

Presentation Topic

Design and Analysis of Algorithm

(AY 2022 – 23)

Department of Computer Engineering



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

**(An Autonomous Institute affiliated to Savitribai Phule Pune University)
Institute Accredited 'A' Grade by NAAC**

CSUA32201: Design and Analysis of Algorithms

TY-SEM-II

CSUA32201: Design and Analysis of Algorithms

| Teaching Scheme | Examination Scheme |
|------------------------|--|
| Credits: 4 | Continuous Evaluation(CE): 20 Marks |
| Lectures: 3 Hrs/week | In-Semester Examination(ISE): 30 Marks |
| Practical : 2 Hrs/week | Skills & Competency Exam(SCE): 20 Marks |
| | End Semester Examination(ESE): 30 Marks |
| | PR/OR: 25 Marks |

Prerequisites :

Discrete Mathematics

Data Structures

Theory of Computation

Course Objectives :

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**

| Course Outcomes: | |
|-------------------------|--|
| | After completion of the course, student will be able to |
| 1. | Analyze algorithms for their time and space complexities in terms of asymptotic performance. |
| 2. | Apply greedy and dynamic programming algorithmic strategies to solve a given problem |
| 3. | Apply backtracking and branch and bound algorithmic strategies to solve a given problem |
| 4 | Identify intractable problems using concept of NP-Completeness |
| 5 | Use advance algorithms to solve intractable problems |
| 6 | Solve problems in parallel and distributed scenarios |

Unit I Introduction

Analysis of Algorithms, Best, Average and Worst case running times of algorithms, Mathematical notations for running times O , Ω , Θ , Master's Theorem

Problem solving principles: Classification of problem, problem solving strategies, classification of time complexities (linear, logarithmic etc.)

Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

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
III**

Backtracking, Branch and Bound

Backtracking: The General Method

8 Queen's problem,

Graph Coloring

Branch and Bound: 0/1 Knapsack, Traveling Salesperson Problem.

**Unit
IV**

Intractable Problems and NP-Completeness

Time-Space trade off, Tractable and Non-tractable Problems, Polynomial and non-polynomial problems, deterministic and non-deterministic algorithms P-class problems, NP-class of problems, Polynomial problem reduction, NP complete problems- Vertex cover and 3-SAT and NP hard problem - Hamiltonian cycle

Unit V Approximation and Randomized Algorithms, Natural Algorithms

Approximation algorithms, Solving TSP by approximation algorithm, approximating Max CliqueConcept of randomized algorithms, randomized quicksort algorithms , Natural Algorithms–Evolutionary Computing and Evolutionary Algorithms, Introduction to Genetic Algorithm, Simulated Annealing

Unit VI Parallel and Concurrent Algorithms

Parallel Algorithms: Sequential and parallel computing, RAM&PRAM models, Amdahl's Law, Brent's theorem, parallel algorithm analysis, multithreaded matrix multiplication, Concurrent Algorithms: Dining philosophers problem

Text Books :

Gilles Brassard, Paul Bratley, "Fundamentals of Algorithmics", PHI, ISBN 978-81-203- 1131-2

Horowitz and Sahani, "Fundamentals of Computer Algorithms", University Press, ISBN: 978 81 7371 6126, 81 7371 61262

Reference Books :

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", MIT Press; ISBN 978-0-262-03384-8

Parag Himanshu Dave, Himanshu Bhalchandra Dave, "Design And Analysis of Algorithms", Pearson Education, ISBN 81-7758-595-9

Rajeev Motwani and Prabhakar Raghavan, "Randomized Algorithms", Cambridge University Press, ISBN: 978-0-521-61390-3

Michael T. Goodrich, Roberto Tamassia , "Algorithm Design: Foundations, Analysis and Internet Examples", Wiley, ISBN 978-81-265-0986-7

Dan Gusfield, "Algorithms on Strings, Trees and Sequences", Cambridge University Press,ISBN:0-521- 7035-7

Perform the following lab assignments using C++/Java/Python

1. Implement Quick Sort using divide and conquer strategy.
2. Implement 0/1 knapsack using Dynamic Programming.
3. Implement 8 queens problem using Backtracking
4. Implement Travelling Salesman problem using branch and bound technique.
5. Implement Travelling Salesman problem using Genetic Algorithm
6. Implement Concurrent Dining Philosopher Problem.
7. Implement multithreaded matrix multiplication.

Objectives & Outcomes of Unit 1

- **Objective :**

Measure the performance of algorithms on the basis of time and space complexity.

- **Outcome :**

Analyze algorithms for their time and space complexities in terms of asymptotic performance

Unit I: Introduction

What is - Design & Analysis of Algorithm

- An **algorithm** is systematic method containing sequence of Instruction to solve a computational problem.
- It takes inputs, performs a well defined sequence steps and produces output.
- Once we **design** an algorithm we need to know how well it performs on any input.
- In particular we need to know whether there are any better algorithms for a problem.
- Hence we need to **analyze** an algorithm on the basis of **efficiency**.

What is efficiency ?

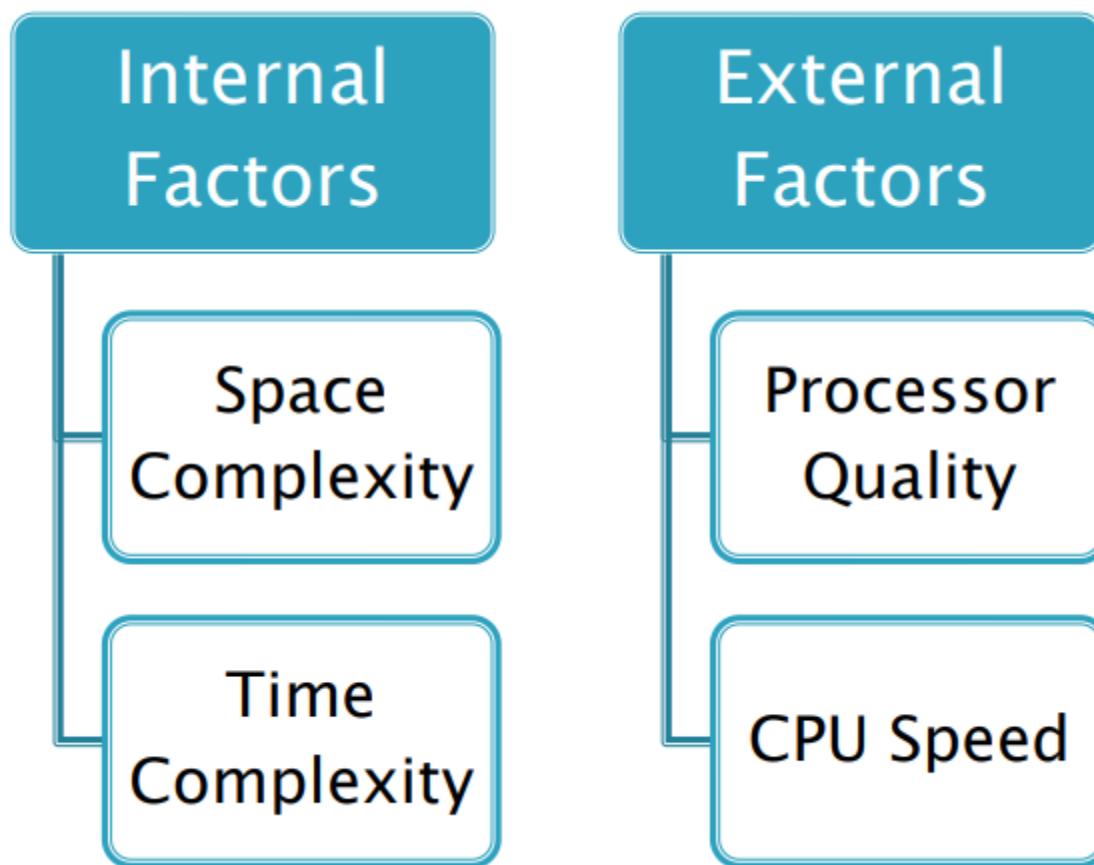
- The rate at which storage memory or time grows as a function of the input size is called efficiency.

- It is a measure to analyze the performance of a program code for a particular problem.

Why do we need Efficiency?

- Several algorithms to solve the same problem
- Which algorithm is “best” ?
- How much **time** does the algorithm require ?
- How much **space** (memory) does the algorithm occupy ?

Algorithm Complexity



Types of Efficiency

- Space Complexity
- Time Complexity
- In general, both depend on the input (typically the input, size of the input).

Space Complexity

- Space complexity is the amount of memory the program requires till its termination.
- It is a function of the size of the input.

Why Space Complexity?

- To know in advance space / memory requirement
- To specify the amount of memory if the program is to run on multi-user system
- Extremely important if the program has to run with limited resources or has to handle input of large size

Types of Space required for a program

- Instruction Space
- Data Space
- Stack Space

Time Complexity

- It is the amount of time required to execute the program
- It indicates the relationship between the running time of the algorithm and size of the input

Why Time Complexity?

- To estimate how long a program will run.
- To help focus on the parts of code that are executed the large number of times.
- To find an alternate solution.

Example

- Suppose Algorithm A has running time $7n^2$;
Algorithm B has running time $2n^3$.
Which algorithm is more efficient?
- Compare n^2 and n^3 for different values of n :

| | $n=10$ | $n=100$ | $n=10000$ | $n=100000$ |
|-------|----------------|----------------|-----------|------------|
| n^2 | 10^{-7} sec. | 10^{-5} sec. | 0.1 sec. | 10 sec. |
| n^3 | 10^{-6} sec. | 10^{-3} sec. | 17 min. | 11.6 days |

- For large n , *n^2 is much faster than n^3* ,
and the coefficients 7 and 2 doesn't make much difference.
- Thus, to evaluate the efficiency of an algorithm we need to identify the most important part in its running time. That important part is known as **algorithm order**.
- To define it formally, we need to introduce **Asymptotic Notation** .

Unit-1

- Analysis of Algorithms
- Best, Average and Worst case running times of algorithms,
- Mathematical notations for running times O , Ω , Θ ,
- Master's Theorem
- Problem solving principles: Classification of problem, problem solving strategies, classification of time complexities (linear, logarithmic etc.)
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

Analysis of Algorithms

Analysis of Algorithm =

- Prediction of how *fast* an algorithm runs for a given **PROBLEM SIZE**
 - **Analysis** based on Time Computations = **TIME COMPLEXITY**
- Prediction of *memory requirements* (primary memory) based on the **PROBLEM SIZE**
 - **Analysis of algorithms** based on Memory Requirements = **SPACE COMPLEXITY**

2 Considerations in Analysis

- TIME and SPACE
 - How much TIME is required to Execute the algorithm?
(Time Complexity)
 - How much SPACE is required to Execute the algorithm?
(Space Complexity)

How to Calculate Time (*fast*)?

Is **absolute time** to run the algorithm possible, because machines running the algorithm may be different, or the environment may be different?

Or something else?

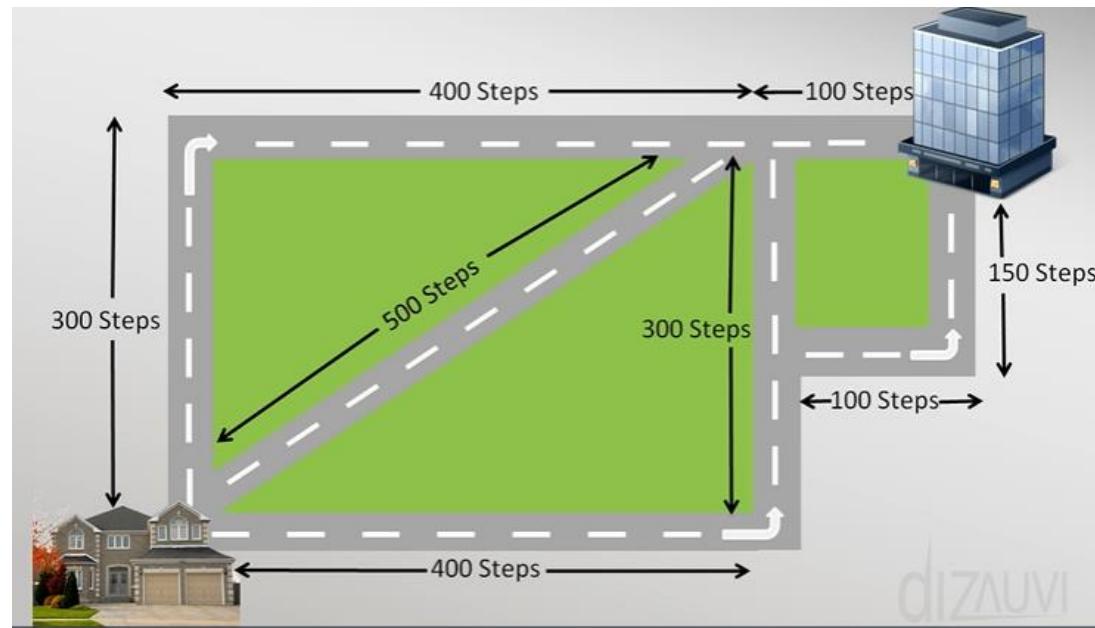


How to calculate time for going from home to college/office?

Should speed of walking or vehicle type/traffic conditions be considered?

Or something else?

Ans: Number of steps ? With assumption that step size is uniform.



PROBLEM SIZE

Types of problem size:-

- Number of **Inputs / Outputs**
 - E.g. For Sorting Algorithm :
 - Inputs = Total number of elements to be sorted
 - Output = Total number of sorted elements
- Number of **Operations Involved** in the algorithm
 - E.g. For Searching Algorithm :
 - Operations Involved = total number of Comparisons with search element

Summary

- Analysis of Algorithm aims at determination of Time Complexity and Space Complexity

Questions

- Define Analysis of Algorithm?
- What is meant by PROBLEM SIZE?

Unit-1

- Analysis of Algorithms
 - **Best, Average and Worst case running times of algorithms,**
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles:
 - Classification of problem,
 - problem solving strategies,
 - classification of time complexities (linear, logarithmic etc.)
-
- Divide and Conquer strategy:
 - General strategy,
 - Quick Sort and Merge Sort w.r.t. Complexity

Best, Average and Worst case running times of algorithms

Types of Time Complexity

Worst Case

Best Case

Average Case

Types of Time Complexity

Worst Case ✓

Best Case

Average Case ✗

Sample – Best, Worst, Average Case - Search for number x in an array

- Best case: First number in array equals x
Time effort: let's say 1 unit
- Worst case: the last number (n th entry) in array equals x or is not found
Time effort: n units
- Average case: ? $(1+2+3+4+\dots+n)/n$ units = $n(n+1)/n$ units

Assumptions to Calculate Time Complexity

RAM Model of Computation

-  We have infinite memory.
-  Each operation(+,-,*,/,=) takes unit time.
-  For each memory access, unit time is consumed.
-  Data may be accessed from RAM or disk, it is assumed that the data is in the RAM.

Step Count for Time Complexity

Step Count for Time Complexity

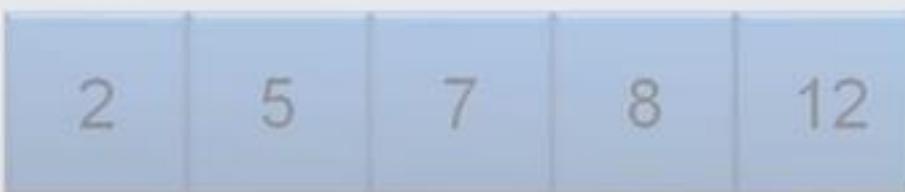
| | Frequency | Total |
|------------------------------------|----------------|-----------------------|
| Algorithm Add_arr (A,B,C,n) | 0 | - |
| { | 0 | - |
| for i:= 1 to n | n+1 | n+1 |
| for j:=1 to n | n(n+1) | n ² +n |
| C[i,j]=B[i,j]+A[i,j]; | n(n) | n ² |
| } | | |
| | TOTAL T(n)= | 2n ² +2n+1 |

The time complexity of Algorithm **Add_arr**, with array size 'n' =

$$T(n) = 2n^2 + 2n + 1 = \mathbf{O}(n^2)$$

Time Complexity - Bubble Sort

Bubble Sort



Pseudo-code :

```
for i = 0 to A.length-2  
    for j = 0 to A.length-2-i  
        if A[j] > A[j+1]  
            tmp = A[j+1]  
            A[j+1] = A[j]  
            A[j] = tmp
```

Time Complexity - Bubble Sort

Bubble Sort

| | | | | |
|---|---|---|---|----|
| 2 | 5 | 7 | 8 | 12 |
|---|---|---|---|----|

Pseudo-code :

```
for i = 0 to A.length-2
    for j = 0 to A.length-2-i
        if A[j] > A[j+1]
            tmp = A[j+1]
            A[j+1] = A[j]
            A[j] = tmp
```

```
for (int i=0; i < A.length-1; i++) {
    for (int j=0; j < A.length-1-i; j++) {
        if (A[j] > A[j+1]) {
            //do the swap as shown
        }
    }
}
```

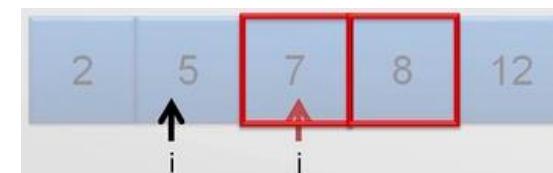
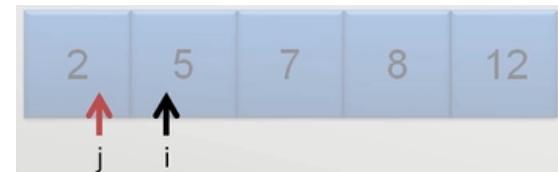
Time Complexity - Bubble Sort

Bubble Sort

2 5 7 8 12

Pseudo-code :

```
for i = 0 to A.length-2  
  for j = 0 to A.length-2-i  
    if A[j] > A[j+1]  
      tmp = A [j+1]  
      A[j+1] = A [j]  
      A[j] = tmp
```



Step Count for Time Complexity

| | Frequency | Total |
|------------------------------------|----------------------------|--|
| Algorithm Bubble (A,length) | 0 | - |
| { | 0 | - |
| for i = 0 to length-2 | length-1 | length-1 |
| for j:= 0 to length-2-i | length-1, length-2, ..., 1 | length(length-1)/2 |
| if (A[j]>A[j+1]) | length-1, length-2, ..., 1 | length(length-1)/2 |
| swap(A[i],A[j]); | length-1, length-2, ..., 1 | 3(length(length-1)/2) |
| } | | |
| | TOTAL T(length)= | (length-1)+5(length(length-1)/2) =5/2(length) ² – 3/2length -1 |

The time complexity of Algorithm Bubble array size length = 'n' =

$$T(n) = 5/2n^2 - 3/2n - 1 = \mathbf{O}(n^2) \text{ (why?)}$$

Time Complexity - Bubble Sort

Bubble Sort

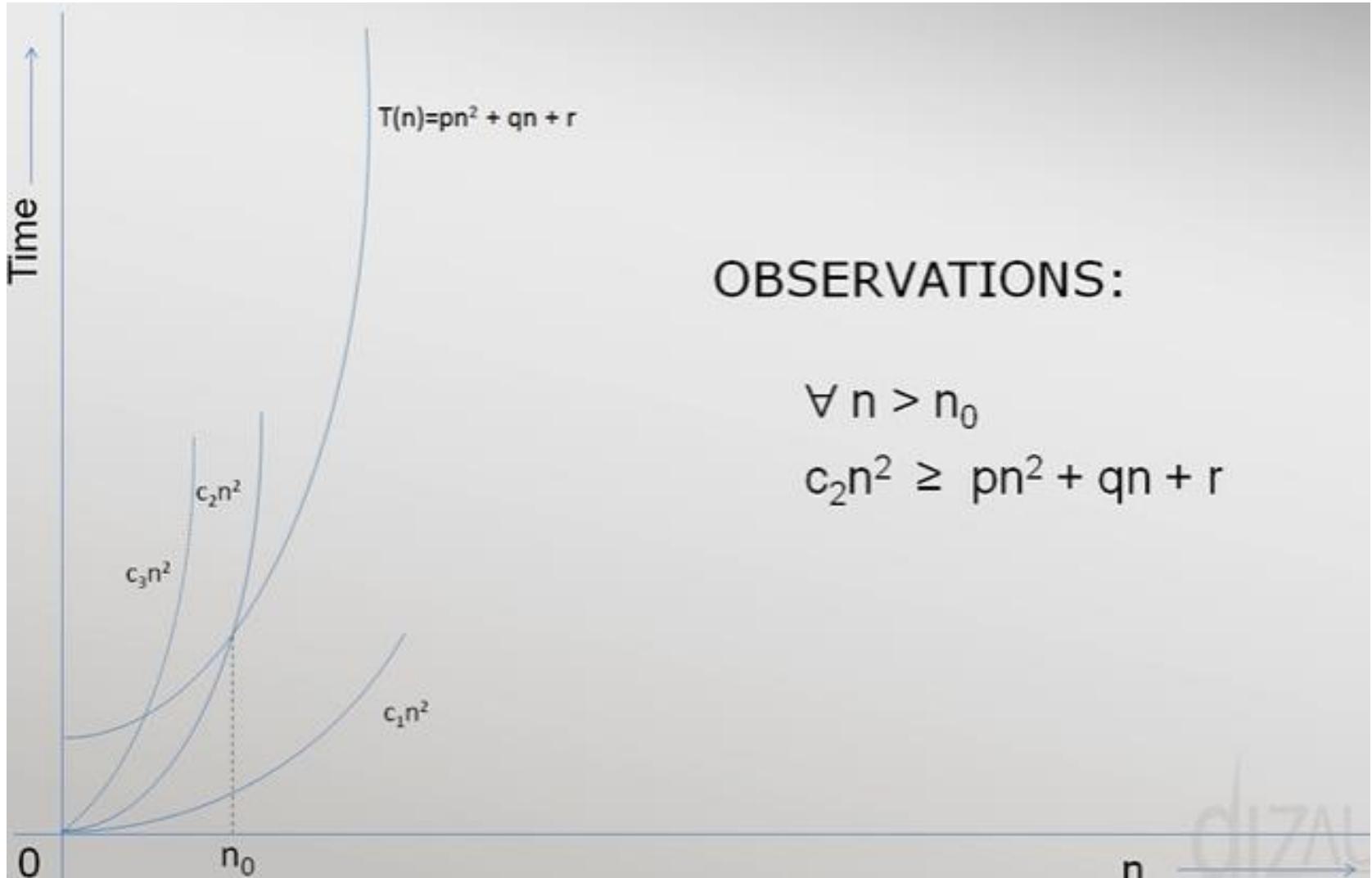
```
2 5 7 8 12
```

Pseudo-code :

```
for i = 0 to A.length-2
    for j = 0 to A.length-2-i
        if A[j] > A[j+1]
            tmp = A[j+1]
            A[j+1] = A[j]
            A[j] = tmp
```

| | | |
|---------|-------------|-------|
| i = 0 | 0 ≤ j < n-2 | (n-1) |
| i = 1 | 0 ≤ j < n-3 | (n-2) |
| : | : | : |
| i = n-2 | 0 ≤ j < 0 | 1 |

Mathematics of Time Complexity



Mathematics of Time Complexity

Big Oh

$O(n^2) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cn^2 \text{ for all } n \geq n_0$

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0$

Big ‘O’ Notation

An algorithm is said to run in $O(f(n))$ time if for some constants c and n_0 ,
the time taken by the algorithm is at most $cf(n)$ for all problem instances
with size $n \geq n_0$

Big ‘O’ Notation

$$5n^2 + 6 \in O(n^2)$$

$$cn^2 \quad c = 6 \quad n_0 = 3$$

$$c = 5.1 \quad n_0 = 8$$

$$5n + 6 \in O(n^2)$$

$$cn^2 \quad c = 11 \quad n_0 = 1$$

$$n^3 + 2n^2 + 4n + 8 \in O(n^2) \quad ??$$

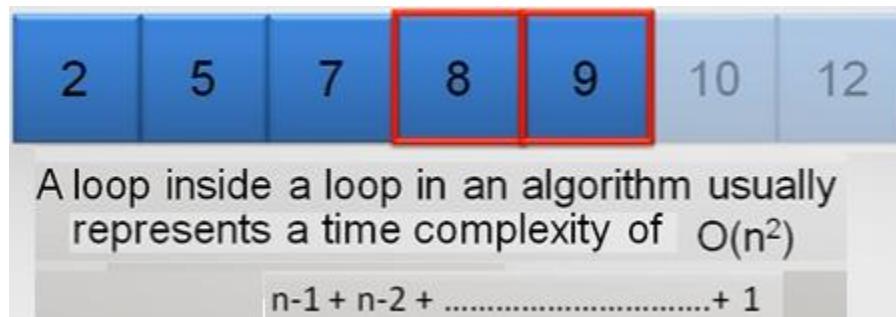
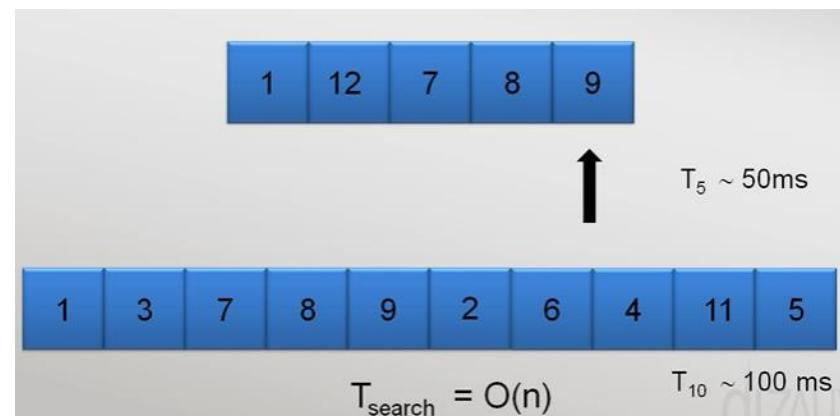
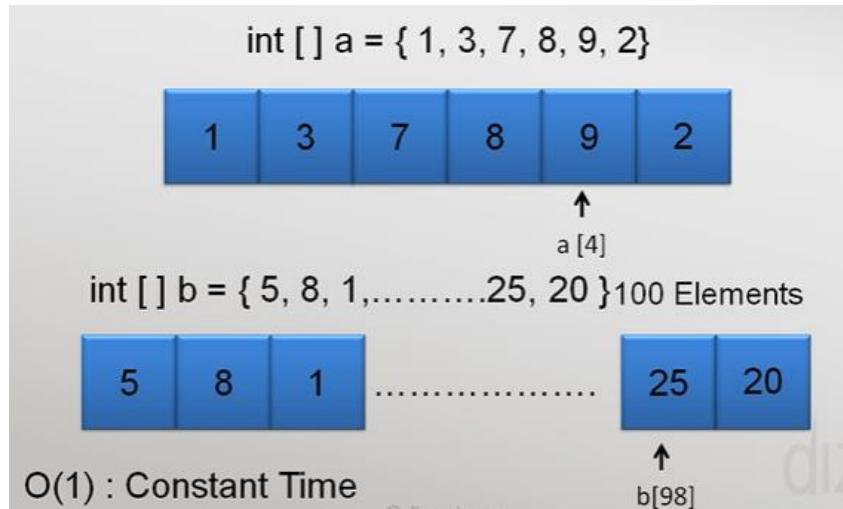
$$cn^2 \geq n^3 + 2n^2 + 4n + 8 \quad x$$

General Thumb Rule for Big 'O'

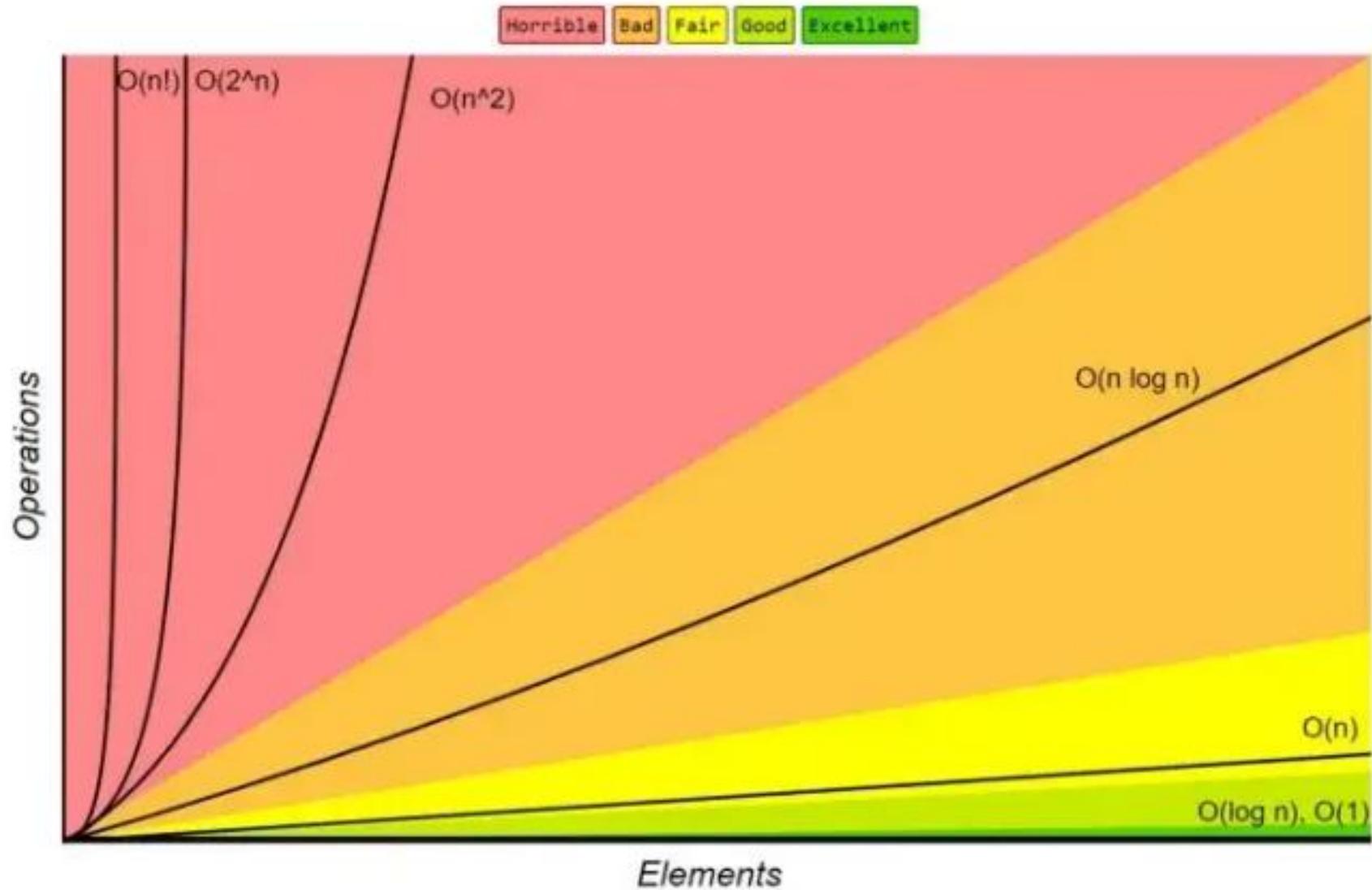
$$a_m n^m + a_{m-1} n^{m-1} - \dots - + a_0 \in O(n^m)$$

$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

Meaning of O(1), O(n), O(n^2)



Big 'O' Complexity



Big-O Complexity Chart: <http://bigocheatsheet.com/>

Time comparisons of the common algorithm orders

| $f(n)$ | $n=10$ | $n=1000$ | $n=10^5$ | $n=10^7$ |
|--------------------|------------------------------|-----------------------------------|----------------------------------|------------------------------------|
| $\log_2 n$ | 3.3×10^{-9} sec. | 10^{-8} sec. | 1.7×10^{-8} sec. | 2.3×10^{-8} sec. |
| n | 10^{-8} sec. | 10^{-6} sec. | 10^{-4} sec. | 0.01 sec. |
| $n \cdot \log_2 n$ | 3.3×10^{-8} sec. | 10^{-5} sec. | 0.0017 sec. | 0.23 sec. |
| n^2 | 10^{-7} sec. | 10^{-3} sec. | 10 sec. | 27.8 min. |
| n^3 | 10^{-6} sec. | 1 sec. | 11.6 min. | 317 cent. |
| 2^n | 10^{-6} sec. | 3.4×10^{28} 4 years | 3.2×10^{30095} years | 3.1×10^{3001022} years |

Recap Questions

- What are the 3 kinds of time complexities?
- Define Big ‘Oh’
- Sort the following in increasing time complexities :

Linear time – $O(n)$, Logarithmic time – $O(\log n)$, Constant time – $O(1)$, Cubic time – $O(n^3)$, Quadratic time – $O(n^2)$

Q)

Normally we are not interested in the best case running times of algorithms, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time

T/F?

