ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)

Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.

Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.

A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.

Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Artificial Intelligence

Academic Year 2023-24

III. B.Tech I Semster

Design And Analysis of Algorithms
(20APE3001)

Prepared By

Mr. N Venkata Ramana ., M.Tech.
Assistant Professor
Department of CSE, AITS

DESIGN AND ANALYSIS OF ALGORITHM

Unit 1.1 Topics: Algorithm- Algorithm specification- Performance analysis

Algorithm:

- An algorithm is a finite set of instructions that accomplishes a particular task.
- In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper.

- An algorithm is composed of a finite set of steps, each of which may require one or more operations.
- Algorithms produce one or more outputs and have zero or more inputs that are externally supplied.
- Each operation must be definite, meaning that it must be perfectly clear what should be done.
- They terminate after a finite number of operations.
- Algorithms that are definite and effective are also called **computational procedures**.
- A program is the expression of an algorithm in a programming language.

What is an Algorithm?

1. **How to devise algorithms**- Creating an algorithm is an art which may never be fully automated.
2. **How to validate algorithms**- Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as algorithm validation.
3. **How to analyze algorithm**- Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.
4. **How to test a program** – Testing a program consists of two phases: debugging and profiling.

- Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
- Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Algorithm specification:

- We can describe an algorithm in many ways.
- We can use a natural language like English, although if we select this option we must make sure that the resulting instructions are definite.
- Graphic representation called flowcharts are another possibility, but they work well only if the algorithm is small and simple.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.

3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **records**. Here is an example:

$$
\begin{array}{ll}
node = \textbf{record} & \\
\{ & datatype\_1 \quad data\_1; \\
 & \quad\vdots \\
 & datatype\_n \quad data\_n; \\
 & node \qquad\quad *link; \\
\} &
\end{array}
$$

In this example, *link* is a pointer to the record type *node*. Individual data items of a record can be accessed with $\rightarrow$ and period. For instance if p points to a record of type *node*, $p \rightarrow data\_1$ stands for the value of the first field in the record. On the other hand, if q is a record of type *node*, $q.data\_1$ will denote its first field.

4. Assignment of values to variables is done using the assignment statement

$$\langle variable \rangle := \langle expression \rangle;$$

5. There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<, \leq, =, \neq, \geq$, and $>$ are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i,j)th element of the array is denoted as $A[i,j]$. Array indices start at zero.

7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

$$
\begin{array}{l}
\textbf{while } \langle condition \rangle \textbf{ do} \\
\{ \\
\quad \langle statement\ 1 \rangle \\
\quad\quad \vdots \\
\quad \langle statement\ n \rangle \\
\}
\end{array}
$$

As long as $\langle condition \rangle$ is **true**, the statements get executed. When $\langle condition \rangle$ becomes **false**, the loop is exited. The value of $\langle condition \rangle$ is evaluated at the top of the loop.

The general form of a **for** loop is

$$
\begin{array}{l}
\textbf{for } variable := value1 \textbf{ to } value2 \;\; \textbf{step } step \textbf{ do} \\
\{ \\
\quad \langle statement\ 1 \rangle \\
\quad\quad \vdots \\
\quad \langle statement\ n \rangle \\
\}
\end{array}
$$

Here *value*1, *value*2, and *step* are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** *step*" is optional and taken as +1 if it does not occur. *step* could either be positive or negative. *variable* is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

$variable := value1;$
$fin := value2;$
$incr := step;$
while $((variable - fin) * step \leq 0)$ **do**
{
  ⟨*statement* 1⟩
  ⋮
  ⟨*statement* *n*⟩
  $variable := variable + incr;$
}

A **repeat-until** statement is constructed as follows:

repeat
  ⟨*statement* 1⟩
  ⋮
  ⟨*statement* *n*⟩
until ⟨*condition*⟩

The statements are executed as long as ⟨*condition*⟩ is **false**. The value of ⟨*condition*⟩ is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement* 1⟩ **else** ⟨*statement* 2⟩

Here ⟨*condition*⟩ is a boolean expression and ⟨*statement*⟩, ⟨*statement* 1⟩, and ⟨*statement* 2⟩ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

case
{
  :⟨*condition* 1⟩**:** ⟨*statement* 1⟩
  ⋮
  :⟨*condition* *n*⟩**:** ⟨*statement* *n*⟩
  :else: ⟨*statement* $n+1$⟩
}

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

Algorithm *Name* (⟨*parameter list*⟩)

As an example, the following algorithm finds and returns the maximum of n given numbers:

```
Algorithm Max(A, n)
// A is an array of size n.
{
    Result := A[1];
    for i := 2 to n do
        if A[i] > Result then Result := A[i];
    return Result;
}
```

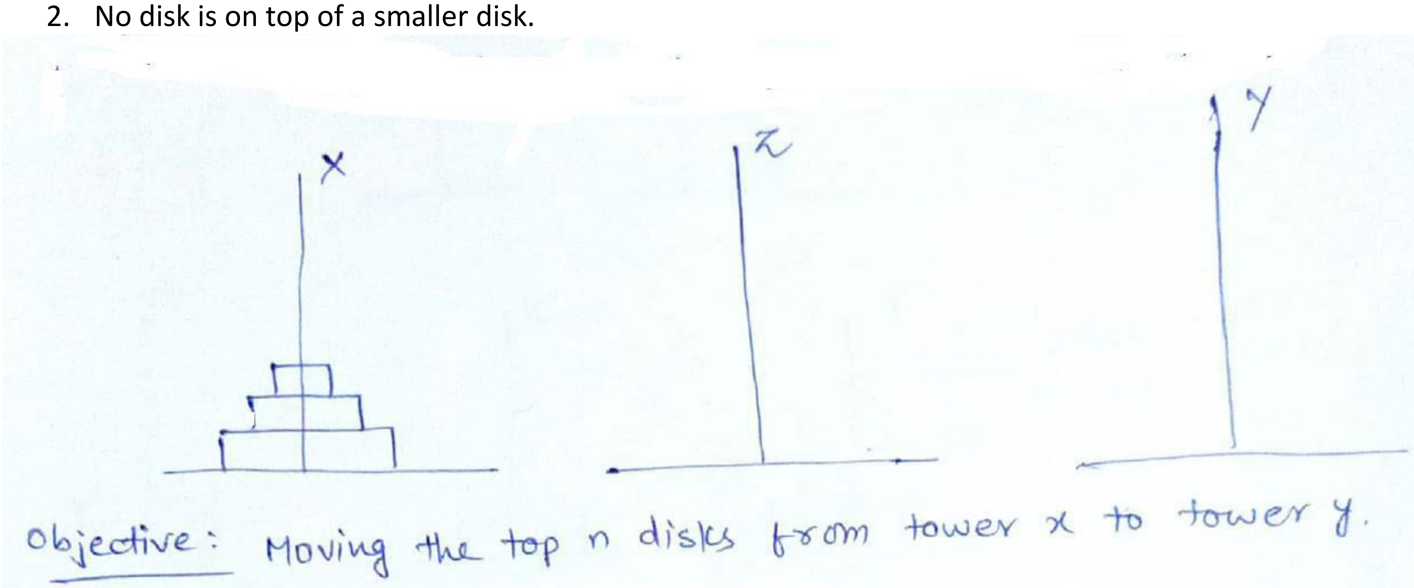
In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Recursive Algorithm:

- A **recursive function** is a function that is defined in terms of it-self.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive.
- Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.
- Example: Towers of Hanoi.

Towers of Hanoi:

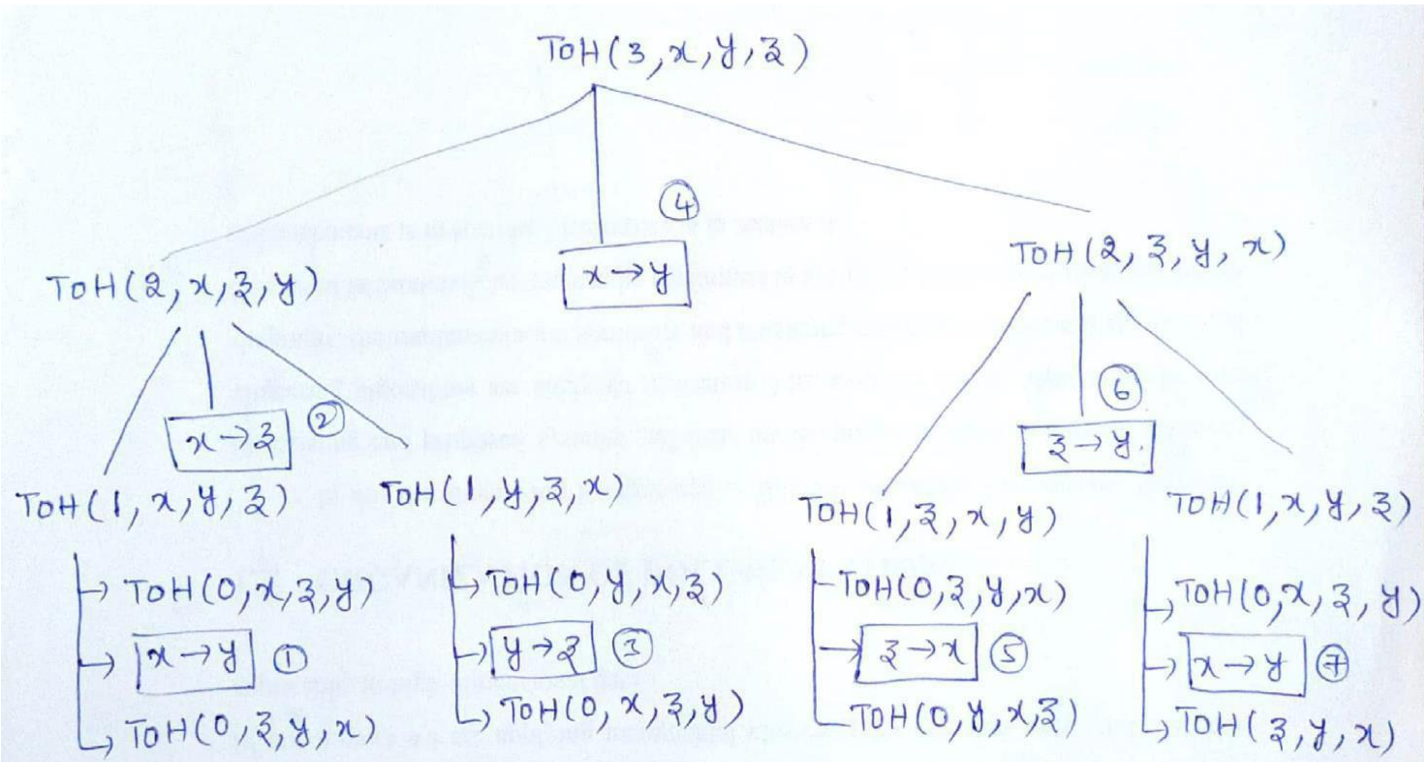
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- **Objective:** Move the disks from tower X to tower Y using tower Z for intermediate storage.
- **Rules:**
  1. As the disks are very heavy, they can be moved only one at a time.
  2. No disk is on top of a smaller disk.



Towers of Hanoi- Recursive Algorithm

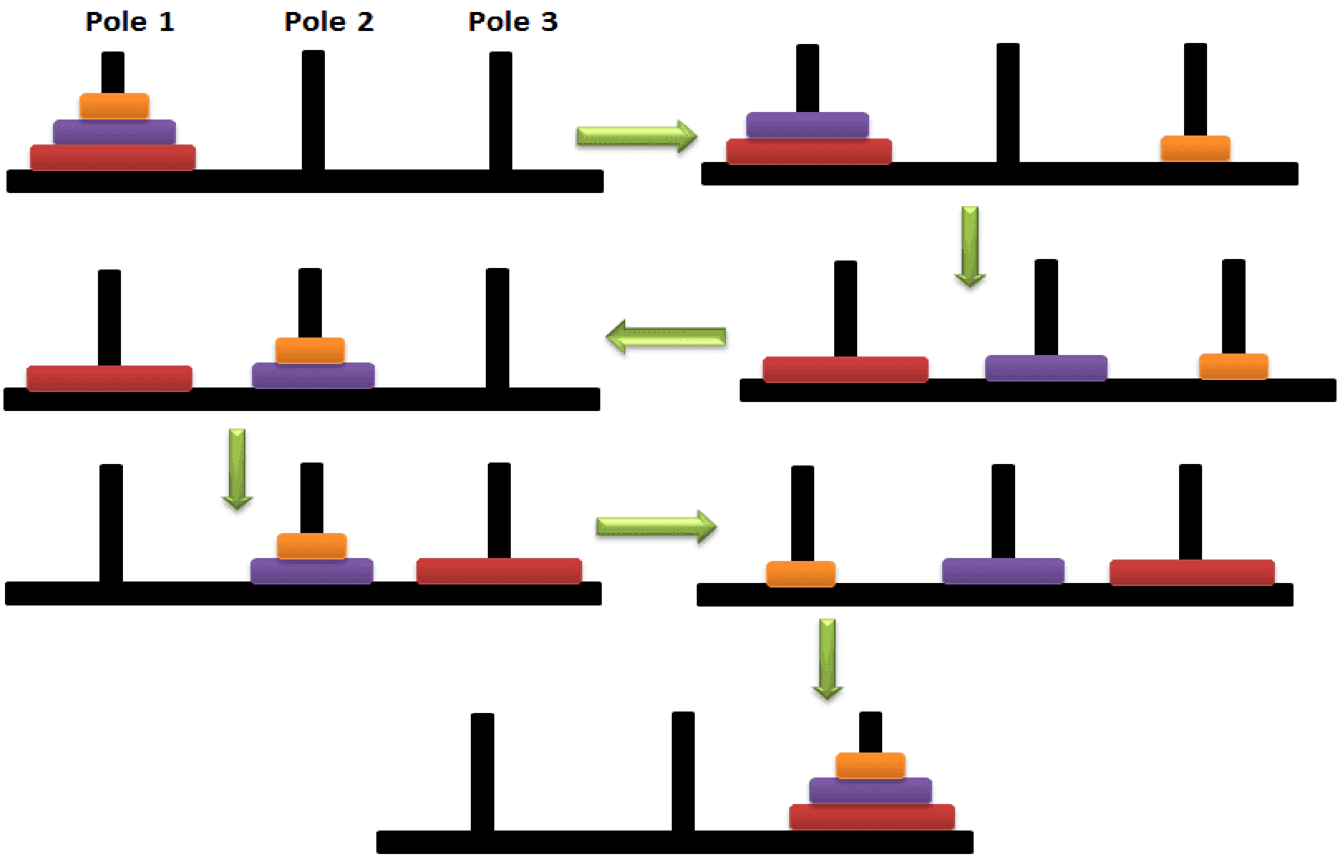
```
Algorithm TowersOfHanoi(n, x, y, z)
// Move the top n disks from tower x to tower y.
{
    if (n ≥ 1) then
    {
        TowersOfHanoi(n − 1, x, z, y);
        write ("move top disk from tower", x,
        "to top of tower", y);
        TowersOfHanoi(n − 1, z, y, x);
    }
}
```

Towers of Hanoi- Algorithm Analysis & Recursive Calls



Towers of Hanoi- Solution For 3 Disks:

- For n=3 disks we have totally seven moves:
  1. X-> Y
  2. X-> Z
  3. Y-> Z
  4. X-> Y
  5. Z-> X
  6. Z->Y
  7. X->Y
- **For n disks, Total Number of Moves= 2^n-1**
- For 3 disks, Total Number of Moves= 2^3-1= 7 moves
- For 4 disks, Total Number of Moves= 2^4-1= 15 moves



<u>Figure:</u> Pole1 is tower X   Pole2 is tower Y & Pole3 is tower Z

<u>Performance Analysis</u>:

- The **space complexity** of an algorithm is the amount of memory it needs to run to completion.
- The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

<u>Space Complexity:</u>

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$, where c is a constant.

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs.
2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables

Example1: Finding result of given expression with fixed values a, b, c

```
Algorithm abc(a, b, c)
{
    return a + b + b * c + (a + b - c)/(a + b) + 4.0;
}
```

Sp= 0 since no instance characteristics & S(P)= c only.

Time Complexity:

- The time T(P)taken by a program P is the sum of the compile time and the run (or execution)time.
- The compile time does not depend on the instance characteristics.
- **Time Complexity can be calculated with two methods:**
  1. Recurrence Relations.
  2. Step count Method
- We may assume that a compiled program will be run several times without recompilation.
  So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

Example- Recurrence Relations

```
Algorithm RSum(a, n)
{
    count := count + 1; // For the if conditional
    if (n ≤ 0) then
    {
        count := count + 1; // For the return
        return 0.0;
    }
    else
    {
        count := count + 1;  // For the addition, function
                             // invocation and return
        return RSum(a, n − 1) + a[n];
    }
}
```

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as *recurrence relations*.

Example- Step Count Method:

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	—	0
2 {	0	—	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	—	0
Total			$2n + 3$

The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	—	0
2 {	0	—	0
3 **for** $i := 1$ **to** m **do**	1	$m + 1$	$m + 1$
4 **for** $j := 1$ **to** n **do**	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	—	0
Total			$2mn + 2m + 1$

Asymptotic Notations:

- Asymptotic analysis of algorithms is used to compare relative performance.
- Time complexity of an algorithm concerns determining an expression of the number of primitive operations needed as a function of the problem size.
- Asymptotic analysis makes use of the following:
  1. Big Oh Notation.
  2. Big Omega Notation.
  3. Big Theta Notation.
  4. Little Oh Notation.
  5. Little Omega Notation.

Big-Oh Notation:

- The big-Oh notation gives an **upper bound** on the growth rate of a function.
- **Definition:**

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c > 0$ and $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \quad \text{for all } n, n \geq n_0$$

This definition is referred to as the "big-Oh" notation. Alternatively, we can also say "$f(n)$ is *order* of $g(n)$". This definition is illustrated in Figure 1.2 (the value of $f(n)$ always lies on or below $cg(n)$).

The big-Oh notation gives an upper bound on the growth rate of a function. The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

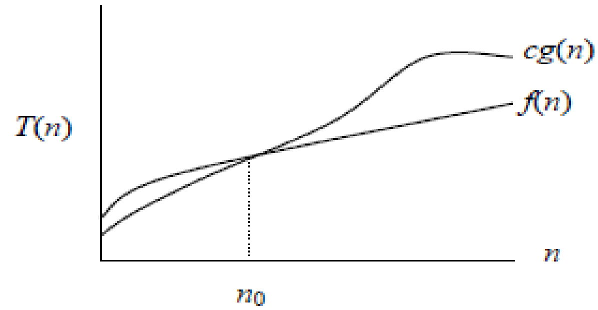


Figure 1.2: $f(n)$ is $O(g(n))$, for $f(n) \le cg(n)$ when $n \ge n_0$

Example:
Let us consider f(n)= 3n+2 g(n)= n c= 4 and the big Oh notation relation is f(n)<=c* g(n)

3n+2 <= 4n for n>=3→ It satisfies the relation

n=3 → 3.3+2<= 4.3 → 11<=12 → True

n=4 → 3.4+2<= 4.4 → 12<=16 → True

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called *linear*, $O(n^2)$ is called *quadratic*, $O(n^3)$ is called *cubic*, and $O(2^n)$ is called *exponential*. If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times–$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$

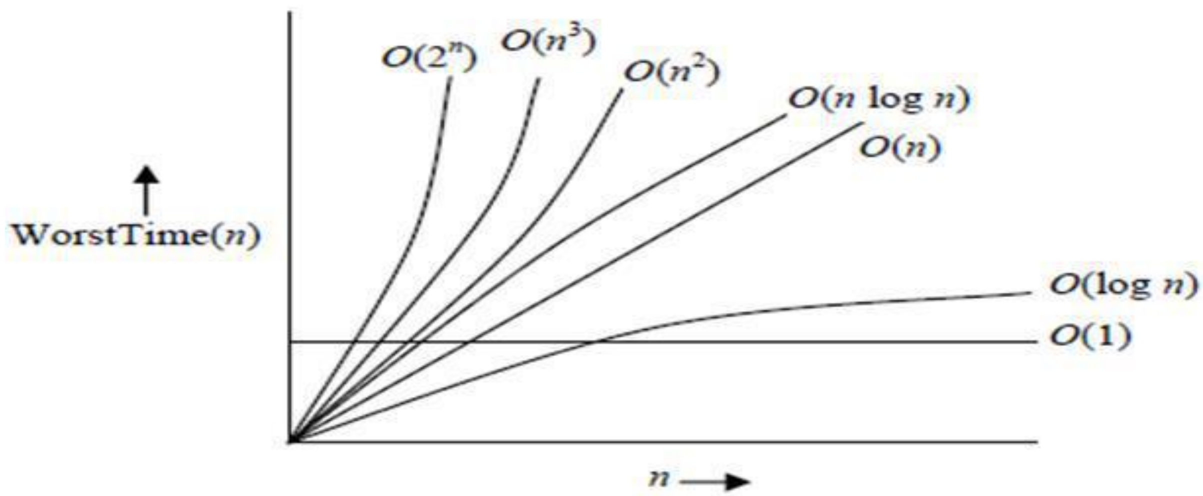


Figure 1.3: Comparison of WorstTime(n) for seven orders of functions

<u>**Big-Omega Notation**</u>**:**

- The big-Omega notation gives an <u>**lower bound**</u> on the growth rate of a function.
- <u>**Definition:**</u>

As O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an *asymptotic lower bound*. Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants $c > 0$ and $n_0 \ge 1$ such that

$$f(n) \ge cg(n) \quad \text{for all } n, n \ge n_0$$

This definition is referred to as the "big-Omega" notation and is illustrated in Figure 1.4: for all values n to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

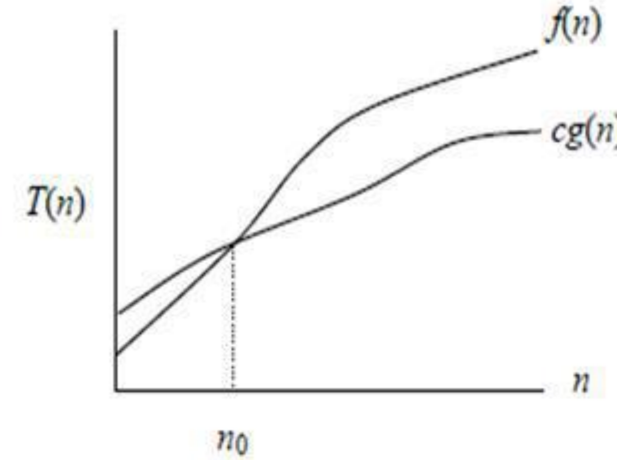


Figure 1.4: $f(n)$ is $\Omega(g(n))$, for $f(n) \geq cg(n)$ when $n \geq n_0$

Example:

Let us consider f(n)= 3n+2 g(n)= n c= 3 and the big Omega notation relation is f(n)>=c* g(n)

3n+2 >= 3n for n>=2→ It satisfies the relation

n=2 → 3.2+2>= 3.2 → 8>= 6→ True

n=3 → 3.3+2<= 4.2 → 9>=8 → True

Big-Theta Notation:

➢ **Definition:**

Let $f(n)$ and $g(n)$ be two asymptotically positive real-valued functions. We say that $f(n)$ is $\Theta(g(n))$ if there is an integer n_0 and positive real constants c_1 and c_2 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ (to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$

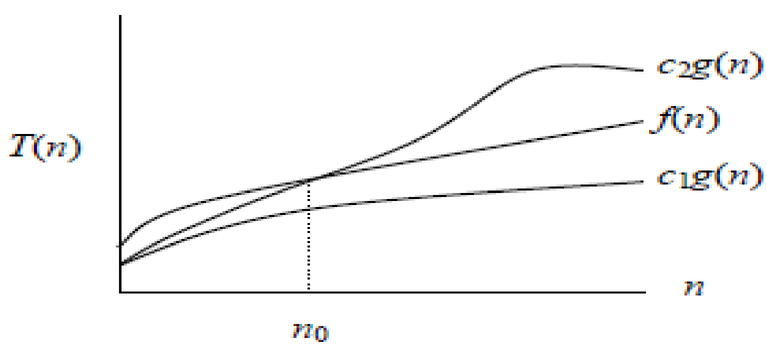


Figure 1.5: $f(n)$ is $\Theta(g(n))$, for $c_1g(n) \leq f(n) \leq c_2g(n)$ when $n \geq n_0$

Example The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

Little Oh & Little Omega Notation

Definition [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Definition [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

****

Unit 1.2 Topics: General method- Binary Search- Finding the maximum and minimum- Merge sort- Quick Sort- Selection- Strassen's matrix multiplication

Divide-And-Conquer- General Method:

- **Divide-and-Conquer** Strategy **breaks (divides)** the given problem into sub problems, solve each sub problems independently and finally **conquer (combine)** all sub problems solutions into whole solution.
- Divide-and-Conquer Strategy suggests splitting the n inputs into k distinct subsets, 1< k < n, yielding k sub-problems.
- If the sub-problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub-problems resulting from a divide-and conquer design are of the same type as the original problem.

DAndC (Algorithm 3.1) is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems $P_1, P_2, \ldots, P_k$ are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are $n_1, n_2, \ldots, n_k$, respectively

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

Algorithm 3.1 Control abstraction for divide-and-conquer

Computing Time:

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases} \qquad (3.1)$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \qquad (3.2)$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Beginning with the recurrence (3.2) and using the substitution method, it can be shown that

$$T(n) = n^{\log_b a}[T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^{k} h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$.

Binary Search:

- A binary search algorithm is a technique for finding a particular value in a sorted list.
- Divide-and-conquer can be used to solve this problem.
- Any given problem P gets divided into one new sub-problem. This division takes only **O(1) time**.
- After a comparison the instance remaining to be solved by using this divide-and-conquer scheme again.
- If the element is found in the list → successful search
- If the element is not found in the list → unsuccessful search
- The Time Complexity of Binary Search is:

successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Recursive Binary Search- Algorithm

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                 return BinSrch(a, i, mid − 1, x);
             else return BinSrch(a, mid + 1, l, x);
    }
}
```

Iterative Binary Search- Algorithm

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
             else return mid;
    }
    return 0;
}
```

BINARY SEARCH- EXAMPLE

Eg:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
10	15	40	50	55	65	75	90	95
low				mid				high

key = 65.

mid = (low + high)/2

= (0+8)/2 = 8/2 = 4

key > a[mid]   low = 5, high = 8

a[5]	a[6]	a[7]	a[8]
65	75	90	95
low	mid		high

mid = (5+8)/2 = 13/2 = 6

key < a(mid)   low = 5, high = 5

a[5]
65
low mid high

key = a[mid] then mid = 5 i.e. key = 65 is found in a[5] th element.

Binary Search Time Complexity

$$T(n) = T(n/2) + 1$$
$$= \{T(n/4) + 1\} + 1$$
$$= T(n/4) + 2$$
$$= [T(n/8) + 1] + 2$$
$$= T(n/8) + 3$$
$$\vdots$$

$n = 2^k$

$k = \log n$

$$T(n) = T(n/2^k) + 1 + 1 + \cdots + k \text{ times}$$
$$= T(2^k/2^k) + \cdots k \text{ times}$$
$$= T(1) + k$$
$$= T(1) + \log n = 1 + \log n$$

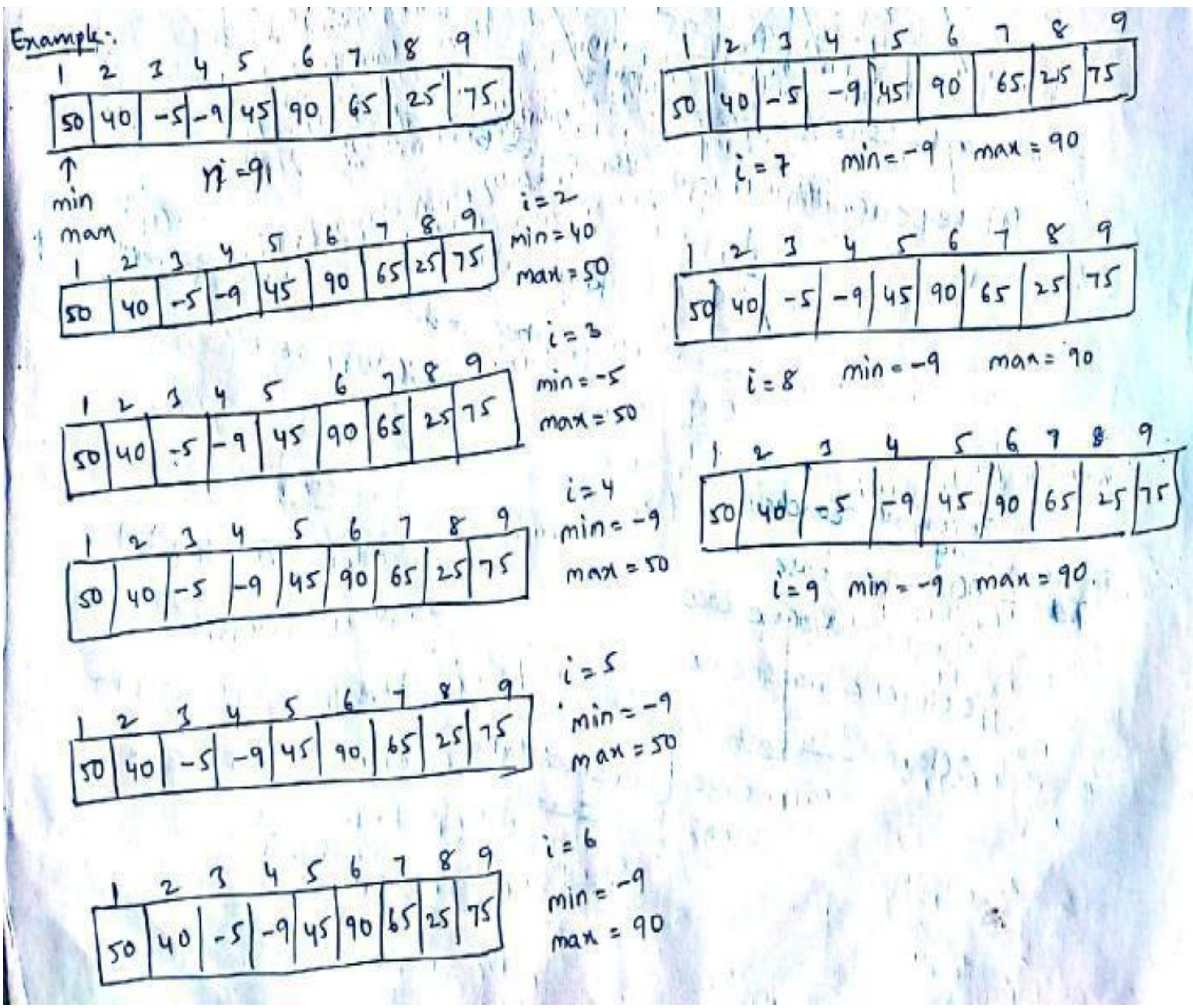
Time Complexity is $O(\log n)$

Finding the Maximum And Minimum:

- The divide and- conquer technique is to find the maximum and minimum items in a set of n elements in the given list.

```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then max := a[i];
        if (a[i] < min) then min := a[i];
    }
}
```

