

DESIGN AND ANALYSIS OF ALGORITHMS

Dr. N. Subhash Chandra

Course Objectives

Upon completion of this course, students will be able to do the following:

1. Analyze the asymptotic performance of algorithms.
2. To understand how the choice of data structures and algorithm design methods impacts the performance of programs.
3. To solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking and branch and bound

Course Outcomes

CO 1: Analyze algorithms, improve the efficiency of algorithms and ability to understand and

estimate the performance of algorithm.

CO 2: Choose the appropriate data structure and algorithms design method for a specified application.

CO 3: Apply different designing methods for development of algorithms to realistic problem, such as divide-and-conquer, greedy algorithms, synthesize divide-and-conquer, greedy algorithms, and analyze them.

CO 4: Describe the dynamic-programming, backtracking paradigm and explain when an algorithm design situation calls for it. Recite algorithms that employ these paradigms.

CO 5: Synthesize dynamic-programming, backtracking algorithms, and analyze them. To apply algorithm design paradigms for complex problems and solve novel problems, by choosing the appropriate algorithm design technique for their solution and justify their selection.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1

Unit - 1

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Algorithm, pseudo code for expressing algorithms. CO1

Definition: An *algorithm* is a sequence of unambiguous instructions for solving a problem. It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

Characteristics of an algorithm:

Every algorithm must be satisfied the following characteristics.

- Input : Zero / more quantities are externally supplied.
- Output : At least one quantity is produced.
- Definiteness : Each instruction is clear and unambiguous.
- Finiteness : If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
- Efficiency : Every instruction must be very basic and runs in short time with effective results better than human computations.

Pseudo code for Expressing Algorithms:

1. An algorithm is a procedure. It has two parts; the first part is head and the second part is body.
2. The Head section consists of keyword Algorithm and Name of the algorithm with parameter list.

E.g. Algorithm name1(p1, p2,...,p3)

The head section also has the following:

//Problem Description:

//Input:

//Output:

3. In the body of an algorithm various programming constructs like if, for, while and some statements like assignments are used.
4. The compound statements may be enclosed with { and } brackets. if, for, while can be open and closed by {, } respectively. Proper indention is must for block.

5. Comments are written using // at the beginning.
6. The identifier should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
7. The left arrow " := " used as assignment operator. E.g. v:=10
8. Boolean operators (TRUE, FALSE), Logical operators (AND, OR, NOT) and Relational operators (<, <=, >, >=, =, ≠, <>) are also used.
9. Input and Output can be done using read and write.
10. Array[], if then else condition, branch and loop can be also used in algorithm.

Example:

The greatest common divisor(GCD) of two nonnegative integers m and n (not-both-zero, $m \leq n$), denoted $\text{gcd}(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$, where $m \bmod n$ is the remainder of the division of m by n , until $m \bmod n$ is equal to 0. Since $\text{gcd}(m, 0) = m$, the last value of m is also the greatest common divisor of the initial m and n .

$\text{gcd}(60, 24)$ can be computed as follows: $\text{gcd}(60, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12$.

Euclid's algorithm for computing $\text{gcd}(m, n)$ in simple steps

- Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.
- Step 2 Divide m by n and assign the value of the remainder to r .
- Step 3 Assign the value of n to m and the value of r to n . Go to Step 1.

Euclid's algorithm for computing $\text{gcd}(m, n)$ expressed in pseudocode

```

ALGORITHM Euclid_gcd(m, n)
{
  //Computes gcd(m, n) by Euclid's algorithm
  //Input: Two nonnegative, not-both-zero integers m and n
  //Output: Greatest common divisor of m and n
  while n ≠ 0 do
  {
    r := m mod n;
  }
}

```

```
        m:=n;  
        n:=r;  
    }  
    return m;  
}
```



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 2

Unit - 1

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Fundamentals Algorithm, Problem Solving: CO1

(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (*instance*) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device

1. In *random-access machine (RAM)*, instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called *sequential algorithms*.
2. In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called *parallel algorithms*.
3. Choice of computational devices like Processor and memory is mainly based on space and time efficiency

(b) Choosing between Exact and Approximate Problem Solving

- i. The next principal decision is to choose between solving the problem exactly or solving it approximately.
- ii. An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- iii. If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm.

i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

1. An *algorithm design technique* (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
2. Algorithms+ *Data Structures* = *Programs*
3. Though Algorithms and Data Structures are independent, but they are combined to develop program. Hence the choice of proper data structure is required before designing the algorithm.
4. Implementation of algorithm is possible only with the help of Algorithms and Data Structures
5. Algorithmic strategy / technique / paradigm is a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

- a. Natural language
- b. Pseudocode
- c. Flowchart

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times, specification of algorithm by using natural language is not clear and thereby we get brief specification.

b. Pseudocode

Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

c. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a *flowchart*, this representation technique has proved to be inconvenient. *Flowchart* is a graphical representation of an algorithm. It is a method

of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

(iv) **Proving an Algorithm's Correctness**

Once an algorithm has been specified then its *correctness* must be proved. An algorithm must yield a required result for every legitimate input in a finite amount of time.

Example: Addition of a and b

```
Start
Input the value of a;
Input the value of b;
c: = a + b;
Display the value of c;
Stop
```

(v) **Analyzing an Algorithm**

For an algorithm the most important is *efficiency*. In fact, there are two kinds of algorithm efficiency. They are:

Time efficiency, indicating how fast the algorithm runs, and
Space efficiency, indicating how much extra memory it uses.

The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.

So factors to analyze an algorithm are:

1. Time efficiency of an algorithm
2. Space efficiency of an algorithm
3. Simplicity of an algorithm
4. Generality of an algorithm

(vi) **Coding an Algorithm**

□ The coding / implementation of an algorithm is done by a suitable programming language

like C, C++, JAVA.

1. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not be reduced by inefficient implementation.

2. Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
3. Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
4. It is very essential to write an optimized code (efficient code) to reduce the burden of
5. compiler.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - 1

Year and Semester: IYr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Performance Analysis: CO1

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

- a) Analysis Framework.
 - b) Asymptotic Notations and its properties.
 - c) Mathematical analysis for Recursive algorithms.
 - d) Mathematical analysis for Non-recursive algorithms.
- a) **Analysis Framework:** There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

Time efficiency, indicating how fast the algorithm runs, and

Space efficiency, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

- i. Measuring an Input's Size
 - ii. Units for Measuring Running Time
 - iii. Orders of Growth
 - iv. Worst-Case, Best-Case, and Average-Case Efficiencies
- i) Measuring an Input's Size: An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward.

For example, it will be the size of the list for problems of sorting, searching. For the problem of evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n , the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.

In computing the product of two $n \times n$ matrices, the choice of a parameter indicating an input size does matter.

Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.

In measuring input size for algorithms solving problems such as checking primality of a positive integer n . the input is just one number.

The input size by the number b of bits in the n 's binary representation is $b = (\log_2 n) + 1$.

(ii) Units for Measuring Running Time : Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- a) Dependence on the speed of a computer.
- b) Dependence on the quality of a program implementing the algorithm.
- c) The compiler used in generating the machine code.
- d) The difficulty of clocking the actual running time of the program.

So, we need metric to measure an *algorithm's* efficiency that does not depend on these extraneous factors. One possible approach is to *count the number of times each of the algorithm's operations is executed*. This approach is excessively difficult.

The most important operation (+, -, *, /) of the algorithm, called the *basic operation*. Computing the number of times the basic operation is executed is easy. The total running time is determined by basic operations count.

(iii) Orders of Growth

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.

For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only. For large values of n , it is the function's order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms.

Table 1.1 Growth of function order

n	\sqrt{n}	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	1	0	1	0	1	1	2	1
2	1.4	1	2	2	4	4	4	2
4	2	2	4	8	16	64	16	24
8	2.8	3	8	$2.4 \cdot 10^1$	64	$5.1 \cdot 10^2$	$2.6 \cdot 10^2$	$4.0 \cdot 10^4$
10	3.2	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
16	4	4	16	$6.4 \cdot 10^1$	$2.6 \cdot 10^2$	$4.1 \cdot 10^3$	$6.5 \cdot 10^4$	$2.1 \cdot 10^{13}$
10^2	10	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

(iv) **Worst-Case, Best-Case, and Average-Case Efficiencies** Consider Sequential Search algorithm some search key K

```

ALGORITHM SequentialSearch(A[0..n - 1], X)
{
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n - 1] and a search key X
//Output: The index of the first element in A that matches K or -1 if
there are no
// matching elements
i ← 0;
while i < n and A[i] ≠ X do
    i ← i + 1;
if i < n return i
else return -1;
}

```

Clearly, the running time of this algorithm can be quite different for the same list size n . In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

Worst-case efficiency

The *worst-case efficiency* of an algorithm is its efficiency for the worst case input of size n . The algorithm runs the longest among all possible inputs of that size. For the input of size n , the running time is $C_{worst}(n) = n$.

Best case efficiency

The *best-case efficiency* of an algorithm is its efficiency for the best case input of size n . The algorithm runs the fastest among all possible inputs of that size n . In sequential search, If we search a first element in list of size n . (i.e. first element equal to a search key), then the running time is $C_{best}(n) = 1$

Average case efficiency

The Average case efficiency lies between best case and worst case. To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n .

Time complexity-Space Complexity

- Two criteria are used to judge algorithms: (i) time complexity (ii) space complexity.
- Space Complexity of an algorithm is the amount of memory it needs to run to completion.
- Time Complexity of an algorithm is the amount of CPU time it needs to run to completion.

Space Complexity:

Memory space $S(P)$ needed by a program P , consists of two components:

- A fixed part: needed for instruction space (byte code), simple variable space, constants space etc. $\rightarrow c$
- A variable part: dependent on a particular instance of input and output data.
 $\rightarrow S_p(\text{instance})$

$$S(P) = c + S_p(\text{instance})$$

Example 1:

Algorithm abc (a, b, c)

```
{  
1.      return a+b+b*c+(a+b-c)/(a+b)+4.0;  
}
```

For every instance 3 computer words required to store variables: a, b, and c.

Therefore $S_p() = 3$. $S(P) = 3$.

Example 2:

Algorithm Sum(a[], n)

```
{  
1.      s := 0.0;  
2.      for i = 1 to n do  
3.          s := s + a[i];  
4.      return s;  
}
```

Every instance needs to store array a[] & n.

1. Space needed to store n = 1 word.
2. Space needed to store a[] = n floating point words (or at least n words)
3. Space needed to store i and s = 2 words

$$S_p(n) = (n + 3). \text{ Hence } S(P) = (n + 3).$$

Time Complexity:

- How to measure $T(P)$?

- Measure experimentally, using a "stop watch"
 - T(P) obtained in secs, msec.
- Count program steps → T(P) obtained as a step count.
- Fixed part is usually ignored; only the variable part $t_p()$ is measured.

	Statements	S/E	Freq.	Total
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 Count				2n+3

What is a program step?

- $a+b+b*c+(a+b)/(a-b)$ → one step;
- comments → zero steps;
- while (<expr>) do → step count equal to the number of times <expr> is executed.
- for i=<expr> to <expr1> do → step count equal to number of times <expr1> is checked.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout - 4

Unit - 1

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Asymptotic notations: CO1

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program. The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use five notations, they are:

- O - Big oh notation
- Ω - Big omega notation
- Θ - Big theta notation
- o- Little oh notation
- ω -Little omega notation

Asymptotically Non-Negative: A function $g(n)$ is *asymptotically nonnegative*, if $g(n) \geq 0$ for all $n \geq n_0$ where n_0 in $N = \{0, 1, 2, 3, \dots\}$

Asymptotic Upper Bound: $O(\text{Big-oh})$

Definition: Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. We say $f(n)$ is in $O(g(n))$ if there is a real positive constant c and a positive Integer n_0 such that for every $n \geq n_0$, $0 \leq f(n) \leq c * g(n)$.

(Or)

$$O(g(n)) = \{ f(n) \mid \text{there exist a positive constant } c \text{ and a positive integer } n_0 \text{ such that } 0 \leq f(n) \leq c * g(n) \text{ for all } n \geq n_0 \}$$

The Figure 1.1 shows the growth function of $f(n)$ and $g(n)$ for case of asymptotic upper bound

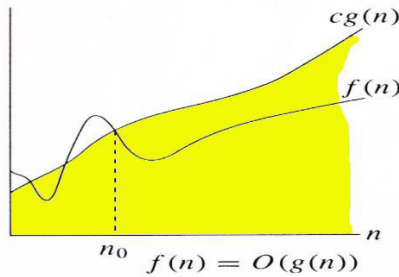


Figure 1.1 $f(n)=O(g(n))$ growth function

Example 1: Verify $5n+2 = O(n)$.

Solution:

From the definition of Big Oh, there must exist $c>0$ and integer $n_0 >0$ such that

$$0 \leq 5n+2 \leq c \cdot n \quad \text{for all } n \geq n_0.$$

Dividing both sides of the inequality by $n>0$ we get:

$$0 \leq 5 + \frac{2}{n} \leq c.$$

Clearly $2/n \leq 2$, since $2/n >0$ becomes smaller when n increases.

There are many choices here for c and n_0 .

If we choose $n_0 = 1$ then $c \geq 5 + 2/1 = 7$.

If we choose $c=6$, then $0 \leq 5 + 2/n \leq 6$. So $n_0 \geq 2$.

In either case (we only need one!) we have a $c>0$ and $n_0 >0$ such that $0 \leq 5n+2 \leq cn$ for all $n \geq n_0$.

So the definition is satisfied and $5n+2 = O(n)$

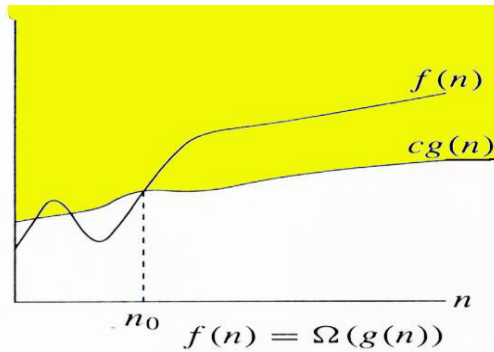
Asymptotic Lower Bound: $\Omega(\text{Big-Omega})$

Definition:

Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. We say $f(n)$ is $\Omega(g(n))$ if there is a positive real constant c and a positive integer n_0 such that for every $n \geq n_0$ $0 \leq c \cdot g(n) \leq f(n)$.

(Or)

$\Omega(g(n)) = \{ f(n) \mid \text{there exist positive constant } c \text{ and a positive integer } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}$



From the definition of Omega, there must exist $c > 0$ and integer $n_0 > 0$ such that $0 \leq c \cdot n \leq 5n - 20$ for all $n \geq n_0$

Dividing the inequality by $n > 0$ we get: $0 \leq c \leq 5 - 20/n$ for all $n \geq n_0$.

$20/n \leq 20$, and $20/n$ becomes smaller as n grows.

There are many choices here for c and n_0 .

Since $c > 0$, $5 - 20/n > 0$ and $n_0 > 4$

For example, if we choose $c=4$, then $5 - 20/n \leq 4$ and $n_0 \geq 20$

In this case we have a $c > 0$ and $n_0 > 0$ such that $0 \leq c \cdot n \leq 5n - 20$ for all $n \geq n_0$. So the definition is satisfied and $5n - 20$ in $\Omega(n)$

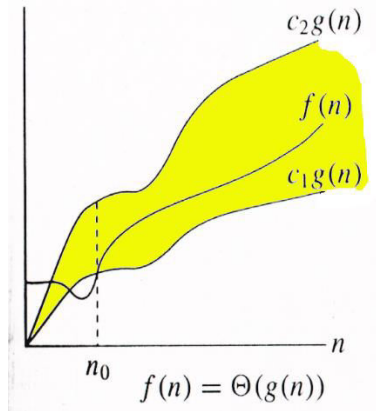
Asymptotic Tightly Bound: θ (Theta)

Definition: Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. We say $f(n)$ is $\theta(g(n))$ if there are positive constants c, d and a positive integer n_0 such that for every $n \geq n_0$

$$0 \leq c \cdot g(n) \leq f(n) \leq d \cdot g(n).$$

(Or)

$\theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c, d \text{ and a positive integer } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \leq d \cdot g(n) \text{ for all } n \geq n_0\}$



Example: Prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Proof:

It is enough to prove that

$$\frac{1}{2}n^2 - 3n = O(n^2)$$

$$\frac{1}{2}n^2 - 3n = \Omega(n^2)$$

From the definition there must exist $c > 0$, and $n_0 > 0$ such that

$$0 \leq \frac{1}{2}n^2 - 3n \leq cn^2 \text{ for all } n \geq n_0.$$

Dividing the inequality by $n^2 > 0$ we get:

$$0 \leq \frac{1}{2} - \frac{3}{n} \leq c \text{ for all } n \geq n_0.$$

Since $3/n > 0$ for finite n , $c < 1/2$.

Choose $c = 1/4$.

So $\frac{1}{2} - \frac{3}{n} \leq \frac{1}{4}$, and $n_0 \geq 12$

There must exist $c > 0$ and $n_0 > 0$ such that

$$0 \leq cn^2 \leq \frac{1}{2}n^2 - 3n \text{ for all } n \geq n_0$$

Dividing by $n^2 > 0$ we get

$$0 \leq c \leq \frac{1}{2} - \frac{3}{n}.$$

Since $c > 0$, $0 < \frac{1}{2} - \frac{3}{n}$ and $n_0 > 6$.

Since $3/n > 0$ for finite n , $c < 1/2$. Choose $c = 1/4$.

$$\frac{1}{4} \leq \frac{1}{2} - \frac{3}{n} \text{ for all } n \geq 12.$$

So $c = 1/4$ and $n_0 = 12$.

Asymptotic *o*(Little-oh)

Definition: Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. We say $f(n)$ is $o(g(n))$ if for every positive real constant c there exists a positive integer n_0 such that for all $n \geq n_0$

$$0 \leq f(n) < c * g(n).$$

(Or)

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a positive integer } n_0 > 0 \text{ such that } 0 \leq f(n) < c * g(n) \text{ for all } n \geq n_0\}$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout - 5

Unit - 1

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Calculating the running time of programs: CO1

Let us now look into how big-O bounds can be computed for some common algorithms.

Example :

$2n^2 + 5n - 6 = O(2^n)$	$2n^2 + 5n - 6 \square\square (2^n)$
$2n^2 + 5n - 6 = O(n^3)$	$2n^2 + 5n - 6 \square\square (n^3)$
$2n^2 + 5n - 6 = O(n^2)$	$2n^2 + 5n - 6 = \square (n^2)$
$2n^2 + 5n - 6 \square O(n)$	$2n^2 + 5n - 6 \square\square (n)$
$2n^2 + 5n - 6 \square\square (2^n)$	$2n^2 + 5n - 6 = o(2^n)$
$2n^2 + 5n - 6 \square\square (n^3)$	$2n^2 + 5n - 6 = o(n^3)$
$2n^2 + 5n - 6 = \square (n^2)$	$2n^2 + 5n - 6 \square o(n^2)$
$2n^2 + 5n - 6 = \square (n)$	$2n^2 + 5n - 6 \square o(n)$

Example:

If the first program takes $100n^2$ milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5n^3/100n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster than those the one with running time $100n^2$. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is $n/20$, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate functions such as $O(n)$ or $O(n \log n)$ are always better.

Example:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i <= n; i++)  
    v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n . We write it as $O(n)$. The actual number of instructions might be $50n$, while the running time might be $17n$ microseconds. It might even be $17n+3$ microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n , the correct notation is $O(n)$ and the code is said to have *linear* running time.

Example:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i <= n; i++)  
    for (j = 1; j <= n; j++)  
        a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes $n \times n = n^2$ times. The assignment statement in the inner loop takes constant time, so the running time of the code is $O(n^2)$ steps. This piece of code is said to have *quadratic* running time.

Example:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two $n \times n$ matrices. The code to compute the matrix product $C = A * B$ is given below.

```

for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        C[i, j] = 0;
        for (k = 1; k <= n; k++)
            C[i, j] = C[i, j] + A[i, k] * B[k, j];

```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes $n * n * n = n^3$ times. The innermost statement, which contains a scalar sum and product takes constant $O(1)$ time. So the algorithm overall takes $O(n^3)$ time.