Q)

Normally we are not interested in the best case running times of algorithms, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time

Ans: True

Q)

- What is the need for worst case time complexity?

Q)

- What is the need for worst case time complexity?

Ans

The advantage of - Analyzing the worst case running times of an algorithm - is that you know **for certain** that the algorithm must perform at least that well.

This is especially important for **real-time applications**, such as for the computers that monitor an air traffic control system.

Here, it would not be acceptable to use an algorithm that can handle airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all airplanes are coming from the same direction

Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - **Mathematical notations for running times O, Ω, Θ,**
 - Master's Theorem
-
- Problem solving principles:
 - Classification of problem,
 - problem solving strategies,
 - classification of time complexities (linear, logarithmic etc.)
-
- Divide and Conquer strategy:
 - General strategy,
 - Quick Sort and Merge Sort w.r.t. Complexity

The Asymptotic Notations – O , Θ , Ω

Mathematics of Time Complexity

Asymptotic Notations

- Asymptotic notations are used to describe time and space complexity i.e. to **measure efficiency with respect to problem size**
 - Indicates behavior of a function with respect to the size of the input (n).
- Different asymptotic notations used are:-
 1. Big Oh Notation (O)
 2. Omega Notation (Ω)
 3. Theta Notation (Θ)

Big Oh

$O(n^2) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cn^2 \text{ for all } n \geq n_0$

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0$

If a function is **$O(n)$** , it is automatically **$O(n^2)$** as well.

Big Oh 'O'

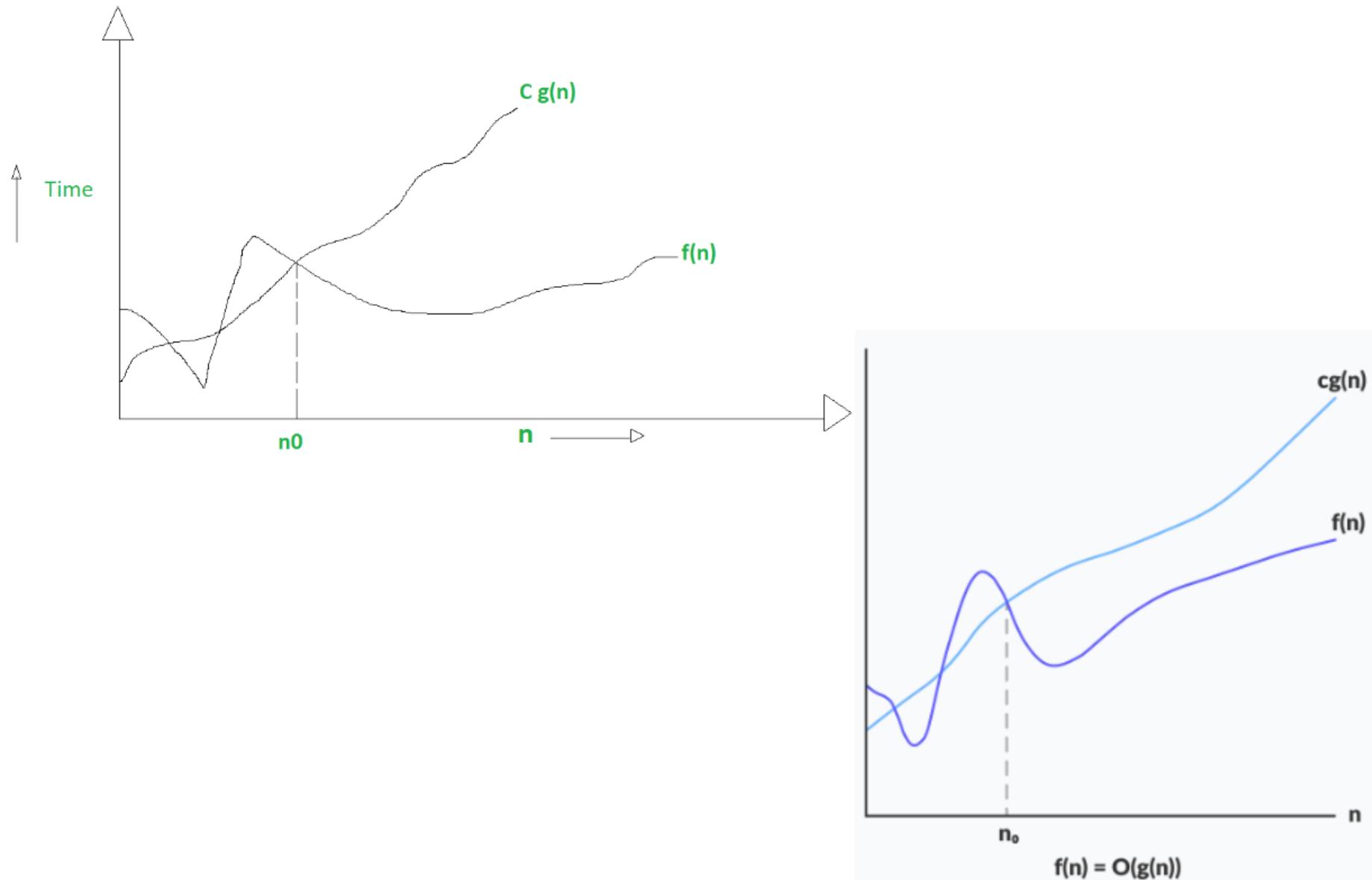
f(n) defines running time of an algorithm

f(n) is said to be **O(g (n))** if there exists positive constant **C** and **(n₀)** such that

$$0 \leq f(n) \leq Cg(n) \quad \text{for all } n \geq n_0$$

If a function is **O(n)** it is automatically **O(nlogn), O(n²), O(n³)** as well

Graphical example for Big Oh (O)



Sample Examples

| | |
|---------------------------|---|
| $3n+2 = O(n)$ | Because $3n+2 \leq 4n$, for $n \geq 2$ |
| $3n+3 = O(n)$ | Because $3n+3 \leq 4n$, for $n \geq 3$ |
| $100n+6 = O(n)$ | Because $100n+6 \leq 101n$, for $n \geq 6$ |
| $10n^2+4n+2 = O(n^2)$ | Because $10n^2+4n+2 \leq 11n^2$, for $n \geq 5$ |
| $1000n^2+100n-6 = O(n^2)$ | Because $1000n^2+100n-6 \leq 1001n^2$, for $n \geq 100$ |
| $6*2^n + n^2 = O(2^n)$ | $6*2^n + n^2 \leq 7(2^n)$, for $n \geq 4$ |
| $3n+2 = O(n^2)$ | Because $3n+2 \leq 3n^2$, for $n \geq 2$ |
| $10n^2+4n+2 = O(n^4)$ | Because $10n^2+4n+2 \leq 10n^4$, for $n \geq 2$ |
| $3n^2+2 \neq O(n)$ | Because $3n^2+2$ is NOT less than or equal to any cn for all $n \geq n_0$ |
| $10n+2 \neq O(1)$ | Because $10n+2$ is NOT less than or equal to any c for all $n \geq n_0$ |

Big Omega notation (Ω)

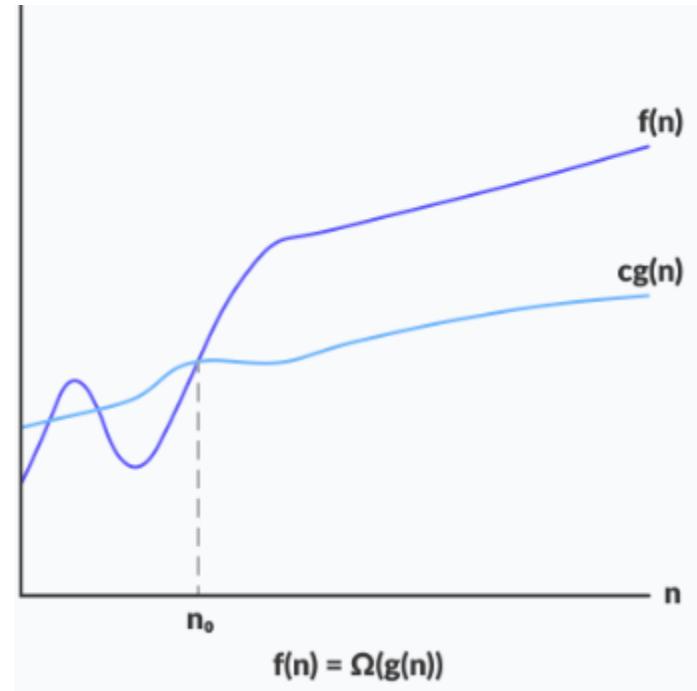
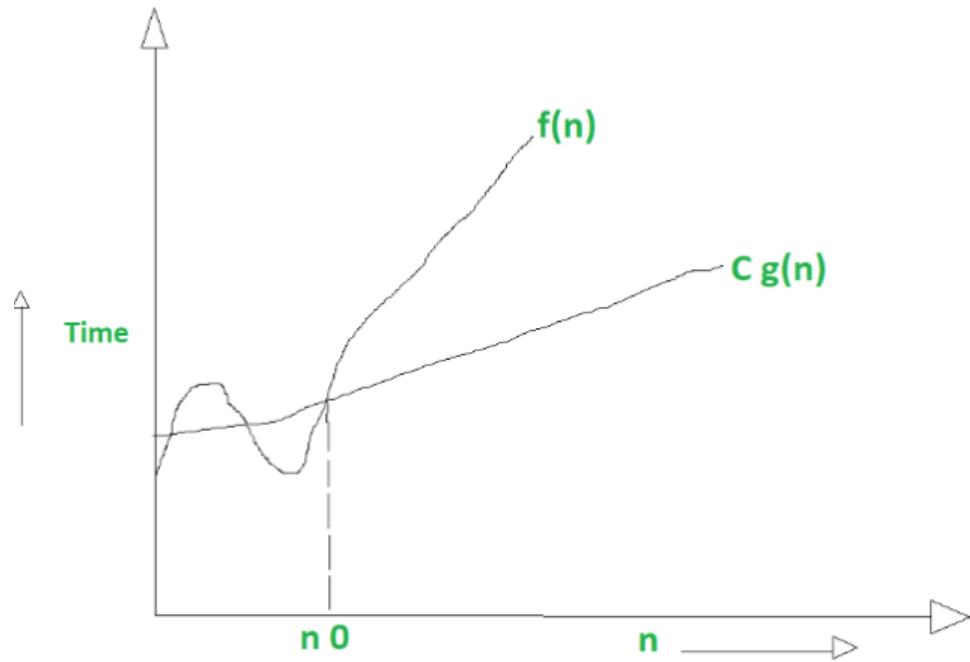
$f(n)$ defines running time of an algorithm

$f(n)$ is said to be $\Omega(g(n))$ if there exists positive constant C and (n_0) such that

$$0 \leq Cg(n) \leq f(n) \text{ for all } n \geq n_0$$

If a function is $\Omega(n^2)$ it is automatically $\Omega(n)$, $\Omega(1)$ as well

Graphical example for Big Omega (Ω)



Sample Examples

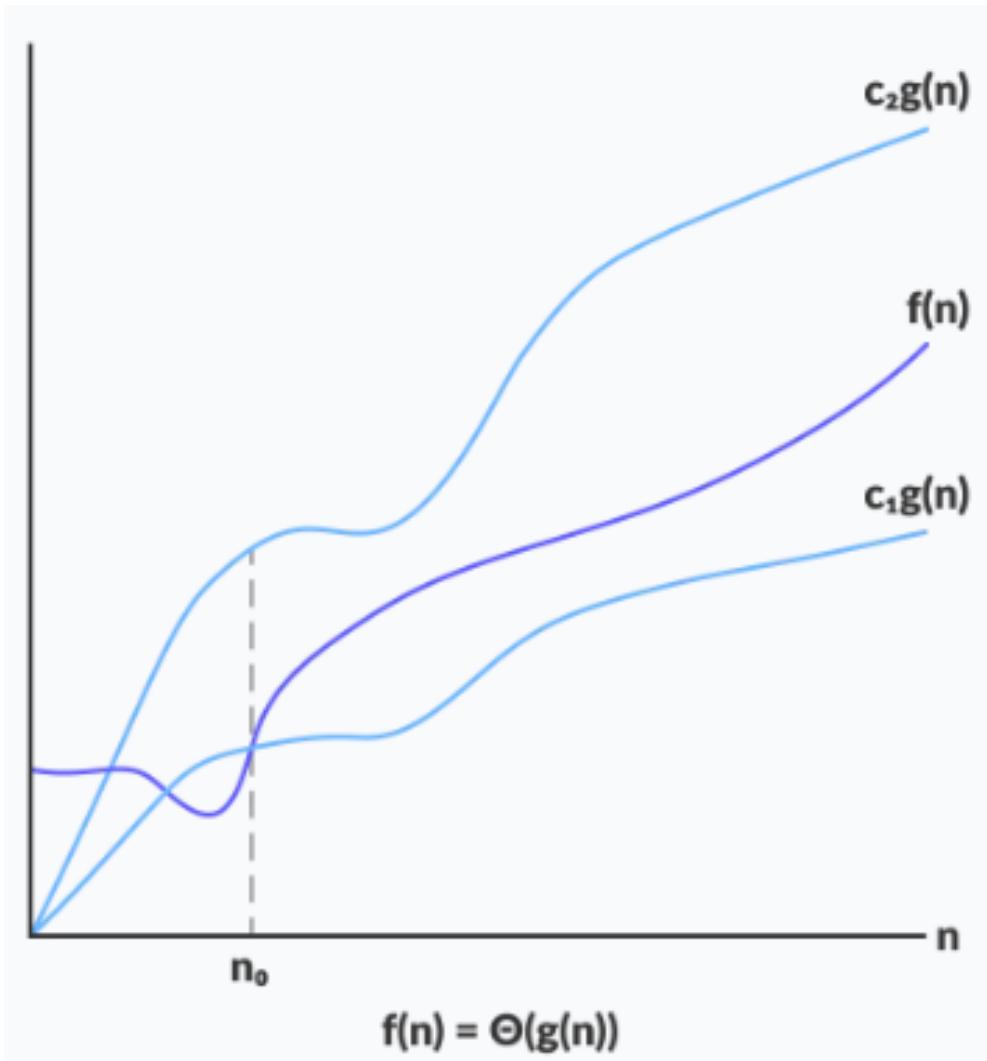
| | |
|--------------------------------|---|
| $3n+2 = \Omega(n)$ | Because $3n+2 \geq 3n$, for $n \geq 1$ |
| $3n+3 = \Omega(n)$ | Because $3n+3 \geq 3n$, for $n \geq 1$ |
| $100n+6 = \Omega(n)$ | Because $100n+6 \geq 100n$, for $n \geq 1$ |
| $10n^2+4n+2 = \Omega(n^2)$ | Because $10n^2+4n+2 \geq 10n^2$, for $n \geq 1$ |
| $1000n^2+100n-6 = \Omega(n^2)$ | Because $1000n^2+100n-6 \geq 1000n^2$, for $n \geq 1$ |
| $6*2^n + n^2 = \Omega(2^n)$ | $6*2^n + n^2 \geq 6(2^n)$, for $n \geq 1$ |
| $3n+2 = \Omega(1)$ | Because $3n+2 \geq 3$, for $n \geq 1$ |
| $10n^2+4n+2 = \Omega(n)$ | Because $10n^2+4n+2 \geq 10n$, for $n \geq 1$ |
| $3n^2+2 \neq \Omega(n^4)$ | Because $3n^2+2$ is NOT greater than or equal to any cn^4 for all $n \geq n_0$ |
| $10n+2 \neq \Omega(n^2)$ | Because $10n+2$ is NOT greater than or equal to any cn^2 for all $n \geq n_0$ |

Big Theta notation (Θ)

It is define as tightest bound

- and tightest bound is the best of all the worst case times that the algorithm can take.
- Let $f(n)$ define running time of an algorithm.
 $f(n)$ is said to
be $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Graphical representation



Example

- Consider $f(n) = 3n + 2$

$$f(n) \geq 1n \Rightarrow c_1 = 1, n_0 \geq 1 \text{ i.e. } f(n) = \Omega(n)$$

$$f(n) \leq 5n \Rightarrow c_2 = 5, n_0 \geq 1 \text{ i.e. } f(n) = O(g(n))$$

$$1n \leq f(n) \leq 5n$$

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for } n \geq n_0$$

Hence $f(n) = \Theta(n)$

Sample Examples

| | | | |
|---------------------------|---|--------------------------------|--|
| $3n+2 = O(n)$ | Because $3n+2 \leq 4n$, for $n \geq 2$ | $3n+2 = \Omega(n)$ | Because $3n+2 \geq 3n$, for $n \geq 1$ |
| $3n+3 = O(n)$ | Because $3n+3 \leq 4n$, for $n \geq 3$ | $3n+3 = \Omega(n)$ | Because $3n+3 \geq 3n$, for $n \geq 1$ |
| $100n+6 = O(n)$ | Because $100n+6 \leq 101n$, for $n \geq 6$ | $100n+6 = \Omega(n)$ | Because $100n+6 \geq 100n$, for $n \geq 1$ |
| $10n^2+4n+2 = O(n^2)$ | Because $10n^2+4n+2 \leq 11n^2$, for $n \geq 5$ | $10n^2+4n+2 = \Omega(n^2)$ | Because $10n^2+4n+2 \geq 10n^2$, for $n \geq 1$ |
| $1000n^2+100n-6 = O(n^2)$ | Because $1000n^2+100n-6 \leq 1001n^2$, for $n \geq 100$ | $1000n^2+100n-6 = \Omega(n^2)$ | Because $1000n^2+100n-6 \geq 1000n^2$, for $n \geq 1$ |
| $6*2^n + n^2 = O(2^n)$ | $6*2^n + n^2 \leq 7(2^n)$, for $n \geq 4$ | $6*2^n + n^2 = \Omega(2^n)$ | $6*2^n + n^2 \geq 6(2^n)$, for $n \geq 1$ |
| $3n+2 = O(n^2)$ | Because $3n+2 \leq 3n^2$, for $n \geq 2$ | $3n+2 = \Omega(1)$ | Because $3n+2 \geq 3$, for $n \geq 1$ |
| $10n^2+4n+2 = O(n^4)$ | Because $10n^2+4n+2 \leq 10n^4$, for $n \geq 2$ | $10n^2+4n+2 = \Omega(n)$ | Because $10n^2+4n+2 \geq 10n$, for $n \geq 1$ |
| $3n^2+2 \neq O(n)$ | Because $3n^2+2$ is NOT less than or equal to any cn for all $n \geq n_0$ | $3n^2+2 \neq \Omega(n^4)$ | Because $3n^2+2$ is NOT greater than or equal to any cn^4 for all $n \geq n_0$ |
| $10n+2 \neq O(1)$ | Because $10n+2$ is NOT less than or equal to any c for all $n \geq n_0$ | $10n+2 \neq \Omega(n^2)$ | Because $10n+2$ is NOT greater than or equal to any cn^2 for all $n \geq n_0$ |

Sample Examples

| | |
|--------------------------------|---|
| $3n+2 = \Theta(n)$ | Because $3n \leq 3n+2 \leq 4n$, for $n \geq 2$ |
| $3n+3 = \Theta(n)$ | Because $3n \leq 3n+3 \leq 4n$, for $n \geq 3$ |
| $100n+6 = \Theta(n)$ | Because $100n \leq 100n+6 \leq 101n$, for $n \geq 6$ |
| $10n^2+4n+2 = \Theta(n^2)$ | Because $10n^2 \leq 10n^2+4n+2 \leq 11n^2$, for $n \geq 5$ |
| $1000n^2+100n-6 = \Theta(n^2)$ | Because $1000n^2 \leq 1000n^2+100n-6 \leq 1001n^2$, for $n \geq 100$ |
| $6*2^n + n^2 = \Theta(2^n)$ | Because $6*2^n \leq 6*2^n + n^2 \leq 7(2^n)$, for $n \geq 4$ |

Mathematics of Time Complexity

- with Asymptotic Notations O , Θ ,
 Ω

Use of Asymptotic Notations

Asymptotic Analysis for Algorithms

- The asymptotic behavior of a function $f(n)$ refers to the growth of $f(n)$ as n gets large.
- We typically ignore small values of n , since we are usually interested in estimating how slow the program will be on large inputs.

Big ‘Oh’ notation (O) for TC

- *Big ‘Oh’ notation represents the upper bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.*

Big Omega notation (Ω) for TC

- *Omega notation represents the lower bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.*
 - For any value of n , the minimum time required by the algorithm is given by Omega $\Omega(g(n))$

Theta notation (Θ) for TC

- *Theta notation encloses the function from above and below*
- *Since it represents the upper and the lower bound of the running time of an algorithm, it gives a tight bound of the time complexity of an algorithm*
- *It is used for analyzing the **average-case** complexity of an algorithm.*

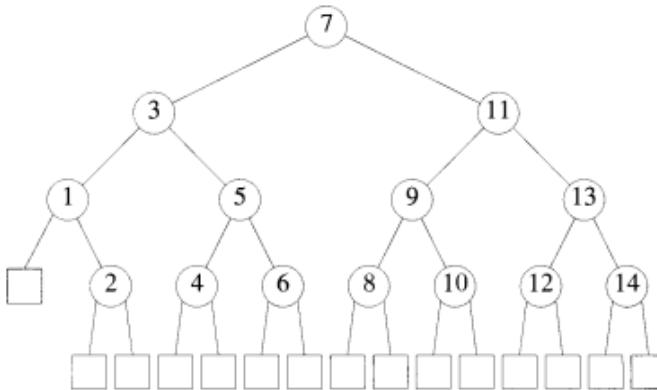
Q) Select the correct statement

- A) Asymptotic notations give correct, meaningful and exact running times of an algorithm
- B) Asymptotic notations give correct, meaningful but in-exact running times of an algorithm

Q) Select the correct statement

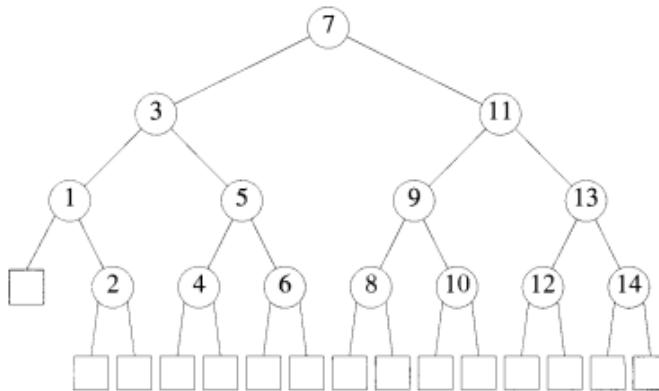
- A) Asymptotic notations give correct, meaningful and exact running times of an algorithm
- B) Asymptotic notations give correct, meaningful but in-exact running times of an algorithm

Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct? [Ans: All])



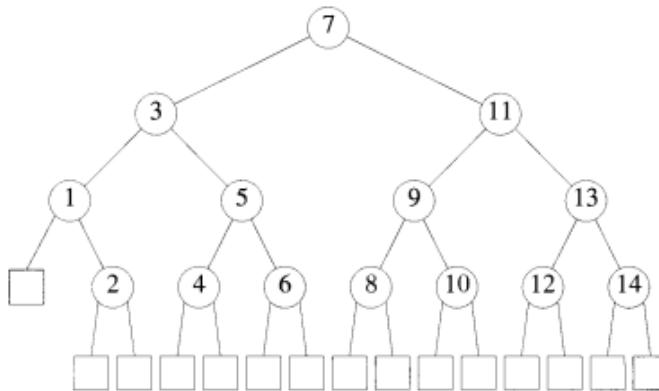
| | | | | | |
|-----------------------|-------------------------------|---------------------------|----------------------------|-------------------------------|---------------------------|
| Successful Searches | | | Unsuccessful Searches | | |
| O(1), Best | O(log n), Average | O(log n) Worst | O(log n), Best | O(log n), Average | O(log n) Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Omega(1)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$ Worst | $\Omega(\log n)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$ Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Theta(1)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$ Worst | $\Theta(\log n)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$ Worst |

Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct?)



| Successful Searches | | | Unsuccessful Searches | | |
|-----------------------|-------------------------------|-----------------------------|----------------------------|-------------------------------|-----------------------------|
| $O(1)$, Best | $O(\log n)$, Average | $O(\log n)$, Worst | $O(\log n)$, Best | $O(\log n)$, Average | $O(\log n)$, Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Omega(1)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$, Worst | $\Omega(\log n)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$, Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Theta(1)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$, Worst | $\Theta(\log n)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$, Worst |

Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct? – Ans: All are correct)



| | | | | | |
|-----------------------|-------------------------------|---------------------------|----------------------------|-------------------------------|---------------------------|
| Successful Searches | | | Unsuccessful Searches | | |
| O(1), Best | O(log n), Average | O(log n) Worst | O(log n), Best | O(log n), Average | O(log n) Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Omega(1)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$ Worst | $\Omega(\log n)$, Best | $\Omega(\log n)$, Average | $\Omega(\log n)$ Worst |
| Successful Searches | | | Unsuccessful Searches | | |
| $\Theta(1)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$ Worst | $\Theta(\log n)$, Best | $\Theta(\log n)$, Average | $\Theta(\log n)$ Worst |

Obtaining Time Complexities of Recursive Functions

```
1  Algorithm DAndC( $P$ )
2  {
3      if Small( $P$ ) then return S( $P$ );
4      else
5      {
6          divide  $P$  into smaller instances  $P_1, P_2, \dots, P_k$ ,  $k \geq 1$ ;
7          Apply DAndC to each of these subproblems;
8          return Combine(DAndC( $P_1$ ),DAndC( $P_2$ ),...,DAndC( $P_k$ ));
9      }
10 }
```

Control abstraction for divide-and-conquer

Recurrence Relation for Time Complexity (TC) of the General Algorithm DAndC

$$T(n) = \begin{cases} g(n) & \text{for } \text{SMALL } n \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{Otherwise} \end{cases}$$

Complexity of Many divide and conquer algorithms is

$$T(n) = \begin{cases} T(1) & \text{for } n = 1 \\ aT(n/b) + f(n) & \text{for } n > 1 \end{cases}$$

Where a, b are known constants.

We assume, for TC calculations, $T(1)$ is known and n is a power of b
i.e. $n = b^k$

Solve the recurrence relation for $a,b=2$
and $f(n)=n$ and $T(1)=7$

- $T(n) = \begin{cases} T(1) & \text{for } n = 1 \\ aT(n/b) + f(n) & \text{for } n > 1 \end{cases}$

Using SUBSTITUTION METHOD

Unit-1

- Analysis of Algorithms
- Best, Average and Worst case running times of algorithms,
- Mathematical notations for running times O , Ω , Θ ,
- **Master's Theorem**
- Problem solving principles: Classification of problem, problem solving strategies, classification of time complexities (linear, logarithmic etc.)
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

What is Master's Theorem?

- The master theorem always yields **asymptotically tight bounds** to recurrences from divide and conquer algorithms that partition an input into smaller subproblems of equal sizes, solve the subproblems recursively, and then combine the subproblem solutions to give a solution to the original problem.

..contd.. What is Master's Theorem?

