DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DIGITAL NOTES

ON

DESIGN AND ANALYSIS OF ALGORITHMS

R22A0506

B. TECH II YEAR–I SEM

(R22) REGULATION

(2024-25)

Prepared by

P Honey Diana, Asst.Professor

MALLA REDDY COLLEGE OF ENGINEERING &TECHNOLOGY

(Autonomous Institution–UGC, Govt.of India)

Recognized under2(f)and12(B) of UGC ACT1956

(Affiliated to JNTUH ,Hyderabad,ApprovedbyAICTE-AccreditedbyNBA&NAAC–'A'Grade-ISO9001:2015Certified)

Maisammaguda, Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech.CSE- I Sem **L/T/P/C**
3/1/-/4

(R22A0506)

DESIGN AND ANALYSIS OF ALGORITHMS

COURSE OBJECTIVES:

1. To analyze performance of algorithms.
2. To choose the appropriate data structure and algorithm design method for a specified application.
3. To understand how the choice of data structures and algorithm design methods impacts the performance of programs.
4. To solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking and branch and bound.
5. To understand the differences between tractable and intractable problems and to introduce P and NP classes.

UNIT I

Introduction: Algorithms, Pseudocode for expressing algorithms, performance analysis-Space complexity, Time Complexity, Asymptotic notation- Big oh notation, omega notation, theta notation and little oh notation.
Divide and Conquer: General method. Applications- Binary search, Quick sort, merge sort, Strassen's matrix multiplication.

UNIT II

Disjoint set operations, Union and Find algorithms, AND/OR graphs, Connected components, Bi-connected components.
Greedy method: General method, applications-Job sequencing with deadlines, Knapsack problem, Spanning trees, Minimum cost spanning trees, Single source shortest path problem.

UNIT III

Dynamic Programming: General method, applications-Matrix chained multiplication, Optimal binary search trees,0/1 Knapsack problem, All pairs shortest path problem, Traveling sales person problem.

UNIT IV

Backtracking: General method Applications-n-queues problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

UNIT V

Branch and Bound: General method, applications- Travelling sales person problem,0/I k Knapsack problem LC branch and Bound solution, FIFO branch and bound solution.

NP-Hard and NP-Complete Problems: Basic concepts, Non deterministic algorithms, NP-Hard and NP-Complete classes, NP-Hard problems,Cook'stheorem.

TEXTBOOKS:

1. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahni and Rajasekharan, Universities press
2. Design and Analysis of Algorithms, P.h.Dave,2ndedition,PearsonEducation.

REFERENCES:

1. Introduction to the Design And Analysis of Algorithms ALevitin Pearson Education
2. Algorithm Design foundations Analysis and Internet examples, M.T.Goodrich and R Tomassia John Wiley and sons
3. Design and Analysis of Algorithms, S.Sridhar, Oxford Univ.Press
4. Design and Analysis of Algorithms, Aho,Ulman and Hopcraft, Pearson Education.
5. Foundations of Algorithms,R.NeapolitanandK.Naimipour,4thedition

COURSE OUTCOMES:

1. Ability to analyze the performance of algorithms.
2. Ability to choose appropriate algorithm design techniques for solving problems.
3. Ability to understand how the choice of data structures and the algorithm design methods to impact the performance of programs.
4. Describe the dynamic programming paradigm and explain when an algorithmic design situation calls for it. Synthesize dynamic programming algorithms and analyze them.
5. Describes NP hard and NP complete classes and also about the importance of Cook's theorem.

INDEX

INDEX

UNIT I:
Introduction- Algorithm definition, Algorithm Specification, Performance Analysis- Space complexity, Time complexity, Randomized Algorithms.
Divide and conquer- General method, applications - Binary search, Merge sort, Quick sort, Strassen′s Matrix Multiplication.

Algorithm:

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

- **Input:** there are zero or more quantities, which are externally supplied;
- **Output:** at least one quantity is produced
- **Definiteness:** each instruction must be clear and unambiguous;
- **Finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
- **Effectiveness:** every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs areentered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Psuedo code for expressing algorithms:

Algorithm Specification: Algorithm can be described in three ways.

1. Natural language like English: When this way is choused care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
   data type – 1  data-1;
            .
            .
            .
  data type – n data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

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

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and FALSE.

→ Logical Operators AND, OR, NOT
→Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
                .
                .
                .

        <statement-n>
}
```

For Loop:

```
For variable: = value-1 to value-2 step step do

{
  <statement-1>
          .
          .
          .
<statement-n>
}
```

repeat-until:

```
repeat
 <statement-1>
 .
      .
      .
                        <statement-n>
               until<condition>
```

8. A conditional statement has the following forms.

   → If <condition> then <statement>

   → If <condition> then <statement-1>
   Else <statement-1>

   Case statement:

```
Case
{
        : <condition-1> : <statement-1>
                          .
                          .
                          .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
    Algorithm, the heading takes the form,

    Algorithm <Name> (<Parameter lists>)

    → As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
1.   Algorithm Max(A,n)
2.   // A is an array of size n
3. {
4.   Result := A[1];
5.   for I:= 2 to n do
6.      if A[I] > Result then
7.            Result :=A[I];
8.    return Result;
9. }
```

    In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing
order. 3.{
4.        for I:=1 to n do
5.        {
6.                j:=I;
7.                for k:=i+1 to n do
8.                        if (a[k]<a[j])
9.                        t:=a[I];
10.                       a[I]:=a[j];
11.                       a[j]:=t;
12.       }
13. }
```

<u>Performance Analysis:</u>

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

i. We would like an algorithm that is easy to understand code anddebug.
ii. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.
2. The quality of code generated by the compiler used to create the object program.
3. The nature and speed of the instructions on the machine used to execute the program,
4. The time complexity of the algorithm underlying theprogram.

Statement	*S/e*	*Frequency*	*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

The total time will be 2n+3

Space Complexity:

The space complexity of a program is the amount of memory it needs to run tocompletion. The space need by a program has the following components: **Instruction space:** Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and classinstances.

Environment stack space: The environment stack is used to save informationneeded to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends onfactors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where ‘c’ is a constant.

Example 2:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
    return s;
}
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
- So,we obtain Ssum(n)>=(n+s)

[n for a[],one each for n,I a& s]

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the inputdata. Mostly, the storage space required by an algorithm is simply a multiple of the data size ‘n’. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size ‘n’ of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

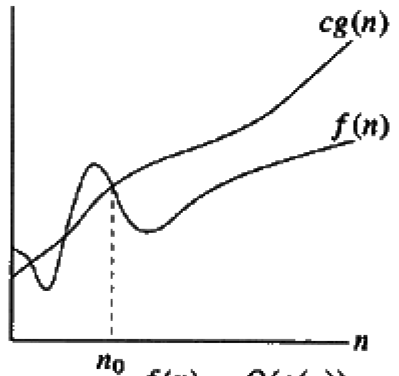
Asymptotic Notations:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH (O)
2. Big–OMEGA (Ω),
3. Big–THETA (*Θ*) and
4. Little–OH (o)

Big–OH O (Upper Bound)

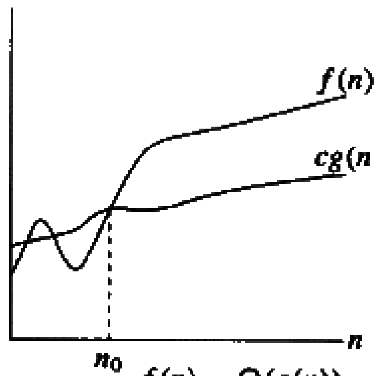
f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal ($\leq$) that of g(n).



