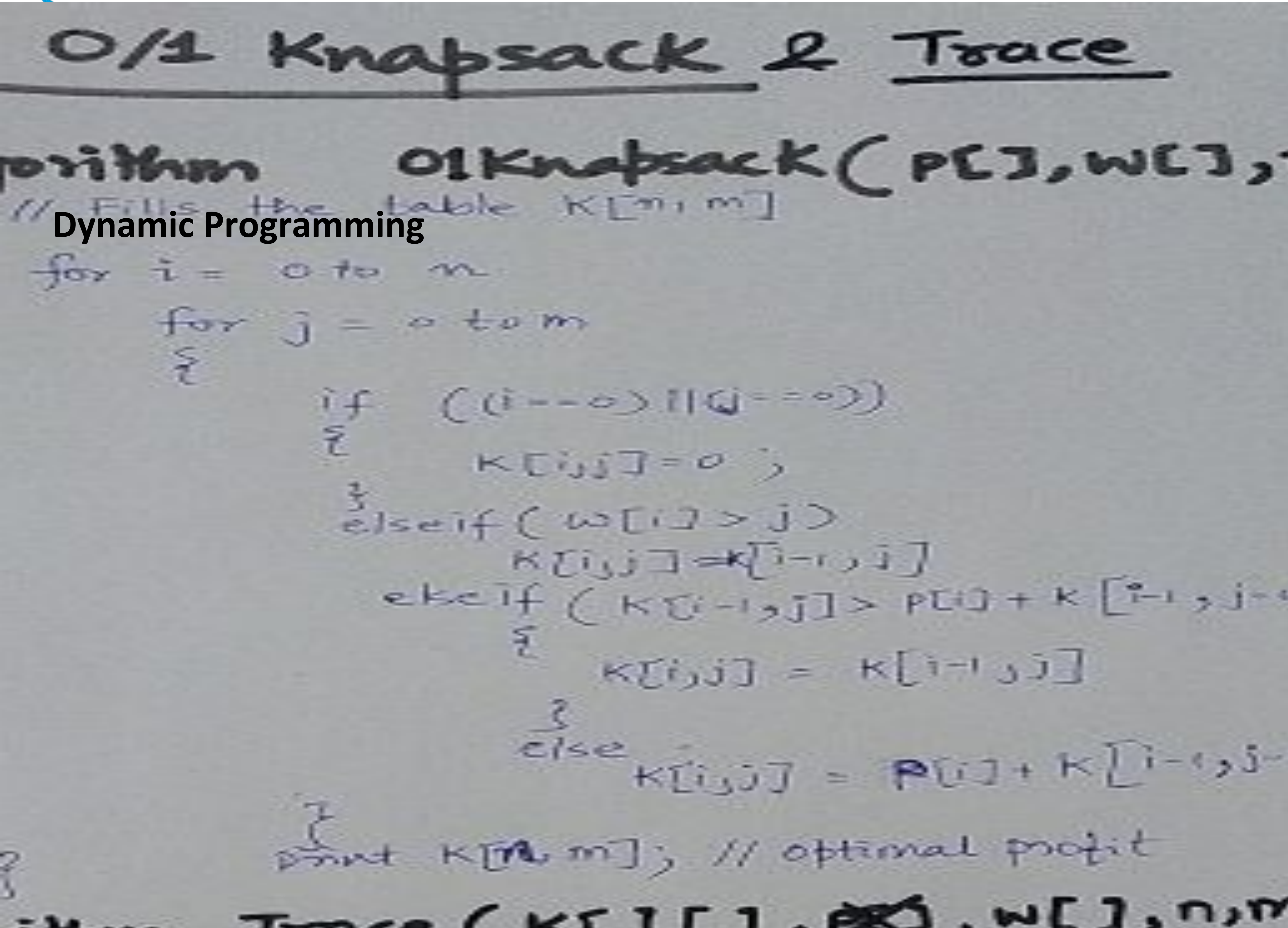


Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The knapsack using dynamic programming

$$V[i,j] = \max(V[i-1,j], V[i-1,j-w_i]+v_i).$$

Dynamic Programming



0/1 Knapsack & Trace

```
orithm 01Knapsack(P[],W[],
  // Fills the table K[n,m]
  for i = 0 to n
      for j = 0 to m
      {
          if ((i==0)||(j==0))
          {
              K[i,j]=0;
          }
          elseif (w[i] > j)
              K[i,j]=K[i-1,j]
            elseif (K[i-1,j] > P[i] + K[i-1, j-
              {
                  K[i,j] = K[i-1,j]
              }
              else
                  K[i,j] = P[i] + K[i-1,j-
      }
      print K[n,m]; // optimal profit
}
ithm Trace(K[][], [illegible], W[],n,m
```

Unit II : Dynamic Programming

Example : Solve the following instance using Dynamic Programming.

n = 4, (w1, w2, w3,w3) = (2, 3, 4, 5) and

m = 8, (p1, p2, p3, p3) = (1, 2, 5, 6)

0/1 Knapsack Problem : Solution by DP strategy

Analysis of Algorithm 0-1-knapsack-dp :

To populate the table :

Time Complexity = **O(nm)**

Space Complexity = **O(nm)**

To trace the solution :

Time Complexity = **O(n+m)**

For **large values** of ‘m’ & ‘n’ **space complexity** is critical.

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- Second case: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
        B[i,0] = 0
        for w = 0 to W
                if $w_i$ <= w // item i can be part of the solution
                        if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
                                B[i,w] = $b_i$ + B[i-1,w- $w_i$]
                        else
                                B[i,w] = B[i-1,w]
                else B[i,w] = B[i-1,w]  // $w_i$ > w
```

Running time

```
for w = 0 to W                    O(W)
   B[0,w] = 0
for i = 0 to n                    Repeat n times
   B[i,0] = 0
   for w = 0 to W                 O(W)
        < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Real-Life Problems : Use 01 Knapsack

0/1 Knapsack problem is used in retail and e-commerce to optimize pricing and inventory management, as well as the allocation of shelf space in physical stores.

It is used in supply chain management to optimize logistics and transportation routes, as well as the allocation of resources such as trucks and warehouses.

Used to model and solve problems in project management, such as resource allocation, scheduling, and budget optimization.

Used in the optimization of manufacturing processes, such as the selection and allocation of machines and resources.

Used in finance and investment management, such as portfolio optimization and asset allocation.

Sample Examples

- Resource Allocation: The knapsack problem is often used to allocate resources in a limited budget. For example, a company may have a limited budget for purchasing raw materials, and the knapsack problem can help them select the best combination of materials that will maximize their profits.
- Portfolio Optimization: In finance, the knapsack problem is used to optimize investment portfolios by selecting the best combination of assets that will maximize the portfolio's returns while staying within a budget.
- Cutting Stock Problem: The knapsack problem is also used in the cutting stock problem, where a manufacturer needs to cut a certain number of items from large sheets of material. The knapsack problem can help determine the best way to cut the material to minimize waste and maximize the number of items produced.
- Data Compression: In computer science, the knapsack problem can be used for data compression by selecting the most efficient combination of bits to represent data.
- Inventory Management: The knapsack problem can be used in inventory management to determine the optimal inventory level for each product, taking into account factors such as demand, storage costs, and ordering costs.
- Project Scheduling: In project management, the knapsack problem can be used to schedule tasks by selecting the most efficient combination of tasks to complete a project within a given time frame.
- Resource Planning: The knapsack problem can be used in resource planning to determine the best allocation of resources such as personnel, equipment, and materials to complete a project within a budget.