Example :Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```

for (i = n-1; i > 1; i--)
    for (j = 1; j <= i; j++)
        if (a[j] > a[j+1])
            swap a[j] and a[j+1];

```

This looks like the double. The innermost statement, the *if*, takes $O(1)$ time. It doesn't necessarily take the same time when the condition is true as it does when it is false, but both times are bounded by a constant. But there is an important difference here. The outer loop executes n times, but the inner loop executes a number of times that depends on i . The first time the inner *for* executes, it runs $i = n-1$ times. The second time it runs $n-2$ times, etc. The total number of times the inner *if* statement executes is therefore:

$$(n-1) + (n-2) + \dots + 3 + 2 + 1$$

This is the sum of an arithmetic series.

The value of the sum is $n(n-1)/2$. So the running time of bubble sort is $O(n(n-1)/2)$, which is $O((n^2-n)/2)$. Using the rules for big- O given earlier, this bound simplifies to $O((n^2)/2)$ by ignoring a smaller term, and to $O(n^2)$, by ignoring a constant factor. Thus, bubble sort is an $O(n^2)$ algorithm.

Example :Analysis of binary search

Binary search is a little harder to analyze because it doesn't have a *for* loop. But it's still pretty easy because the search interval halves each time we iterate the search. The sequence of search intervals looks something like this:

$$n, n/2, n/4, \dots, 8, 4, 2, 1$$

It's not obvious how long this sequence is, but if we

take logs, it is: $\log_2 n$, $\log_2 n - 1$, $\log_2 n - 2$, ...,

3, 2, 1, 0

Since the second sequence decrements by 1 each time down to 0, its length must be

$\log_2 n + 1$. It takes only constant time to do each test of binary search, so the total running time is just the number of times that we iterate, which is $\log_2 n + 1$. So binary search is an $O(\log_2 n)$ algorithm. Since the base of the log doesn't matter in an asymptotic bound, we can write that binary search is $O(\log n)$.

General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is 'n' the input size.

1. The running time of each assignment read and write statement can usually be taken to be $O(1)$. (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).
2. The running time of a sequence of statements is determined by the sum rule.
I.e. the running time of the sequence is, to within a constant factor, the largest running time of any statement in the sequence.
3. The running time of an if-statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally $O(1)$ the time for an if-then-else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.
4. The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is $O(1)$). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 6

Unit - 1

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Probabilities Analysis: CO1

Probabilities Analysis:

this analysis uses probability

Example:

A sample space S will for us be some collection on elementary events.

For instance, results of coin flips. Then $S = \{HH, TH, HT, TT\}$.

An event E is any subset of S .

For example, $E = \{TH, HT\}$ be the event of S

A probability distribution $P\{\}$ on S is a mapping from events on S to the real numbers satisfying for any events A and B . A' be the complement of A

(a) $P\{A\} \geq 0$ (b) $P\{S\} = 1$ (c) $P\{A \cup B\} = P\{A\} + P\{B\}$ if $A \cap B = \emptyset$

Result 1 : $P\{S \cup \emptyset\} = P\{S\} + P\{\emptyset\} = 1 + P\{\emptyset\}$. So $P\{\emptyset\} = 0$.

Result 2 : $P\{S\} = P\{A \cup A'\} = P\{A\} + P\{A'\}$. So $P\{A'\} = 1 - P\{A\}$.

Conditional Probability and Independence

The conditional probability of an event A given an event B is defined to be: $P\{A|B\} = P\{A \cap B\} / P\{B\}$.

- Two events are independent if $Pr\{A \cap B\} = Pr\{A\}Pr\{B\}$
 - Given a collection A_1, A_2, \dots, A_k of events we say they are pairwise independent if $Pr\{A_i \cap A_j\} = Pr\{A_i\}Pr\{A_j\}$ for any i and j .
- They are mutually independent if for an subset $A_{i_1}, A_2, \dots, A_{i_m}$ of then $Pr\{A_{i_1} \cap \dots \cap A_{i_m}\} = Pr\{A_{i_1}\} * \dots * Pr\{A_{i_m}\}$

A discrete random variable X is a function from a finite sample space S to the real numbers.

- Given such a function X we can define the probability density function for X as: $f(x) = Pr\{X = x\}$ where the little x is a real number.

The expected value of a random variable X is defined to be:

- The variance of X , $\text{Var}[X]$ is defined to be: $E[(X - E(X))^2] = E[X^2] - (E[X])^2$
- The standard deviation of X , σ_X , is defined to be the $(\text{Var}[X])^{1/2}$.

Indicator Random Variables

- In order to analyze the hiring problem we need a convenient way to convert between probabilities and expectations.
- We will use indicator random variables to help us do this.
- Given a sample space S and an event A . Then the indicator random variable $I\{A\}$ associated with event A is defined as:

$$I(A) = \begin{cases} 1 & \text{if } A \text{ occur} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

Example:

Suppose our sample space $S = \{H, T\}$ with $P\{H\} = P\{T\} = 1/2$.

We can define an indicator random variable X_H associated with the coin coming up heads:

$$X_H = I(H) = \begin{cases} 1 & \text{if } H \text{ occur} \\ 0 & \text{if } T \text{ occur} \end{cases}$$

The expected number of heads in one coin flip is then

$$\begin{aligned} E[X_H] &= P(H) \cdot I(H) + P(T) \cdot I(T) \\ &= \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = \frac{1}{2}. \end{aligned}$$

Lemma 1: Given a sample space S and an event A in S , let $X_A = I\{A\}$. Then $E[X_A] = P\{A\}$.

Proof: $E[X_A] = E[I\{A\}]$

$$\begin{aligned} &= 1 \cdot P\{A\} + 0 \cdot P\{A^c\} \\ &= P\{A\}. \end{aligned}$$

Indicator random variables are more useful if we are dealing with more than one coin flip.

Let X_i be the indicator that indicates whether the result of the i th coin flip was a head.

Consider the random variable: $X = \sum_{i=1}^n X_i$

Then the expected number of head in n tosses is

$$E[X] = E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n \frac{1}{2} = n/2$$

The Hiring Problem

We will now begin our investigation of randomized algorithms with a toy problem:

- You want to hire an office assistant from an employment agency.
- You want to interview candidates and determine if they are better than the current assistant and if so replace the current assistant.

- You are going to eventually interview every candidate from a pool of n candidates.
- You want to always have the best person for this job, so you will replace an assistant with a better one as soon as you are done the interview.
- However, there is a cost to fire and then hire someone.
- You want to know the expected price of following this strategy until all n candidates have been interviewed.

Algorithm Hire-Assistant(n)

```

{
  best := dummy candidate;
  for i := 1 to n do
  {
    interview of candidate i ;
    if (candidate i is better than best) then
    {
      best := i;
      hire candidate i;
    }
  }
}

```

- Interviewing has a low cost c_i .
- Hiring has a high cost c_h .
- Let n be the number of candidates to be interviewed and let m be the number of people hired.
- The total cost then goes as $O(n \cdot c_i + m \cdot c_h)$
- The number of candidates is fixed so the part of the algorithm we want to focus on is the $m \cdot c_h$ term.
- This term governs the cost of hiring.

Worst-case Analysis

- In the worst case, everyone we interview turns out to be better than the person we currently have.
- In this case, the hiring cost for the algorithm will be $O(n \cdot c_h)$.
- This bad situation presumably doesn't typically happen so it is interesting to ask what happens in the average case.

Analysis of the Hiring Problem

- Let X_i be the indicator random variable which is 1 if candidate i is hired and 0 otherwise.
- Let
- By our lemma $E[X_i] = \Pr\{\text{candidate } i \text{ is hired}\}$
- Candidate i will be hired if i is better than each of candidates 1 through $i-1$.
- As each candidate arrives in random order, any one of the first candidate i is equally likely to be the best candidate so far. So $E[X_i] = 1/i$.

$$E[X] = E\left[\sum_{i=1}^n x_i\right] = \sum_{i=1}^n E[x_i] = \sum_{i=1}^n \frac{1}{i} = \ln(n) + O(1)$$

Assume that the candidates are presented in random order, then algorithm Hire-Assistant has a hiring cost of $O(ch \cdot \ln n)$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 7

Unit - 1

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr.N. Subhash Chandra**, Professor of CSE

Amortized Analysis: CO1

What is Amortized Analysis ?

- In amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed.
- ▷ No involvement of probability
- ▷ Average performance on a sequence of operations, even some operation is expensive.
- ▷ Guarantee average performance of each operation among the sequence in worst case.
- ▷ Methods of Amortized Analysis

Aggregate Method: we determine an upper bound $T(n)$ on the total sequence of n operations. The cost of each will then be $T(n)/n$.

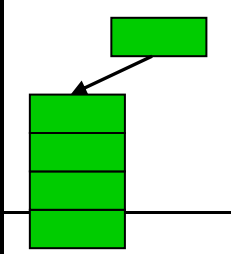
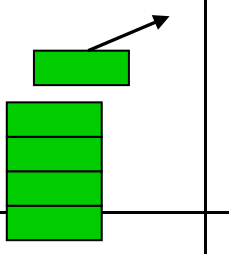
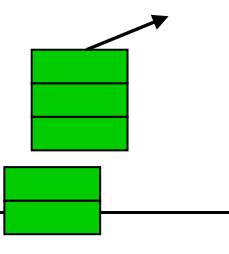
Accounting Method: we overcharge some operations early and use them to as prepaid charge later.

Potential Method: we maintain credit as potential energy associated with the structure as a whole.

1. Aggregate Method

- Show that for all n , a sequence of n operations take worst-case time $T(n)$ in total
- In the worst case, the average cost, or amortized cost, per operation is $T(n)/n$.
- The amortized cost applies to each operation, even when there are several types of operations in the sequence.

Aggregate Analysis: Stack Example

3 ops:			
	Push(S,x)	Pop(S)	Multi-pop(S,k)
Worst-case cost:	$O(1)$	$O(1)$	$O(\min(S ,k)) = O(n)$

- Sequence of n push, pop, Multipop operations
 - ❑ Worst-case cost of Multipop is $O(n)$
 - ❑ Have n operations
 - ❑ Therefore, worst-case cost of sequence is $O(n^2)$
- Observations
 - ❑ Each object can be popped only once per time that it's pushed
 - ❑ Have $\leq n$ pushes $\Rightarrow \leq n$ pops, including those in Multipop
 - ❑ Therefore total cost = $O(n)$
 - ❑ Average over n operations $\Rightarrow O(1)$ per operation on average
- Notice that no probability involved

2. Accounting Method

Charge i th operation a fictitious amortized cost \hat{c}_i , where \$1 pays for 1 unit of work (i.e., time).

- ❑ Assign different charges (amortized cost) to different operations
 - Some are charged more than actual cost
 - Some are charged less
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the bank for use by subsequent operations.
- The bank balance (the credit) must not go negative!

We must ensure that

for all n .

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

- Thus, the total amortized costs provide an upper bound on the total true costs.

3 ops:			
	Push(S,x)	Pop(S)	Multi-pop(S,k)
	•Assigned cost:	2	0
•Actual cost:	1	1	min(S ,k)

- When pushing an object, pay \$2
 - \$1 pays for the push
 - \$1 is prepayment for it being popped by either pop or Multipop
 - Since each object has \$1, which is credit, the credit can never go negative
 - Therefore, total amortized cost = $O(n)$, is an upper bound on total actual cost

Accounting Method: Binary Counter

- k-bit Binary Counter: $A[0..k-1]$

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

INCREMENT(A)

- $i \leftarrow 0$
- while $i < \text{length}[A]$ and $A[i] = 1$
- do $A[i] \leftarrow 0$ \triangleright reset a bit
- $i \leftarrow i + 1$
- if $i < \text{length}[A]$
- then $A[i] \leftarrow 1$ \triangleright set a bit

Consider a sequence of n increments. The worst-case time to execute one increment is $Q(k)$. Therefore, the worst-case time for n increments is $n \cdot Q(k) = Q(n \cdot k)$.

WRONG! In fact, the worst-case cost for n increments is only $Q(n) \ll Q(n \cdot k)$.

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	Cost
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	3
3	0	0	0	1	1	4
4	0	0	1	0	0	7
5	0	0	1	0	1	8
6	0	0	1	1	0	10
7	0	0	1	1	1	11
8	0	1	0	0	0	15
9	0	1	0	0	1	16
10	0	1	0	1	0	18
11	0	1	0	1	1	19

Total cost of n operations

A[0] flipped every op n

A[1] flipped every 2 ops $n/2$

A[2] flipped every 4 ops $n/2^2$

A[3] flipped every 8 ops $n/2^3$

...

A[i] flipped every 2^i ops $n/2^i$

Cost of n increments

$$= \sum_{i=1}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor$$

$$< n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$$

$$= \Theta(n)$$

Thus, the average cost of each increment operation is $Q(n)/n = Q(1)$.

3. Potential Method

IDEA: View the bank account as the potential energy (as in physics) of the dynamic set.

FRAMEWORK:

- Start with an initial data structure D_0 .
- Operation i transforms D_{i-1} to D_i .
- The cost of operation i is c_i .
- Define a *potential function* $F : \{D_i\} \rightarrow \mathbb{R}$, such that $F(D_0) = 0$ and $F(D_i) \geq 0$ for all i .
- The *amortized cost* \hat{c}_i with respect to F is defined to be $\hat{c}_i = c_i + F(D_i) - F(D_{i-1})$.
- Like the accounting method, but think of the credit as *potential* stored with the *entire data structure*.
 - Accounting method stores credit with specific objects while potential method stores potential in the data structure as a whole.
 - Can release potential to pay for future operations
- Most flexible of the amortized analysis methods).
- $\hat{c}_i = c_i + F(D_i) - F(D_{i-1})$
- If $DF_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $DF_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i .

The total amortized cost of n operations is

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides telescopically

$$= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^n c_i \text{ since } \Phi(D_n) \geq 0 \text{ and } \Phi(D_0) = 0.$$

Stack Example

Define: $\phi(D_i) = \# \text{items in stack}$ Thus, $\phi(D_0) = 0$.

Plug in for operations:

$$\begin{aligned} \text{Push: } \hat{c}_i &= c_i + \phi(D_i) - \phi(D_{i-1}) \\ &= 1 + j - (j-1) \end{aligned}$$

$$= 2$$

$$\begin{aligned} \text{Pop: } \hat{c}_i &= c_i + \phi(D_i) - \phi(D_{i-1}) \\ &= 1 + (j-1) - j \\ &= 0 \end{aligned}$$

$$\begin{aligned} \text{Multi-pop: } \hat{c}_i &= c_i + \phi(D_i) - \phi(D_{i-1}) \\ &= k' + (j-k') - j \quad k' = \min(|S|, k) \\ &= 0 \end{aligned}$$

Potential Method: Binary Counter

Define the potential of the counter after the i^{th} operation by $F(D_i) = b_i$, the number of 1's in the counter after the i^{th} operation.

Note:

- $F(D_0) = 0$,
- $F(D_i) \geq 0$ for all i .

Example

```
0 0 0 1 0 1 0
0 0 0 1$1 0 1$1 0
```

Assume i^{th} INCREMENT resets t_i bits (in line 3).

Actual cost $c_i = (t_i + 1)$

Number of 1's after i^{th} operation: $b_i = b_{i-1} - t_i + 1$

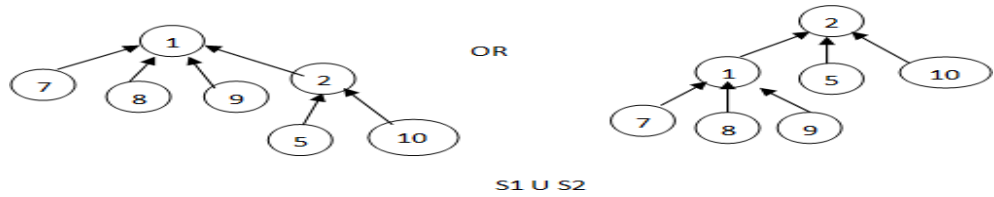
The amortized cost of the i^{th} INCREMENT is

$$\begin{aligned} \hat{c}_i &= c_i + F(D_i) - F(D_{i-1}) \\ &= (t_i + 1) + (1 - t_i) \\ &= 2 \end{aligned}$$

Therefore, n INCREMENTS cost $Q(n)$ in the worst case

Disjoint Union:

To perform disjoint set union between two sets S_i and S_j can take any one root and make it sub-tree of the other. Consider the above example sets S_1 and S_2 then the union of S_1 and S_2 can be represented as any one of the following.

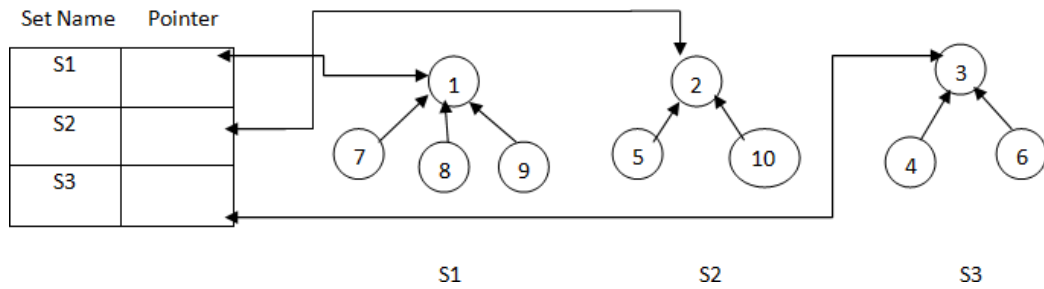


Find:

To perform find operation, along with the tree structure we need to maintain

mai

the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout - 2

Unit - II

Year and Semester: IIyr &II Sem

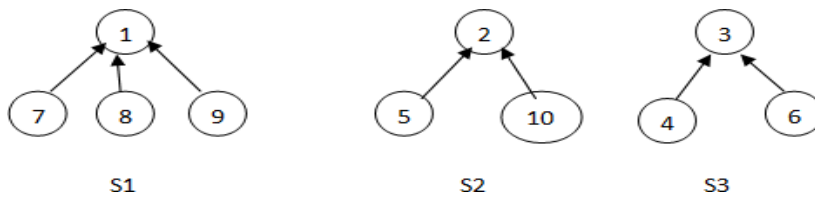
Subject: **Design and Analysis of Algorithms**

Union and Find Algorithms: CO2

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



<i>i</i>	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
<i>p</i>	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

Algorithm SimpleUnion(i,j)

```
{
    P[i]=j;
}
```

Algorithm for find operation:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

Algorithms SimpleFind(i)

```
{
    while( P[i]≥0)
    do i:=P[i];
    return i;
}
```

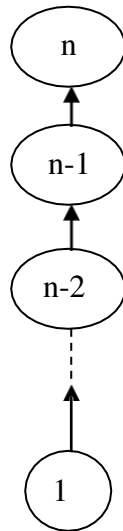
Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(

The sequence of Union operations results the degenerate tree as below.



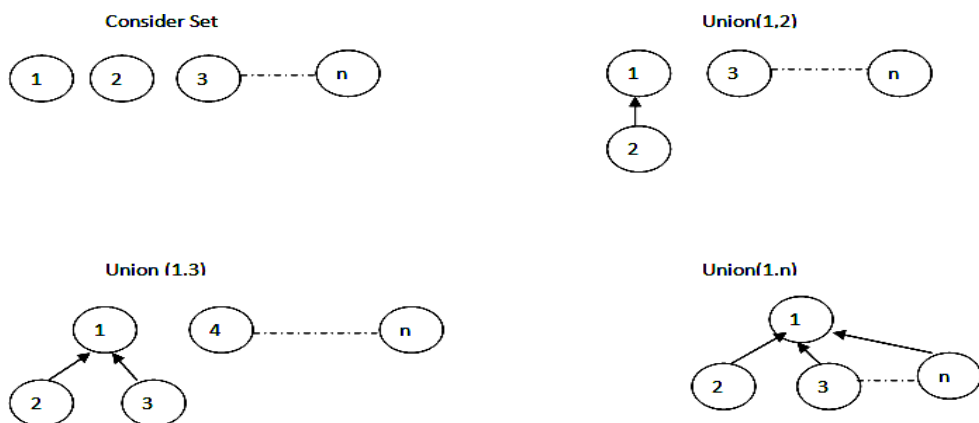
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time $O(n)$. And for the sequence of Find operations it will take time

complexity of $O\left(\sum_{i=1}^n i\right) = O(n^2)$.

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

Weighting rule for Union:

If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.



To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i.

Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

```

Algorithm WeightedUnion(i,j)
//Union sets with roots i and j , i≠j using the weighted rule
// P[i]=-count[i] and p[j]=-count[j]
    
```

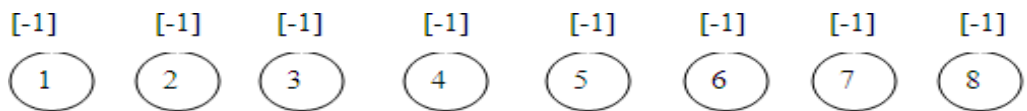
```

{
  temp:= P[i]+P[j];
  if (P[i]>P[j]) then
  { // i has fewer nodes P[i]:=j;
    P[j]:=temp;
  }
  else
  { // j has fewer nodes P[j]:=i;
    P[i]:=temp;
  }
}
}

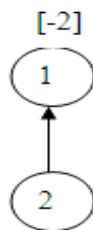
```

Collapsing rule for find:

If j is a node on the path from i to its root and $p[i] \neq \text{root}[i]$, then set $P[j]$ to $\text{root}[i]$. Consider the tree created by `WeightedUnion()` on the sequence of $1 \leq i \leq 8$. `Union(1,2)`, `Union(3,4)`, `Union(5,6)` and `Union(7,8)`



Union(1,2)



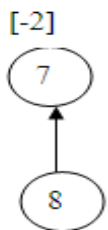
Union(3,4)



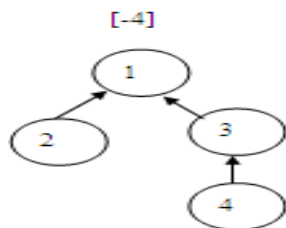
Union(5,6)



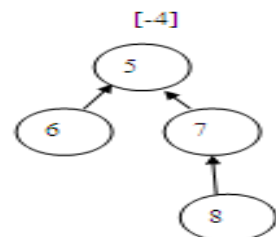
Union(7,8)



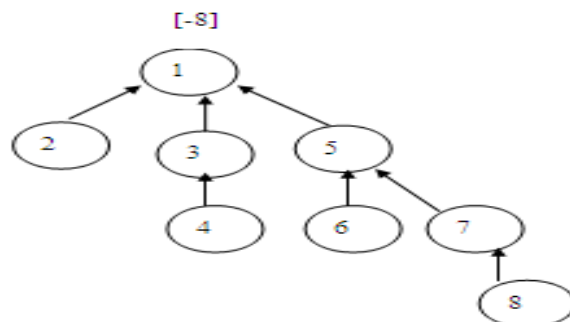
Union(1,3)



Union(5,7)



Union(1,5)



Now process the following eight find

operations Find(8),

Find(8).....Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .

When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

Algorithm CollapsingFind(i)

// Find the root of the tree containing element i

// use the collapsing rule to collapse all nodes from i to root.