$f(n) = O(g(n))$

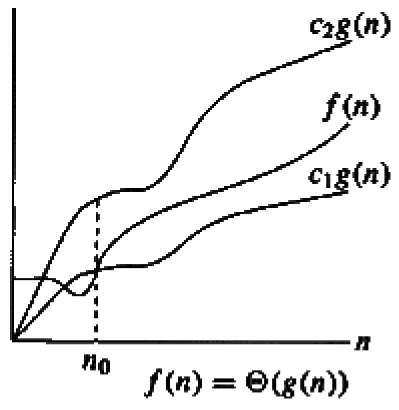
Big–OMEGA Ω (Lower Bound)

f(n) **=** Ω ***(g(n))*** (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



$f(n) = \Omega(g(n))$

Big–THETA Θ (Same order)
f(n) = Θ (g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = ***Θ*** (g(n)].



little-o notation

Definition: A theoretical measure of the execution of an *algorithm*, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity. The notation is read, "fof n is little oh of g of n".
Formal Definition: f(n) = o(g(n)) means for all c > 0 there exists some k > 0 such that $0 \leq f(n) < cg(n)$ forall $n \geq k$. The value of k must not depend on n, but may depend on c.

Different time complexities
Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearlythe complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O($\log_2$ n), O(n), O(n. $\log_2$ n), O(n^2), O(n^3), O(2^n), n! and n^n

Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1	Next instructions of most programs are executed once or at most only a fewtimes. If all the instructions of a program have this property, we say that its running time is a constant.
Log n	When the running time of a program is logarithmic, the program getsslightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.
n	When the running time of a program is linear, it is generally the case that a

small amount of processing is done on each input element. This is theoptimal situation for an algorithm that must process n inputs.

n log n This running time arises for algorithms that solve a problem by breaking itup into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases fourfold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems. Whenever n doubles, the running timesquares.

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	**log2 n**	**n*log2n**	**n^2**	**n^3**	**2^n**
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Divide and Conquer

General Method:

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 $p_1$, $p_2$, …. $P_k$, $k \geq 1$;
                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, . . . , 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:

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

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 $x_1 < x_2 < \ldots < x_n$. 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[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[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[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 $\mathbf{\log_2 n}$

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.

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

Recursive binary search

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

Iterative binary search

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches			**un-successful searches**
Θ(1),	Θ(log n),	Θ(log n)	Θ(log n)
Best	average	worst	best, average and worst

Analysis for worst case

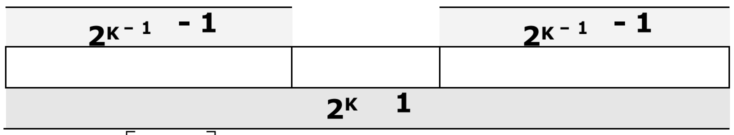
Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

$$
\begin{aligned}
T(0) &= 0 \\
T(n) &= 1 && \text{if } x = a\,[mid] \\
&= 1 + T([(n + 1) / 2] - 1) && \text{if } x < a\,[mid] \\
&= 1 + T(n - [(n + 1)/2]) && \text{if } x > a\,[mid]
\end{aligned}
$$

Let us restrict 'n' to values of the form n = $2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2^K - 1 + 1}{2} \right\rfloor = 2^{K-1}$ for K > 1

Giving,

$$
\begin{aligned}
T(0) &= 0 \\
T(2^k - 1) &= 1 && \text{if } x = a\,[mid] \\
&= 1 + T(2^{K-1} - 1) && \text{if } x < a\,[mid] \\
&= 1 + T(2^{k-1} - 1) && \text{if } x > a\,[mid]
\end{aligned}
$$

In the worst case the test x = a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

$$= \quad 1 + [1 + w(2^{k-2} - 1)]$$
$$= \quad 1 + [1 + [1 + w(2^{k-3} - 1)]]$$
$$= \quad \ldots\ldots\ldots$$
$$= \quad \ldots\ldots\ldots$$
$$= \quad i + w(2^{k-i} - 1)$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

Merge Sort:

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 mort 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.