Optimal Binary Search Tree

Unit II : Dynamic Programming

Optimal Binary Search Tree (OBST) : Introduction

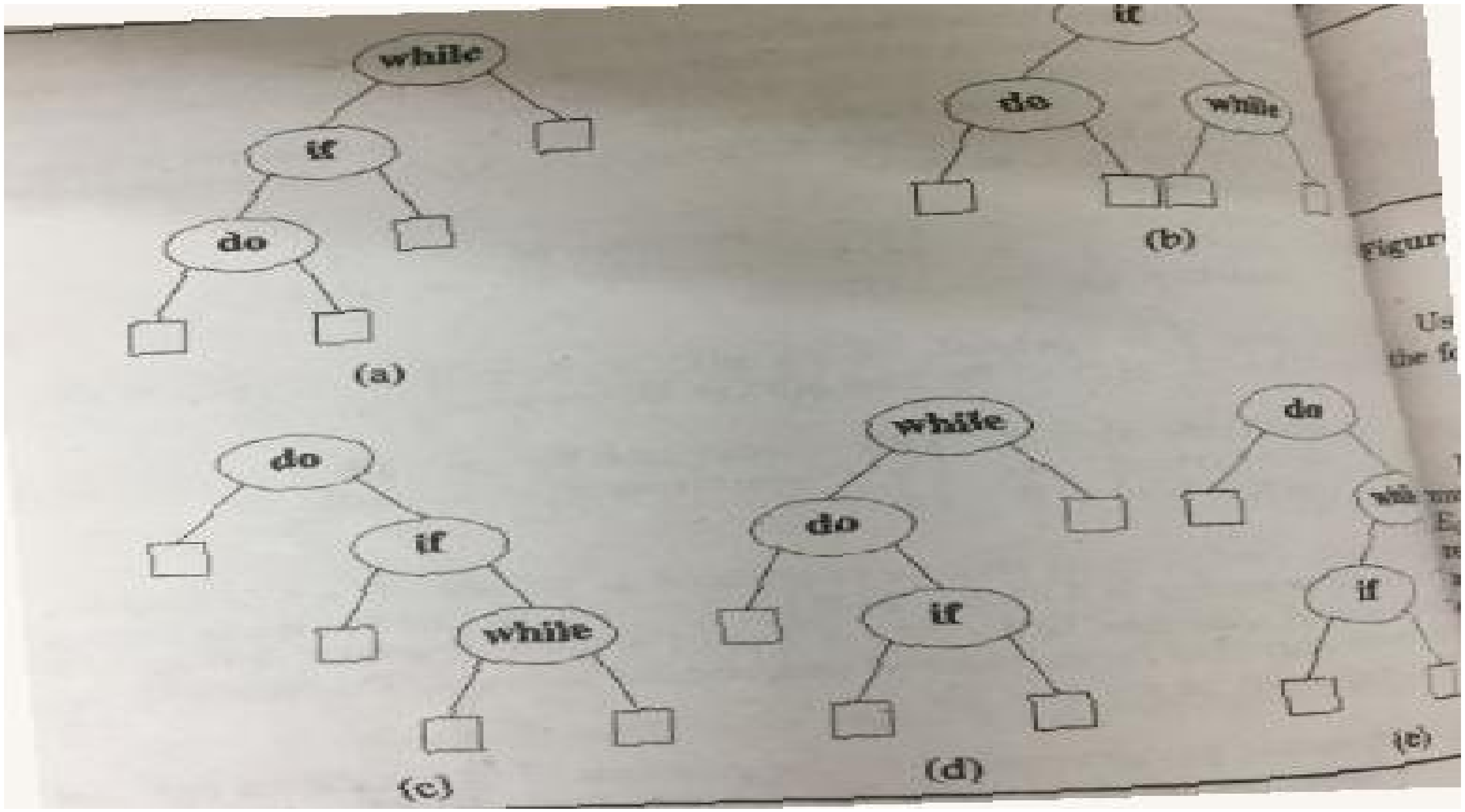
- Binary Search Trees (BSTs) are used to for search an identifier or a symbol in a table. There are many practical applications of BST. Sometimes we need to construct an Optimal BST (OBST) (with weights based on frequency of occurrence) so that searching an identifier can be done in optimal time. For example any language compiler or interpreter maintains symbol tables which are searched to know whether the given identifier is present in the symbol table.

- **Example :** {a1, a2, a3} = {do, if, while}, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?
- Find OBST if weights are not equal with (p1, p2, p3) = (0.5, 0.1, 0.05) and (q0, q1, q2, q3) = (0.15, 0.1, 0.05, 0.05)

For given ‘n’ identifiers and their corresponding weights, there are $\mathbf{({}^{2n}C_n)/(n+1)}$ number of different possible trees. We can work out the total cost of each tree and then select the optimal BST. But it is quite exhaustive even for a moderate value of ‘n’.

Unit II : Dynamic Programming

- **Example :** {a1, a2, a3} = {do, if, while}, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?
- Find OBST if weights are not equal with (p1, p2, p3) = (0.5, 0.1, 0.05) and (q0, q1, q2, q3) = (0.15, 0.1, 0.05, 0.05)



Unit II : Dynamic Programming

- **Example :** {a1, a2, a3} = {do, if, while}, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?
- Find OBST if weights are not equal with (p1, p2, p3) = (0.5, 0.1, 0.05) and (q0, q1, q2, q3) = (0.15, 0.1, 0.05, 0.05)

Cost contribution of internal node ai is **p(i) * level(ai)**

Unsuccessful searches terminate with i = 0 (i.e. at an external node) in algorithm SEARCH. The identifiers not in the binary search tree may be partitioned into n + 1 equivalence classes Ei 0 <=i <= n. Eo contains all identifiers x such that x < a1. Ei contains all identifiers x such that ai < x < ai+1, 1 <= i < n. En contains all identifiers x, x > an. It is easy to see that for all identifiers in the same class Ei, the search terminates at the same external node. For identifiers in different Ei the search terminates at different external nodes. If the failure node for Ei is at level l then only l - 1 iterations of the while loop are made. Hence, the cost contribution of this node is **q (i) * (level(Ei) - 1).**

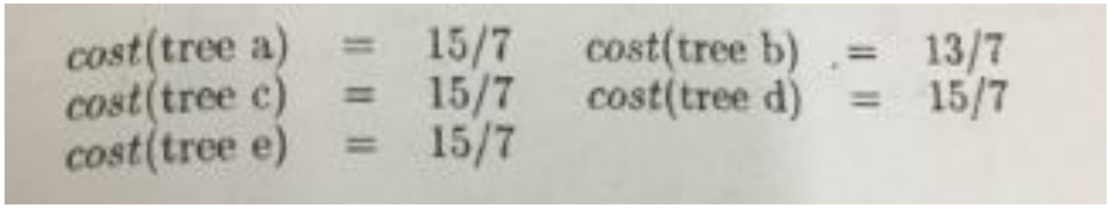
- $\text{Cost} = \sum_{1 \le i \le n} p_i * \text{level}(a_i) + \sum_{0 \le i \le n} q_i * (\text{level}(E_i) - 1)$

Unit II : Dynamic Programming

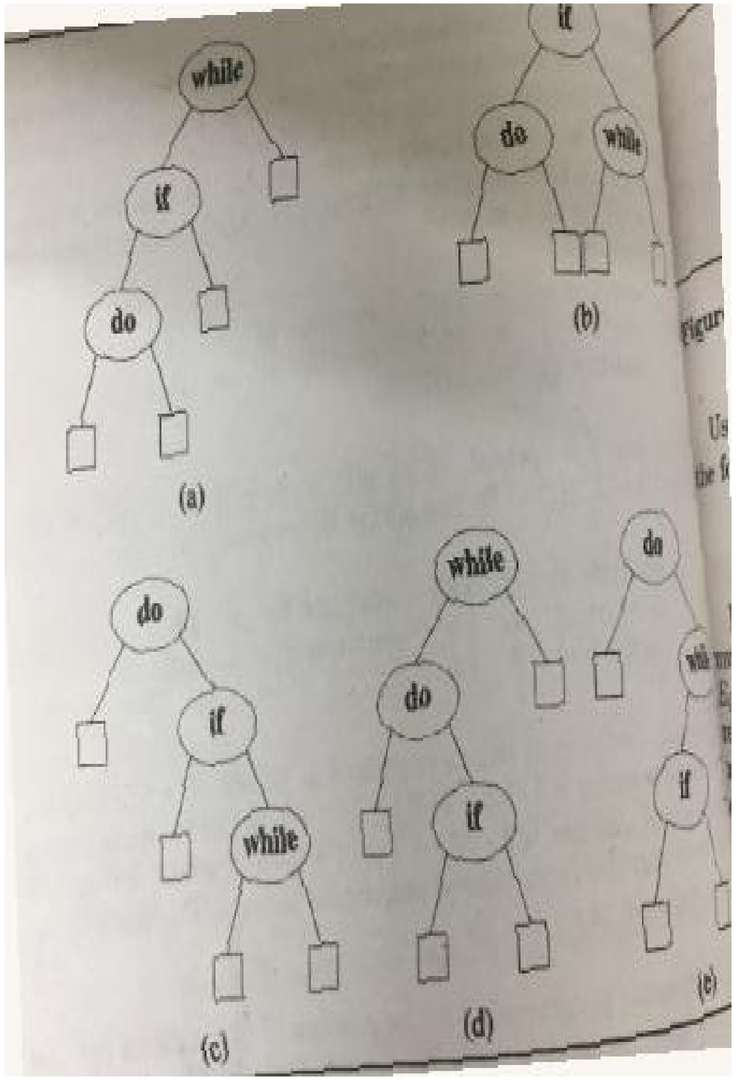
- $\text{Cost} = \Sigma\, pi * \text{level}\,(ai) + \Sigma\, qi * (\text{level}\,(Ei) - 1)$

  $1 <= i <= n \qquad 0 <= i <= n$

- {a1, a2, a3} = {do, if, while}, assume
- Equal probabilities pi = qi = 1/7.