```
{
    r:=i;
    while(P[r]>0) do
        r:=P[r]; //Find root
        while(i≠r)
        {
            //reset the parent node from element i
            to the root s:=P[i];
            P[i]:=r;
            i:=s;
        }
}
```



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - II

Year and Semester: IYr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Efficient non-recursive binary tree traversal Algorithms: CO2

Search means finding a path or traversal between a start node and one of a set of goal nodes. Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.

Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree:

1. Preorder
2. Inorder
3. postorder

In all the three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among the three orders comes from the difference in the time at which a node is visited.

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for preorder traversal is as follows:

```
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);
    }
}
```

Preorder Traversal:

In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

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);
    }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for preorder traversal is as follows:

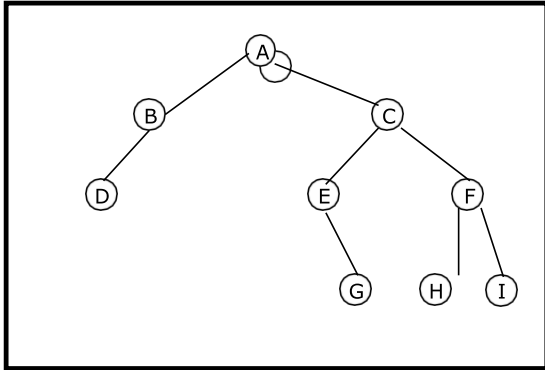
Algorithm Postorder (t)

```
// 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);
    }
}
```


Examples for binary tree traversal/search technique:

Example 1:

Traverse the following binary tree in pre, post and in-order.



Binary Tree

Preordering of the vertices:
A, B, D, C, E, G, F, H, I.

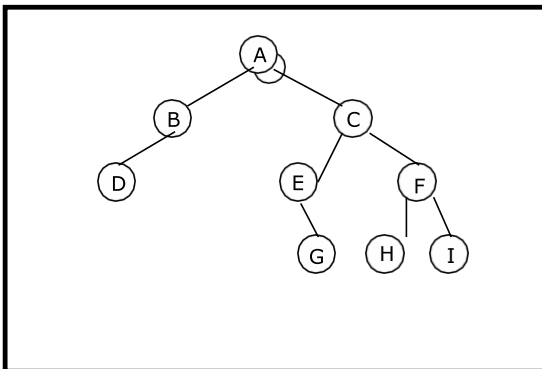
Postordering of the vertices:
D, B, G, E, H, I, F, C, A.

Inordering of the
vertices: D, B, A, E, G,
C, H, F, I

Pre, Post and In-order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



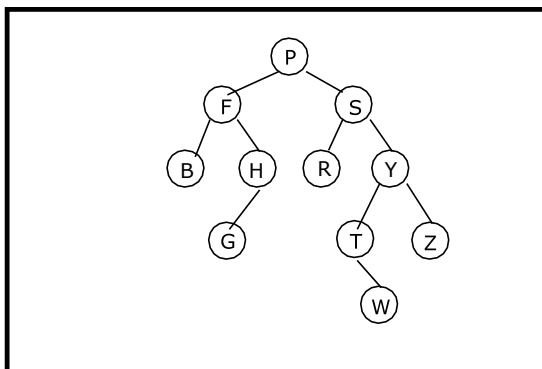
Binary Tree

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Level order traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



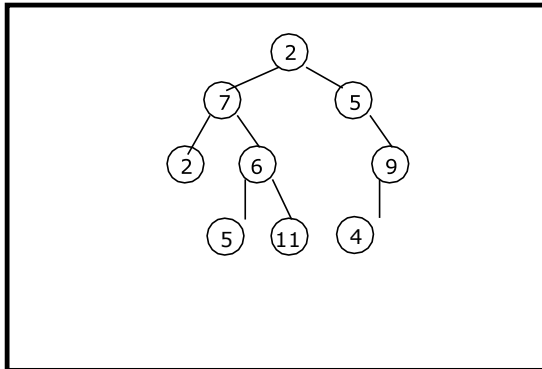
Binary Tree

- Preorder traversal yields:
P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



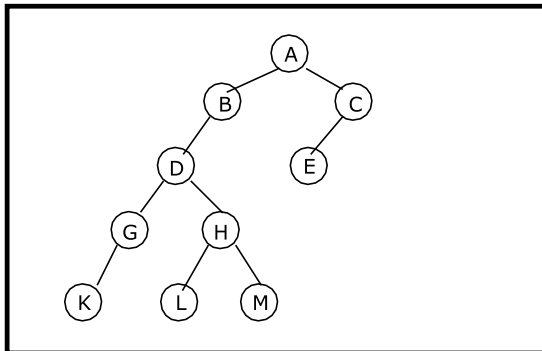
Binary Tree

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9
- Level order traversal yields:
2, 7, 5, 2, 6, 9, 5, 11, 4

Pre, Post, Inorder and level order Traversal

Example 5:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C
- Level order traversal yields:
A, B, C, D, E, G, H, K, L, M

Pre, Post, Inorder and level order Traversal

Non Recursive Binary Tree Traversal Algorithms:

At first glance, it appears we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

The algorithm for inorder Non Recursive traversal is as follows:

```
Algorithm inorder()
{
    stack[1] = 0
    vertex = root
top:  while(vertex ≠ 0)
    {
        push the vertex into the stack
        vertex = leftson(vertex)
    }

    pop the element from the stack and make it as vertex

    while(vertex ≠ 0)
    {
        print the vertex node
        if(rightson(vertex) ≠ 0)
        {
            vertex = rightson(vertex)
            goto top
        }
        pop the element from the stack and made it as vertex
    }
}
```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

```
Algorithm preorder( )
{
    stack[1]: = 0
    vertex := root.
    while(vertex ≠ 0)
    {
        print vertex node
        if(rightson(vertex) ≠ 0)
            push the right son of vertex into the stack.
        if(leftson(vertex) ≠ 0)
            vertex := leftson(vertex)
        else
            pop the element from the stack and made it as vertex
    }
}
```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push $-(\text{right son of vertex})$ onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

The algorithm for postorder Non Recursive traversal is as follows:

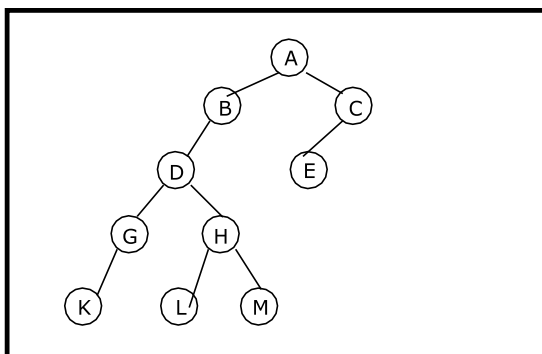
Algorithm **postorder**()

```
{
    stack[1] := 0
    vertex := root

top: while(vertex ≠ 0)
    {
        push vertex onto stack
        if(rightson(vertex) ≠ 0)
            push -leftson(vertex) onto stack
        vertex := leftson(vertex)
    }
    pop from stack and make it as vertex
    while(vertex > 0)
    {
        print the vertex node
        pop from stack and make it as vertex
    }
    if(vertex < 0)
    {
        vertex := -(vertex)
        goto top
    }
}
```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
H	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H
	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0		Stop since stack is empty

Preorder Traversal:

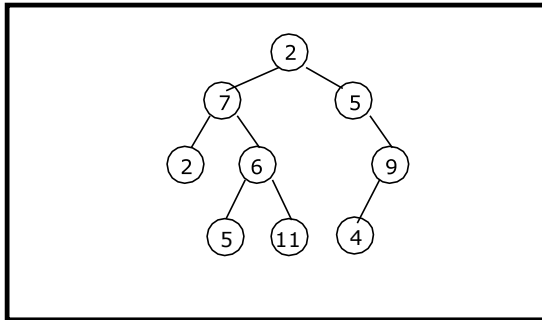
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex \neq 0 then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder traversal yields:
2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2, 7, 5, 6, 11, 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
11	0 2	2 7 5 6 11	
5	0 5	2 7 5 6 11 2	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex \neq 0 then return to step one otherwise exit.

Current vertex	Stack	Processed nodes	Remarks
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11	
9	0	2 7 2 6 5 11 5	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 5

Unit - II

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

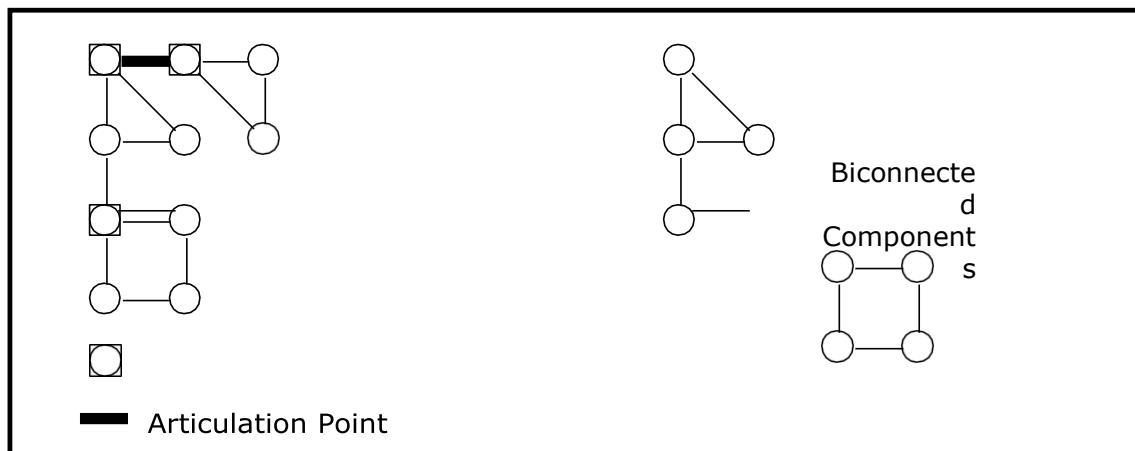
Articulation Points and Biconnected Components: CO2

Let $G = (V, E)$ be a connected undirected graph. Consider the following definitions:

Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components. This is illustrated in the following figure:



Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected.

Let us consider the typical case of vertex v , where v is not a leaf and v is not the root. Let w_1, w_2, \dots, w_k be the children of v . For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of v , then if we remove v , this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point.

On the other hand, if every one of the subtree rooted at the children of v have back edges to proper ancestors of v , then if v is removed, the graph remains connected (the back edges hold everything together). This leads to the following:

Observation 1: An internal vertex v of the DFS tree (other than the root) is

an articulation point if and only if there is a subtree rooted at a child of v such that there is no back edge from any vertex in this subtree to a proper ancestor of v .

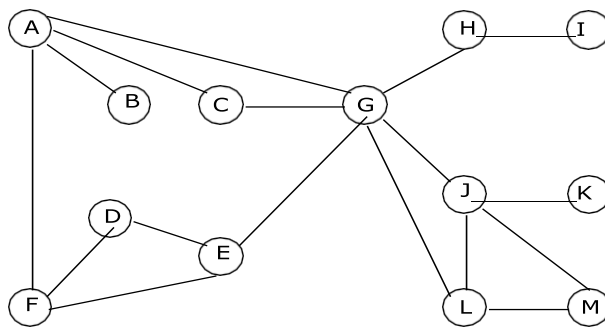
Observation 2: A leaf of the DFS tree is never an articulation point, since a leaf will not have any subtrees in the DFS tree.

Thus, after deletion of a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children. If the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

Articulation Points by Depth First Search:

Determining the articulation turns out to be a simple extension of depth firstsearch. Consider a depth first spanning tree for this graph.

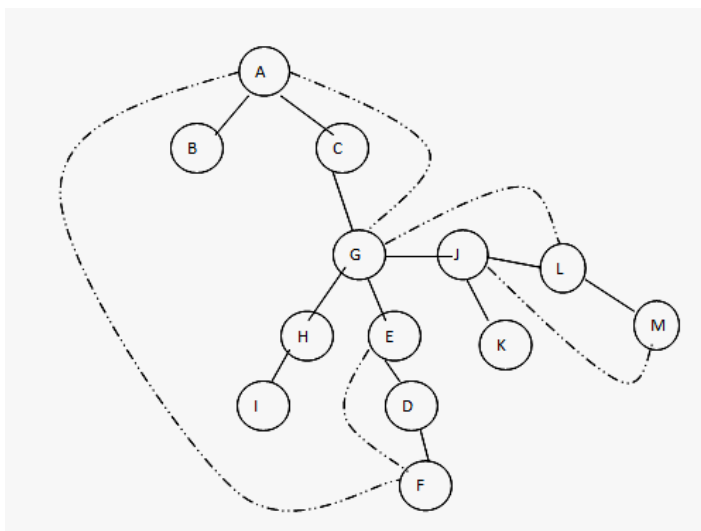


Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points.

Deleting node E does not disconnect the graph because G and D both have dotted links (back edges) that point above E, giving alternate paths from them to F. On the other hand, deleting G does disconnect the graph because there are no such alternate paths from L or H to E (G's parent).

A vertex 'x' is not an articulation point if every child 'y' has some node lower in the tree connect (via a dotted link) to a node higher in the tree than 'x', thus providing an alternate connection from 'x' to 'y'. This rule will not work for the root node since there are no *nodes higher in the tree*. The root is an articulation point if it has two or more children.

Depth First Spanning Tree for the above graph is:



By using the above observations the articulation points of this graph are:

- A : because it connects B to the rest of the graph.
- H : because it connects I to the rest of the graph.
- J : because it connects K to the rest of the graph.
- G : because the graph would fall into three pieces if G is deleted.

Biconnected components are: {A, C, G, D, E, F}, {G, J, L, M}, B, H, I and K

This observation leads to a simple rule to identify articulation points. For each u define $L(u)$ as follows:

$$L(u) = \min \{ \text{DFN}(u), \min \{ L(w) \mid w \text{ is a child of } u \}, \min \{ \text{DFN}(w) \mid (u, w) \text{ is a back edge} \} \}.$$

$L(u)$ is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge. It follows that, If u is not the root then u is an articulation point iff u has a child w such that:

$$L(w) \geq \text{DFN}(u)$$

6.6.2. Algorithm for finding the Articulation points:

Pseudocode to compute DFN and L.

Algorithm Art (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initialized to zero and that // the
// global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn [u] := num; L [u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if (dfn [w] = 0) then
        {
            Art (w, u); // w is unvisited.
            L [u] := min (L [u], L [w]);
        }
        else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
    }
}
```

6.6.1. Algorithm for finding the Biconnected Components:

Algorithm BiComp (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initially zero and that the
// global variable num is initialized to 1. n is the number of vertices in G.
{
    dfn [u] := num; L [u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if ((v ≠ w) and (dfn [w] ≤ dfn [u])) then
```

```

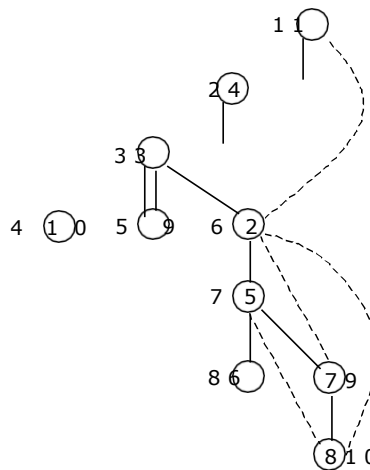
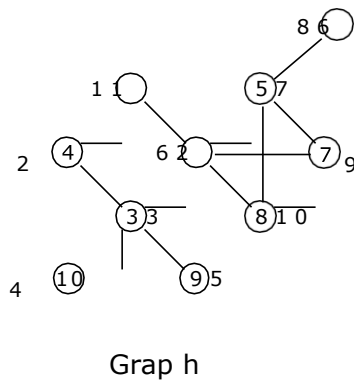
    add (u, w) to the top of a stack s;
    if (dfn [w] = 0) then
    {
        if (L [w] ≥ dfn [u]) then
        {
            write ("New bicomponent");
            repeat
            {
                Delete an edge from the top of stack s;
                Let this edge be (x, y);
                Write (x, y);
            } until ((x, y) = (u, w) or ((x, y) = (w, u)));
        }