- The time for such an algorithm can be expressed by **adding the work** that they perform at the top level of their recursion (to **divide** the problems into subproblems and then **combine** the subproblem solutions) together with the time made in the recursive calls of the algorithm
- This can be expressed by a recurrence relation that takes the form:

Master Method

- The master method provides a “cookbook” method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function

Master Theorem

$T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$ and $f(n)$ is a given positive function;

$$f(n) = \theta(n^k \log^p n)$$

case 1: if $\log_b a > k$, $\theta(n^{\log_b a})$

case 2: if $\log_b a = k$,

$$\text{if } p > -1, \quad \theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1, \quad \theta(n^k \log \log n)$$

$$\text{if } p < -1, \quad \theta(n^k)$$

case 3: if $\log_b a < k$,

$$\text{if } p \geq 0, \quad \theta(n^k \log^p n)$$

$$\text{if } p < 0, \quad O(n^k)$$

Master Theorem Example Case 1

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, f(n) = \theta(n^k \log^p n)$$

case 1: if $\log_b a > k$, $\theta(n^{\log_b a})$

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

$$a=2, b=2$$

$$f(n) = 1;$$

$$k=0$$

$$p=0$$

$$\log_b a = \log_2 2 = 1$$

$\log_b a > 0$ Hence Case 1

$$T(n) = \theta(n^{\log_b a})$$

$$T(n) = \theta(n^1)$$

Master Theorem Example Case 1

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, f(n) = \theta(n^k \log^p n)$$

case 1: if $\log_b a > k$, $\theta(n^{\log_b a})$

$$T(n) = 4 T(n/2) + n$$

$$a=4, b=2$$

$$f(n) = 1;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 4 = 2$$

$\log_b a > 1$ Hence Case 1

$$T(n) = \theta(n^{\log_b a})$$

$$T(n) = \theta(n^2)$$

Master Theorem Example Case 1

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, f(n) = \theta(n^k \log^p n)$$

case 1: if $\log_b a > k$, $\theta(n^{\log_b a})$

$$T(n) = 8T(n/2) + n$$

$$a=8, b=2$$

$$f(n) = n;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 8 = 3$$

$\log_b a > k$ Hence Case 1

$$T(n) = \theta(n^{\log_b a})$$

$$T(n) = \theta(n^3)$$

Master Theorem Case 2

$T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, $f(n) = \theta(n^k \log^p n)$

case 2: if $\log_b a = k$,

if $p > -1$, $\theta(n^k \log^{p+1} n)$

if $p = -1$, $\theta(n^k \log \log n)$

if $p < -1$, $\theta(n^k)$

$$T(n) = 2 T(n/2) + n$$

$$a=2, b=2$$

$$f(n) = n;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p>-1$$

$$T(n) = \theta(n^k \log^{p+1} n)$$

$$T(n) = \theta(n \log n)$$

Master Theorem Case 2

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, f(n) = \theta(n^k \log^p n)$$

case 2: if $\log_b a = k$,

$$\text{if } p > -1, \quad \theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1, \quad \theta(n^k \log \log n)$$

$$\text{if } p < -1, \quad \theta(n^k)$$

$$T(n) = 2T(n/2) + n/\log n$$

$$a=2, b=2$$

$$f(n) = n/\log n;$$

$$k=1$$

$$p=-1$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p = -1$$

$$T(n) = \theta(n^k \log \log n)$$

$$\text{Therefore } T(n) = \theta(n \log \log n)$$

Master Theorem Case 2

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1, f(n) = \theta(n^k \log^p n)$$

case 2: if $\log_b a = k$,

$$\text{if } p > -1, \quad \theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1, \quad \theta(n^k \log \log n)$$

$$\text{if } p < -1, \quad \theta(n^k)$$

$$T(n) = 2 T(n/2) + n/\log^2 n$$

$$a=2, b=2$$

$$f(n) = n/\log^2 n;$$

$$k=1$$

$$p=-2$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p < -1$$

$$T(n) = \theta(n^k)$$

$$\text{Therefore } T(n) = \theta(n)$$

Master Theorem Case 3

$$T(n) = aT(n/b) + f(n), \text{ where } \mathbf{a \geq 1, b > 1}, f(n) = \theta(n^k \log^p n)$$

Case 3: if $\log_b a < k$,

| | |
|-----------------|------------------------|
| if $p \geq 0$, | $\theta(n^k \log^p n)$ |
| if $p < 0$, | $O(n^k)$ |

$$T(n) = 2T(n/2) + n^2 \log n$$

$$a=2, b=2$$

$$f(n) = n^2 \log n;$$

$$k=2$$

$$p=1$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a < k \text{ Hence Case 3 } p>0$$

$$T(n) = \theta(n^k \log^p n)$$

$$\text{Therefore } T(n) = \theta(n^2 \log n)$$

Master Theorem Case 3

$$T(n) = aT(n/b) + f(n), \text{ where } \mathbf{a \geq 1, b > 1}, f(n) = \theta(n^k \log^p n)$$

Case 3: if $\log_b a < k$,

| | |
|-----------------|------------------------|
| if $p \geq 0$, | $\theta(n^k \log^p n)$ |
| if $p < 0$, | $O(n^k)$ |

$$T(n) = T(n/2) + n^2$$

$$a=1, b=2$$

$$f(n) = n^2;$$

$$k=2$$

$$p=0$$

$$\log_b a = \log_2 1 = 0$$

$$\log_b a < k \text{ Hence Case 3 } p = 0$$

$$T(n) = \theta(n^k \log^p n)$$

$$\text{Therefore } T(n) = \theta(n^2)$$

Solve using Master Theorem

$$1. T(n) = 9 T(n/3) + 1$$

$$2. T(n) = 8 T(n/2) + n$$

$$3. T(n) = 8 T(n/2) + n \log n$$

$$4. T(n) = 4 T(n/2) + n^2$$

$$5. T(n) = 4 T(n/2) + n^2 \log n$$

$$6. T(n) = 8 T(n/2) + n^3$$

$$7. T(n) = 4 T(n/2) + n^3$$

Solution to Given problems

1. $T(n) = 9 T(n/3) + 1$

$a=9$ $b=3$ $\log_b a = \log_3 9 = 2 > k=0$, thus answer = $\theta(n^2)$

2. $T(n) = 8 T(n/2) + n$

$a=8$ $b=2$ $\log_b a = \log_2 8 = 3 > k=1$, thus answer = $\theta(n^3)$

3. $T(n) = 8 T(n/2) + n \log n$

$a=8$ $b=2$ $\log_b a = \log_2 8 = 3 > k=1$, thus answer = $\theta(n^3)$

4. $T(n) = 4 T(n/2) + n^2$

$a=4$ $b=2$ $\log_b a = \log_2 4 = 2 = k=2$, $P = 0 > -1$, thus answer = $\theta(n^2 \log n)$

5. $T(n) = 4 T(n/2) + n^2 \log n$

$a=4$ $b=2$ $\log_b a = \log_2 4 = 2 = k=2$, $P = 1 > -1$, thus answer = $\theta(n^2 \log^2 n)$

6. $T(n) = 8 T(n/2) + n^3$

$a=8$ $b=2$ $\log_b a = \log_2 8 = 3 = k=3$, $P = 0 > -1$, thus answer = $\theta(n^3 \log n)$

7. $T(n) = 4 T(n/2) + n^3$

Unit-1

- Analysis of Algorithms
- Best, Average and Worst case running times of algorithms,
- Mathematical notations for running times O , Ω , Θ ,
- Master's Theorem
- **Problem solving principles: Classification of problem,**
- problem solving strategies,
- classification of time complexities (linear, logarithmic etc.)
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

Problem solving

- Application of ideas, skills, or factual information to achieve the solution to a problem or to reach a desired outcome. Let's talk about different types of problems and different types of solutions.

Kinds of problems encountered

- A **well-defined problem** is one that has a clear goal or solution, and problem solving strategies are easily developed.
 - In contrast, **poorly-defined** problem is the opposite. It is unclear, abstract, or confusing, and that does not have a clear problem solving strategy.
- **Routine problem** is one that is typical and has a simple solution
 - In contrast, **non-routine** problem is more abstract or subjective and requires a strategy to solve

Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles: Classification of problem,
 - **problem solving strategies,**
 - classification of time complexities (linear, logarithmic etc.)
-
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles: Classification of problem,
 - **problem solving strategies,**
 - classification of time complexities (linear, logarithmic etc.)
-
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

Strategies

- To solve a routine problem algorithm can be used. Algorithms are step-by-step strategies or processes for how to solve a problem or achieve a goal.
- Another solution that many people use to solve problems is called heuristics.
 - Heuristics are general strategies used to make quick, short-cut solutions to problems that sometimes lead to solutions but sometimes lead to errors.
 - Heuristics are based on past experiences.

Principle of Problem Solving

- Identify a problem
- Understand the problem
- Identify alternative ways to solve a problem
- Select best way to solve a problem from the list of alternative solutions
- Evaluate the solution

Problem Solving Strategies in Course

1. Divide & Conquer
2. Greedy Method
3. Dynamic Programming
4. Backtracking
5. Branch and Bound

6. Randomized algorithms
7. Approximation algorithms
8. Natural Algorithms
9. Parallel and concurrent

Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles: Classification of problem,
 - problem solving strategies,
 - **classification of time complexities (linear, logarithmic etc.)**
-
- Divide and Conquer strategy: General strategy,
 - Quick Sort and Merge Sort w.r.t. Complexity

Increasing Time Complexity Functions

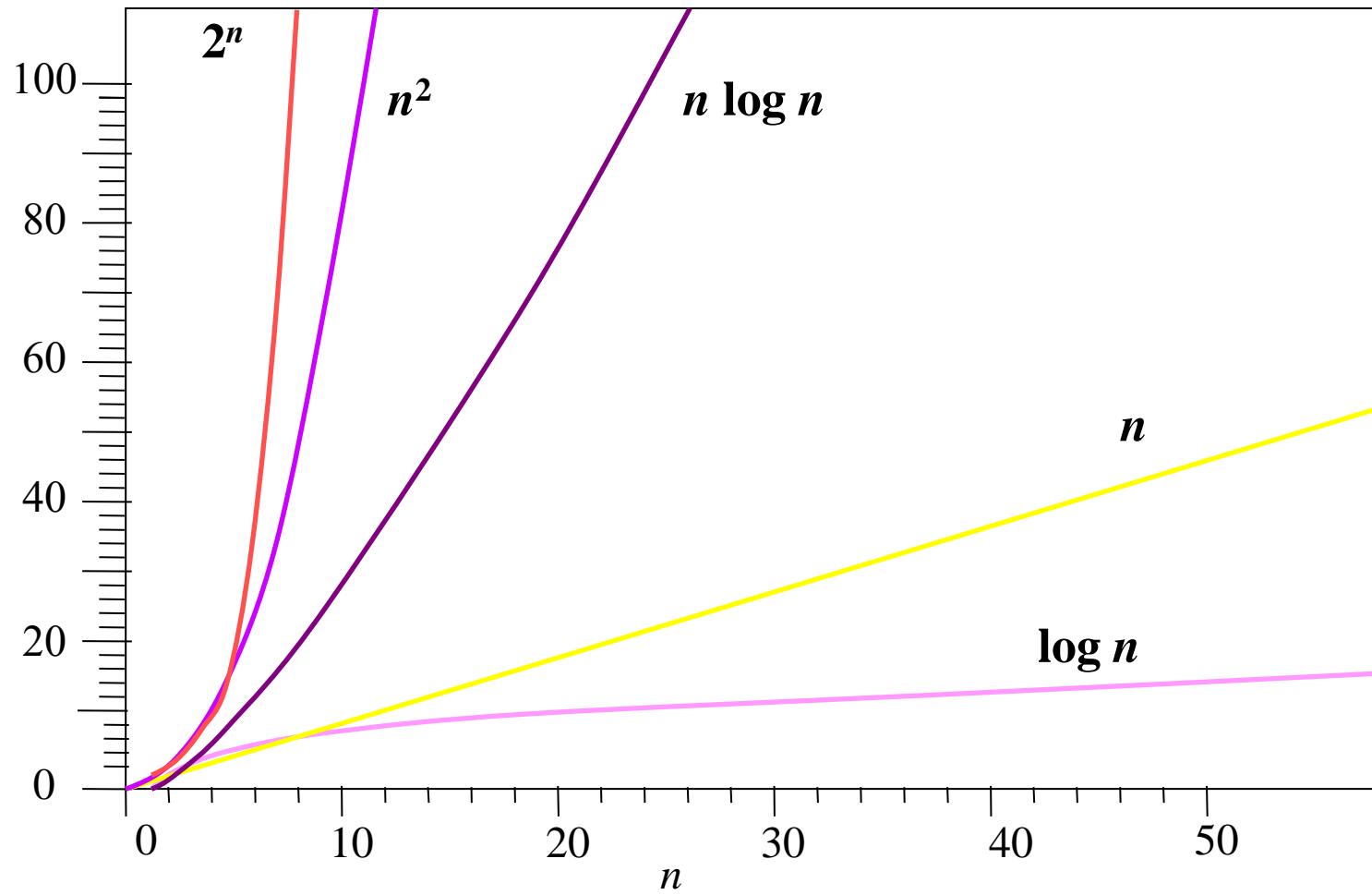
$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

| Name | Time Complexity |
|------------------|-----------------|
| Constant Time | $O(1)$ |
| Logarithmic Time | $O(\log n)$ |
| Linear Time | $O(n)$ |
| Quasilinear Time | $O(n \log n)$ |
| Quadratic Time | $O(n^2)$ |
| Exponential Time | $O(2^n)$ |
| Factorial Time | $O(n!)$ |

Classification of Time Complexity

| Notation | Complexity | Description | Example |
|------------------|--------------|--------------------|-------------------|
| $O(1)$ | Constant | Simple statement | Addition |
| $O(\log(n))$ | Logarithmic | Divide in half | Binary search |
| $O(n)$ | Linear | loop | Linear search |
| $O(n * \log(n))$ | Linearithmic | Divide & Conquer | Merge sort |
| $O(n^2)$ | Quadratic | Double loop | Check all pairs |
| $O(n^3)$ | Cubic | Triple loop | Check all triples |
| $O(2^n)$ | Exponential | Exhaustive search | Check all subsets |
| $O(n!)$ | Factorial | Recursive function | Factorial |

Graphical Comparison



Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles: Classification of problem,
 - problem solving strategies,
 - classification of time complexities (linear, logarithmic etc.)
-
- **Divide and Conquer strategy: General strategy,**
 - Quick Sort and Merge Sort w.r.t. Complexity

Divide & Conquer

- Given a function to compute on n inputs the divide and conquer strategy suggests splitting the inputs into k distinct substs, $1 < k \leq n$, yielding k sub problems.
- These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole.
- Sub problems resulting from a divide and conquer strategy design are of the same type as the original problem.
- Hence reapplication of the divide and conquer principle is naturally expressed as recursive algorithm.

Control abstraction for DA&C

Algorithm DA&C(P)

{

 if Small(P) then return $S(P)$;

 else

 {

 divide P into smaller instances of P_1, P_2, \dots, P_k , $k \geq 1$;

 Apply DA&C to each of these sub problems;

 return Combine(DA&C(P_1), DA&C(P_2), ..., DA&C(P_k));

 }

}

Time Complexity of DA&C

$$T(n) = T(1) \quad n=1$$

$$a T(n/b) + f(n) \quad n>1$$

a and b are known constants .

We assume $T(1)$

is known and n is a power of b(i. e $n=b^k$)

Unit-1

- Analysis of Algorithms
 - Best, Average and Worst case running times of algorithms,
 - Mathematical notations for running times O , Ω , Θ ,
 - Master's Theorem
-
- Problem solving principles: Classification of problem,
 - problem solving strategies,
 - classification of time complexities (linear, logarithmic etc.)
-
- Divide and Conquer strategy: General strategy,
 - **Quick Sort and Merge Sort w.r.t. Complexity**

Quicksort

Quicksort

The three-step divide-and-conquer process for sorting a typical subarray $A[p \dots r]$

Divide: Partition (rearrange) the array $A[p \dots r]$ into two (possibly empty) subarrays $A[p \dots q - 1]$ and $A[q + 1 \dots r]$ such that each element of $A[p \dots q - 1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q + 1 \dots r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \dots q - 1]$ and $A[q + 1 \dots r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \dots r]$ is now sorted.

Quicksort

```
QUICKSORT( $A, p, r$ )
```

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 
```

To sort an entire array A , the initial call is $\text{QUICKSORT}(A, 1, A.\text{length})$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \dots r]$ in place.

```
PARTITION( $A, p, r$ )
```

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

Quicksort

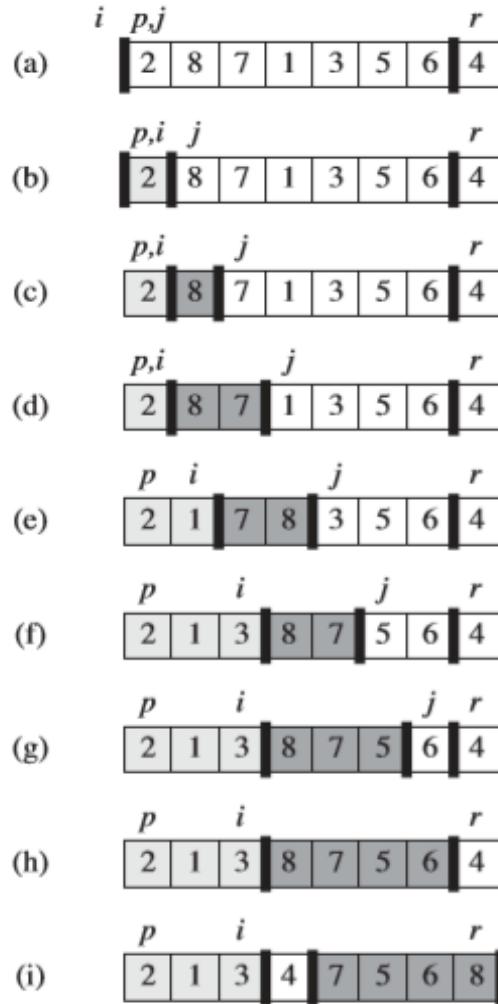
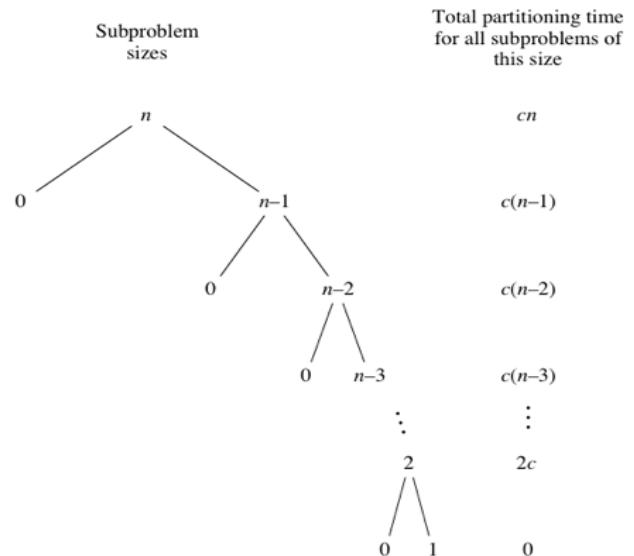


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x . Lightly shaded array elements are all in the first partition with values no greater than x . Heavily shaded elements are in the second partition with values greater than x . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Quick Sort Analysis (Worst Case)

When quicksort always has the most unbalanced partitions possible, then the original call takes cn time for some constant c , the recursive call on $n - 1$ elements takes $c(n - 1)$ time, the recursive call on $n - 2$ elements takes $c(n - 2)$ time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



When we sum all the partition times at each level :

$$cn + c(n-1) + c(n-2) + \dots + 2c = c(n + (n-1) + (n-2) + \dots + 2) = c((n+1)(n/2)-1)$$

Therefore timecomplexity is $\Theta(n^2)$

Quick Sort Analysis (Worst Case) by substitution

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + n & \text{if } n > 0 \end{cases}$$

$$T(n) = T(n-1) + n \quad \text{----- (eq 1)}$$

Find value of $T(n-1)$ by replacing $n-1$ in place of n in eq 1

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

Put value of $T(n-1)$ in equation 1 we have

$$T(n) = T(n-2) + (n-1) + n \quad \text{----- (eq 2)}$$

Find value of $T(n-2)$ by replacing $n-2$ in place of n in eq 1

$$T(n-2) = T(n-3) + n-2$$

Put value of $T(n-2)$ in equation 2 we have

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad \text{----- (eq 3)}$$

|

$$T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

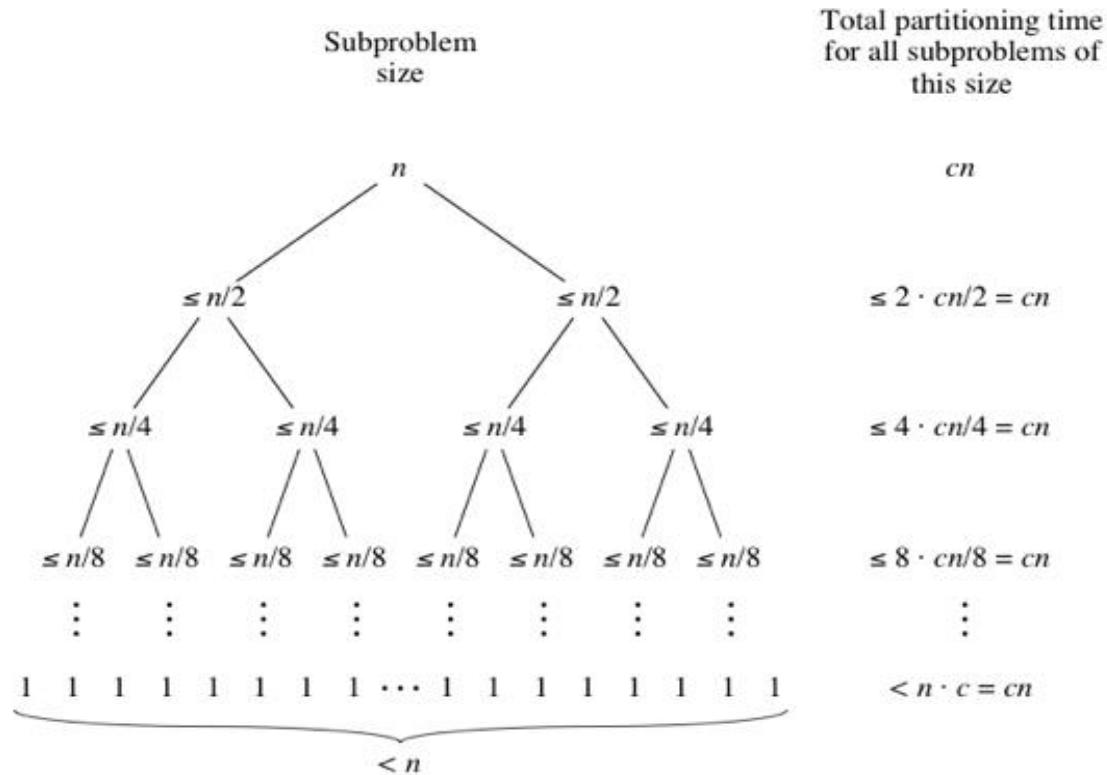
Assume $n - k = 0$ therefore $k = n$

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

$$T(n) = T(0) + 1 + 2 + 3 + \dots + (n-1) + n$$

$$T(n) = 1 + n(n+1)/2 = \Theta(n^2)$$

Quick Sort Analysis (Best Case)



Best case occurs when portioning the element array divides it into equal size sub arrays. The time required for comparison at each level is n and there are $\log n$ levels (height of tree). Therefore time complexity is $\Theta(n \log_2 n)$.

Quick Sort Analysis (Best Case)

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$T(n) = 2 T(n/2) + n \quad \text{(eq 1)}$$

$$T(n/2) = 2 T(n/2^2) + n/2$$

Put $T(n/2)$ in eq 1 we have

$$\begin{aligned} T(n) &= 2 [2 T(n/2^2) + n/2] + n \\ &= 2^2 T(n/2^2) + n + n \\ &= 2^2 T(n/2^2) + 2n \quad \text{(eq 2)} \end{aligned}$$

$$T(n/2^2) = 2 T(n/2^3) + n/2^2$$

Put $2 T(n/2^2)$ in eq 2 we have

$$\begin{aligned} T(n) &= 2^2 [2 T(n/2^3) + n/2^2] + 2n \\ &= 2^3 T(n/2^3) + n + 2n \\ &= 2^3 T(n/2^3) + 3n \quad \text{(eq 3)} \end{aligned}$$

$$T(n) = 2^k T(n/2^k) + kn$$

Solve $T(n/2^k)$ till $T(1)$

Quick Sort Analysis (Best Case)

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$T(n) = 2^k T(n/2^k) + kn$$

Solve $T(n/2^k)$ till it becomes $T(1)$

Therefore $T(n/2^k) = T(1)$

$n/2^k = 1$, therefore $n = 2^k$, thus $k = \log n$

$$\begin{aligned} T(n) &= 2^k T(1) + kn \\ &= n * 1 + n \log n \end{aligned}$$

Therefore $T(n) = \Theta(n \log n)$

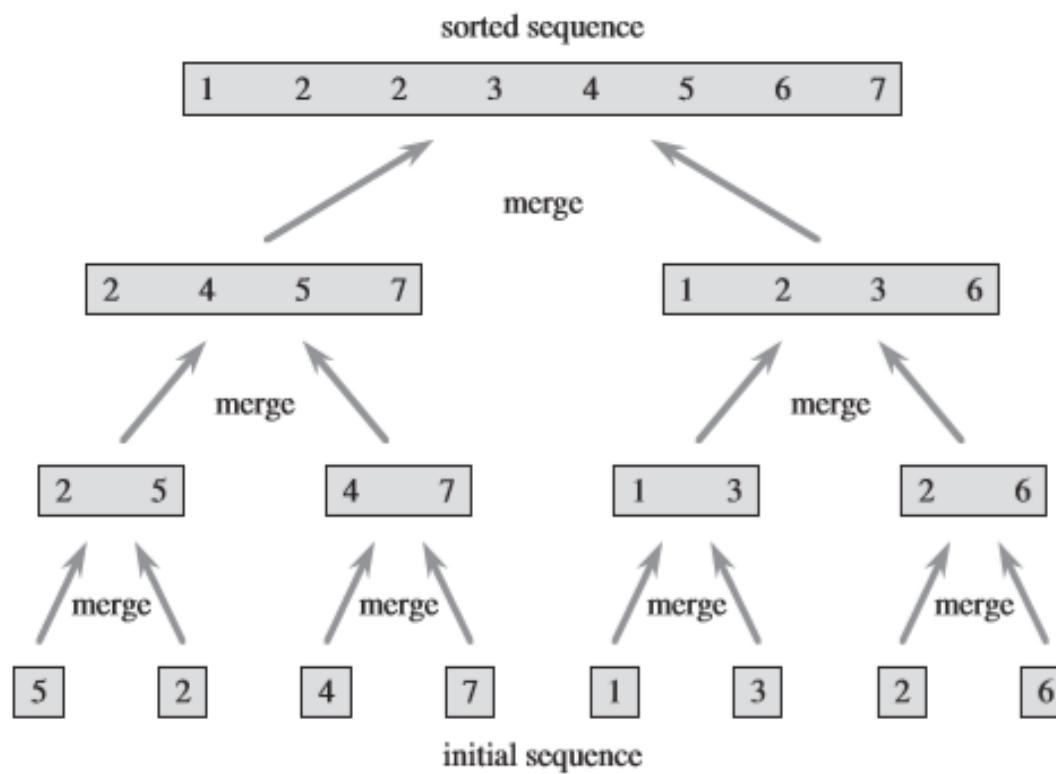
Mergesort

Merge Sort

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n -element sequence to be sorted into two subsequences of $n=2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

Merge Sort Example



Merge sort

MERGE-SORT(A, p, r)

- 1 if $p < r$
- 2 $q = \lfloor (p + r)/2 \rfloor$
- 3 MERGE-SORT(A, p, q)
- 4 MERGE-SORT($A, q + 1, r$)
- 5 MERGE(A, p, q, r)

Merge sort

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5     $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7     $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else  $A[k] = R[j]$ 
17      $j = j + 1$ 
```

Recurrence Relation for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n -element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n , that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the “conquer” step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \quad (2.1)$$

Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases} \quad (2.2)$$

Practice Assignment

- Derive the time complexity of Merge sort using :
 1. Master theorem
 2. Using Substitution method

https://docs.google.com/forms/d/e/1FAIpQLSdCop_NyAbz2kx09PRAHzo80h-Rm1WHupCtb-T4PXcfGcgucQ/viewform?usp=sf_link

References

1. Fundamentals Of Computer Algorithms

- E.Horowitz , S.Sahni , S.Rajasekaran

2. Fundamentals of Algorithm

- Gilles Brassard, Paul Brately

3. Introduction to Algorithms

- T. H. Cormen , C. E. Leiserson, R. L. Rivest,
Clifford Stein

Thank You!!

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)**

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)

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 w_i ; and the knapsack has a capacity M .

If a fraction x_i , $0 \leq x_i \leq 1$, of object i is placed into the knapsack then a profit of $P_i X_i$ 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 $\sum_{1 \leq i \leq n} P_i X_i$ (1)

subject to

$$\sum_{1 \leq i \leq n} w_i X_i \leq M \quad (2)$$

$$0 \leq x_i \leq 1, \quad p_i \geq 0, \quad w_i \geq 0 \quad \dots\dots\dots (3)$$

- 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) | $\sum w_i x_i$ | $\sum p_i x_i$ |
|------|-------------------|----------------|----------------|
| 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

| Que s | Q1) | Q2) | Q3) | Q4) | Q5) | Q6) | Q7) | Q8) | Q9) | Q10) | Q11) | Q12) |
|-------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|
| Mar ks | 5 | 5 | 6 | 7 | 7 | 8 | 9 | 10 | 10 | 14 | 20 | 24 |
| Spec ific Stud ent Time | 3 | 5 | 2 | 8 | 12 | 4 | 10 | 12 | 16 | 17 | 16 | 25 |

Soln

| Que s | Q1) | Q2) | Q3) | Q4) | Q5) | Q6) | Q7) | Q8) | Q9) | Q10) | Q11) | Q12) |
|-------------------------------------|------|-----|-----|------|------|-----|-----|------|------|-----------|------|------|
| Mar ks | 5 | 5 | 6 | 7 | 7 | 8 | 9 | 10 | 10 | 14 | 20 | 24 |
| Spec ific Stud ent Time | 3 | 5 | 2 | 8 | 12 | 4 | 10 | 12 | 16 | 17 | 16 | 25 |
| | 1.66 | 1 | 3 | .875 | .583 | 2 | .9 | .833 | .625 | .823 5 | 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

($p_1, p_2, p_3, p_4, p_5, p_6, p_7$) = (10, 5, 15, 7, 6, 18, 3)

($w_1, w_2, w_3, w_4, w_5, w_6, w_7$) = (2, 3, 5, 7, 1, 4, 1)

2. No. of objects =5, m=100,

(p_1, p_2, p_3, p_4, p_5) = (20, 30, 66, 40, 60)

(w_1, w_2, w_3, w_4, w_5) = (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) ≥ 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 ← 0 //initialize solution to zero//
    cu ← M //cu = remaining knapsack capacity//
    for i ← 1 to n do
        if W(i) > cu then exit endif
        X(i) ← 1
        cu ← cu - W(i)
    repeat
        if i ≤ n then X(i) ← 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 = $O(n \log n)$
2. While loop will be executed 'n' times in worst case, so the
computation time = $O(n)$
3. **Time complexity = $O(n \log n)$**
4. **Space complexity = $c + \text{space required for } x[1:n] = O(n)$**

Scheduling Algorithms : Job scheduling



Scheduling Algorithms : Job scheduling

Given a set of n jobs.