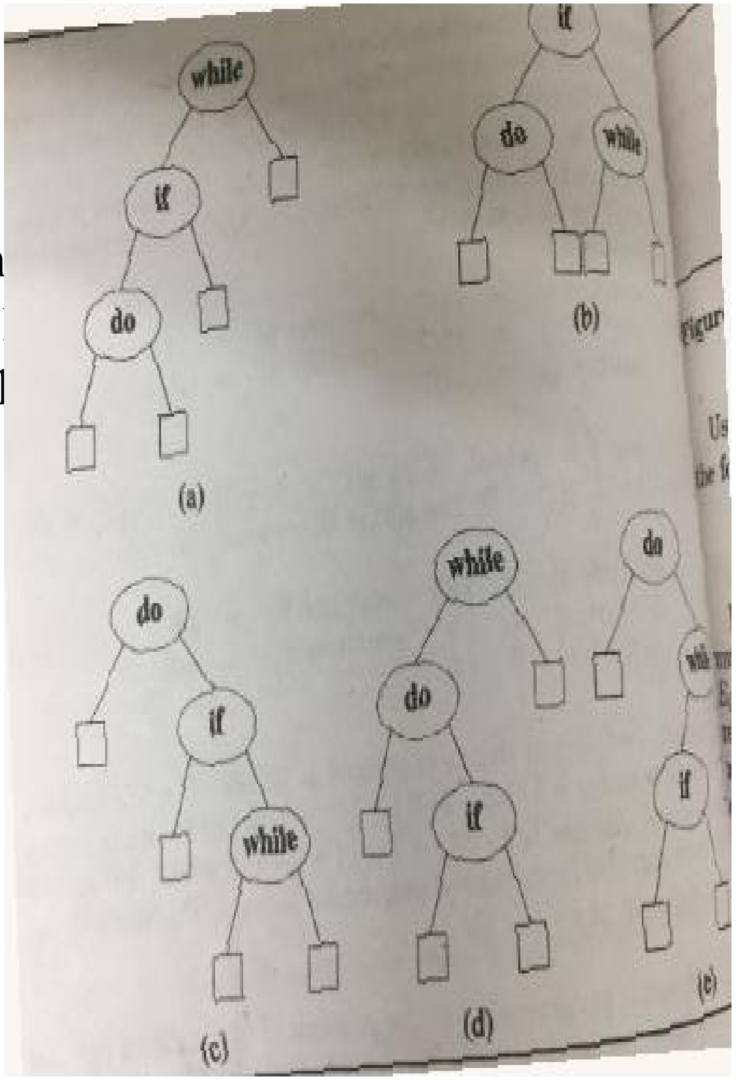


cost(tree a)	=	15/7	*cost*(tree b)	=	13/7
cost(tree c)	=	15/7	*cost*(tree d)	=	15/7
cost(tree e)	=	15/7			



Find OBST if weights are n

- (p1, p2, p3) = (0.5, 0.
- (q0, q1, q2, q3) = (0.15, 0.

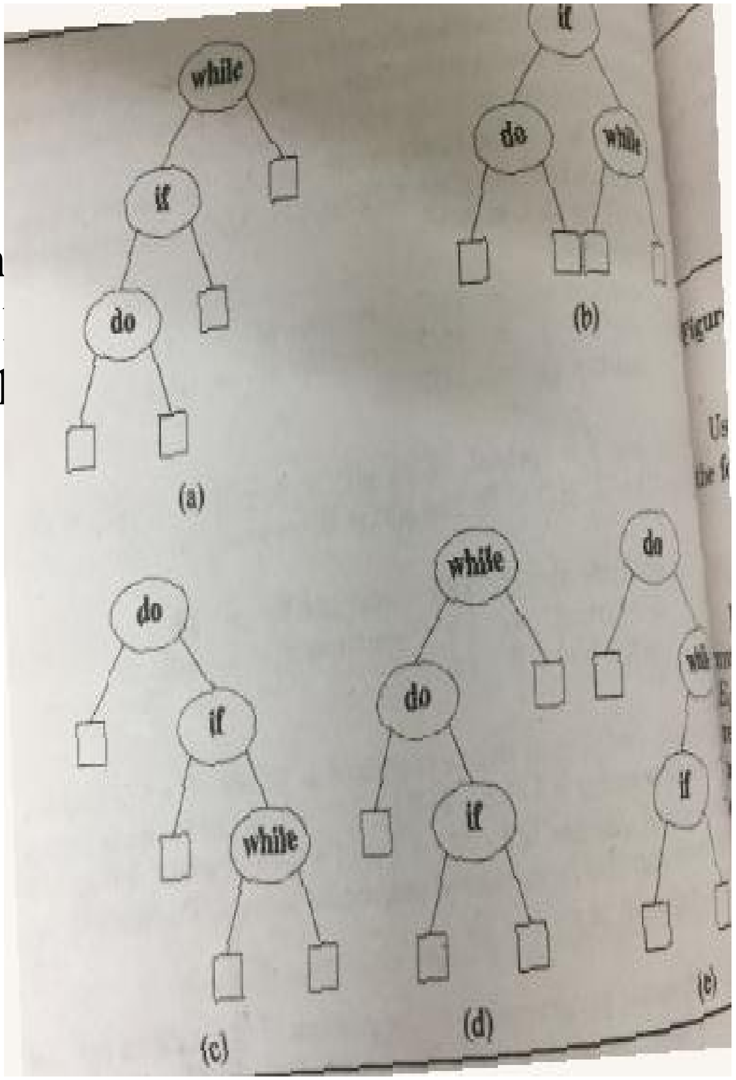


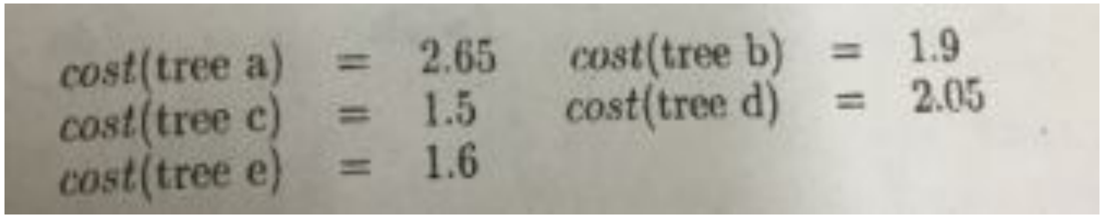
- Cost = Σ pi * level (ai) + Σ qi * (level (Ei) – 1)
  1<= i <=n   0<= i <=n

Find OBST if weights are n

•(p1, p2, p3) = (0.5, 0.

•(q0, q1, q2, q3) = (0.15, 0.1





$cost(\text{tree a}) = 2.65$ $cost(\text{tree b}) = 1.9$

$cost(\text{tree c}) = 1.5$ $cost(\text{tree d}) = 2.05$

$cost(\text{tree e}) = 1.6$

Unit II : Dynamic Programming

Optimal Binary Search Tree (OBST) : Problem Definition

Problem Specifications :

Let us assume given set of identifiers as {a1, a2, …, an} with a1<a2<, … ,<an.

- Let p(i) be the probability with which we search for ai and let q(i) be the probability that the identifier x being searched for is such that ai < x < ai+1. , $0 \leq i \leq n$.
- Further assume that a0 = - ∞ and an+1= +∞.
- Probability of unsuccessful searches = $\sum_{0 \le i \le n} q[i]$
- Probability of successful searches = $\sum_{1 \le i \le n} p[i]$
- $\sum_{1 \le i \le n} p[i] + \sum_{0 \le i \le n} q[i] = 1$
- With this data available now problem is to construct a BST with minimum cost i.e. OBST.

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

- To apply DP approach for obtaining OBST, we need to view the construction of such a tree as the result of **sequence of decisions** and observe that the principle of optimality holds.
- A possible approach to this would be to decide which of the ai's (with weights pi's) should be selected as the root node of the tree.
- If we choose ak as the root node from a1, a2, …, an (sorted in non-decreasing order) , then it is clear that the internal nodes a1, a2, …, ak-1 and external nodes for classes E0, E1, E2, …, Ek-1 will be in the **left sub tree *l*** and the internal nodes ak+1, ak+2, …, an and external nodes for classes Ek+1, Ek+2, …, En will be in the **right sub tree *r***. Let root ak be at level 1.
- $$\text{Cost}(l) = \sum_{1 \le i < k} pi * \text{level}(ai) + \sum_{0 \le i < k} qi * (\text{level}(Ei) - 1)$$

  $$\text{Cost}(r) = \sum_{k < i \le n} pi * \text{level}(ai) + \sum_{k \le i \le n} qi * (\text{level}(Ei) - 1)$$

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

- Let w(i, j) = Total weight of BST containing identifiers ai+1, ai+2, …, aj and qi, qi+1, …, qj

$$w(i, j) = q_i + \sum_{l=i+1}^{j} (q_l + p_l)$$

$$\mathbf{w(i, j) = p_j + q_j + w(i, j-1)} \qquad \textbf{… (1)}$$

- Cost of expected BST

$$= p(k) + cost(l) + cost(r) + w(0, k-1) + w(k, n) \qquad \textbf{… (2)}$$

- For the OBST, equation (2) must be minimum i.e. cost(*l*) & cost(*r*) must also be minimum,

$$cost(l) = c(0, k-1)$$

$$cost(r) = c(k, n)$$

Dynamic Programming approach to construct OBST :

- k must be chosen such that,

  p(k)+ c (0, k-1) + c(k, n) + w(0, k-1) + w(k, n) is minimum

- Cost of OBST

  $$c(0, n) = \min_{1 <= k <= n}\{c(0, k-1) + c(k, n) + p(k) + w(0, k-1) + w(k, n)\} \quad \textbf{… (3)}$$

- We can generalize equation (3) above as,

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

  $$\mathbf{c(i, j) = \min_{i < k <= j}\{c(i, k-1) + c(k, j) + w(i, j)\}} \quad \textbf{… (4)}$$

  with the knowledge that,

  $$\mathbf{c(i, i) = 0, \quad w(i, i) = qi \quad \& \quad r(i, i) = 0} \quad \textbf{… (5)}$$

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

Example :

Construct OBST for the following instance,

n = 4, (a1, a2, a3, a4) = (do, if, int, while)

(p1, p2, p3, p4) = (3, 3, 1, 1)

(q0, q1, q2, q3, q4) = (2, 3, 1, 1, 1)

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

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

n = 4, (a1, a2, a3, a4) = (do, if, int, while) (p1, p2, p3, p4) = (3, 3, 1, 1) (q0, q1, q2, q3, q4) = (2, 3, 1, 1, 1)