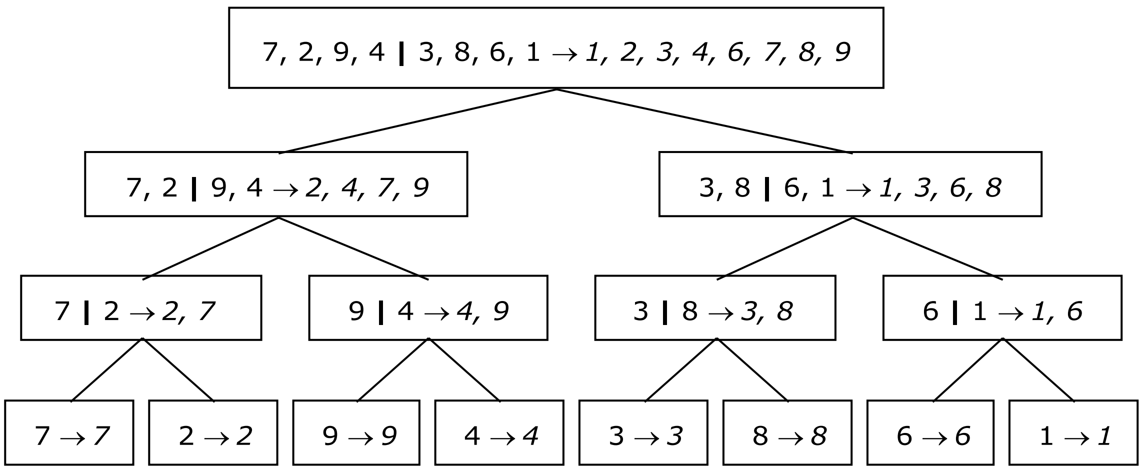
Algorithm

```
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 + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

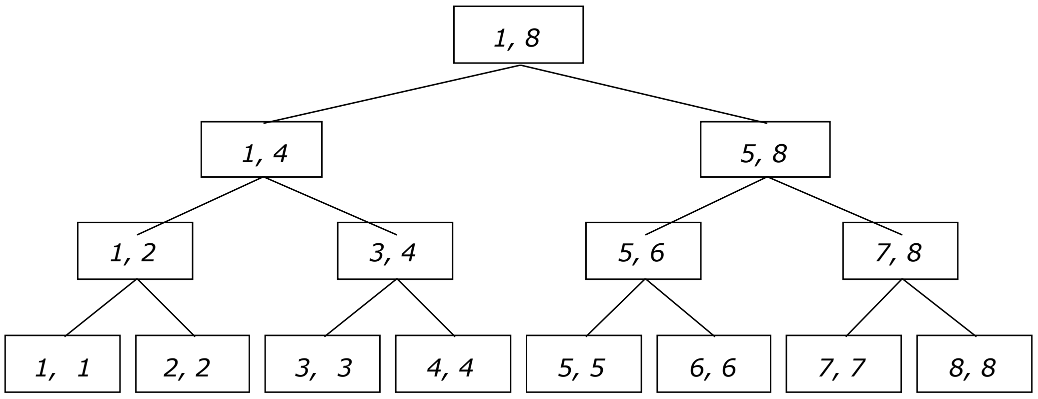
Example

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



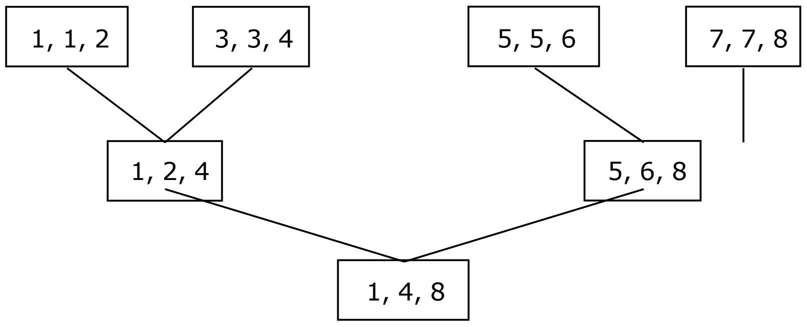
Tree Calls of MERGESORT(1, 8)

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 be 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:

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

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

$$\begin{aligned} T(n) &= 2^{\log_2 n}\,T\left(\frac{2^k}{2}\right) + \log_2 n\,.\,n \\ &= n\,T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

T(n) = **O(n log n)**

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication ofsingle numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$

$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$

$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$

$C_{22} = A_{21} . B_{12} + A_{22} . B_{22}$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$T(1) = 1$
$T(n) = 8\,T(n/2)$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})\ B_{11}$

$R = A_{11}\ (B_{12} - B_{22})$

$S = A_{22}\ (B_{21} - B_{11})$

$T = (A_{11} + A_{12})\ B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$T(1) = 1$$
$$T(n) = 7\,T(n/2)$$

Solving this for the case of $n = 2^k$ is easy:

$$T(2^k) = 7\,T(2^{k-1})$$
$$= 7^2\,T(2^{k-2})$$
$$= \text{- - - - - -}$$
$$= \text{- - - - - -}$$
$$= 7^i\,T(2^{k-i})$$

Put i = k

$$= 7^k\,T(1)$$
$$= 7^k$$

That is, $T(n) = 7^{\log_2 n}$

$$= n^{\log_2 7}$$
$$= O(n^{\log_2 7}) = O(^2n^{.81})$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ 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 his 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.

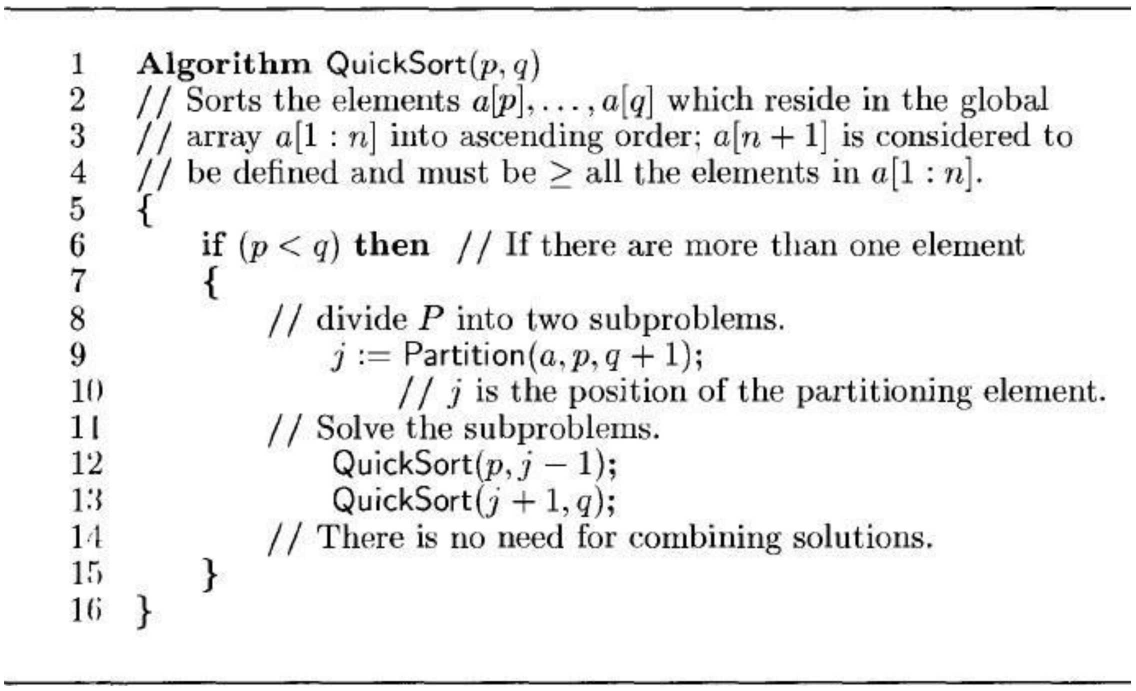
The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

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

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

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

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

    } until ($i \geq j$);

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

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

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j

					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	
							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
											pivot, j	i	swap pivot & j
											70		
												79 pivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

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:

$$T(n) = T(i) + T(n - i - 1) + C\,n \qquad - \qquad (1)$$

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:

$$T(n) = T(n - 1) + C\,n \qquad n > 1 \qquad - \qquad (2)$$

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

$$T(n) = T(1) + \sum_{i=2}^{n} i$$

$$= \mathbf{O(n^2)} \qquad - \qquad (3)$$

Best and 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) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

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

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

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 subsititution:

$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$
$T(n-1)/n = 2/n + T(n-2)/(n-1)$
$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$
$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$
. .
. .
$T(3)/4 = 2/4 + T(2)/3$
$T(2)/3 = 2/3 + T(1)/2$ $T(1)/2 = 2/2 + T(0)$

Adding both sides:
$T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + --------------+ T(2)/3 + T(1)/2]$
$= [T(n-1)/n + T(n-2)/(n-1) + --------------+ T(2)/3 + T(1)/2] + T(0) +$
$[2/(n+1) + 2/n + 2/(n-1) + ----------- +2/4 + 2/3]$
Cancelling the common terms:
$T(n)/(n+1) = 2[1/2 +1/3 +1/4+ --------------- +1/n+1/(n+1)]$

$T(n) = (n+1)2[\sum_{2 \le k \le n+1} 1/k$

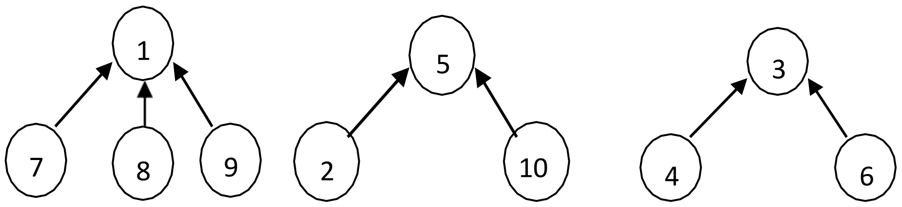
=2(n+1) [–]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2

T(n)= O(n log n)

UNIT II:
Disjoint set operations, union and find algorithms, AND/OR graphs, Connected Components and Spanning trees, Bi-connected components.
Greedy method: General method, applications- Job sequencing with deadlines, Knapsack problem, Spanning trees, Minimum cost spanning trees, Single source shortest path problem.

Sets and Disjoint Set Union:

Disjoint Set Union: Considering a set S={1,2,3…10} (when n=10), then elements can be partitioned into three disjoint sets s1={1,7,8,9},s2={2,5,10} and s3={3,4,6}. Possible tree representations are:



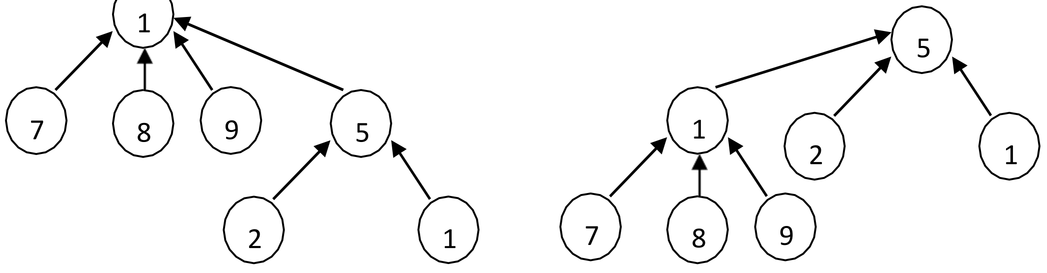
In this representation each set is represented as a tree. Nodes are linked from the child toparent rather than usual method of linking from parent to child.

The operations on these sets are:
1. Disjoint set union
2. Find(i)
3. Min Operation
4. Delete
5. Intersect

1. **Disjoint Set union:**
   If S_i and S_j are two disjoint sets, then their union S_i U S_j = all the elements X such that x is in S_i or S_j. Thus S_1 U S_2 ={1,7,8,9,2,5,10}.
2. **Find(i):**
   Given the element I, find the set containing i. Thus, 4 is in set S_3, 9 is in S_1.

UNION operation:

Union(i,j) requires two tree with roots i and j be joined. S_1 U S_2 isobtained by making any one of the sets as sub tree of other.



Simple Algorithm for Union:

```
Algorithm Union(i,j)
{
//replace the disjoint sets with roots i and j, I not equal to j by theirunion Integer i,j;
P[j] :=i;
}
```

Example:

Implement following sequence of operations Union(1,3),Union(2,5),Union(1,2)

Solution:

Initially parent array contains zeros.

0	0	0	0	0	0
1	2	3	4	5	6

1. After performing union(1,3)operation Parent[3]:=1

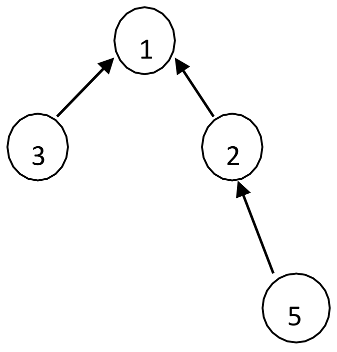
0	0	1	0	0	0
1	2	3	4	5	6

2. After performing union(2,5)operation Parent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

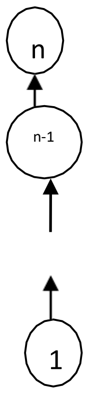
3. After performing union(1,2)operation Parent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations Union(1,2),Union(2,3) Union(n-1,n)

Degenerate Tree:



The time taken for n-1 unions is O(n).

Find(i) operation: determines the root of the tree containing

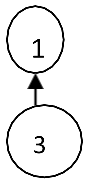
element i. Simple Algorithm for Find:

Algorithm Find(i)

```
{
    j:=i;
    while(p[j]>0) do
    j:=p[j]; return j;
}
```

Find Operation: Find(i) implies that it finds the root node of ith node, in other words it returns the name of the set i.

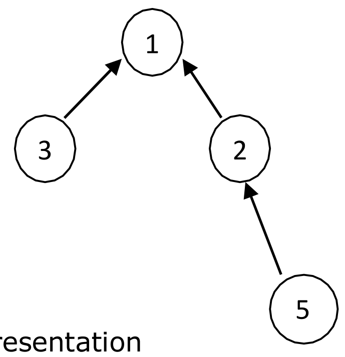
Example: Consider the Union(1,3)



Find(1)=0
Find(3)=1, since its parent is 1. (i.e, root is 1)

Example:

Considering



Array Representation

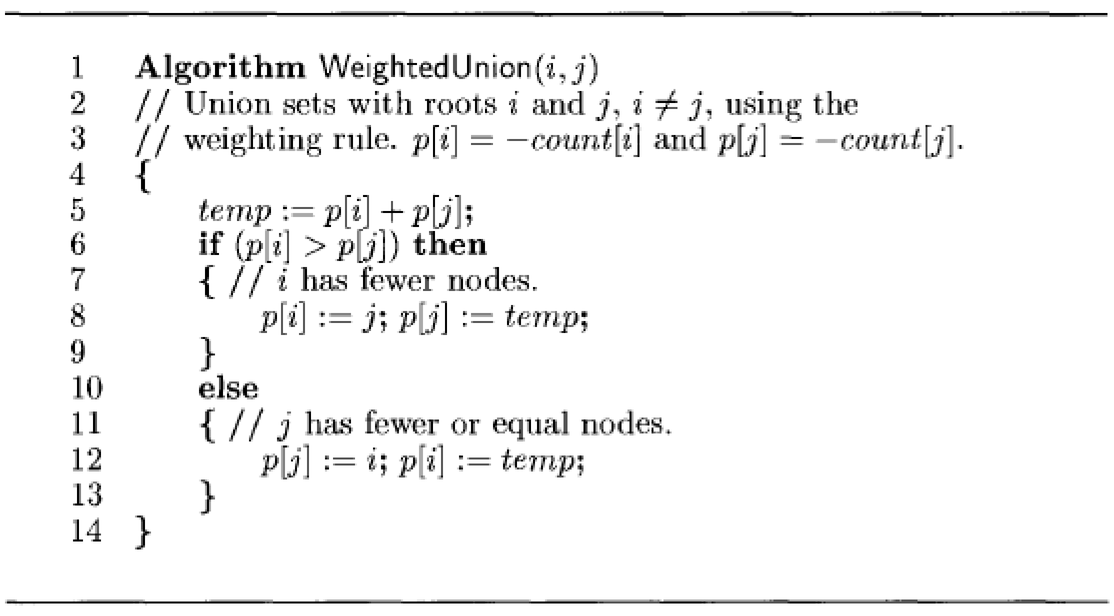
P[i]	0	1	1	2
i	1	2	3	5

Find(5)=1
Find(2)=1
Find(3)=1

The root node represents all the nodes in the tree. Time Complexity of ‘n’ find operations is O(n^2).

To improve the performance of union and find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

Weighting Rule for Union(i,j)

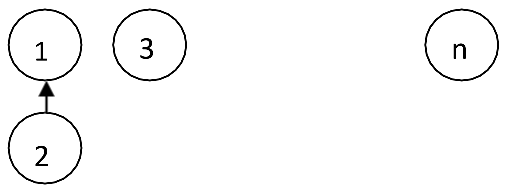