Algorithm 3.5 Straightforward maximum and minimum



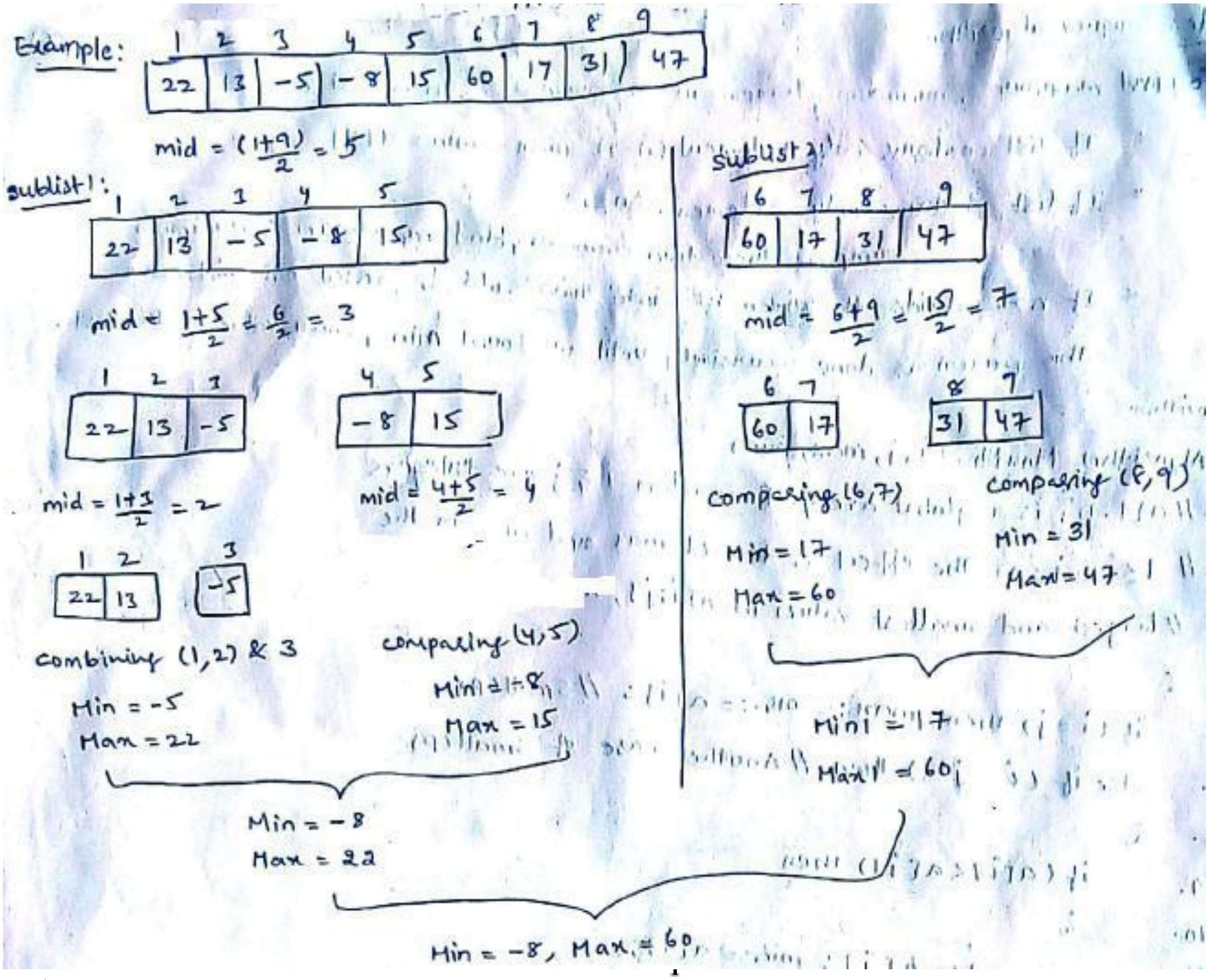
Finding the Maximum And Minimum- Divide & Conquer:

- If the list contains only **one element**, then

  Maximum=Minimum= a[1] element only.
- If the list contains **two elements**, then **compare these two elements** & find maximum and minimum.
- If the list contains **more than two elements**, then **divide** the given list into sub lists based on **middle value.**
- Recursively perform this process until minimum & maximum value is found in the given list.
- **Time Complexity = O(n)**

Algorithm:

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i : j], respectively.
{
    if (i = j) then max := min := a[i]; // Small(P)
    else if (i = j − 1) then  // Another case of Small(P)
        {
            if (a[i] < a[j]) then
            {
                max := a[j]; min := a[i];
            }
            else
            {
                max := a[i]; min := a[j];
            }
        }
        else
        {   // If P is not small, divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(i + j)/2⌋;
            // Solve the subproblems.
                MaxMin(i, mid, max, min);
                MaxMin(mid + 1, j, max1, min1);
            // Combine the solutions.
                if (max < max1) then max := max1;
                if (min > min1) then min := min1;
        }
}
```



$T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned}$$

Merge sort:

- In Merge Sort, the elements are to be sorted in non decreasing order.
- Given a sequence of n elements(also called keys) a[l],,.a[n]the general idea is to imagine them split into two sets a[l]….a[n/2]and a[[n/2]+1]….a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Merge sort Algorithm:

MergeSort (Algorithm 3.7) describes this process very succinctly using recursion and a function Merge (Algorithm 3.8) which merges two sorted sets. Before executing MergeSort, the n elements should be placed in $a[1:n]$. Then MergeSort(1,n) causes the keys to be rearranged into nondecreasing order in a.

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Algorithm 3.7 Merge sort

Merge- Algorithm:

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

Merge sort- Example

Example 3.7 Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\,]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$).

$$(310 \mid 285 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

$$(285, 310 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

Then $a[3]$ is merged with $a[1:2]$ and

$$(179, 285, 310 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

$$(179,\ 285,\ 310 \mid 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

and then $a[1:3]$ and $a[4:5]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423 \mid 861 \mid 254 \mid 450,\ 520)$$

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 861 \mid 450,\ 520)$$

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 450,\ 520,\ 861)$$

At this point there are two sorted subarrays and the final merge produces the fully sorted result

$$(179,\ 254,\ 285,\ 310,\ 351,\ 423,\ 450,\ 520,\ 652,\ 861)$$

Merge sort- Time Complexity

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

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

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\ \vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

$n = 2^k$, $T(1) = a$ & $k = \log n$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

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

Quick Sort:

- In quick sort, the division into two sub arrays is made so that the sorted sub arrays do not need to be merged later.
- Three Steps involved here is:
  1. Partitioning given array into sub arrays.
  2. Interchanging two elements in the array.
  3. Searching the input element.

Time Complexity = O(n*log n)

Quick Sort- Partitioning & Interchanging Algorithm:

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

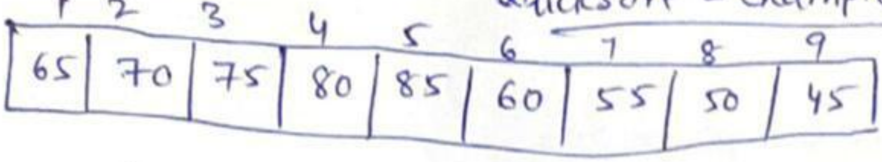
Quick Sort Algorithm:

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j - 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Algorithm 3.13 Sorting by partitioning

Quick sort- Example:

Quicksort - Example



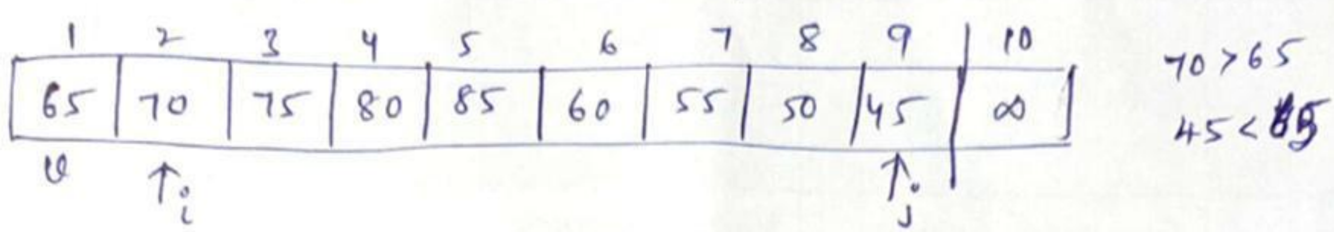
$v = a[1] = 65 \rightarrow$ fixed

Increment i value, until $a[i] \geq v$ set $a[10] = \infty$

Decrement j value, until $a[j] \leq v$

if $i < j \rightarrow$ swap $a[i]$ & $a[j]$

$i \geq j \rightarrow$ do partition. & swap $a[j]$ & v



$2 < 9 \rightarrow$ swap $a[2]$ & $a[9]$

1	2	3	4	5	6	7	8	9	10
65	45	75	80	85	60	55	50	70	∞
v		↑ i					↑ j		

75 > 65
50 < 65

$3 < 8 \rightarrow$ swap a[3] & a[8]

1	2	3	4	5	6	7	8	9	10
65	45	50	80	85	60	55	75	70	∞
v			↑ i			↑ j			

80 > 65
55 < 65

$4 < 7 \rightarrow$ swap a[4] & a[7]

1	2	3	4	5	6	7	8	9	10
65	45	50	55	85	60	80	75	70	∞
v				↑ i	↑ j				

85 > 65
60 < 65

$5 < 6 \rightarrow$ swap a[5] & a[6]

1	2	3	4	5	6	7	8	9	10
65	45	50	55	60	85	80	75	70	∞
v				↑ j	↑ i				

85 > 65
60 < 65

$6 \geq 5 \rightarrow$ do partition & swap a[j] & v

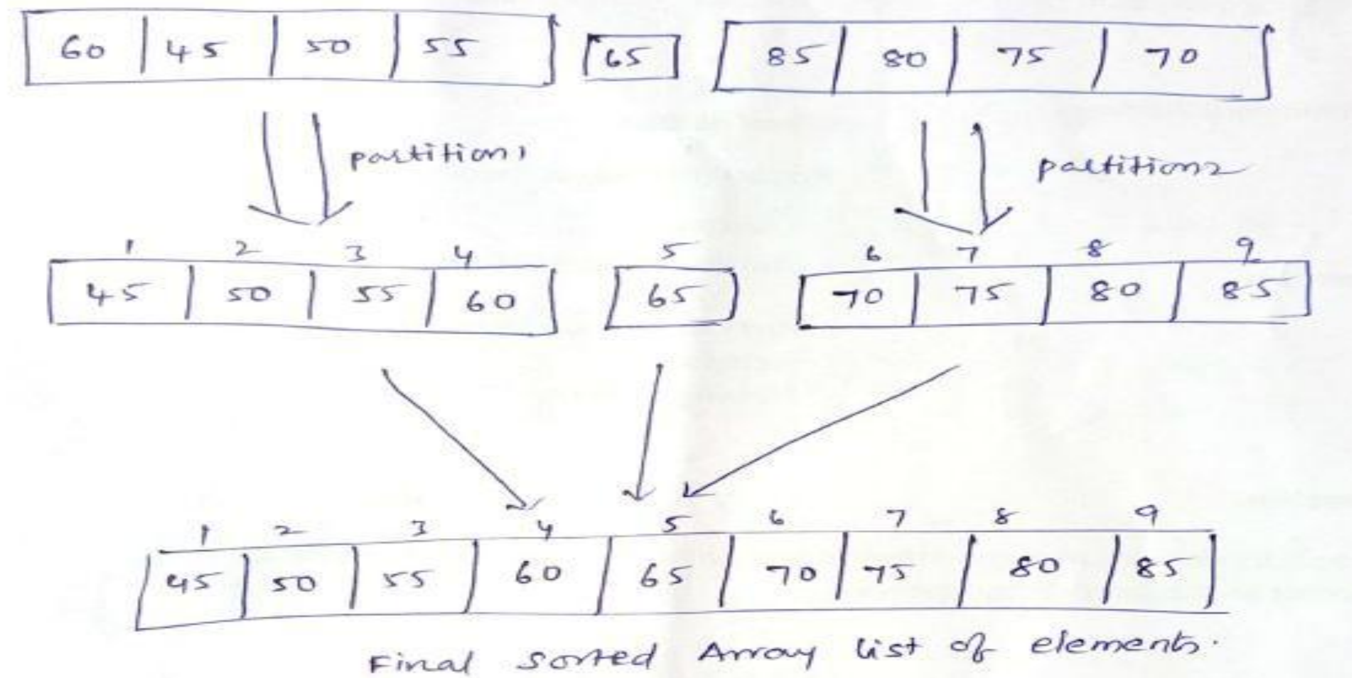
1	2	3	4	5	6	7	8	9	10
60	45	50	55	(65)	85	80	75	70	∞
partition 1				↑ fixed	partition 2				

Quick sort (1, 4) | Quicksort (6, 9)

Apply Quicksort partitioning algorithms for

Quicksort (1,4)

Quicksort (6,9)



Quick sort- Time Complexity:

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

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

n= 2^k , T(1)= a & k = log n

It is easy to see that if $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore

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

Selection:

- Selection is used to **find kth smallest element and place kth position** in the given array of n elements say a[1:n].
- j is the position in which k element is there.
- Here we are partitioning the given array into three parts (say k is small element):
    1. Sub array which contains 1 to j-1 elements.
    2. if k=j => element is found in jth position.
    3. Sub array which contains (n-j) elements.
- This partitioning process continues until we found kth smallest element.
- **Time Complexity= O(n^2)**

Selection- Partitioning & Interchanging Algorithm:

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

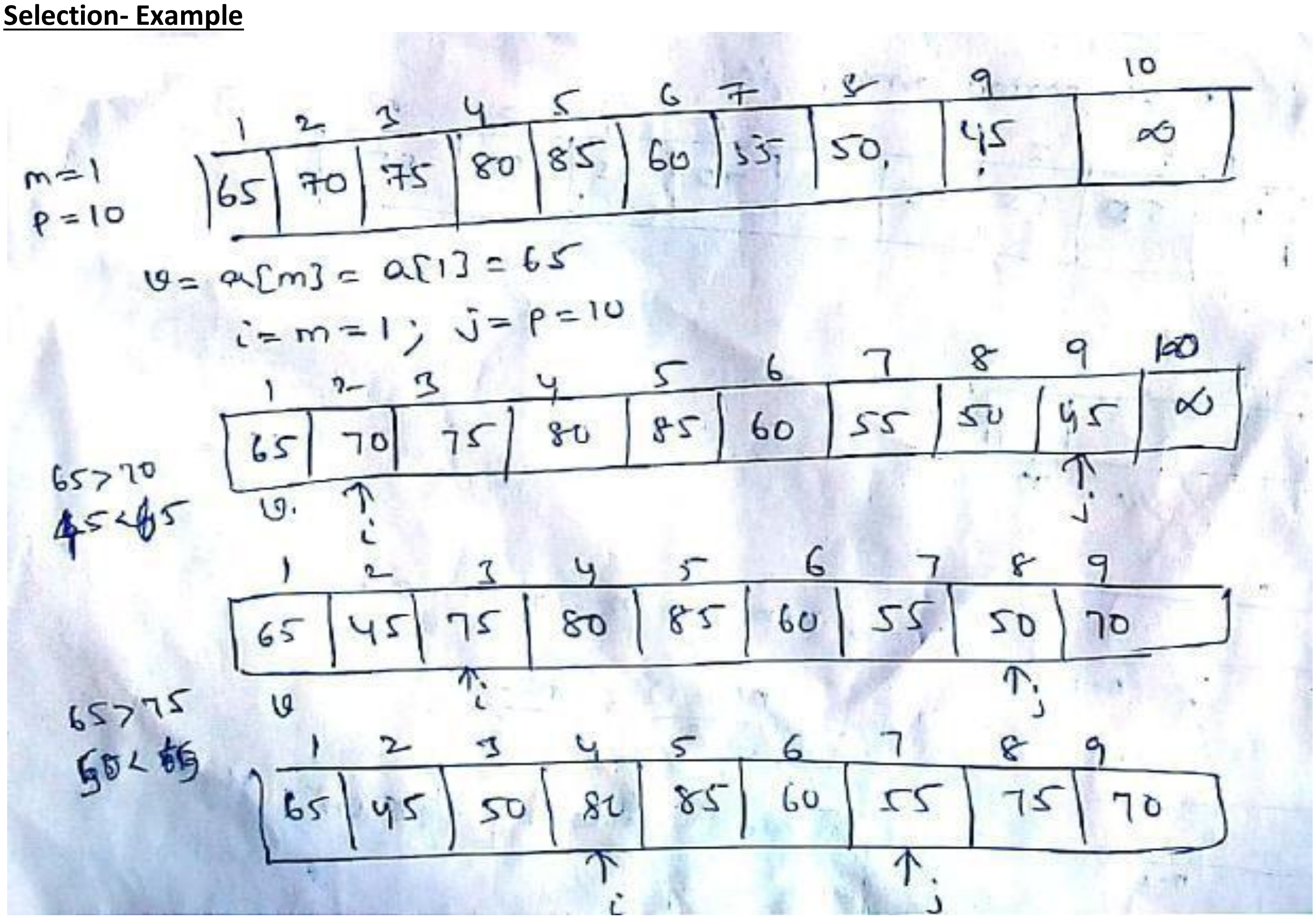
Algorithm 3.12 Partition the array $a[m : p - 1]$ about $a[m]$

Selection- Algorithm

```
Algorithm Select1(a, n, k)
// Selects the kth-smallest element in a[1 : n] and places it
// in the kth position of a[ ]. The remaining elements are
// rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
// a[m] ≥ a[k] for k < m ≤ n.
{
    low := 1; up := n + 1;
    a[n + 1] := ∞; // a[n + 1] is set to infinity.
    repeat
    {
        // Each time the loop is entered,
        // 1 ≤ low ≤ k ≤ up ≤ n + 1.
        j := Partition(a, low, up);
            // j is such that a[j] is the jth-smallest value in a[ ].
        if (k = j) then return;
        else if (k < j) then up := j; // j is the new upper limit.
            else low := j + 1; // j + 1 is the new lower limit.
    } until (false);
}
```

Algorithm 3.17 Finding the kth-smallest element

Selection- Example



65>85
60<65

1	2	3	4	5	6	7	8	9
65	40	50	55	85	60	80	75	70

lb; i ↑ at 5, j ↑ at 6

1	2	3	4	5	6	7	8	9
65	40	50	55	60	85	80	75	70

lb; j ↑ at 5, i ↑ at 6

swap 65 & 60

1	2	3	4	5	6	7	8	9
60	40	50	55	(65)	85	80	75	70

Partition1 (1–4) — fixed (5) — partition2 (6–9)

k = 5 → placing 5th smallest element in 5th position i.e 65

k = 7 → call Partition (6,9)

1	2	3	4
60	40	50	55

place 1st small in 1st position

1	2	3	4
40	60	50	55

place 2nd small in 2nd position

1	2	3	4
40	50	60	55

Place 3rd small in 3rd position

1	2	3	4
40	50	55	60

5	6	7	8	9
65	85	80	75	70

Place 6th smallest element in 6th position

5	6	7	8	9
65	70	80	75	85

swap 85 & 70 because a[6] = 8

place 7th small in 7th position

5	6	7	8	9
65	70	75	80	85

lly place 8th in 8th position

place 9th in 9th position

Final sorted array is

1	2	3	4	5	6	7	8	9
40	50	55	60	65	70	75	80	85

Strassen's matrix multiplication:

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

for all i and j between 1 and n. To compute $C(i,j)$ using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (3.11)$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \quad (3.12)$$

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$.

Volker Strassen has discovered a way to compute the C_{ij}'s of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U, and V as in (3.13). Then the C_{ij}'s are computed using the formulas in (3.14). As can be seen, P, Q, R, S, T, U, and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

Strassen's matrix multiplication- Formula:

$$\begin{aligned} P &= (A_{11}+A_{22})(B_{11}+B_{22}) \\ Q &= (A_{21}+A_{22})B_{11} \\ R &= A_{11}(B_{12}-B_{22}) \\ S &= A_{22}(B_{21}-B_{11}) \\ T &= (A_{11}+A_{12})B_{22} \\ U &= (A_{21}-A_{11})(B_{11}+B_{12}) \\ V &= (A_{12}-A_{22})(B_{21}+B_{22}) \end{aligned} \qquad (3.13)$$

$$\begin{aligned} C_{11} &= P+S-T+V \\ C_{12} &= R+T \\ C_{21} &= Q+S \\ C_{22} &= P+R-Q+U \end{aligned} \qquad (3.14)$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2)+an^2 & n > 2 \end{cases} \qquad (3.15)$$

Strassen's matrix multiplication- Example:

$$AB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

We define the following eight $n/2$ by $n/2$ matrices:

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} \quad A_{12} = \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix} \quad B_{11} = \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \quad B_{12} = \begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} \quad A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \quad B_{21} = \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix} \quad B_{22} = \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

$$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$$

$$P_2 = (A_{21} + A_{22}) \times B_{11} = \begin{bmatrix} 14 & 23 \\ 14 & 23 \end{bmatrix}$$

$$P_3 = A_{11} \times (B_{12} - B_{22}) = \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) = \begin{bmatrix} -3 & 2 \\ 5 & -20 \end{bmatrix}$$

$$P_5 = (A_{11} + A_{12}) \times B_{22} = \begin{bmatrix} 34 & 18 \\ 45 & 9 \end{bmatrix}$$

$$P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = \begin{bmatrix} 3 & 27 \\ -18 & -18 \end{bmatrix}$$

$$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = \begin{bmatrix} 18 & 16 \\ 3 & 0 \end{bmatrix}$$

$$C_{11} = P_1 + P_4 - P_5 + P_7 = \begin{bmatrix} 17 & 22 \\ 21 & 18 \end{bmatrix}$$

$$C_{12} = P_3 + P_5 = \begin{bmatrix} 31 & 30 \\ 33 & 33 \end{bmatrix}$$

$$C_{21} = P_2 + P_4 = \begin{bmatrix} 11 & 25 \\ 19 & 3 \end{bmatrix}$$

$$C_{22} = P_1 + P_3 - P_2 + P_6 = \begin{bmatrix} 22 & 38 \\ 14 & 30 \end{bmatrix}$$

$$C = \begin{bmatrix} 17 & 22 & 31 & 30 \\ 21 & 18 & 33 & 33 \\ 11 & 25 & 22 & 38 \\ 19 & 3 & 14 & 30 \end{bmatrix}$$

Strassen's matrix multiplication- Time Complexity:

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \qquad (3.15)$$

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n},\ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

****

DESIGN AND ANALYSIS OF ALGORITHM – GREEDY METHOD

Unit 2.1 Topics: General method- Knapsack Problem- Job scheduling with deadlines- Minimum cost spanning trees- Optimal storage on tapes- Single source shortest path

Greedy Method-General Method:

- It is straightforward design technique and applied to a wide variety of problems.
- Most of these problems have n inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies those constraints is called a **feasible solution.**
- We need to find a feasible solution that either maximizes or minimizes a given objective function. A Feasible solution that does this is called an **optimal solution.**
- The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time.
- A Greedy technique that will result in algorithm those general sub optimal solutions is called **subset paradigm.**

Greedy Algorithm:

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

- The function **Select** selects an input from a[ ] and removes it.
- The selected input's value is assigned to x.
- **Feasible** is a Boolean-valued function that determines whether x can be included into the solution vector.
- The function **Union** combines x with the solution and updates the objective function.
- Once a particular problem is chosen and the functions Select, Feasible and Union are properly implemented.

Knapsack Problem:

- The greedy method is applied to solve the knapsack problem.
- We are given n objects and a knapsack or bag.
- Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 < x_i < 1$, 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 the problem can be stated as:

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.3: There are four items that have a profit and weight list below. The knapsack capacity is 4 kgs. We apply the above three greedy strategies.

item (i)	p_i	w_i
1	5	1
2	9	3
3	4	2
4	8	2

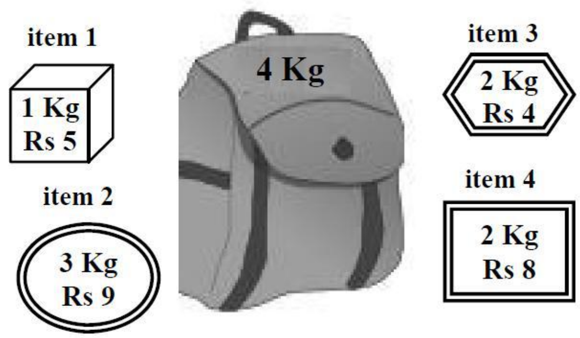


Figure 4.3: A Knapsack problem

Algorithm for greedy strategies for knapsack problem

```
Algorithm: Greedy knapsack(m,n)
// p[1:n] and w[1:n] contains the profits & weights respectively
// of the n objects ordered such that p[i]/w[i] >= p[i+1]/w[i+1]
// m is the knapsack size and x[1:n] is the solution vector.
{
    for i:=1 to n do x[i]:=0.0; // Initialize x
    U:=m;
    for i:=1 to n do
    {
        if (w[i] > U) then break;
        x[i]:=1.0; U = U - w[i];
    }
    if (i<=n) then x[i]:= U/w[i];
}
```

Knapsack Problem-Example:

Example 4.1 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)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance.

Solution: This knapsack problem can be solved by using Greedy method.

Let x_i is a fraction whose value is $0 \le x_i \le 1$.

$n = 3 \rightarrow x_1, x_2, x_3$

$n = 3 \rightarrow$ we have $(n+1)$ feasible solutions $\rightarrow$ "4 solutions"

4 solutions can be calculated →
1. Based on Assumption
2. Based on Algorithm
3. Based on Assumption
4. Based on Algorithm

1st solution: Based on Assumption

$n = 4$ 1, 2, 3, 4

Divide one by remaining numbers i.e ½, ⅓, ¼

solution 1 = (½, ⅓, ¼)

2nd solution: Based on Algorithm

$W[1] = 18$
$W[2] = 15$
$W[3] = 10$

$i = 1, 2, 3 \rightarrow X[1] = 0, X[2] = 0, X[3] = 0$ $U = 20$

$i = 1$: $W[1] > 20$
$18 > 20$ F $\rightarrow X[1] = 1$, $U = 20 - 18 = 2$

$i = 2$: $W[2] > 2$
$15 > 2$ F $\rightarrow$ break $\rightarrow X[2] = U/W[i] = 2/15$

$i = 3$: $W[3] > 2$
$10 > 2$ T $\rightarrow$ break $\rightarrow X[3] = U/W[i] = 2/10 = 0.5 \simeq 0$

solution 2 = (1, 2/15, 0)

3rd solution: Based on Assumption

Solution1	1/2	1/3	1/4
Solution2	1	2/15	0
Solution3	0	2/3	1

Exchange 1st & 3rd
Take numerator on solution2
Take Denominator on solution1

4th solution: Based on Algorithm

put X[1] = 0 and find X[2] & X[3] with help of Algorithm.

i = 2, 3 → X[2] = 0, X[3] = 0 U = 20

i = 2: W[2] > 20
15 > 20 F → X[2] = 1, U = 20 − 15 = 5

i = 3: W[3] > 5
10 > 5 T → break → X[3] = U/W[3] = 5/10 = 1/2   X[3] = 1/2

Solution4 = (0, 1, 1/2)

Types of Knapsack Problem:

Two types of Knapsack problem:

1. 0/1 knapsack problem – solves it either selecting each item as whole or none.
   ↳ solved by Dynamic programming.
2. Fractional knapsack problem
   ↳ solves it by allowing fractional to maximise
   ↳ solved by Greedy Method.

Fractional knapsack