(i, j)	w(i,j)= p(j)+q(j)+w(i,j-1)	C(i,j)=w(i, j) + min {c(i, k-1) + c(k, j)} i< k ≤ j	Int. Values	Min. Value	k
(0, 1)	p(1)+q(1) + w(0,0)= 3 + 3 + 2 = 8	w(0, 1) + c(0, 0) + c(1,1)	8 + 0	8	1
(1, 2)	p(2)+q(2) + w(1,1) = 3 + 1 + 3 = 7	w(1, 2) + c(1, 1) + c(2,2)	7 + 0	7	2
(2, 3)	p(3)+q(3) + w(2,2) = 1 + 1 + 1 = 3	w(2, 3) + c(2, 2) + c(3,3)	3 + 0	3	3
(3, 4)	p(4)+q(4) + w(3,3) = 1 + 1 + 1 = 3	w(3, 4) + c(3, 3) + c(4,4)	3 + 0	3	4
(0, 2)	p(2)+q(2) + w(0,1) = 3 + 1 + 8 = 12	w(0, 2) + min ({c(0, 0) + c(1, 2)},{c(0, 1) + c(2, 2)})	12 + min{7, 8 }	19	1 , 2
(1, 3)	p(3)+q(3) + w(1,2) =1 + 1 + 7 = 9	w(1, 3) + min({c(1, 1) + c(2, 3)},{c(1, 2) + c(3, 3)})	9 + min{3, 7}	12	2 , 3
(2, 4)	p(4)+q(4) + w(2,3) = 1 + 1 + 3 = 5	w(2, 4) + min ({c(2, 2) + c(3, 4)} ,{c(2, 3) + c(4, 24})	5 + min{3, 4}	8	3 , 4
(0, 3)	p(3)+q(3) + w(0,2) = 1 + 1 + 12 = 14	w(0, 3) + min {c(0, 0) + c(1, 3)} ,{c(0, 1) + c(2, 3)} ,{c(0, 2) + c(3, 3)}	14 + min{12, 11,19}	25	1, 2, 3
(1,4)	p(4)+q(4) + w(1,3) = 1 + 1 + 9 = 11	w(1, 4) + min {c(1, 1) + c(2, 4)}, {c(1, 2) + c(3, 4)} , {c(1, 3) + c(4, 4)}	11 + min{8, 10,12}	19	2 ,3, 4
(0,4)	p(4)+q(4) + w(0,3) = 1 + 1 + 14 = 16	w(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)}	16 + min{19, 16, 22, 25}	32	1, 2, 3, 4

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

	0	1	2	3	4
0	w00 = 2 c00 = 0 r00 = 0	w11 = 3 c11 = 0 r11 = 0	w22 = 1 c22 = 0 r22 = 0	w33 = 1 c33 = 0 r33 = 0	w44 = 1 c44 = 0 r44 = 0
1	w01 = 8 c01 = 8 r01 = 1	w12 = 7 c12 = 7 r12 = 2	w23 = 3 c23 = 3 r23 = 3	w34 = 3 c34 = 3 r34 = 4	
2	w02 = 12 c02 = 19 r02 = 1	w13 = 9 c13 = 12 r13 = 2	w24 = 5 c24 = 8 r24 = 3		
3	w03= 14 c03 = 25 r03 = 2	w14 = 11 c14 = 19 r14 = 2			
4	w04 = 16 c04 = 32 r04 = 2				

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

Example : Construct OBST for the following instance,

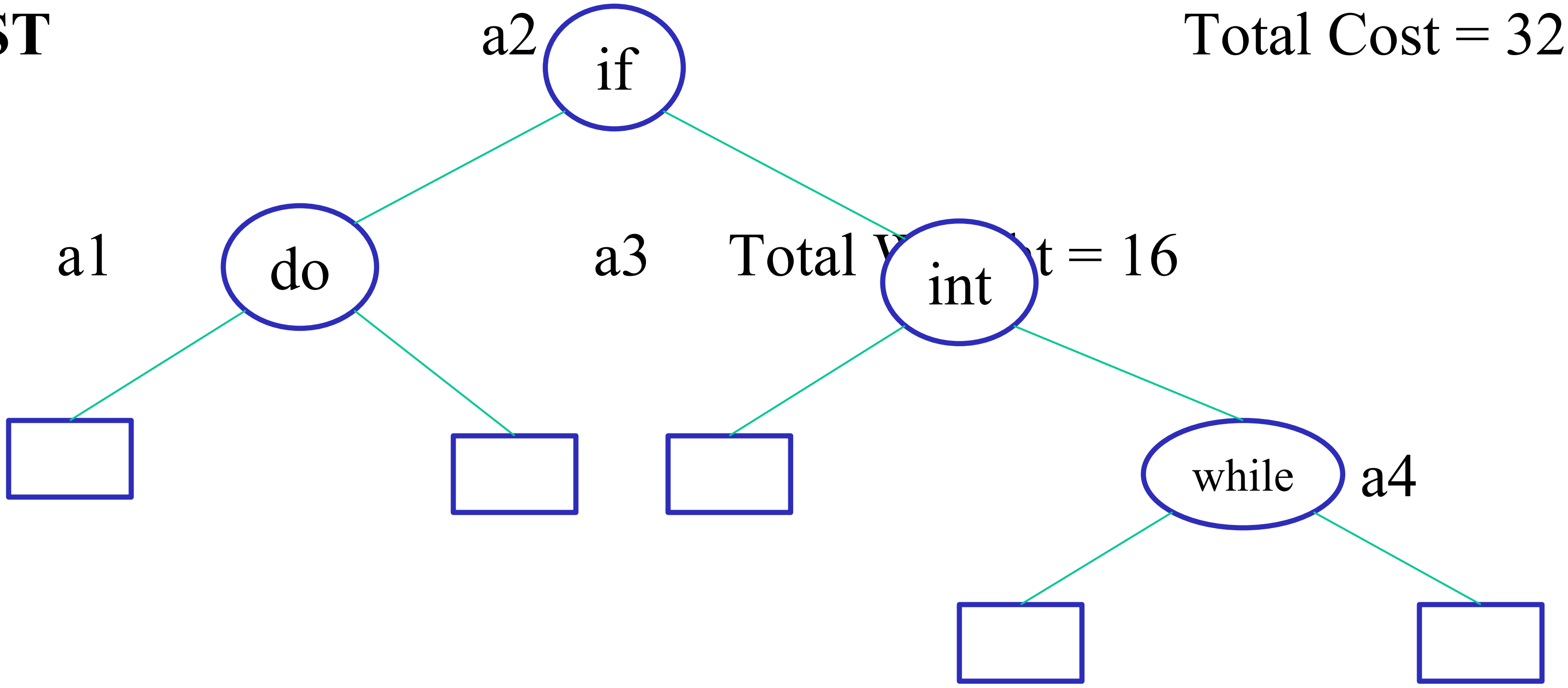
n = 4, (a1, a2, a3, a4) = (do, if, int, while)

(p1, p2, p3, p4) = (3, 3, 1, 1)

(q0, q1, q2, q3, q4) = (2, 3, 1, 1, 1)

OBST

Total Cost = 32



Unit II : Dynamic Programming

Analysis of Algorithm OBST :

- To get OBST we can compute the values of c's and r's by using equations (1), (4) and (5).
- We need to compute c(i, j) for (j – i) = 1, 2, …, n, in that order.
  When (j – i) = m , there are **(n – m + 1)** values of c(i, j)'s to compute.
- Computation of each c(i, j) requires to compute **'m'** quantities (m = j – i)
- Total number of steps to compute c(i, j) = (n – m +1)*m
  i.e. **$O(nm - m^2)$**
- Total time to get OBST = sum of all c(i, j) time

$$= \sum_{1 \le m \le n} (nm - m^2) = \mathbf{O(n^3)}$$

- **Time Complexity = $O(n^3)$**
- **Space Complexity = $O(n^2)$**
- Using result due to D.E. Knuth, Time Complexity is reduced to **$O(n^2)$.** This fact is used in **Algorithm OBST**.

Real-Life Examples of OBST: Information Retrieval

Search engines rely on binary search trees to retrieve information quickly and efficiently
The search algorithm begins at the root of the tree and follows a path through the
tree based on the comparison of the search key with the keys in the nodes.

The optimal binary search tree problem is used to determine the arrangement of keys in
the binary search tree that **minimizes the expected number of comparisons** needed to
retrieve a record. By constructing an optimal binary search tree, search engines can
reduce the time it takes to retrieve information and improve the accuracy of their search results.

Optimal binary search trees are also used in other information retrieval
applications, such as spell checkers, natural language processing systems,
and data compression algorithms. By using an optimal binary search tree,
these systems can improve their efficiency and accuracy, which is critical for many real-world
applications.

Multistage Graph Problem

Unit II : Dynamic Programming

Multi-stage Graphs : Problem Definition

- A multi-stage graph G = (V, E) is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets Vi , 1 <= i <= k, and |V1 |= |Vk |= 1.