```
Algorithm WeightedUnion($i, j$)
// Union sets with roots $i$ and $j$, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if ($p[i] > p[j]$) then
    { // $i$ has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // $j$ has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}
```

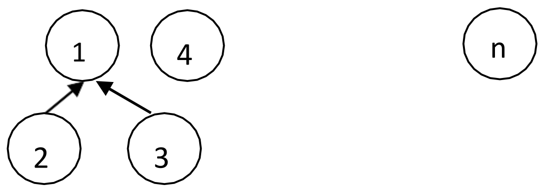
Union algorithm with weighting rule

Tree obtained withweighted Initially

Union(1,2)



Union(1,3)

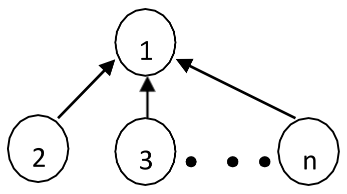


:

:

:

Union(1,n)



Collapsing Rule for Find(i)

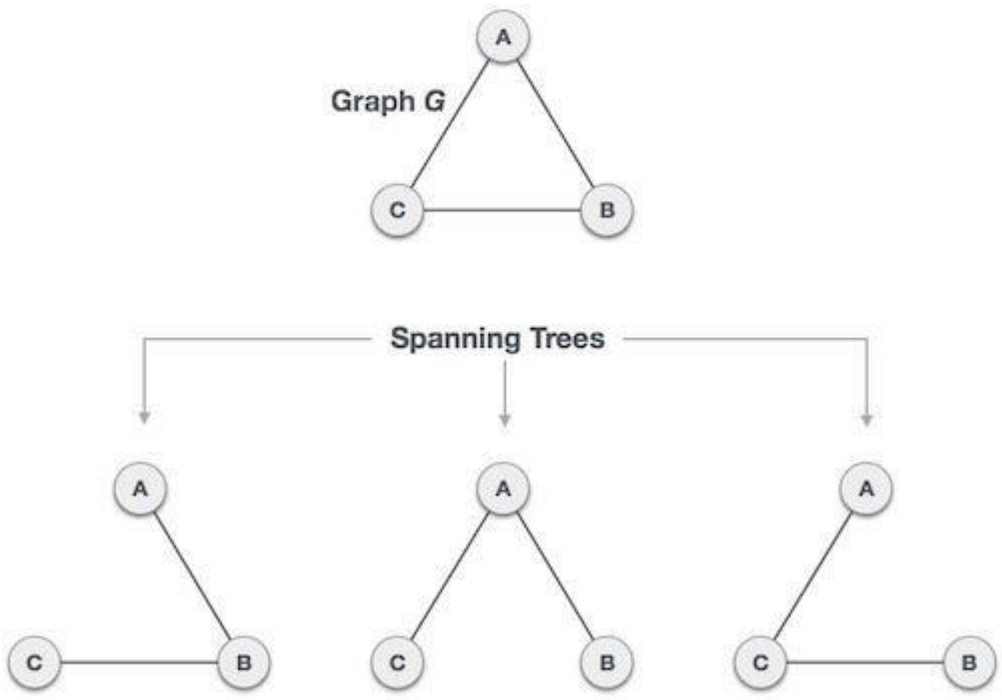
```
Algorithm CollapsingFind($i$)
// Find the root of the tree containing element $i$. Use the
// collapsing rule to collapse all nodes from $i$ to the root.
{
    $r := i$;
    while ($p[r] > 0$) do $r := p[r]$; // Find the root.
    while ($i \neq r$) do  // Collapse nodes from $i$ to root $r$.
    {
        $s := p[i]$; $p[i] := r$; $i := s$;
    }
    return $r$;
}
```

Find algorithm with collapsing rule

Spanning Trees:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$spanning trees are possible.

GeneralPropertiesofSpanningTree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

MathematicalPropertiesof SpanningTree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

ApplicationofSpanningTree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

AND/OR GRAPH:

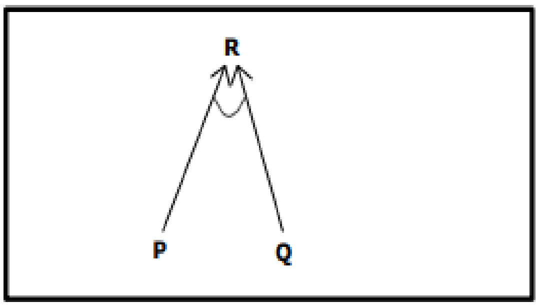
And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of: N, a set of nodes,

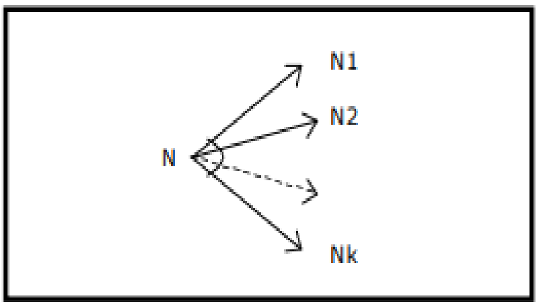
H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.

An ordinary graph is a special case of hypergraph in which all the sets of decendent 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. And/or graph for the expression P and Q -> R is follows:



E xpre s s io n f or P a n d Q -> R



A K- Connector

The K-connector is represented as a fan of arrows with a single tie is shown above. The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, in order to process the compound database to termination, all the compound databases must be processed to termination. For example consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).
2. 1 winning conker (C) for a bat (B) and a stamp (M).
3. 1 bat (B) for two stamps (M, M).
4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)
2. Transformation rules:
   a. If C then (D, S)
   b. If C then (B, M)
   c. If B then (M, M)
   d. If A then (B, B, M)
3. The goal state is to left with only stamps (M,............ , M)

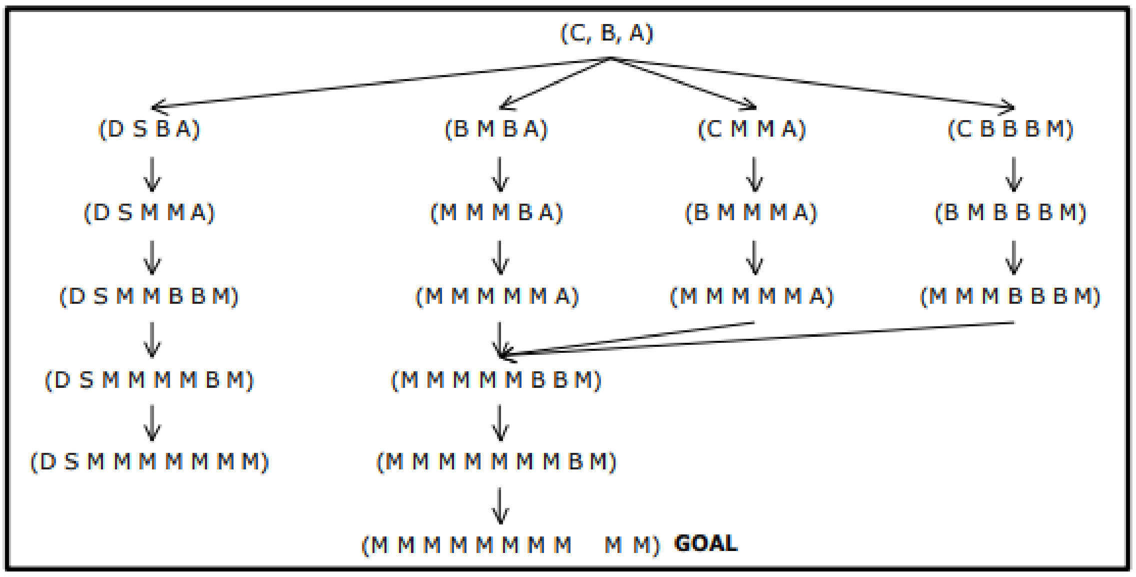
The figure shows that, a lot of extra work is done by redoing many of the transformations.This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).
2. They can be dynamically reordered during processing.

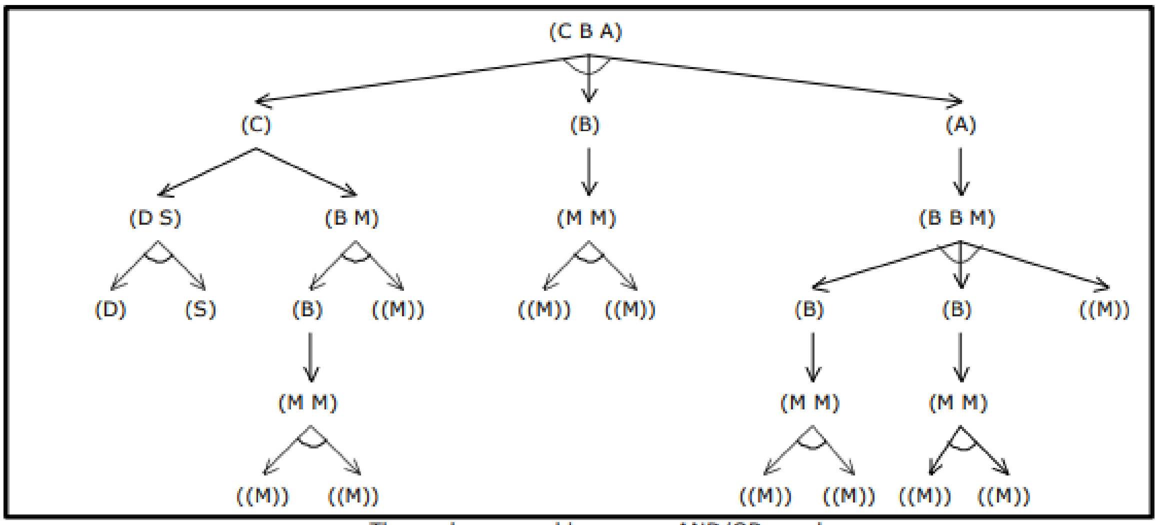
The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be:

Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap hisown bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

The previous exchange problem, when implemented as an and/or graph looks as follows:



Expansion for the exchange problem using OR connectors only

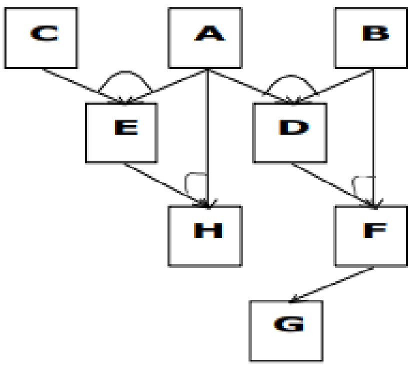


The exchange problem as an AND/OR graph

Example 1:

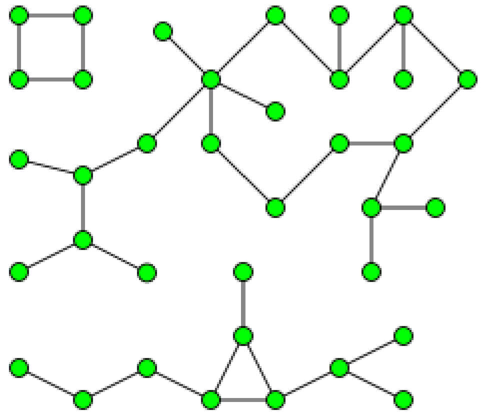
Draw an AND/OR graph for the following prepositions:

1. A
2. B
3. C
4. A ^ B -> D
5. A ^ C -> E
6. B ^ D -> F
7. F -> G
8. A ^ E -> H



Connected components

In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph. For example, the graph shown in the illustration has three connected components. A vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.



A graph with three connected components.

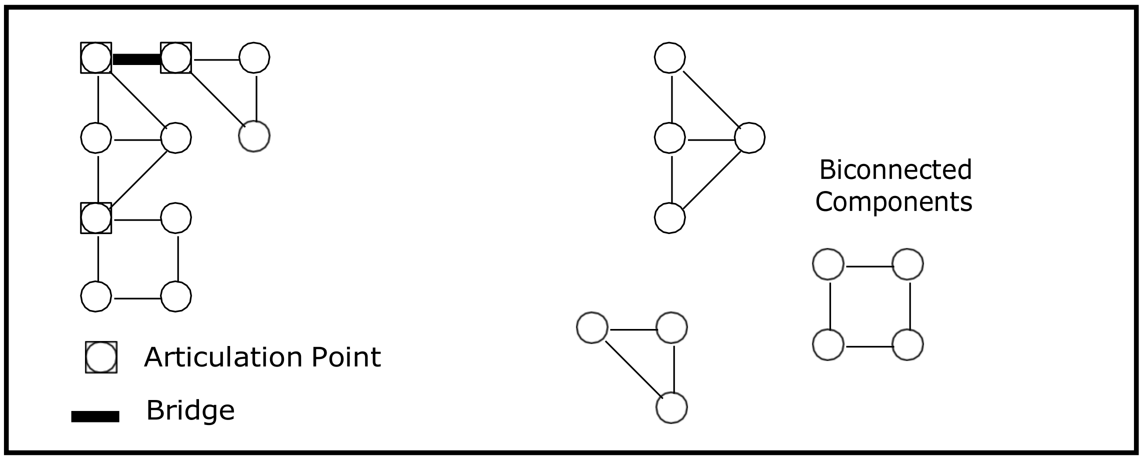
Biconnected Components:

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:



Articulation Points and Bridges

Biconnected graphs and articulation points are of great interest in thedesign of network algorithms,
because these are the "critical" points, whose failure will result in thenetwork 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, ... 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 vis an articulation point.

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

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) ≥ DFN (u)

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 wadjacent 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]);
        }
}
```