```
Algorithm Fractional-Knapsack (W[1:n], P[1:n], W)
  for i = 1 to n do
      x[i] = 0
    weight = 0
  for i = 1 to n
    if weight + W[i] ≤ W then
        x[i] = 1
        weight = weight + W[i]
    else
      x[i] = (W - weight)/W[i]
        weight = W
        break
  return x
```

Fractional Knapsack- Example Problem:

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Pi/ Wi	7	10	6	5

Arranging the above the tables with descending order of **Pi/ Wi**

Item	B	A	C	D
Profit	100(P1)	280(P2)	120(P3)	120(P4)
Weight	10 (W1)	40(W2)	20(W3)	24(W4)
Pi/ Wi	10	7	6	5

Consider the knapsack capacity W=60

Algorithm Analysis

i = 1, 2, 3, 4   x[1] = 0, x[2] = 0, x[3] = 0, x[4] = 0

weight = 0 $(W_1, W_2, W_3, W_4) = (10, 40, 20, 24)$ $(P_1, P_2, P_3, P_4) = (100, 280, 120, 120)$

i = 1: 0 + 10 ≤ 60 T

x[1] = 1

weight = 0 + 10 = 10

B A C D

solution = (1, 1, 1, 0)

i = 2: 10 + 40 ≤ 60 T

x[2] = 1

weight = 10 + 40 = 50

i = 3: 50 + 20 ≤ 60 F

x[3] = (60 − 50)/20 = 10/20

weight = 60 → break

Total weight = 10 + 40 + 20 * (10/20) = 10 + 40 + 10 = 60 ✓

Total Profit = 100 + 280 + 120 * (10/20) = 380 + 60 = 440 ✓

optimal solution = (1, 1, 1, 0)
B A C D

Based on capacity, selecting B, A and ½th of C item for getting maximum profit

Job Scheduling With Deadlines:

- Greedy Method is applied for this problem.
- Initially we are given a set of n jobs.
- **Associated with job i is an integer deadline di >=0 and a profit pi >0.**
- **For any job i the profit pi is earned iff the job is completed by its deadline**.

Conditions:

- 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.
- **An optimal solution is a feasible solution with maximum value.**

Algorithm for Job Scheduling:

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

Algorithm 4.5 High-level description of job sequencing algorithm

Algorithm for Job Scheduling with Deadlines:

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ 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]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ 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;
}
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

Example1:

Let number of jobs= n=3

(p1, p2, p3, p4)= (100, 10, 15, 27)   deadline=1→ job should be done on first day

(d1, d2, d3, d4)= (2, 1, 2,1)   deadline=2→ job should be done on first day or second day

Here maximum deadline=2 means only two jobs can be done per day & no parallel execution of jobs is done.

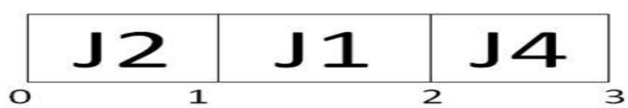
Feasible Solution	Processing Sequence	Value	Explanation
(1, 2)	2,1	110	2's deadline <1's deadline
(1, 3)	1,3 or 3,1	115	1's deadline = 3's deadline
(1,4)	**4,1**	**127 (Maximum Profit)**	4's deadline <1's deadline
(2, 3)	2, 3	25	2's deadline < 3's deadline
(2,4)	Impossible because of parallel execution		Both are having deadline=1
(3, 4)	4, 3	42	4's deadline <3's deadline
(1)	1	100	
(2)	2	10	
(3)	3	15	
(4)	4	27	

Example2:

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40.



<u>Minimum Cost Spanning Trees</u>:

- **A Spanning Tree of a graph is a tree that has all the vertices of the graph is connected by some edges.**
- A Graph can have one or more spanning trees.
- If a graph contains n vertices, then spanning trees contains (n-1) edges.
- A **Minimum Cost Spanning Tree (MST)** is a spanning tree that has minimum weight than all other spanning trees of the graph.
- **A Spanning Tree does not contain cycles.**
- Two Algorithms are used to find **Minimum Cost Spanning Tree:**
  1. Prim's Algorithm.
  2. Kruskal Algorithm.

Figure 4.5 An undirected graph and three of its spanning trees

1. Prim’s Algorithm:

In this algorithm we choose a neighboring or adjacent vertex for finding minimum cost spanning tree.

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

Algorithm 4.8 Prim’s minimum-cost spanning tree algorithm

Example:

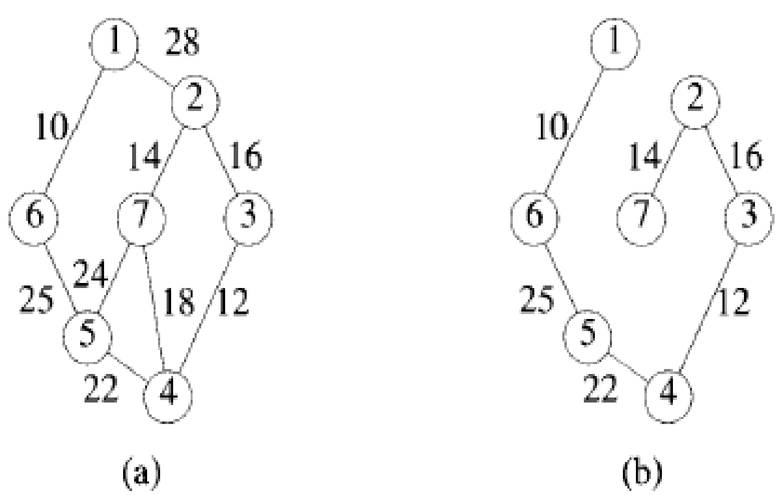


Figure 4.6 A graph and its minimum cost spanning tree

Node	cost	
1	1-2	28
	1-6	10 ✓
2	2-1	28
	2-3	16
	2-7	14 ✓
3	3-2	16 ✓
	3-4	12
4	4-3	12 ✓
	4-5	22
	4-7	18

Node	cost	
5	5-4	22 ✓
	5-6	25
	5-7	24
6	6-1	10
	6-5	25 ✓
7	7-2	14
	7-4	18
	7-5	24

Min cost spanning tree = 10 + 12 + 14 + 16 + 22 + 25 = 99

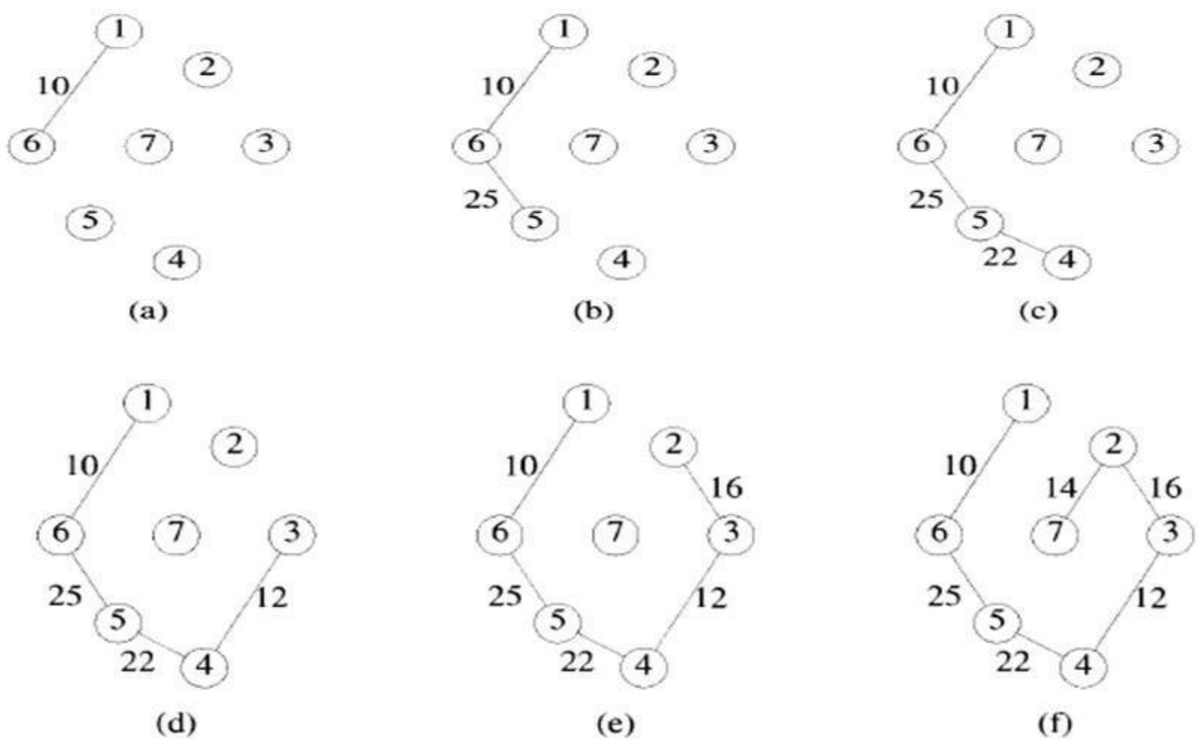


Figure 4.7 Stages in Prim's algorithm

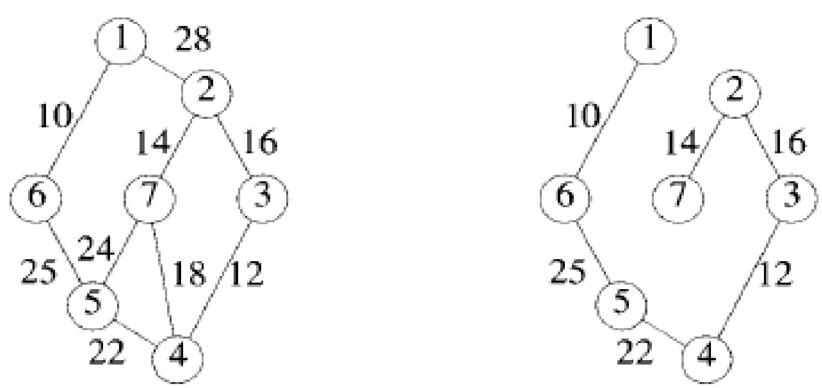
2. Kruskal Algorithm:

In this algorithm we list out costs between vertices in ascending order & add one by one vertex to spanning tree which doesn't forms any cycles.

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

Algorithm 4.10 Kruskal's algorithm

Example:



(a) (b)

Figure 4.6 A graph and its minimum cost spanning tree

Note down the cost of edges in ascending order

1-6	10	✓
3-4	12	✓
2-7	14	✓
2-3	16	✓
4-7	18	→ form's cycle.
4-5	22	✓
5-7	24	→ form's cycle
5-6	25	✓
1-2	28	→ form's cycle.

Min cost Spanning tree = 10 + 12 + 14 + 16 + 22 + 25 = 99

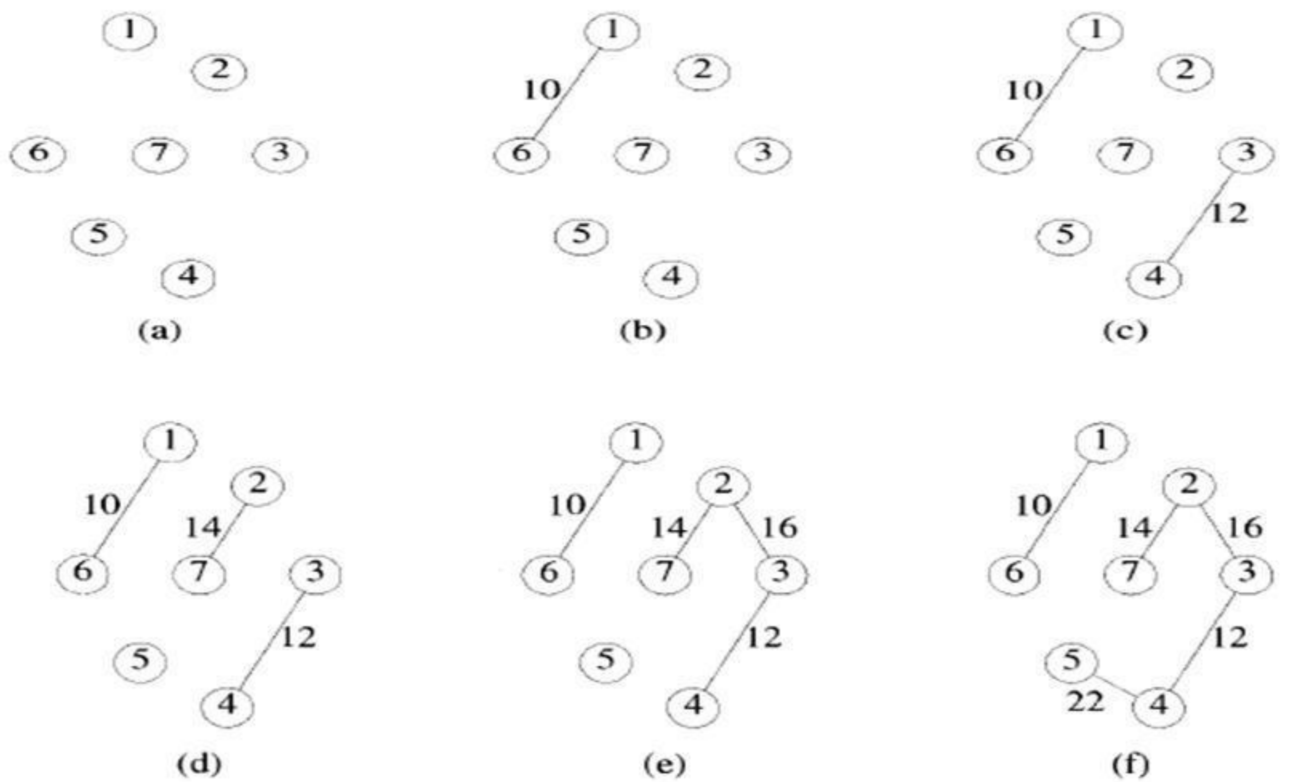


Figure 4.8 Stages in Kruskal's algorithm

Optimal storage on tapes:

Initially tape position is at front

Tape → n programs are to be stored
↓
length l

↳ for each program i, we have a length l_i $1 \le i \le n$

in a order $I = i_1, i_2, \ldots i_n$, the time t_j needed to retrieve from this tape is proportional to $\sum_{1 \le k \le j} l_{i_k}$ programs are stored

If all programs are retrieved equally often, then expected or Mean Retrieval Time (MRT) is $\left(\frac{1}{n}\right) \sum_{1 \le j \le n} t_j$

based on the order stored in tape, we minimize MRT

Minimizing MRT is equivalent to minimizing $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$

Example:

Let number of inputs=3

(l1, l2, l3)= (5, 10, 3)

For n number of jobs we have n! possible orderings.

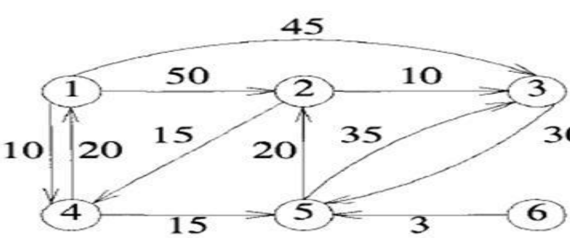
For 3 jobs we have 3! = 6 possible orderings.

Ordering I	d(I)	MRT
1, 2, 3	5 +(5+10)+(5+10+3)	38
1, 3, 2	5 +(5+3)+(5+3+10)	31
2, 1, 3	10+(10+5)+(10+5+3)	43
2, 3, 1	10+(10+3)+(10+3+5)	41
3, 1 ,2	**3+(3+5)+(3+5+10)**	**29**
3, 2,1	3+(3+10)+(3+10+5)	34

The optimal ordering is **3, 1, 2** is having minimum MRT value as 29.

Single-Source Shortest Paths:

- **Graphs** can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges can then be **assigned weights** which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway.



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

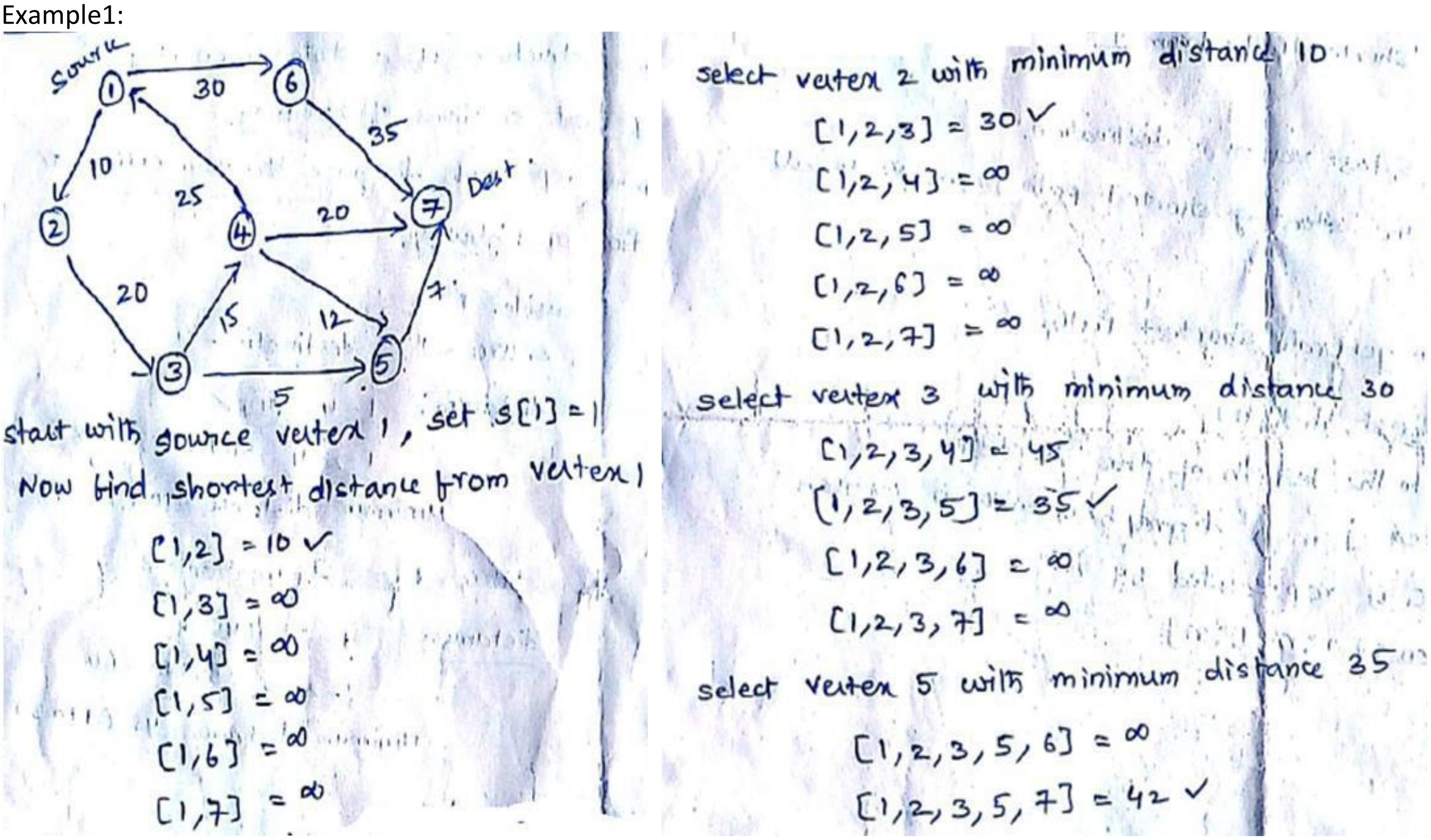
Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

Single-Source Shortest Paths- Algorithm:

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n − 1 do
    {
        // Determine n − 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```

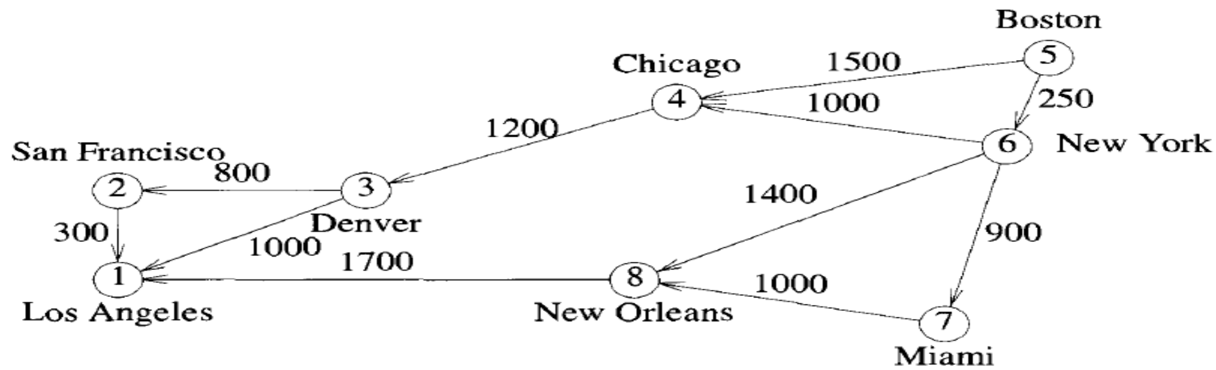
Algorithm 4.14 Greedy algorithm to generate shortest paths

Example1:



For the example, to reach source to destination (1 to 7) we have shortest path with value 42.

Example2:



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Iteration	S	Vertex selected	Distance							
			LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

****

Unit 2.2 Topics: General Method, Multistage graphs, All-pairs shortest paths, Optimal binary search trees, 0/1 knapsack, The traveling sales person problem.

Dynamic Programming- General Method:

- **Dynamic programming** is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a **sequence of decisions**.
- Both Greedy Method & Dynamic Programming solve a problem by breaking it down into several sub-problems that can be solved recursively.
- **Dynamic programming is a bottom-up technique** that usually begins by solving the smaller sub-problems, saving these results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained.
- Whereas **divide-and-conquer approach**, which solves problems in a **top-down method**.

Example:

```
fib(n)
{   if (n = 0) return 0;
    if (n = 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

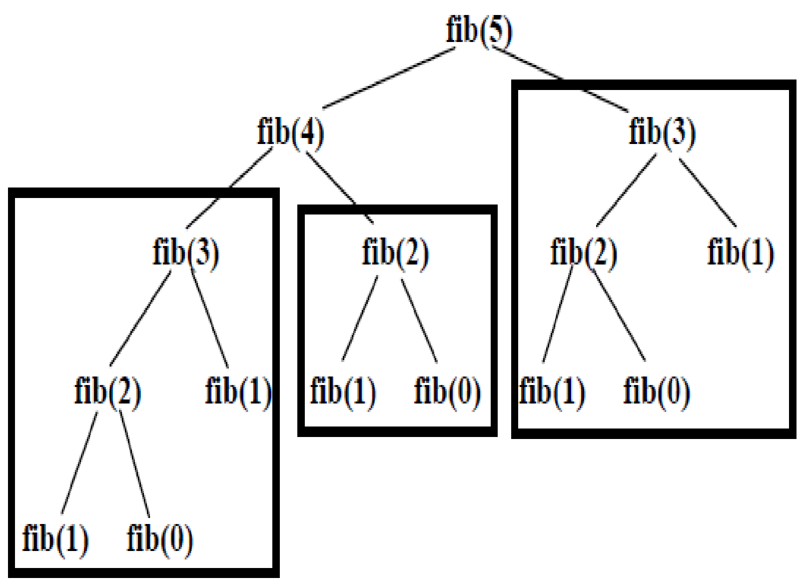


Figure 5.1: Recursive computation of Fibonacci numbers

- In Dynamic Programming, an optimal sequence of decisions is obtained by making explicit appeal to The **Principle of Optimality**.
- **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 decision.
- **Steps in Dynamic Programming**:
  1. Characterize the structure of optimal solution.
  2. Recursively defines the value of optimal solution.
  3. The optimal solution has to be constructed from information.
  4. Compute an optimal solution from computed information.

Difference between Greedy Method & Dynamic Programming:

Greedy Method	Dynamic Programming
1. It is used for obtaining optimal solution.	1. It is also used for obtaining optimal solution.
2. In this, a set of feasible solutions are generated, among these we select an optimal solution.	2. There is no set of feasible solutions.
3. Optimal Solution is generated without revising previously generated solution.	3. It considers all possible sequences in order to obtain optimal solution.
4. No guarantee of optimal solution.	4. Guarantee of optimal solution is achieved using principle of optimality.

Multistage graphs:

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.

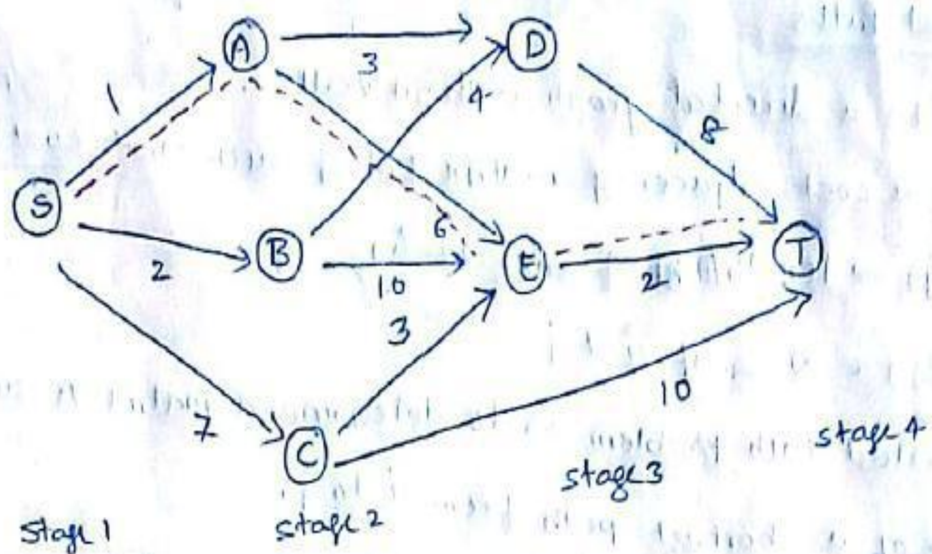
- Let s is a source vertex & t is sink(destination) vertex.
- Let c (i , j)= cost of edges of (i , j).
- The cost of path from s to t is sum of cost of the edges on the path.
- **The multi stage graph problem is to find minimum cost path from s to t.**
- Two approaches in multi stage graph:
  1. Forward approach.
  2. Backward approach.

Multistage graphs- Forward Approach:

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

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

Multistage graphs- Forward Approach- Example:



Forward Approach

d(S,A) = 1
d(S,B) = 2
d(S,C) = 7

d(D,T) = 8
d(E,T) = 2
d(C,T) = 10

d(S,D) = min{1 + d(A,D), 2 + d(B,D)}

= min{1+3, 2+4} = 4

d(S,D) = 4

d(S,E) = min{2 + d(B,E), 1 + d(A,E), 7 + d(C,E)}

= min{2+10, 1+6, 7+3} = 7

d(S,E) = 7

d(S,T) = min{d(S,D) + d(D,T), d(S,E) + d(E,T), d(S,C) + d(E,T)}

= min{4+8, 7+2, 7+10} = 9

d(S,T) = 9

Path choosen to reach S to T is S-A-E-T

Multistage graphs- Backward Approach:

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

Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

Multistage graphs- Backward Approach-Example:

Backward Approach

$d(D,T) = 8$

$d(E,T) = 2$

$d(C,T) = 10$

$d(A,T) = \min\{3 + d(D,T), 6 + d(E,T)\}$

$= \min\{3+8, 6+2\} = 8$ A - E - T $d(A,T) = 8$

$d(B,T) = \min\{4 + d(D,T), 10 + d(E,T)\}$ A - D - T

$= \min\{4+8, 10+2\} = 12$ A - E - T $d(B,T) = 12$

$d(C,T) = \min\{3 + d(E,T), d(C,T)\}$ C - E - T $d(C,T) = 5$

$= \min\{3+2, 10\} = 5$

$d(S,T) = \min\{1 + d(A,T), 2 + d(B,T), 7 + d(C,T)\}$

$= \min\{1+8, 2+12, 7+5\} = 9$

$d(S,T) = 9$ Path choosen : S - A - E - T

All-pairs shortest paths:

Let $G = (V, E)$ be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that $cost(i,i) = 0$, $1 \le i \le n$. Then $cost(i,j)$ is the length (or cost) of edge $\langle i,j \rangle$ if $\langle i,j \rangle \in E(G)$ and $cost(i,j) = \infty$ if $i \ne j$ and $\langle i,j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j.

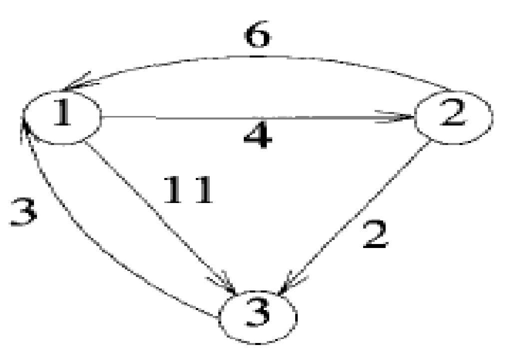
$$A^k(i,j) = \min\ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}, \quad k \ge 1$$

All-pairs shortest paths-Algorithm:

```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Algorithm 5.3 Function to compute lengths of shortest paths

Example:



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

i=3, j=2,k=1

$$A[3,2] = \min\{A[3,2],\ A[3,1] + A[1,2]\}$$
$$= \min\{\infty, 3+4\} = 7$$

↓ via node 1

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

i=1, j=3,k=2

$$A[1,3] = \min(A[1][3],\ A[1,2] + A[2,3]) \rightarrow \text{via node2}$$
$$= \min(11, 4+2) = 6$$

i=2, j=1,k=3

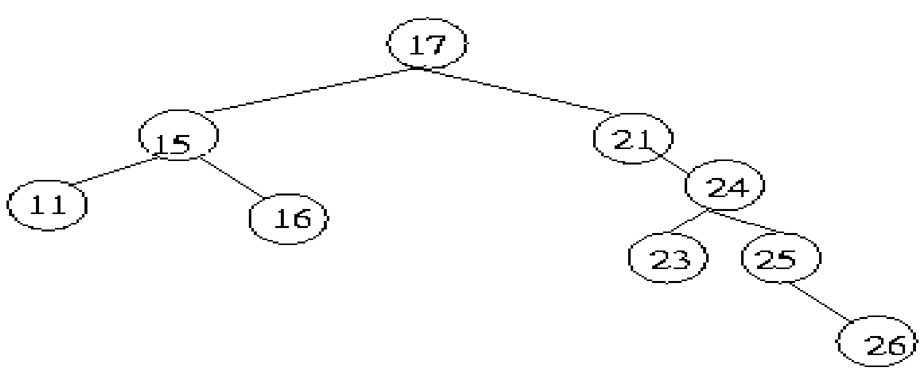
$$A[2,1] = \min(A[2,1],\ A[2,3] + A[3,1])$$
$$= \min(6, 2+3) = 5$$

Optimal binary search trees:

A **binary search tree** is a tree where the key values are ordered that all the keys in the left sub tree are less than the keys in the node, and all the keys in the right sub tree are greater. Clearly

LST<=ROOT<=RST

LST= Left Sub Tree & RST= Right Sub Tree



Optimal binary search trees- Algorithm:

```
Algorithm OBST(p, q, n)
// Given n distinct identifiers a_1 < a_2 < ··· < a_n and probabilities
// p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
// the cost c[i, j] of optimal binary search trees t_ij for identifiers
// a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
// w[i, j] is the weight of t_ij.
{
    for i := 0 to n − 1 do
    {
        // Initialize.
        w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
        // Optimal trees with one node
        w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
        r[i, i + 1] := i + 1;
        c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
    }
    w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
    for m := 2 to n do // Find optimal trees with m nodes.
        for i := 0 to n − m do
        {
            j := i + m;
            w[i, j] := w[i, j − 1] + p[j] + q[j];
            // Solve 5.12 using Knuth's result.
            k := Find(c, r, i, j);
                // A value of l in the range r[i, j − 1] ≤ l
                // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
            c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
            r[i, j] := k;
        }
    write (c[0, n], w[0, n], r[0, n]);
}
```