- d_i is an integer deadline associated with job i
- $d_i \geq 0$ and a profit $p_i \geq 0$.
- 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
```

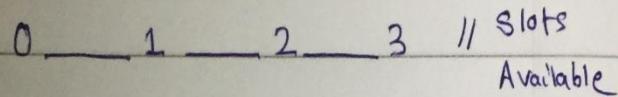
Job Sequencing Example (Not efficient for large number of jobs/ deadlines)

$$n = 5$$

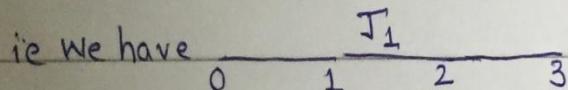
$$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$$

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

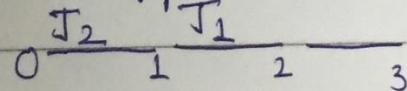
Maximum deadline is 3 that means we can include maximum 3 jobs



Choose 1st Job, deadline is 2, place it in Slot 1 → 2
i.e. we have

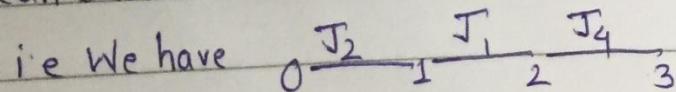


Choose 2nd Job deadline is 2, Now 1 → 2 is occupied, so check if previous slot is free i.e. we have



choose 3rd Job, deadline is 1, as both 0 → 1 & 1 → 2 are filled we reject it.

Then choose 4th Job, deadline is 3 ∵ we can choose 2 → 3 slot



Now further we cannot take more job as max deadline is 3, and we have chosen 3 jobs.

$$\therefore \text{Profit} = \{20 + 15 + 5\} \text{ and Jobs} = \{J_2, J_1, J_4\}$$

Seq or $\{J_1, J_2, J_4\}$

Solved Example

Example 4.4 Let $n = 5$, $(p_1, \dots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \dots, d_5) = (2, 2, 1, 3, 3)$. 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 $J = \{1, 2, 4\}$. \square

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  $p[1] \geq p[2] \geq \dots \geq p[n]$ .  $J[i]$ 
// 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 $n = 5$, $(p_1, \dots, p_5) = (60, 100, 20, 40, 20)$

$(d_1, \dots, d_5) = (2, 1, 3, 2, 1)$

Example 2 :

Let $n = 7$, $(p_1, \dots, p_7) = (35, 30, 25, 20, 15, 12, 5)$

$(d_1, \dots, d_7) = (3, 4, 4, 2, 3, 1, 2)$

Example 3:

Let $n = 6$, $(p_1, \dots, p_6) = ($

$50, 45, 40, 30, 25, 10)$ $(d_1, \dots, d_6) = (4, 1, 3, 1, 2, 2)$

Solution

1. Let $n = 5$, $(p_1, \dots, p_5) = (60, 100, 20, 40, 20)$

$(d_1, \dots, d_5) = (2, 1, 3, 2, 1)$

Answer : { J1 , J2, J3 } Profit : 180

2. Let $n= 7$, $(p_1, \dots, p_7) = (35, 30, 25, 20, 15, 12, 5)$

$(d_1, \dots, d_7) = (3, 4, 4, 2, 3, 1, 2)$

Answer : { J4 , J3, J1 , J2 } Profit :110

3. Let $n= 6$, $(p_1, \dots, p_6) = (50, 45, 40, 30, 25, 10)$

$(d_1, \dots, d_6) = (4, 1, 3, 1, 2, 2)$

Answer : {J2, J5, J3, J1 } , Profit : 160

Word problems

(1) An appliance repair situation: Let $n = 5$, $(p_1, \dots, p_5) = (60, 100, 20, 40, 20)$

$(d_1, \dots, d_5) = (2, 1, 3, 2, 1)$

Select which appliance to repair

(2) A tailor shop situation : Let $n= 7$, $(p_1, \dots, p_7) = (35, 30, 25, 20, 15, 12, 5)$

$(d_1, \dots, d_7) = (3, 4, 4, 2, 3, 1, 2)$

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:

ABCBDBCCDAABBEEEBEAB – 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:

ABBCDBCCDAABBEEBEAB – 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

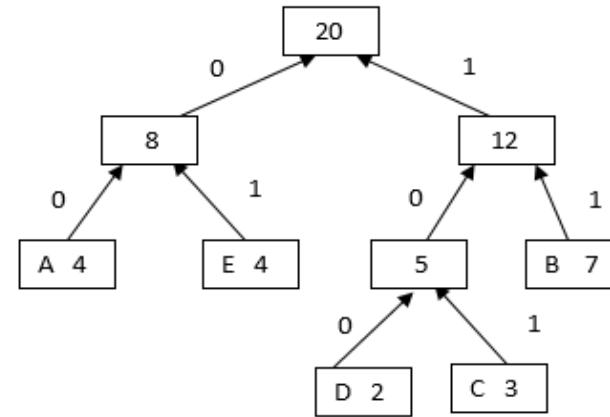
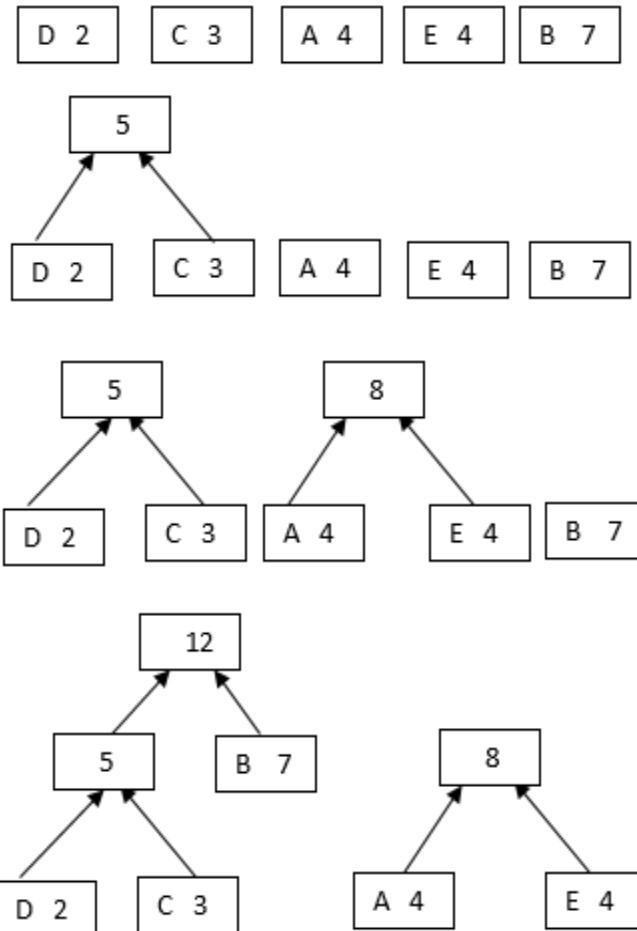
Consider the following text:

ABCBDBCCDAABBEEEEEAB – 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 | 00 |
| B | 7 | 11 |
| C | 3 | 101 |
| D | 2 | 100 |
| E | 4 | 01 |

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

$$\text{Length} = 4*2 + 7*2 + 3*3 + 2*3 + 4*2 = \\ (8 + 14 + 9 + 6 + 8) = 45 \text{ bits}$$

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

$$5 * 8 + (2 + 2 + 3 + 3 + 2) = 40 + 12 = 52 \text{ bits}$$

Therefore total number of bits transferred are :
 $45 + 52 = 97 \text{ bits}$

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)

```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // 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 $O(n \log n)$.

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
- 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

After completion of the course, student will be able to

- 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

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.

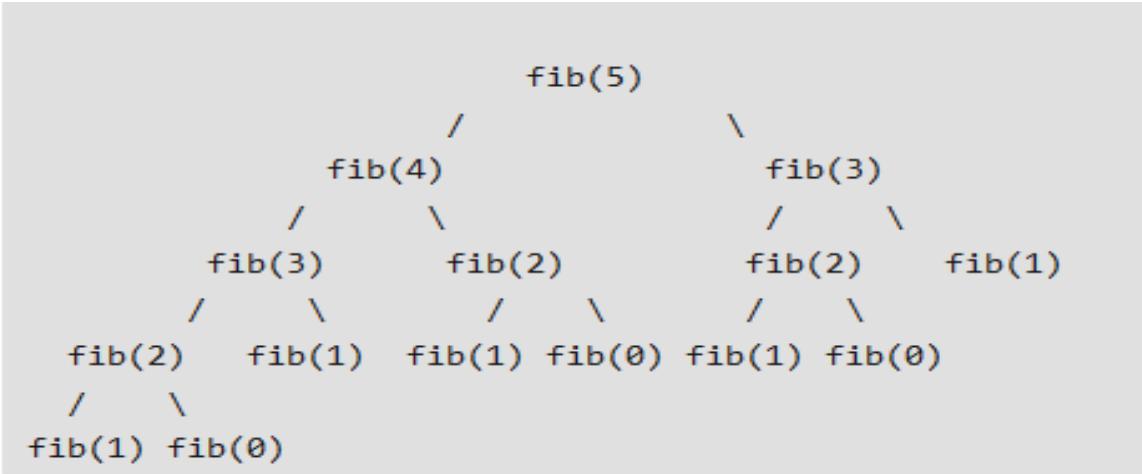
2. Optimal Substructure:

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);
}
```



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 w_i , $1 \leq i \leq n$, and the knapsack has a capacity ‘m’ (maximum weight the knapsack can hold).

If an object $x_i \in \{0, 1\}$ is placed into the knapsack then a profit of p_{xi} 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 $\sum p_i x_i$ (1)

$$1 \leq i \leq n$$

subject to $\sum w_i x_i \leq m$ (2)

$$1 \leq i \leq n$$

$$x_i \in \{0, 1\}, \quad 1 \leq i \leq n \quad (3)$$

$$p_i \geq 0, \quad w_i \geq 0 \quad (4)$$

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 : $n = 4$, $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$, $(p_1, p_2, p_3, p_4) = (2, 3, 5, 7)$, $m=7$

Example : $n = 6$, $(w_1, w_2, w_3, w_4, w_5, w_6) = (p_1, p_2, p_3, p_4, p_5, p_6) = (100, 50, 20, 10, 7, 3)$, $m=165$

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 S_1^0 to compute S^1

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}\}$$

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 .

$$S^0 = \{(0, 0)\}; S_1^0 = \{(1, 2)\}$$

$$S^1 = \{(0, 0), (1, 2)\}; S_1^1 = \{(2, 3), (3, 5)\}$$

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

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

Example 5.21 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

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

$$S^1_1 = \{(1, 2)\}$$

Include first object

$$S^1_2 = \{(0, 0), (1, 2)\}$$

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

Include second object

$$S^1_2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

$$S^1_3 = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$$

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

Delete pair $(3, 5)$ by dominance rule

Delete pairs $(7, 7)$ 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 : $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, $m=6$

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

Example 5.19 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 $(x_1, x_2, x_3) = (1, 0, 1)$. \square

Solving Knapsack using Sets method

Example : $n = 4$, $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$,
 $(p_1, p_2, p_3, p_4) = (2, 3, 5, 7)$, $m=7$

Solve above example of 0/1 knapsack using sets method

Informal Algorithm

```
1  Algorithm DKP( $p, w, n, m$ )
2  {
3       $S^0 := \{(0, 0)\}$ ;
4      for  $i := 1$  to  $n - 1$  do
5      {
6           $S_1^{i-1} := \{(P, W) | (P - p_i, W - w_i) \in S^{i-1} \text{ and } W \leq m\}$ ;
7           $S^i := \text{MergePurge}(S^{i-1}, S_1^{i-1})$ ;
8      }
9      ( $PX, WX$ ) := last pair in  $S^{n-1}$ ;
10     ( $PY, WY$ ) :=  $(P' + p_n, W' + w_n)$  where  $W'$  is the largest  $W$  in
11         any pair in  $S^{n-1}$  such that  $W + w_n \leq m$ ;
12     // Trace back for  $x_n, x_{n-1}, \dots, x_1$ .
13     if ( $PX > PY$ ) then  $x_n := 0$ ;
14     else  $x_n := 1$ ;
15     TraceBackFor( $x_{n-1}, \dots, x_1$ );
16 }
```

Unit II : Dynamic Programming

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

- Let $g_i(y)$ denote the value of optimal solution to $\text{knap}(i+1, n, y)$, where $(i+1)$ to n are the objects and y is the capacity of knapsack.
- Clearly $g_0(m)$ is the value of an optimal solution to $\text{knap}(1, n, m)$
- The possible decisions for x_1 are 0 or 1. From the principle of optimality it follows that ,

$$g_0(m) = \max \{ g_1(m), (g_1(m - w_1) + p_1) \}$$

if $x_1 = 0$ if $x_1 = 1$

- The above equation can be generalized as,

$$g_i(y) = \max \{ g_{i+1}(y), (g_{i+1}(y - w_{i+1}) + p_{i+1}) \}$$

if $x_{i+1} = 0$ if $x_{i+1} = 1$

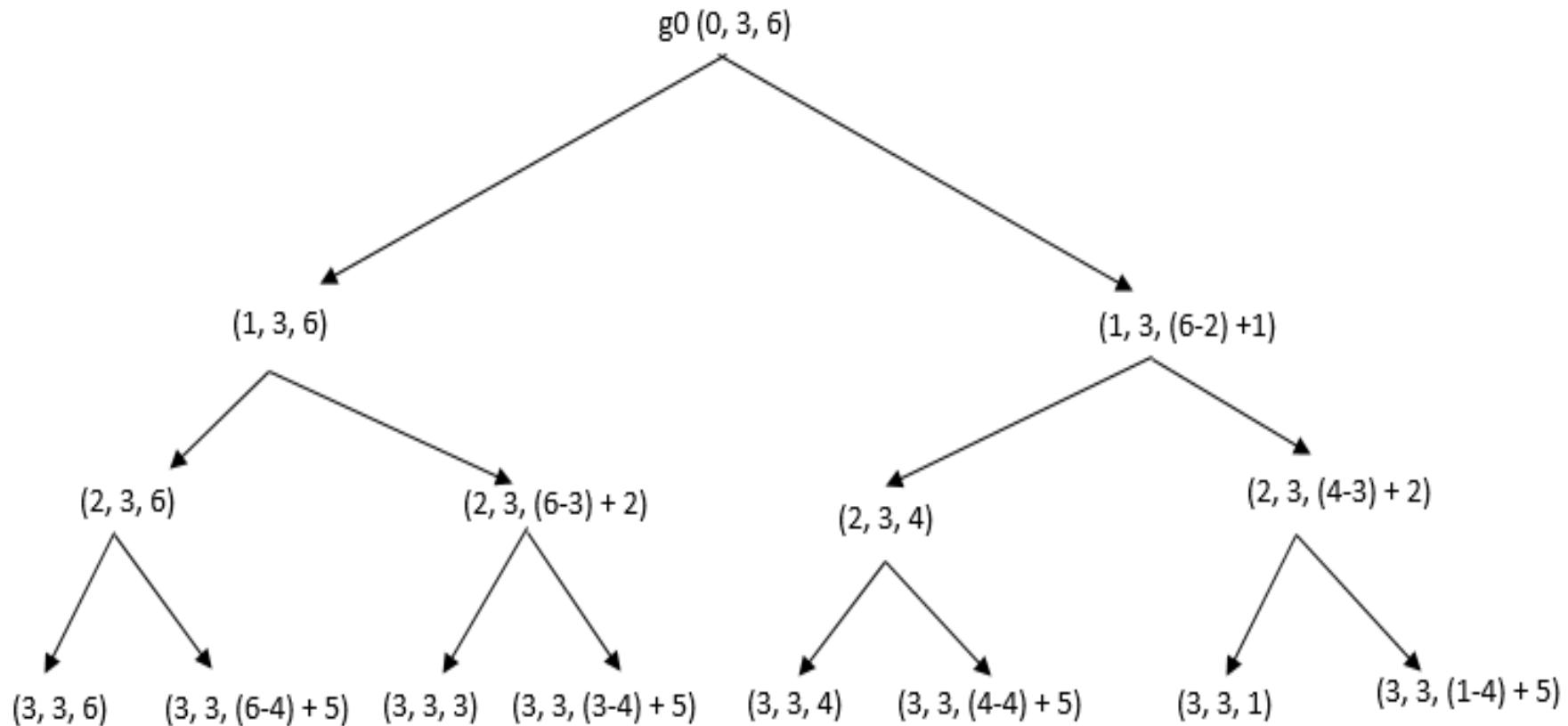
- This equation can be used to obtain $g_{n-1}(y)$ from $g_n(y)$ and can be further used recursively to obtain **optimal solution $g_0(y)$** with the knowledge that $g_n(y) = 0$ for all $y \geq 0$ and

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

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.

$$n = 3, \quad (w_1, w_2, w_3) = (2, 3, 4) \text{ and}$$

$$m = 6, \quad (p_1, p_2, p_3) = (1, 2, 5)$$

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(2ⁿ)**

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.

$$n = 3, \quad (w_1, w_2, w_3) = (2, 3, 4) \text{ and}$$

$$m = 6, \quad (p_1, p_2, p_3) = (1, 2, 5)$$

| | | ... 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 |

Unit II : Dynamic Programming

Example : Solve the following instance using Dynamic Programming.

$n = 4$, $(w_1, w_2, w_3, w_4) = (1, 3, 4, 5)$ and

$m = 7$, $(p_1, p_2, p_3, p_4) = (1, 4, 5, 7)$

$\dots j \rightarrow$

| i ↓ | Weight Limit 1 to n -> | 00 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|--------------------|---------------------------|----|---|---|---|---|---|---|---|
| $w_1 = 1, p_1 = 1$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $w_2 = 3, p_2 = 4$ | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 | 5 |
| $w_3 = 4, p_3 = 5$ | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 | |
| $w_4 = 5, p_4 = 7$ | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 | |

Example : Solve the following instance using Dynamic Programming.

$$n = 5, \quad (w_1, w_2, w_3, w_4, w_5) = (1, 2, 5, 6, 7) \text{ and}$$

$$m = 11, \quad (p_1, p_2, p_3, p_4, p_5) = (1, 6, 18, 22, 28)$$

| 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

Algo 0/1 Knapsack & Trace

Algorithm 01Knapsack(p[], w[], n, m)
 { // 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];
                else if (K[i-1][j] >= p[i] + K[i-1][j-w[i]])
                    {
                        K[i][j] = K[i-1][j];
                    }
                else
                    K[i][j] = p[i] + K[i-1][j-w[i]];
            }
        }
        print K[n][m]; // optimal profit
    }
```

Algorithm Trace(K[][] , p[], w[], n, m)
 { // Algorithm that prints the solution to 0/1 knapsack
 x[1..n] = 0; i = n; j = m;
 repeat
 {
 if ((i == 0) || (j == 0)) break;
 if (K[i][j] == K[i-1][j])
 x[i] = 0; i = i-1;
 else
 x[i] = 1; i = i-1; j = j - w[i];
 } until (false);
 print x[1..n];
 }

Unit II : Dynamic Programming

Example : Solve the following instance using Dynamic Programming.

$n = 4$, $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$ and

$m = 8$, $(p_1, p_2, p_3, p_4) = (1, 2, 5, 6)$

Unit II : Dynamic Programming

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: $w_k \leq w$. 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 $w_i \leq w$ // item i can be part of the solution

$$\text{if } b_i + B[i-1, w-w_i] > B[i-1, w]$$

$$B[i,w] = b_i + B[i-1, w-w_i]$$

else

$$B[i,w] = B[i-1,w]$$

$$\text{else } B[i,w] = B[i-1,w] \text{ // } w_i > w$$

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.

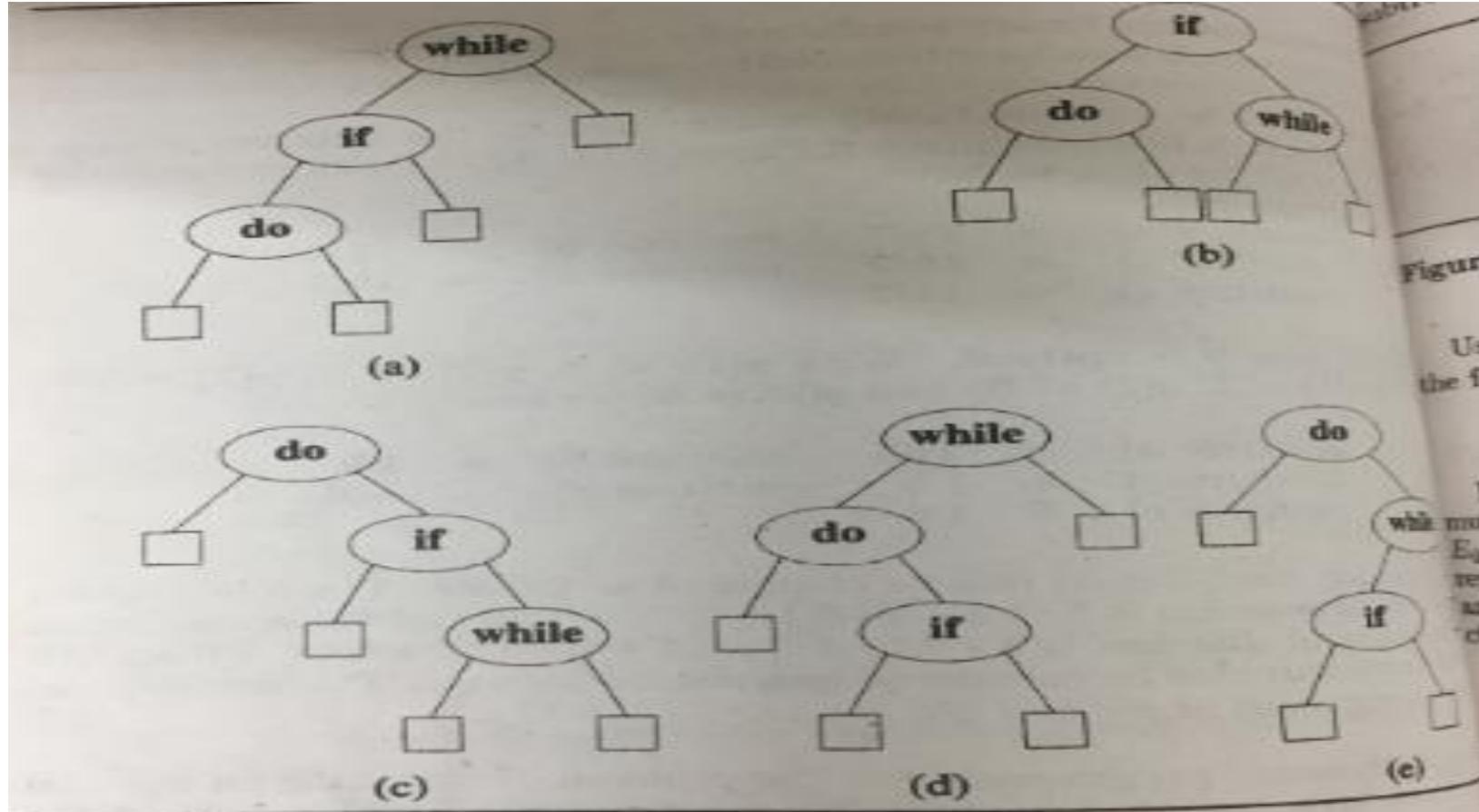
•**Example :** {a1, a2, a3} = {do, if, while}, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?

•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 $(2^n C_n)/(n+1)$ 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 :** $\{a_1, a_2, a_3\} = \{\text{do, if, while}\}$, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?
- Find OBST if weights are not equal with $(p_1, p_2, p_3) = (0.5, 0.1, 0.05)$ and $(q_0, q_1, q_2, q_3) = (0.15, 0.1, 0.05, 0.05)$



Unit II : Dynamic Programming

- **Example :** $\{a_1, a_2, a_3\} = \{\text{do, if, while}\}$, assume equal weights. Find all possible BSTs and their costs. Which BST is OBST?
- Find OBST if weights are not equal with $(p_1, p_2, p_3) = (0.5, 0.1, 0.05)$ and $(q_0, q_1, q_2, q_3) = (0.15, 0.1, 0.05, 0.05)$

Cost contribution of internal node a_i is $p(i) * \text{level}(a_i)$

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 E_i $0 \leq i \leq n$. E_0 contains all identifiers x such that $x < a_1$. E_i contains all identifiers x such that $a_i < x < a_{i+1}$, $1 \leq i < n$. E_n contains all identifiers x , $x > a_n$. It is easy to see that for all identifiers in the same class E_i , the search terminates at the same external node. For identifiers in different E_i the search terminates at different external nodes. If the failure node for E_i is at level 1 then only $l - 1$ iterations of the while loop are made. Hence, the cost contribution of this node is $q(i) * (\text{level}(E_i) - 1)$.

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

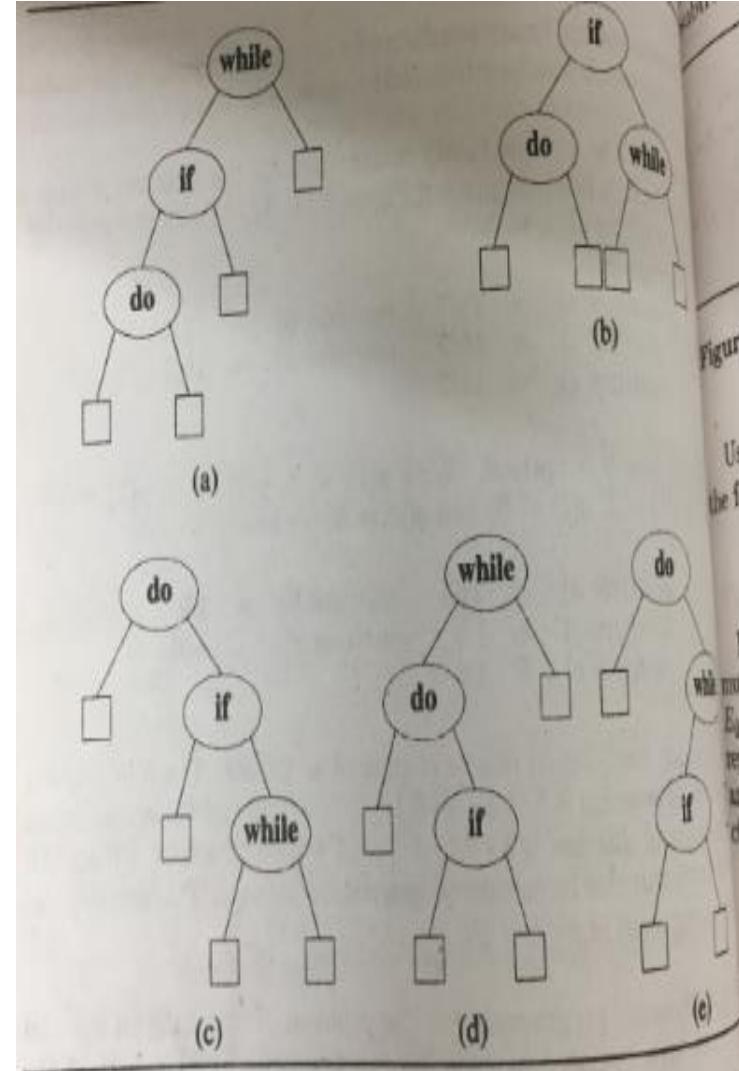
Unit II : Dynamic Programming

$$\begin{aligned} \text{• Cost} &= \sum p_i * \text{level}(a_i) + \sum q_i * (\text{level}(E_i) - 1) \\ 1 &\leq i \leq n \quad 0 \leq i \leq n \end{aligned}$$

• $\{a_1, a_2, a_3\} = \{\text{do, if, while}\}$, assume

• Equal probabilities $p_i = q_i = 1/7$.

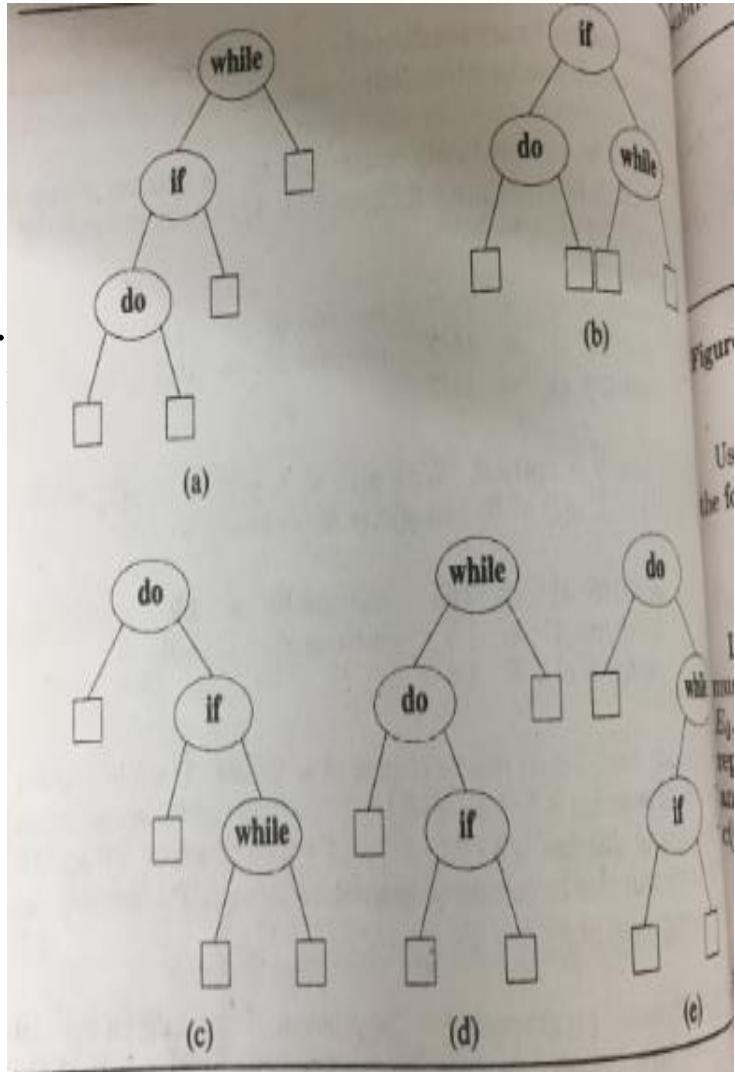
$$\begin{array}{lll} \text{cost(tree a)} & = & 15/7 \\ \text{cost(tree c)} & = & 15/7 \\ \text{cost(tree e)} & = & 15/7 \end{array} \quad \begin{array}{lll} \text{cost(tree b)} & = & 13/7 \\ \text{cost(tree d)} & = & 15/7 \end{array}$$



Find OBST if weights are not equal with

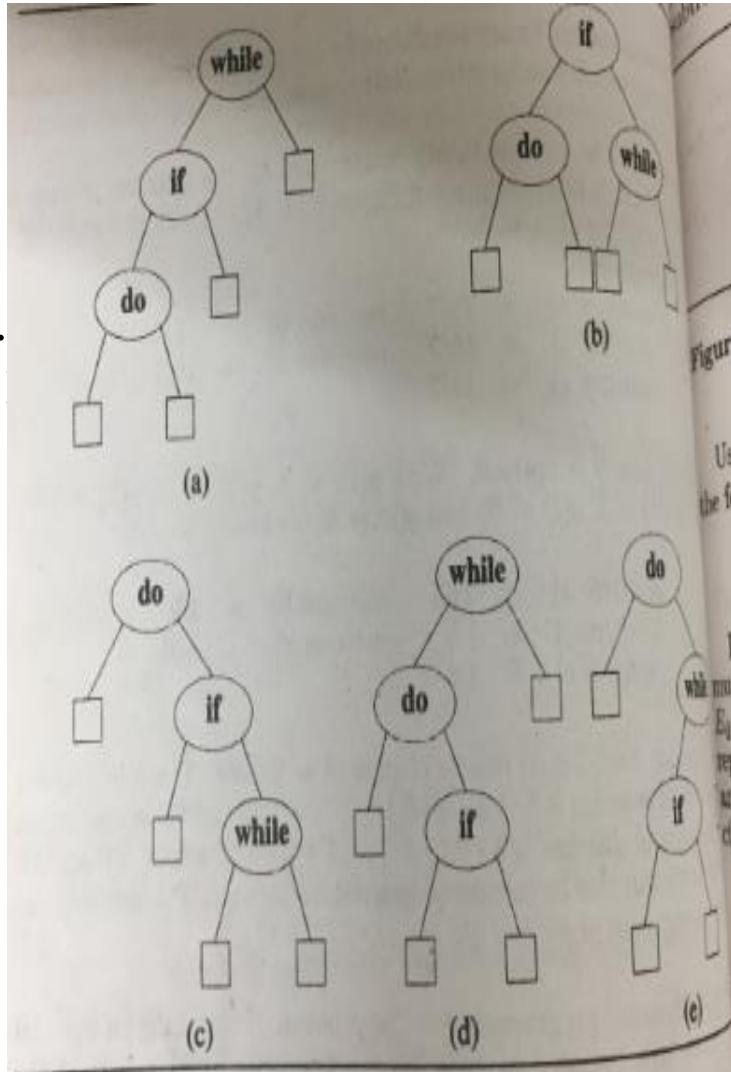
- $(p_1, p_2, p_3) = (0.5, 0, 0)$
- $(q_0, q_1, q_2, q_3) = (0.15, 0.15, 0.15, 0.15)$

$$\text{• Cost} = \sum p_i * \text{level}(a_i) + \sum q_i * (\text{level}(E_i) - 1)$$
$$1 \leq i \leq n \quad 0 \leq i \leq n$$



Find OBST if weights are not equal with

- $(p_1, p_2, p_3) = (0.5, 0, 0)$
- $(q_0, q_1, q_2, q_3) = (0.15, 0, 0, 0)$



$$\begin{array}{ll} \text{cost(tree a)} & = 2.65 \\ \text{cost(tree c)} & = 1.5 \\ \text{cost(tree e)} & = 1.6 \end{array}$$

$$\begin{array}{ll} \text{cost(tree b)} & = 1.9 \\ \text{cost(tree d)} & = 2.05 \end{array}$$

Unit II : Dynamic Programming

Optimal Binary Search Tree (OBST) : Problem Definition

Problem Specifications :

Let us assume given set of identifiers as $\{a_1, a_2, \dots, a_n\}$ with $a_1 < a_2 < \dots < a_n$.