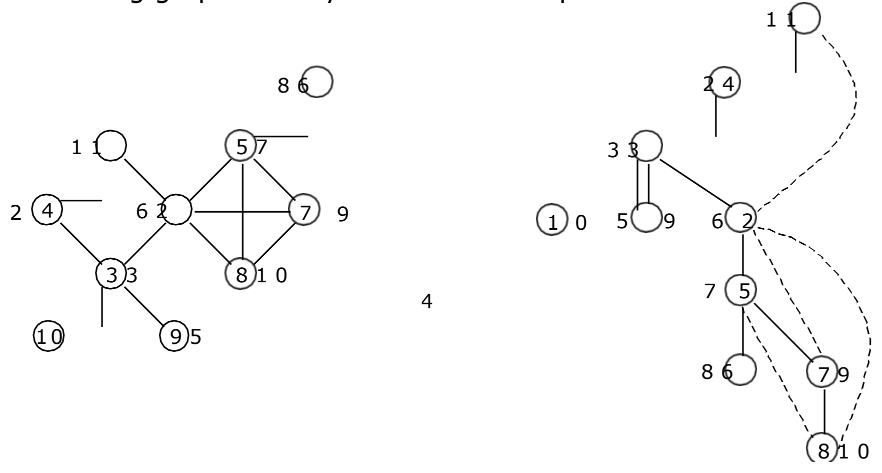
Algorithm for finding the Biconnected Components:

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]);
        }
}
```

Example:

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



4 Graph Dept h First Spanning Tree

To identify the articulation points, we use:

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w 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.

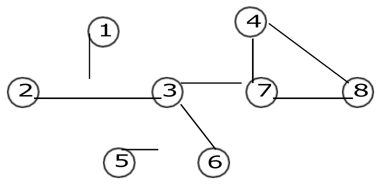
Vertex 7: is not an articulation point as child 8 has L (8) = 6, and DFN (7) = 9, So, the condition L (8) $\geq$ DFN (7) is false.

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

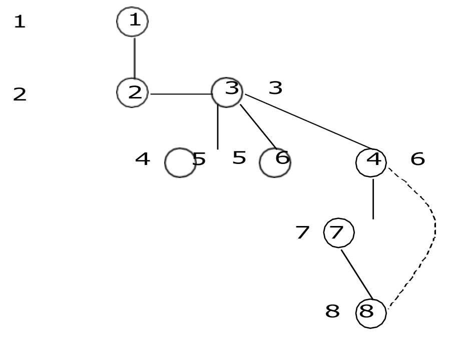
Therefore, the articulation points are {2, 3, 5}.

Example:

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



G ra p h



D F S s p a n ni n g T re e

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w 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 ofu. 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}.

GENERAL METHOD

Greedy method- General method, applications- Knapsack problem, Job sequencing with deadlines, Minimum cost spanning trees, Single source shortest path problem.

Greedy Method

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 := φ;              // 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:

$$\text{maximize } \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_i \quad x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

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 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 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 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 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 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 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 / x_i. Select the object with the maximum p_i / x_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 / x_i ratio, so x_3 = ½ 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 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

This solution is the optimal solution.

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 pi 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 ≤ r ≤ k are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ 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]] ≤ r, 1 ≤ r ≤ 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	

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

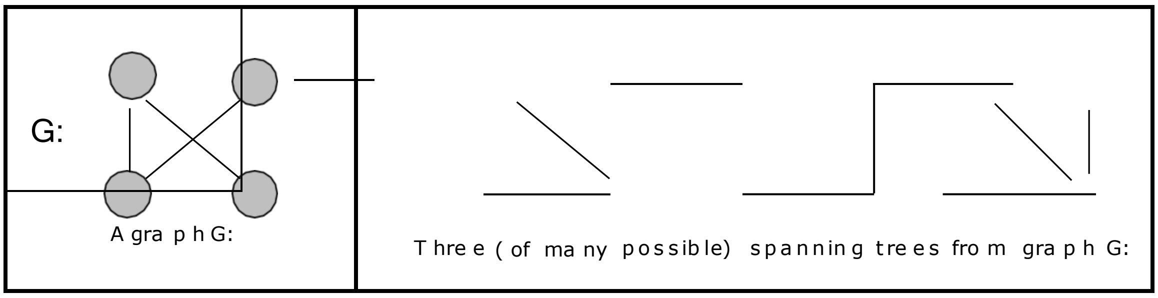
Algorithm GreedyJob (d, J, n)

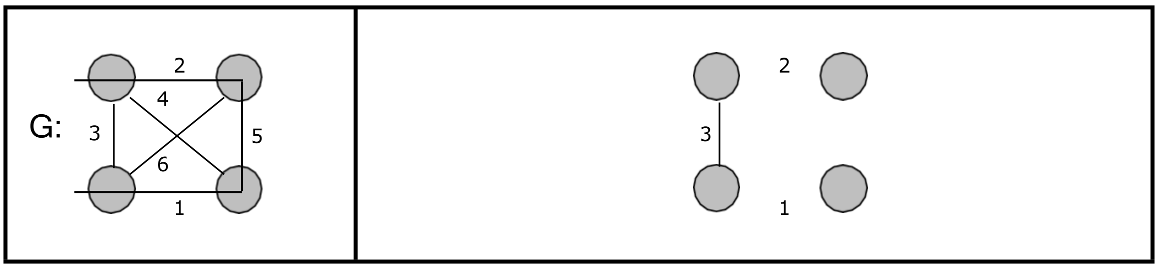
```
// J is a set of jobs that can be completed by their deadlines.
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

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.


G:
A gra p h G:
T hre e (of ma ny p o s sib le) s p a n n in g tre e s fro m gra p h G:



A w eig ht e d gra ph G: The min i ma l s p a n n in g tre e fro m w eig ht e d gra ph G:

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.