```
Algorithm Find(c, r, i, j)
{
    min := ∞;
    for m := r[i, j − 1] to r[i + 1, j] do
        if (c[i, m − 1] + c[m, j]) < min then
        {
            min := c[i, m − 1] + c[m, j]; l := m;
        }
    return l;
}
```

Algorithm 5.5 Finding a minimum-cost binary search tree

→ OBST is given with list of identifiers $\{a_1, a_2 \cdots a_n\}$ $a_1 < a_2 < \cdots a_n$

Let $p(i)$ be probabilities for successful search for a_i

Let $q(i)$ be probabilities for unsuccessful search.

$$\text{clearly} \sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} q(i) = 1$$

→ A tree with minimum cost is obtained by adding p_i & q_i

→ A binary search tree with optimal cost is called Optimal Binary Search Tree

→ cost is calculated with

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

$c(i,i) = 0$ $r(i,j) = k.$

$w(i,i) = q_i$ $0 \le i \le n$

$r(i,i) = 0$ $0 \le i \le n$

$w(i,j) = p_j + q_j + w(i,j-1)$

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

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

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

solution: $c(i,i) = 0$, $w(i,i) = q_i$, $r(i,i) = 0$

$w_{00} = q_0 = 2$	$c_{00} = 0$	$r_{00} = 0$
$w_{11} = q_1 = 3$	$c_{11} = 0$	$r_{11} = 0$
$w_{22} = q_2 = 1$	$c_{22} = 0$	$r_{22} = 0$
$w_{33} = q_3 = 1$	$c_{33} = 0$	$r_{33} = 0$
$w_{44} = q_4 = 1$	$c_{44} = 0$	$r_{44} = 0$

$r(i,j) = k \quad W(i,j) = P_j + q_j + w(i, j-1)$

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

$(P_1, P_2, P_3, P_4) = (3, 3, 1, 1)$

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

$W_{00} = 2$

$W_{11} = 3$

$W_{22} = 1$

$W_{33} = 1$

$W_{44} = 1$

→ $W_{01} = P_1 + q_1 + W_{00} = 3 + 3 + 2 = 8$

$C_{01} = W_{01} + \min_{0<k\le 1} \{ C_{00} + C_{11} \} = 8 + \min\{0+0\} = 8 \quad (k=1)$

$r_{01} = k = 1$

$W_{01} = 8, C_{01} = 8, r_{01} = 1$

→ $W_{12} = P_2 + q_2 + W_{11} = 3 + 1 + 3 = 7$

$C_{12} = W_{12} + \min_{1<k\le 2} \{ C_{11} + C_{22} \} = 7 + \min\{0+0\} = 7 \quad (k=2)$

$r_{12} = k = 2$

$W_{12} = 7, C_{12} = 7, r_{12} = 2$

→ $W_{23} = P_3 + q_3 + W_{22} = 1 + 1 + 1 = 3$

$C_{23} = W_{23} + \min_{2<k\le 3} \{ C_{22} + C_{33} \} = 3 + \min\{0+0\} = 3 \quad (k=3)$

$r_{23} = k = 3$

$W_{23} = 3, C_{23} = 3, r_{23} = 3$

→ $W_{34} = P_4 + q_4 + W_{33} = 1 + 1 + 1 = 3$

$C_{34} = W_{34} + \min_{3<k\le 4} \{ C_{33} + C_{44} \} = 3 + \min\{0+0\} = 3 \quad (k=4)$

$r_{34} = k = 4$

$W_{34} = 3, C_{34} = 3, r_{34} = 4$

→ $W_{02} = P_2 + q_2 + W_{01} = 3 + 1 + 8 = 12$

$k=1$

$C_{02} = W_{02} + \min_{0<k\le 2} \{ C_{00} + C_{12} \} = 12 + \min\{0+7\} = 12 + 7 = 19$

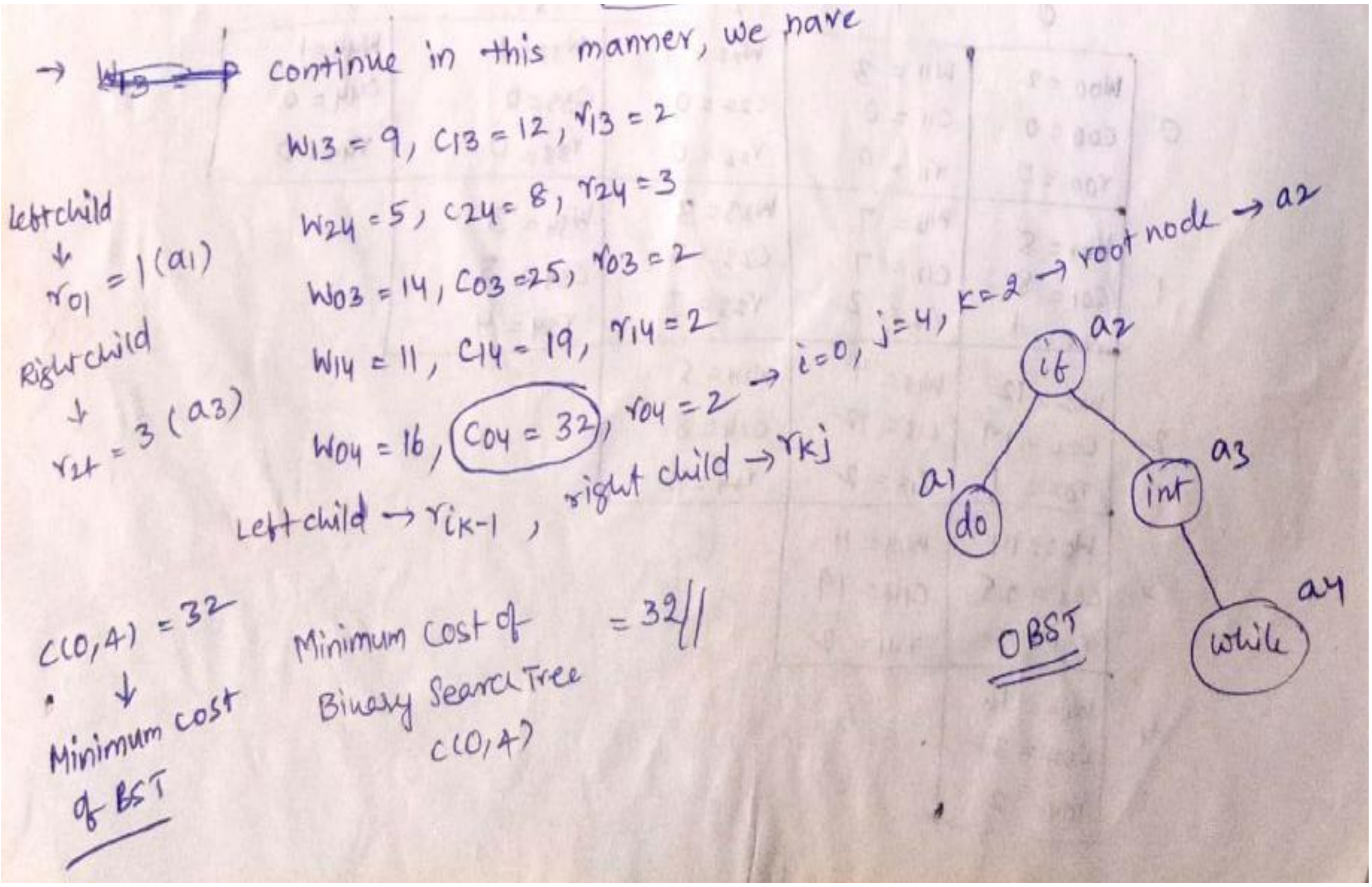
$k=2$

$C_{02} = W_{02} + \min\{ C_{01} + C_{22} \} = 12 + \min\{8+0\} = 12 + 8 = 20$

$k=1 \rightarrow C_{02} = 19$
$k=2 \rightarrow C_{02} = 20$
} select minimum value $C_{02} = 19, k = 1$

$r_{02} = k = 1$

$W_{02} = 12, C_{02} = 19, r_{02} = 1$



	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 5.16 Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$

0/1 knapsack:

- Knapsack is a bag which has a capacity M.
- In fractional knapsack, we place the items one by one by considering Pi/ Wi.
- In 0/1 Knapsack, we cannot consider any fractions.
- **In Dynamic Programming, we consider knapsack problem for placing maximum elements with maximum profit & weight that does not exceed knapsack capacity.**
- **0 means we cannot consider that item.**
- **1 means we consider that item to placed in knapsack.**
- The solution for Dynamic Programming is:

$$f_n(m) = \max \ \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\}$$

0/1 knapsack- Algorithm:

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

Algorithm 5.6 Informal knapsack algorithm

0/1 knapsack- Example:

Solving 0/1 knapsack problem by using dynamic programming follows 3 steps:

① Addition operation: $S_1^i = S^i + (P_{i+1}, W_{i+1})$

② Merging operation: $S^{i+1} = S^i \cup S_1^i$

③ Purging Rule / Dominance Rule

Take any two sets S^i & S_1^i consisting (P_j, W_j) and (P_K, W_K)

The purging rule states that if $P_j \le P_K$ & $W_j \ge W_K$ then (P_j, W_j) is deleted

If $(P_i, w_i) \in S^n$
$(P_i, w_i) \in S^{n-1}$ → then set $x_n = 0$ otherwise $x_n = 1$

Example: knapsack capacity $m = 6$ | no. of inputs, $n = 3$

$(P_1, w_1) = (1,2)$; $(P_2, w_2) = (2,3)$; $(P_3, w_3) = (5,4)$

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

$S_1^0 = S^0 + (P_1, w_1) = \{(0,0)\} + \{(1,2)\} = \{(1,2)\}$

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

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

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

No purging rule is applied since $0 \le 1$ & $0 \le 2$

$S_1^1 = S^1 + (P_2, w_2) = \{(0,0), (1,2)\} + \{(2,3)\}$
$= \{(2,3), (3,5)\}$

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

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

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

No purging rule is applied.

$S_1^2 = S^2 + (P_3, W_3)$

$= \{(0,0), (1,2), (2,3), (3,5)\} + (5,4)$

$S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = S^2 \cup S_1^2$

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

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

consider two tuples $(3,5)$ & $(5,4)$ $3 \le 5$ & $5 \ge 4$
$P_j\ W_j$ $P_k\ W_k$

Apply purging rule → (P_j, W_j) is removed i.e $(3,5)$ is deleted

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

since knapsack capacity $m = 6$ only discard $(7,7)$ & $(8,9)$ exceeds

$m = 6$ → consider $(6,6)$ ↳ max capacity

$(6,6) \in S^3$
$(6,6) \notin S^2$ then mark $x_3 = 1$

To get Next tuple $(6,6) - (P_3, W_3)$

$(6,6) - (5,4) = (1,2)$

$(1,2) \in S^2$
$(1,2) \in S^1$ then mark $x_2 = 0$

$(1,2) \in S^1$
$(1,2) \notin S^0$ then mark $x_1 = 1$

Hence the optimal solution is $(x_1, x_2, x_3) = (1,0,1)$

<u>The traveling sales person problem:</u>

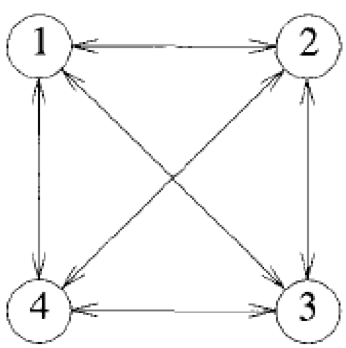
- Let G=(V,E) be a directed graph with edge costs Cij.
- The variable Cij is defined such that
  - **Cij >0 for all i & j.**
  - **Cij= ∞ if (i ,j) € E.**
- Let |V|= n and assume that n>1.
- **A tour of G is a directed simple cycle that includes every vertex in V.**
- **The cost of a tour is the sum of the cost of the edges on the tour.**
- **The traveling sales person problem is to find a tour of minimum cost.**
- **<u>Notations</u>:**
- g(i, s) = length of the shortest path starting at vertex i, going through all vertices in s and terminating at vertex 1.
- g(1, V-{1}) = length of an optimal sales person tour.
- **Principle of Optimality states that**

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \tag{5.20}$$

Generalizing (5.20), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \tag{5.21}$$

<u>Example:</u>



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Figure 5.21 Directed graph and edge length matrix c

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

$|S| = 0$ $g(2, \phi) = c_{21} = 5$ ✓

$g(3, \phi) = c_{31} = 6$ 1-2-4-3-1

$g(4, \phi) = c_{41} = 8$ select vertex $S = 1$

We obtain

$|S| = 1$

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

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

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

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

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

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

1-2-4-3

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

1-2-4

$g(2, \{3, 4\}) = \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$

$= \min\{9 + 20, 10 + 15\} = \min\{29, 25\} = 25$

$g(3, \{2, 4\}) = \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\}$

$= \min\{13 + 18, 12 + 13\} = \min\{31, 25\} = 25$

$g(4, \{2, 3\}) = \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$= \min\{8 + 15, 9 + 18\} = \min\{23, 27\} = 23$

|S| = 3

Finally, we obtain

$$g(1,\{2,3,4\}) = \min\{c_{12}+g(2,\{3,4\}), c_{13}+g(3,\{2,4\}), c_{14}+g(4,\{2,3\})\}$$

(1-2 circled around $c_{12}+g(2,\{3,4\})$)

$$= \min\{10+25, 15+25, 20+23\}$$

$$= \min\{35, 40, 43\}$$

$$= 35$$

Optimal tour of Graph has length 35.

$J(1,\{2,3,4\}) = 2$ tour starts from 1 & goes to 2.

$J(2,\{3,4\}) = 4$ next edge (2,4)

$g(4,\{3\}) = 3$ next edge (4,3)

The Optimal tour is 1, 2, 4, 3, 1

****

DESIGN AND ANALYSIS OF ALGORITHM

Unit 3.1 Topics: **Basic Traversal And Search Techniques-** Techniques for Binary Trees- Techniques for Graphs- Connected Components and Spanning Trees- Bi-connected components and DFS

3.1.1 Techniques for Binary Trees:

- In Normal Tree, any node can have any number of children.
- Binary Tree is a special tree in which every node can have a maximum of two children.
- In Binary Tree, each node can have 0 or 1 or 2 children but not more than 2 children.
- Displaying or Visiting order of nodes in binary trees is called Binary Tree Traversal.
- 3 Types of Binary Tree Traversals:
    1. In-order (LVR) Traversal- order: left child, root node, right child.
    2. Pre-order (VLR) Traversal -order: root node, left child, right child.
    3. Post-order (LRV) Traversal-order: left child, right child, root node.

Algorithm for In-order traversal

```
treenode = record
{
    Type data; // Type is the data type of data.
    treenode *lchild; treenode *rchild;
}

Algorithm InOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        InOrder(t → lchild);
        Visit(t);
        InOrder(t → rchild);
    }
}
```

Algorithm 6.1 Recursive formulation of inorder traversal

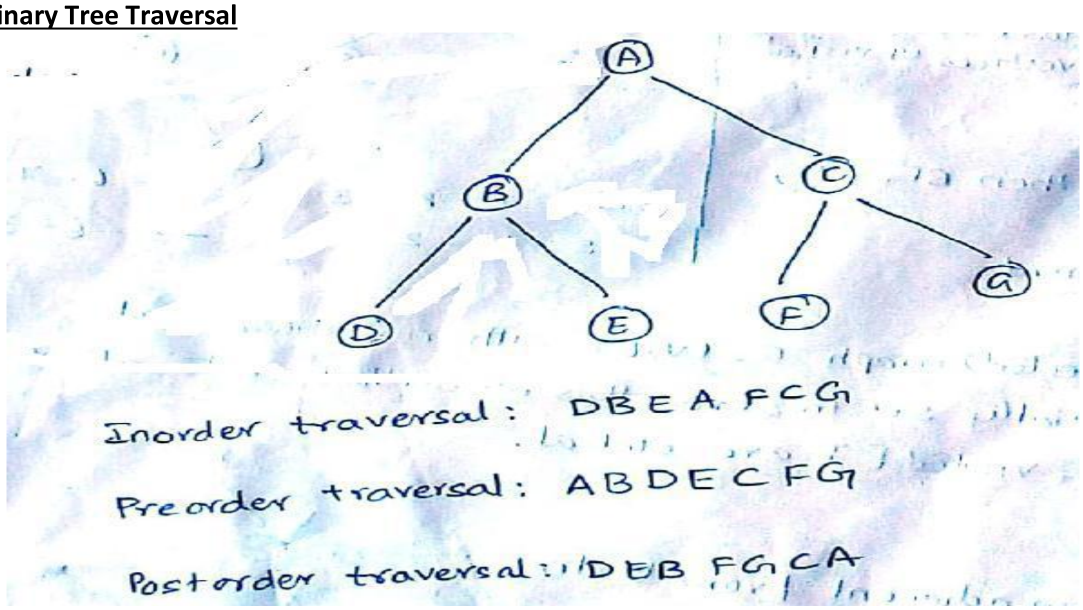
Algorithm for Pre-order traversal

```
Algorithm PreOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        Visit(t);
        PreOrder(t → lchild);
        PreOrder(t → rchild);
    }
}
```

Algorithm for Post-order traversal

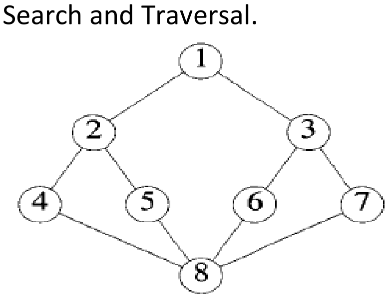
```
Algorithm PostOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        PostOrder(t → lchild);
        PostOrder(t → rchild);
        Visit(t);
    }
}
```

Example for Binary Tree Traversal

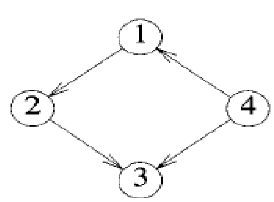


3.1.2 Techniques for Graphs

- A Graph G = (V, E) is defined such that this path starts at vertex v and ends at vertex u.
- We describe two search methods for this:
    1. Breadth First Search and Traversal.
    2. Depth First Search and Traversal.



(a) Undirected graph *G*



(b) Directed graph

Breadth First Search and Traversal (level by level traversing)

- In breadth first search we start at a vertex v and mark it as having been reached (visited).
- **The vertex v is at this time said to be unexplored.**
- A vertex is said to have been **explored** by an algorithm when the algorithm has **visited all vertices adjacent from it**.
- The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices.
- The first vertex on this list is the next to be explored.
- Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as a queue and can be represented using any of the **standard queue representation.**

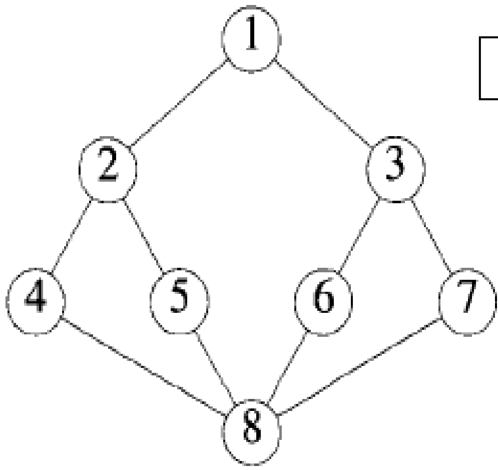
Algorithm for BFS

```
Algorithm BFS(v)
// A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i] = 1 if i has
// already been visited. The graph G and array visited[ ]
// are global; visited[ ] is initialized to zero.
{
    u := v; // q is a queue of unexplored vertices.
    visited[v] := 1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if (visited[w] = 0) then
            {
                Add w to q; // w is unexplored.
                visited[w] := 1;
            }
        }
        if q is empty then return; // No unexplored vertex.
        Delete u from q; // Get first unexplored vertex.
    } until(false);
}
```

Algorithm for Breadth first graph traversal

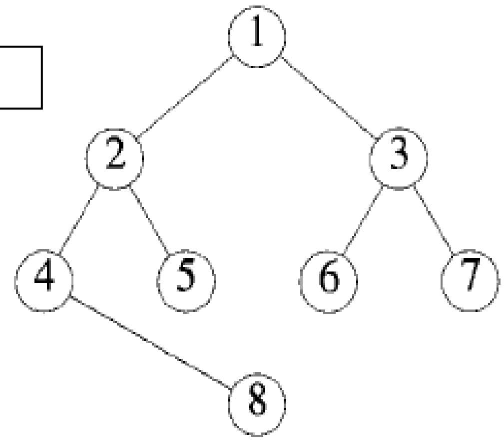
```
Algorithm BFT(G, n)
// Breadth first traversal of G
{
    for i := 1 to n do  // Mark all vertices unvisited.
        visited[i] := 0;
    for i := 1 to n do
        if (visited[i] = 0) then BFS(i);
}
```

Example for BFS



BFS Order: 1, 2, 3, 4, 5, 6, 7, 8

Applying BFS



BFS

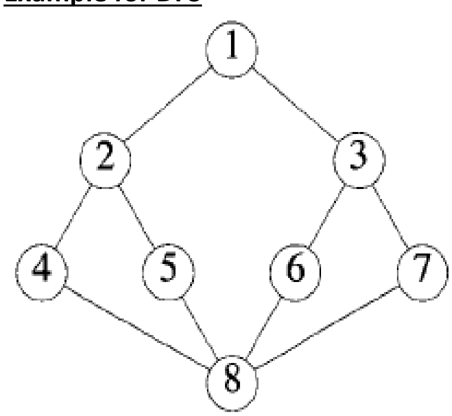
Depth First Search and Traversal

- A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached.
- At this time the exploration of the new vertex u begins.
- When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.

Algorithm for DFS

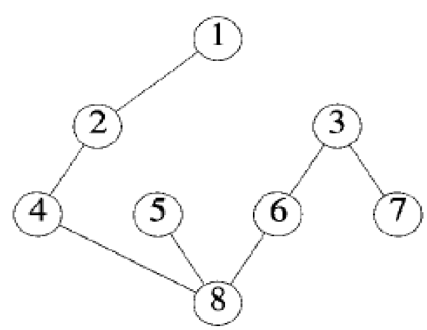
```
Algorithm DFS(v)
// Given an undirected (directed) graph G = (V, E) with
// n vertices and an array visited[ ] initially set
// to zero, this algorithm visits all vertices
// reachable from v. G and visited[ ] are global.
{
    visited[v] := 1;
    for each vertex w adjacent from v do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```

Example for DFS



DFS Order: 1, 2, 4, 8, 5, 6, 3, 7

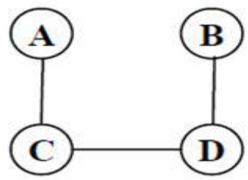
Applying DFS



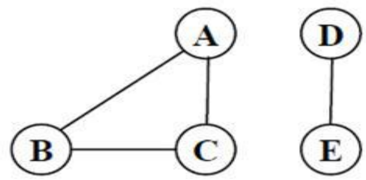
DFS

3.1.3 Connected Components and Spanning Trees

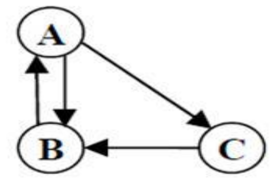
- **A graph is said to be connected if at least one path exists between every pair of vertices in the graph.**
- **Two vertices are connected component if there exists a path between them.**
- **A directed graph is said to be strongly connected if every two nodes in the graph are reachable from each other.**



Connected graph



A graph that is not connected



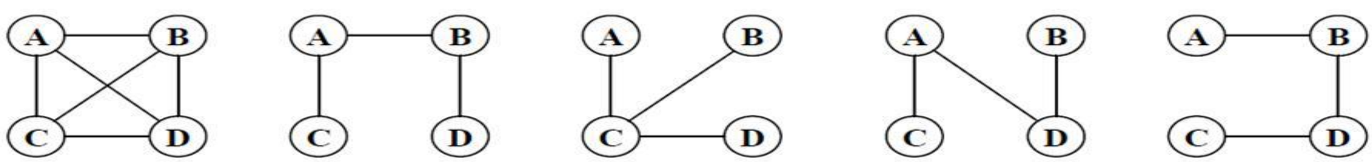
A digraph that is strongly connected

Spanning Trees

- A spanning tree of an undirected graph of n nodes is a set of n-1 edges that connects all nodes.
- A graph may have many spanning trees.
- For finding minimum cost spanning trees we have two algorithms
  1. Prim’s Algorithm
  2. Kruskal Algorithm

Properties of spanning trees:

- There is no cycle – a cycle needs n edges in an n-node graph.
- There is exactly one path between any two nodes – there is at least one path between any two nodes because all nodes are connected. Further, there is not more than one path between a pair of nodes.

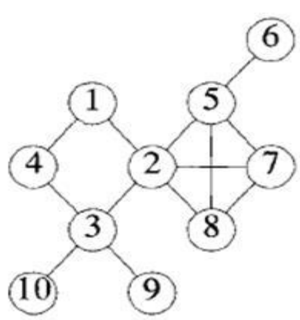


Undirected graph and four of the spanning trees of the graph

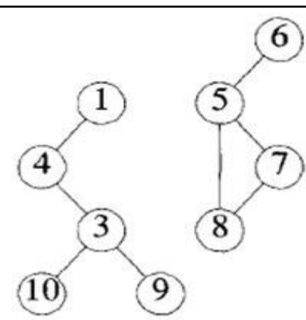
3.1.4 Bi-connected components and DFS

- A vertex v in a connected graph G is an **articulation point** if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more non empty components.
- **A graph G is bi-connected if and only if it contains no articulation points.**

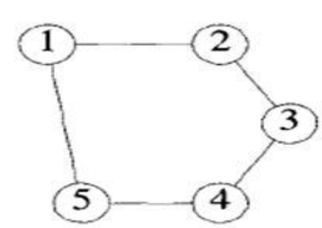
Node 2 is articulation point of Graph G



(a) Graph G



(b) Result of deleting vertex 2

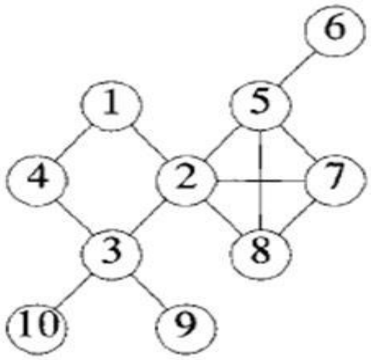


A biconnected graph

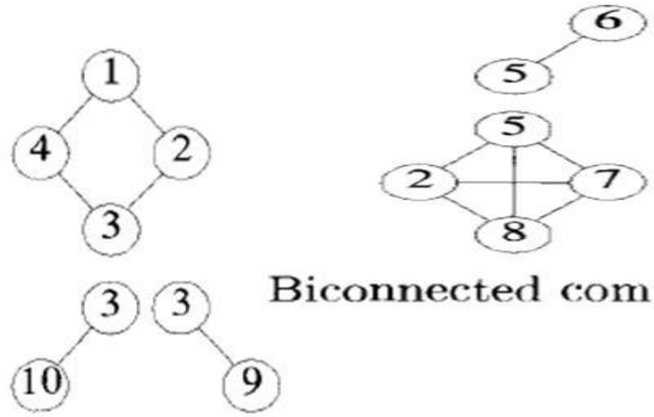
Bi Connected Component

$G' = (V', E')$ is a maximal biconnected subgraph of G if and only if G has no biconnected subgraph $G'' = (V'', E'')$ such that $V' \subseteq V''$ and $E' \subset E''$. A maximal biconnected subgraph is a *biconnected component*.

Example:



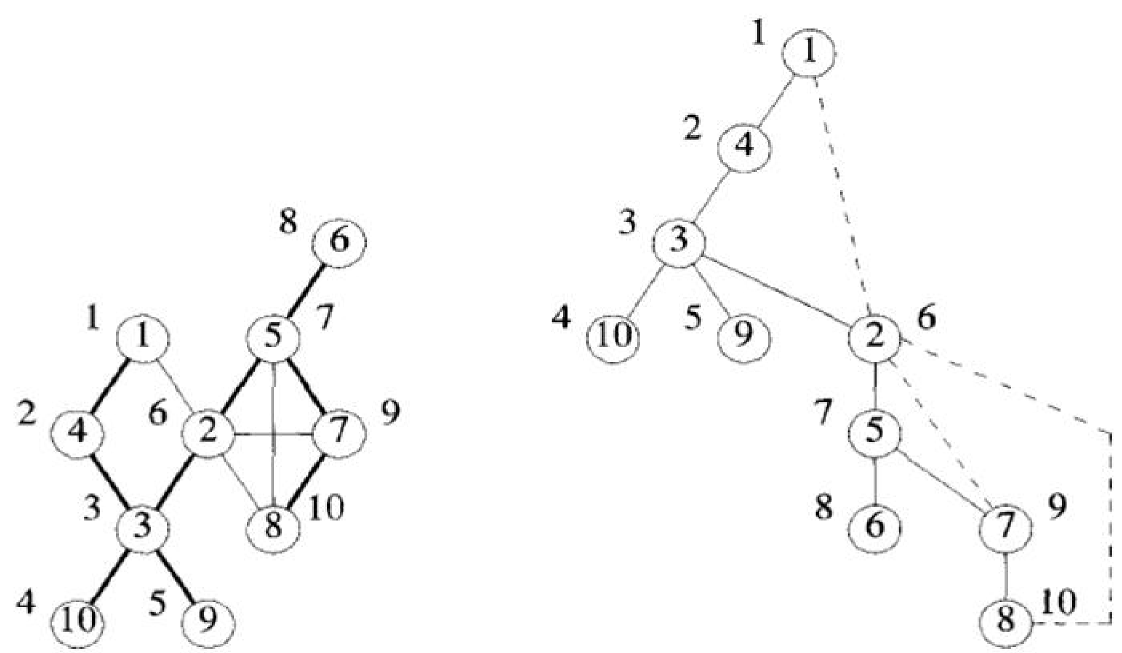
(a) Graph G



Biconnected components

DFS

Depth First Spanning Trees are used to identify articulation points and bi connected components.



(a) (b)

A depth first spanning tree of the graph

****

Unit 3.2 Topics: **Backtracking-** General Method- 8 Queens Problem- Sum of subset problem- Graph coloring- Hamiltonian Cycles- Knapsack Problem

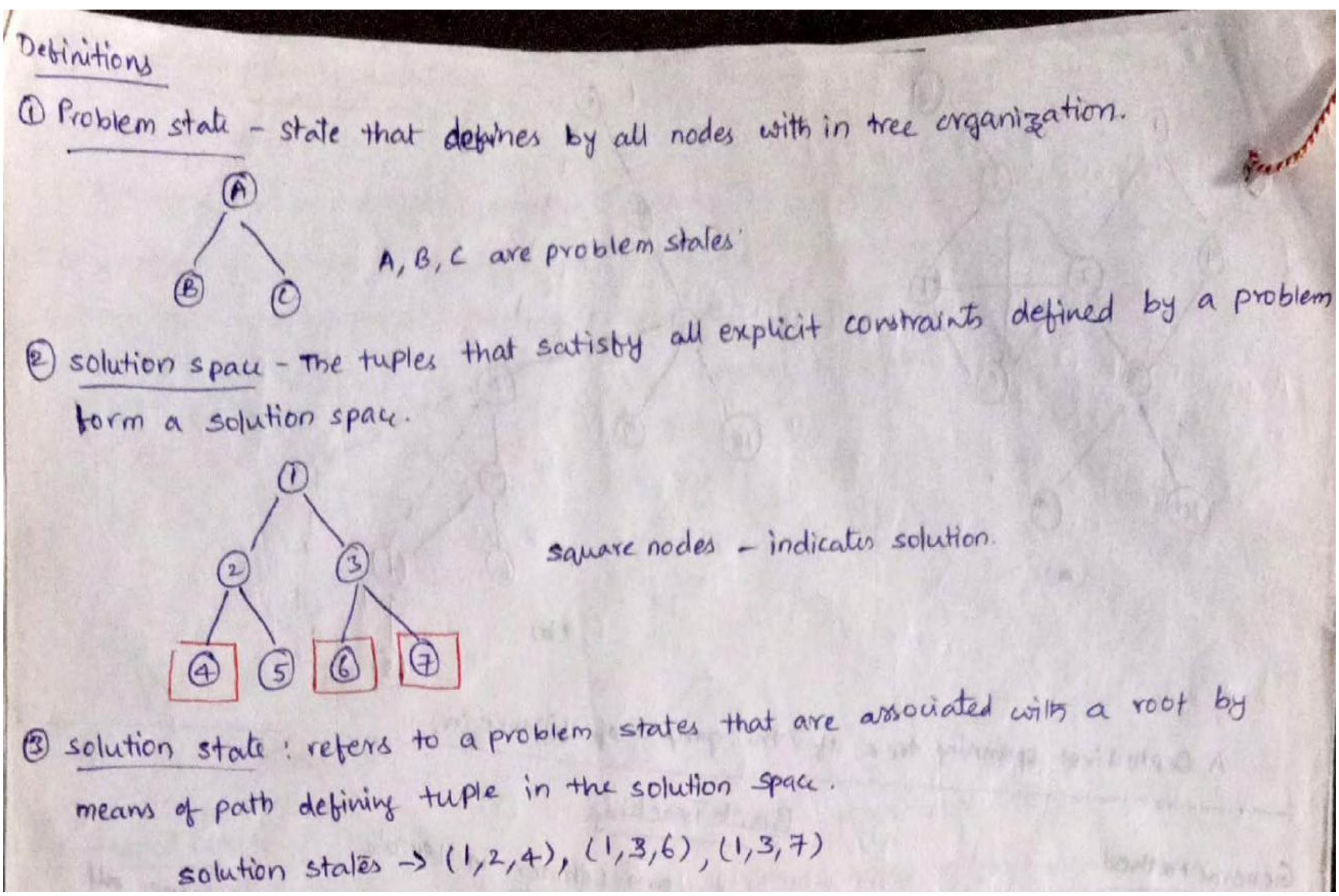
3.2.1 Backtracking- General Method:

- The name Backtrack was first coined by **D.H.Lehmer** in 1950's.
- It is a method of determining the correct solution to a problem **by examining all the available paths**.
- If a particular path leads to unsuccessful solution then its previous solution is examined in-order to final correct solution.
- In many applications of the backtrack method, the desired solution is expressible as an n-tuple (x1, x2,.. xn) where xi is chosen from some finite set Si. Often the problem to be solved calls for finding one vector that maximizes or minimizes or satisfies a **criterion function** P(x1,x2… xn)
- In **Brute force algorithm**, we consider all feasible solutions for finding **optimal solution**.
- In **Backtracking algorithm,** it is having ability to yield same answer with far fewer than m trails.
- Many of the problems, we solve using backtracking require that all solutions satisfy the complex set of **constraints.**
- Two types of Constraints

  1. **Explicit Constraints** are the rules that restrict each xi to take on values only from a given set.
  Eg: xi>=0 or Si= {all non negative real numbers}
  Xi= 0 or 1 or Si= { 0 , 1}

  2. **Implicit Constraints** are the rules that determine which of the tuples in the solution space I satisfy the criterion functions. Thus Implicit Constraints describe the way in which the xi must relate to each other.

Some Important Definitions:


Definitions

① Problem state - state that defines by all nodes with in tree organization.

A, B, C are problem states

② solution space - The tuples that satisfy all explicit constraints defined by a problem form a solution space.

square nodes - indicates solution.

③ solution state : refers to a problem states that are associated with a root by means of path defining tuple in the solution space.

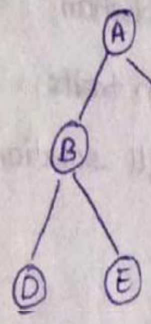
solution states -> (1,2,4), (1,3,6), (1,3,7)

④ **State space tree**: If a solution space is ~~organi~~ represented in the form of a tree then that tree is called state space tree.

⑤ **Answer states** - are those solution states ~~from the~~ for which the path from the root to define a tuple that is a member of set of solutions that satisfies implicit constraints of the Problem.