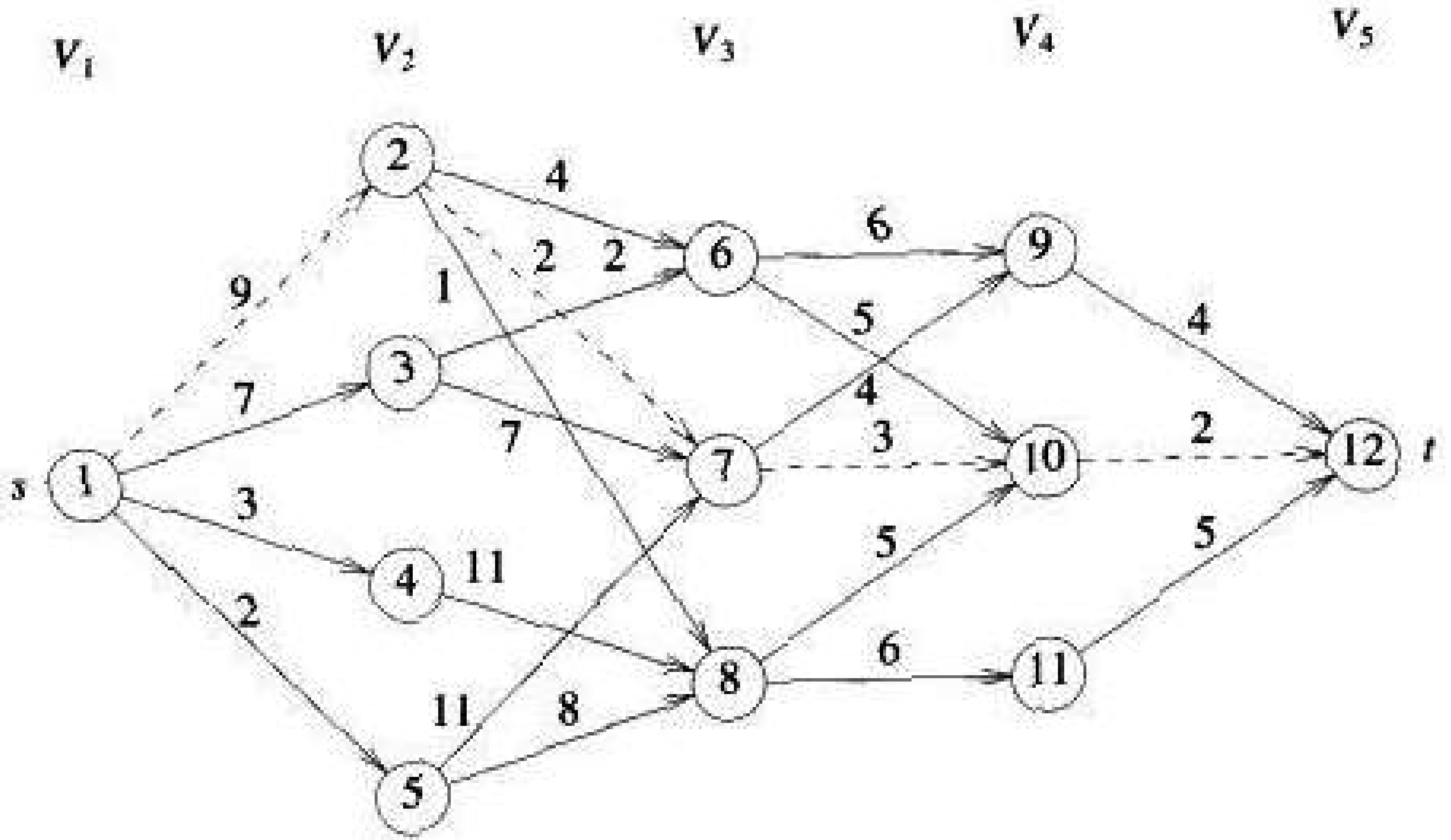
- Each set Vi defines a **stage** in the graph.

- Let ‘s’ and ‘t’ be vertices in V1 and Vk respectively. The vertex **‘s’** is called as a **source** and vertex **‘t’** is called as a **sink**. For any edge **<i, j>, i Є stage Vk** and **j Є stage Vk+1,** 1<= i < k.

- Let c(i, j) be the cost of an edge <i, j>. The cost of a path from ‘s’ to ‘t’ is the sum of the costs of the edges on the path

- The multi-stage graph problem is to **find a minimum cost path from ‘s’ to ‘t’.**

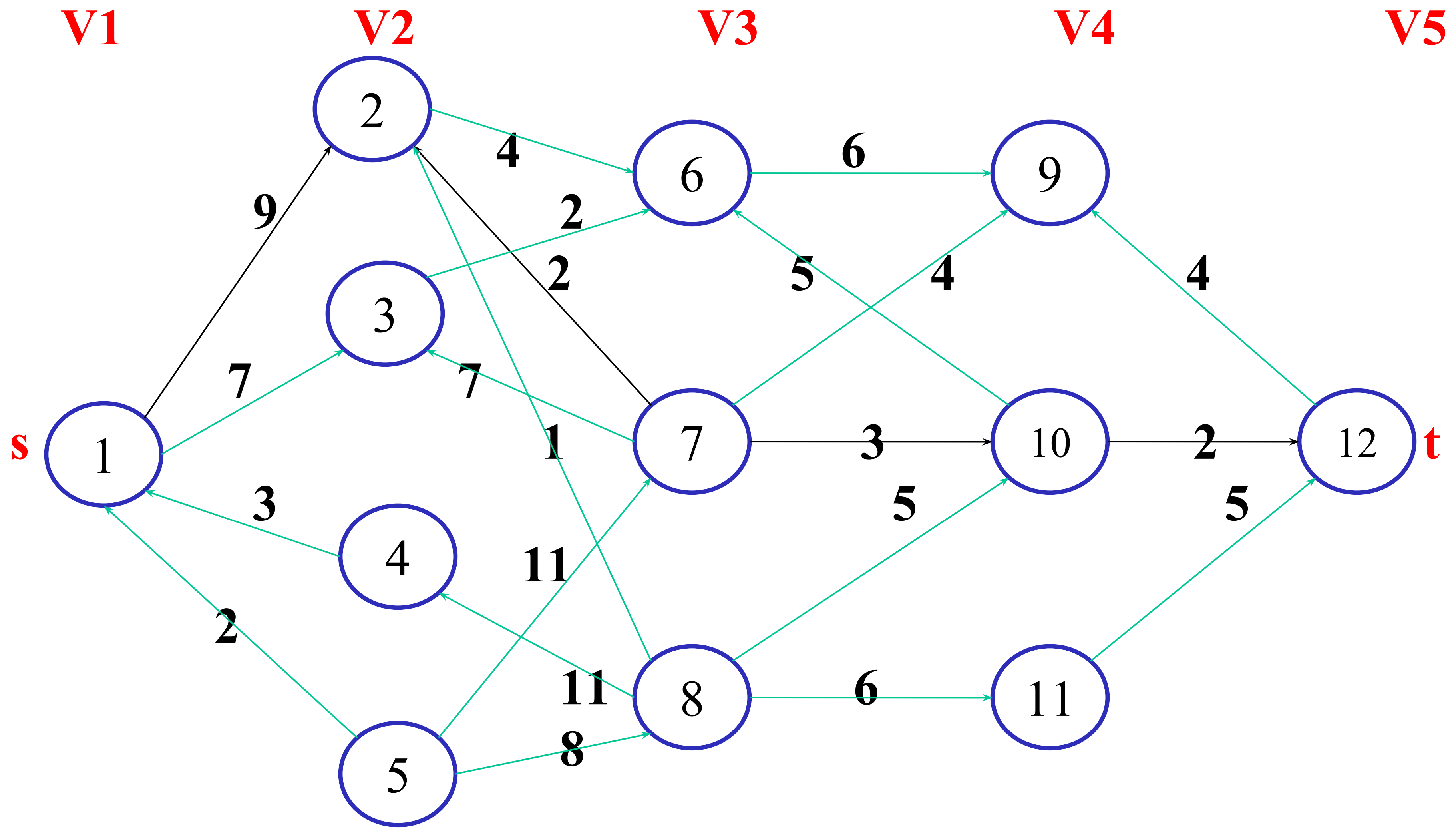
- Because of a constraint on edges, every path from ‘s’ to ‘t’ starts in stage 1, goes to stag 2, then to stage 3 and so on and eventually terminates in stage k.