        BiComp (w, u); // w is unvisited.
        L [u] := min (L [u], L [w]);
    }
    else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
}
}

```

6.7.1. Example:

For the following graph identify the articulation points and Biconnected components:



To identify the articulation points, we use:

$$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$$

$$L(1) = \min \{DFN(1), \min \{L(4)\}\} = \min \{1, L(4)\} = \min \{1, 1\} = 1$$

$$L(4) = \min \{DFN(4), \min \{L(3)\}\} = \min \{2, L(3)\} = \min \{2, 1\} = 1$$

$$L(3) = \min \{DFN(3), \min \{L(10), L(9), L(2)\}\} = \min \{3, \min \{L(10), L(9), L(2)\}\} = \min \{3, \min \{4, 5, 1\}\} = 1$$

$$L(10) = \min \{DFN(10)\} = 4$$

$$L(9) = \min \{DFN(9)\} = 5$$

$$L(2) = \min \{DFN(2), \min \{L(5)\}, \min \{DFN(1)\}\} \\ = \min \{6, \min \{L(5)\}, 1\} = \min \{6, 6, 1\} = 1$$

$$L(5) = \min \{DFN(5), \min \{L(6), L(7)\}\} = \min \{7, 8, 6\} = 6$$

$$L(6) = \min \{DFN(6)\} = 8$$

$$L(7) = \min \{DFN(7), \min \{L(8), \min \{DFN(2)\}\}\} \\ = \min \{9, L(8), 6\} = \min \{9, 6, 6\} = 6$$

$$L(8) = \min \{DFN(8), \min \{DFN(5), DFN(2)\}\} \\ = \min \{10, \min \{7, 6\}\} = \min \{10, 6\} = 6$$

Therefore, $L(1:10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)$

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has $L(5) = 6$ and $DFN(2) = 6$,
So, the condition $L(5) = DFN(2)$ is true.

Vertex 3: is an articulation point as child 10 has $L(10) = 4$ and $DFN(3) = 3$,
So, the condition $L(10) > DFN(3)$ is true.

Vertex 4: is not an articulation point as child 3 has $L(3) = 1$ and $DFN(4) = 2$,
So, the condition $L(3) \geq DFN(4)$ is false.

Vertex 5: is an articulation point as child 6 has $L(6) = 8$, and $DFN(5) = 7$,
So, the condition $L(6) > DFN(5)$ is true.

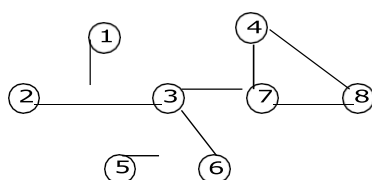
Vertex 7: is not an articulation point as child 8 has $L(8) = 6$, and $DFN(7) = 9$,
So, the condition $L(8) \geq DFN(7)$ is false.

Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

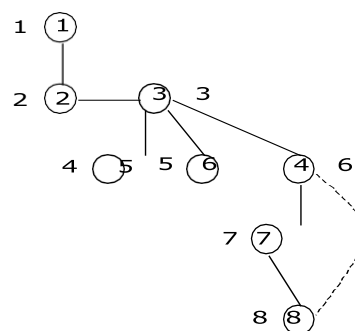
Therefore, the articulation points are $\{2, 3, 5\}$.

Example:

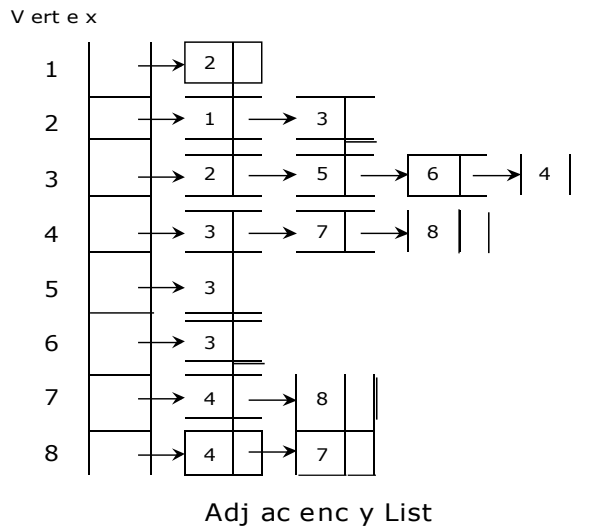
For the following graph identify the articulation points and Biconnected components:



Graph



DFS spanning Tree



$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$

$$L(1) = \min \{DFN(1), \min \{L(2)\}\} = \min \{1, L(2)\} = \min \{1, 2\} = 1$$

$$L(2) = \min \{DFN(2), \min \{L(3)\}\} = \min \{2, L(3)\} = \min \{2, 3\} = 2$$

$$L(3) = \min \{DFN(3), \min \{L(4), L(5), L(6)\}\} = \min \{3, \min \{6, 4, 5\}\} = 3$$

$$L(4) = \min \{DFN(4), \min \{L(7)\}\} = \min \{6, L(7)\} = \min \{6, 6\} = 6$$

$$L(5) = \min \{DFN(5)\} = 4$$

$$L(6) = \min \{DFN(6)\} = 5$$

$$L(7) = \min \{DFN(7), \min \{L(8)\}\} = \min \{7, 6\} = 6$$

$$L(8) = \min \{DFN(8), \min \{DFN(4)\}\} = \min \{8, 6\} = 6$$

Therefore, $L(1:8) = \{1, 2, 3, 6, 4, 5, 6, 6\}$

Finding the Articulation Points:

Check for the condition if $L(w) \geq DFN(u)$ is true, where w is any child of u .

Vertex 1: Vertex 1 is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as $L(3) = 3$ and $DFN(2) = 2$.

So, the condition is true

Vertex 3: is an articulation Point as:

- I. $L(5) = 4$ and $DFN(3) = 3$
- II. $L(6) = 5$ and $DFN(3) = 3$ and
- III. $L(4) = 6$ and $DFN(3) = 3$

So, the condition true in above cases

Vertex 4: is an articulation point as $L(7) = 6$ and $DFN(4) = 6$.
So, the condition is true

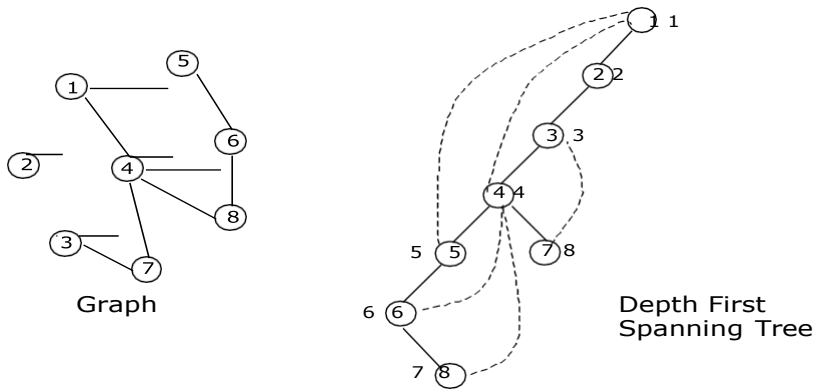
Vertex 7: is not an articulation point as $L(8) = 6$ and $DFN(7) = 7$.
So, the condition is False

Vertex 5, Vertex 6 and Vertex 8 are leaf nodes.

Therefore, the articulation points are $\{2, 3, 4\}$.

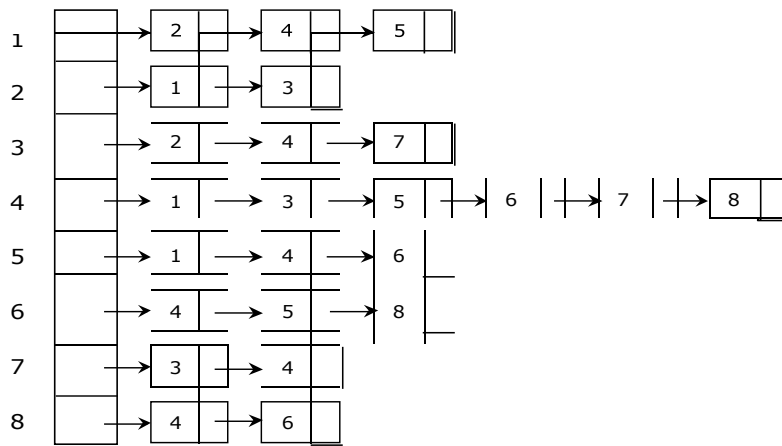
Example:

For the following graph identify the articulation points and Biconnected components:



$$DFN(1: 8) = \{1, 2, 3, 4, 5, 6, 8, 7\}$$

Vertex



Adj ac enc y List

$$L(u) = \min \{DFN(u), \min \{L(w) \mid w \text{ is a child of } u\}, \min \{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$$

$$L(1) = \min \{DFN(1), \min \{L(2)\}\} = \min \{1, L(2)\} = 1$$

$$L(2) = \min \{DFN(2), \min \{L(3)\}\} = \min \{2, L(3)\} = \min\{2, 1\} = 1$$

$$L(3) = \min \{DFN(3), \min \{L(4)\}\} = \min \{3, L(4)\} = \min \{3, L(4)\} \\ = \min \{3, 1\} = 1$$

$$L(4) = \min \{DFN(4), \min \{L(5), L(7)\}, \min \{DFN(1)\}\} \\ = \min \{4, \min \{L(5), L(7)\}, 1\} = \min \{4, \min \{1, 3\}, 1\} \\ = \min \{4, 1, 1\} = 1$$

$$L(5) = \min \{DFN(5), \min \{L(6)\}, \min \{DFN(1)\}\} = \min \{5, L(6), 1\} \\ = \min \{5, 4, 1\} = 1$$

$$L(6) = \min \{DFN(6), \min \{L(8)\}, \min \{DFN(4)\}\} = \min \{6, L(8), 4\} \\ = \min \{6, 4, 4\} = 4$$

$$L(7) = \min \{DFN(7), \min \{DFN(3)\}\} = \min \{8, 3\} = 3$$

$$L(8) = \min \{DFN(8), \min \{DFN(4)\}\} = \min \{7, 4\} = 4$$

Therefore, $L(1:8) = \{1, 1, 1, 1, 1, 4, 3, 4\}$

Finding the Articulation Points:

Check for the condition if $L(w) \geq DFN(u)$ is true, where w is any child of u .

Vertex 1: is not an articulation point.

It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is not an articulation point. As $L(3) = 1$ and $DFN(2) = 2$.
So, the condition is False.

Vertex 3: is not an articulation Point as $L(4) = 1$ and $DFN(3) = 3$.
So, the condition is False.

Vertex 4: is not an articulation Point as:
 $L(3) = 1$ and $DFN(2) = 2$ and
 $L(7) = 3$ and $DFN(4) = 4$
So, the condition fails in both cases.

Vertex 5: is not an Articulation Point as $L(6) = 4$ and $DFN(5) = 6$.
So, the condition is False

Vertex 6: is not an Articulation Point as $L(8) = 4$ and $DFN(6) = 7$.
So, the condition is False

Vertex 7: is a leaf node.

Vertex 8: is a leaf node.

So they are no articulation points.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 6

Unit - II

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

AND/OR GRAPH: CO2

And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of:

N, a set of nodes,

H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.

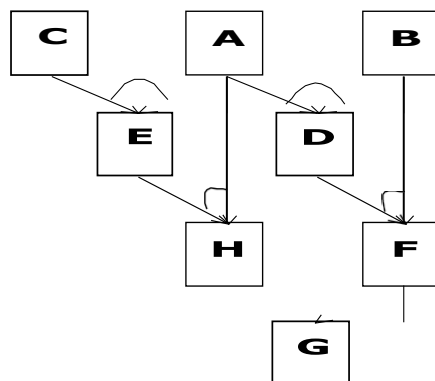
An ordinary graph is a special case of hypergraph in which all the sets of descendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If $K = 1$, the descendent may be thought of as an OR nodes. If $K > 1$, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link.

Example 1:

Draw an And/Or graph for the following prepositions:

1. A
2. B
3. C
4. $A \wedge B \rightarrow D$
5. $A \wedge C \rightarrow E$
6. $B \wedge D \rightarrow F$
7. $F \rightarrow G$
8. $A \wedge E \rightarrow H$





CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Divide and Conquer General Method: CO3

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from $O(n^2)$ to $O(n \log n)$ to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

- Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.
- Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

DANDC (P)

```
{
    if SMALL (P) then
        return S (p); else
        {
            divide p into smaller instances  $p_1, p_2, \dots$ 
             $p_k, k \geq 1$ ; apply DANDC to each of these
            sub problems;
            return (COMBINE (DANDC ( $p_1$ ) , DANDC ( $p_2$ ),..., DANDC ( $p_k$ )));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub

problems. These sub problems p_1, p_2, \dots, p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, $T(n)$ is the time for DANDC on 'n' inputs
 $g(n)$ is the time to complete the answer directly for small inputs and $f(n)$ is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[\text{mid}]$, and compare 'x' with $a[\text{mid}]$. If $x = a[\text{mid}]$ then the desired record has been found. If $x < a[\text{mid}]$ then 'x' must be in that portion of the file that precedes $a[\text{mid}]$, if there at all. Similarly, if $a[\text{mid}] > x$, then further search is only necessary in that part of the file which follows $a[\text{mid}]$. If we use recursive procedure of finding the middle key $a[\text{mid}]$ of the un-searched portion of a file, then every un-successful comparison of 'x' with $a[\text{mid}]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and $a[\text{mid}]$, and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

BINSRCH (a, n, x)

```
// array a(1 : n) of elements in increasing order, n ≥ 0,
// determine whether 'x' is present, and if so, set j such that x = a(j)
// else return j
```

```
{
    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;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 2

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Merge Sort: C03

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge sort in the *best case*, *worst case* and *average case* is $O(n \log n)$ and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr*, *b ptr* and *c ptr*, which are initially set to the beginning of their respective arrays. The smaller of $a[a\ ptr]$ and $b[b\ ptr]$ is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
<i>h ptr</i>			

5	6	7	8
2	15	27	28
<i>j ptr</i>			

1	2	3	4	5	6	7	8
1							
<i>i ptr</i>							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h ptr</i>		

5	6	7	8
2	15	27	28
<i>j ptr</i>			

1	2	3	4	5	6	7	8
1	2						
	<i>i ptr</i>						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
	<i>h ptr</i>		

5	6	7	8
2	15	27	28
	<i>j ptr</i>		

1	2	3	4	5	6	7	8
1	2	13					
		<i>i ptr</i>					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h ptr</i>	

5	6	7	8
2	15	27	28
	<i>j ptr</i>		

1	2	3	4	5	6	7	8
1	2	13	15				
			<i>i ptr</i>				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		<i>h ptr</i>	

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24			
				<i>i ptr</i>			

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
			<i>h ptr</i>

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26		
					<i>i ptr</i>		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	2	3	4
1	13	24	26
			<i>h ptr</i>

5	6	7	8
2	15	27	28
		<i>j ptr</i>	

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28
							<i>i ptr</i>

Algorithm

```

Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := |(low + high)/2|           //finds where to split the set
        MERGESORT(low, mid)              //sort one subset
        MERGESORT(mid+1, high)           //sort the other subset
        MERGE(low, mid, high)           // combine the results
    }
}

```

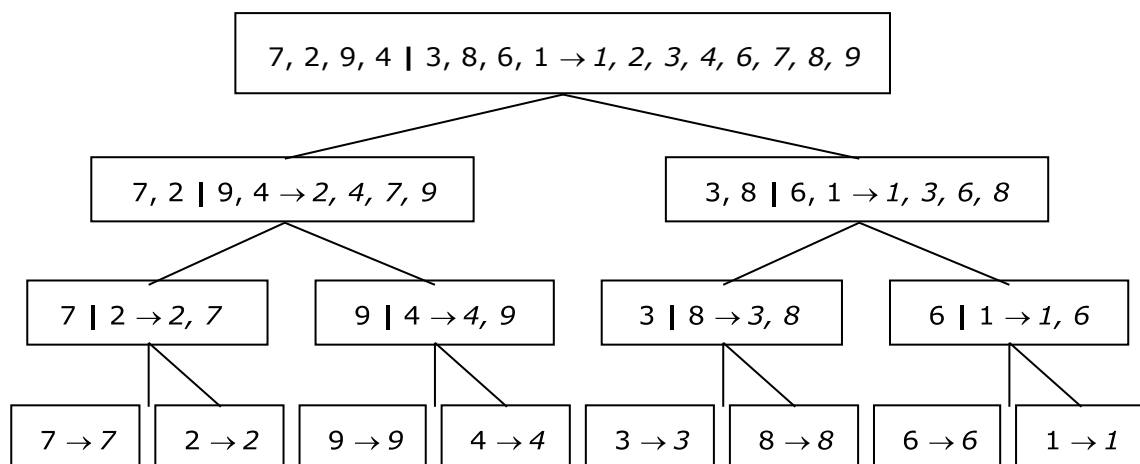
```

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 objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
    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];
}

```

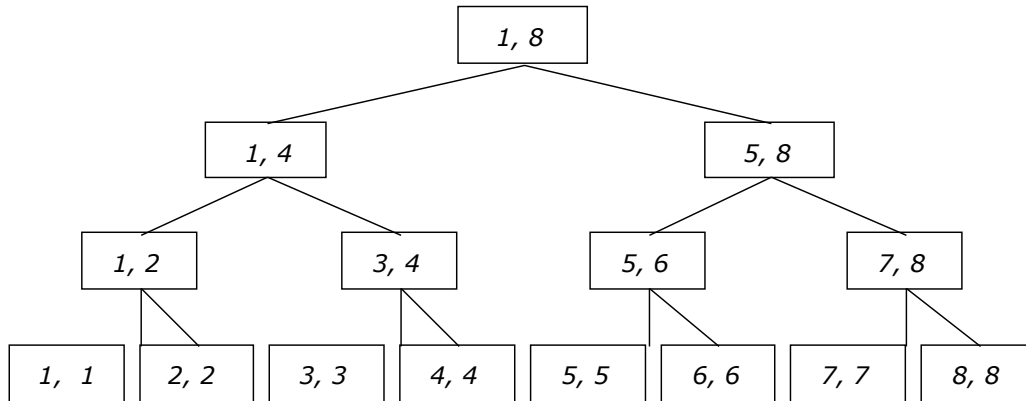
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



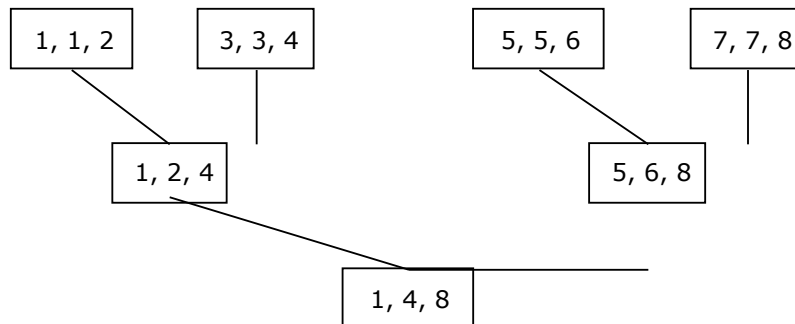
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For $n = 1$, the time to merge sort is constant, which we will denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size $n/2$, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$

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

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right-hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute $n/2$ into this main equation

$$\begin{aligned} 2 T(n/2) &= 2 (2 (T(n/4)) + n/2) \\ &= 4 T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2 T(n/4) + n \\ T(n) &= 4 T(n/4) + 2n \end{aligned}$$

Again, by substituting $n/4$ into the main equation, we see that

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

So we have,

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

Continuing in this manner, we obtain:

$$T(n) = 2^k T(n/2^k) + K \cdot n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$T(n) = 2^{\log_2 n}$$

$$\begin{aligned}
 &= n T(1) + n \log n \\
 &= n \log n + n
 \end{aligned}$$

Representing this in O notation:

$$T(n) = \mathbf{O(n \log n)}$$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence w_1, w_2, \dots, w_n take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare has devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is $O(n \log n)$.

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until $a[i] \geq \text{pivot}$.
- Repeatedly decrease the pointer 'j' until $a[j] \leq \text{pivot}$.
- If $j > i$, interchange $a[j]$ with $a[i]$
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition $\text{low} \geq \text{high}$ is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], x[j-1] and x[j+1], x[j+2],x[high].
- It calls itself recursively to sort the left sub-array x[low], x[low+1], x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], x[high] between positions j+1 and high.

Algorithm

Algorithm

QUICKSORT(low,

high)

/* sorts the elements a(low), , a(high) which reside in the global array A(1 : n) into ascending order a(n + 1) is considered to be defined and must be greater than all elements in a(1 : n); A(n + 1) = +∞ */

```
{
    if low < high then
    {
        j := PARTITION(a, low, high+1);
        // J is the position of the partitioning element
        QUICKSORT(low, j - 1);
        QUICKSORT(j + 1, high);
    }
}
```

Algorithm PARTITION(a, m, p)

```
{
    v ← a(m); i ← m; j ← p; // A(m) is the partition
    element do
    {
        loop i := i + 1 until a(i) ≥ v // i moves left to
        right loop j := j - 1 until a(j) ≤ v // p moves right to
        left if (i < j) then INTERCHANGE(a, i, j)
    } while (i ≥ j);
    a[m] := a[j]; a[j] := v; // the partition element belongs at position P
    return j;
}
```

Algorithm INTERCHANGE(a, i, j)

```
{
    p:=a[i];
    a[i] :=
    a[j]; a[j]
    := p;
}
```

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \quad - \quad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1 .

Worst Case Analysis

The pivot is the smallest element, all the time. Then $i=0$ and if we ignore $T(0)=1$, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + Cn \quad n > 1 \quad - \quad (2)$$

Using equation - (1) repeatedly, thus

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

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

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

$$\begin{aligned} T(n) &= T(1) + \sum_{i=2}^n i \\ &= O(n^2) \quad - \quad (3) \end{aligned}$$

Best Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big - oh answer.

$$T(n) = 2T(n/2) + Cn \quad - \quad (4)$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \quad - \quad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \quad - \quad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \quad - \quad (7)$$

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \quad - \quad (8)$$

We add all the equations from 4 to 8 and note that there are $\log n$ of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \quad - \quad (9)$$

Which yields, $T(n) = C n \log n + n = O(n \log n)$ - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

$$T(n) = \text{comparisons for first call on quicksort} \\ + \\ \left\{ \sum_{1 \leq n_{\text{left}}, n_{\text{right}} \leq n} [T(n_{\text{left}}) + T(n_{\text{right}})] \right\} / n = (n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-1)] / n$$

$$nT(n) = n(n+1) + 2 [T(0) + T(1) + T(2) + \dots + T(n-2) + T(n-1)]$$

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

Subtracting both sides:

$$nT(n) - (n-1)T(n-1) = [n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)$$

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

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

The recurrence relation

obtained is: $T(n)/(n+1) =$

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

Using the method of substitution:

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

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

$$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$$

$$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$$

.

.

$$T(3)/4 = 2/4 + T(2)/3$$

$$T(2)/3 = 2/3 + T(1)/2 \quad T(1)/2 = 2/2 + T(0)$$

Adding both sides:

$$T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + \dots + T(2)/3 + T(1)/2]$$

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

$$[2/(n+1) + 2/n + 2/(n-1) + \dots + 2/4 + 2/3]$$

Cancelling the common terms:

$$T(n)/(n+1) = 2[1/2 + 1/3 + 1/4 + \dots + 1/n + 1/(n+1)]$$

$$T(n) = (n+1)2 \left[\sum_{2 \leq k \leq n+1} 1/k \right]$$

$$= 2(n+1) [\dots]$$

$$= 2(n+1)[\log(n+1) - \log 2]$$

$$= 2n \log(n+1) + \log(n+1) - 2n \log 2 - \log 2$$

$$\mathbf{T(n) = O(n \log n)}$$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 4

Unit - III

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Greedy Method : CO3

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm*.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm*.

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
    solution :=  $\phi$ ;           // initialize the solution to empty
    for i:=1 to n do
    {
        x := select (a);
        if feasible (solution, x) then
            solution := Union (Solution, x);
    }
    return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The 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 fill 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'. The problem is stated as:

$$\begin{aligned} & \text{maximize } \sum_{i=1}^n p_i x_i \\ & \text{subject to } \sum_{i=1}^n a_i x_i \leq M \quad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n \end{aligned}$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of $p[i] / w[i]$ then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

// P[1 : n] and w[1 : n] contain the profits and weights respectively of

// Objects ordered so that $p[i] / w[i] > p[i + 1] / w[i + 1]$.

// m is the knapsack size and x[1: n] is the solution vector.