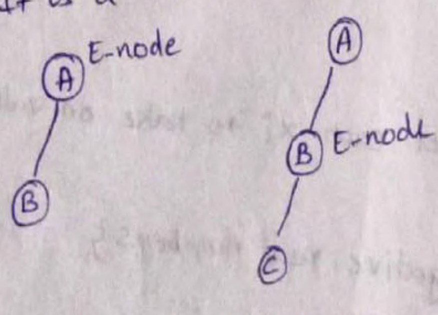
Answer states - 4, 6, 7.

⑥ **Live node**: A node which has been generated and all of whose children have not yet been generated.

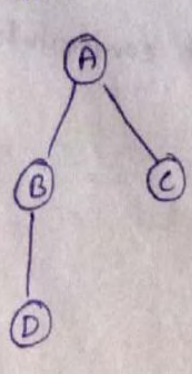


D, E, C are Live nodes - children of these nodes are yet not been generated.

⑦ **E-node**: It is a live node that can be expanded to generated its children node



⑧ **Dead node**: It is a generated node which is not ~~to~~ be expanded further or all of whose children have been generated.



A, C, D } dead node

Recursive Backtracking Algorithm:

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k − 1 values
// x[1], x[2], ..., x[k − 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], ..., x[k − 1]) do
    {
        if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if (x[1], x[2], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

Algorithm 7.1 Recursive backtracking algorithm

Iterative Backtracking Algorithm:

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1 : n] and printed
// as soon as they are determined.
{
    k := 1;
    while (k ≠ 0) do
    {
        if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
            x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
        {
            if (x[1], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            k := k + 1; // Consider the next set.
        }
        else k := k − 1; // Backtrack to the previous set.
    }
}
```

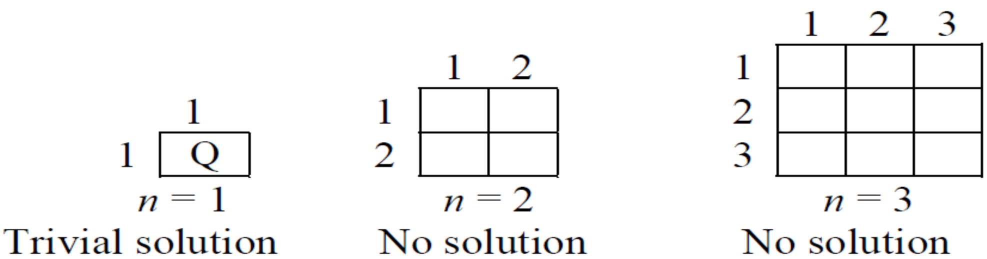
Algorithm 7.2 General iterative backtracking method

Applications of Backtracking:

- Backtracking method is applied to solve various problems like:
    1. N Queens Problem
    2. Sum of Subsets Problem
    3. Graph Coloring
    4. Hamiltonian Cycles
    5. Knapsack Problem

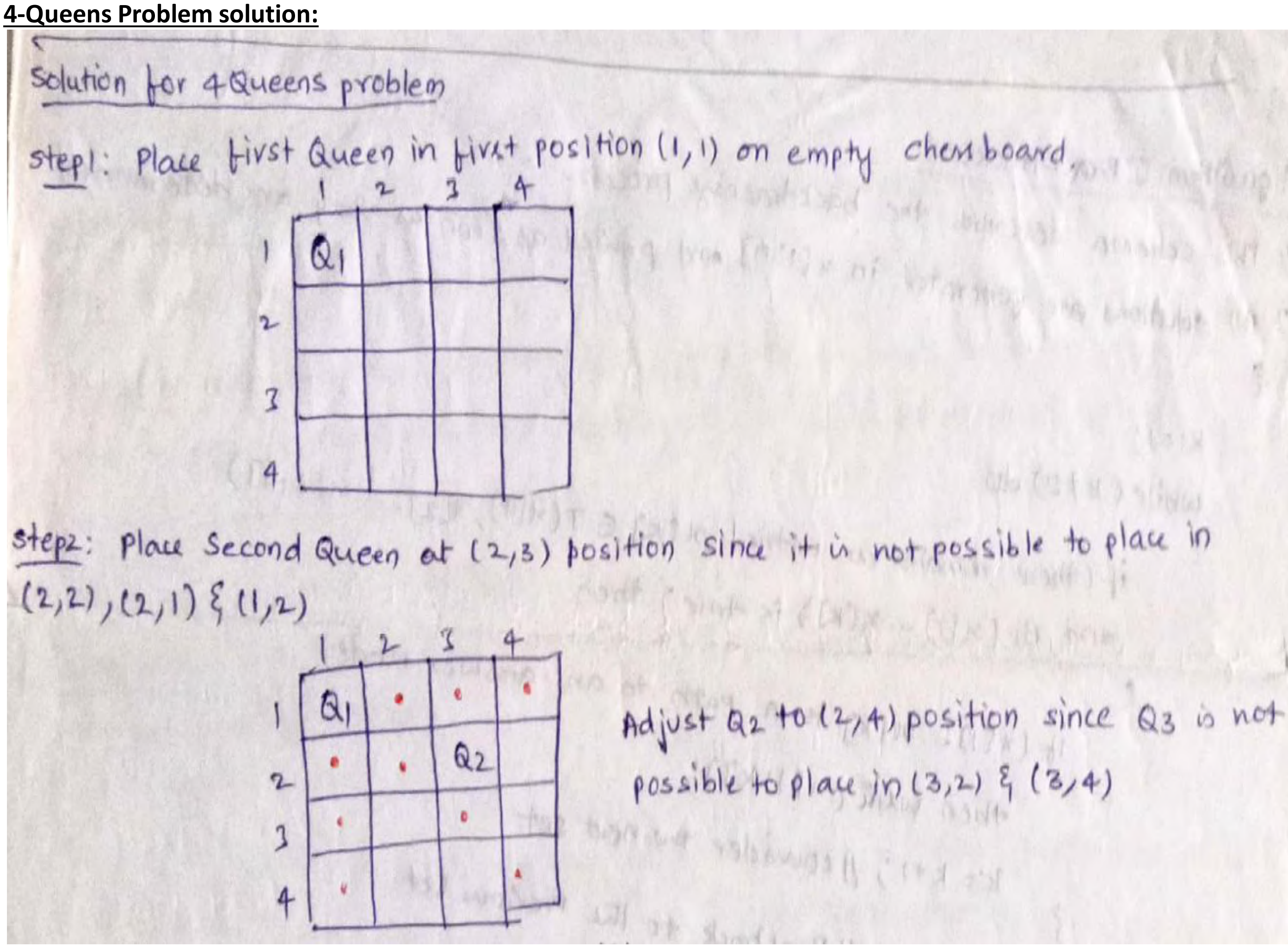
3.2.2 N Queens Problem (8 Queens Problem)

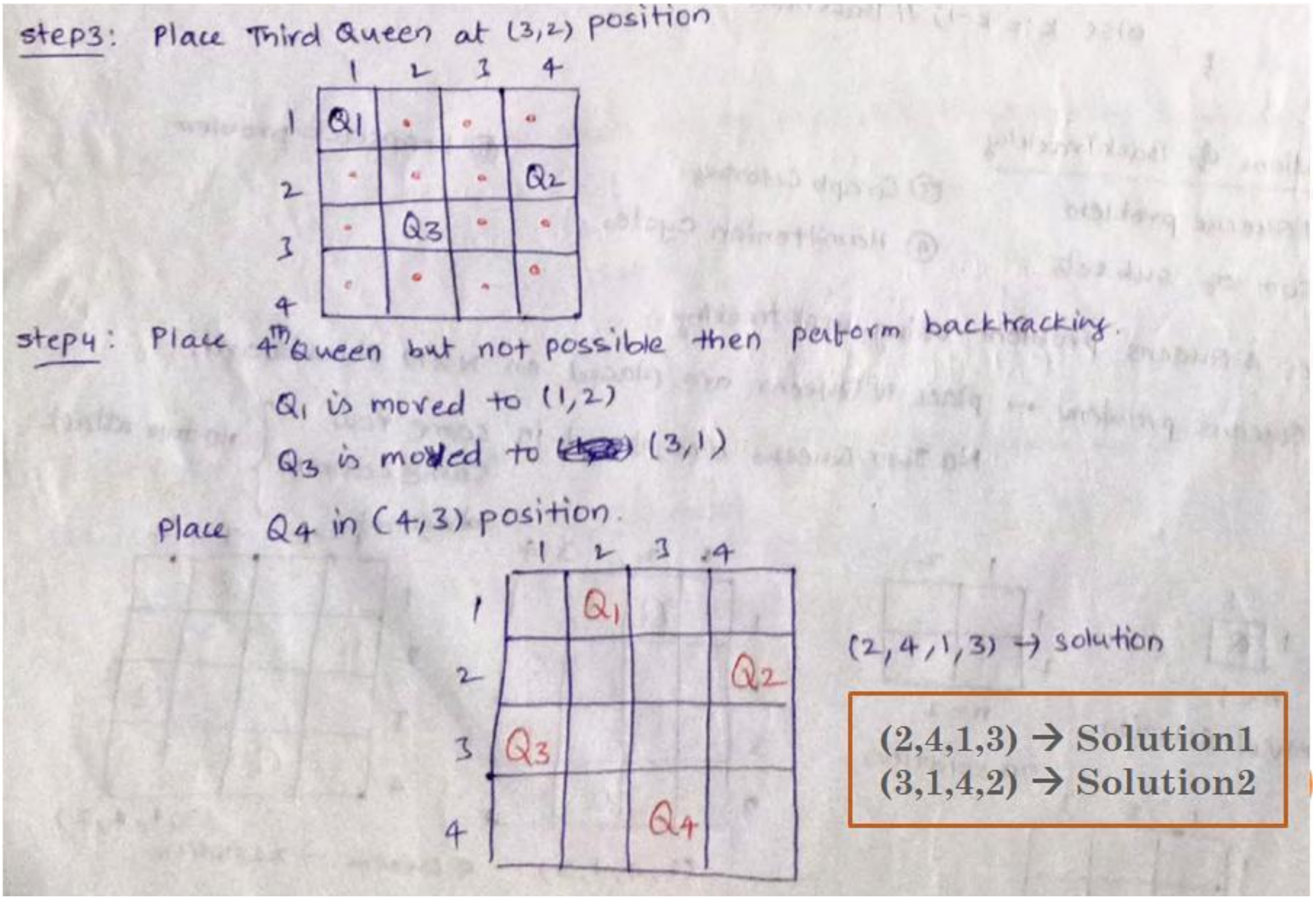
- N Queens Problems means:
    1. Place N Queens placed on N X N chess board.
    2. No Two Queens are placed in same row or same column or diagonal.
    3. No Two Queens attack to each other.



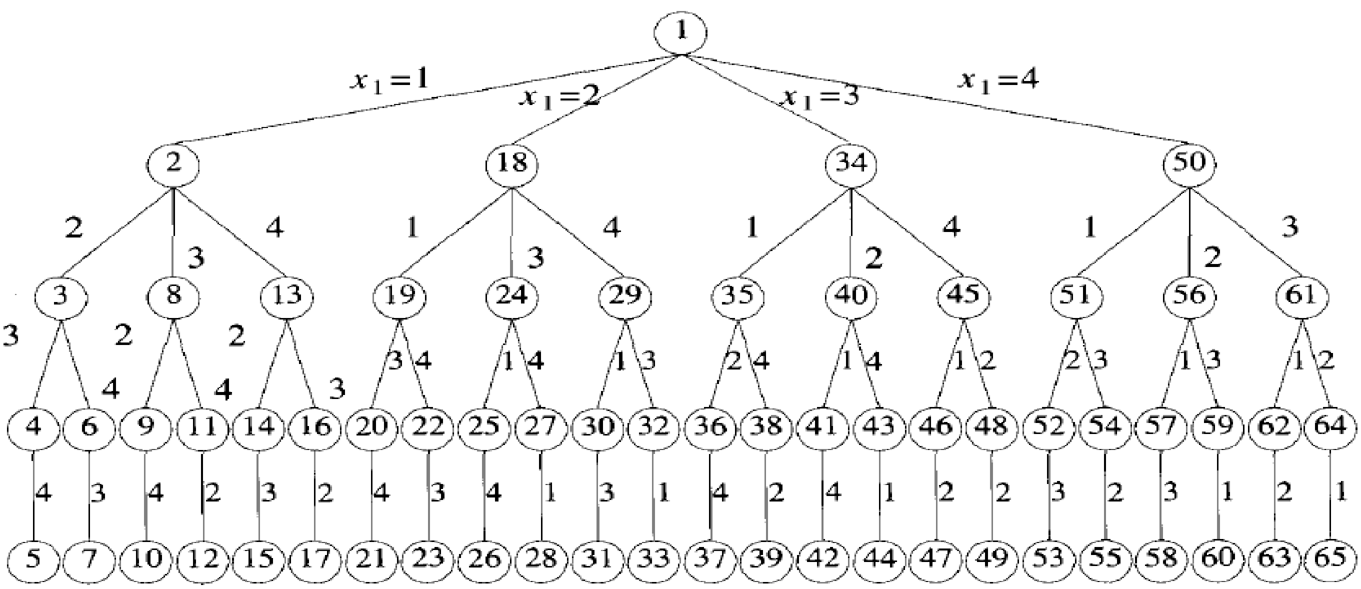
Examples of n-queens problem

4-Queens Problem solution:





4-Queens Problem –state space tree:



Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

<u>N-Queens Problem- algorithm1</u>: Placing a new queen in kth row & ith column.

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
             or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

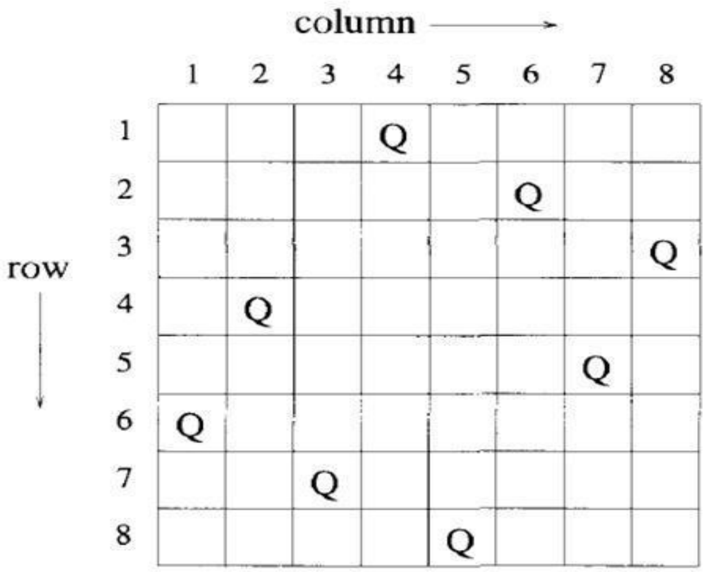
Algorithm 7.4 Can a new queen be placed?

<u>N-Queens Problem- algorithm2</u>: All solutions for N Queens Problem.

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

Algorithm 7.5 All solutions to the n-queens problem

8-Queens Problem solution:



One solution to the 8-queens problem

12 solutions

(3,6,2,7,1,4,8,5)
(2,6,8,3,1,4,7,5)
(6,3,7,2,4,8,1,5)
(3,6,8,2,4,1,7,5)
(4,8,1,3,6,2,7,5)
(7,2,6,3,1,4,8,5)
(2,6,1,7,3,8,4,5)
(1,6,8,3,7,4,2,5)
(1,5,8,6,3,7,2,4)
(2,4,6,8,3,1,7,5)
(6,3,1,8,4,2,7,5)
(4,6,8,2,7,1,3,5)

3.2.3 Sum of subset problem

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Example: n=6 total sum(m) = 30

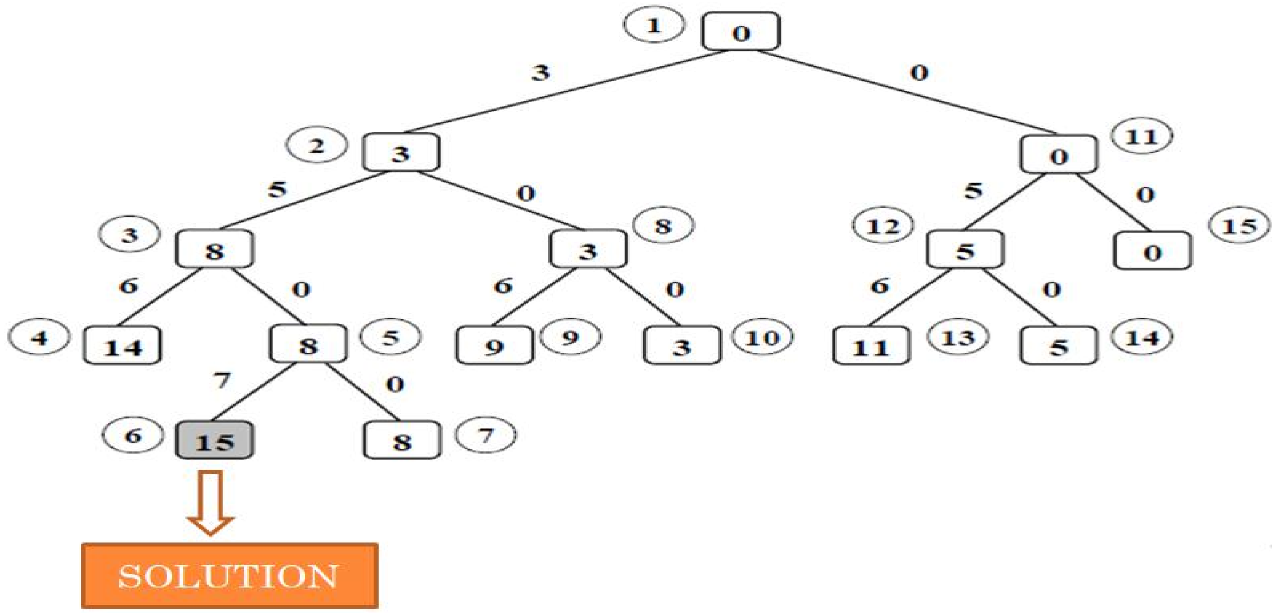
w[1:6] = {5, 10, 12, 13, 15, 18}

we have the solutions for getting sum = 30 is as follows

	x_1	x_2	x_3	x_4	x_5	x_6	
Solution1:	1	1	0	0	1	0	5+10+15 = 30
Solution2:	1	0	1	1	0	0	5+12+13 = 30
Solution3:	0	0	1	0	0	1	12+18 = 30

SUM OF SUBSETS PROBLEM-EXAMPLE

$X = \{3, 5, 6, 7\}$ and $S = 15$



Sum of Subsets Problem-Algorithm

```
Algorithm SumOfSub(s, k, r)
// Find all subsets of w[1 : n] that sum to m. The values of x[j],
// 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
    // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
    x[k] := 1;
    if (s + w[k] = m) then write (x[1 : k]); // Subset found
        // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
    else  if (s + w[k] + w[k + 1] ≤ m)
          then SumOfSub(s + w[k], k + 1, r - w[k]);
    // Generate right child and evaluate B_k.
    if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

Sum of Subsets Problem-Example

Example: $w = \{5, 10, 12, 13, 15, 18\}$ $n = 6$ $m = 30$

Output $\rightarrow$ (1, 1, 0, 0, 1) & (1, 0, 1, 1) & (0, 0, 1, 0, 0, 1) respectively.

$r = 5 + 10 + 12 + 13 + 15 + 18 = 73$ ✓

$s = 0, k = 1, r = 73$ ✓

$s + w[k] = 0 + w[1] = 0 + 5 \neq 30 \rightarrow$ Generate Left child & right child.

Left child: $\rightarrow s + w[k] + w[k+1] \leq m \Rightarrow 0 + w[1] + w[2] \leq 30$

$0 + 5 + 10 \leq 30 \Rightarrow 15 \leq 30$ T

Left child $\rightarrow s + w[k], k+1, r - w[k]$

$0 + w[1], 1+1, 73 - w[1] \Rightarrow$ 5, 2, 68

$0 + 5, 2, 73 - 5$

Right child $\rightarrow s + r - w[k] \geq m$ & $s + w[k+1] \leq m$

$0 + 73 - w[1] \geq 30$ & $0 + w[2] \leq 30$

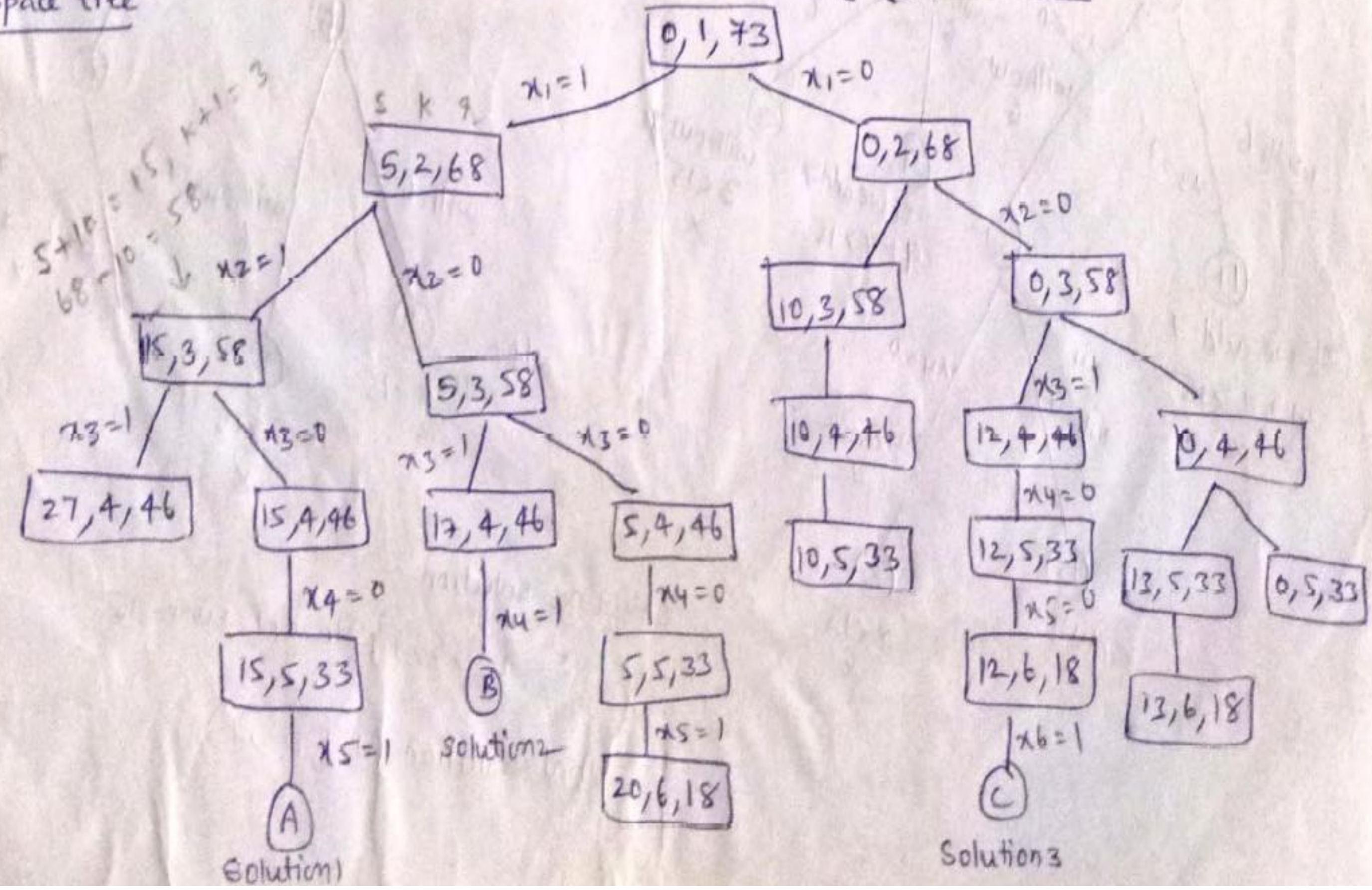
$73 - 5 \geq 30$ & $0 + 10 \leq 30 \Rightarrow 68 \geq 30$ and $10 \leq 30$

right child $\rightarrow s, k+1, r - w[k] \Rightarrow$ 0, 2, 68

$73 - w[1]$ continuing in this manner

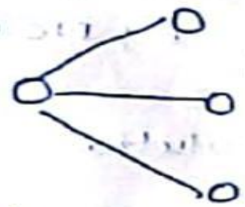
continuing in this manner

space tree



3.2.4 Graph coloring:

- Let G be a graph and m be a positive integer.
- It is to find whether that nodes of G can be **colored** in such a way that **no two adjacent nodes have the same color** yet only m colors are used where **m is a chromatic number**.
- **If d is degree of a given graph G, then it is colored with d+ 1 colors.**
- **Degree** means number of edges connected to that node.



colors can be given for this is d+1 colors.
for 4 nodes, we give 4 colors.
degree = 3

For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

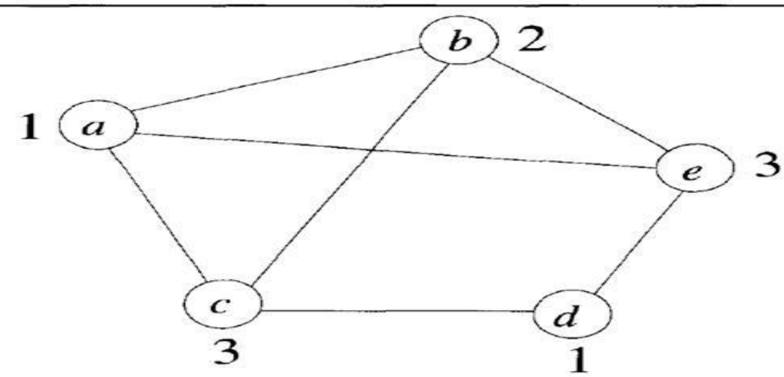


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other.

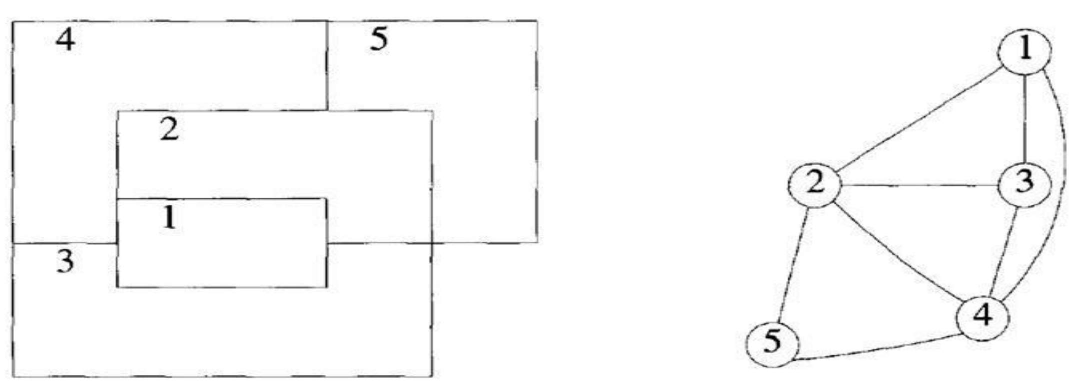


Figure 7.12 A map and its planar graph representation

Graph coloring- m coloring algorithm

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

Algorithm 7.7 Finding all m-colorings of a graph

Graph coloring- state space tree

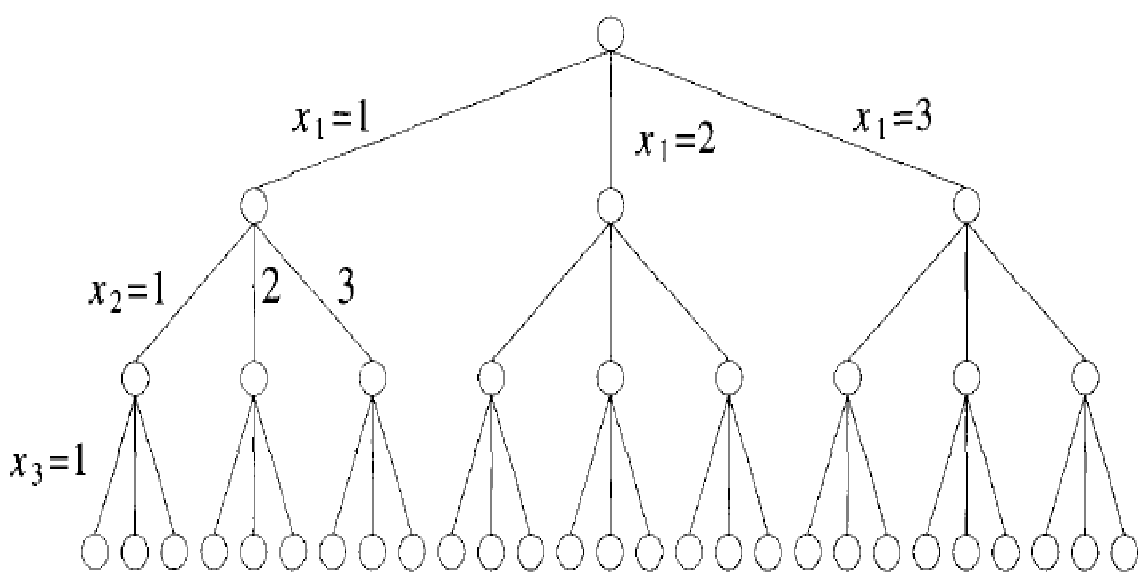


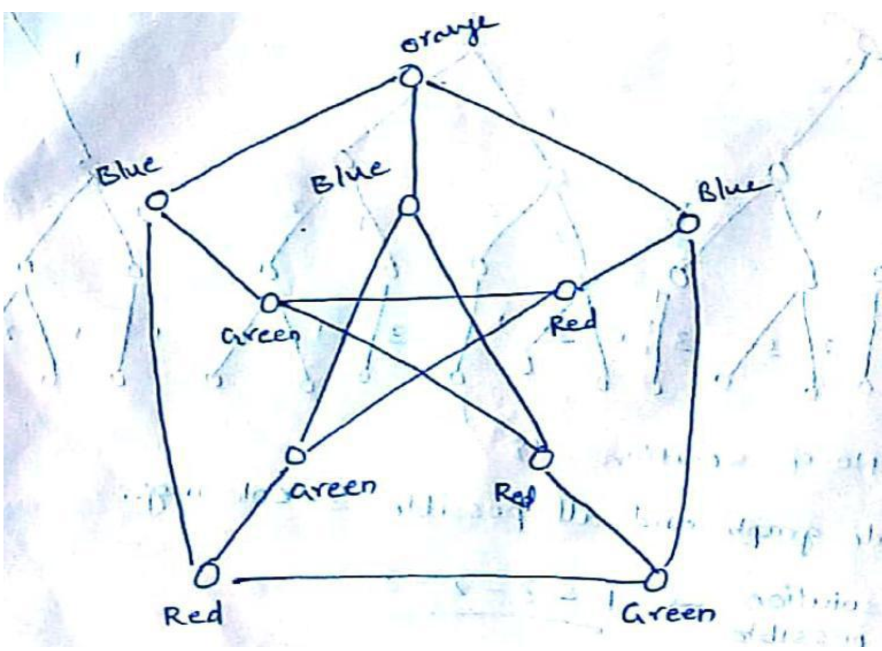
Figure 7.13 State space tree for mColoring when n =3 and m = 3

Graph coloring- generating color algorithm

```
Algorithm NextValue(k)
// x[1], . . . , x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

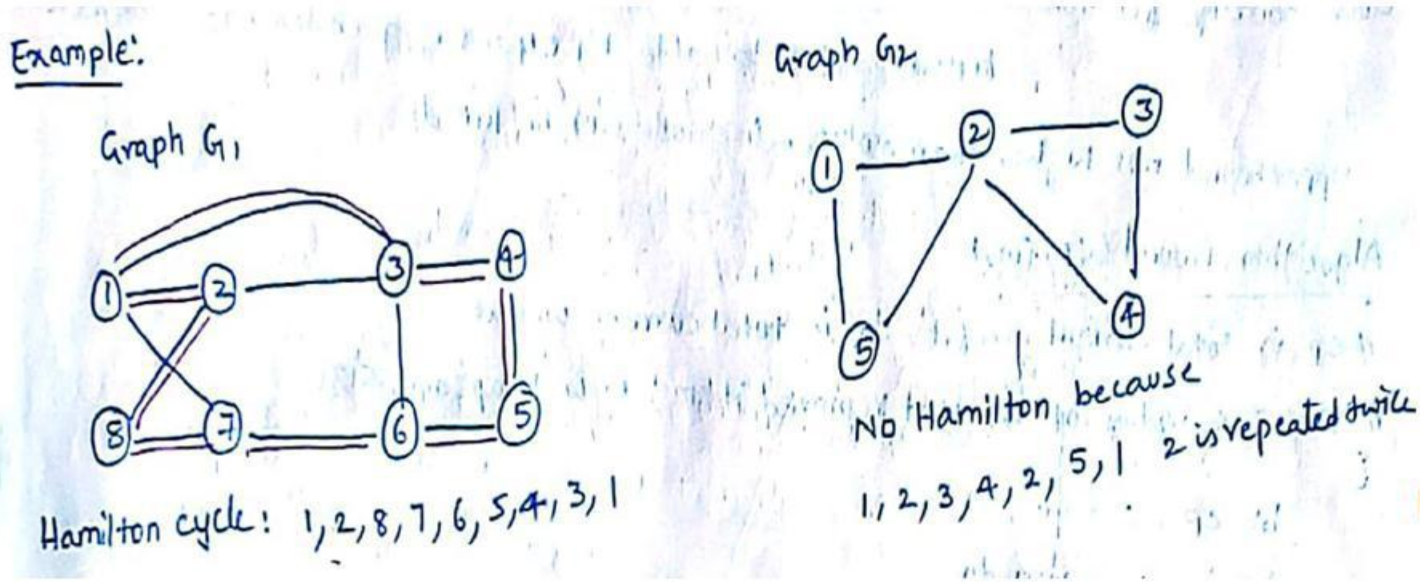
Algorithm 7.8 Generating a next color

Graph coloring- another example



3.2.5 Hamiltonian Cycles

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.



Hamiltonian Cycles-Generating a next vertex Algorithm

```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                        // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

Algorithm 7.9 Generating a next vertex

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

Algorithm 7.10 Finding all Hamiltonian cycles

3.2.6 Knapsack Problem

Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

The x_i's constitute a zero-one-valued vector.