V_1
V_2
V_3
V_4
V_5
s
t
1
2
3
4
5
6
7
8
9
10
11
12
9
7
3
2
4
2
1
7
11
11
8
2
6
5
4
3
5
6
4
2
5

Unit II : Dynamic Programming

Multi-stage Graphs : Problem Definition (Diagram)



Five-stage Graph

Unit II : Dynamic Programming

Multi-stage Graphs :

- This problem can be solved by using Dijkstra's algorithm in time $O(n^2)$, but we can DP approach to solve this problem in time **$\theta(|V| + |E|)$.**

Dynamic Programming Approach

- DP formulation for a k stage graph problem is obtained from (k – 2) sequence of decision to get shortest path from 's' to 't'.
- The ith decision involves determining which vertex in Vi , 1<=i<=(k – 2), is to be on the path. Hence POO applies.
- Let p(i, j) = minimum cost path from vertex j in Vi to vertex 't'

cost (i, j) = the cost of the path p(i, j).

c(j, *l*) = the cost of edge from vertex j in Vi & vertex *l* in Vi+1

- **cost (i, j) = min{c(j, *l*) + cost(i+1, *l*)}**
  _l_ Є Vi+1 and < j, _l_ > Є E

Unit II : Dynamic Programming

Multi-stage Graphs : Solution by DP

Forward Approach :

- $\text{cost}(i, j) = \min\{c(j, l) + \text{cost}(i+1, l)\}$

  $l \in V_{i+1}$ and $< j, l > \in E$

Backward Approach :

- $\text{bcost}(i, j) = \min\{c(l, j) + \text{bcost}(i-1, l)\}$

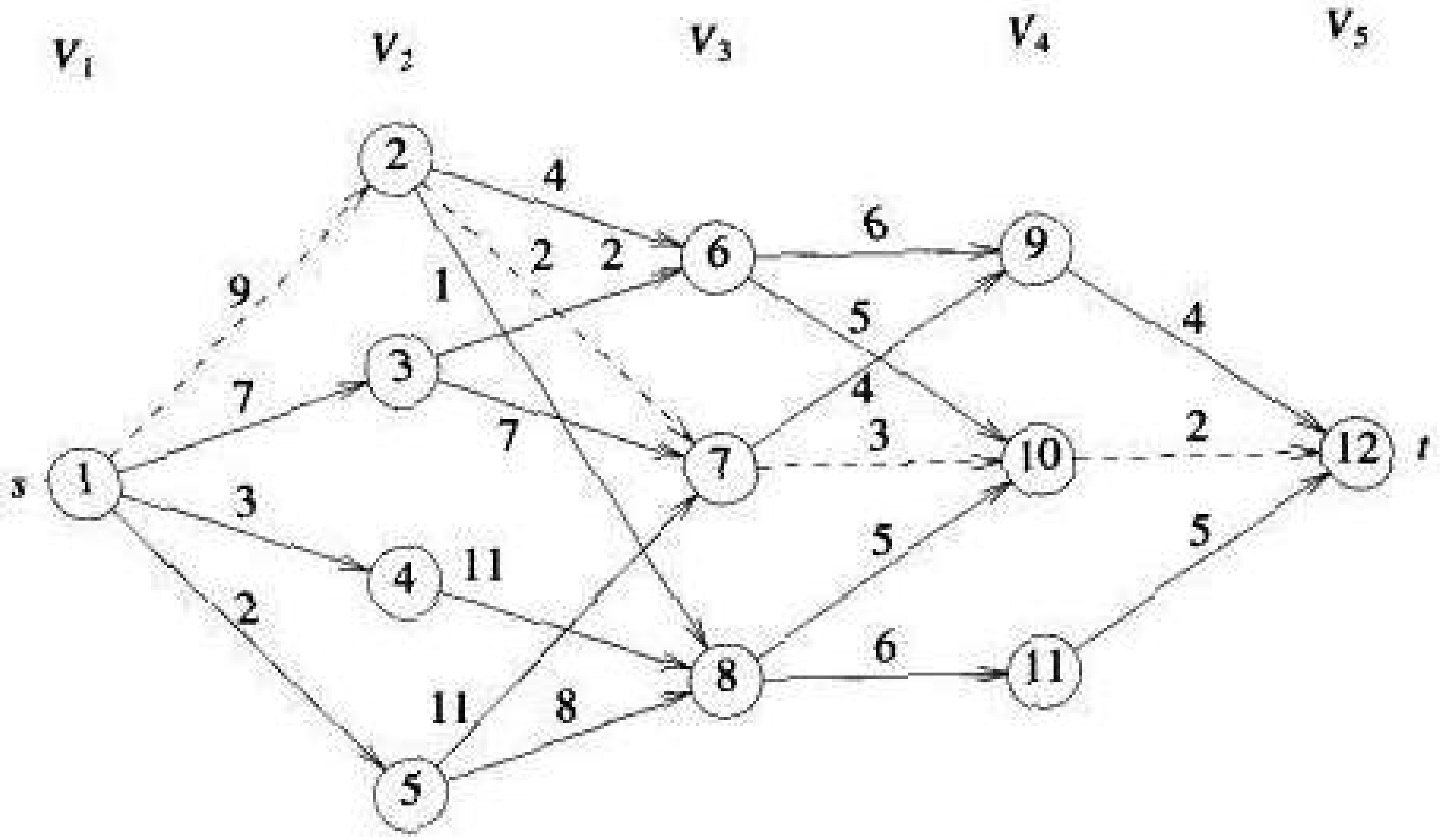
  $l \in V_{i-1}$ and $< l, j > \in E$

Example : Find shortest path from s to t for the above multi-stage graph using forward and backward approach.

Analysis (Fgraph and Bgraph):

Time Complexity = $\theta(|V| + |E|)$

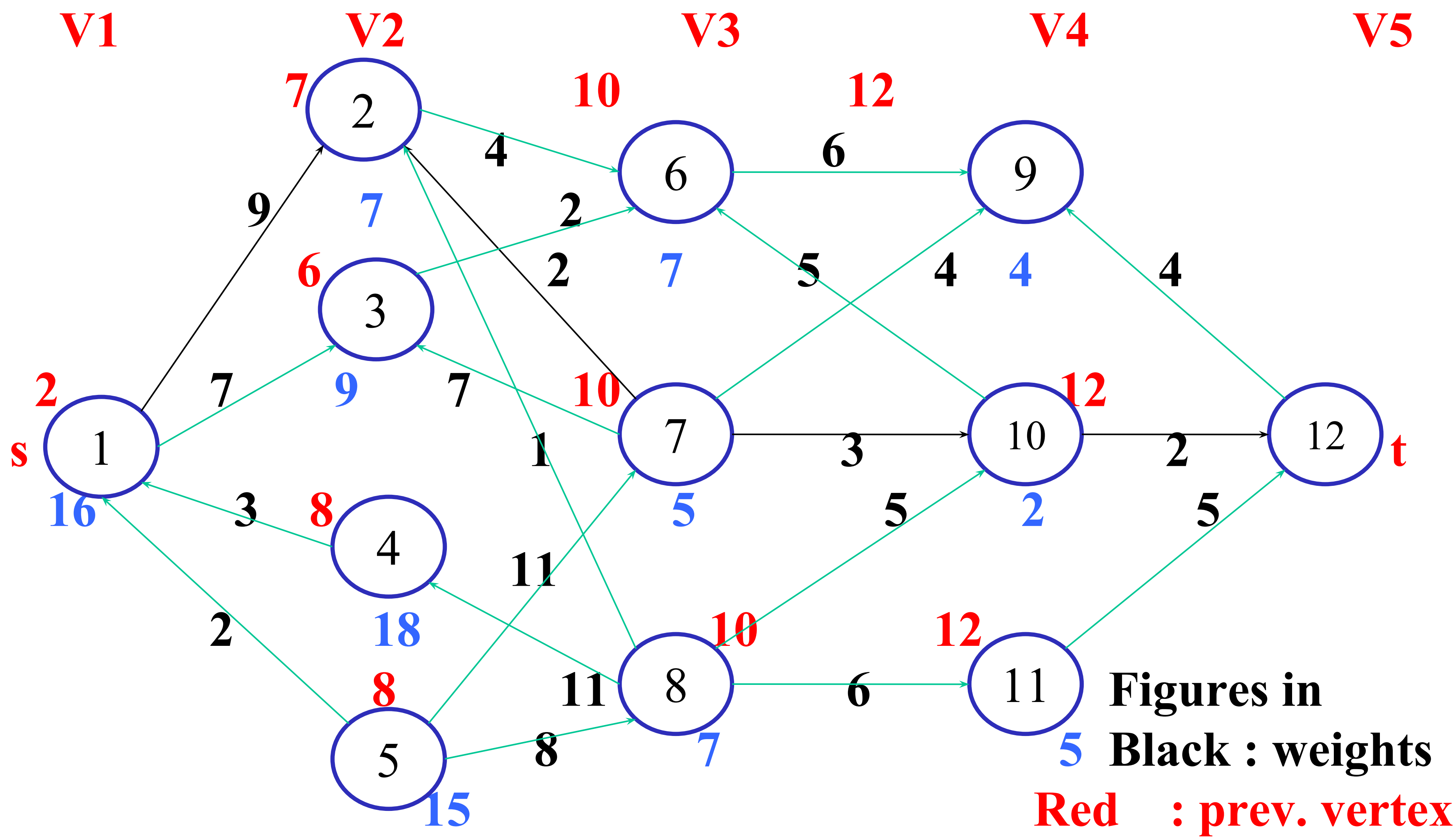
Space Complexity = $\theta(n+k)$ … k is number of stages.