- Let $p(i)$ be the probability with which we search for a_i and let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$.

- Further assume that $a_0 = -\infty$ and $a_{n+1} = +\infty$.

- Probability of unsuccessful searches $= \sum q[i] \quad 0 \leq i \leq n$

- Probability of successful searches $= \sum p[i] \quad 1 \leq i \leq n$

- $\sum p[i] + \sum q[i] = 1 \quad 1 \leq i \leq n \quad 0 \leq i \leq n$

- 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 a_i 's (with weights p_i 's) should be selected as the root node of the tree.
- If we choose a_k as the root node from a_1, a_2, \dots, a_n (sorted in non-decreasing order) , then it is clear that the internal nodes a_1, a_2, \dots, a_{k-1} and external nodes for classes $E_0, E_1, E_2, \dots, E_{k-1}$ will be in the **left sub tree l** and the internal nodes $a_{k+1}, a_{k+2}, \dots, a_n$ and external nodes for classes $E_{k+1}, E_{k+2}, \dots, E_n$ will be in the **right sub tree r**. Let root a_k be at level 1.

$$\text{Cost } (l) = \sum_{1 \leq i < k} p_i * \text{level } (a_i) + \sum_{0 \leq i < k} q_i * (\text{level } (E_i) - 1)$$

$$\text{Cost } (r) = \sum_{k < i \leq n} p_i * \text{level } (a_i) + \sum_{k \leq i \leq n} q_i * (\text{level } (E_i) - 1)$$

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

- Let $w(i, j) =$ Total weight of BST containing identifiers $a_{i+1}, a_{i+2}, \dots, a_j$ and q_i, q_{i+1}, \dots, q_j

$$w(i, j) = q_i + \sum_{l=i+1}^j (ql + pl)$$
$$w(i, j) = p_j + q_j + w(i, j - 1) \quad \dots (1)$$

- Cost of expected BST

$$= p(k) + cost(l) + cost(r) + w(0, k-1) + w(k, n) \quad \dots (2)$$

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

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

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

- k must be chosen such that,

$$p(k) + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n) \text{ is minimum}$$

- Cost of OBST

$$c(0, n) = \min\{c(0, k-1) + c(k, n) + p(k) + w(0, k-1) + w(k, n)\} \dots (3)$$
$$1 \leq k \leq n$$

- We can generalize equation (3) above as,

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

$$c(i, j) = \min\{c(i, k-1) + c(k, j) + w(i, j)\} \dots (4)$$
$$i < k \leq j$$

with the knowledge that,

$$c(i, i) = 0, \quad w(i, i) = q_i \quad \& \quad r(i, i) = 0$$
$$\dots (5)$$

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :
Example :

Construct OBST for the following instance,

$n = 4$, $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$

$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

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

Unit II : Dynamic Programming

Dynamic Programming approach to construct OBST :

$n = 4$, $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$ $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

| (i, j) | $w(i,j) = p(j) + q(j) + w(i,j-1)$ | $C(i,j) = w(i, j) + \min_{i < k \leq j} \{c(i, k-1) + c(k, j)\}$ | 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) | $p(4) + q(4) + w(2,3) = 1 + 1 + 3 = 5$ | $w(2, 4) + \min(\{c(2, 2) + c(3, 4)\}, \{c(2, 3) + c(4, 4)\})$ | $5 + \min\{3, 4\}$ | 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 :

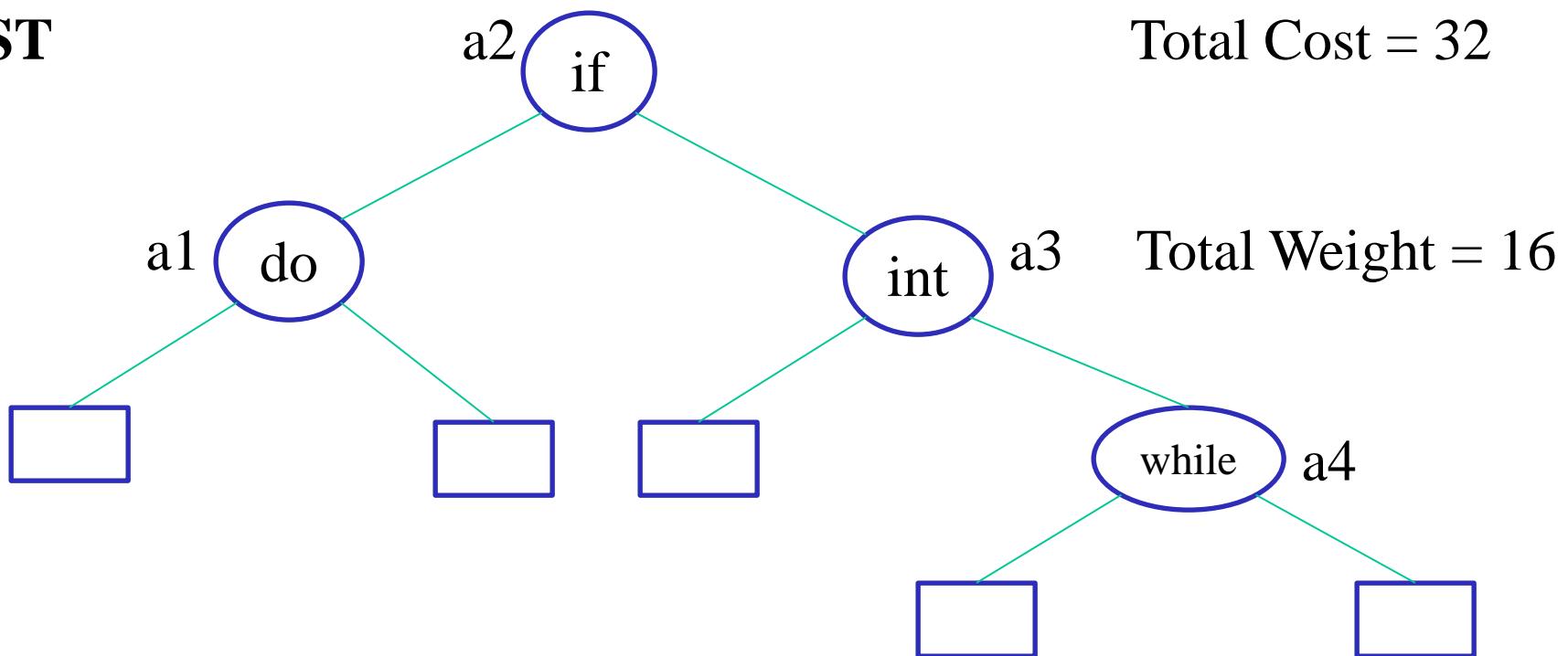
Example : Construct OBST for the following instance,

$$n = 4, \quad (a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$$

$$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$$

$$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$$

OBST



Unit II : Dynamic Programming

Analysis of Algorithm OBST :

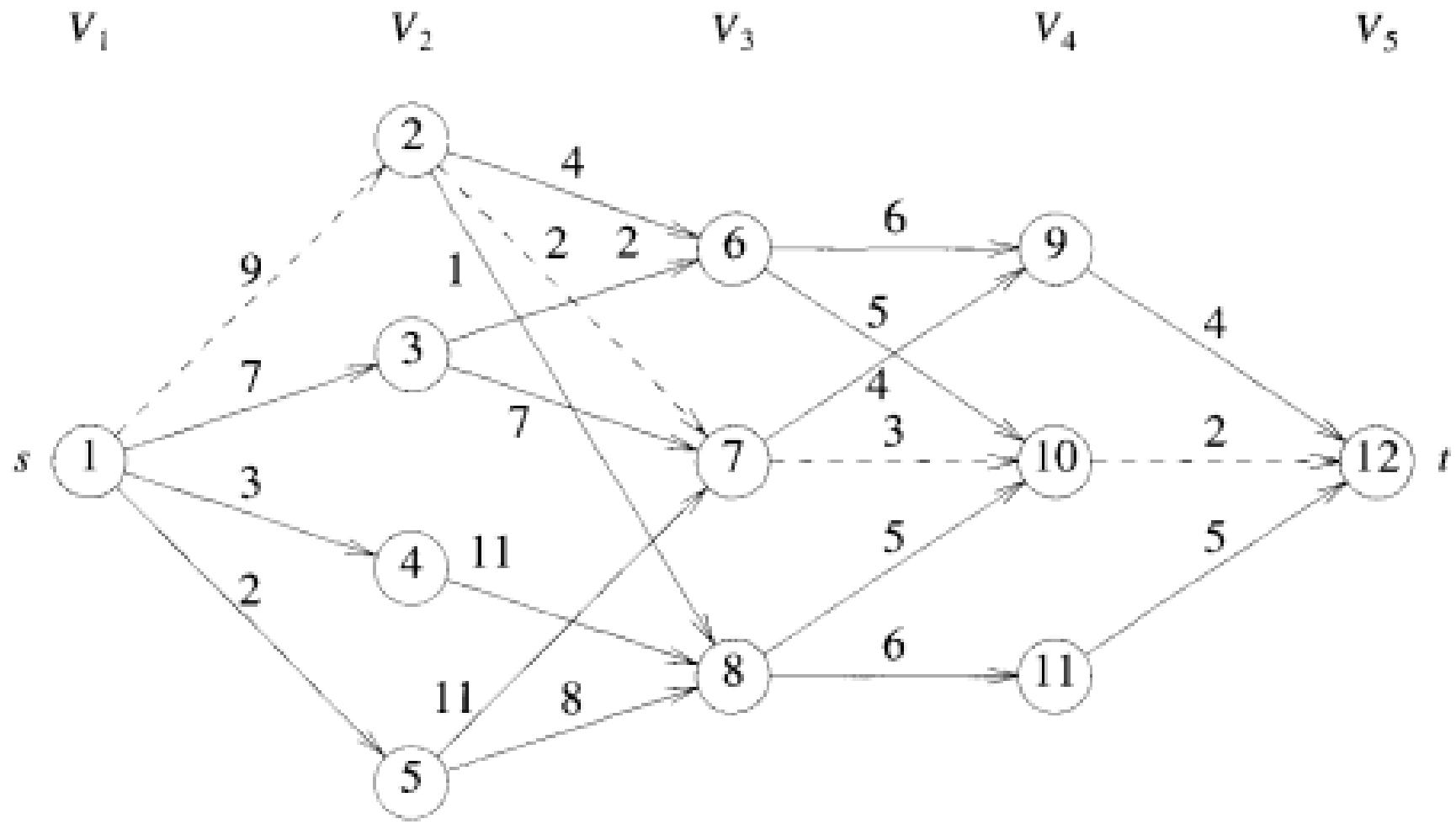
- To get OBST we can compute the values of c's and r's by using equations (1), (4) and (5).
- We need to compute $c(i, j)$ for $(j - i) = 1, 2, \dots, n$, in that order.
When $(j - i) = m$, there are $(n - m + 1)$ values of $c(i, j)$'s to compute.
- Computation of each $c(i, j)$ requires to compute ' 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
$$= \sum_{1 \leq m \leq n} (nm - m^2) = O(n^3)$$
- 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**.

Multistage Graph Problem

Unit II : Dynamic Programming

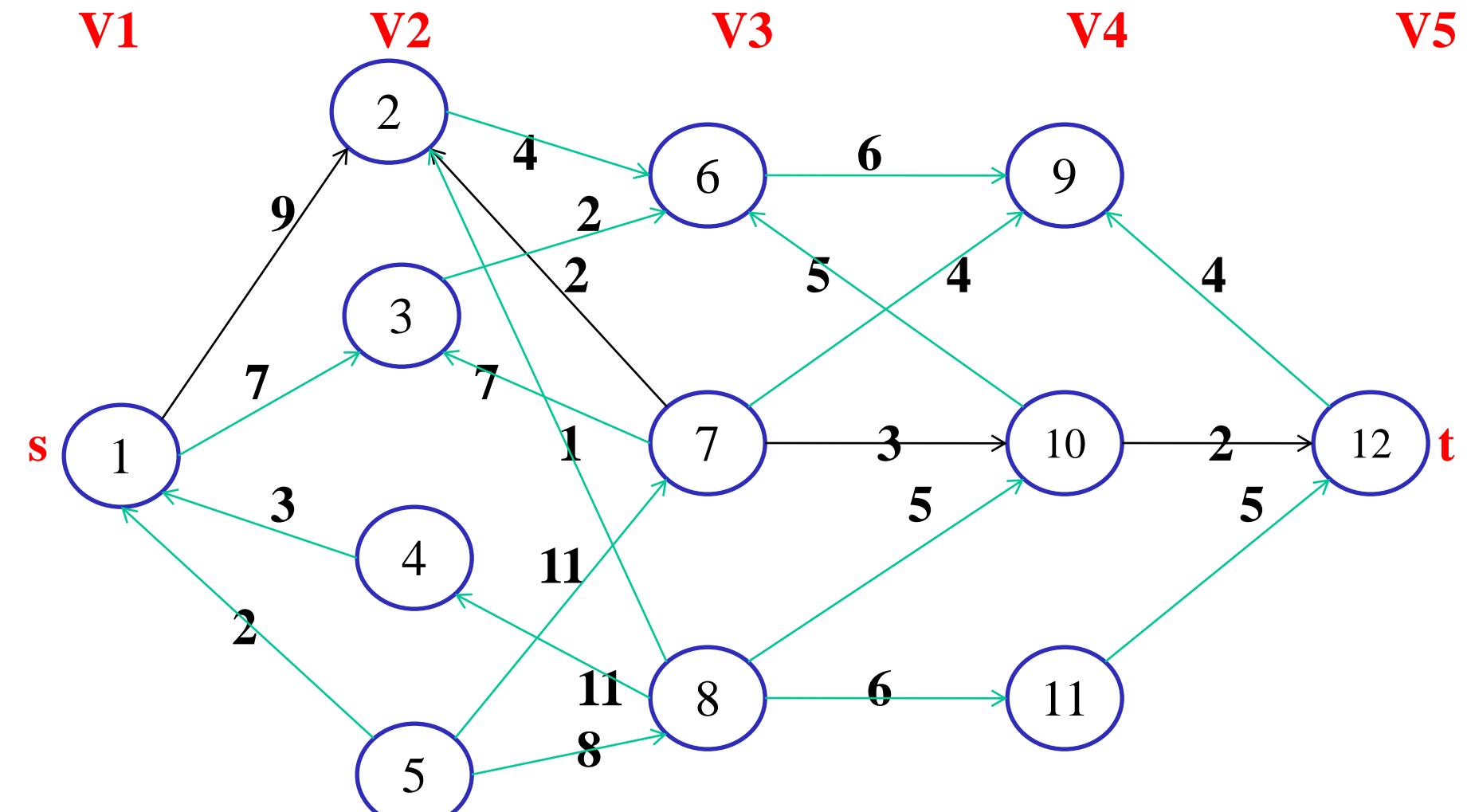
Multi-stage Graphs : Problem Definition

- A multi-stage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i , $1 \leq i \leq k$, and $|V_1| = |V_k| = 1$.
- Each set V_i defines a **stage** in the graph.
- Let ‘s’ and ‘t’ be vertices in V_1 and V_k respectively. The vertex ‘s’ is called as a **source** and vertex ‘t’ is called as a **sink**. For any edge $\langle i, j \rangle$, $i \in \text{stage } V_k$ and $j \in \text{stage } V_{k+1}$, $1 \leq i < k$.
- Let $c(i, j)$ be the cost of an edge $\langle i, j \rangle$. 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 stage 2, then to stage 3 and so on and eventually terminates in stage k .



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 i th decision involves determining which vertex in V_i , $1 \leq i \leq (k - 2)$, is to be on the path. Hence POO applies.
- Let $p(i, j) =$ minimum cost path from vertex j in V_i to vertex 't'
 $\text{cost}(i, j) =$ the cost of the path $p(i, j)$.

$c(j, l) =$ the cost of edge from vertex j in V_i & vertex l in V_{i+1}

$$\begin{aligned} \bullet \text{cost}(i, j) &= \min\{c(j, l) + \text{cost}(i+1, l)\} \\ l &\in V_{i+1} \text{ and } j, l \in E \end{aligned}$$

Unit II : Dynamic Programming

Multi-stage Graphs : Solution by DP

Forward Approach :

$$\begin{aligned} \text{•cost } (i, j) = \min\{c(j, l) + \text{cost}(i+1, l)\} \\ l \in V_{i+1} \text{ and } < j, l > \in E \end{aligned}$$

Backward Approach :

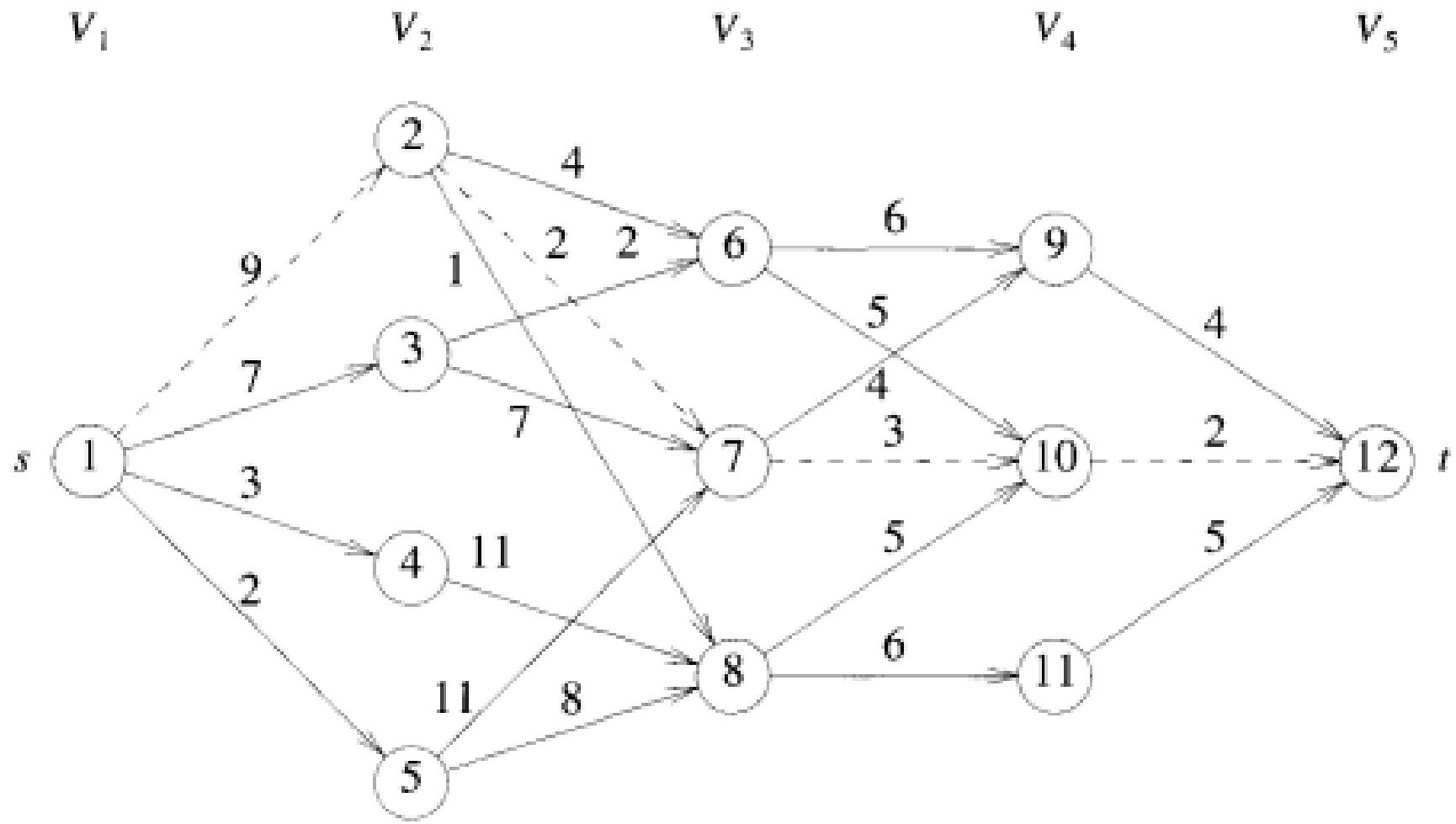
$$\begin{aligned} \text{•bcost } (i, j) = \min\{c(l, j) + \text{bcost}(i-1, l)\} \\ l \in V_{i-1} \text{ and } < l, j > \in E \end{aligned}$$

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

Analysis (Fgraph and Bgraph):

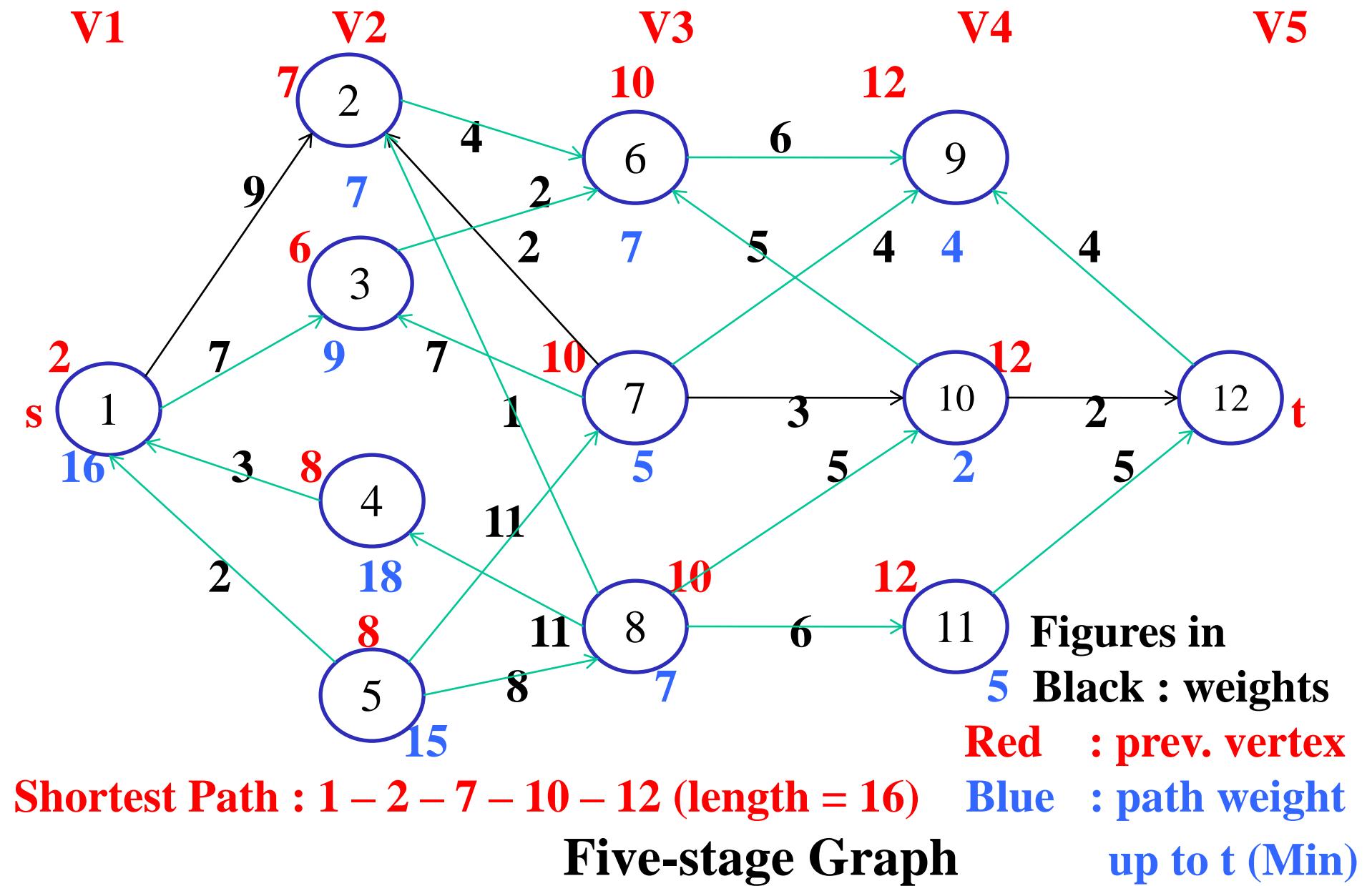
Time Complexity = $\theta(|V| + |E|)$

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



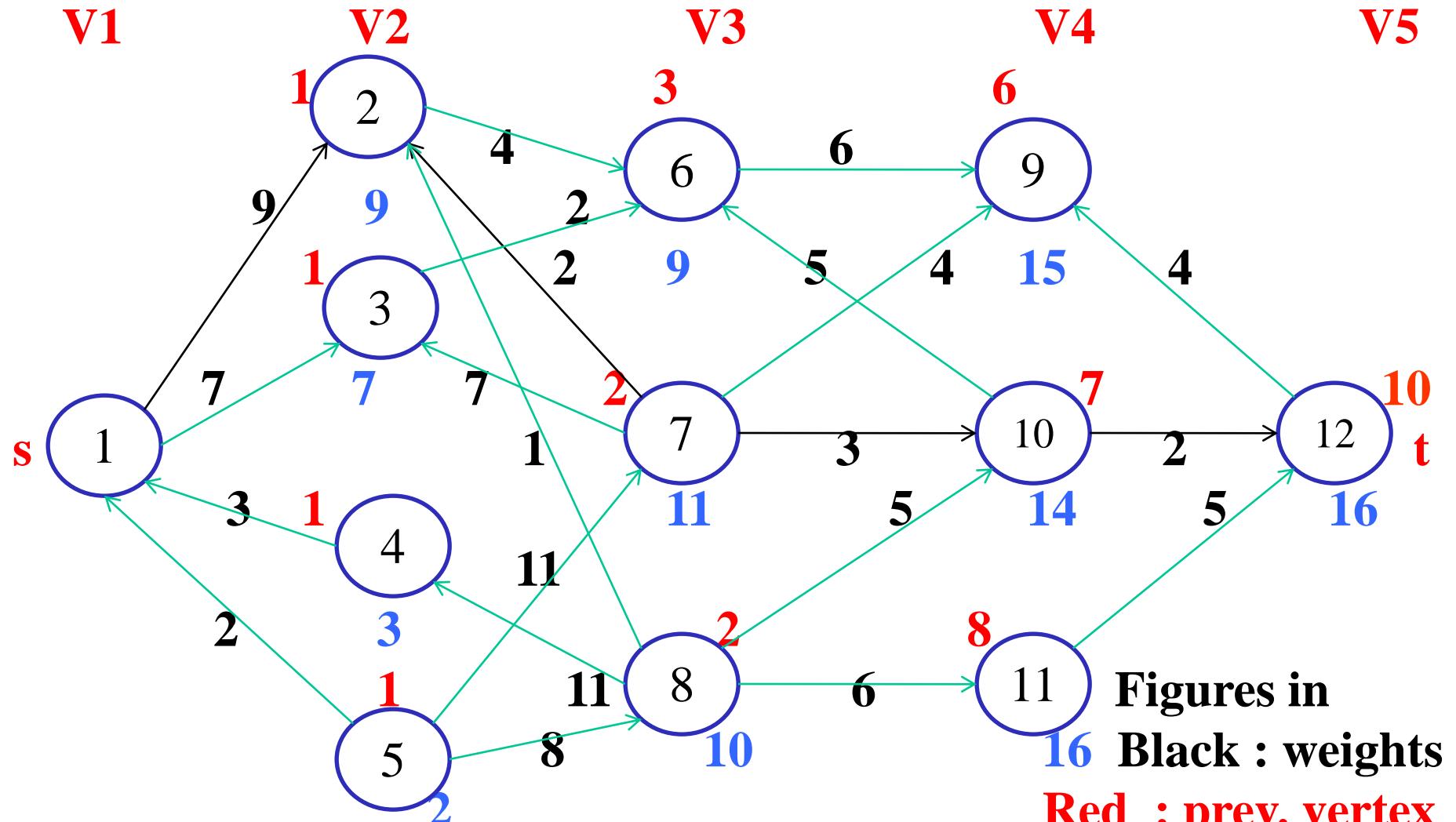
Unit II : Dynamic Programming

Multi-stage Graphs : Solution by Forward Approach



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

Pseudocode for forward approach

```
1  Algorithm FGraph( $G, k, n, p$ )
2    // The input is a  $k$ -stage graph  $G = (V, E)$  with  $n$  vertices
3    // indexed in order of stages.  $E$  is a set of edges and  $c[i, j]$ 
4    // is the cost of  $\langle i, j \rangle$ .  $p[1 : k]$  is a minimum-cost path.
5    {
6       $cost[n] := 0.0$ ;
7      for  $j := n - 1$  to 1 step  $-1$  do
8        { // Compute  $cost[j]$ .
9          Let  $r$  be a vertex such that  $\langle j, r \rangle$  is an edge
10         of  $G$  and  $c[j, r] + cost[r]$  is minimum;
11          $cost[j] := c[j, r] + cost[r]$ ;
12          $d[j] := r$ ;
13       }
14       // Find a minimum-cost path.
15        $p[1] := 1; p[k] := n$ ;
16       for  $j := 2$  to  $k - 1$  do  $p[j] := d[p[j - 1]]$ ;
17     }
```

Pseudocode for Backward approach

```
1  Algorithm BGraph( $G, k, n, p$ )
2    // Same function as FGraph
3    {
4       $bcost[1] := 0.0;$ 
5      for  $j := 2$  to  $n$  do
6        { // Compute  $bcost[j]$ .
7          Let  $r$  be such that  $\langle r, j \rangle$  is an edge of
8             $G$  and  $bcost[r] + c[r, j]$  is minimum;
9           $bcost[j] := bcost[r] + c[r, j];$ 
10          $d[j] := r;$ 
11       }
12       // Find a minimum-cost path.
13        $p[1] := 1; p[k] := n;$ 
14       for  $j := k - 1$  to 2 do  $p[j] := d[p[j + 1]];$ 
15     }
```

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 $n! \gg 2n$ for large value of n.