- Knapsack Problem- Place the items/ objects in the knapsack which will have maximum profit.
- Bounding functions are needed to help for killing some live nodes without expanding them.
- Good Bounding function is obtained by using upper bound on the value of the best feasible solution obtainable by expanding given live nodes.
- Upper bound is not higher than value- live nodes can killed.

Knapsack Problem- Bounding Function Algorithm

```
Algorithm Bound(cp, cw, k)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        c := c + w[i];
        if (c < m) then b := b + p[i];
        else return b + (1 − (c − m)/w[i]) * p[i];
    }
    return b;
}
```

Algorithm 7.11 A bounding function

Backtracking Knapsack Problem- Algorithm

```
Algorithm BKnap(k, cp, cw)
// m is the size of the knapsack; n is the number of weights
// and profits. w[ ] and p[ ] are the weights and profits.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
// knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
// is not in the knapsack; else x[k] = 1.
{
    // Generate left child.
    if (cw + w[k] ≤ m) then
    {
        y[k] := 1;
        if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
        if ((cp + p[k] > fp) and (k = n)) then
        {
            fp := cp + p[k]; fw := cw + w[k];
            for j := 1 to k do x[j] := y[j];
        }
    }
    // Generate right child.
    if (Bound(cp, cw, k) ≥ fp) then
    {
        y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
        if ((cp > fp) and (k = n)) then
        {
            fp := cp; fw := cw;
            for j := 1 to k do x[j] := y[j];
        }
    }
}
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

Knapsack Problem- Example

Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: $p = \{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$.

set initially maximum profit fp = -1, cp = 0, cw = 0 (current profit & current weight)

k=1, cp=0, cw=0

step1: cw + w[k] ≤ m ⇒ 0 + w[1] < 110 ⇒ 0+1 ≤ 110 True

set y[1] = 1

k<n ⇒ 1<8 T ⇒ Bk(k+1, cp+p[k], cw+w[k]) ⇒ Bk(2, 0+11, 0+1)

Bk(2, 11, 1)

step2: k=2, cp=11, cw=1

cw + w[k] ≤ m ⇒ 1 + w[2] < 110 ⇒ 1+11 ≤ 110 True

set y[2] = 1

k<n ⇒ 2<8 T ⇒ Bk(k+1, cp+p[k], cw+w[k]) ⇒ Bk(3, 11+21, 1+11)

Bk(3, 32, 12)

step3: k=3, cp=32, cw=12

cw + w[k] ≤ m ⇒ 12 + w[3] ≤ 110 ⇒ 12+21 ≤ 110 True

set y[3] = 1

k<n ⇒ 3<8 T ⇒ Bk(k+1, cp+p[k], cw+w[k]) ⇒ Bk(4, 32+31, 12+21)

Bk(4, 63, 33)

step4: k=4, cp=63, cw=33

cw + w[k] ≤ m ⇒ 33 + 23 ≤ 110 ⇒ 56 ≤ 110 True

set y[4] = 1

k<n ⇒ 4<8 T ⇒ Bk(k+1, cp+p[k], cw+w[k]) ⇒ Bk(5, 63+33, 33+23)

Bk(5, 96, 56)

step5: k=5, cp=96, cw=56

cw + w[k] ≤ m ⇒ 56 + w[5] ≤ 110 ⇒ 56+33 ≤ 110 True

Set y[5] = 1

k<n ⇒ 5<8 T ⇒ Bk(k+1, cp+p[k], cw+w[k]) ⇒ Bk(6, 96+43, 56+33)

Bk(6, 139, 89)

step6: $k=6, cp=139, cw=89$ Bound function is called

$cw+w[k] \le m \Rightarrow 89+43 \le 110 \Rightarrow 132 \le 110$ False

$b=cp=139, \; c=cw=89, \; i=k+1=6+1=7 \Rightarrow i=7$

$c=c+w[7]=89+45=134$

$c<m \Rightarrow 134<110$ F $\Rightarrow b = b+(1-(c-m)/w[i])*p[i])$

$= 139+(1-(134-110)/45*55)$

$=139+(1-(24/45)*55) = 164.85$

$164.85 \geq -1$ True $\rightarrow$ set $y[6]=0$; $Bk(k+1, cp, cw) \Rightarrow Bk(7, 139, 89)$

step7: $k=7, cp=139, cw=89$ Bound function is called

$cw+w[k] \le m \Rightarrow 89+45 \le 110 \Rightarrow 134 \le 110$ False

$i=k+1=7+1=8 \Rightarrow i=8$

$b=cp=139, \; c=cw=89,$

$c=c+w[8]=89+55=144$

$c<m \Rightarrow 144<110$ F $\Rightarrow b=b+(1-(c-m)/w[i]*p[i])$

$=139+(1-(144-110)/55*65)$

$=139+(1-(34/55)*65) = 163.7$

$163.7 \geq -1$ True $\rightarrow$ set $y[7]=0$; $Bk(k+1, cp, cw)$

$Bk(8, 139, 89)$

steps: $k=8, cp=139, cw=89$ Bounding function

$cw+w[k] \le m \Rightarrow 89+w[8] \le 110 \Rightarrow 89+55 \le 110 \Rightarrow 144 \le 110$ False

$b=cp=139, \; c=cw=89, \; i=k+1 \Rightarrow i=9$

$9<n \Rightarrow 9<8$ False $\rightarrow$ for loop not entered.

$b=cp=139$ returned

set $y[8]=0$

$8<n \Rightarrow 8<8$ False

$cp>fp$ & $k=n \Rightarrow 139>-1$ & $8=8$ True

$fp=cp=139, \; fw=cw=89$

$x[j] = y[j] \Rightarrow x[1] = y[1] \Rightarrow x[1] = 1$

$x[2] = y[2] \Rightarrow x[2] = 1$

$x[3] = y[3] \Rightarrow x[3] = 1$

$x[4] = y[4] \Rightarrow x[4] = 1$

$x[5] = y[5] \Rightarrow x[5] = 1$

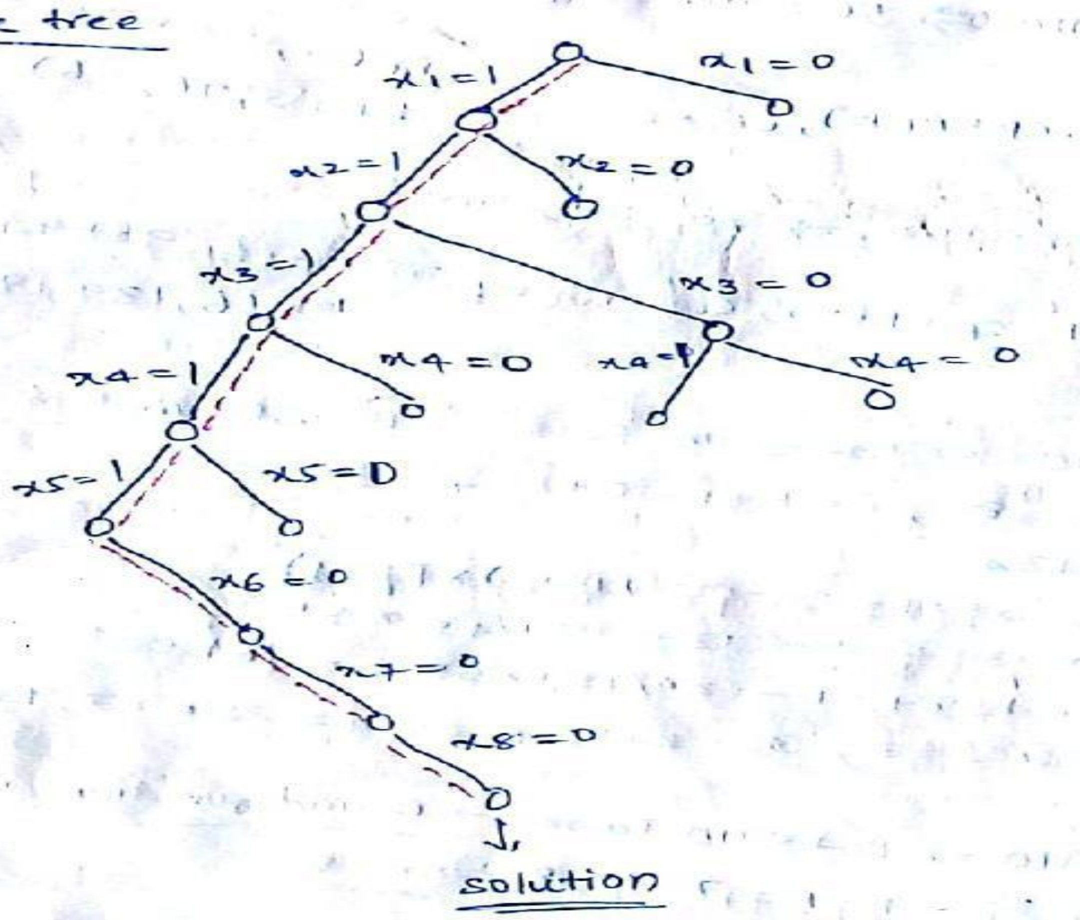
$x[6] = y[6] \Rightarrow x[6] = 0$

$x[7] = y[7] \Rightarrow x[7] = 0$

$x[8] = y[8] \Rightarrow x[8] = 0$

Final solution = (1, 1, 1, 1, 1, 0, 0, 0) having final profit 139 & final weight 89

State space tree



****

UNIT-4 DESIGN AND ANALYSIS OF ALGORITHM

Branch and Bound: The method, Travelling salesperson, 0/1 Knapsack problem.
Lower Bound Theory: Comparison trees, Lower bounds through reductions – Multiplying triangular matrices, inverting a lower triangular matrix, computing the transitive closure

4.1 Branch and Bound (B & B)- The method:

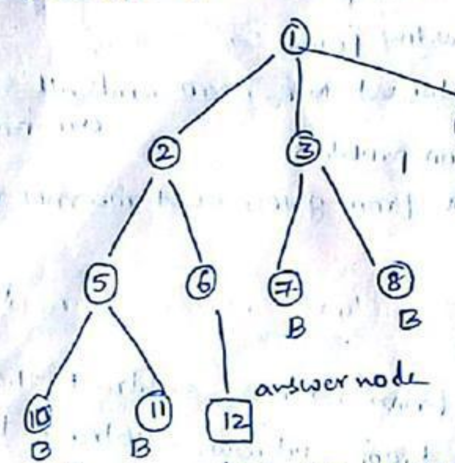
- **B & B** is a general algorithmic method for finding optimal solutions of various problems.
- In B & B, a state space tree is built and all the children of E-nodes are generated before any other node become a live node.
- **E node** is a live node that can be expanded to generate its children node.
- **Live node** is a node that can be expanded without generating its children node.
- B & B is used only for optimization problem.
- B & B needs two additional values when compared to backtracking.
    1. A bound value of objective function for every node of state space tree.
    2. Value of best solution is compared to node's bound.
- If node's bound is better than best solution node is terminated.
- **Lower bound is for minimization problems.**
- **Upper bound is for maximization problems.**
- In **Branching**, we define tree structure from set of candidates in a recursive manner.
- In **Bounding**, we calculate lower bound & upper bound of each node in the tree.
- Lower bound > Upper bound → first node is discarded from the search → **Pruning.**
- B & B is based on **advanced BFS** which is done with priority queue instead of traditional list. That means **highest priority element is always on first position**.
- Bounding functions are useful because **it doesn't allow to generate sub tree that has no answer nodes.**
- 3 types of search strategies:
    1. FIFO (First- In- First- Out) Search or BFS.
    2. LIFO (Last- In- First- Out) Search or DFS.
    3. Least Count (LC) Search.

Difference between Backtracking & Branch and Bound:

Backtracking	B&B
→ Sol is obtained using DFS method.	→ Any of the search methods DFS or BFS or best first search can be used to obtain sol.
→ It provides solution for decision problems.	→ It provides solution for optimization problems.
→ There is possibility to obtain the bad solutions.	→ No bad solutions are generated.
→ A state space tree is not searched completely instead the process of searching terminates as soon as solution is obtained.	→ State space tree using B&B is searched completely since there is possibility of obtaining an optimum.

4.1.1. FIFO (First- In- First- Out) Search or BFS:

Let us consider the state space tree



First take E-Node as node 1. We generate children of 1; we place these in a queue.

Delete 2 & generate children of 2 in queue | 2 | 3 | 4 | 5 | 6 | (struck out)

Delete 3 & generate children of 3 → but 7 & 8 are live nodes → killed by bounding function

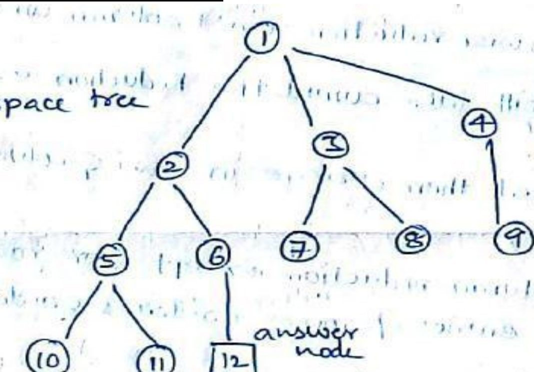
Delete 4 → killed by bounding function (4's children)

Delete 5 → killed by bounding function

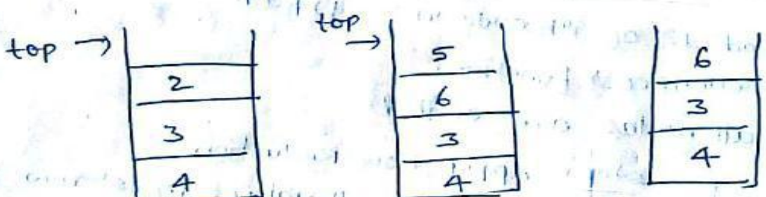
Delete 6 → child of 6 is 12 satisfies the solution of the Problem search process terminates.

4.1.2. LIFO (Last- In- First- Out) Search or DFS:

consider state space tree



First take E-node as node 1. We generate children of 1. We place these in a stack.



Remove 2 → children of 2 is placed on the top of the stack.

Remove 5 → children of 5 are 10, 11 are killed by bounding function

Remove 6 → children of 6 is 12 which is answer node

search process terminates.

4.1.3. Least Count (LC) Search:

Least Count Search control abstraction

Let t - state space tree

c() - cost function for the nodes in t.

x - node in t.

c(x) - cost of minimum cost answer node in subtree with root x.

c(t) - cost of minimum cost answer node in t.

It is easy to compute and generally has the property that if x is either an answer node or a leaf node then $c(x) = \hat{c}(x)$. $\hat{c}$ - used to find answer node.

→ This algorithm uses 2 functions:

① Least() - finds live nodes with least $\hat{c}()$. This node is deleted from the list of livenodes and returned.

② Add(x) - adds the new live node x to list of live nodes.