V_1
V_2
V_3
V_4
V_5
s
t
1
2
3
4
5
6
7
8
9
10
11
12
9
7
3
2
4
2
2
1
7
11
11
8
6
5
4
3
5
6
4
2
5

Unit II : Dynamic Programming

Multi-stage Graphs : Solution by Forward Approach



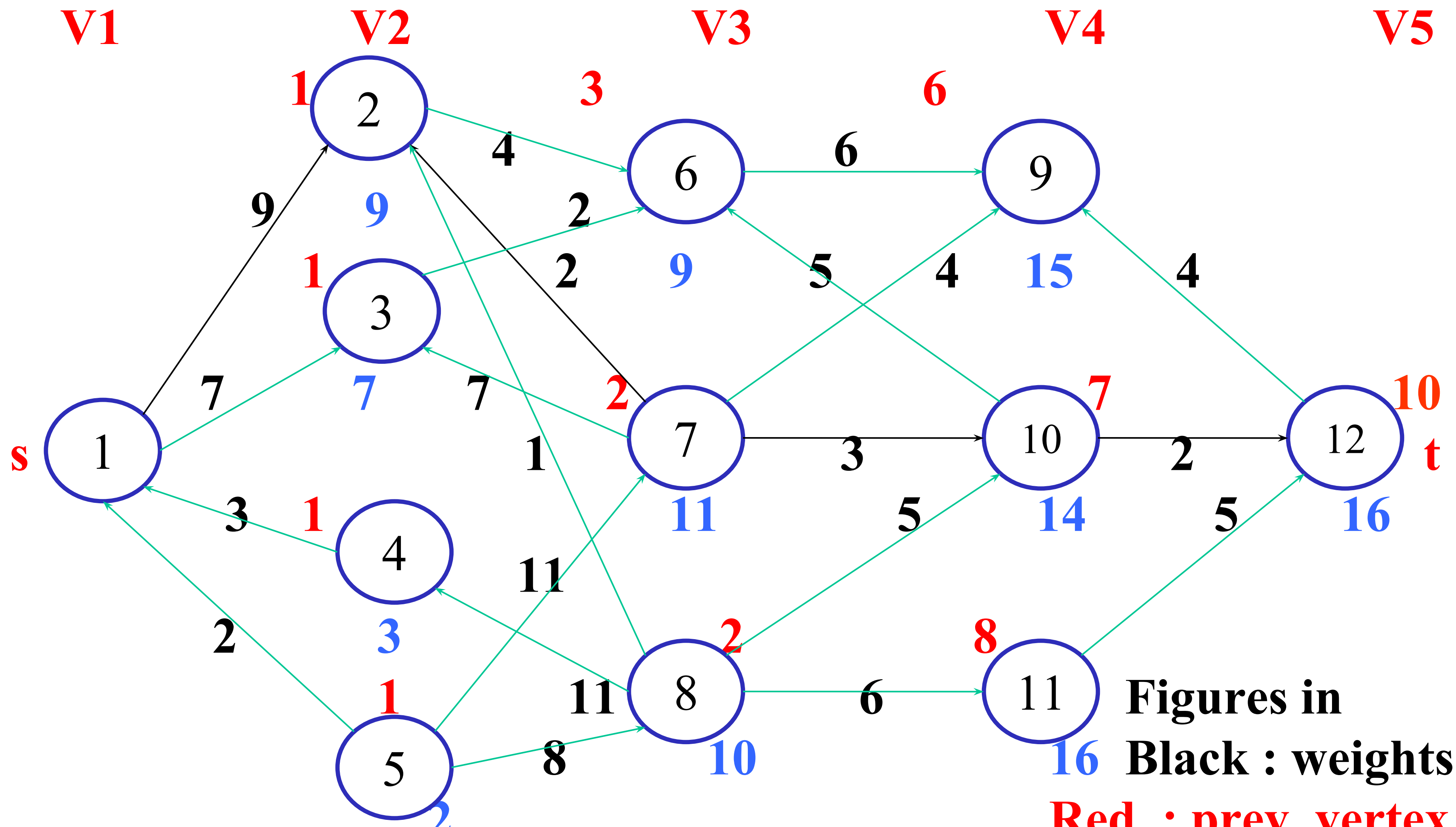
Figures in Black : weights
Red : prev. vertex
Blue : path weight up to t (Min)

Shortest Path : 1 – 2 – 7 – 10 – 12 (length = 16)

Five-stage Graph

Unit II : Dynamic Programming

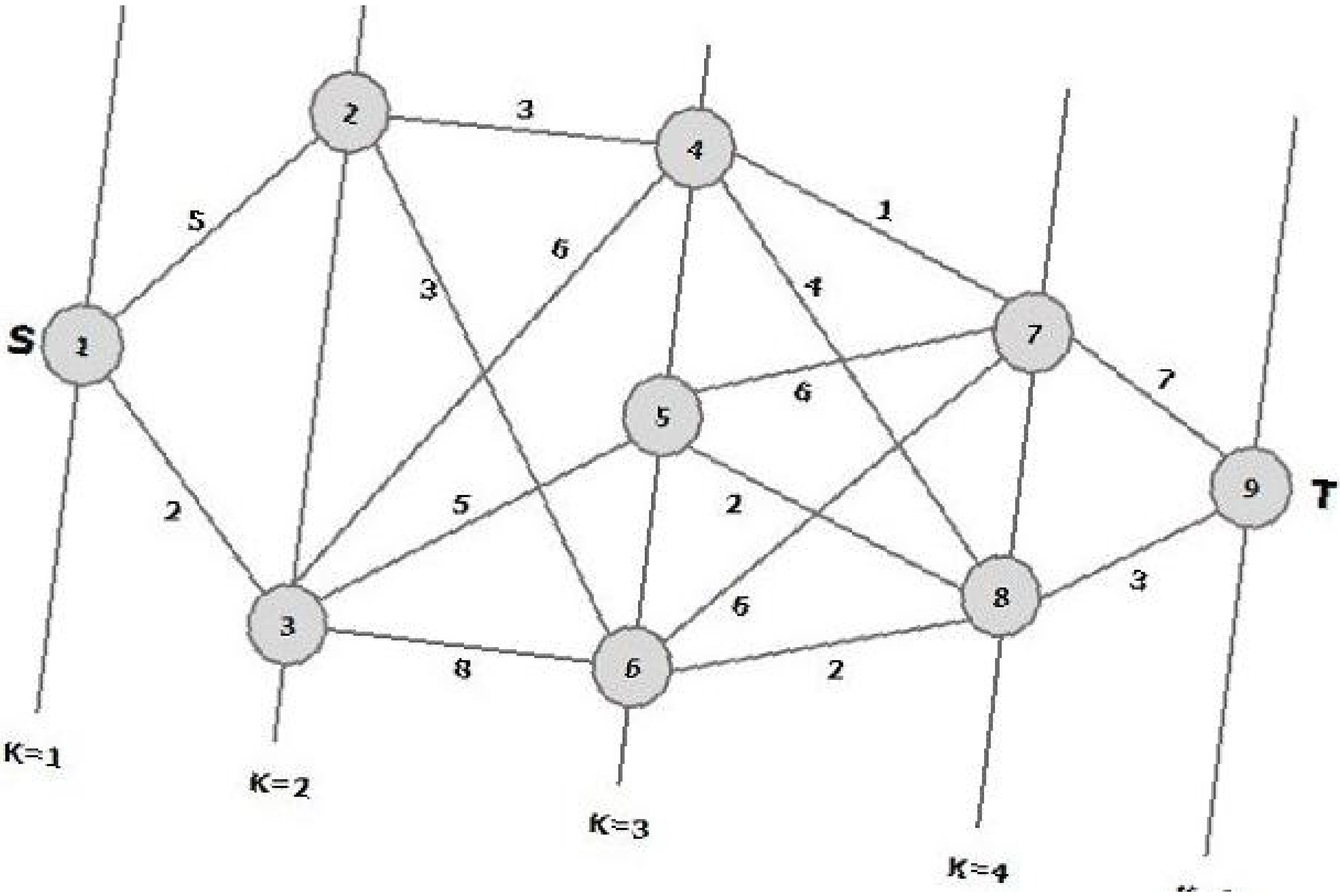
Multi-stage Graphs : Solution by Backward Approach



Figures in
Black : weights
Red : prev. vertex
Blue : path weight up to s (Min)

Shortest Path : 1 – 2 – 7 – 10 – 12 (length = 16)

Five-stage Graph


S
1
2
3
4
5
6
7
8
9
T
5
2
3
3
6
1
4
6
7
5
2
6
3
8
2
K=1
K=2
K=3
K=4

ence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

Pseudocode for forward approach

```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j\rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n] := 0.0$;
    for $j := n - 1$ to 1 step $-1$ do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r\rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]]$;
}
```

Pseudocode for Backward approach

```
**Algorithm** BGraph($G, k, n, p$)
// Same function as FGraph
{
    $bcost[1] := 0.0$;
    **for** $j := 2$ **to** $n$ **do**
    { // Compute $bcost[j]$.
        Let $r$ be such that $\langle r, j \rangle$ is an edge of
        $G$ and $bcost[r] + c[r, j]$ is minimum;
        $bcost[j] := bcost[r] + c[r, j]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    **for** $j := k - 1$ **to** 2 **do** $p[j] := d[p[j + 1]]$;
}
```

Travelling Salesperson Problem

Dynamic Programming

Traveling Salesperson Problem (TSP) : Problem Definition
We have seen that 0-1-knapsack problem is an example of subset selection where we have to select a subset from **2n** subsets. Now we shall see how to solve permutation problem where we have to select a permutation from **n!** permutations. Permutation problems are hard to solve as compared to subset problems since **n! >> 2n** for large value of n.

TSP Problem : Let G = (V, E) be a directed graph with edge costs cij for the edge <i, j> Є E. If there is no edge between i and j then cij = ∞ . Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of the tour is the sum of the cost of edges on the tour. TSP is to find a tour of minimum cost.

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Real life Applications :

To route a postal van to pick up mail from mail boxes located at ‘n’ different sites. Find the route of minimum distance.

To use a robot arm to tighten the nuts on some piece of machinery on an assembly line. Time to move arm from one position to other differs. Find the optimal sequence with minimum time.