TSP Problem : Let $G = (V, E)$ be a directed graph with edge costs c_{ij} for the edge $\langle i, j \rangle \in E$. If there is no edge between i and j then $c_{ij} = \infty$. 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 $\langle 1, k \rangle$ for some $k \in V - \{1\}$ 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.

$$\text{Optimal Tour} = g(1, V-\{1\}) = \min \{ c_{1k} + g(k, V-\{1, k\}) \} \dots \quad (1)$$

$$2 \leq k \leq n$$

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,

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

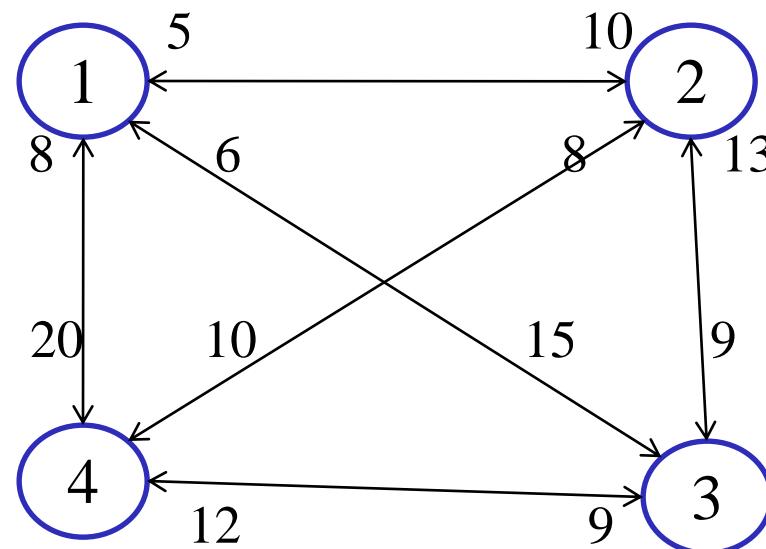
•Equation (2) can be used to find all values of g for $|S| = 0, |S| = 1, |S| = 2, |S| = 3$, and so on ... until $|S| = n - 2$, with the knowledge that $g(i, \emptyset) = c_{i1}$, for $1 \leq i \leq n$

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.

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



Graph $G = (V, E)$

$V = \{1, 2, 3, 4\}$

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Example : $\begin{array}{cccc|c} 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 10 & 15 & 20 \\ 2 & 5 & 0 & 9 & 10 \\ 3 & 6 & 13 & 0 & 12 \\ 4 & 8 & 8 & 9 & 0 \end{array}$
$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad (5.19)$$

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

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

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

Using (5.20) we obtain

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

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

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

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Analysis of algorithm tsp-dp :

- Let $N = \text{number of } g(i, S) \text{ values to be computed}$
 $= \text{number of } g(i, S) \text{ values for } |S| = 0 +$
 $\quad \text{number of } g(i, S) \text{ values for } |S| = 1 +$
 $\quad \text{number of } g(i, S) \text{ values for } |S| = 2 +$
 $\quad \dots +$
 $\quad \text{number of } g(i, S) \text{ values for } |S| = n - 2$

 $= 3*1 + 3*2 + 3*1 \quad \dots \text{ for } n = 4$
 $= 3*C(2,0) + 3*C(2,1) + 3*C(2,2)$
 $= (4 - 1) * [C(2,0) + C(2,1) + C(2,2)]$
 $= (n - 1) * [C(n-2, 0) + C(n-2, 1) + \dots + C(n-2, n-2)]$
 $= (n - 1) * \sum_{k=0}^{n-2} C(n - 2, k) \quad \dots \text{ In General}$
 $= (n - 1) * 2^{n-2}$

Dynamic Programming

Traveling Salesperson Problem (TSP) :

Analysis of algorithm tsp-dp :

$$\begin{aligned} N &= (n - 1) * 2^{n-2} \\ &= O(n^2 2^n) = \text{total number of } g \text{ values} \end{aligned}$$

Time Complexity = $O(n^2 2^n)$ better than $O(n!)$

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 = $O(n^2 2^n)$ 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

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

| | |
|---------------------|---------------------------------------|
| Unit III | Backtracking, Branch and Bound |
|---------------------|---------------------------------------|

Backtracking: The General Method
8 Queen's problem,
Graph Coloring

Branch and Bound: 0/1 Knapsack,
Traveling Salesperson Problem.

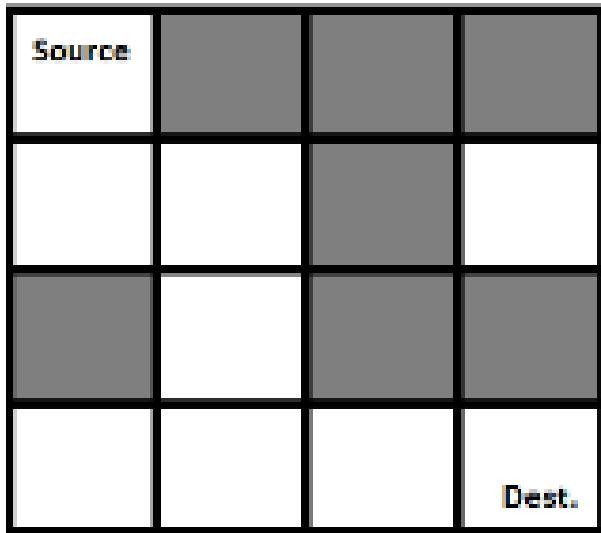
Backtracking

Backtracking

- Suppose you have to make a **series of decisions**, among various *choices*, where
 - You don't have enough information to know what to choose
 - Each decision leads to a new set of choices
 - Some sequence of choices (possibly more than one) may be a solution to your problem

Backtracking

- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”
- Sample problem : MAZE PROBLEM and its binary matrix representation



{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Backtrack method

- Mostly, desired solution is expressible as an n-tuple (x_1, x_2, \dots, x_n) , x_i is chosen from a finite set S_i
- Often, problem calls for finding one vector that **maximizes (minimizes/satisfies)** a *criterion function* $P(x_1, \dots, x_n)$
- Sometimes, all vectors satisfying P are required

Example :Sorting a[1:n]: *for*

purpose of understanding only

- **Backtrack solution** ([this sorting, never used practically](#))
 - Solution is n-tuple
 - Where, x_i is index in a of the i th smallest element
 - Criterion function P is
 - $a[x_i] \leq a[x_{i+1}] \quad \text{for } 1 \leq i < n$
 - Set S_i is a finite set containing integers 1 to n

Comparison with brute force method

- Say set S_i has m_i elements
- There are $m = m_1 m_2 \dots m_n$ n-tuples candidate for satisfying the function P
- Brute force – form all these n-tuples
 - Evaluate each
 - Save the best (optimal)
- Backtracking – solution in far fewer than above m solutions

Basic Idea of backtracking

- Build up solution vector, one component at a time
- Use *modified criterion functions* $P_i(x_1, \dots, x_n)$, sometimes called *bounding functions* to test whether the vector being formed has any chance of success
- If partial vector (x_1, \dots, x_i) cannot lead to success, then $m_{i+1} \dots m_n$ possible test vectors can be ignored entirely

Contd..

- Backtracking is more advanced and optimized than Brute Force.
- It usually uses recursion
- It is applied to problems having low input constraints (for example $N \leq 20$).

Constraints

- Many problems using backtracking require that all solutions **satisfy a complex set of constraints**
- **2 categories of constraints**
 - Implicit
 - Explicit

Explicit constraint

- It restricts x_i s to take values only from a given set
 - $x_i > 0$
 - $x_i = 0$ or 1
 - $j < x_i < k$ (i.e. between j and k)
- It depends on the particular instance I of the problem to be solved
- All tuples that **satisfy** the explicit constraints define a possible solution space for I

Implicit constraints

- Rules that determine which of the tuples in the solution space of I satisfy the **criterion function**
- They describe the way in which the **x_i must relate to each other**

Backtracking algorithms

- Find a solution by trying one of several choices.
- If the choice proves incorrect, computation *backtracks* or restarts at the point of choice and tries another choice.
- It is often convenient to maintain choice points and alternate choices using *recursion*.
- *Conceptually, a backtracking algorithm does a depth-first search of a tree of possible (partial) solutions. Each choice is a node in the tree.*

Revision of backtracking algorithm

- **Backtracking algorithms** try each possibility until they find the right one.
- It is a **depth-first search** of the set of possible solutions.
- During the search, if an alternative doesn't work, the search **backtracks** to the choice point, the place which presented different alternatives, and tries the next alternative.

Contd..

- When the alternatives are exhausted, the search returns to the **previous choice point** and tries the next alternative there.
- If there are no more choice points, the search fails.

8 Queens

8 Queens problem

- Place 8 queens on a 8 by 8 chessboard so that no 2 queens attack
 - No 2 queens are on the same row, col or diagonal
- Number the rows, cols as : 1,2,3,4,5,6,7,8
- Sol is 8 tuple, representing 8 rows
 $(_, _, _, _, _, _, _, _)$
- Values in the tuple represent column at that row
- $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$

8 Queens : Explicit constraint

- It restricts x_i s to take values only from a given set
 - $x_i = 1, 2, 3, 4$ for 4 Queens
 - $x_i = 1, 2, 3, 4, 5, 6, 7$ for 7 Queens
 - $x_i = 1, 2, 3, 4, 5, 6, 7, 8$ for 8 Queens
- It depends on the particular instance I of the problem to be solved
- All tuples that **satisfy** the explicit constraints define a possible solution space for I

8 Queens : Implicit constraints

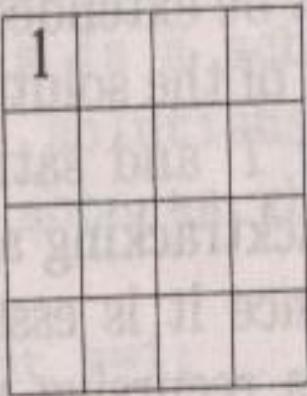
Rules that describe how **x_i must relate to each other** – i.e. satisfy **criterion function**

- no 2 x_i s must be same i.e, no 2 queens in same column
- And, no 2 in the same diagonal

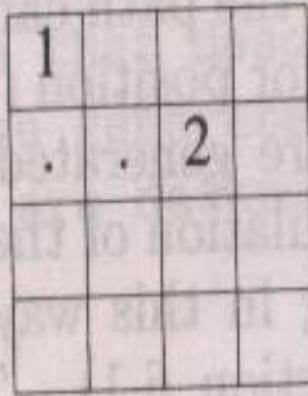
(Rules that determine which of the tuples in the solution space of I satisfy the **criterion function**)

They describe the way in which the **x_i must relate to each other**)

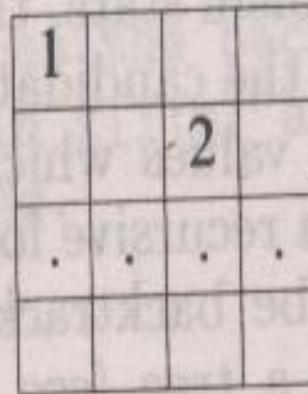
Backtracking: 4Queens



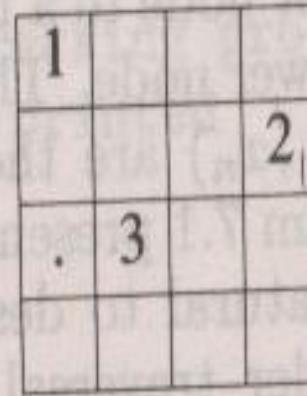
(a)



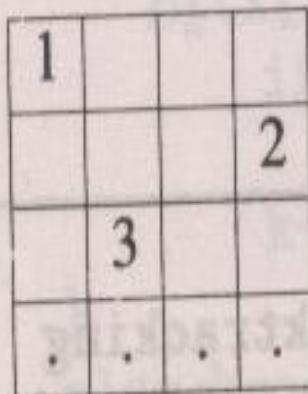
(b)



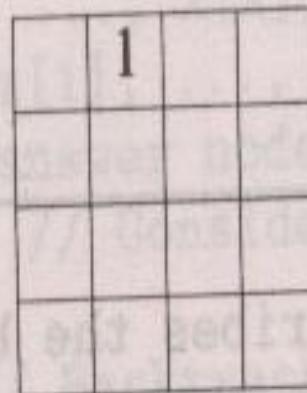
(c)



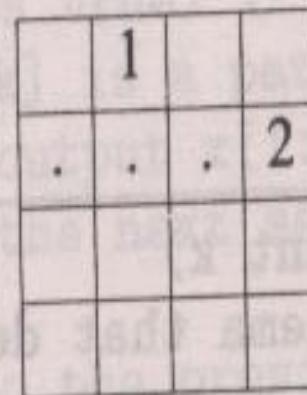
(d)



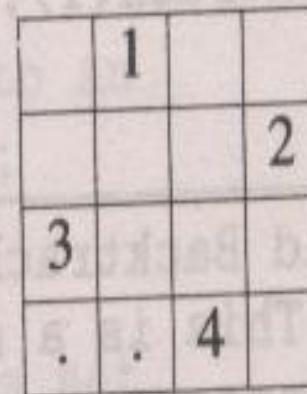
(e)



(f)

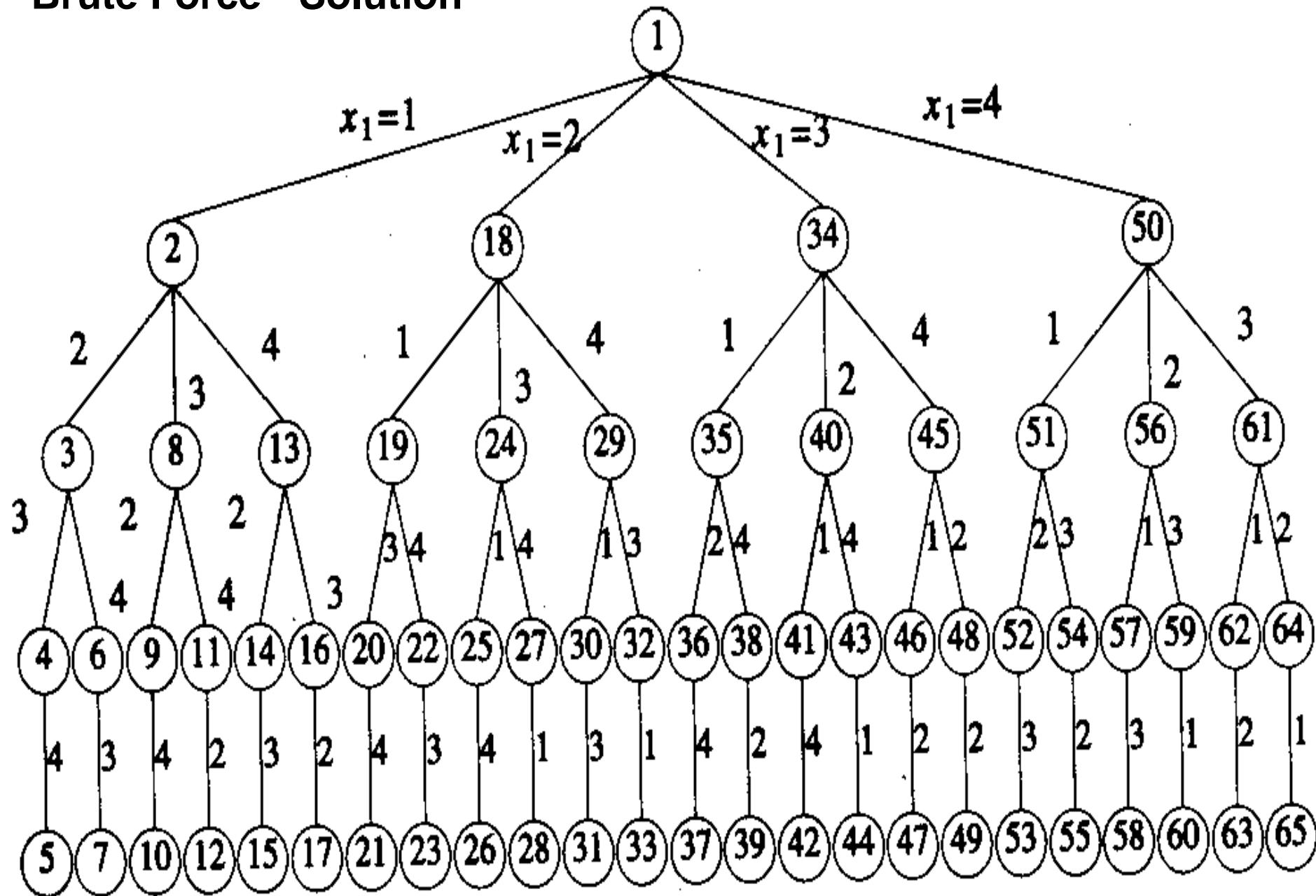


(g)

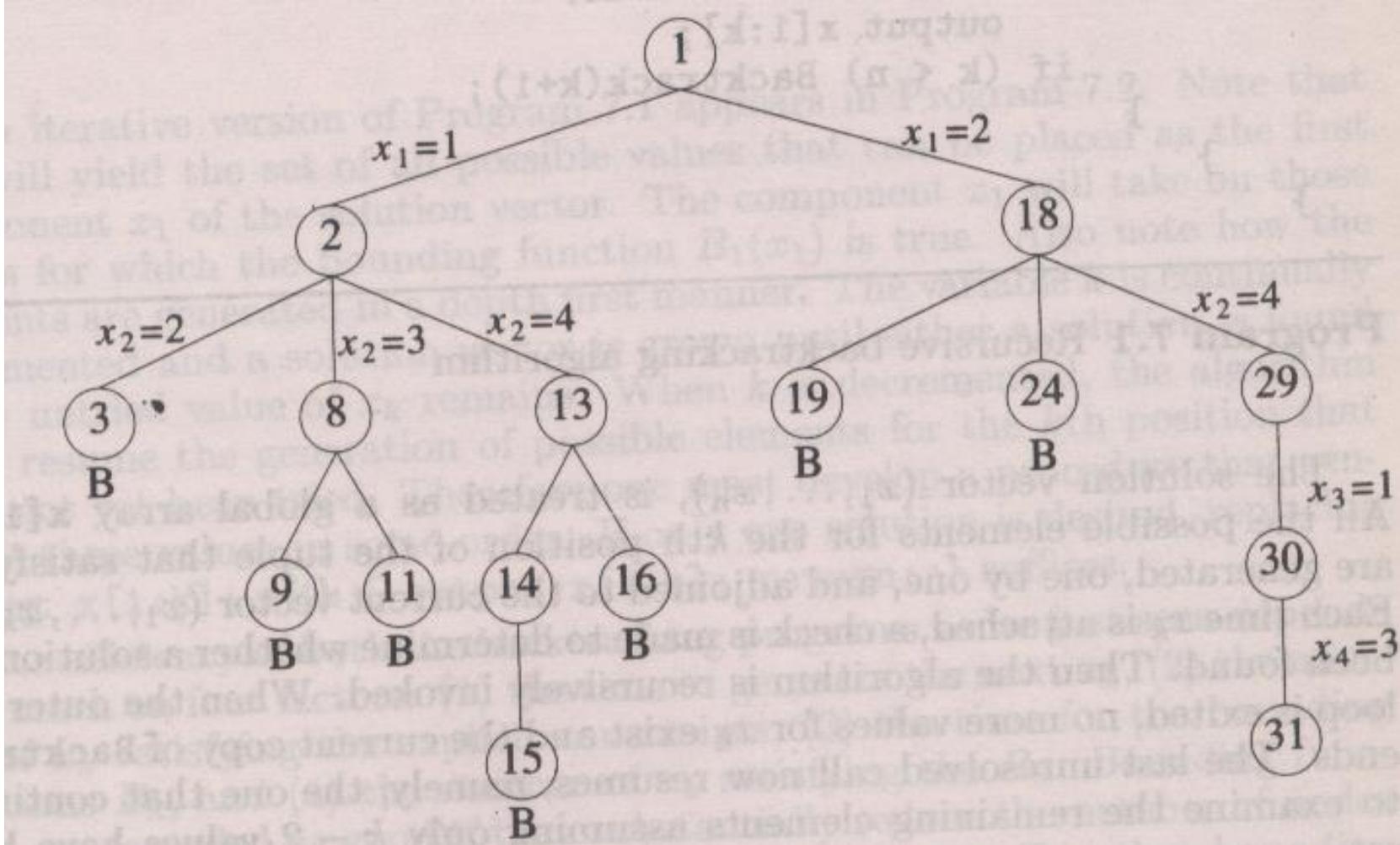


(h)

Brute Force - Solution



Backtrack - Solution



Tree organization of **solution space** (terminologies)

- Each NODE= ***problem state***
- All Paths from root to node = ***state space***
- ***Path*** from root to a state S if defines tuple in solution space , implies S is ***Solution state***

Some terminologies

Answer states are those solution states s for which the path from **root** to s defines a tuple that is a member of the set of solutions of the problem i.e. those that satisfy *implicit constraints*.

State space tree : tree organization of solution space

Terminologies Continued...

- **Live node** : A Node which has been generated and all of whose children have not yet been generated
- **E-node** : The live node whose children are currently being generated.
- **Dead node** : It is a generated node which is not to be further expanded.
- Depth first node generation with **bounding function** is called **backtracking**
- In **branch and bound** E-Node remains E-Node till it is dead

Terminologies Continued...

- **Static state space tree**
organizations:

Tree organizations that are
INDEPENDENT
of the problem instance being
solved, as shown on the right

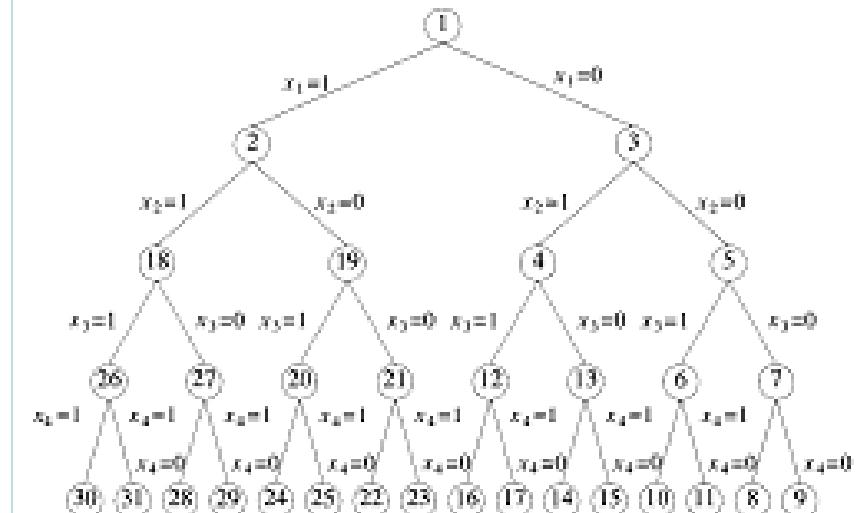
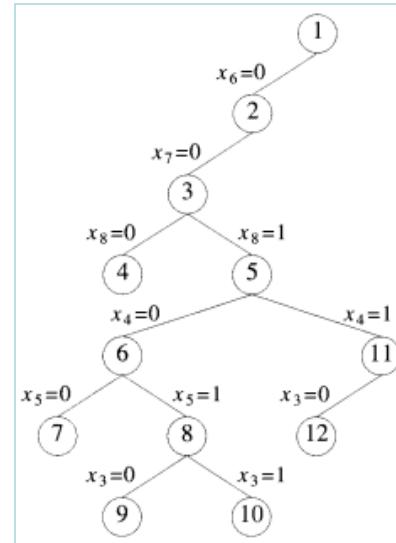
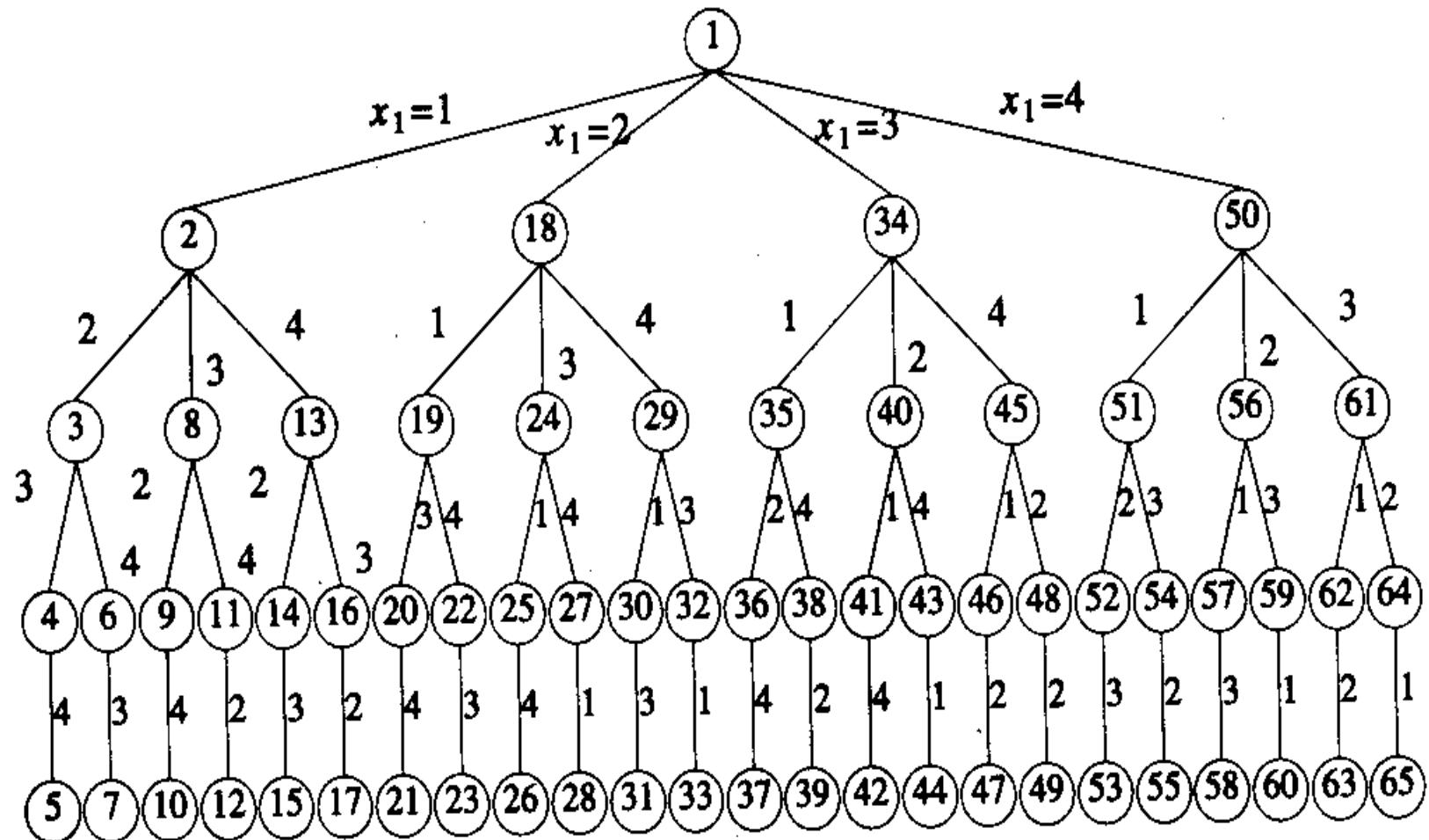


Figure 7.4. Another possible organization for the sum of subsets problems.
Nodes are numbered as in D-search.