```
listnode = record {
    listnode *next, *parent; float cost;
}

Algorithm LCSearch(t)
// Search t for an answer node.
{
    if *t is an answer node then output *t and return;
    E := t; // E-node.
    Initialize the list of live nodes to be empty;
    repeat
    {
        for each child x of E do
        {
            if x is an answer node then output the path
                from x to t and return;
            Add(x); // x is a new live node.
            (x → parent) := E; // Pointer for path to root.
        }
        if there are no more live nodes then
        {
            write ("No answer node"); return;
        }
        E := Least();
    } until (false);
}
```

Algorithm 8.1 LC-search

4.2 Travelling Sales Person using B & B

Travelling Sales Person Problem using B&B

Def: Find the tour of minimum cost starting from a node s going to other nodes only once and returning to the starting point s.

Procedure for solving travelling sales person problem

Step1: Find out the reduced cost matrix from given cost matrix. This can be obtained as follows:

① Row Reduction

② column Reduction

Row Reduction: Take minimum element from 1st row, subtract that element from 1st row, next take minimum element from 2nd row, subtract that element from 2nd row ... apply this procedure for all rows.

column Reduction: Take minimum element from 1st column, subtract that element from 1st column ... apply same process for all columns.

Now we will find row wise reduction sum & column wise reduction sum.

Row wise Reduction sum = sum of elements which subtracted from rows

column wise Reduction sum = sum of elements which subtracted from columns.

cumulative Reduction = row wise reduction sum + column wise reduction sum

step2: For starting node, we will take cumulative Reduction as lower bound and ∞ as upper bound.

a) if path (i, j) is considered then change in row i & column j of A to ∞.

b) set A[j, 1) to ∞.

c) Apply row reduction & column reduction except for rows and columns containing to ∞ [i.e all entries of row or contains contains ∞]

Also find cumulative reduction (r).

d) r is calculated in step (c) so

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

$\hat{c}(R)$ – lowest bound (L) of ith node in (i, j) path.

$\hat{c}(S)$ – ranking function (cost function)

Repeat step2 until all nodes are visited.

TSP Example:

Example:

Given cost matrix is

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

step1: Apply Row Reduction

Deduct 10 from all values in 1st row

Deduct 2 from 2nd row

Deduct 2 from 3rd row

Deduct 3 from 4th row

Deduct 4 from 5th row

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row wise Reduction sum = 10+2+2+3+4 = 21

Apply column reduction

Deduct 1 from 1st column

Deduct 0 from 2nd column

Deduct 3 from 3rd column

Deduct 0 from 4th column

Deduct 0 from 5th column

Reduced cost matrix

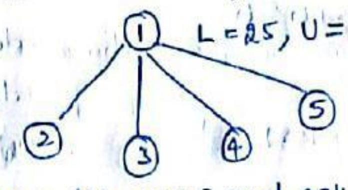
$$A = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

After Reducing Rows & Columns

column wise reduction sum = 1+0+3+0+0 = 4

Cumulative Reduction = Row wise reduction sum + column wise reduction sum = 21+4 = 25.

Step2: starting with vertex (node) is 1, next he can visit 2nd, 3rd, 4th or 5th.



(*) select A(1,2) and change 1st row & 2nd columns are ∞ $\boxed{A(1,2) = 10}$

set $A(2,1)=\infty$. The Resultant Matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row reduction & column reduction → but here should not apply for rows and columns contains all entries as ∞.

row reduction sum $= 0+0+0+0 = 0$

column reduction sum $= 0+0+0+0 = 0$

cumulative reduction (γ) = row wise + column wise $= 0+0 = 0$

$\hat{c}(S) = \hat{c}(R) + A(1,2) + \gamma$

$= 25 + 10 + 0 = 35 \Rightarrow \boxed{\hat{c}(S) = 35}$

similarly we apply for A(1,3) $\boxed{A(1,3) = 17}$

(*) select A(1,3) and change 1st row & 3rd columns are ∞

set $A(3,1) = \infty$. The resultant matrix is

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

Row reduction

deduct 0 from 2nd row
deduct 0 from 3rd row
deduct 0 from 4th row
deduct 0 from 5th row

column reduction

deduct 11 from 1st col
deduct 0 from 2nd col
deduct 0 from 4th col
deduct 0 from 5th col

Apply Row reduction & column reduction

row reduction sum $= 0+0+0+0 = 0$

col reduction sum $= 11+0+0+0 = 11$

Cumulative reduction (γ) $= 0 + 11 = 11$

$\hat{c}(S) = \hat{c}(R) + A(1,3) + \gamma$

$= 25 + 17 + 11 = 53$

$\boxed{\hat{c}(S) = 53}$

(A) Select A(1,4) and change 1st row & 4th columns are ∞

set A(4,1) = ∞ . The resultant matrix is

$A(1,4) = 0$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Row Reduction

deduce 0 from 2nd row

deduce 0 from 3rd row

deduce 0 from 4th row

deduce 0 from 5th row

col reduction

deduce 0 from 1st col

deduce 0 from 2nd col

deduce 0 from 3rd col

deduce 0 from 5th col

Apply Row wise & column wise reduction

row reduction sum = 0+0+0+0 = 0

col reduction sum = 0+0+0+0 = 0

Cumulative reduction (r) = 0

$\hat{c}(S) = \hat{c}(R) + A(1,4) + r$

$= 25 + 0 + 0 = 25$

$\hat{c}(S) = 25$

(A) Select A(1,5) and change 1st row & 5th columns are ∞.

set A(5,1) = ∞ . The resultant matrix is

$A(1,5) = 1$

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

Row reduction

deduce 2 from 2nd row

deduce 0 from 3rd row

deduce 3 from 4th row

deduce 0 from 5th row

col reduction

deduce 0 from 1st col

deduce 0 from 2nd col

deduce 0 from 3rd col

deduce 0 from 4th col

Apply Row wise & column wise reduction

row reduction sum = 2+0+3+0 = 5

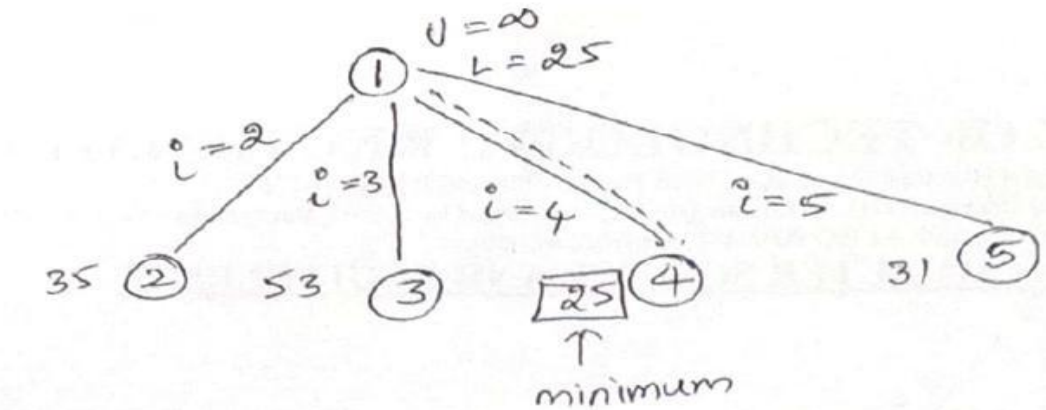
col reduction sum = 0+0+0+0 = 0

Cumulative reduction (r) = 5+0 = 5

$\hat{c}(S) = \hat{c}(R) + A(1,5) + r$

$= 25 + 1 + 5 = 31$

$\hat{c}(S) = 31$



Select 1 → 4 with minimum value = 25

$$\text{Reduced Matrix} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Consider Now
4→2, 4→3, 4→5

Consider the path A(4,2) and change 4th row & 2nd column are ∞. set A(2,1) = ∞. The resultant matrix is

$$A(4,2) = 3$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

row reduction sum = 0+0+0 = 0
col reduction sum = 0+0+0 = 0

Apply Raw reduction & column reduction

cummulative reduction (r) = 0 + 0 = 0

$$\hat{c}(S) = \hat{c}(R) + A(4,2) + r$$

$$= 25 + 3 + 0$$

$$\boxed{\hat{c}(S) = 28}$$

consider the path A(4,3) and change 4th row & 3rd column are ∞.

set A(3,1) = ∞. The resultant matrix is **A(4,3) = 12**

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

row reduction sum = 0+2+0 = 2

col reduction sum = 11+0+0 = 11

reduced Matrix

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply Row reduction & column reduction

cummulative reduction (r) = 2+11 = 13

$\hat{c}(S) = \hat{c}(R) + A(4,3) + r$

$= 25 + 13 + 12 = 50$

$\hat{c}(S) = 50$

consider the path A(4,5) and change 4th row & 5th column are ∞.

set A(5,1) = ∞. The resultant matrix is **A(4,5) = 0**

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

row reduction sum = 11+0+0 = 11

col reduction sum = 0+0+0 = 0

Reduced Matrix

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

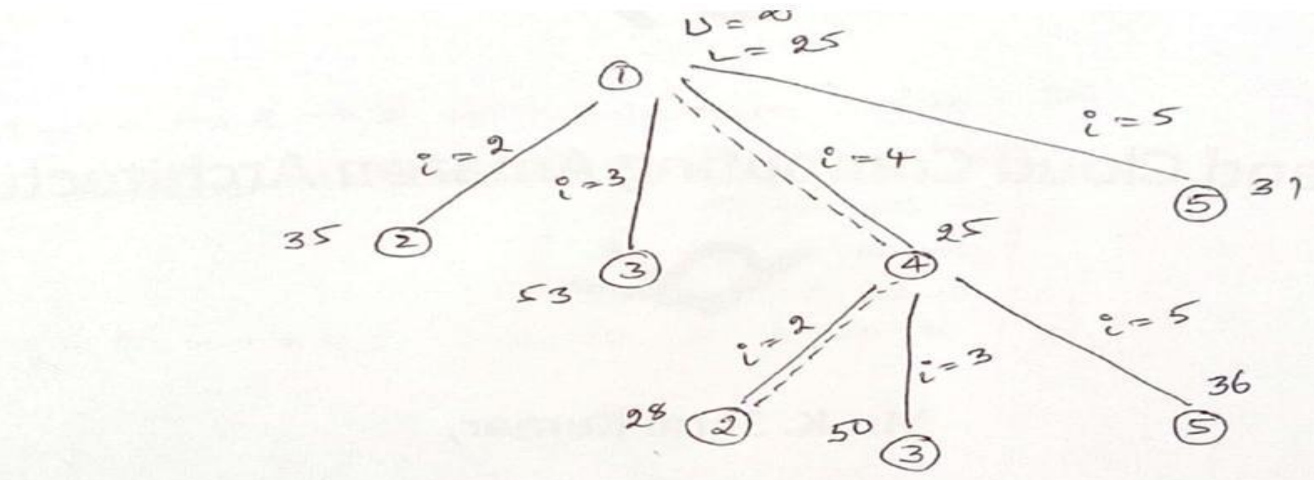
Apply Row reduction & column reduction

cummulative reduction (r) = 11+0 = 11

$\hat{c}(S) = \hat{c}(R) + A(4,5) + r$

$= 25 + 0 + 11 = 36$

$\hat{c}(S) = 36$



select 1 → 4 → 2 with minimum value = 28

Reduced Matrix =

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Consider Now
2→3, 2→5

consider path A(2,3) and change 2nd row & 3rd columns are ∞.

set A(3,1) = ∞. The resultant matrix is

$A(2,3) = 11$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

row reduction sum = 2 + 0 = 2

col reduction sum = 11 + 0 = 11

Apply Row reduction & column reduction

Cummulative reduction (r) = 2 + 11 = 13

$$\hat{c}(S) = \hat{c}(R) + A(2,3) + r$$

$$= 28 + 11 + 13 = 52$$

$\hat{c}(S) = 52$

Reduced matrix

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

consider path A(2,5) and change 2nd row & 5th columns are ∞.

Set A(5,1) = ∞. The resultant matrix is

A(2,5) = 0

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

row reduction sum = 0

column reduction sum = 0

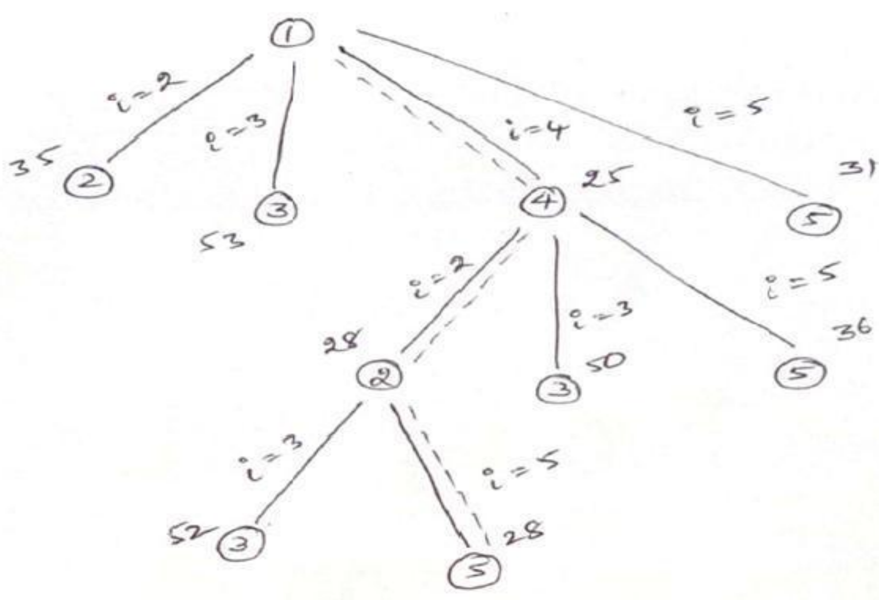
Apply Row reduction & column reduction

cummulative reduction (r) = 0 + 0 = 0

$$\hat{c}(S) = \hat{c}(R) + A(2,5) + r$$

$$= 28 + 0 + 0$$

$$\boxed{\hat{c}(S) = 28}$$



select 1 → 4 → 2 → 5 with minimum value = 28

Reduced Matrix = $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Consider Now
5→3

5th row & 3rd columns are ∞.

consider path A(5,3) change 5th row & 3rd columns are ∞. $A(5,3)=0$

set $A(3,1)=\infty$. The resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

row reduction sum = 0

col reduction sum = 0

Apply Row reduction & column reduction

cumulative reduction $(r) = 0 + 0 = 0$

$\hat{C}(S) = \hat{C}(R) + A(5,3) + r$

$= 28 + 0 + 0 = 28$

$\hat{C}(S) = 28$

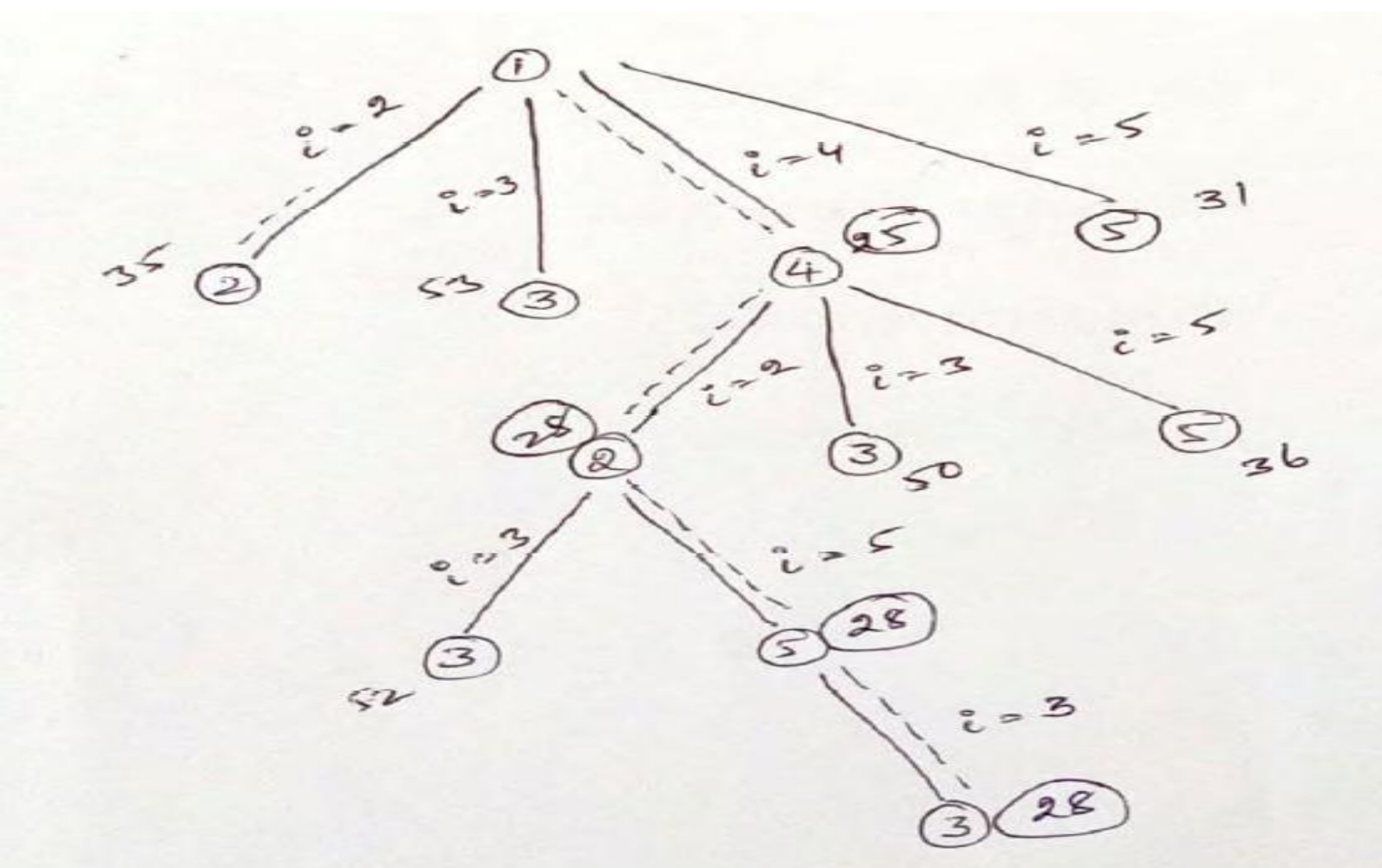
Initial cost Matrix Given

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

The path of travelling salesman problem is

$1 \to 4 \to 2 \to 5 \to 3 \to 1$

Minimum cost $= 10 + 6 + 2 + 7 + 3 = 28$ //



4.3 0/1 Knapsack problem using B & B

0/1 knapsack problem using Branch & Bound

Place the items in bag & get maximum profit → should not exceed knapsack capacity.

$M = 15$, $n = 4$ $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$

$(W_1, W_2, W_3, W_4) = (2, 4, 6, 9)$

→ In this problem, we will calculate lower bound & upper bound for each node.

Place 1st item → remaining weight $= 15 - 2 = 13$

Place 2nd item → remaining weight $= 13 - 4 = 9$

Place 3rd item → remaining weight $= 9 - 6 = 3$

We can't place 4th item → knapsack capacity exceeds.

Profit $= P_1 + P_2 + P_3 = 10 + 10 + 12 = 32$

Upperbound $= 32$

To calculate lower bound → we place W_4 in bag since fractions are allowed.

To calculate upper bound → we can't place W_4 → since fractions are not allowed.

lowerbound $= 10 + 10 + 12 + \left(\frac{3}{9} \times 18\right)$ (3 → remaining weight, 9 → weight of W_4, 18 → Profit of P_4)

lowerbound $= 32 + 6 = 38$

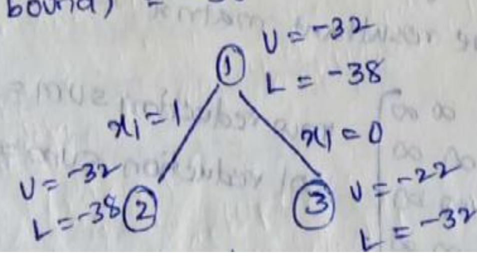
knapsack problem → maximization problem

B&B → applied for only minimization problem

To convert maximization problem into minimization problem we have take negative sign for upper bound & lower bound.

U (upperbound) $= -32$

L (Lower bound) $= -38$



calculating upper bounds & lower bounds

Node 2: $x_1 = 1 \rightarrow$ include first item in bag

$U = 10+10+12 = 32$

$U = -32$

$L = 10+10+12+\left(\frac{3}{9}\times 18\right) = 38$

$L = -38$

Node 3: $x_1 = 0 \rightarrow$ not include 1st item

$U = 10+12 = 22$

$U = -22$

$L = 10+12+\left(\frac{5}{9}\times 18\right) = 32$

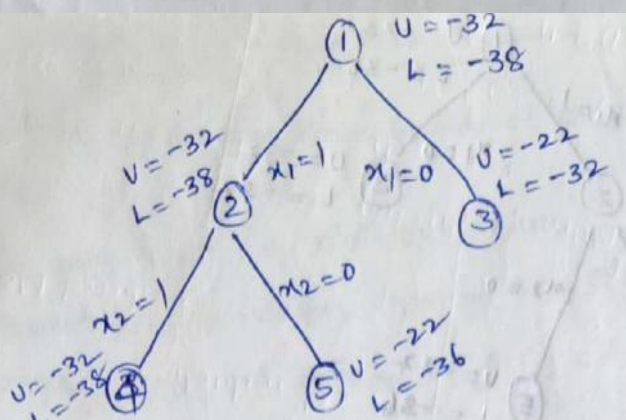
$L = -32$

Next, we calculate difference of upper bound & lower bound for nodes 2, 3

for node 2 $\rightarrow$ $U - L = -32+38 = 6$

for node 3 $\rightarrow$ $U - L = -22+32 = 10$

choose node 2 since it has minimum difference value 6.



Calculating upper bounds & lower bounds

Node 4: $x_2 = 1 \rightarrow$ Include 2nd item

$U = 10+10+12 = 32$

$U = -32$

$L = 10+10+12+\left(\frac{3}{9}\times 18\right) = 38$

$L = -38$

Node 5: $x_2 = 0 \rightarrow$ do not include 2nd item

$U = 10+12 = 22$

$U = -22$

$L = 10+12+\left(\frac{7}{9}\times 18\right) = 36$

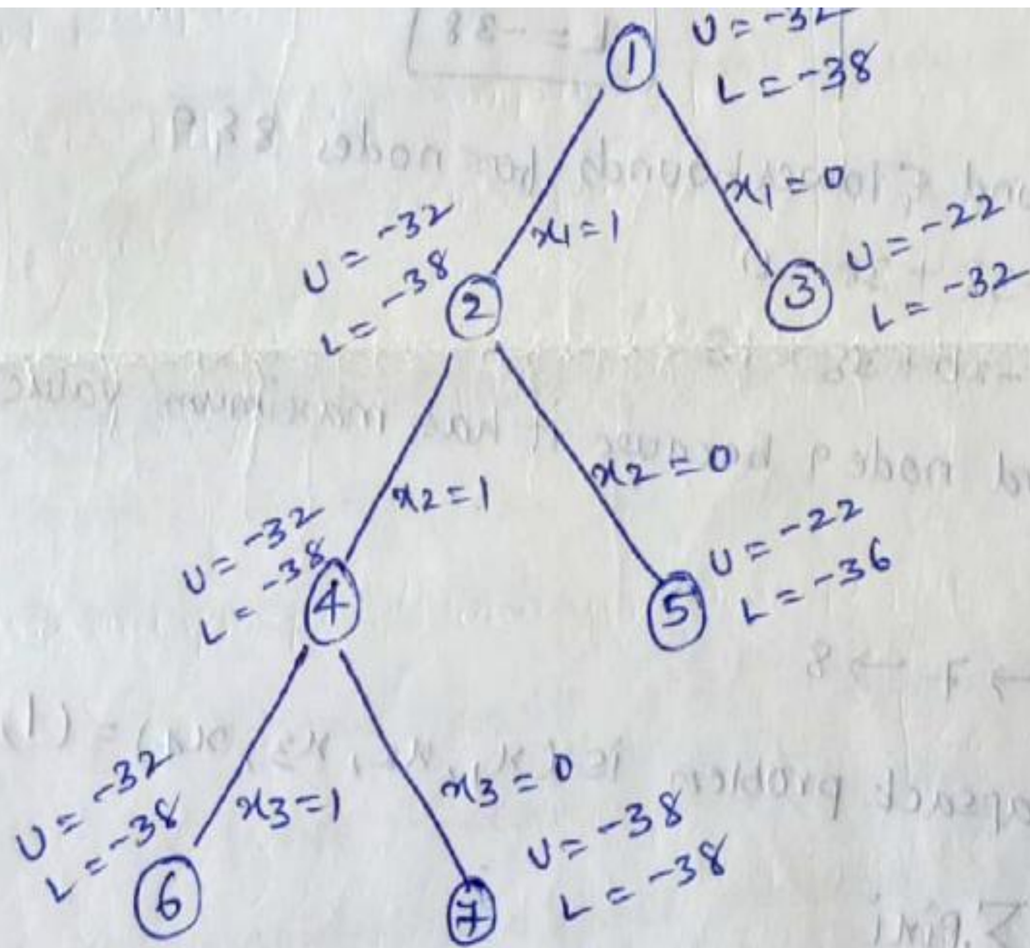
$L = -36$

Next, we calculate difference of upper bound & lower bound for nodes 4, 5

for node 4 $\rightarrow$ $U - L = -32+38 = 6$

for node 5 $\rightarrow$ $U - L = -22+36 = 14$

choose node 4 since it has minimum difference value 6.



calculating upper bounds & lower bounds

Node 6: $x_3 = 1 \rightarrow$ include 3rd item

$U = 10 + 10 + 12 = 32$

$U = -32$

$L = 10 + 10 + 12 + \left(\frac{3}{9} \times 18\right) = 38$

$L = -38$

Node 7: $x_3 = 0 \rightarrow$ do not include 3rd item

$U = 10 + 10 + 18 = 38$

$U = -38$

$L = 10 + 10 + 18 + \left(\frac{0}{9} \times 18\right) = 38$

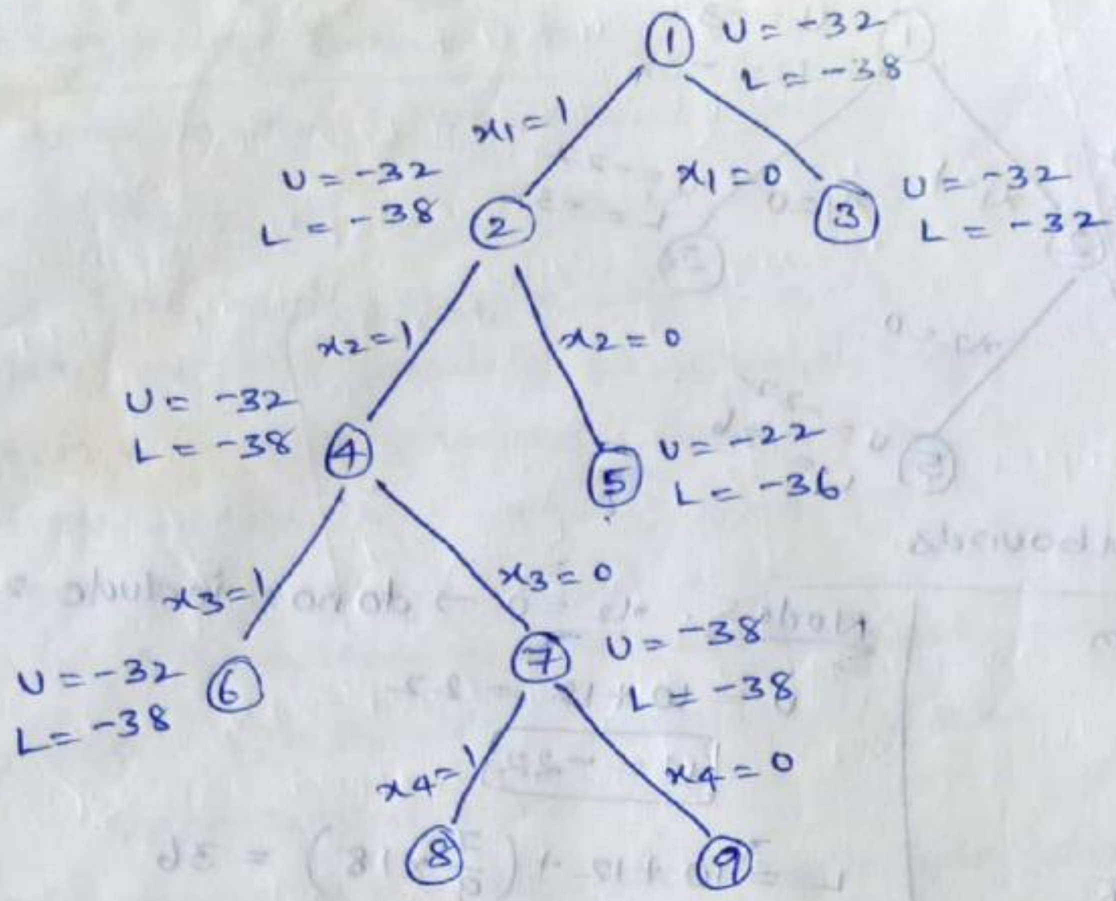
$L = -38$

Next, we calculate difference of upper bound & lower bound for nodes 6, 7.

for node 6 $\rightarrow U - L = -32 + 38 = 6$

for node 7 $\rightarrow U - L = -38 + 38 = 0$

choose node 7 since it has minimum difference value 0.



calculating upperbounds & lower bounds

Node 8: $x_4 = 1$ → include 4th item

$U = 10 + 10 + 18 = 38$

$U = -38$

$L = 10 + 10 + 18 + \left(\frac{0}{9} \times 18\right) = 38$

$L = -38$

Node 9: $x_4 = 0$ → do not include 4th item

$U = 10 + 10 = 20$

$U = -20$

$L = 10 + 10 + \left(\frac{9}{9} \times 18\right) = 38$

$L = -38$

Next we calculate upperbound & lowerbounds for nodes 8 & 9

for node 8 → $U - L = -38 + 38 = 0$

for node 9 → $U - L = -20 + 38 = 18$

choose node 8 → discard node 9 because it has maximum value.

consider the path

$1 \to 2 \to 4 \to 7 \to 8$

The solution for 0/1 knapsack problem is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

Maximum profit $= \sum p_i x_i$

$= p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4$

$= 10\times1 + 10\times1 + 12\times0 + 18\times1$

$= 10 + 10 + 0 + 18 = 38$

Maximum profit = 38

4.4 Lower Bound Theory - Comparison trees

We use comparison trees for deriving lower bounds on problem that collectively called sorting & searching.

Sorting problem – We have a set S of n distinct values

Permutations of integers 1 to n → P(1), P(2) ... P(n)

storage in A[1:n] satisfy A[P(1)] < A[P(2)] < ... A[P(n)]

ordered Searching problem – whether a given element $x \in S$ occurs in A[1:n] that are ordered so that A[1] < ... A[n]

if x is in A[1:n] determine position i between 1 to x such that A[i] = x.

Merging Problem - two ordered sets of distinct inputs from S are $A[1:m]$, $B[1:n]$

$A[1] < A[2] \ldots A[m]$ $\quad$ $B[1] < B[2] \ldots B[m]$

these m+n values are to be rearranged in an Array $C[1:m+n]$

$C[1] < C[2] \ldots C[m+n]$.

These problems can be solved by making comparisons b/w elements. These algorithms used here is called <u>comparison based algorithms.</u>

① <u>ordered sorting</u>

We consider comparison based algorithms in which every comparision b/w two elements of S of the type "<u>compare x and A[i]</u>"

There are 3 possible outcomes of this comparison

$x < A[i]$ → left branch

$x = A[i]$ → algorithm terminates

$x > A[i]$ → right branch

comparing x & each element in A[i] → algorithm used.

no i is found → search unsuccessful.

Let $A[1:n]$, $n \geq 1$ contain n distinct elements, ordered so that $A[1] < \ldots A[n]$

Let FIND(n) be the minimum no. of comparisons needed, in the worst case, by any comparison-based algorithm to recognize whether $x \in A[1:n]$. Then

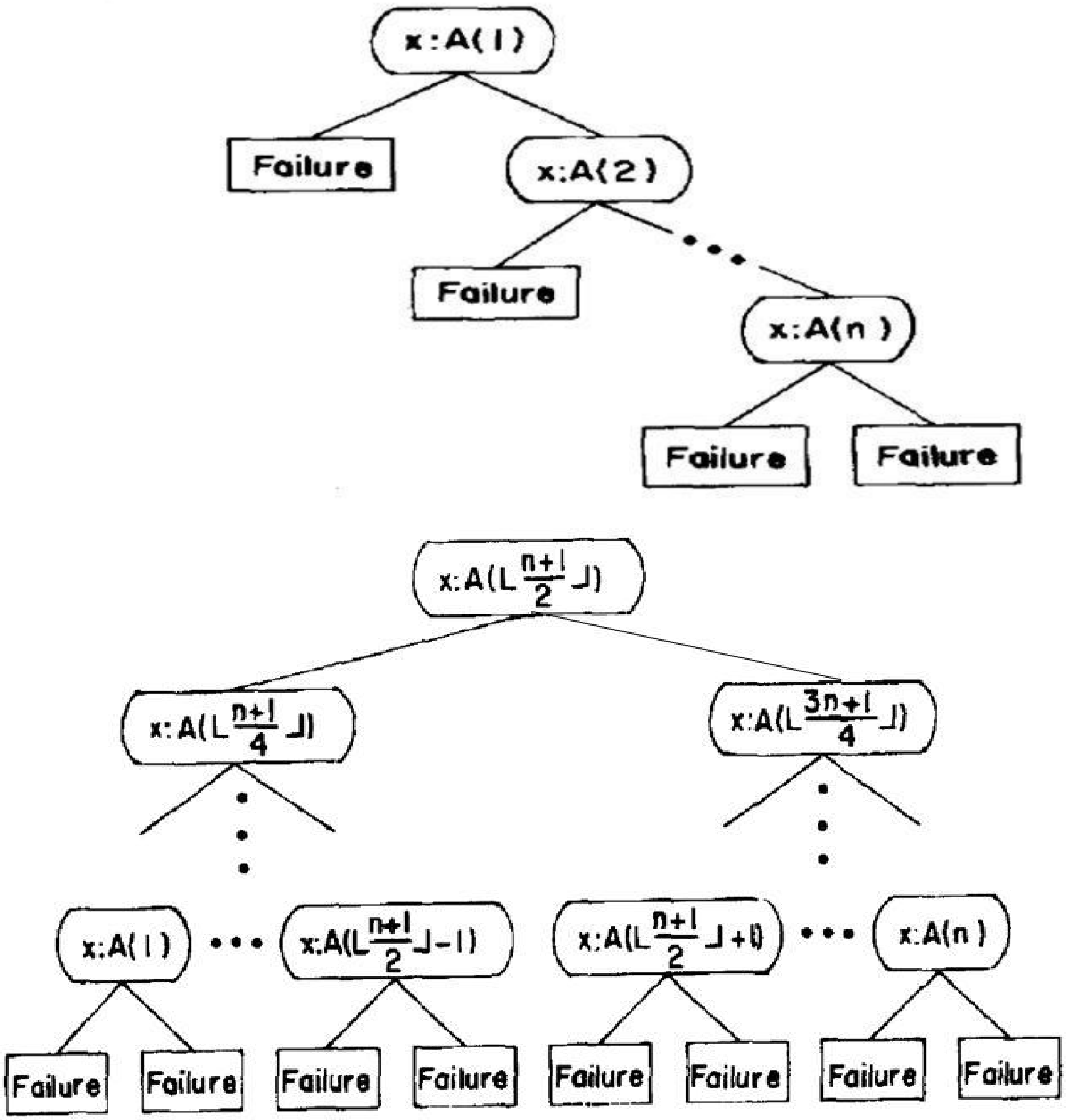
$$FIND(n) \geq \lceil \log(n+1) \rceil$$

② Sorting:

consider n numbers in A[1:n] is to be sorted.

comparing A[i] & A[j] (2 possibilities) → binary tree

- A[i] < A[j] → proceeds down to left branch
- A[i] > A[j] → proceeds down to right branch



Comparison trees for two searching algorithms

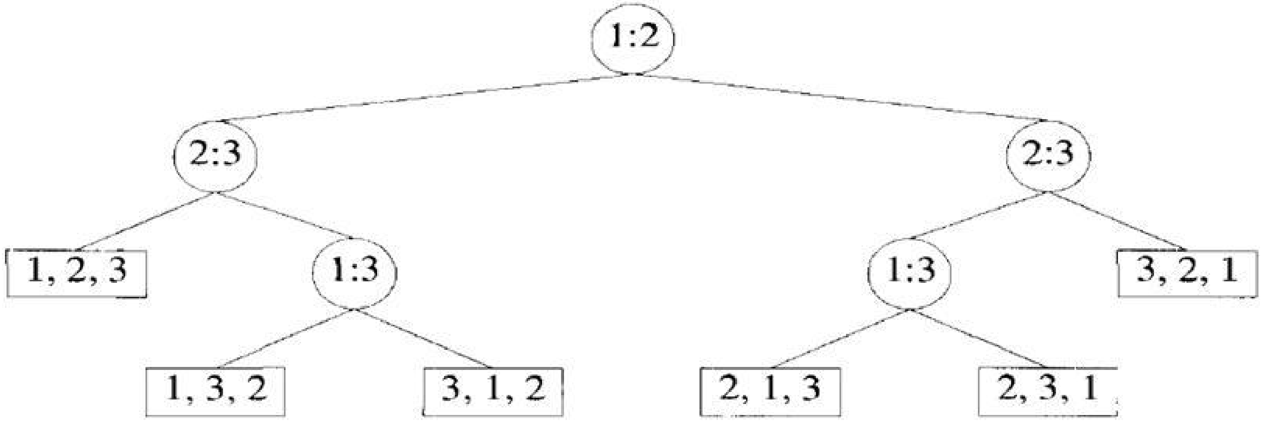


Figure 10.2 A comparison tree for sorting three items

Example 10.1 Let $A[1] = 21, A[2] = 13$, and $A[3] = 18$. At the root of the comparison tree (in Figure 10.2), 21 and 13 are compared, and as a result, the computation proceeds to the right subtree. Now, 13 and 18 are compared and the computation proceeds to the left subtree. Then $A[1]$ and $A[3]$ are compared and the computation proceeds to the right subtree to yield the permutation $A[2], A[3], A[1]$.

③ Selection

To Find maximum of n elements → atleast 2^{n-1} external nodes

* since each path from root to any external node must contain atleast n−1 internal nodes.

→ Atleast n−1 comparisons needed for otherwise atleast two of input item

suppose $L_k(n)$ denotes Lower bound for no. of comparisons

→ A comparison algorithm is to determine largest, second largest ... kth largest out of n elements. In worst case

$$L_k(n) \geq \lceil \log n(n-1) \cdots (n-k+1) \rceil$$ comparisions are needed.

Lower bounds through reductions –

1. Multiplying triangular matrices
2. Inverting a lower triangular matrix
3. Computing the transitive closure

Definition 10.1 Let P_1 and P_2 be any two problems. We say P_1 *reduces* to P_2 (also written $P_1 \propto P_2$) in time $\tau(n)$ if an instance of P_1 can be converted into an instance of P_2 and a solution for P_1 can be obtained from a solution for P_2 in time $\leq \tau(n)$.

Example 10.2 Let P_1 be the problem of selection (discussed in Section 3.6) and P_2 be the problem of sorting. Let the input have n numbers. If the numbers are sorted, say in an array $A[\ ]$, the ith-smallest element of the input can be obtained as $A[i]$. Thus P_1 reduces to P_2 in $O(1)$ time.

Example 10.3 Let S_1 and S_2 be two sets with m elements each. The problem P_1 is to check whether the two sets are *disjoint*, that is, whether $S_1 \cap S_2 = \emptyset$. P_2 is the sorting problem. We can show that $P_1 \propto P_2$ in $O(m)$ time as follows.

Let $S_1 = \{k_1, k_2, \ldots, k_m\}$ and $S_2 = \{\ell_1, \ell_2, \ldots, \ell_m\}$. The instance of P_2 to be created has $n = 2m$ and the sequence of keys to be sorted is $X = (k_1, 1), (k_2, 1), \ldots, (k_m, 1), (\ell_1, 2), (\ell_2, 2), \ldots, (\ell_m, 2)$. In other words, each key in X is a tuple and the sorting has to be done in lexicographic order. The conversion of P_1 to P_2 takes $O(m)$ time, since it involves the creation of $2m$ tuples.

4.5 Multiplying triangular matrices

Triangular Matrix Definition:

An $n \times n$ matrix A whose elements are $\{a_{ij}\}$, $1 \leq i, j \leq n$, is said to be *upper triangular* if $a_{ij} = 0$ whenever $i > j$. It is said to be *lower triangular* if $a_{ij} = 0$ for $i < j$. A matrix that is either upper triangular or lower triangular is said to be *triangular*.

Lower Triangular $\quad$ Upper Triangular

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & -1 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & -1 & -2 \\ 0 & 4 & -1 \\ 0 & 0 & 3 \end{bmatrix}$$

Lemma 10.5 $M_t(n) = \Omega(M(n))$.

Proof: We show that P_1 reduces to P_2 in $O(n^2)$ time. Note that $M(n) = \Omega(n^2)$ since there are $2n^2$ elements in the input and n^2 in the output. Let the two matrices to be multiplied be A and B and of size $n \times n$ each. The instance of P_2 to be created is the following:

$$A' = \begin{bmatrix} O & O & O \\ O & O & O \\ O & A & O \end{bmatrix} \qquad B' = \begin{bmatrix} O & O & O \\ B & O & O \\ O & O & O \end{bmatrix}$$

Here O stands for the *zero matrix*, that is, an $n \times n$ matrix all of whose entries are zeros. Both A' and B' are of size $3n \times 3n$ each. Multiplying the two matrices, we get

$$A'B' = \begin{bmatrix} O & O & O \\ O & O & O \\ AB & O & O \end{bmatrix}$$

Thus the product AB is easily obtainable from the product $A'B'$. Problem P_1 reduces to P_2 in $O(n^2)$ time. This reduction implies that $M(n) \leq M_t(3n) + O(n^2)$; this in turn means $M_t(n) \geq M(\frac{n}{3}) - O(n^2)$. Since $M(n) = \Omega(n^2)$, $M(\frac{n}{3}) = \Omega(M(n))$. Hence, $M_t(n) = \Omega(M(n))$.

Note that the above lemma also implies that $M_t(n) = \Theta(M(n))$.

4.6 Inverting a Lower Triangular Matrix:

Let A be an $n \times n$ matrix. Also, let I be the $n \times n$ *identity matrix*, that is, the matrix for which $i_{kk} = 1$, for $1 \leq k \leq n$, and whose every other element is zero. The elements a_{kk} of any matrix A are called the *diagonal elements* of A. Every element of I is zero except for the diagonal elements which are all ones. If there exists an $n \times n$ matrix B such that $AB = I$, then we say B is the *inverse* of A and A is said to be *invertible*. The inverse of A is also denoted as A^{-1}. Not every matrix is invertible. For example, the zero matrix has no inverse.

$$A = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix}$$

$$A\,A^{-1} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Lemma 10.6 $M(n) = O(I_t(n))$.

Proof: The claim is that P_1 reduces to P_2 in $O(n^2)$ time, from which the lemma follows. Let A and B be the two $n \times n$ full matrices to be multiplied. We construct the following lower triangular matrix in $O(n^2)$ time:

$$C = \begin{bmatrix} I & O & O \\ B & I & O \\ O & A & I \end{bmatrix}$$

where the O's and I's are $n \times n$ zero matrices and identity matrices, respectively. C is a $3n \times 3n$ matrix. The inverse of C is

$$C^{-1} = \begin{bmatrix} I & O & O \\ -B & I & O \\ AB & -A & I \end{bmatrix}$$

where $-A$ refers to A with all the elements negated. Here also we see that the product AB is obtainable easily from the inverse of C. Thus we get $M(n) \leq I_t(3n) + O(n^2)$, and hence $M(n) = O(I_t(n))$.

Lemma 10.7 $I_t(n) = O(M(n))$.

Proof: Let A be the $n \times n$ lower triangular matrix to be inverted. Partition A into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each as follows:

$$A = \begin{bmatrix} A_{11} & O \\ A_{21} & A_{22} \end{bmatrix}$$

Both A_{11} and A_{22} are lower triangular matrices and A_{21} could possibly be a full matrix. The inverse of A can be verified to be

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & O \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}$$

The above equation suggests a divide-and-conquer algorithm for inverting A. To invert A which is of size $n \times n$, it suffices to invert two lower triangular matrices (A_{11} and A_{22}) of size $\frac{n}{2} \times \frac{n}{2}$ each and perform two matrix multiplications (i.e., compute D=$A_{22}^{-1}(A_{21}A_{11}^{-1})$) and negate a matrix ($D$). D can be negated in $\frac{n^2}{4}$ time. The run time of such a divide-and-conquer algorithm satisfies the following recurrence relation:

$$I_t(n) \leq 2I_t\left(\frac{n}{2}\right) + 2M\left(\frac{n}{2}\right) + \frac{n^2}{4}$$

Using repeated substitution, we get

$$I_t(n) \leq 2M\left(\frac{n}{2}\right) + 2^2M\left(\frac{n}{2^2}\right) + \cdots + O(n^2)$$

Since $M(n) = \Omega(n^2)$, the above simplifies to

$$I_t(n) = O(M(n) + n^2) = O(M(n)).$$

Lemmas 10.6 and 10.7 together imply that $I_t(n) = \Theta(M(n))$.

4.7 Computing the Transitive Closure

Let G be a directed graph whose adjacency matrix is A. Recall that the reflexive transitive closure (or simply the transitive closure) of G, denoted A^*, is a matrix such that $A^*(i,j) = 1$ if and only if there is a directed path of length zero or more from node i to node j in G.

Lemma 10.10 $M(n) \leq T(3n) + O(n^2)$, and hence $M(n) = O(T(n))$.

Proof: If A and B are the given $n \times n$ matrices to be multiplied, form the following $3n \times 3n$ matrix C in $O(n^2)$ time:

$$C = \begin{bmatrix} O & A & O \\ O & O & B \\ O & O & O \end{bmatrix}$$

C^2 is given by

$$C^2 = \begin{bmatrix} O & O & AB \\ O & O & O \\ O & O & O \end{bmatrix}$$

Also, $C^k = O$ for $k \geq 3$. Therefore, using Lemma 10.9,

$$C^* = I + C + C^2 + \cdots + C^{n-1} = I + C + C^2 = \begin{bmatrix} I & A & AB \\ O & I & B \\ O & O & I \end{bmatrix}$$

Given C^*, it is easy to obtain the product AB.

Lemma 10.11 $T(n) = O(M(n))$.

Proof: The proof is analogous to that of Lemma 10.7. Let $G(V, E)$ be the graph under consideration and A its adjacency matrix. The matrix A is partitioned into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Recall that row i of A corresponds to edges going out of node i. Let V_1 be the set of nodes corresponding to rows $1, 2, \ldots, \frac{n}{2}$ of A and V_2 be the set of nodes corresponding to the rest of the rows.

The entry $A_{11}^*(i, j) = 1$ if and only if there is a path from node $i \in V_1$ to node $j \in V_1$ all of whose intermediate nodes are also from V_1. A similar property holds for A_{22}^*.

Let $D = A_{12}A_{21}$ and let u and $v \in V_1$. Then, $D(u, v) = 1$ if and only if there exists a $w \in V_2$ such that $\langle u, w \rangle$ and $\langle w, v \rangle$ are in E. A similar statement holds for $A_{21}A_{12}$.

Let the transitive closure of G be given by

$$A^* = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Our goal is to derive a divide-and-conquer algorithm for computing A^*. Therefore, we should find a way of computing C_{11}, C_{12}, C_{21}, and C_{22} from A_{11}^* and A_{22}^*.

Using similar reasoning, the rest of A^* can also be determined: $C_{12} = C_{11}A_{12}A_{22}^*$, $C_{21} = A_{22}^*A_{21}C_{11}$, and $C_{22} = A_{22}^* + A_{22}^*A_{21}C_{11}A_{12}A_{22}^*$.

Thus the above divide-and-conquer algorithm for computing A^* performs two transitive closures on matrices of size $\frac{n}{2} \times \frac{n}{2}$ each (A_{22}^* and $(A_{11} + A_{12}A_{22}^*A_{21})^*$), six matrix multiplications, and two matrix additions on matrices of size $\frac{n}{2} \times \frac{n}{2}$ each. Therefore we get

$$T(n) \leq 2T\left(\frac{n}{2}\right) + 6M\left(\frac{n}{2}\right) + O(n^2)$$

Repeated substitution yields

$$T(n) \leq \left[6M\left(\frac{n}{2}\right) + 12M\left(\frac{n}{4}\right) + 24M\left(\frac{n}{8}\right) + \cdots\right] + O(n^2)$$

But, $M(n) \geq n^2$, and hence $M(n/2) \leq 4M(n)$. Using this fact, we see that $T(n) = O(M(n) + n^2) = O(M(n))$.

Lemmas 10.10 and 10.11 show that $T(n) = \Theta(M(n))$.

****

UNIT-5 DESIGN AND ANALYSIS OF ALGORITHM (P, NP, NP-COMPLETE, NP-HARD PROBLEMS)

Introduction to Problems:

There are two groups in which a problem can be classified. The first group consists of the problems that can be solved in polynomial time. For example : searching of an element from the list O(   ), sorting of elements O(logn).

The second group consists of problems that can be solved in non-deterministic polynomial time. For example : Knapsack problem $O(2^{n/2})$ and Travelling Salesperson problem $(O(n^2 2^n))$.

- Any problem for which answer is either yes or no is called decision problem. The algorithm for decision problem is called **decision algorithm.**
- Any problem that involves the identification of optimal cost (minimum or maximum) is called optimization problem. The algorithm for optimization problem is called **optimization algorithm.**

Types of Algorithms

- Two types of Algorithms:
  1. **Deterministic Algorithm**: It has a property that result of every operation is uniquely defined.
  2. **Non Deterministic Algorithm**: It terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.
- To specify such algorithms, we introduce 3 functions:

1. Choice(S) arbitrarily chooses one of the elements of set S.
2. Failure() signals an unsuccessful completion.
3. Success() signals a successful completion.

Example 11.1 Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is Algorithm 11.1.