```
{
  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];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only $O(n)$ time.

Example:

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

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	$18 \times 1/2 + 15 \times 1/3 + 10 \times 1/4 = 16.5$	$25 \times 1/2 + 24 \times 1/3 + 15 \times 1/4 = 24.25$

2. Select the object with the maximum profit first ($p = 25$). So, $x_1 = 1$ and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit ($p = 24$). So, $x_2 = 2/15$

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	2/15	0	$18 \times 1 + 15 \times 2/15 = 20$	$25 \times 1 + 24 \times 2/15 = 28.2$

3. Considering the objects in the order of non-decreasing weights w_i .

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	$15 \times 2/3 + 10 \times 1 = 20$	$24 \times 2/3 + 15 \times 1 = 31$

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / w_i . Select the object with the maximum p_i / w_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / w_i ratio, so $x_3 = 1/2$ and the profit earned is 7.5.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	$15 \times 1 + 10 \times 1/2 = 20$	$24 \times 1 + 15 \times 1/2 = 31.5$

This solution is the optimal solution.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 5

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

4.4. OPTIMAL STORAGE ON TAPES

There are 'n' programs that are to be stored on a computer tape of length 'L'. Each program 'i' is of length l_i , $1 \leq i \leq n$. All the programs can be stored on the tape if and only if the sum of the lengths of the programs is at most 'L'.

We shall assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. If the programs are stored in the order $i = i_1, i_2, \dots, i_n$, the time t_j needed to retrieve program i_j is proportional to

$$\sum_{1 \leq k \leq j} l_{i_k}$$

If all the programs are retrieved equally often then the expected or mean retrieval time (MRT) is:

$$\frac{1}{n} \cdot \sum_{1 \leq j \leq n} t_j$$

For the optimal storage on tape problem, we are required to find the permutation for the 'n' programs so that when they are stored on the tape in this order the MRT is minimized.

$$d(I) = \sum_{j=1}^n \sum_{k=1}^j l_{i_k}$$

Example

Let $n = 3, (l_1, l_2, l_3) = (5, 10, 3)$. Then find the optimal ordering?

Solution:

There are $n! = 6$ possible orderings. They are:

<u>Ordering I</u>	<u>d(I)</u>		
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

From the above, it simply requires to store the programs in non-decreasing order (increasing order) of their lengths. This can be carried out by using a efficient sorting algorithm (Heap sort). This ordering can be carried out in $O(n \log n)$ time using heap sort algorithm.

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, T_0, \dots, T_{m-1} , then the programs are to be distributed over these tapes.

The total retrieval time (RT) is $\sum_{j=0}^{m-1} d(I_j)$

The objective is to store the programs in such a way as to minimize RT.

The programs are to be sorted in non decreasing order of their lengths l_i 's, $l_1 \leq l_2 \leq \dots \leq l_n$.

The first 'm' programs will be assigned to tapes T_0, \dots, T_{m-1} respectively. The next 'm' programs will be assigned to T_0, \dots, T_{m-1} respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$.

Algorithm:

The algorithm for assigning programs to tapes is as follows:

Algorithm Store (n, m)

```
// n is the number of programs and m the number of tapes
{
    j := 0; // next tape to store
    on for i :=1 to n do
    {
        Print ('append program', i, 'to permutation for tape',
            j); j := (j + 1) mod m;
    }
}
```

On any given tape, the programs are stored in non-decreasing order of their lengths.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit p_i is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that $j [r], 1 \leq r \leq k$ are the jobs in 'j' and $d (j [1]) \leq d (j [2]) \leq \dots \leq d (j [k])$. To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d [J[r]] \leq r, 1 \leq r \leq k+1$.

Example:

Let $n = 4, (P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	OPTIMAL
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

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 U {i} can be completed by their dead lines)  
        then J := J U {i};  
    }  
}
```



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 6

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Graph Algorithms: CO3

Basic Definitions:

- **Graph G** is a pair (V, E) , where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote $n := |V|$, $m := |E|$.
- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If $(u, v) \in E$, then vertices u , and v are adjacent.
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called **weighted**.
- **Degree** of a vertex v is the number of vertices u for which $(u, v) \in E$ (denote $\text{deg}(v)$). The number of **incoming edges** to a vertex v is called **in-degree** of the vertex (denote $\text{indeg}(v)$). The number of **outgoing edges** from a vertex is called **out-degree** (denote $\text{outdeg}(v)$).

Representation of Graphs:

Consider graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$.

Adjacency matrix represents the graph as an $n \times n$ matrix $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

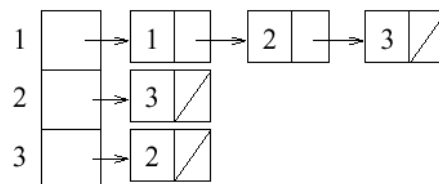
$$a_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ \text{default}, & \text{otherwise,} \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be ∞ , meaning value larger than any other value).

Adjacency List: An array Adj [1 n] of pointers where for $1 \leq v \leq n$, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

Adjacency matrix



Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices $(v_1, v_2, \dots, v_k, v_{k+1} = v_1)$, where for all i , $(v_i, v_{i+1}) \in E$ and all vertices in the cycle are distinct except pair v_1, v_{k+1} .

Subgraphs and Spanning Trees:

Subgraphs: A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v . If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

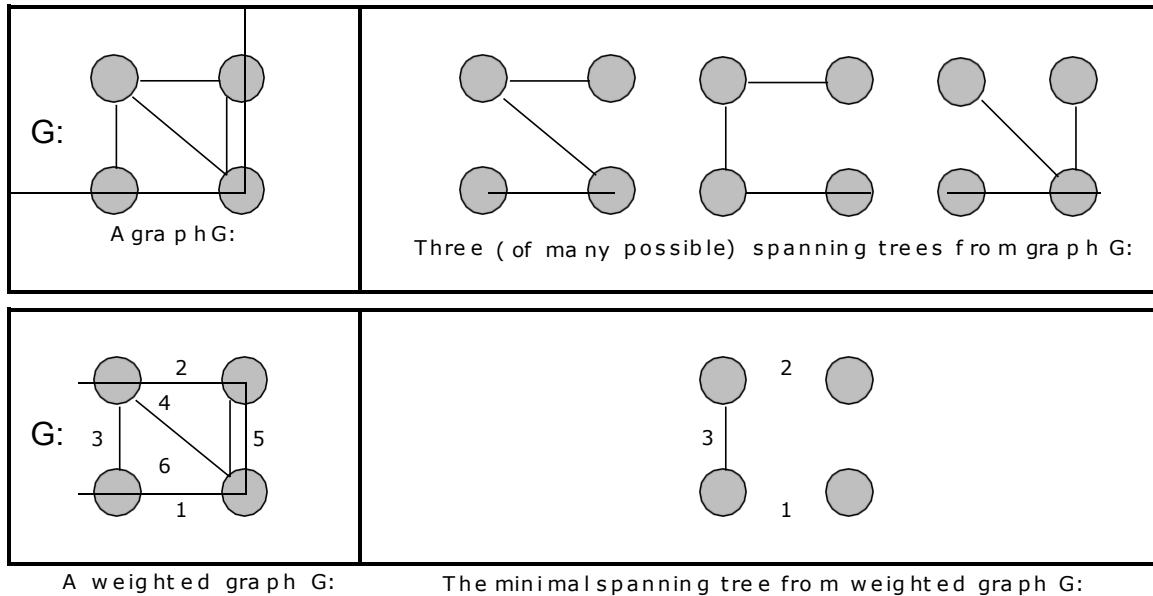
Lemma 1: Let T be a spanning tree of a graph G . Then

1. Any two vertices in T are connected by a unique simple path.
2. If any edge is removed from T , then T becomes disconnected.
3. If we add any edge into T , then the new graph will contain a cycle.
4. Number of edges in T is $n-1$.

Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

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.

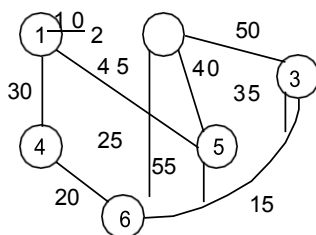
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap and
        re-heapify using Adjust;
        j := Find (u); k := Find (v);
        if (j ≠ k) then
        {
            i := i + 1;
            t [i, 1] := u; t [i, 2] := v;
            mincost := mincost + cost [u, v];
            Union (j, k);
        }
    }
    if (i ≠ n-1) then write ("no spanning tree");
    else return mincost;
}
```

Running time:

- The number of finds is at most $2e$, and the number of unions at most $n-1$. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than $O(n + e)$.
- We can add at most $n-1$ edges to tree T . So, the total time for operations on T is $O(n)$.

Summing up the various components of the computing times, we get $O(n + e \log e)$ as asymptotic complexity

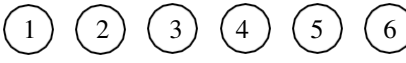

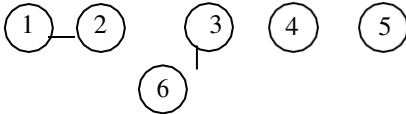
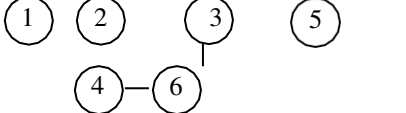
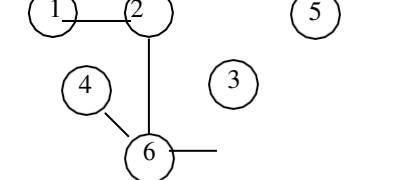
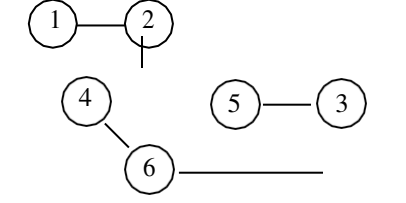
Example 1:



Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Finds** on the vertex sets suffice. When an edge is included in T , two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$	
(1, 2)	10		$\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}$	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		$\{1, 2\}, \{3, 6\}, \{4\}, \{5\}$	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		$\{1, 2\}, \{3, 4, 6\}, \{5\}$	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		$\{1, 2, 3, 4, 6\}, \{5\}$	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		$\{1, 2, 3, 4, 5, 6\}$	The vertices 3 and 5 are in the same set, so the edge is combined



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout - 7

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM: CO3

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

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  $\infty$  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]  $\neq$  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]  $\neq$  0) and (cost [k, near [k]] > cost [k,
            j])) then near [k] := j;
    }
    return mincost;
}
```

Running time:

We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get $O(n^2)$ time when we implement dist with array, $O(n + |E| \log n)$ when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in $\theta(1 + \deg(u))$ time. Therefore the running time is:

$$\theta \left(\sum_{v \in V} 1 + \deg(v) \right) = \theta \left(n + \sum_{v \in V} \deg(v) \right) = \theta(n + m)$$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout - 7

Unit - III

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS: CO3

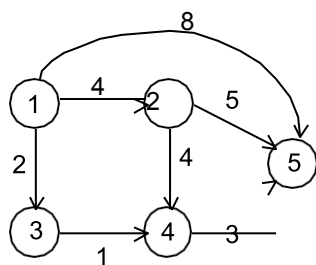
In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

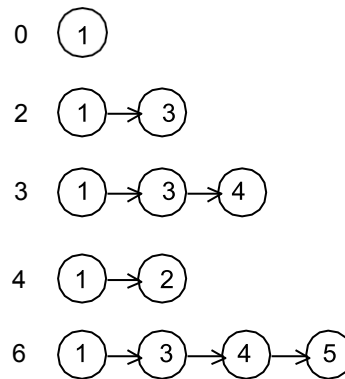
Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between them (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Graph



Shortest Paths

Algorithm:

Algorithm Shortest-Paths (v, cost, dist, n)

// dist [j], $1 \leq j \leq n$, is set to the length of the shortest path

// from vertex v to vertex j in the 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

 {

 S [i] := false;

 // Initialize S.

 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
            if (dist [w] > (dist [u] + cost [u, w]) then // Update distances
                dist [w] := dist [u] + cost [u, w];
            }
    }
}

```

Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements A
- at most $m = |E|$ times decrease the value of an item mB
- n times select the smallest value nC
- For array $A = O(n)$; $B = O(1)$; $C = O(n)$ which gives $O(n^2)$ total.
 - For heap $A = O(n)$; $B = O(\log n)$; $C = O(\log n)$ which gives $O(n + m \log n)$ total.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Dynamic Programming General method: CO4

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations*, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seem to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

5.1 MULTI STAGE GRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets $V_i, 1 \leq i \leq k$. In addition, if $\langle u, v \rangle$ is an edge in E , then $u \in V_i$ and $v \in V_{i+1}$ for some $i, 1 \leq i < k$.

Let the vertex 's' be the source, and 't' the sink. Let $c(i, j)$ be the cost of edge $\langle i, j \rangle$. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E , every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k .

A dynamic programming formulation for a k -stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of $k - 2$ decisions. The i^{th} decision involves determining which vertex in $V_{i+1}, 1 \leq i \leq k - 2$, is to be on the path. Let $c(i, j)$ be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost}(i, j) = \min_{\substack{l \in V_{i+1} \\ \langle j, l \rangle \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

ALGORITHM:

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  $\langle j, r \rangle$  is an edge
      of  $G$  and  $c[j, r] + \text{cost}[r]$  is minimum;
      cost[j] :=  $c[j, r] + \text{cost}[r]$ ;
      d[j] := r;
    }
  p[1] := 1; p[k] := n; // Find a minimum cost path.
  for j := 2 to k - 1 do p[j] := d[p[j - 1]];
}
```

The multistage graph problem can also be solved using the backward approach. Let $bp(i, j)$ be a minimum cost path from vertex s to j vertex in V_i . Let $Bcost(i, j)$ be the cost of $bp(i, j)$. From the backward approach we obtain:

$$Bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j \rangle \in E}} \{Bcost(i - 1, l) + c(l, j)\}$$

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]];
}
```

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has $|V|$ vertices and $|E|$ edges, then the time for the first for loop is $\Phi(|V| + |E|)$.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 2

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

All pairs shortest paths: CO3

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G . That is, for every pair of vertices (i, j) , we are to find a shortest path from i to j as well as one from j to i . These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

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 . The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time.

The dynamic programming solution, called Floyd's algorithm, runs in $O(n^3)$ time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G , $i \neq j$ originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j . If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j , respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let $A^k(i, j)$ represent the length of a shortest path from i to j going through no vertex of index greater than k , we obtain:

$$A^k(i, j) = \{ \min_{1 \leq k \leq n} \{ \min \{ A^{k-1}(i, k) + A^{k-1}(k, j) \}, c(i, j) \} \}$$

Algorithm All Paths (Cost, A, n)