- **Dynamic state space tree**
organizations:
Tree organizations that are
DEPENDENT on
the problem instance being solved
i.e 0/1 knapsack

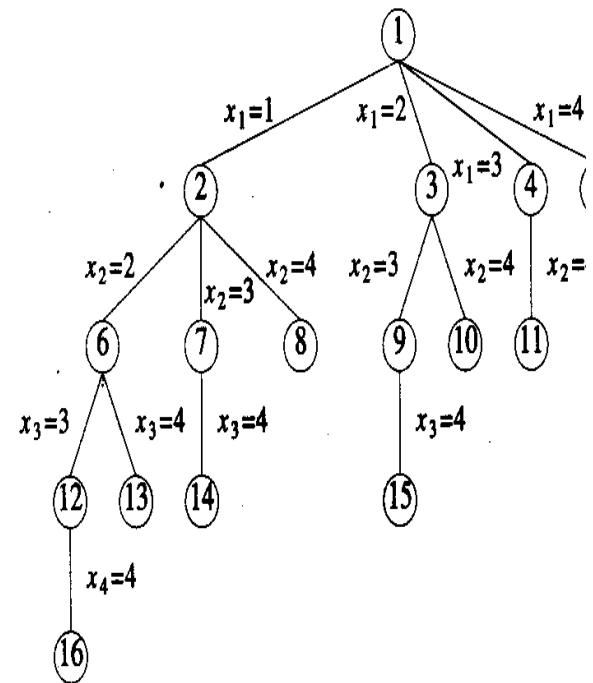
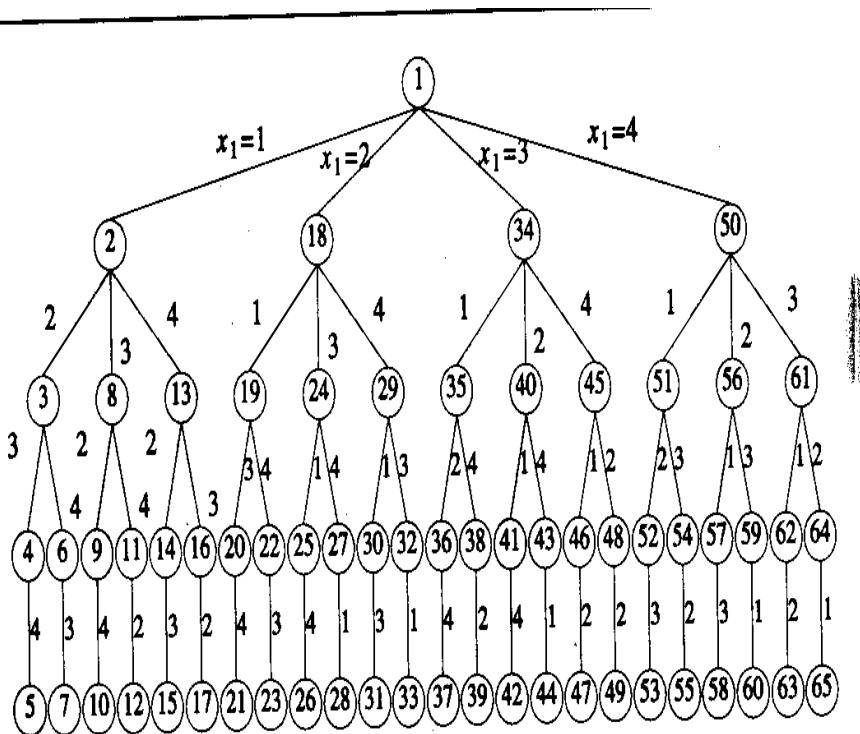


Static State Space tree for 4 queens problem



State Space tree for 4 queens problem

Example Solution: Using Backtracking



8 Queens Problem

- Generalization of problem to $n \times n$ chessboard.
- If Chessboard squares are numbered as indices of two dimensional array $[1 \dots n, 1 \dots n]$ then we observe that every element on same diagonal that runs from upper left to lower right has same row – column value. Example : $a[3,1], a[5,3], a[6,4], a[7,5], a[8,6]$. All these have row – column value = 2
- Every element on same diagonal that runs from upper right to lower left has same row + column value. Example : $a[1,5], a[2,4], a[3,3], a[4,2], a[5,1]$
- Suppose two queen are placed at position (i,j) and (k,l) then from above rules we have :
$$(i-j) = (k-l) \text{ or } (i+j) = (k+l)$$
First Equation implies $j - l = i - k$
&
Second Equation Implies $j - l = k - i$
- There fore two queens lie on same diagonal if and only if $|j - l| = |k - i|$

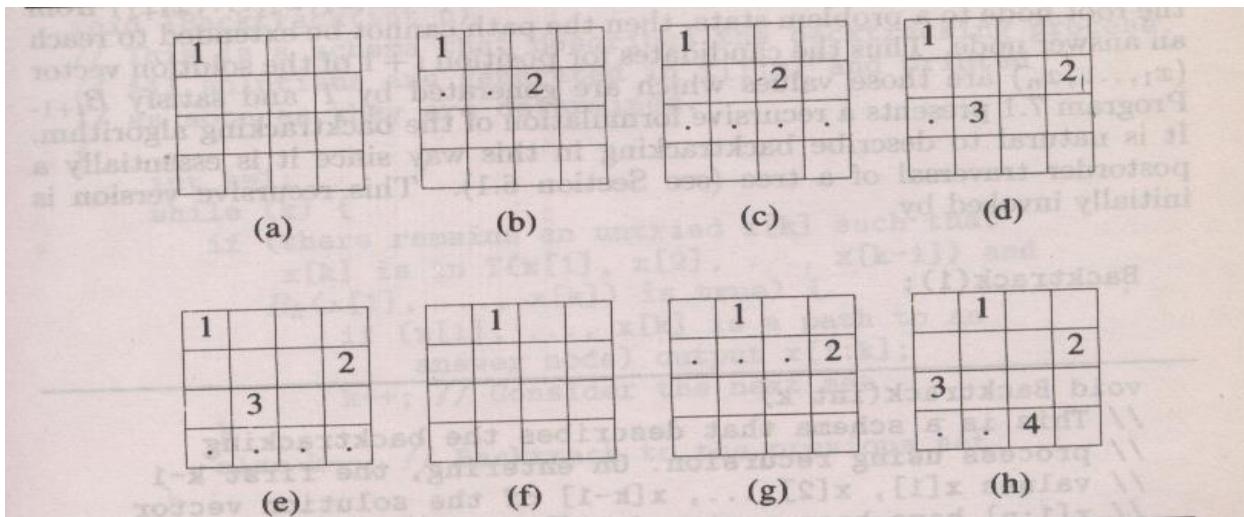
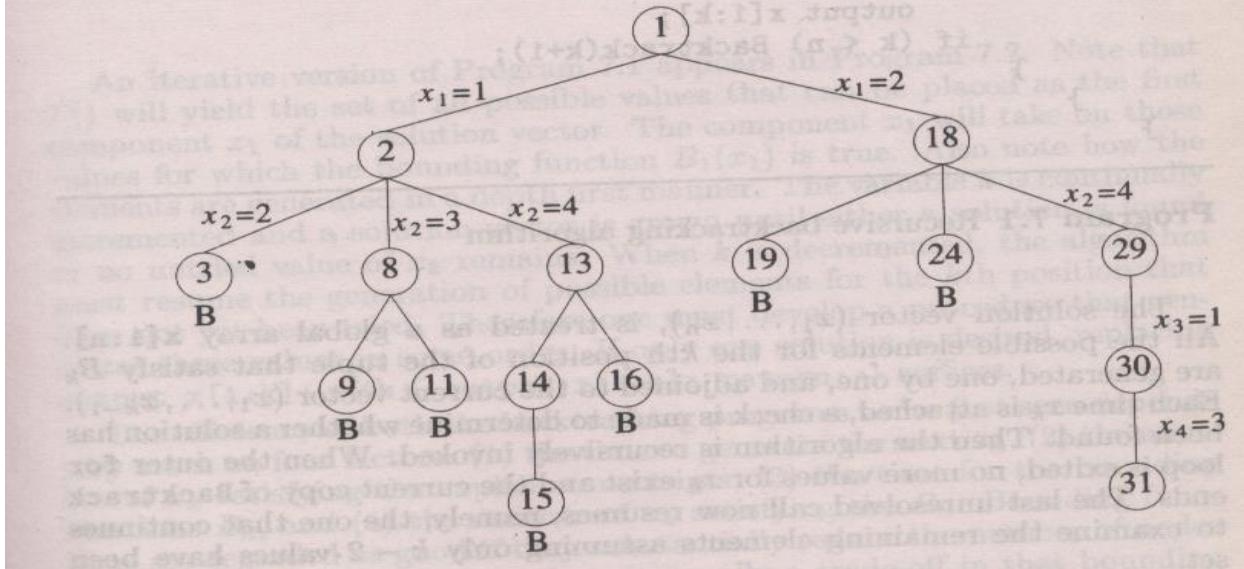


Figure 7.5 Example of a backtrack solution to the 4-queens problem



```
1  Algorithm Place( $k, i$ )
2  // Returns true if a queen can be placed in  $k$ th row and
3  //  $i$ th column. Otherwise it returns false.  $x[ ]$  is a
4  // global array whose first  $(k - 1)$  values have been set.
5  //  $\text{Abs}(r)$  returns the absolute value of  $r$ .
6  {
7      for  $j := 1$  to  $k - 1$  do
8          if  $((x[j] = i) \text{ // Two in the same column}$ 
9              or  $(\text{Abs}(x[j] - i) = \text{Abs}(j - k)))$ 
10             // or in the same diagonal
11             then return false;
12     return true;
13 }
```

orithm 7.4 Can a new queen be placed?

```
1  Algorithm NQueens( $k, n$ )
2    // Using backtracking, this procedure prints all
3    // possible placements of  $n$  queens on an  $n \times n$ 
4    // chessboard so that they are nonattacking.
5    {
6      for  $i := 1$  to  $n$  do
7        {
8          if Place( $k, i$ ) then
9            {
10               $x[k] := i;$ 
11              if ( $k = n$ ) then write ( $x[1 : n]$ );
12              else NQueens( $k + 1, n$ );
13            }
14        }
15    }
```

orithm 7.5 All solutions to the n -queens problem

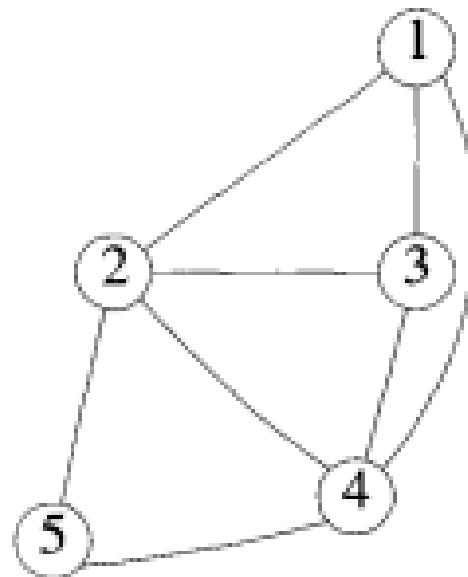
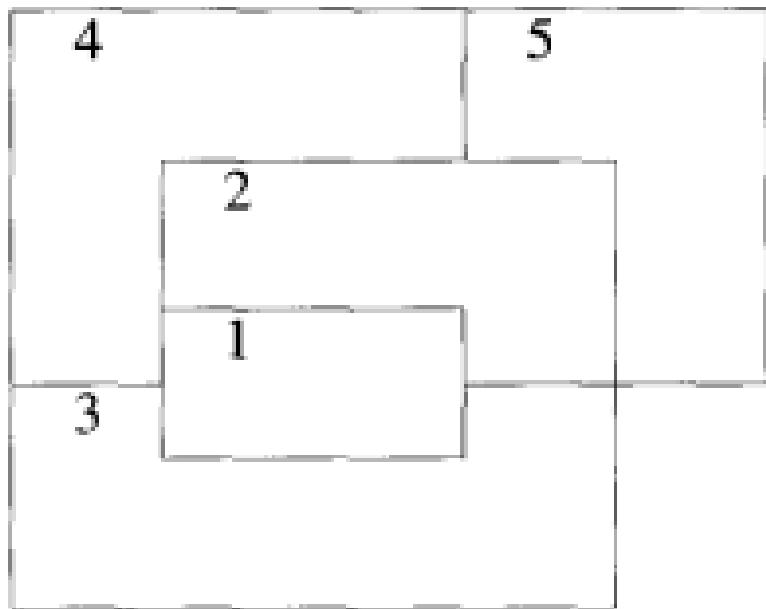
Analysis of N Queens problem

- Using brute force approach for an 8x8 chess board there are $(64 \text{ C } 8)$ possible ways to place 8 queens or approximately 4.4 billion 8 tuples to examine.
 - (Using brute force approach for an $n \times n$ chess board there are $(n^2 \text{ C } n)$ possible ways to place n queens or n tuples to examine)
- However using NQueens function we allow placement of 8 queens on distinct rows and columns, hence we require to examine of at most $8!$ Or only 40,320 tuples
- So time complexity N queens problem is $O(N!)$

Graph Coloring

Backtracking

Planar Graph



Graph Coloring problem

- Map coloring: In cartography, the problem of coloring a map with different colors such that no two adjacent regions have the same color is an instance of the graph coloring problem. This problem arises when creating political maps or geographical maps.

Graph Coloring problem

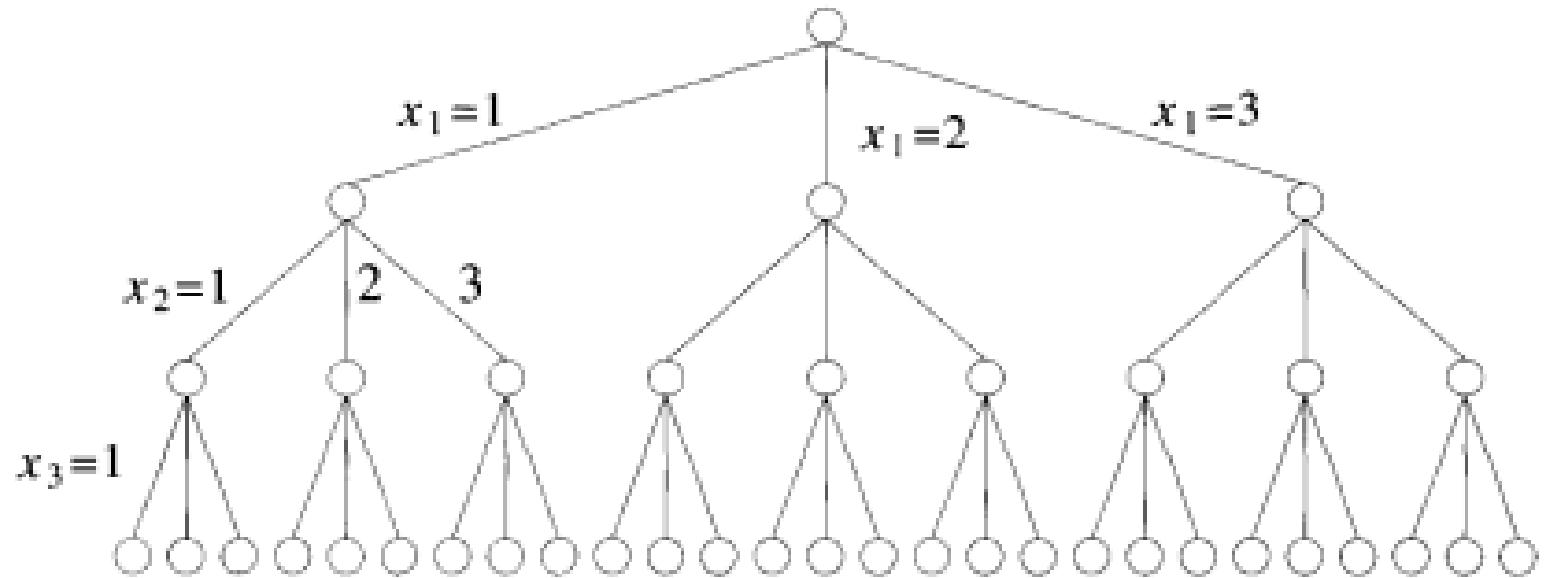
- Assign colors to vertices in a graph such that no two adjacent vertices have the same color.
- i.e. given a graph, we want to color each vertex of the graph with one of the available colors (say, k colors), such that no two adjacent vertices have the same color.
 - The **minimum number of colors** required to color a graph such that no two adjacent vertices have the same color is called the chromatic number of the graph.
- Real life applications:
 - Scheduling problems
 - Register allocation in compilers
 - Frequency assignment in wireless communication networks
 - etc

Graph Coloring Real life Problem :

- Scheduling: Institute situation - Limited number of classrooms, teachers, and time slots available. The scheduling problem can be modeled as a graph, where vertices represent courses and edges represent conflicting time slots. By assigning colors to vertices (courses), we can ensure that no two conflicting courses are scheduled at the same time.
- Timetabling: In universities or schools, there are multiple courses that need to be scheduled, and the problem is to schedule them in such a way that there are no conflicts. This problem can be formulated as a graph coloring problem, where vertices represent courses and edges represent conflicts between courses. By assigning colors to vertices, we can ensure that no two conflicting courses are scheduled at the same time.
- Wireless channel allocation: In wireless communication networks, channels need to be assigned to different users to avoid interference. This problem can be formulated as a graph coloring problem, where vertices represent users and edges represent interference between users. By assigning colors (channels) to vertices, we can ensure that no two adjacent users are assigned the same channel.
- Register allocation in compilers: When compiling a program, the compiler needs to allocate registers to different variables to optimize the use of memory. This problem can be modeled as a graph coloring problem, where vertices represent variables and edges represent conflicts between variables.

Graph Coloring problem

- The problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve the problem exactly for all possible inputs.
- However, there are many efficient algorithms and heuristics that can solve the problem approximately, with varying degrees of accuracy and efficiency.



State space tree for mColoring when $n = 3$ and $m = 3$

Time Complexity of Graph Coloring

- In general, the time complexity of backtracking algorithms for the graph coloring problem can be expressed **as $O(k^n)$** , where **k is the maximum degree of the graph** and **n is the number of vertices**.
 - This is because each vertex can be colored with **at most k colors**, and there are n vertices in the **graph**, so there are at most k^n possible color assignments
- However, **the actual running time** of backtracking algorithms can be much better than $O(k^n)$ if the algorithm can prune large portions of the search space using heuristics or other techniques.

M – Coloring Problem

- It is a **DECISION** problem :-
 - To find if it is **possible** to **assign nodes** with **M** different colors, such that no two adjacent vertices of the graph are of the same colors
 - Ans required is – Y/N or True / false
- This can be used for the **Optimization Problem of Graph coloring** – where we need to find the **MINIMUM** number of colors required to color the nodes of a given graph, G, such that no 2 adjacent nodes have same color

```
1 Algorithm mColoring( $k$ )
2 // This algorithm was formed using the recursive backtracking
3 // schema. The graph is represented by its boolean adjacency
4 // matrix  $G[1 : n, 1 : n]$ . All assignments of  $1, 2, \dots, m$  to the
5 // vertices of the graph such that adjacent vertices are
6 // assigned distinct integers are printed.  $k$  is the index
7 // of the next vertex to color.
8 {
9     repeat
10    { // Generate all legal assignments for  $x[k]$ .
11        NextValue( $k$ ); // Assign to  $x[k]$  a legal color.
12        if ( $x[k] = 0$ ) then return; // No new color possible
13        if ( $k = n$ ) then // At most  $m$  colors have been
14                                // used to color the  $n$  vertices.
15            write ( $x[1 : n]$ );
16            else mColoring( $k + 1$ );
17    } until ( $\text{false}$ );
18 }
```

```

1 Algorithm NextValue( $k$ )
2 //  $x[1], \dots, x[k - 1]$  have been assigned integer values in
3 // the range  $[1, m]$  such that adjacent vertices have distinct
4 // integers. A value for  $x[k]$  is determined in the range
5 //  $[0, m]$ .  $x[k]$  is assigned the next highest numbered color
6 // while maintaining distinctness from the adjacent vertices
7 // of vertex  $k$ . If no such color exists, then  $x[k]$  is 0.
8 {
9     repeat
10    {
11         $x[k] := (x[k] + 1) \bmod (m + 1)$ ; // Next highest color.
12        if ( $x[k] = 0$ ) then return; // All colors have been used.
13        for  $j := 1$  to  $n$  do
14            { // Check if this color is
15                // distinct from adjacent colors.
16                if (( $G[k, j] \neq 0$ ) and ( $x[k] = x[j]$ ))
17                    // If  $(k, j)$  is an edge and if adj.
18                    // vertices have the same color.
19                    then break;
20            }
21            if ( $j = n + 1$ ) then return; // New color found
22    } until (false); // Otherwise try to find another color.
23 }

```

| | |
|---------------------|---------------------------------------|
| Unit III | Backtracking, Branch and Bound |
|---------------------|---------------------------------------|

Backtracking: The General Method
8 Queen's problem,
Graph Coloring

**Branch and Bound: 0/1 Knapsack,
Traveling Salesperson Problem.**

General Strategy: Branch and Bound

The General Technique

- Like Backtracking, this also searches for the solution in state space tree, however, the state space tree is searched in a manner that, all children of the E-node are generated, before any other live node can become E-node.
 - Or we can say that – E-node (node being expanded / whose children are being generated) remains E-node till it is dead (i.e all its children are generated)
 - BFS, and D-Search both can be generalized to Branch and Bound
- Branch and bound can solve optimization problems, such as finding the minimum or maximum of a function or finding the shortest path in a graph.
- In branch and bound, the smaller subproblems are solved recursively. The solutions to the subproblems are used to determine if certain branches of the **solution space** can be pruned, resulting in a more efficient search.
- The implementation can use different strategies. The choice of strategy depends on the nature of the problem and the resources available.

Branch and Bound

- First we need to conceive a state-space tree for the given problem
- Then generate nodes such that E-node remains E-node till all it's children are generated
- Each answer node x has cost $c(x)$ associated with it, and a minimum

3 common strategies for implementing the branch and bound algorithm

- These strategies use queue or stack to store the subproblems to be solved.
 - FIFO (First In, First Out)
 - LIFO (Last In, First Out)
 - LC (Least Cost)
 - FIFO is useful when the solution space is relatively flat and uniform,
 - LIFO is useful when the solution space is deep and narrow,
 - and LC is useful when there is a good heuristic estimate of the optimal solution.

3 common strategies for implementing the branch and bound algorithm

- **FIFO (First In, First Out):** In this strategy, the subproblems are stored in a queue, and the first subproblem to be added to the queue is the first one to be removed for processing. This strategy is also known as breadth-first search because it explores all the subproblems at a given level of the search tree before moving on to the next level.
- **LIFO (Last In, First Out):** In this strategy, the subproblems are stored in a stack, and the last subproblem to be added to the stack is the first one to be removed for processing. This strategy is also known as depth-first search because it explores the deepest branches of the search tree first before backtracking to shallower levels.
- **LC (Least Cost):** In this strategy, the subproblems are sorted by their estimated cost, and the subproblem with the lowest estimated cost is selected for processing first. This strategy is also known as best-first search because it explores the most promising branches of the search tree first. The estimated cost can be based on factors such as the distance to the goal, the remaining time, or the number of unsatisfied constraints.

FIFO – 4 Queens

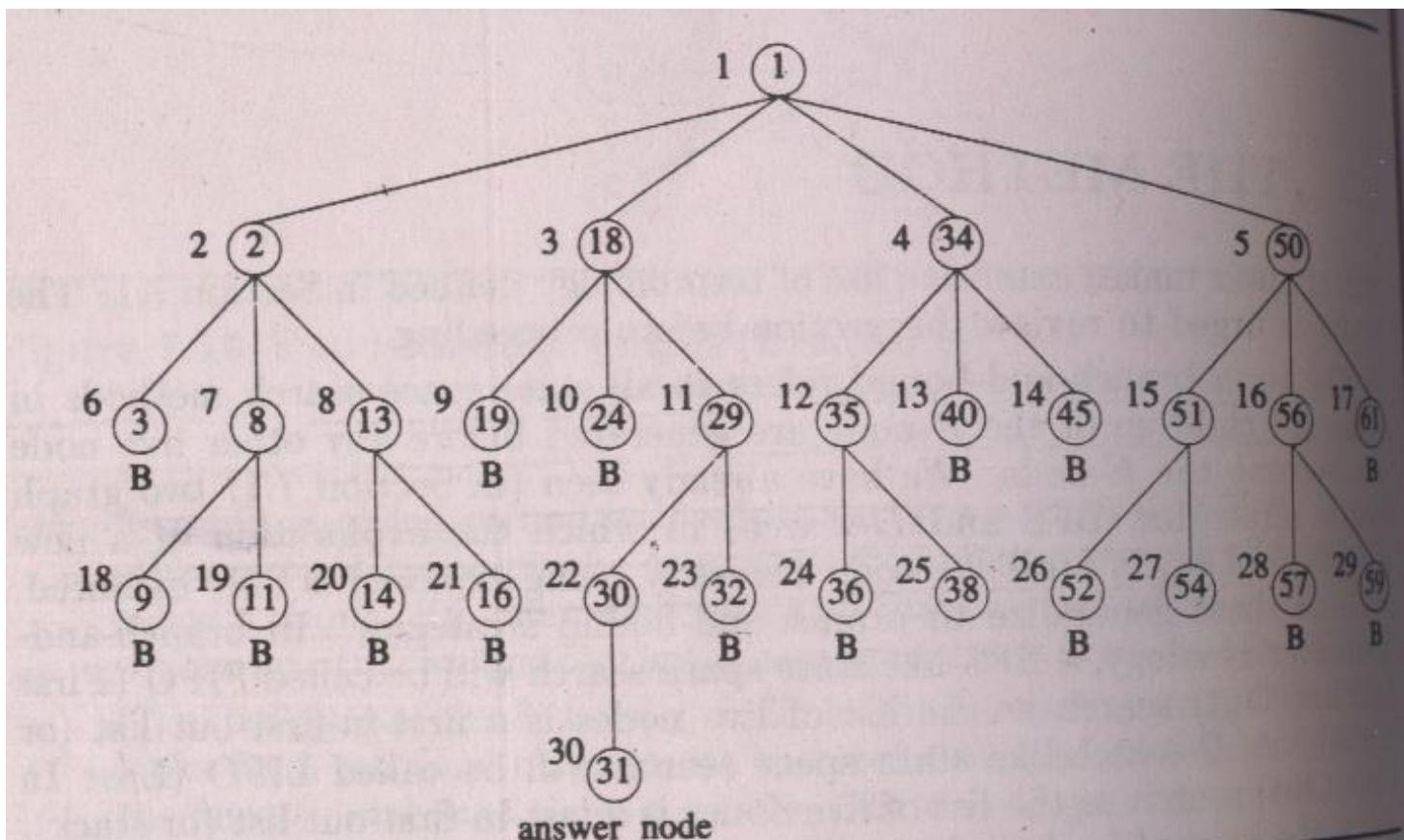


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

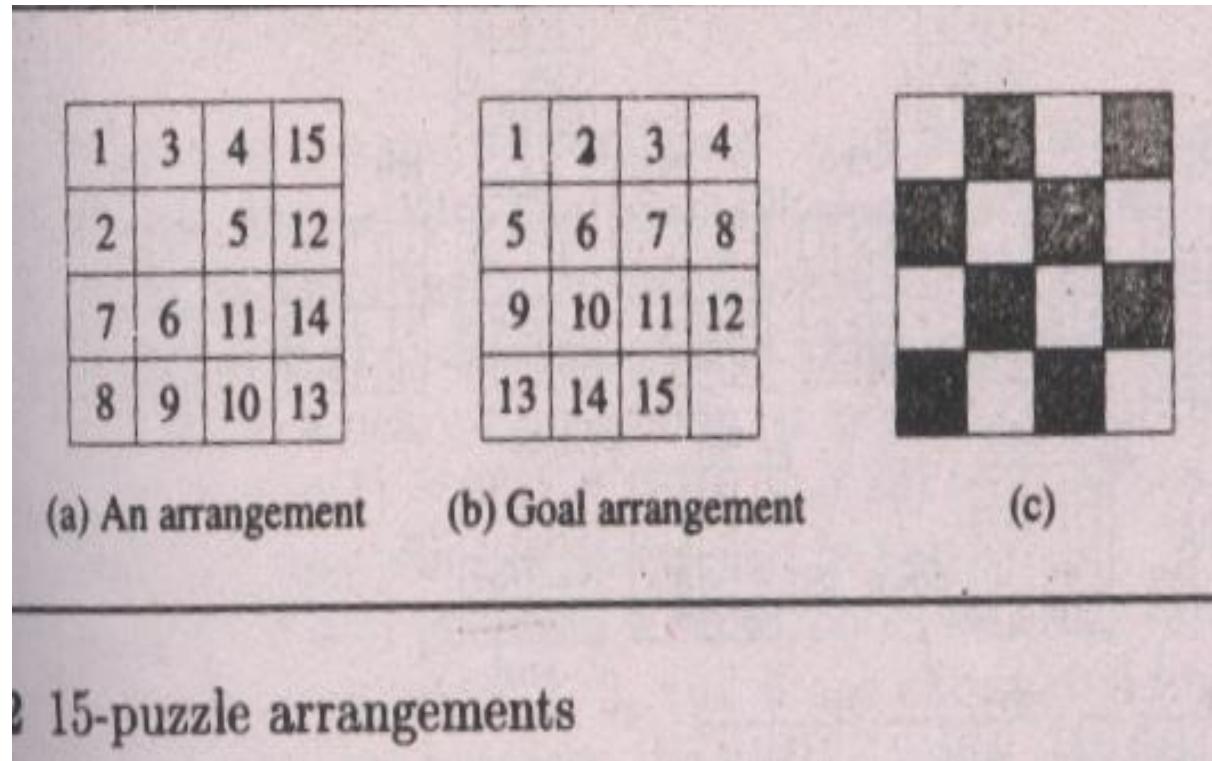
LC Branch and Bound

```
1 Algorithm LSearch( $t$ )
2 // Search  $t$  for an answer node.
3 {
4     if  $*t$  is an answer node then output  $*t$  and return;
5      $E := t$ ; //  $E$ -node.
6     Initialize the list of live nodes to be empty;
7     repeat
8     {
9         for each child  $x$  of  $E$  do
10        {
11            if  $x$  is an answer node then output the path
12                from  $x$  to  $t$  and return;
13            Add( $x$ ); //  $x$  is a new live node.
14             $(x \rightarrow parent) := E$ ; // Pointer for path to root.
15        }
16        if there are no more live nodes then
17        {
18            write ("No answer node"); return;
19        }
20         $E := \text{Least}();$ 
21    } until (false);
22 }
```

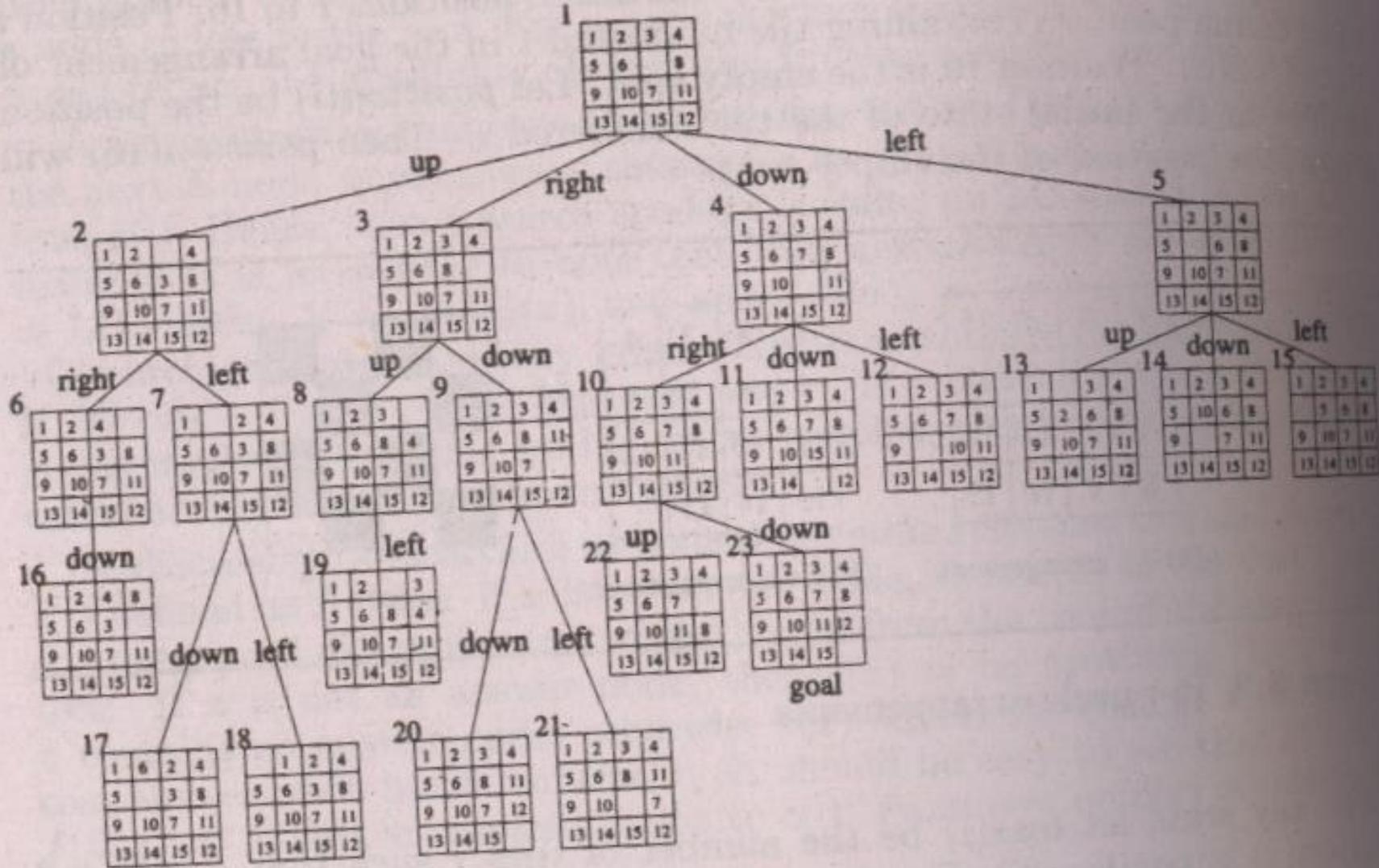
15-Puzzle problem

(State space tree)

- (a) A Random arrangement of 15 Puzzle
- (b) A Goal arrangement of 15 Puzzle
- (c) 4 x 4 locations for 15 Puzzle problem



2 15-puzzle arrangements



Edges are labeled according to the direction
in which the empty space moves

Figure 8.3 Part of the state space tree for the 15-puzzle

Depth First (Not Suitable)