To manufacture several commodities on the same set of machines. Changeover time from one commodity to other differs. Find the minimum schedule.

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Solution :

- Without loss of generality we shall regard a tour to be a simple path that starts at vertex 1 and ends at vertex 1.
- It consists of an edge <1, k> for some k Є V – {1} and the path from vertex k to vertex 1 which goes through each vertex in V – {1} exactly once.
- For optimal tour, the path from k to 1 must be a shortest path going through all vertices in V – {1, k}.
- Let us denote g(i, S) be the length of a shortest path starting at vertex i, going through all vertices in S and termination at vertex 1.
- Therefore g(1, V-{1}) is the length of optimal salesperson tour.
- Thus Principle of optimality holds good here.

Optimal Tour $= g(1, V-\{1\}) = \min_{2<=k<=n} \{c1k + g(k, V-\{1, k\}\}$ … (1)

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Solution :

- Generalizing above equation we get,

replacing 1 by i, v – {1} by S and k by j we get,

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(k, S - \{ j\}\} \qquad \ldots (2)$$

- Equation (2) can be used to find all values of g for | S| = 0, |S| = 1, |S| = 2, |S| = 3, and so on … until |S| = n – 2,

with the knowledge that g(i, Ǿ)= ci1, for 1<=i<=n

Dynamic Programming

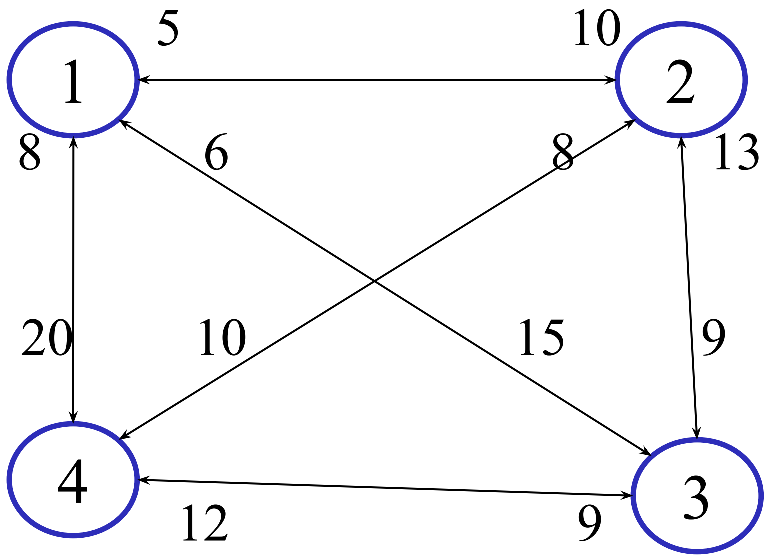
Traveling Salesperson Problem (TSP) :

Example : For the following directed graph represented as adjacency matrix (length of edges) find the optimal tour with shortest path length.

$$\begin{array}{c|cccc|}
 & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\
\mathbf{1} & \mathbf{010} & \mathbf{15} & \mathbf{20} & \\
\mathbf{2} & \mathbf{50} & \mathbf{9} & \mathbf{10} & \\
\mathbf{3} & \mathbf{6} & \mathbf{13} & \mathbf{0} & \mathbf{12} \\
\mathbf{4} & \mathbf{88} & \mathbf{9} & \mathbf{0} &
\end{array}$$

Graph G = (V,E)

V = {1, 2, 3, 4}



Dynamic Programming

Traveling Salesperson Problem (TSP) :

Example :

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 10 & 15 & 20 \\ 2 & 5 & 0 & 9 & 10 \\ 3 & 6 & 13 & 0 & 12 \\ 4 & 8 & 8 & 9 & 0 \end{array}$$

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \qquad (5.19)$$

Generalizing (5.19) we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \qquad (5.20)$$

$g(2, \phi) = c_{21} = 5$; $g(3, \phi) = c_{31} = 6$ and $g(4, \phi) = c_{41} = 8$.

Using (5.20) we obtain

$$\begin{array}{ll} g(2, \{3\}) = c_{23} + g(3, \phi) = 15; & g(2, \{4\}) = 18 \\ g(3, \{2\}) = 18; & g(3, \{4\}) = 20 \\ g(4, \{2\}) = 13; & g(4, \{3\}) = 15 \end{array}$$

Next, we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{aligned} g(2, \{3, 4\}) &= \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25 \\ g(3, \{2, 4\}) &= \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25 \\ g(4, \{2, 3\}) &= \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23 \end{aligned}$$

Finally, from (5.19) we obtain

$$\begin{aligned} g(1, \{2, 3, 4\}) &= \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

```
{{0, 22, 26, 30},
{30, 0, 45, 35},
{25, 45, 0, 60},
 {30, 35, 40, 0}}
```

Ans 120

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Analysis of algorithm tsp-dp :

- Let N = number of g(i, S) values to be computed
  = number of g(i, S) values for $|S| = 0$ +
  number of g(i, S) values for $|S| = 1$ +
  number of g(i, S) values for $|S| = 2$ +
  ………………………………………. +
  number of g(i, S) values for $|S| = n - 2$

  $= 3*1 + 3*2 + 3*1$ … for $n = 4$
  $= 3*C(2,0) + 3*C(2,1) + 3*C(2,2)$
  $= (4 - 1) * [C(2,0) + C(2,1) + C(2,2)]$
  $= (n - 1) * [C(n-2, 0) + C(n-2, 1) + … + C(n-2, n-2)]$ =

$(n - 1) * \Sigma C(n - 2, k)$ … **In General**
$0 <= k <= n-2$
$= (n - 1) * 2^{n-2}$

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Analysis of algorithm tsp-dp :

$N = (n - 1) * 2^{n-2}$

$= \mathbf{O(n^2\ 2^n)}$ = total number of g values

Time Complexity = $\mathbf{O(n^2 2^n)}$ better than **$\mathbf{O(n!)}$**

Suppose we have n = 10.

For O(n!), we have 10! = 3,628,800 operations.

For O(n^2 2^n), we have 10^2 * 2^10 = 102,400 operations.

Space Complexity = $\mathbf{O(n2^n)}$ to store g values

References :

1. Fundamentals of Computer Algorithms by Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajsekaran

2. https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/

3. https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-2/

Thank You!!

Extra Material

Real-Life Examples of OBST

(Only Sample problems to explore)

*Show how the optimal binary search tree problem is used in information retrieval systems such as **search engines**, **spell checkers**, and **natural language processing systems** to improve the efficiency and accuracy of these systems.*

*Highlight how the optimal binary search tree problem is used in **data compression algorithms** to reduce the size of data and improve the speed of data transmission and storage.*

*Use practical examples to illustrate how the optimal binary search tree problem can be used to model and solve problems in finance, economics, and accounting, such as **portfolio optimization** and **risk management**.*

*Showcase how the optimal binary search tree problem is used in **machine learning algorithms** to improve the performance of **classification** and **prediction models**.*

*Discuss how the optimal binary search tree problem is used in image processing and computer vision applications such as **object recognition**, **face detection**, and **motion tracking**.*

Real-Life Examples of OBST: Risk Management

Risk management involves identifying, analyzing, and minimizing the risks associated with financial investments.
The OBST can be used to evaluate the probability and potential impact of different risk scenarios.

For example, the OBST can be used to determine the optimal allocation of investments across different asset classes, based on their expected returns and their associated risks.

By constructing an optimal binary search tree, risk managers can minimize the expected cost of managing risk, while maximizing the potential return on investment.

Real-Life Examples of OBST: Portfolio Management

Portfolio optimization involves constructing a portfolio of assets that maximizes returns while minimizing risk.

The OBST can be used to determine the optimal asset allocation in a portfolio by minimizing the expected cost of searching for assets.

In this context, the cost of searching is the time and money spent in gathering and analyzing information about each asset.

The OBST can help in organizing the assets in the most efficient way and thus, minimize the expected cost of searching for an asset in the portfolio.

OBST in AI-ML

In neural networks, the cost function used for training the network can be optimized by constructing an OBST that represents the various decisions that must be made during the training process.
By minimizing the expected cost of traversing the tree, the training algorithm can be made more efficient and effective, which improves the overall performance of the neural network.

Constructing an OBST can help optimize the cost function used for training neural networks by identifying the most promising sets of parameters to evaluate during the training process.
Consider a neural network with 100,000 parameters.
During training, the cost function needs to be evaluated for every possible combination of these parameters, which is an incredibly time-consuming process.
By constructing an OBST, we can reduce the number of combinations that need to be evaluated by identifying the most promising parameters early in the process.
To do this, we start by constructing a binary search tree with the parameters as the keys.
We assign probabilities to each key based on how likely it is to lead to a good solution.
We can then use the probabilities to compute the expected cost of traversing the tree.
By minimizing the expected cost, we can identify the most promising set of parameters to evaluate next.
As the training process continues, the OBST is updated to reflect the new information gained during training.
This allows the **cost function to be optimized more efficiently** , leading to faster and more effective training.

Control Abstraction for Greedy

```
procedure GREEDY(A,n)
   //A(1:n) contains the inputs//
   solution :=Φ  //initialize the solution to empty //
for i -:= 1 to n do
   x := SELECT(A)
   if FEASIBLE(solution, x) then
     solution := UNION(solution, x)
   endif
repeat
return (solution)
end GREEDY
```

Complexity: **O(n)**