// cost [1:n, 1:n] is the cost adjacency matrix of a graph which

// n vertices; $A[i, j]$ is the cost of a shortest path from vertex

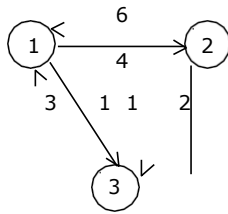
// i to vertex j . cost $[i, i] = 0.0$, for $1 \leq i \leq n$.

```
{
  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]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating $n+1$ matrices, each of size $n \times n$. Therefore, the algorithm has a complexity of $O(n^3)$.

Example 1:

Given a weighted digraph $G = (V, E)$ with weight. Determine the length of the shortest path between all pairs of vertices in G . Here we assume that there are no cycles with zero or negative cost.



$$\text{Cost adjacency matrix } (A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

General formula: $\min_{1 \leq k \leq n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)$

Solve the problem for different values of $k = 1, 2$ and 3

Step 1: Solving the equation for, $k = 1$;

$$\begin{aligned}
A^1(1, 1) &= \min \{(A^0(1, 1) + A^0(1, 1)), c(1, 1)\} = \min \{0 + 0, 0\} = 0 \\
A^1(1, 2) &= \min \{(A^0(1, 1) + A^0(1, 2)), c(1, 2)\} = \min \{(0 + 4), 4\} = 4 \\
A^1(1, 3) &= \min \{(A^0(1, 1) + A^0(1, 3)), c(1, 3)\} = \min \{(0 + 11), 11\} = 11 \\
A^1(2, 1) &= \min \{(A^0(2, 1) + A^0(1, 1)), c(2, 1)\} = \min \{(6 + 0), 6\} = 6 \\
A^1(2, 2) &= \min \{(A^0(2, 1) + A^0(1, 2)), c(2, 2)\} = \min \{(6 + 4), 0\} = 0 \\
A^1(2, 3) &= \min \{(A^0(2, 1) + A^0(1, 3)), c(2, 3)\} = \min \{(6 + 11), 2\} = 2 \\
A^1(3, 1) &= \min \{(A^0(3, 1) + A^0(1, 1)), c(3, 1)\} = \min \{(3 + 0), 3\} = 3 \\
A^1(3, 2) &= \min \{(A^0(3, 1) + A^0(1, 2)), c(3, 2)\} = \min \{(3 + 4), \infty\} = 7 \\
A^1(3, 3) &= \min \{(A^0(3, 1) + A^0(1, 3)), c(3, 3)\} = \min \{(3 + 11), 0\} = 0
\end{aligned}$$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, $K = 2$;

$$\begin{aligned}
A^2(1, 1) &= \min \{(A^1(1, 2) + A^1(2, 1)), c(1, 1)\} = \min \{(4 + 6), 0\} = 0 \\
A^2(1, 2) &= \min \{(A^1(1, 2) + A^1(2, 2)), c(1, 2)\} = \min \{(4 + 0), 4\} = 4 \\
A^2(1, 3) &= \min \{(A^1(1, 2) + A^1(2, 3)), c(1, 3)\} = \min \{(4 + 2), 11\} = 6 \\
A^2(2, 1) &= \min \{(A^1(2, 2) + A^1(2, 1)), c(2, 1)\} = \min \{(0 + 6), 6\} = 6 \\
A^2(2, 2) &= \min \{(A^1(2, 2) + A^1(2, 2)), c(2, 2)\} = \min \{(0 + 0), 0\} = 0 \\
A^2(2, 3) &= \min \{(A^1(2, 2) + A^1(2, 3)), c(2, 3)\} = \min \{(0 + 2), 2\} = 2 \\
A^2(3, 1) &= \min \{(A^1(3, 2) + A^1(2, 1)), c(3, 1)\} = \min \{(7 + 6), 3\} = 3 \\
A^2(3, 2) &= \min \{(A^1(3, 2) + A^1(2, 2)), c(3, 2)\} = \min \{(7 + 0), 7\} = 7 \\
A^2(3, 3) &= \min \{(A^1(3, 2) + A^1(2, 3)), c(3, 3)\} = \min \{(7 + 2), 0\} = 0
\end{aligned}$$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, $k = 3$;

$$\begin{aligned}
A^3(1, 1) &= \min \{(A^2(1, 3) + A^2(3, 1)), c(1, 1)\} = \min \{(6 + 3), 0\} = 0 \\
A^3(1, 2) &= \min \{(A^2(1, 3) + A^2(3, 2)), c(1, 2)\} = \min \{(6 + 7), 4\} = 4 \\
A^3(1, 3) &= \min \{(A^2(1, 3) + A^2(3, 3)), c(1, 3)\} = \min \{(6 + 0), 6\} = 6 \\
A^3(2, 1) &= \min \{(A^2(2, 3) + A^2(3, 1)), c(2, 1)\} = \min \{(2 + 3), 6\} = 5 \\
A^3(2, 2) &= \min \{(A^2(2, 3) + A^2(3, 2)), c(2, 2)\} = \min \{(2 + 7), 0\} = 0 \\
A^3(2, 3) &= \min \{(A^2(2, 3) + A^2(3, 3)), c(2, 3)\} = \min \{(2 + 0), 2\} = 2 \\
A^3(3, 1) &= \min \{(A^2(3, 3) + A^2(3, 1)), c(3, 1)\} = \min \{(0 + 3), 3\} = 3 \\
A^3(3, 2) &= \min \{(A^2(3, 3) + A^2(3, 2)), c(3, 2)\} = \min \{(0 + 7), 7\} = 7
\end{aligned}$$

$$A^3(3, 3) = \min \{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min \{(0 + 0), 0\} = 0$$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

TRAVELLING SALESPERSON PROBLEM: CO4

Let $G = (V, E)$ be a directed graph with edge costs C_{ij} . The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \alpha$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V . The cost of 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. The tour is to be a simple path that starts and ends at vertex 1.

Let $g(i, S)$ be the length of shortest path starting at vertex i , going through all vertices in S , and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad 1$$

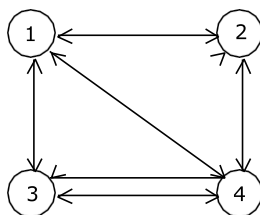
Generalizing equation 1, we obtain (for $i \notin S$)

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

The Equation can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k .

Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix =

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

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(j, s - \{j\})\} \quad \text{--} \quad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \leq i \leq n$. So,

$$g(2, \Phi) = C_{21} = 5$$

$$g(3, \Phi) = C_{31} = 6$$

$$g(4, \Phi) = C_{41} = 8$$

Using equation – (2) we obtain:

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

$$\begin{aligned} g(2, \{3, 4\}) &= \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} \\ &= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\} \end{aligned}$$

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

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

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

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

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

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

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

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

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

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

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

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

The optimal tour for the graph has length = 35

The optimal tour is: 1, 2, 4, 3, 1.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

OPTIMAL BINARY SEARCH TREE:CO4

Let us assume that the given set of identifiers is $\{a_1, \dots, a_n\}$ with $a_1 < a_2 < \dots < a_n$. Let $p(i)$ be the probability with which we search for a_i . Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons needed to access an element at depth ' d ' is $d + 1$, so if ' a_i ' is placed at depth ' d_i ', then we want to minimize:

$$\sum_{i=1} P_i (1 + d_i) .$$

Let $P(i)$ be the probability with which we shall be searching for ' a_i '. Let $Q(i)$ be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for ' a_i ' is:

$$P(i) * \text{level}(a_i) .$$

Unsuccessful search terminate with $I = 0$ (i.e at an external node). Hence the cost contribution for this node is:

$$Q(i) * \text{level}((E_i) - 1)$$

The expected cost of binary search tree is:

$$\sum_{i=1}^n P(i) * level(a_i) + \sum_{i=0}^n Q(i) * level((E_i) - 1)$$

Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these $c(i, j)$'s requires us to find the minimum of m quantities. Hence, each such $c(i, j)$ can be computed in time $O(m)$. The total time for all $c(i, j)$'s with $j - i = m$ is therefore $O(nm - m^2)$.

The total time to evaluate all the $c(i, j)$'s and $r(i, j)$'s is therefore:

$$\sum_{1 \leq i < j \leq n} (nm - m^2) = O(n^3)$$

Example 1:

Let $n = 4$, and $(a_1, a_2, a_3, a_4) = (\text{do, if, need, while})$ Let $P(1:4) = (3, 3, 1, 1)$ and $Q(0:4) = (2, 3, 1, 1, 1)$

Solution:

Table for recording $W(i, j)$, $C(i, j)$ and $R(i, j)$:

Column Row	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially, $W(i, i) = Q(i)$ and $C(i, i) = 0$ and $R(i, i) = 0, 0 \leq i < 4$.

Solving for $C(0, n)$:

First, computing all $C(i, j)$ such that $j - i = 1; j = i + 1$ and as $0 \leq i < 4; i = 0, 1, 2$ and $3; i < k \leq j$. Start with $i = 0; so j = 1; as i < k \leq j$, so the possible value for $k = 1$

$$W(0, 1) = P(1) + Q(1) + W(0, 0) = 3 + 3 + 2 = 8$$

$$C(0, 1) = W(0, 1) + \min \{C(0, 0) + C(1, 1)\} = 8$$

$$R(0, 1) = 1 \text{ (value of 'K' that is minimum in the above equation).}$$

Next with $i = 1; so j = 2; as i < k \leq j$, so the possible value for $k = 2$

$$W(1, 2) = P(2) + Q(2) + W(1, 1) = 3 + 1 + 3 = 7$$

$$C(1, 2) = W(1, 2) + \min \{C(1, 1) + C(2, 2)\} = 7$$

$$R(1, 2) = 2$$

Next with $i = 2; so j = 3; as i < k \leq j$, so the possible value for $k = 3$

$$W(2, 3) = P(3) + Q(3) + W(2, 2) = 1 + 1 + 1 = 3$$

$$C(2, 3) = W(2, 3) + \min \{C(2, 2) + C(3, 3)\} = 3 + [(0 + 0)] = 3$$

$$R(2, 3) = 3$$

Next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$W(3, 4) = P(4) + Q(4) + W(3, 3) = 1 + 1 + 1 = 3$$

$$C(3, 4) = W(3, 4) + \min \{[C(3, 3) + C(4, 4)]\} = 3 + [(0 + 0)] = 3$$

$$R(3, 4) = 4$$

Second, Computing all $C(i, j)$ such that $j - i = 2$; $j = i + 2$ and as $0 \leq i < 3$; $i = 0, 1, 2$; $i < k \leq J$. Start with $i = 0$; so $j = 2$; as $i < k \leq J$, so the possible values for $k = 1$ and 2 .

$$W(0, 2) = P(2) + Q(2) + W(0, 1) = 3 + 1 + 8 = 12$$

$$C(0, 2) = W(0, 2) + \min \{(C(0, 0) + C(1, 2)), (C(0, 1) + C(2, 2))\}$$

$$= 12 + \min \{(0 + 7, 8 + 0)\} = 19$$

$$R(0, 2) = 1$$

Next, with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 2$ and 3 .

$$W(1, 3) = P(3) + Q(3) + W(1, 2) = 1 + 1 + 7 = 9$$

$$C(1, 3) = W(1, 3) + \min \{[C(1, 1) + C(2, 3)], [C(1, 2) + C(3, 3)]\}$$

$$= W(1, 3) + \min \{(0 + 3), (7 + 0)\} = 9 + 3 = 12$$

$$R(1, 3) = 2$$

Next, with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 3$ and 4 .

$$W(2, 4) = P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5$$

$$C(2, 4) = W(2, 4) + \min \{[C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)]\}$$

$$= 5 + \min \{(0 + 3), (3 + 0)\} = 5 + 3 = 8$$

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

Third, Computing all $C(i, j)$ such that $J - i = 3$; $j = i + 3$ and as $0 \leq i < 2$; $i = 0, 1$; $i < k \leq J$. Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$W(0, 3) = P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14$$

$$C(0, 3) = W(0, 3) + \min \{[C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)],$$

$$[C(0, 2) + C(3, 3)]\}$$

$$= 14 + \min \{(0 + 12), (8 + 3), (19 + 0)\} = 14 + 11 = 25$$

$$R(0, 3) = 2$$

Start with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$W(1, 4) = P(4) + Q(4) + W(1, 3) = 1 + 1 + 9 = 11$$

$$C(1, 4) = W(1, 4) + \min \{[C(1, 1) + C(2, 4)], [C(1, 2) + C(3, 4)],$$

$$[C(1, 3) + C(4, 4)]\}$$

$$= 11 + \min \{(0 + 8), (7 + 3), (12 + 0)\} = 11 + 8 = 19$$

$$R(1, 4) = 2$$

Fourth, Computing all $C(i, j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \leq i < 1$; $i = 0$; $i < k \leq J$.

Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned}
W(0, 4) &= P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16 \\
C(0, 4) &= W(0, 4) + \min \{ [C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], \\
&\quad [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)] \} \\
&= 16 + \min [0 + 19, 8 + 8, 19 + 3, 25 + 0] = 16 + 16 = 32 \\
R(0, 4) &= 2
\end{aligned}$$

From the table we see that $C(0, 4) = 32$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4) . The root of the tree 'T₀₄' is 'a₂'.

Hence the left sub tree is 'T₀₁' and right sub tree is T₂₄. The root of 'T₀₁' is 'a₁' and the root of 'T₂₄' is a₃.

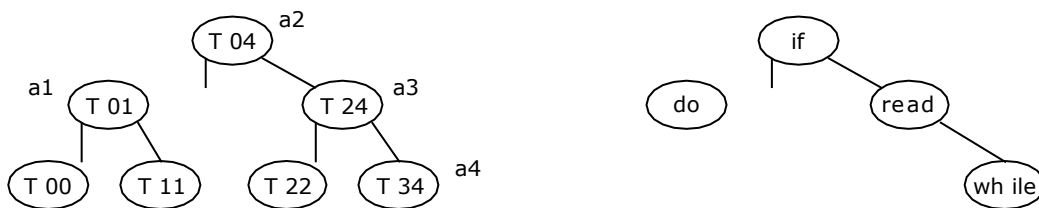
The left and right sub trees for 'T₀₁' are 'T₀₀' and 'T₁₁' respectively. The root of T₀₁ is 'a₁'

The left and right sub trees for T₂₄ are T₂₂ and T₃₄ respectively.

The root of T₂₄ is 'a₃'.

The root of T₂₂ is null

The root of T₃₄ is a₄.



Example 2:

Consider four elements a_1, a_2, a_3 and a_4 with $Q_0 = 1/8, Q_1 = 3/16, Q_2 = Q_3 = Q_4 = 1/16$ and $p_1 = 1/4, p_2 = 1/8, p_3 = p_4 = 1/16$. Construct an optimal binary search tree. Solving for $C(0, n)$:

First, computing all $C(i, j)$ such that $j - i = 1; j = i + 1$ and as $0 \leq i < 4; i = 0, 1, 2$ and 3; $i < k \leq j$. Start with $i = 0$; so $j = 1$; as $i < k \leq j$, so the possible value for $k = 1$

$$\begin{aligned}
W(0, 1) &= P(1) + Q(1) + W(0, 0) = 4 + 3 + 2 = 9 \\
C(0, 1) &= W(0, 1) + \min \{ C(0, 0) + C(1, 1) \} = 9 + [(0 + 0)] = 9 \\
R(0, 1) &= 1 \text{ (value of 'K' that is minimum in the above equation).}
\end{aligned}$$

Next with $i = 1$; so $j = 2$; as $i < k \leq j$, so the possible value for $k = 2$

$$\begin{aligned}
W(1, 2) &= P(2) + Q(2) + W(1, 1) = 2 + 1 + 3 = 6 \\
C(1, 2) &= W(1, 2) + \min \{ C(1, 1) + C(2, 2) \} = 6 + [(0 + 0)] = 6 \\
R(1, 2) &= 2
\end{aligned}$$

Next with $i = 2$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 3$

$$\begin{aligned}
W(2, 3) &= P(3) + Q(3) + W(2, 2) = 1 + 1 + 1 = 3 \\
C(2, 3) &= W(2, 3) + \min \{ C(2, 2) + C(3, 3) \} = 3 + [(0 + 0)] = 3
\end{aligned}$$

$$R(2, 3) = 3$$

Next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$\begin{aligned} W(3, 4) &= P(4) + Q(4) + W(3, 3) = 1 + 1 + 1 = 3 \\ C(3, 4) &= W(3, 4) + \min \{ [C(3, 3) + C(4, 4)] \} = 3 + [(0 + 0)] = 3 \\ R(3, 4) &= 4 \end{aligned}$$

Second, Computing all $C(i, j)$ such that $j - i = 2$; $j = i + 2$ and as $0 \leq i < 3$; $i = 0, 1, 2$; $i < k \leq J$

Start with $i = 0$; so $j = 2$; as $i < k \leq j$, so the possible values for $k = 1$ and 2 .

$$\begin{aligned} W(0, 2) &= P(2) + Q(2) + W(0, 1) = 2 + 1 + 9 = 12 \\ C(0, 2) &= W(0, 2) + \min \{ (C(0, 0) + C(1, 2)), (C(0, 1) + C(2, 2)) \} \\ &= 12 + \min \{ (0 + 6, 9 + 0) \} = 12 + 6 = 18 \\ R(0, 2) &= 1 \end{aligned}$$

Next, with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 2$ and 3 .

$$\begin{aligned} W(1, 3) &= P(3) + Q(3) + W(1, 2) = 1 + 1 + 6 = 8 \\ C(1, 3) &= W(1, 3) + \min \{ [C(1, 1) + C(2, 3)], [C(1, 2) + C(3, 3)] \} \\ &= W(1, 3) + \min \{ (0 + 3), (6 + 0) \} = 8 + 3 = 11 \\ R(1, 3) &= 2 \end{aligned}$$

Next, with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 3$ and 4 .

$$\begin{aligned} W(2, 4) &= P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5 \\ C(2, 4) &= W(2, 4) + \min \{ [C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)] \} \\ &= 5 + \min \{ (0 + 3), (3 + 0) \} = 5 + 3 = 8 \\ R(2, 4) &= 3 \end{aligned}$$

Third, Computing all $C(i, j)$ such that $J - i = 3$; $j = i + 3$ and as $0 \leq i < 2$; $i = 0, 1$; $i < k \leq J$. Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$\begin{aligned} W(0, 3) &= P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14 \\ C(0, 3) &= W(0, 3) + \min \{ [C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)], \\ &\quad [C(0, 2) + C(3, 3)] \} \\ &= 14 + \min \{ (0 + 11), (9 + 3), (18 + 0) \} = 14 + 11 = 25 \\ R(0, 3) &= 1 \end{aligned}$$

Start with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$\begin{aligned} W(1, 4) &= P(4) + Q(4) + W(1, 3) = 1 + 1 + 8 = 10 \\ C(1, 4) &= W(1, 4) + \min \{ [C(1, 1) + C(2, 4)], [C(1, 2) + C(3, 4)], \\ &\quad [C(1, 3) + C(4, 4)] \} \\ &= 10 + \min \{ (0 + 8), (6 + 3), (11 + 0) \} = 10 + 8 = 18 \\ R(1, 4) &= 2 \end{aligned}$$

Fourth, Computing all $C(i, j)$ such that $J - i = 4$; $j = i + 4$ and as $0 \leq i < 1$; $i = 0$; $i < k \leq J$. Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned} W(0, 4) &= P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16 \\ C(0, 4) &= W(0, 4) + \min \{ [C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], \\ &\quad [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)] \} \end{aligned}$$

$$= 16 + \min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33$$

$$R(0, 4) = 2$$

Table for recording $W(i, j)$, $C(i, j)$ and $R(i, j)$

Column Row	0	1	2	3	4
0	2, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	9, 9, 1	6, 6, 2	3, 3, 3	3, 3, 4	
2	12, 18, 1	8, 11, 2	5, 8, 3		
3	14, 25, 2	11, 18, 2			
4	16, 33, 2				

From the table we see that $C(0, 4) = 33$ is the minimum cost of a binary search tree for (a_1, a_2, a_3, a_4)

The root of the tree ' T_{04} ' is ' a_2 '.

Hence the left sub tree is ' T_{01} ' and right sub tree is T_{24} . The root of ' T_{01} ' is ' a_1 ' and the root of ' T_{24} ' is a_3 .

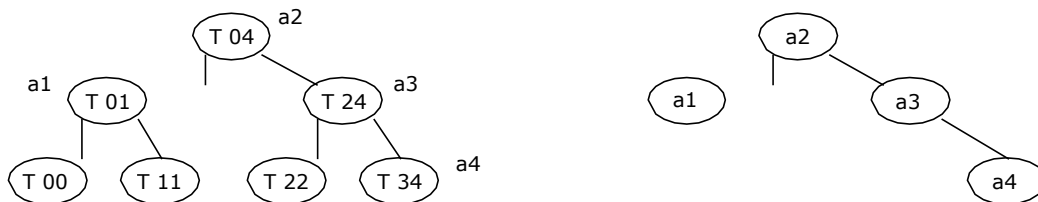
The left and right sub trees for ' T_{01} ' are ' T_{00} ' and ' T_{11} ' respectively. The root of T_{01} is ' a_1 '

The left and right sub trees for T_{24} are T_{22} and T_{34} respectively.

The root of T_{24} is ' a_3 '.

The root of T_{22} is null.

The root of T_{34} is a_4 .



Example 3:

<u>WORD</u>	<u>PROBABILITY</u>
A	4
B	2
C	1
D	3
E	5
F	2
G	1

and all other elements have zero probability.

Solving c(0,n):

First computing all $c(i, j)$ such that $j - i = 1; j = i + 1$ and as $0 \leq i < 7; i = 0, 1, 2, 3, 4, 5$ and $6; i < k \leq j$. Start with $i = 0$; so $j = 1$; as $i < k \leq j$, so the possible value for $k = 1$

$$\begin{aligned} W(0, 1) &= P(1) + Q(1) + W(0, 0) = 4 + 0 + 0 = 4 \\ C(0, 1) &= W(0, 1) + \min \{ C(0, 0) + C(1, 1) \} = 4 + [(0 + 0)] = 4 \\ R(0, 1) &= 1 \end{aligned}$$

next with $i = 1$; so $j = 2$; as $i < k \leq j$, so the possible value for $k = 2$

$$\begin{aligned} W(1, 2) &= P(2) + Q(2) + W(1, 1) = 2 + 0 + 0 = 2 \\ C(1, 2) &= W(1, 2) + \min \{ C(1, 1) + C(2, 2) \} = 2 + [(0 + 0)] = 2 \\ R(1, 2) &= 2 \end{aligned}$$

next with $i = 2$; so $j = 3$; as $i < k \leq j$, so the possible value for $k = 3$

$$\begin{aligned} W(2, 3) &= P(3) + Q(3) + W(2, 2) = 1 + 0 + 0 = 1 \\ C(2, 3) &= W(2, 3) + \min \{ C(2, 2) + C(3, 3) \} = 1 + [(0 + 0)] = 1 \\ R(2, 3) &= 3 \end{aligned}$$

next with $i = 3$; so $j = 4$; as $i < k \leq j$, so the possible value for $k = 4$

$$\begin{aligned} W(3, 4) &= P(4) + Q(4) + W(3, 3) = 3 + 0 + 0 = 3 \\ C(3, 4) &= W(3, 4) + \min \{ C(3, 4) + C(4, 4) \} = 3 + [(0 + 0)] = 3 \\ R(3, 4) &= 4 \end{aligned}$$

next with $i = 4$; so $j = 5$; as $i < k \leq j$, so the possible value for $k = 5$

$$\begin{aligned} W(4, 5) &= P(5) + Q(5) + W(4, 4) = 5 + 0 + 0 = 5 \\ C(4, 5) &= W(4, 5) + \min \{ C(4, 4) + C(5, 5) \} = 5 + [(0 + 0)] = 5 \\ R(4, 5) &= 5 \end{aligned}$$

next with $i = 5$; so $j = 6$; as $i < k \leq j$, so the possible value for $k = 6$

$$\begin{aligned} W(5, 6) &= P(6) + Q(6) + W(5, 5) = 2 + 0 + 0 = 2 \\ C(5, 6) &= W(5, 6) + \min \{ C(5, 5) + C(6, 6) \} = 2 + [(0 + 0)] = 2 \\ R(5, 6) &= 6 \end{aligned}$$

next with $i = 6$; so $j = 7$; as $i < k \leq j$, so the possible value for $k = 7$

$$\begin{aligned} W(6, 7) &= P(7) + Q(7) + W(6, 6) = 1 + 0 + 0 = 1 \\ C(6, 7) &= W(6, 7) + \min \{ C(6, 6) + C(7, 7) \} = 1 + [(0 + 0)] = 1 \\ R(6, 7) &= 7 \end{aligned}$$

Second, computing all $c(i, j)$ such that $j - i = 2; j = i + 2$ and as $0 \leq i < 6; i = 0, 1, 2, 3, 4$ and $5; i < k \leq j$; Start with $i = 0$; so $j = 2$; as $i < k \leq j$, so the possible values for $k = 1$ and 2 .

$$\begin{aligned} W(0, 2) &= P(2) + Q(2) + W(0, 1) = 2 + 0 + 4 = 6 \\ C(0, 2) &= W(0, 2) + \min \{ C(0, 0) + C(1, 2), C(0, 1) + C(2, 2) \} \\ &= 6 + \min \{ 0 + 2, 4 + 0 \} = 8 \\ R(0, 2) &= 1 \end{aligned}$$

next with $i = 1$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 2$ and 3 .

$$\begin{aligned}W(1, 3) &= P(3) + Q(3) + W(1, 2) = 1 + 0 + 2 = 3 \\C(1, 3) &= W(1, 3) + \min \{C(1, 1) + C(2, 3), C(1, 2) + C(3, 3)\} \\&= 3 + \min\{0 + 1, 2 + 0\} = 4 \\R(1, 3) &= 2\end{aligned}$$

next with $i = 2$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 3$ and 4 .

$$\begin{aligned}W(2, 4) &= P(4) + Q(4) + W(2, 3) = 3 + 0 + 1 = 4 \\C(2, 4) &= W(2, 4) + \min \{C(2, 2) + C(3, 4), C(2, 3) + C(4, 4)\} \\&= 4 + \min\{0 + 3, 1 + 0\} = 5 \\R(2, 4) &= 4\end{aligned}$$