- Each step gets us further from GOAL

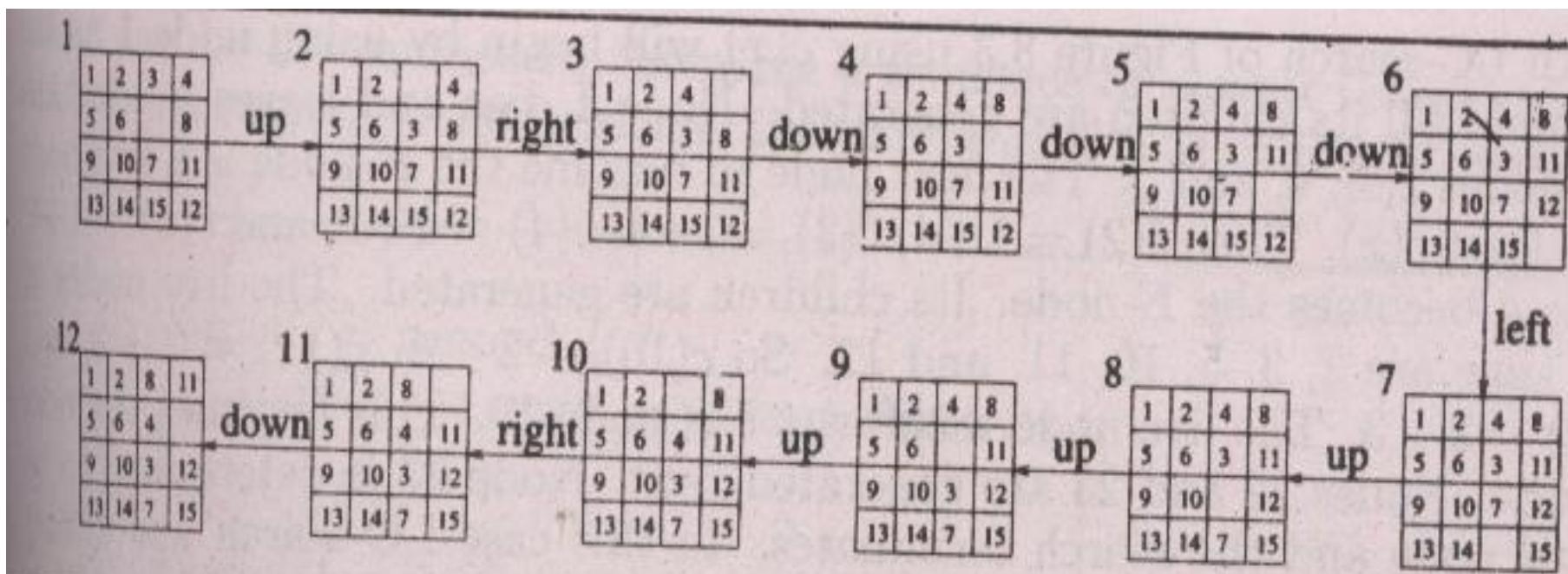
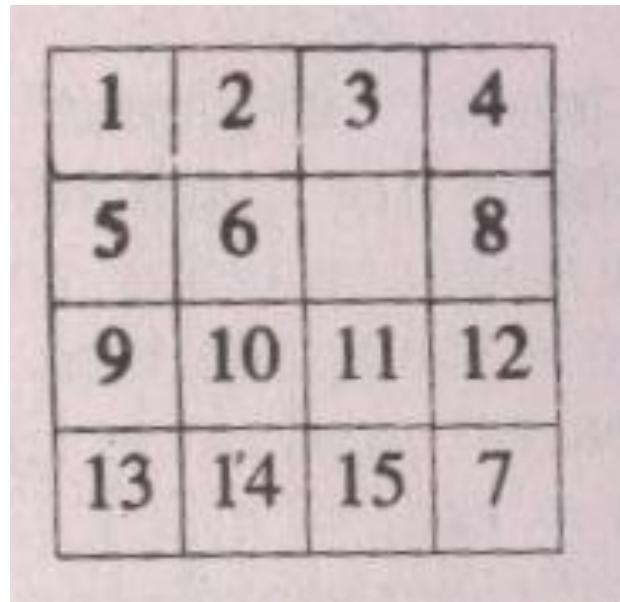


Figure 8.4 First ten steps in a depth first search

15 puzzle sample state

- Only 1 tile out of place but, more than 1 moves will be required for putting that tile in place



0/1 Knapsack

Branch and Bound

Problem: 0/1 Knapsack

$$\text{minimize} \quad - \sum_{i=1}^n p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^n w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

Sample Problem

- $n=4, m=15$; $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$;
 $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$

Bound

```
1 Algorithm Bound( $cp, cw, k$ )
2   //  $cp$  is the current profit total,  $cw$  is the current
3   // weight total;  $k$  is the index of the last removed
4   // item; and  $m$  is the knapsack size.
5   {
6      $b := cp; c := cw;$ 
7     for  $i := k + 1$  to  $n$  do
8       {
9          $c := c + w[i];$ 
10        if ( $c < m$ ) then  $b := b + p[i];$ 
11        else return  $b + \underline{(1 - (c - m)/w[i]) * p[i]};$ 
12      }
13      return  $b;$ 
}
```

```

1 Algorithm Bound( $cp, cw, k$ )
2 //  $cp$  is the current profit total,  $cw$  is the current
3 // weight total;  $k$  is the index of the last removed
4 // item; and  $m$  is the knapsack size.
5 {
6      $b := cp$ ;  $c := cw$ ;
7     for  $i := k + 1$  to  $n$  do
8     {
9          $c := c + w[i]$ ;
10        if ( $c < m$ ) then  $b := b + p[i]$ ;
11        else return  $b + (1 - (c - m)/w[i]) * p[i]$ ;
12    }
13    return  $b$ ;
}

```

Explanation of Line 11: we add to “ b ” the profit obtained by adding fraction of object “ i ”

This can be understood by observing that, in line 9: weight $w[i]$ is added to c (even if it will exceed m).

So in line 11: when $c>m$, we subtract that amount $(c-m)$ and divide it by weight $w[i]$, and then take the fraction $1- ((c-m)/w[i])$ to get the actual fraction to multiply with $p[i]$, to get the correct fractional profit

For example before line 9: :

If $m=15$, $w[i]=9$, and $c=12$,

We actually need to add only 3 of the $w[i]$ (so as not to exceed 15), and so $(3/w[i]) * p[i]$ needs to be added to total profit, which is $(3/9)*p[i]$, but

after line 9, $c=12+9=21$, which is 6 greater than 15.

So, in line 11: we do $(c-m) =6$, then divide it by $w[i]$ to get $6/w[i]$ and then subtract this fraction from 1, to get $3/w[i]$ and then multiply it by $p[i]$, to get $(3/w[i]) * p[i]$

Finding upper bound for 0/1 knapsack

```
1  Algorithm UBound( $cp, cw, k, m$ )
2    //  $cp, cw, k$ , and  $m$  have the same meanings as in
3    // Algorithm 7.11.  $w[i]$  and  $p[i]$  are respectively
4    // the weight and profit of the  $i$ th object.
5  {
6     $b := cp; c := cw;$ 
7    for  $i := k + 1$  to  $n$  do
8    {
9      if ( $c + w[i] \leq m$ ) then
10        {
11           $c := c + w[i]; b := b - p[i];$ 
12        }
13    }
14    return  $b;$ 
15 }
```

Sample Problem

$n=4, m=15$; $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$; $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$

Call initially with – Bound(0,0,0) UBound(0,0,0,15)

```
1 Algorithm Bound( $cp, cw, k$ )
2 //  $cp$  is the current profit total,  $cw$  is the current
3 // weight total;  $k$  is the index of the last removed
4 // item; and  $m$  is the knapsack size.
5 {
6      $b := cp$ ;  $c := cw$ ;
7     for  $i := k + 1$  to  $n$  do
8     {
9          $c := c + w[i]$ ;
10        if ( $c < m$ ) then  $b := b + p[i]$ ;
11        else return  $b + (1 - (c - m)/w[i]) * p[i]$ ;
12    }
13    return  $b$ ;
}
```

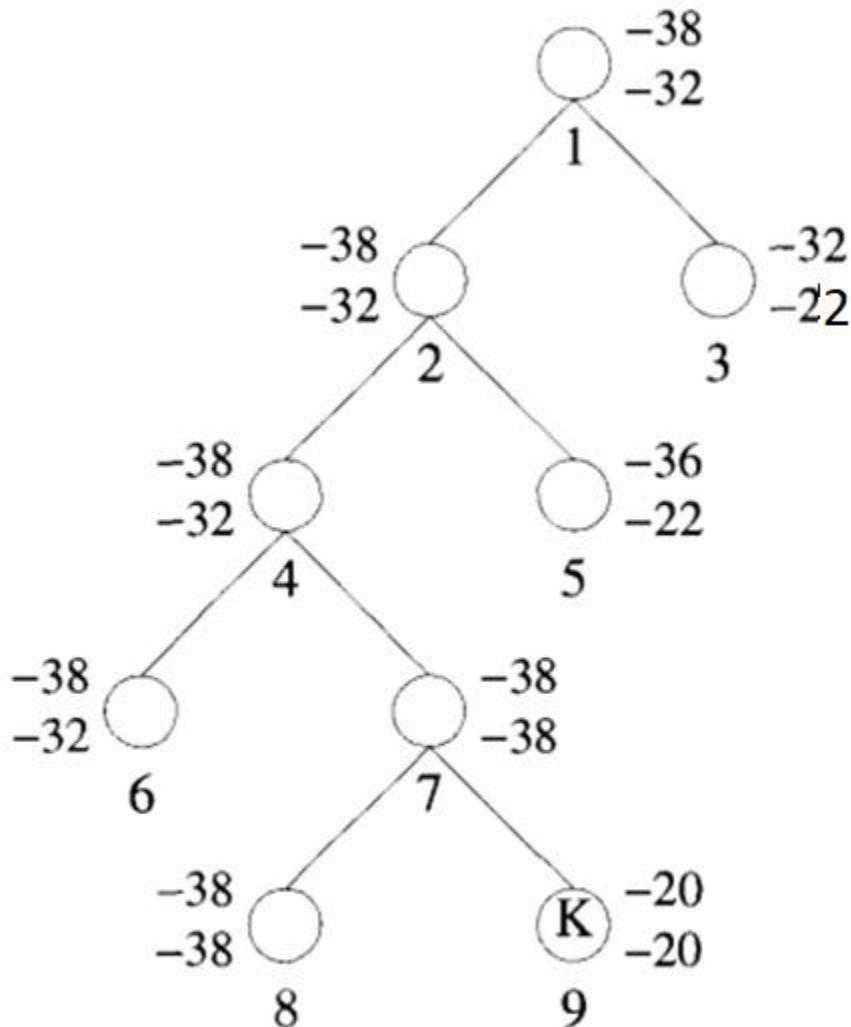
```
1 Algorithm UBound( $cp, cw, k, m$ )
2 //  $cp, cw, k$ , and  $m$  have the same meanings as in
3 // Algorithm 7.11.  $w[i]$  and  $p[i]$  are respectively
4 // the weight and profit of the  $i$ th object.
5 {
6      $b := cp$ ;  $c := cw$ ;
7     for  $i := k + 1$  to  $n$  do
8     {
9         if ( $c + w[i] \leq m$ ) then
10            {
11                 $c := c + w[i]$ ;  $b := b - p[i]$ ;
12            }
13        }
14    return  $b$ ;
15 }
```

Algorithm Function $u(\cdot)$ for knapsack problem

Sample Problem

$n=4, m=15$;

$(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$
; $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$



Upper number = \hat{c}

Lower number = u

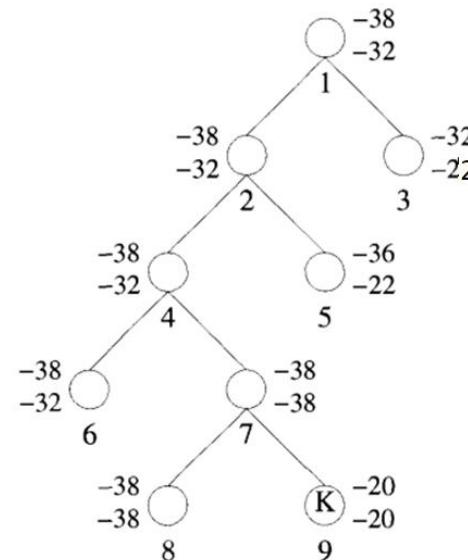
Sample Problem

$n=4, m=15$;

$(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$;

$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$

Call bound initially with –
Bound(0,0,1)



Upper number = \hat{c}

Lower number = u

```
1 Algorithm Bound( $cp, cw, k$ )
2 //  $cp$  is the current profit total,  $cw$  is the current
3 // weight total;  $k$  is the index of the last removed
4 // item; and  $m$  is the knapsack size.
5 {
6      $b := cp; c := cw;$ 
7     for  $i := k + 1$  to  $n$  do
8     {
9          $c := c + w[i];$ 
10        if ( $c < m$ ) then  $b := b + p[i];$ 
11        else return  $b + (1 - (c - m)/w[i]) * p[i];$ 
12    }
13 }
```

```
1 Algorithm UBound( $cp, cw, k, m$ )
2 //  $cp, cw, k$ , and  $m$  have the same meanings as in
3 // Algorithm 7.11.  $w[i]$  and  $p[i]$  are respectively
4 // the weight and profit of the  $i$ th object.
5 {
6      $b := cp; c := cw;$ 
7     for  $i := k + 1$  to  $n$  do
8     {
9         if ( $c + w[i] \leq m$ ) then
10            {
11                 $c := c + w[i]; b := b - p[i];$ 
12            }
13        }
14     return  $b;$ 
15 }
```

Travelling Salesperson Problem (TSP)

Branch and Bound

Goal of TSP

- The goal of the TSP is - to find the shortest possible route that **visits a set of cities exactly once** and **returns to the starting city**.

TSP-BB : Points to be covered

- Problem statement
- LCBB technique for TSP
- Reduced Cost Matrix
- State Space Tree

The least cost branch and bound technique (LCBB) algorithm for solving the traveling salesperson problem (TSP).

TSP with LCBB

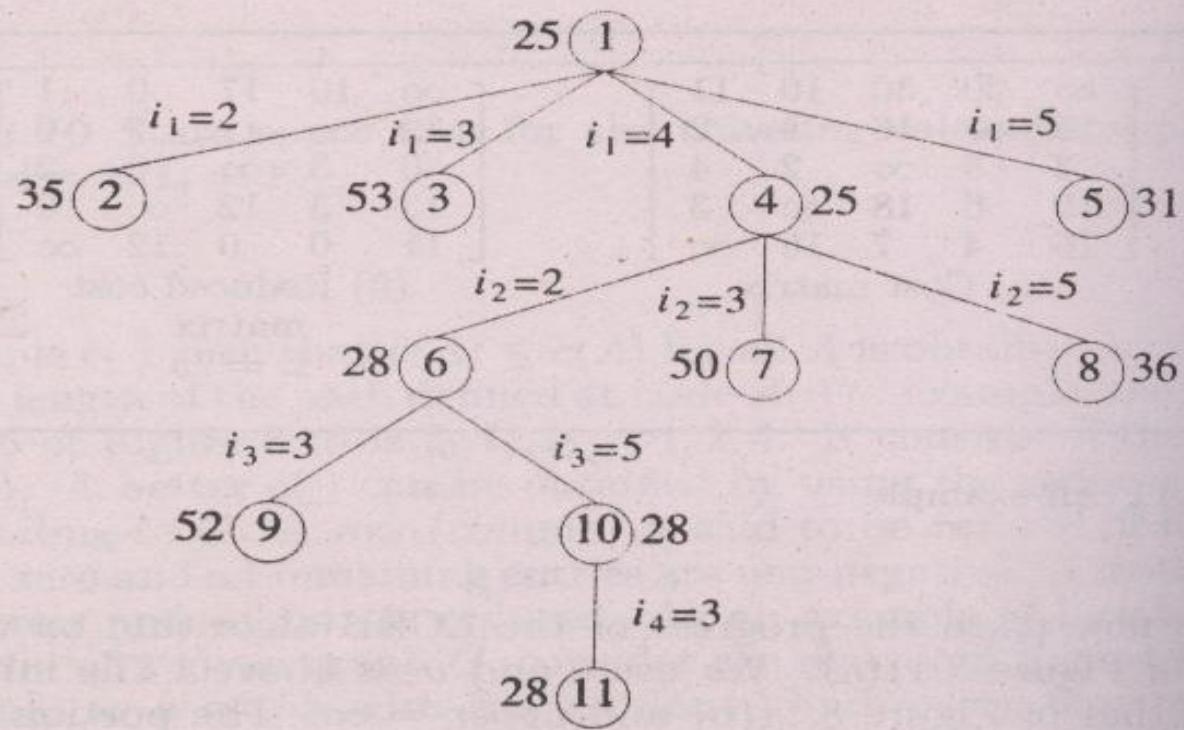
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost
matrix
 $L = 25$

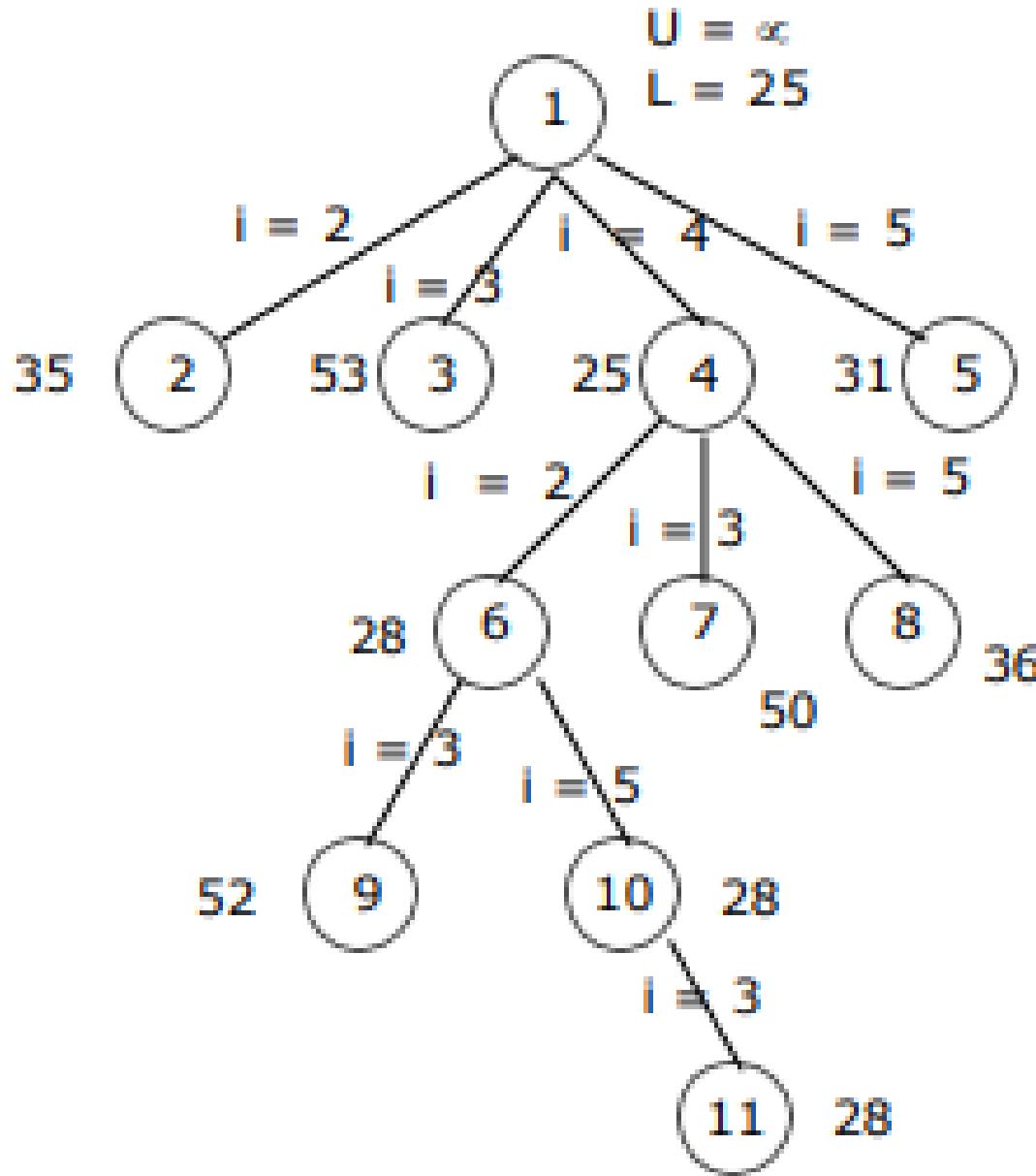
$$\hat{c}(S) = \hat{c}(R) + A(i, j) + r$$



Numbers outside the node are \hat{c} values

2 State space tree generated by procedure LCBB

State Space
Tree for the
given
problem



| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | 11 | 2 | 0 |
| 0 | ∞ | ∞ | 0 | 2 |
| 15 | ∞ | 12 | ∞ | 0 |
| 11 | ∞ | 0 | 12 | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | ∞ | ∞ | 2 | 0 |
| ∞ | 3 | ∞ | 0 | 2 |
| 4 | 3 | ∞ | ∞ | 0 |
| 0 | 0 | ∞ | 12 | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 12 | ∞ | ∞ | 11 | ∞ |
| 0 | 3 | ∞ | ∞ | 2 |
| ∞ | 3 | 12 | ∞ | 0 |
| 11 | 0 | 0 | ∞ | ∞ |

(a) Path 1,2; node 2

(b) Path 1,3; node 3

(c) Path 1,4; node 4

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 10 | ∞ | 9 | 0 | ∞ |
| 0 | 3 | ∞ | 0 | ∞ |
| 12 | 0 | 9 | ∞ | ∞ |
| ∞ | 0 | 0 | 12 | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | 11 | ∞ | 0 |
| 0 | ∞ | ∞ | ∞ | 2 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 11 | ∞ | 0 | ∞ | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | ∞ | ∞ | ∞ | 0 |
| ∞ | 1 | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 0 | ∞ | ∞ | ∞ |

(d) Path 1,5; node 5

(e) Path 1,4,2; node 6

(f) Path 1,4,3; node 7

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 1 | ∞ | 0 | ∞ | ∞ |
| 0 | 3 | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 0 | 0 | ∞ | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | ∞ | ∞ | ∞ | ∞ |

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | ∞ | ∞ | ∞ | ∞ |

(g) Path 1,4,5; node 8

(h) Path 1,4,2,3; node 9

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | 10 | ∞ | 0 | 1 |
| ∞ | ∞ | 11 | 2 | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 3 | 12 | ∞ | 0 |
| ∞ | 0 | 0 | 12 | ∞ |

(a) Node 2

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | 10 | 17 | 0 | 1 |
| 1 | ∞ | 11 | 2 | 0 |
| ∞ | 3 | ∞ | 0 | 2 |
| 4 | 3 | 12 | ∞ | 0 |
| 0 | 0 | 0 | 12 | ∞ |

(b) Node 3

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | 7 | ∞ | 0 | ∞ |
| ∞ | ∞ | ∞ | 2 | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 0 | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |

(c) Node 4

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | 10 | ∞ | 0 | 1 |
| ∞ | ∞ | 0 | 2 | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 3 | 1 | ∞ | 0 |
| ∞ | 0 | ∞ | 12 | ∞ |

(d) Node 5

| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 0 | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | ∞ | ∞ |

(e) Node 6

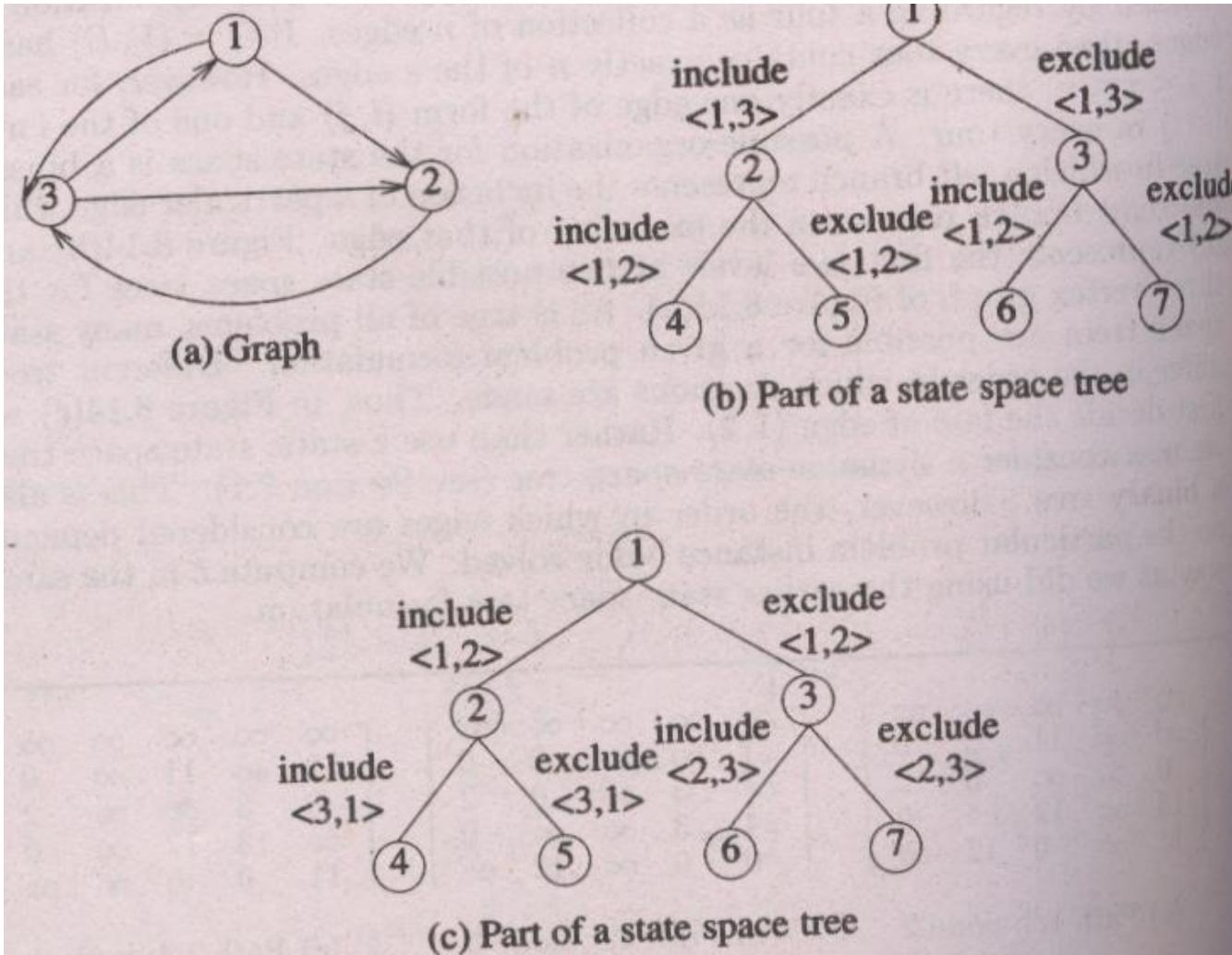
| | | | | |
|----------|----------|----------|----------|----------|
| ∞ | 0 | ∞ | ∞ | ∞ |
| ∞ | ∞ | ∞ | 0 | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |
| ∞ | 0 | ∞ | ∞ | 0 |
| ∞ | ∞ | ∞ | ∞ | ∞ |

(f) Node 7

Procedure for solving traveling sale person problem

```
1  Algorithm LCSearch(t)
2    // Search t for an answer node.
3    {
4      if *t is an answer node then output *t and return;
5      E := t; // E-node.
6      Initialize the list of live nodes to be empty;
7      repeat
8        {
9          for each child x of E do
10            {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15            }
16            if there are no more live nodes then
17            {
18              write ("No answer node"); return;
19            }
20            E := Least();
21        } until (false);
22    }
```

LCBB with Dynamic binary tree formulation

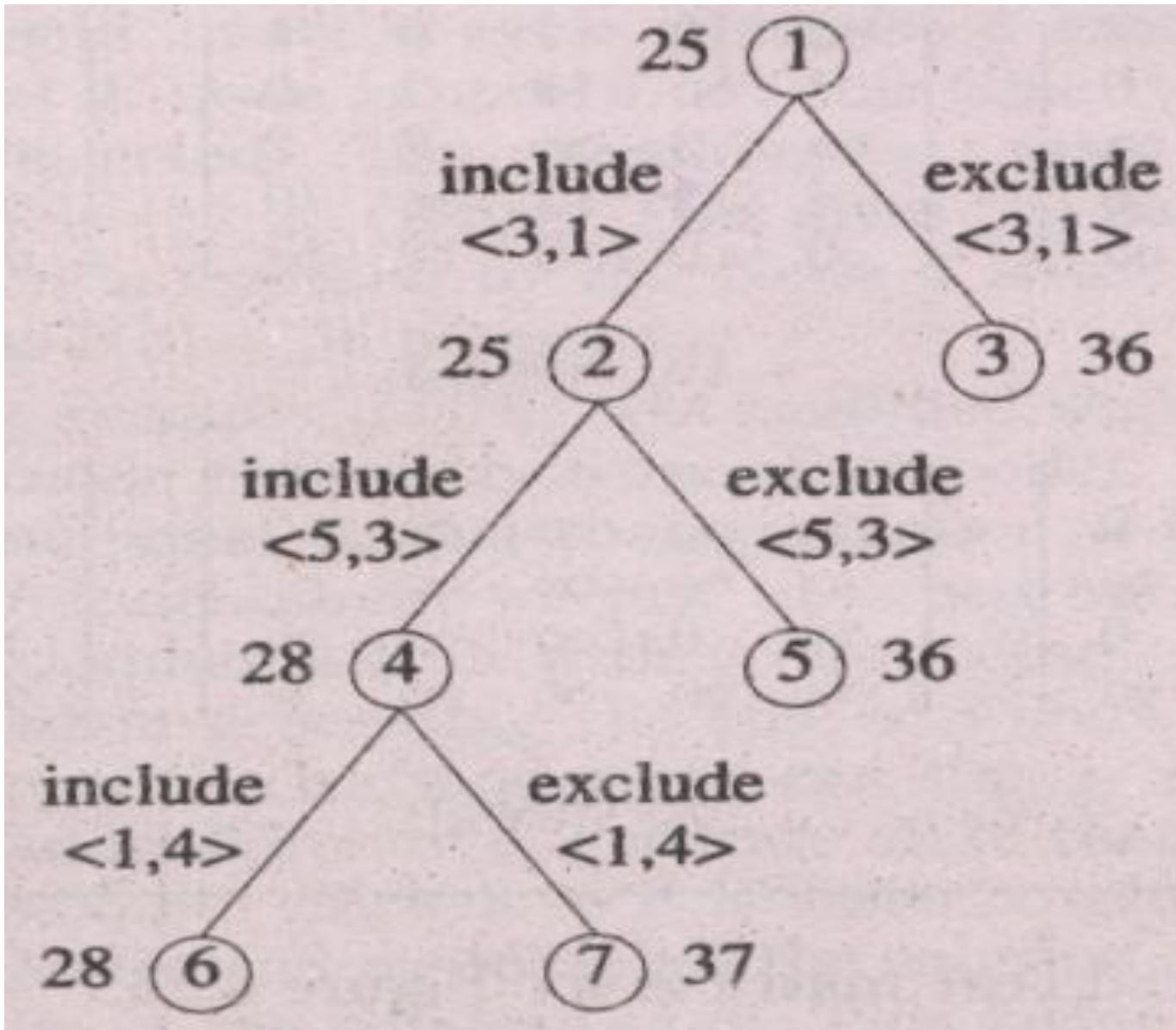


(Refer to Book: DAA by Horowitz, Sahani)

1. Consider the traveling salesperson instance defined by the cost matrix

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

- (a) Obtain the reduced cost matrix
- (b) Using a state space tree formulation similar to that of Figure 8.10 and \hat{c} as described in Section 8.3, obtain the portion of the state space tree that will be generated by LCBB. Label each node by its \hat{c} value. Write out the reduced matrices corresponding to each of these nodes.
- (c) Do part (b) using the reduced matrix method and the dynamic state space tree approach discussed in Section 8.3.



Thank You !!