```
j := Choice(1, n);
if A[j] = x then {write (j); Success();}
write (0); Failure();
```

Algorithm 11.1 Nondeterministic search

Example 11.2 [Sorting] Let $A[i]$, $1 \le i \le n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) (Algorithm 11.2) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1:n]$ is used for convenience.

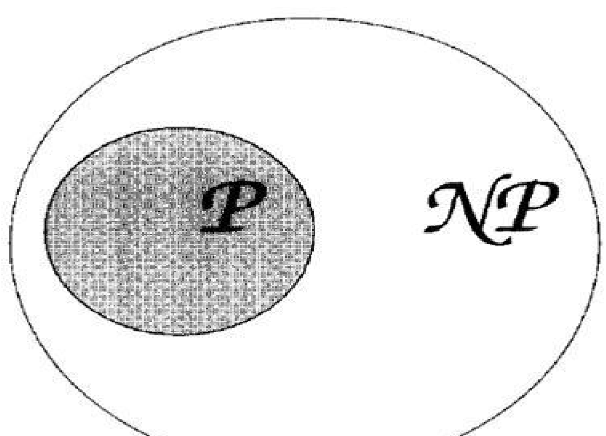
```
Algorithm NSort(A, n)
// Sort n positive integers.
{
    for i := 1 to n do B[i] := 0; // Initialize B[ ].
    for i := 1 to n do
    {
        j := Choice(1, n);
        if B[j] ≠ 0 then Failure();
        B[j] := A[i];
    }
    for i := 1 to n − 1 do  // Verify order.
        if B[i] > B[i + 1] then Failure();
    write (B[1 : n]);
    Success();
}
```

Algorithm 11.2 Nondeterministic sorting

P, NP PROBLEMS

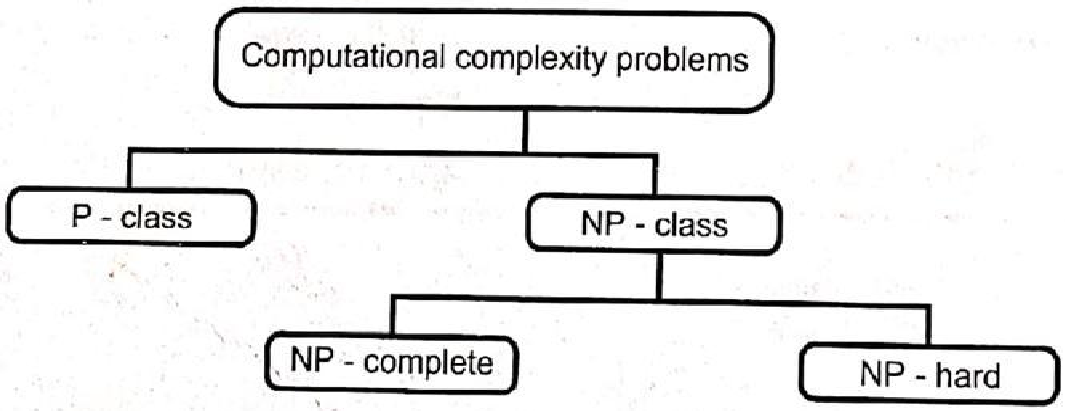
$\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic ones, we conclude that $\mathcal{P} \subseteq \mathcal{NP}$. What we do not know, and what has become perhaps the most famous unsolved problem in computer science, is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.



Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

NP-HARD & NP-COMPLETE



REDUCIBILITY

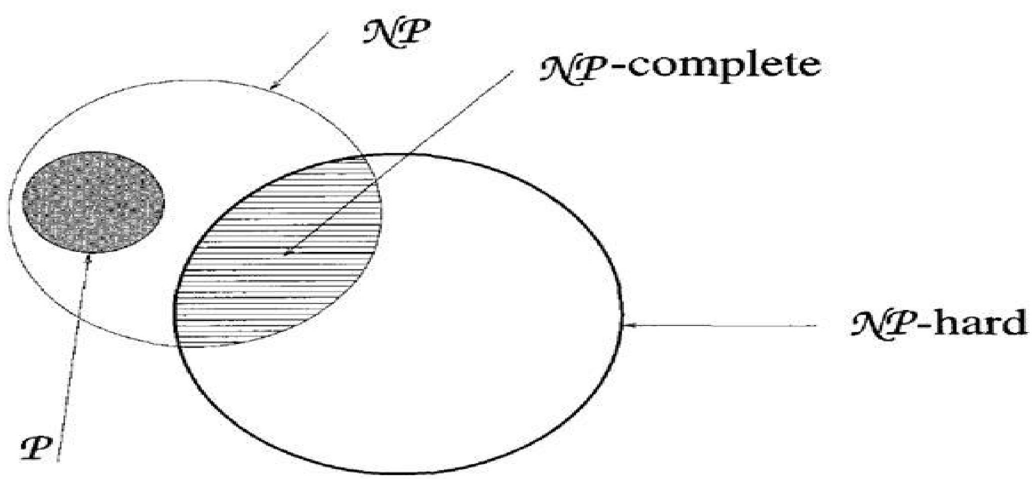
Let L_1 and L_2 be problems. Problem L_1 *reduces* to L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

NP HARD PROBLEM:

- Every problem in NP class can be reduced into another set using polynomial time, then it is called NP Hard Problem.

NP COMPLETE PROBLEM:

- The group of problems which are both in NP & NP Hard problem are known as NP Complete Problem.
- All NP problems are NP Hard but all NP Hard problems are not NP Complete problem.



Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems

Cook's Theorem:

Cook's theorem (Theorem 11.1) states that satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$. We now prove this important theorem. We have already seen that satisfiability is in $\mathcal{NP}$ (Example 11.9). Hence, if $\mathcal{P} = \mathcal{NP}$, then satisfiability is in $\mathcal{P}$. It remains to be shown that if satisfiability is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$. To do this, we show how to obtain from any polynomial time nondeterministic decision algorithm A and input I a formula $Q(A, I)$ such that Q is satisfiable iff A has a successful termination with input I.

Before going into the construction of Q from A and I, we make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions do not in any way alter the class of decision problems in $\mathcal{NP}$ or $\mathcal{P}$. The simplifying assumptions are as follows.

1. The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction, and so on

2. A *simple expression* is an expression that contains at most one operator and all operands are simple variables (i.e., no array variables are used). Some sample simple expression are $-B, B + C, D$ **or** E, and F. We assume that all assignments in A are in one of the following forms:

   (a) ⟨simple variable⟩ := ⟨simple expression⟩

   (b) ⟨array variable⟩ := ⟨simple variable⟩

   (c) ⟨simple variable⟩ := ⟨array variable⟩

   (d) ⟨simple variable⟩ := Choice(S), where S is a finite set $\{S_1, S_2, \ldots, S_k\}$ or l,u. In the latter case the function chooses an integer in the range $[l : u]$.

   Indexing within an array is done using a simple integer variable and all index values are positive. Only one-dimensional arrays are allowed. Clearly, all assignment statements not falling into one of the above categories can be replaced by a set of statements of these types. Hence, this restriction does not alter the class $\mathcal{NP}$.

3. All variables in A are of type integer or boolean.

4. Algorithm A contains no **read** or **write** statements. The only input to A is via its parameters. At the time A is invoked, all variables (other than the parameters) have value zero (or **false** if boolean).

5. Algorithm A contains no constants. Clearly, all constants in any algorithm can be replaced by new variables. These new variables can be added to the parameter list of A and the constants associated with them can be part of the input.

6. In addition to simple assignment statements, A is allowed to contain only the following types of statements:

   (a) The statement **goto** k, where k is an instruction number.

   (b) The statement **if** c **then goto** a;. Variable c is a simple boolean variable (i.e., not an array) and a is an instruction number.

   (c) Success(), Failure().

   (d) Algorithm A may contain type declaration and dimension statements. These are not used during execution of A and so need not be translated into Q.

7. Let $p(n)$ be a polynomial such that A takes no more than $p(n)$ time units on any input of length n. Because of the complexity assumption of 1), A cannot change or use more than $p(n)$ words of memory.

Formula Q makes use of several boolean variables. We state the semantics of two sets of variables used in Q:

1. $B(i, j, t)$, $1 \leq i \leq p(n)$, $1 \leq j \leq w$, $0 \leq t < p(n)$

   $B(i, j, t)$ represents the status of bit j of word i following t steps (or time units) of computation. The bits in a word are numbered from right to left. The rightmost bit is numbered 1. Q is constructed so that in any truth assignment for which Q is true, $B(i, j, t)$ is true if and only if the corresponding bit has value 1 following t steps of some successful computation of A on input I.

2. $S(j, t)$, $1 \leq j \leq \ell$, $1 \leq t \leq p(n)$

   Recall that ℓ is the number of instructions in A. $S(j, t)$ represents the instruction to be executed at time t. Q is constructed so that in any truth assignment for which Q is true, $S(j, t)$ is true if and only if the instruction executed by A at time t is instruction j.

Q is made up of six subformulas, C, D, E, F, G, and H. $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$. These subformulas make the following assertions:

C: The initial status of the $p(n)$ words represents the input I. All non-input variables are zero.

D: Instruction 1 is the first instruction to execute.

E: At the end of the ith step, there can be only one next instruction to execute. Hence, for any fixed i, exactly one of the $S(j, i)$, $1 \leq j \leq \ell$, can be true.

F: If $S(j, i)$ is true, then $S(j, i+1)$ is also true if instruction j is a Success or Failure statement. $S(j+1, i+1)$ is true if j is an assignment statement.

G: If the instruction executed at step t is not an assignment statement, then the $B(i, j, t)$'s are unchanged. If this instruction is an assignment and the variable on the left-hand side is X, then only X may change. This change is determined by the right-hand side of the instruction.

H: The instruction to be executed at time $p(n)$ is a Success instruction. Hence the computation terminates successfully.

Clearly, if C through H make the above assertions, then $Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$ is satisfiable if and only if there is a successful computation of A on input I.

Satifiability problem

[Satisfiability] Let $x_1, x_2, \ldots$ denote boolean variables (their value is either true or false). Let $\bar{x_i}$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x_4})$ and $(x_3 \vee \bar{x_4}) \wedge (x_1 \vee \bar{x_2})$. The symbol $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) if and only if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x_4})$ is in DNF whereas $(x_3 \vee \bar{x_4}) \wedge (x_1 \vee \bar{x_2})$ is in CNF. The **satisfiability** problem is to determine whether a formula is true for some assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

3 CNF Satifiability problem

2. A 3SAT problem

A 3SAT problem is a problem which takes a Boolean formula S in CNF form with each clause having **exactly three literals** and check whether S is satisfied or not. [Note that CNF means each literal is ORed to form a clause, and each clause is ANDed to form Boolean formula S].

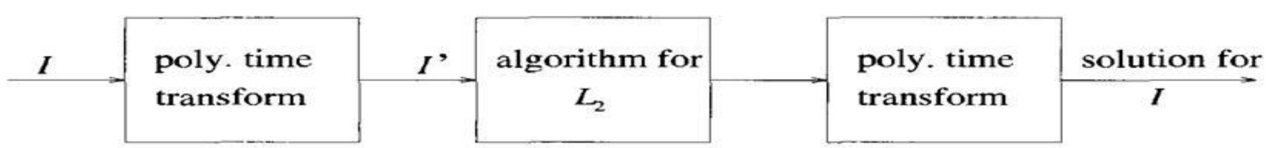
Following formula is an instance of 3SAT problem :

$$(\bar{a} + b + \bar{g})\ (c + \bar{e} + f)\ (\bar{b} + d + \bar{f})\ (a + e + \bar{h})$$

Theorem : 3SAT is in NP complete.

Proof : Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S. If the S is evaluated as 1 then S is satisfied. Thus we can prove that 3SAT is in NP-complete.

REDUCTIONS FOR SOME KNOWN PROBLEMS



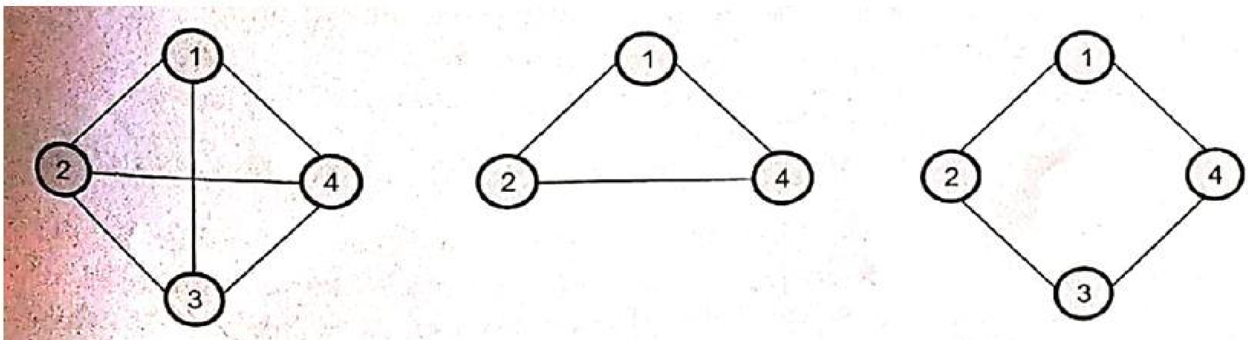
Reduction of L_1 to L_2

The strategy we adopt to show that a problem L_2 is $\mathcal{NP}$-hard is:

1. Pick a problem L_1 already known to be $\mathcal{NP}$-hard.
2. Show how to obtain (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I of L_1 (see Figure 11.3).
3. Conclude from step (2) that $L_1 \propto L_2$.
4. Conclude from steps (1) and (3) and the transitivity of $\propto$ that L_2 is $\mathcal{NP}$-hard.

MAXIMUM CLIQUE PROBLEM

[Maximum clique] A maximal complete subgraph of a graph $G = (V, E)$ is a *clique*. The size of the clique is the number of vertices in it. The *max clique problem* is an optimization problem that has to determine the size of a largest clique in G. The corresponding decision problem is to determine whether G has a clique of size at least k for some given k. Let DClique(G, k) be a deterministic decision algorithm for the clique decision problem. If the number of vertices in G is n, the size of a max clique in G can be found by making several applications of DClique. DClique is used once for each k, $k = n, n-1, n-2, \ldots$, until the output from DClique is 1.



Graph G | Clique size = 3 | Clique size = 4

CLIQUE DECISION PROBLEM

The clique decision problem (CDP) is NP-complete.

Proof : Let, F be a formula for CNF which is satisfiable.

$$F = C_1 \wedge C_2 \wedge \ldots C_k$$

where C is a clause. Every clause in CNF is denoted by a_i where $1 \le i \le n$. I length of F is F and is obtained in time O(m) then we can obtain polynomia time algorithm in CDP.

Let us design a graph G = (V, E) with set of vertices

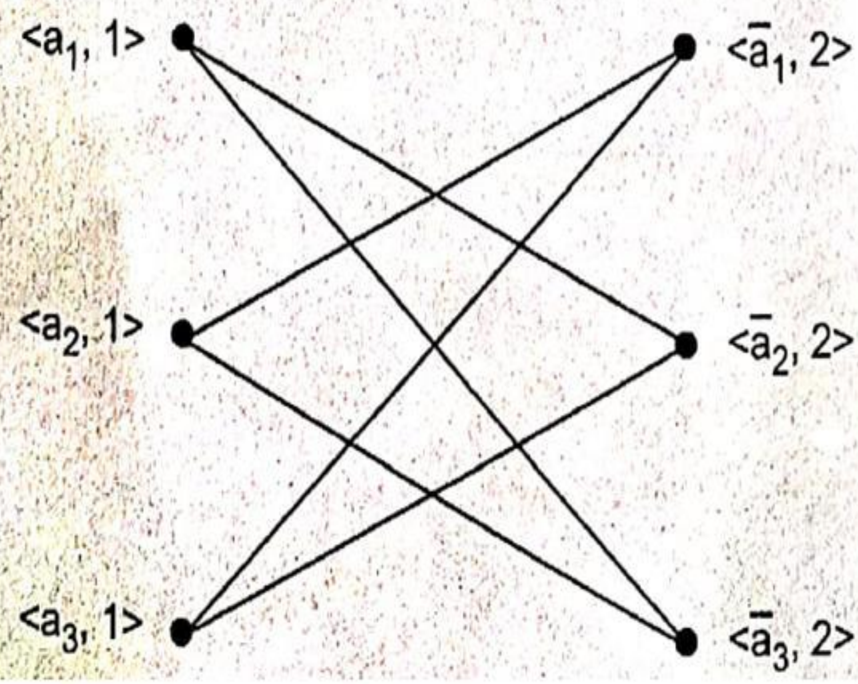
$V = \{<\sigma, i> | \sigma \text{ is a literal in } C_i\}$ and set of edges

$E = \{(<\sigma, i>, <\delta, j>) | i \ne j \text{ and } \sigma \ne \bar{\delta}\}$

For example :

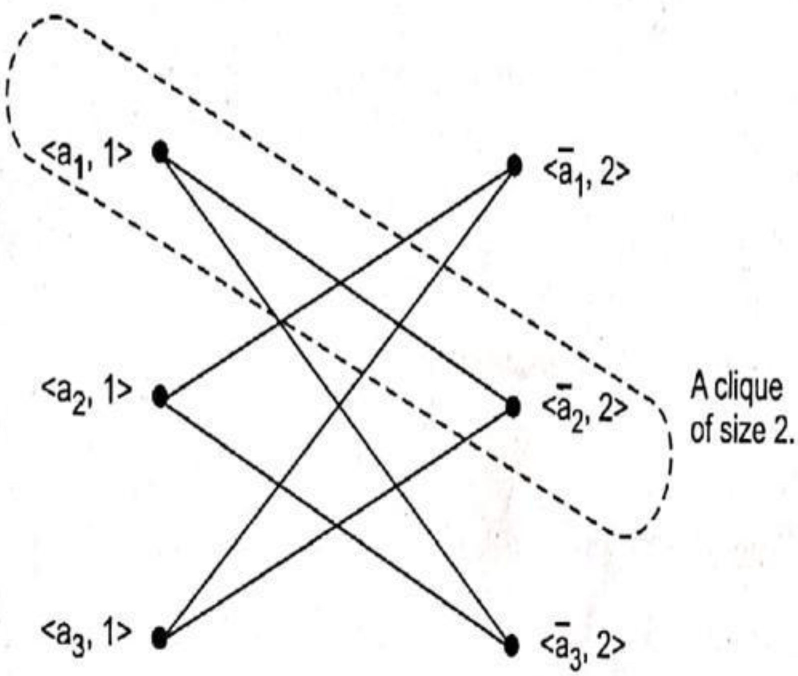
$$F = \left(\underbrace{(a_1 \vee a_2 \vee a_3)}_{C_1}\right) \wedge \left(\underbrace{(\bar{a}_1 \vee \bar{a}_2 \vee \bar{a}_3)}_{C_2}\right)$$

The graph G can be drawn as follows



The formula F is satisfiable if and only if G has a clique of size > k. The F will be satisfiable only when at least one literal σ in C_i is true.

Thus the graph as six cliques of size two. Note that F is satisfiable as well.

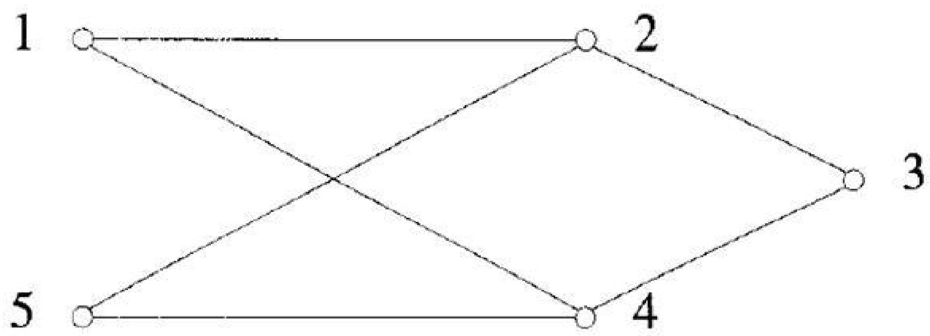


As CNF satisfiability is NP complete CDP is also NP complete.

NODE COVER DECISION PROBLEM

A set $S \subseteq V$ is a *node cover* for a graph $G = (V, E)$ if and only if all edges in E are incident to at least one vertex in S. The size $|S|$ of the cover is the number of vertices in S.

Example 11.12 Consider the graph of Figure 11.5. $S = \{2, 4\}$ is a node cover of size 2. $S = \{1, 3, 5\}$ is a node cover of size 3.



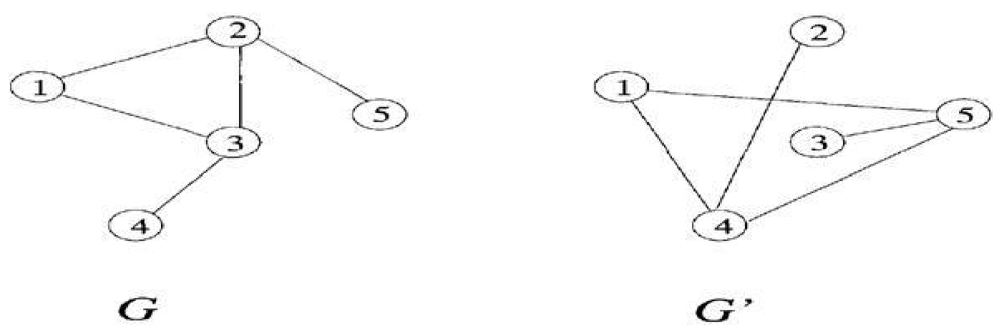
A sample graph and node cover

In the node cover decision problem we are given a graph G and an integer k. We are required to determine whether G has a node cover of size at most k.

Theorem 11.3 The clique decision problem $\propto$ the node cover decision problem.

Proof: Let $G = (V, E)$ and k define an instance of CDP. Assume that $|V| = n$. We construct a graph G' such that G' has a node cover of size at most $n - k$ if and only if G has a clique of size at least k. Graph G' is given by $G' = (V, \bar{E})$, where $\bar{E} = \{(u, v) \mid u \in V, v \in V \text{ and } (u, v) \notin E\}$. The set G' is known as the *complement* of G.

Example 11.13 Figure 11.6 shows a graph G and its complement G'. In this figure, G' has a node cover of $\{4, 5\}$, since every edge of G' is incident either on the node 4 or on the node 5. Thus, G has a clique of size $5 - 2 = 3$ consisting of the nodes 1, 2, and 3.

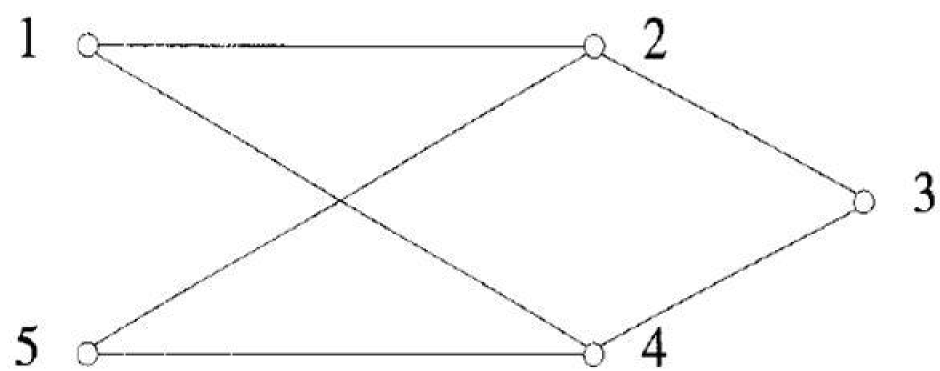


A graph and its complement

CHROMATIC NUMBER DECISION PROBLEM

A coloring of a graph $G = (V, E)$ is a function $f : V \to \{1, 2, \ldots, k\}$ defined for all $i \in V$. If $(u, v) \in E$, then $f(u) \neq f(v)$. The *chromatic number decision problem* is to determine whether G has a coloring for a given k.

Example 11.14 A possible 2-coloring of the graph of Figure 11.5 is $f(1) = f(3) = f(5) = 1$ and $f(2) = f(4) = 2$. Clearly, this graph has no 1-coloring.



DIRECTED HAMILTONIAN CYCLE

A directed Hamiltonian cycle in a directed graph $G = (V, E)$ is a directed cycle of length $n = |V|$. So, the cycle goes through every vertex exactly once and then returns to the starting vertex. The DHC problem is to determine whether G has a directed Hamiltonian cycle.

Example 11.15 1, 2, 3, 4, 5, 1 is a directed Hamiltonian cycle in the graph of Figure 11.7. If the edge $\langle 5, 1 \rangle$ is deleted from this graph, then it has no directed Hamiltonian cycle.

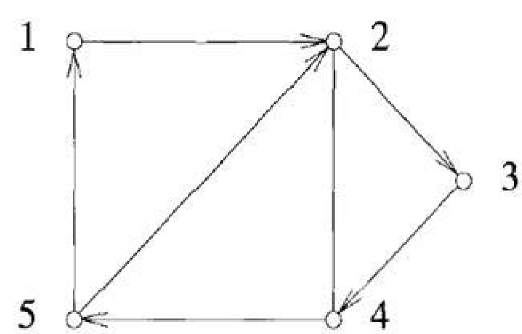


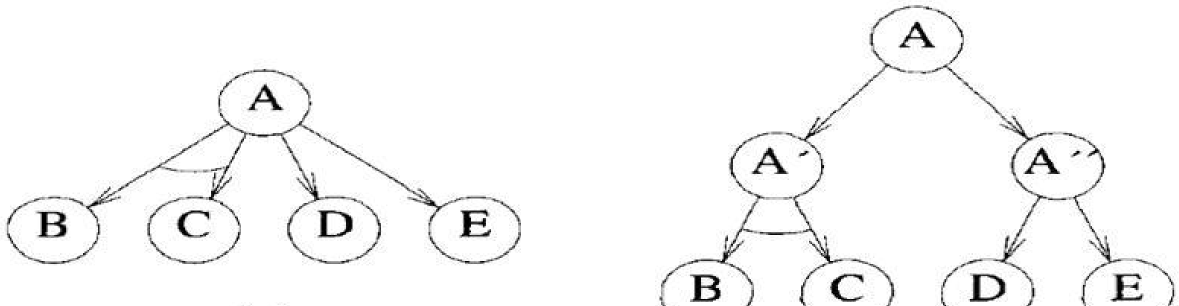
Figure 11.7 A sample graph and Hamiltonian cycle

TRAVELLING SALES PERSON DECISION PROBLEM

The traveling salesperson problem was introduced in Chapter 5. The corresponding decision problem is to determine whether a complete directed graph $G = (V, E)$ with edge costs $c(u, v)$ has a tour of cost at most M.

Theorem 11.6 Directed Hamiltonian cycle (DHC) $\propto$ the traveling salesperson decision problem (TSP).

Proof: From the directed graph $G = (V, E)$ construct the complete directed graph $G' = (V, E')$, $E' = \{\langle i, j\rangle \mid i \neq j\}$ and $c(i, j) = 1$ if $\langle i, j\rangle \in E$; $c(i, j) = 2$ if $i \neq j$ and $\langle i, j\rangle \notin E$. Clearly, G' has a tour of cost at most n iff G has a directed Hamiltonian cycle.



(a)

Graphs representing problems (b)

****