next with $i = 3$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 4$ and 5 .

$$\begin{aligned}W(3, 5) &= P(5) + Q(5) + W(3, 4) = 5 + 0 + 3 = 8 \\C(3, 5) &= W(3, 5) + \min \{C(3, 3) + C(4, 5), C(3, 4) + C(5, 5)\} \\&= 8 + \min\{0 + 5, 3 + 0\} = 11 \\R(3, 5) &= 5\end{aligned}$$

next with $i = 4$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 5$ and 6 .

$$\begin{aligned}W(4, 6) &= P(6) + Q(6) + W(4, 5) = 2 + 0 + 5 = 7 \\C(4, 6) &= W(4, 6) + \min \{C(4, 4) + C(5, 6), C(4, 5) + C(6, 6)\} \\&= 7 + \min\{0 + 2, 5 + 0\} = 9 \\R(4, 6) &= 5\end{aligned}$$

next with $i = 5$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 6$ and 7 .

$$\begin{aligned}W(5, 7) &= P(7) + Q(7) + W(5, 6) = 1 + 0 + 2 = 3 \\C(5, 7) &= W(5, 7) + \min \{C(5, 5) + C(6, 7), C(5, 6) + C(7, 7)\} \\&= 3 + \min\{0 + 1, 2 + 0\} = 4 \\R(5, 7) &= 6\end{aligned}$$

Third, computing all $c(i, j)$ such that $j - i = 3$; $j = i + 3$ and as $0 \leq i < 5$; $i = 0, 1, 2, 3, 4$ and $I < k \leq j$.

Start with $i = 0$; so $j = 3$; as $i < k \leq j$, so the possible values for $k = 1, 2$ and 3 .

$$\begin{aligned}W(0, 3) &= P(3) + Q(3) + W(0, 2) = 1 + 0 + 6 = 7 \\C(0, 3) &= W(0, 3) + \min \{C(0, 0) + C(1, 3), C(0, 1) + C(2, 3), C(0, 2) + C(3, 3)\} \\&= 7 + \min\{0 + 4, 4 + 1, 8 + 0\} = 7 \\R(0, 3) &= 1\end{aligned}$$

next with $i = 1$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 2, 3$ and 4 .

$$\begin{aligned}W(1, 4) &= P(4) + Q(4) + W(1, 3) = 3 + 0 + 3 = 6 \\C(1, 4) &= W(1, 4) + \min \{C(1, 1) + C(2, 4), C(1, 2) + C(3, 4), C(1, 3) + C(4, 4)\} \\&= 6 + \min\{0 + 5, 2 + 3, 4 + 0\} = 10 \\R(1, 4) &= 4\end{aligned}$$

next with $i = 2$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 3, 4$ and 5 .

$$\begin{aligned}W(2, 5) &= P(5) + Q(5) + W(2, 4) = 5 + 0 + 4 = 9 \\C(2, 5) &= W(2, 5) + \min \{C(2, 2) + C(3, 5), C(2, 3) + C(4, 5), C(2, 4) + C(5, 5)\} \\&= 9 + \min\{0 + 11, 1 + 5, 5 + 0\} = 14 \\R(2, 5) &= 5\end{aligned}$$

next with $i = 3$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 4, 5$ and 6 .

$$\begin{aligned} W(3, 6) &= P(6) + Q(6) + W(3, 5) = 2 + 0 + 8 = 10 \\ C(3, 6) &= W(3, 6) + \min \{C(3, 3) + C(4, 6), C(3, 4) + C(5, 6), C(3, 5) + C(6, 6)\} \\ &= 10 + \min\{0 + 9, 3 + 2, 11 + 0\} = 15 \\ R(3, 6) &= 5 \end{aligned}$$

next with $i = 4$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 5, 6$ and 7 .

$$\begin{aligned} W(4, 7) &= P(7) + Q(7) + W(4, 6) = 1 + 0 + 7 = 8 \\ C(4, 7) &= W(4, 7) + \min \{C(4, 4) + C(5, 7), C(4, 5) + C(6, 7), C(4, 6) + C(7, 7)\} \\ &= 8 + \min\{0 + 4, 5 + 1, 9 + 0\} = 12 \\ R(4, 7) &= 5 \end{aligned}$$

Fourth, computing all $c(i, j)$ such that $j - i = 4$; $j = i + 4$ and as $0 \leq i < 4$; $i = 0, 1, 2, 3$ for $i < k \leq j$. Start with $i = 0$; so $j = 4$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3$ and 4 .

$$\begin{aligned} W(0, 4) &= P(4) + Q(4) + W(0, 3) = 3 + 0 + 7 = 10 \\ C(0, 4) &= W(0, 4) + \min \{C(0, 0) + C(1, 4), C(0, 1) + C(2, 4), C(0, 2) + C(3, 4), \\ &\quad C(0, 3) + C(4, 4)\} \\ &= 10 + \min\{0 + 10, 4 + 5, 8 + 3, 11 + 0\} = 19 \\ R(0, 4) &= 2 \end{aligned}$$

next with $i = 1$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4$ and 5 .

$$\begin{aligned} W(1, 5) &= P(5) + Q(5) + W(1, 4) = 5 + 0 + 6 = 11 \\ C(1, 5) &= W(1, 5) + \min \{C(1, 1) + C(2, 5), C(1, 2) + C(3, 5), C(1, 3) + C(4, 5), \\ &\quad C(1, 4) + C(5, 5)\} \\ &= 11 + \min\{0 + 14, 2 + 11, 4 + 5, 10 + 0\} = 20 \\ R(1, 5) &= 4 \end{aligned}$$

next with $i = 2$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 3, 4, 5$ and 6 .

$$\begin{aligned} W(2, 6) &= P(6) + Q(6) + W(2, 5) = 2 + 0 + 9 = 11 \\ C(2, 6) &= W(2, 6) + \min \{C(2, 2) + C(3, 6), C(2, 3) + C(4, 6), C(2, 4) + C(5, 6), \\ &\quad C(2, 5) + C(6, 6)\} = 11 + \min\{0 + 15, 1 + 9, 5 + 2, 14 + 0\} = 18 \\ R(2, 6) &= 5 \end{aligned}$$

next with $i = 3$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 4, 5, 6$ and 7 .

$$\begin{aligned} W(3, 7) &= P(7) + Q(7) + W(3, 6) = 1 + 0 + 11 = 12 \\ C(3, 7) &= W(3, 7) + \min \{C(3, 3) + C(4, 7), C(3, 4) + C(5, 7), C(3, 5) + C(6, 7), \\ &\quad C(3, 6) + C(7, 7)\} = 12 + \min\{0 + 12, 3 + 4, 11 + 1, 15 + 0\} = 19 \\ R(3, 7) &= 5 \end{aligned}$$

Fifth, computing all $c(i, j)$ such that $j - i = 5$; $j = i + 5$ and as $0 \leq i < 3$; $i = 0, 1, 2$, $i < k \leq j$. Start with $i = 0$; so $j = 5$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4$ and 5 .

$$\begin{aligned} W(0, 5) &= P(5) + Q(5) + W(0, 4) = 5 + 0 + 10 = 15 \\ C(0, 5) &= W(0, 5) + \min \{C(0, 0) + C(1, 5), C(0, 1) + C(2, 5), C(0, 2) + C(3, 5), \\ &\quad C(0, 3) + C(4, 5), C(0, 4) + C(5, 5)\} \\ &= 15 + \min\{0 + 20, 4 + 14, 8 + 11, 19 + 0\} = 28 \end{aligned}$$

$$R(0, 5) = 2$$

next with $i = 1$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4, 5$ & 6 .

$$\begin{aligned} W(1, 6) &= P(6) + Q(6) + W(1, 5) = 2 + 0 + 11 = 13 \\ C(1, 6) &= W(1, 6) + \min \{C(1, 1) + C(2, 6), C(1, 2) + C(3, 6), C(1, 3) + C(4, 6), \\ &\quad C(1, 4) + C(5, 6), C(1, 5) + C(6, 6)\} \\ &= 13 + \min\{0 + 18, 2 + 15, 4 + 9, 10 + 2, 20 + 0\} = 25 \\ R(1, 6) &= 5 \end{aligned}$$

next with $i = 2$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(2, 7) &= P(7) + Q(7) + W(2, 6) = 1 + 0 + 11 = 12 \\ C(2, 7) &= W(2, 7) + \min \{C(2, 2) + C(3, 7), C(2, 3) + C(4, 7), C(2, 4) + C(5, 7), \\ &\quad C(2, 5) + C(6, 7), C(2, 6) + C(7, 7)\} \\ &= 12 + \min\{0 + 18, 1 + 12, 5 + 4, 14 + 1, 18 + 0\} = 21 \\ R(2, 7) &= 5 \end{aligned}$$

Sixth, computing all $c(i, j)$ such that $j - i = 6$; $j = i + 6$ and as $0 \leq i < 2$; $i = 0, 1$; $i < k \leq j$. Start with $i = 0$; so $j = 6$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4, 5$ & 6 .

$$\begin{aligned} W(0, 6) &= P(6) + Q(6) + W(0, 5) = 2 + 0 + 15 = 17 \\ C(0, 6) &= W(0, 6) + \min \{C(0, 0) + C(1, 6), C(0, 1) + C(2, 6), C(0, 2) + C(3, 6), \\ &\quad C(0, 3) + C(4, 6), C(0, 4) + C(5, 6), C(0, 5) + C(6, 6)\} \\ &= 17 + \min\{0 + 25, 4 + 18, 8 + 15, 19 + 2, 31 + 0\} = 37 \\ R(0, 6) &= 4 \end{aligned}$$

next with $i = 1$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 2, 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(1, 7) &= P(7) + Q(7) + W(1, 6) = 1 + 0 + 13 = 14 \\ C(1, 7) &= W(1, 7) + \min \{C(1, 1) + C(2, 7), C(1, 2) + C(3, 7), C(1, 3) + C(4, 7), \\ &\quad C(1, 4) + C(5, 7), C(1, 5) + C(6, 7), C(1, 6) + C(7, 7)\} \\ &= 14 + \min\{0 + 21, 2 + 18, 4 + 12, 10 + 4, 20 + 1, 21 + 0\} = 28 \\ R(1, 7) &= 5 \end{aligned}$$

Seventh, computing all $c(i, j)$ such that $j - i = 7$; $j = i + 7$ and as $0 \leq i < 1$; $i = 0$; $i < k \leq j$. Start with $i = 0$; so $j = 7$; as $i < k \leq j$, so the possible values for $k = 1, 2, 3, 4, 5, 6$ and 7 .

$$\begin{aligned} W(0, 7) &= P(7) + Q(7) + W(0, 6) = 1 + 0 + 17 = 18 \\ C(0, 7) &= W(0, 7) + \min \{C(0, 0) + C(1, 7), C(0, 1) + C(2, 7), C(0, 2) + C(3, 7), \\ &\quad C(0, 3) + C(4, 7), C(0, 4) + C(5, 6), C(0, 5) + C(6, 7), C(0, 6) + C(7, 7)\} \\ &= 18 + \min\{0 + 28, 4 + 21, 8 + 18, 19 + 4, 31 + 1, 37 + 0\} = 41 \\ R(0, 7) &= 4 \end{aligned}$$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 4

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

0/1 – KNAPSACK-CO4

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i . The knapsack can carry a weight not exceeding W . Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x_1, x_2, \dots, x_n . A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the x_i are made in the order x_n, x_{n-1}, \dots, x_1 . Following a decision on x_n , we may be in one of two possible states: the capacity remaining in $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, \dots, x_1 must be optimal with respect to the problem state resulting from the decision on x_n . Otherwise, x_n, \dots, x_1 will not be optimal. Hence, the principal of optimality holds.

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

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad \text{--} \quad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then f_1, f_2, \dots, f_n can be successively computed using equation-2.

When the w_i 's are integer, we need to compute $f_i(y)$ for integer y , $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n . When the w_i 's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i 's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore **an alternative method for both cases**.

The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y 's, $0 = y_1 < y_2 < \dots < y_k$, such that $f_i(y_1) < f_i(y_2) < \dots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f_i(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W) , where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S^i by first computing:

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

Now, S^{i+1} can be computed by merging the pairs in S_1^i and S^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j) .

Example 1:

Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(P_1, P_2, P_3) = (1, 2, 5)$ and $M = 6$.

Solution:

Initially, $f_0(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$.

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

$$F_3(6) = \max \{f_2(6), f_2(6 - 4) + 5\} = \max \{f_2(6), f_2(2) + 5\}$$

$$F_2(6) = \max \{f_1(6), f_1(6 - 3) + 2\} = \max \{f_1(6), f_1(3) + 2\}$$

$$F_1(6) = \max (f_0(6), f_0(6 - 2) + 1) = \max \{0, 0 + 1\} = 1$$

$$F_1(3) = \max (f_0(3), f_0(3 - 2) + 1) = \max \{0, 0 + 1\} = 1$$

$$\text{Therefore, } F_2(6) = \max (1, 1 + 2) = 3$$

$$F_2(2) = \max (f_1(2), f_1(2 - 3) + 2) = \max \{f_1(2), -\infty + 2\}$$

$$F_1(2) = \max (f_0(2), f_0(2 - 2) + 1) = \max \{0, 0 + 1\} = 1$$

$$F_2(2) = \max \{1, -\infty + 2\} = 1$$

$$\text{Finally, } f_3(6) = \max \{3, 1 + 5\} = 6$$

Other Solution:

For the given data we have:

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

$$S^1 = (S^0 \cup S^0_1) = \{(0, 0), (1, 2)\}$$

$$\begin{array}{ll} X - 2 = 0 \Rightarrow x = 2. & y - 3 = 0 \Rightarrow y = 3 \\ X - 2 = 1 \Rightarrow x = 3. & y - 3 = 2 \Rightarrow y = 5 \end{array}$$

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

$$S^2 = (S^1 \cup S^{1_1}) = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

$$\begin{array}{ll} X - 5 = 0 \Rightarrow x = 5. & y - 4 = 0 \Rightarrow y = 4 \\ X - 5 = 1 \Rightarrow x = 6. & y - 4 = 2 \Rightarrow y = 6 \\ X - 5 = 2 \Rightarrow x = 7. & y - 4 = 3 \Rightarrow y = 7 \\ X - 5 = 3 \Rightarrow x = 8. & y - 4 = 5 \Rightarrow y = 9 \end{array}$$

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

$$S^3 = (S^2 \cup S^{2_1}) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

By applying Dominance rule,

$$S^3 = (S^2 \cup S^{2_1}) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$$

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 5

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Reliability Design- CO4

The problem is to design a system that is composed of several devices connected in series. Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is $\prod r_i$. Even if the individual devices are very reliable (the r_i 's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = 0.99$, $i \leq i \leq 10$, then $\prod r_i = .904$. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains m_i copies of device D_i . Then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$.

The reliability of stage ' i ' is given by a function $\phi_i(m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.

We wish to solve:

$$\begin{aligned} &\text{Maximize } \prod_{1 \leq i \leq n} \phi_i(m_i) \\ &\text{Subject to } \sum_{1 \leq i \leq n} C_i m_i < C \\ &m_i \geq 1 \text{ and interger, } 1 \leq i \leq n \end{aligned}$$

Example 1:

Design a three stage system with device types D_1, D_2 and D_3 . The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if stage I has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \frac{C + C_i - \sum_{j=1}^n C_j}{C_i} \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3 .

$$\begin{aligned} u_1 &= \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2 \\ u_2 &= \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3 \\ u_3 &= \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3 \end{aligned}$$

We use $S_i^j \rightarrow i$: stage number and J : no. of devices in stage $i = m_i$

$$S^0 = \{f_0(x), x\} \quad \text{initially } f_0(x) = 1 \text{ and } x = 0, \text{ so, } S^0 = \{1, 0\}$$

Compute S^1, S^2 and S^3 as follows:

$S^1 =$ depends on u_1 value, as $u_1 = 2$, so

$$S^1 = \left\{ S_1^1, S_2^1 \right\}$$

$S^2 =$ depends on u_2 value, as $u_2 = 3$, so

$$S^2 = \{S_1^2, S_2^2, S_3^2\}$$

S^3 = depends on u_3 value, as $u_3 = 3$, so

$$S^3 = \{S_1^3, S_2^3, S_3^3\}$$

Now find $S_1^1 = \{(f_1(x), x)\}$

$f_1(x) = \{\phi_1(1) f_0(x), \phi_1(2) f_0(x)\}$ With devices $m_1 = 1$ and $m_2 = 2$

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_1(mi) = 1 - (1 - r_i)^{mi}$

$$\phi_1(1) = 1 - (1 - r_1)^{m_1} = 1 - (1 - 0.9)^1 = 0.9$$

$$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$$

$$S_1^1 = \{f_1(x), x\} = (0.9, 30)$$

$$S_2^1 = \{0.99, 30 + 30\} = (0.99, 60)$$

Therefore, $S^1 = \{(0.9, 30), (0.99, 60)\}$

Next find $S_1^2 = \{(f_2(x), x)\}$

$f_2(x) = \{\phi_2(1) * f_1(x), \phi_2(2) * f_1(x), \phi_2(3) * f_1(x)\}$

$$\phi_2(1) = 1 - (1 - r_{1m_1}) = 1 - (1 - 0.8) = 1 - 0.2 = 0.8$$

$$\phi_2(2) = 1 - (1 - 0.8)^2 = 0.96$$

$$\phi_2(3) = 1 - (1 - 0.8)^3 = 0.992$$

$$S_1^2 = \{(0.8(0.9), 30 + 15), (0.8(0.99), 60 + 15)\} = \{(0.72, 45), (0.792, 75)\}$$

$$S_2^2 = \{(0.96(0.9), 30 + 15 + 15), (0.96(0.99), 60 + 15 + 15)\} \\ = \{(0.864, 60), (0.9504, 90)\}$$

$$S_3^2 = \{(0.992(0.9), 30 + 15 + 15 + 15), (0.992(0.99), 60 + 15 + 15 + 15)\} \\ = \{(0.8928, 75), (0.98208, 105)\}$$

$$S^2 = \{S_1^2, S_2^2, S_3^2\}$$

By applying Dominance rule to S^2 :

Therefore, $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Dominance Rule:

If S^i contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2) , hence by dominance rule (f_2, x_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded from S^i .

Case 1: if $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

Case 2: if $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

Case 3: otherwise simply write (f_1, x_1)

$$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$$

$$\phi_3(1) = 1 - (1 - r_I)^{m_1} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$$

$$\phi_3(2) = 1 - (1 - 0.5)^2 = 0.75$$

$$\phi_3(3) = 1 - (1 - 0.5)^3 = 0.875$$

$$S^3_1 = \{(0.5 (0.72), 45 + 20), (0.5 (0.864), 60 + 20), (0.5 (0.8928), 75 + 20)\}$$

$$S^3_1 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$$

$$S^3_2 = \{(0.75 (0.72), 45 + 20 + 20), (0.75 (0.864), 60 + 20 + 20), (0.75 (0.8928), 75 + 20 + 20)\}$$

$$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$$

$$S^3_3 = \{(0.875 (0.72), 45 + 20 + 20 + 20), (0.875 (0.864), 60 + 20 + 20 + 20), (0.875 (0.8928), 75 + 20 + 20 + 20)\}$$

$$S^3_3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$$

If cost exceeds 105, remove that tuples

$$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i 's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 6

Unit - IV

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

BACKTRACKING -CO4

General Method:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x_1, \dots, x_n) where each $x_i \in S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, \dots, x_n)$. Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I .

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i 's must relate to each other.

- For 8-queens problem:

Explicit constraints using 8-tuple formation, for this problem are $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Depth first node generation with bounding functions is called **backtracking**. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., 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.

N-Queens Problem:

Let us consider, $N = 8$. Then 8-Queens Problem is to place eight queens on an 8×8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x_1, \dots, x_8) , where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i 's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to $8!$ Tuples.

The promising function must check whether two queens are in the same column or

diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45° diagonal if:

$$i - j = k - l.$$

This implies, $j - l = i - k$

- Diag 135 conflict:

$$i + j = k + l.$$

This implies, $j - l = k - i$

Therefore, two queens lie on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Where, j be the column of object in row i for the i^{th} queen and l be the column of object in row ' k ' for the k^{th} queen.

To check the diagonal clashes, let us take the following tile configuration:

	*						
				*			
*							
							*
			*				
						*	
		*					
					*		

In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

case whether the queens on

Let us consider for the 3^{rd} row and 8^{th} row

are conflicting or not. In this case $(i, j) = (3, 1)$ and $(k, l) = (8, 6)$. Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8| \\ \Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.

						*	
				*			
		*					
*							

Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less than 8, repeat Step 1.

Step 3:

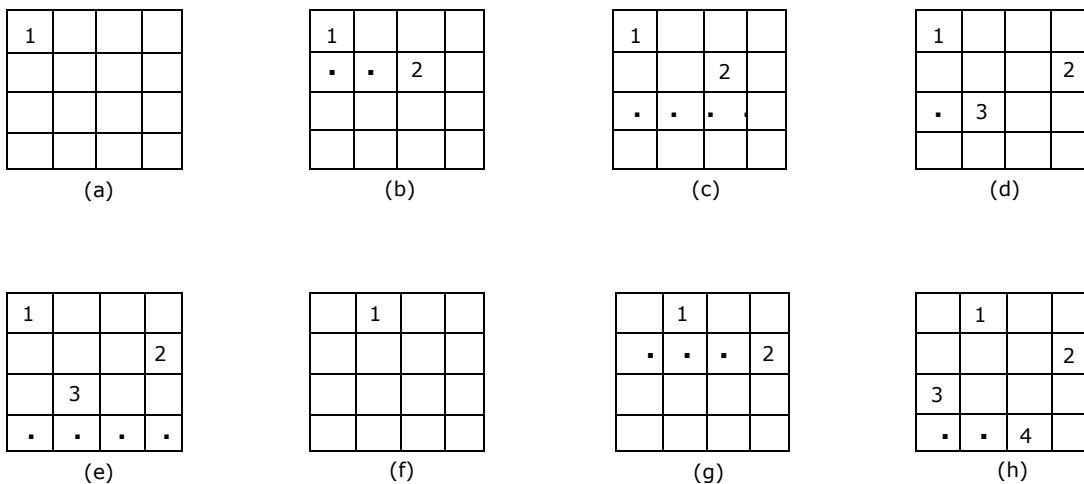
If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

On a chessboard, the **solution** will look like:

					*		
				*			
		*					
*							
					*		
							*
	*						
			*				

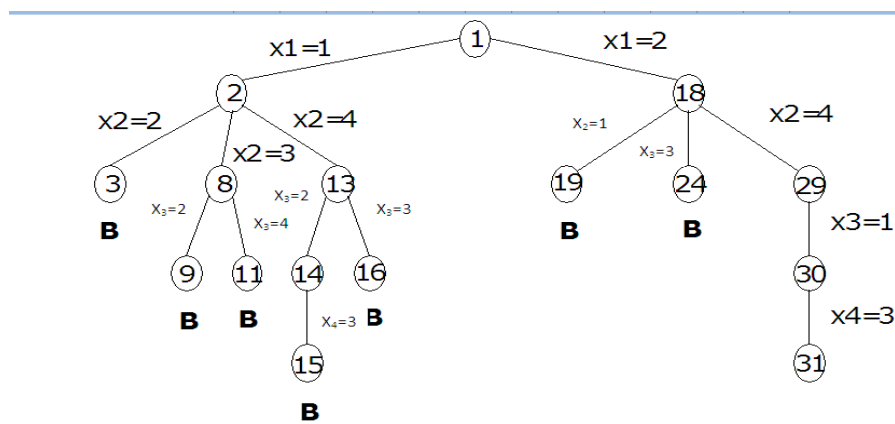
4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 7

Unit - IV

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Sum of Subsets-CO4

Given positive numbers w_i , $1 \leq i \leq n$, and m , this problem requires finding all subsets of w_i whose sums are m .

All solutions are k -tuples, $1 \leq k \leq n$.

Explicit constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i 's be m .
- $x_i < x_{i+1}$, $1 \leq i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

A better formulation of the problem is where the solution subset is represented by an n -tuple (x_1, \dots, x_n) such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distinct tuples.

For example, $n = 4$, $w = (11, 13, 24, 7)$ and $m = 31$, the desired subsets are (11, 13, 7) and (24, 7).

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i . At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left most sub-tree defines all subsets containing w_1 , the next sub-tree defines all subsets containing w_2 but not w_1 , and so on.

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the 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. This is termed the m -colorability decision problem. The m -colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array $x []$ to zero. The colors are represented by the integers $1, 2, \dots, m$ and the solutions are given by the n-tuple (x_1, x_2, \dots, x_n) , where x_i is the color of node i .

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement `mcoloring(1)`;

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 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;
        }
    }
}
```

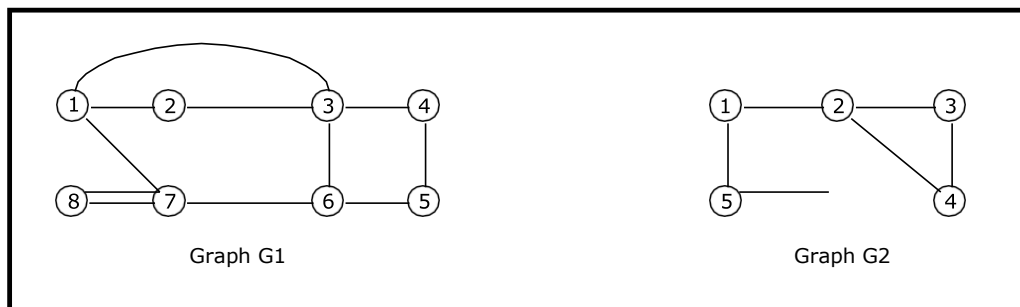
```

    }
    if (j = n+1) then return;           // New color found
} until (false);                       // Otherwise try to find another color.
}

```

Hamiltonian Cycles:

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by 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 vertices of G are visited in the order v_1, v_2, \dots, v_{n+1} , then the edges (v_i, v_{i+1}) are in E , $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1} , which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector (x_1, \dots, x_n) is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from x_1, x_2, \dots, x_{k-1} and v is connected by an edge to x_{k-1} . The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1 .

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix $G[1: n, 1: n]$, then setting $x[2: n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

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

```

```

        {

```

```

            for j := 1 to k - 1 do if (x [j] = x [k]) then break;

```

```

            // Is there an edge?
            // 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 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  
  {  
    NextValue (k); // Generate values for x [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);  
}
```



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1

Unit - V

Year and Semester: IYr & II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

Branch and Bound General method: CO5

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node

Branch and Bound is the generalization of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Definition 1: Live node is a node that has been generated but whose children have not yet been generated.

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Definition 4: Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Definition 5: The adjective "heuristic", means "related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node is rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where, $\hat{c}(x)$ is the cost of x .

$h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any non-decreasing function.

$\hat{g}(x)$ is an estimate of the additional effort needed to reach an answer node from x .

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$ is called a LC-search (Least Cost search)

BFS and D-search are special cases of LC-search. If $\hat{g}(x) = 0$ and $f(h(x)) = \text{level of node } x$, then an LC search generates nodes by levels. This is eventually the same as a BFS. If $f(h(x)) = 0$ and $\hat{g}(x) > \hat{g}(y)$ whenever y is a child of x , then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost $c(x)$ with each node x in the state space tree. It is not possible to easily compute the function $c(x)$. So we compute a estimate $\hat{c}(x)$ of $c(x)$.

Control Abstraction for LC-Search:

Let t be a state space tree and $c(\cdot)$ a cost function for the nodes in t . If x is a node in t , then $c(x)$ is the minimum cost of any answer node in the subtree with root x . Thus, $c(t)$ is the cost of a minimum-cost answer node in t .

A heuristic $\hat{c}(\cdot)$ is used to estimate $c(\cdot)$. This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$.

LC-search uses \hat{c} to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

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

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

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);
}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x . If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $c(x) \geq \hat{c}(x) > \text{upper}$. The starting value for upper can be obtained by some heuristic or can be set to ∞ .

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function $c(\cdot)$, such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(\cdot)$.

- For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution.
- For nodes representing infeasible solutions, $c(x) = \infty$.
- For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x .

Since, $c(x)$ is generally hard to compute, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x .



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 2

Unit - V

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

FIFO Branch and Bound:CO5

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with $upper = \infty$ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$.

The variable upper is updated to 14 when node 3 is generated. Since $c(4)$ and $c(5)$ are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated. Then $u(6) = 9$ and so upper is updated to 9. The cost $c(7) = 10 > upper$ and node 7 gets killed. Node 8 is infeasible and so it is killed.

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then $u(9) = 8$ and so upper becomes 8. The cost $c(10) = 11 > upper$, and this node is killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $c(x) > \text{upper}$ each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $c(x) > \text{upper}$ are distributed in some random way in the queue. Instead, live nodes with $c(x) > \text{upper}$ can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(\cdot)$ and $u(\cdot)$ is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with $\text{upper} = \infty$ and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4) > \text{upper}$ and $\hat{c}(5) > \text{upper}$.

Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10 > \text{upper}$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

The path = $1 \xrightarrow{2} 3 \xrightarrow{3} 9 = 5 + 3 = 8$

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let $G = (V, E)$ is a connected graph. Let $C(i, j)$ be the cost of edge $\langle i, j \rangle$. $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$ and let $|V| = n$, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by $S = \{1, \pi, 1 \mid \pi \text{ is a}$

permutation of $(2, 3, \dots, n)$ and $|S| = (n - 1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \dots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

- a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
- b) Find the sum of elements, which were subtracted from rows.
- c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

- d) Find the sum of elements, which were subtracted from columns.
- e) Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R . Let A is the reduced cost matrix for node R . Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i, j \rangle$ in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

- a) Change all entries in row i and column j of A to ∞ .
- b) Set $A(j, 1)$ to ∞ .
- c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞ . Let r is the total amount subtracted to reduce the matrix.

c) Find $c(S) = c(R) + A(i, j) + r$, where ' r ' is the total amount subtracted to reduce the matrix, $c(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $c(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3

Unit - V

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

0/1 Knapsack Problem: CO5

Consider the instance: $M = 15, n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ and $(W_1, W_2, W_3, W_4) = (2, 4, 6, 9)$.

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is $15 - 2 = 13$. Place next item w_2 in knapsack and the remaining weight of knapsack is $13 - 4 = 9$. Place next item w_3 in knapsack then the remaining weight of knapsack is $9 - 6 = 3$. No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

$$\text{Profit} = P_1 + P_2 + P_3 = 10 + 10 + 12$$

$$\text{So, Upper bound} = 32$$

To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

$$\text{Lower bound} = 10 + 10 + 12 + \left(\frac{3}{9} \times 18\right) = 32 + 6 = 38$$

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.



CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 4

Unit - V

Year and Semester: IIyr &II Sem

Subject: **Design and Analysis of Algorithms**

Branch: **CSE**

Faculty: **Dr. N. Subhash Chandra**, Professor of CSE

NP Hard and NP-Complete: CO 5

Basic concepts:

NP □ Nondeterministic Polynomial time

The problems has best algorithms for their solutions have "Computing times", that cluster into two groups

Group 1	Group 2
<ul style="list-style-type: none"> ➤ Problems with solution time bound by a polynomial of a small degree. ➤ It also called "Tractable Algorithms" ➤ Most Searching & Sorting algorithms are polynomial time algorithms ➤ Ex: Ordered Search ($O(\log n)$), Polynomial evaluation $O(n)$ Sorting $O(n \cdot \log n)$ 	<ul style="list-style-type: none"> ➤ Problems with solution times not bound by polynomial (simply non polynomial) ➤ These are hard or intractable problems ➤ None of the problems in this group has been solved by any polynomial time algorithm ➤ Ex: Traveling Sales Person $O(n^2)$ Knapsack $O(2^{n/2})$

No one has been able to develop a polynomial time algorithm for any problem in the 2nd group (i.e., group 2)

So, it is compulsory and finding algorithms whose computing times are greater than polynomial very quickly because such vast amounts of time to execute that even moderate size problems cannot be solved.

Theory of NP-Completeness:

Show that may of the problems with no polynomial time algorithms are computational time algorithms are computationally related.

There are two classes of non-polynomial time problems

1. NP-Hard
2. NP-Complete

NP Complete Problem: A problem that is NP-Complete can be solved in polynomial time if and only if (iff) all other NP-Complete problems can also be solved in polynomial time.

NP-Hard: Problem can be solved in polynomial time then all NP-Complete problems can be solved in polynomial time.

All NP-Complete problems are NP-Hard but some NP-Hard problems are not known to be NP-Complete.

Nondeterministic Algorithms:

Algorithms with the property that the result of every operation is uniquely defined are termed as deterministic algorithms. Such algorithms agree with the way programs are executed on a computer.

Algorithms which contain operations whose outcomes are not uniquely defined but are limited to a specified set of possibilities. Such algorithms are called nondeterministic algorithms.

The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.

To specify nondeterministic algorithms, there are 3 new functions. Choice(S) □ arbitrarily chooses one of the elements of sets S Failure () □ Signals an Unsuccessful completion

Success () □ Signals a successful completion.

Example for Non Deterministic algorithms:

<p>Algorithm Search(x){ //Problem is to search an element x //output J, such that A[J]=x; or J=0 if x is not in A J:=Choice(1,n); if(A[J]:=x) then { Write(J); Success() ; } else{ write(0) ; failure() ; }</p>	<p>Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates.</p> <p>A Nondeterministic algorithm terminates unsuccessfully if and only if (iff) there exists no set of choices leading to a successful signal.</p>
---	---

Nondeterministic Knapsack algorithm	
<p>Algorithm DKP(p, w, n, m, r, x){ W:=0; P:=0; for i:=1 to n do{ x[i]:=choice(0, 1); W:=W+x[i]*w[i]; P:=P+x[i]*p[i]; } if((W>m) or (P<r)) then Failure(); else Success(); }</p>	<p>p□ given Profits w□ given Weights n□ Number of elements (number of p or w) m□ Weight of bag limit P□Final Profit W□Final weight</p>

The Classes NP-Hard & NP-Complete:

For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity $p()$ such that the computing time of A is $O(p(n))$ for every input of size n .

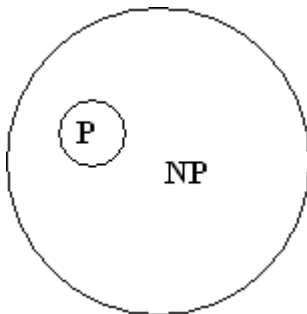
Decision problem/ Decision algorithm: Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

Optimization problem/ Optimization algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

P is the set of all decision problems solvable by deterministic algorithms in polynomial time.

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, by this we concluded that $P \subseteq NP$



Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether $P=NP$ or $P \neq NP$ In considering this problem, s.cook formulated the following question.

If there any single problem in NP, such that if we showed it to be in 'P' then that would imply that $P=NP$.

Cook answered this question with

Theorem: Satisfiability is in P if and only if (iff) $P=NP$

□ Notation of Reducibility

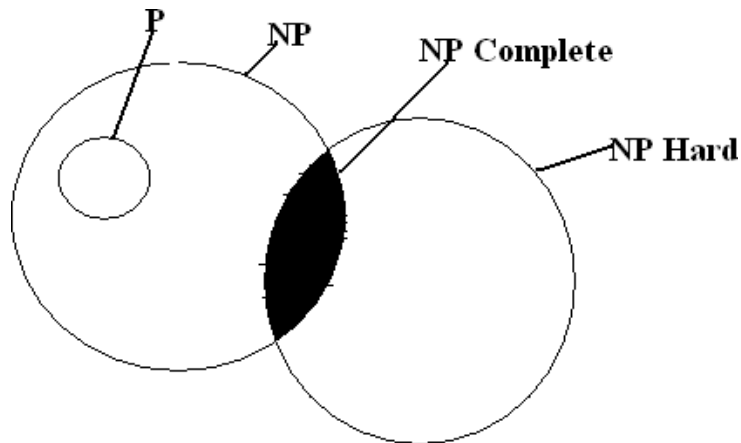
Let L_1 and L_2 be problems, Problem L_1 reduces to L_2 (written $L_1 \alpha L_2$) iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time

This implies that, if we have a polynomial time algorithm for L_2 , Then we can solve L_1 in polynomial time.

Here α is a transitive relation i.e., $L_1 \alpha L_2$ and $L_2 \alpha L_3$ then $L_1 \alpha L_3$

A problem L is NP-Hard if and only if (iff) satisfiability reduces to L i.e., **Satisfiability α L**

A problem L is NP-Complete if and only if (iff) L is NP-Hard and **$L \in NP$**



Commonly believed relationship among P, NP, NP-Complete and NP-Hard

Most natural problems in NP are either in P or NP-complete.

Examples of NP-complete problems:

- Packing problems: SET-PACKING, INDEPENDENT-SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3-COLOR, CLIQUE.
- Constraint satisfaction problems: SAT, 3-SAT.
- Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK.

Cook's Theorem: States that satisfiability is in P if and only if

$P=NP$ If $P=NP$ then satisfiability is in P

If satisfiability is in P, then

$P=NP$ To do this

- $A \square$ Any polynomial time nondeterministic decision algorithm. $I \square$ Input of that algorithm
Then formula $Q(A, I)$, Such that Q is satisfiable iff ' A ' has a successful termination with Input I .
- If the length of ' I ' is ' n ' and the time complexity of A is $p(n)$ for some polynomial $p()$ then length of Q is $O(p^3(n) \log n) = O(p^4(n))$
The time needed to construct Q is also $O(p^3(n) \log n)$.
- A deterministic algorithm ' Z ' to determine the outcome of ' A ' on any input ' I '
Algorithm Z computes ' Q ' and then uses a deterministic algorithm for the satisfiability problem to determine whether ' Q ' is satisfiable.
- If $O(q(m))$ is the time needed to determine whether a formula of length ' m ' is satisfiable then the complexity of ' Z ' is $O(p^3(n) \log n + q(p^3(n) \log n))$.
- If satisfiability is ' p ', then ' $q(m)$ ' is a polynomial function of ' m ' and the complexity of ' Z ' becomes ' $O(r(n))$ ' for some polynomial ' $r()$ '.
- Hence, if satisfiability is in p , then for every nondeterministic algorithm A in NP , we can obtain a deterministic Z in p .

By this we shows that satisfiability is in p then $P=NP$