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;
}
```

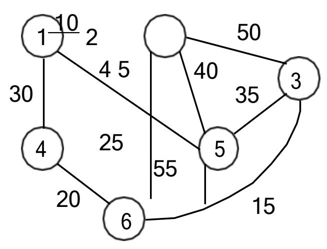
Running time:

- 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

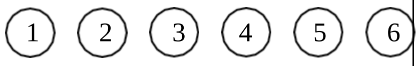
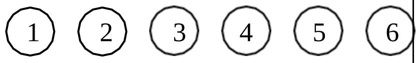
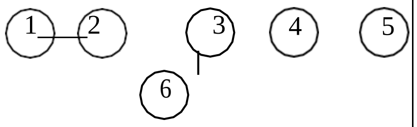
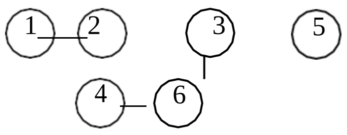
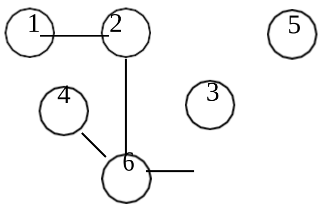
Example 1:

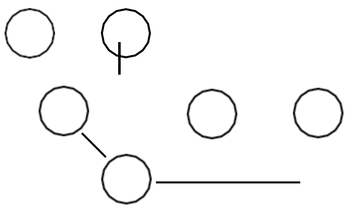


Arrange all the edges in the increasing order of their costs:

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 **Find**s 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 4 5_____3 6	{1, 2, 3, 4, 5, 6}	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

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 spanningtree.

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.

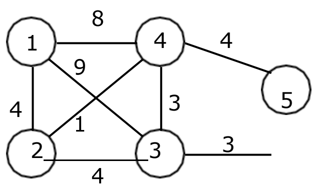
Prim's algorithm is an example of a greedy algorithm.

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

Example:

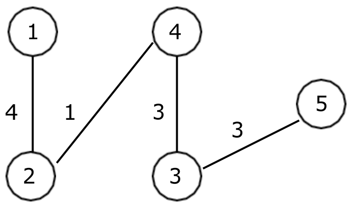
Considering the following graph, find the minimal spanning tree using prim’s algorithm.



The cost adjacent matrix is $\begin{pmatrix} \infty & 4 & 9 & 8 & \infty \\ 4 & \infty & 4 & 1 & \infty \\ 9 & 4 & \infty & 3 & 3 \\ 8 & 1 & 3 & \infty & 4 \\ \infty & \infty & 3 & 4 & \infty \end{pmatrix}$

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

<u>The steps as per the algorithm are as follows:</u>

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4
Min cost = cost (2, 4) = 1

T [1, 1] = 2

T [1, 2] = 4

for i = 1 to 5	Near matrix	Edges added to min spanning tree:
Begin i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2 i = 2 is cost (2, 4) < cost (2, 2) 1 < ∝, Yes So near [2] = 4 i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4 i = 4 is cost (4, 4) < cost (4, 2) ∝ < 1, no So near [4] = 2 i = 5 is cost (5, 4) < cost (5, 2) 4 < ∞, yes So near [5] = 4 end near [k] = near [l] = 0 near [2] = near[4] = 0	$\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & & & & \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$ $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & & & \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$ $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & & \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$ $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & 2 & \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$ $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 4 & 4 & 2 & 4 \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$ $\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 0 & 4 & 0 & 4 \\ \hline 1 & 2 & 3 & 4 & 5 \end{array}$	T [1, 1] = 2 T [1, 2] = 4

for i = 2 to n-1 (4) do
 	
<u>i = 2</u>
 	
for j = 1 to 5
j = 1
near(1)≠0 and cost(1, near(1))
2 ≠ 0 and cost (1, 2) = 4
 	
j = 2
near (2) = 0
 	
j = 3
is near (3) ≠ 0
4 ≠ 0 and cost (3, 4) = 3		

j = 4 near (4) = 0 J = 5 Is near (5) $\neq$ 0 4 $\neq$ 0 and cost (4, 5) = 4 select the min cost from the above obtained costs, which is 3 and corresponding J = 3 min cost = 1 + cost(3, 4) = 1 + 3 = 4 T (2, 1) = 3 T (2, 2) = 4		T (2, 1) = 3 T (2, 2) = 4
Near [j] = 0 i.e. near (3) =0	$\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 0 & 0 & 0 & 4 \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
for (k = 1 to n) K = 1 is near (1) $\neq$ 0, yes 2 $\neq$ 0 and cost (1,2) > cost(1, 3) 4 > 9, No K = 2 Is near (2) $\neq$0, No K = 3 Is near (3) $\neq$ 0, No K = 4 Is near (4) $\neq$ 0, No		
K = 5 Is near (5) $\neq$ 0 4 $\neq$ 0, yes and is cost (5, 4) > cost (5, 3) 4 > 3, yes than near (5) = 3	$\begin{array}{\|c\|c\|c\|c\|c\|}\hline 2 & 0 & 0 & 0 & 3 \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
i = 3 for (j = 1 to 5) J = 1 is near (1) $\neq$0 2 $\neq$ 0 cost (1, 2) = 4 J = 2 Is near (2) $\neq$0, No		

J = 3 Is near (3) $\neq$ 0, no Near (3) = 0 J = 4 Is near (4) $\neq$ 0, no Near (4) = 0 J = 5 Is near (5) $\neq$ 0 Near (5) = 3 ➔ 3 $\neq$ 0, yes And cost (5, 3) = 3 Choosing the min cost from the above obtaining costs which is 3 and corresponding J = 5 Min cost = 4 + cost (5, 3) = 4 + 3 = 7 T (3, 1) = 5 T (3, 2) = 3 Near (J) = 0 ➔ near (5) = 0 <u>for (k=1 to 5)</u> k = 1 is near (1) $\neq$ 0, yes and cost(1,2) > cost(1,5) 4 > $\propto$, No K = 2 Is near (2) $\neq$ 0 no K = 3 Is near (3) $\neq$ 0 no K = 4 Is near (4) $\neq$ 0 no K = 5 Is near (5) $\neq$ 0 no **<u>i = 4</u>** <u>for J = 1 to 5</u> J = 1 Is near (1) $\neq$ 0 2 $\neq$ 0, yes cost (1, 2) = 4 j = 2 is near (2) $\neq$ 0, No	2 | 0 | 0 | 0 | 0 1 2 3 4 5	T (3, 1) = 5 T (3, 2) = 3

J = 3 Is near (3) $\neq$ 0, No Near (3) = 0 J = 4 Is near (4) $\neq$ 0, No Near (4) = 0 J = 5 Is near (5) $\neq$ 0, No Near (5) = 0 Choosing min cost from the above it is only '4' and corresponding J = 1 Min cost = 7 + cost (1,2) = 7+4 = 11	0 | 0 | 0 | 0 | 0	
T (4, 1) = 1 T (4, 2) = 2	1 2 3 4 5	T (4, 1) = 1 T (4, 2) = 2
Near (J) = 0 ➔ Near (1) = 0 <u>for (k = 1 to 5)</u> K = 1 Is near (1) $\neq$ 0, No K = 2 Is near (2) $\neq$ 0, No K = 3 Is near (3) $\neq$ 0, No K = 4 Is near (4) $\neq$ 0, No K = 5 Is near (5) $\neq$ 0, No End.		

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

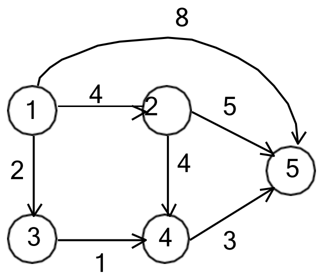
In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

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.

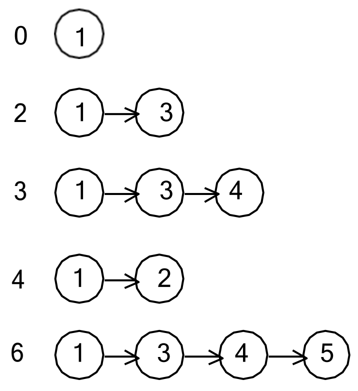
Dijkstra’s algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra’s algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis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

UNIT III:
Dynamic Programming: General method, applications-Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

Dynamic Programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that canbe 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 formulationsof 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 theoptimal solution.
- Perform a trace back step in which the solution itself isconstructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamicprogramming 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 toform 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 seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

All pairs shortest paths:

In the all pairs shortest path problem, we are to find a shortest path between everypair 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 thesame 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’salgorithm 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 ≠ 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:

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

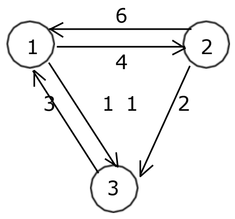
Algorithm All Paths (Cost, A, n)

```
// 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 ≤ i ≤ n.
{
        for i := 1 to n do
                for j:= 1 to n do
                        A [i, j] := cost  [i, j];                    // copy cost
        into A.for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                                A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x 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 nocycles with zero or negative cost.



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

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

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

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

A^1 (1, 1) = min {(A^o (1, 1) + A^o (1, 1)), c (1, 1)} = min {0 + 0, 0} = 0

A^1 (1, 2) = min {(A^o (1, 1) + A^o (1, 2)), c (1, 2)} = min {(0 + 4), 4} = 4

A^1 (1, 3) = min {(A^o (1, 1) + A^o (1, 3)), c (1, 3)} = min {(0 + 11), 11} = 11

A^1 (2, 1) = min {(A^o (2, 1) + A^o (1, 1)), c (2, 1)} = min {(6 + 0), 6} = 6

A^1 (2, 2) = min {(A^o (2, 1) + A^o (1, 2)), c (2, 2)} = min {(6 + 4), 0)} = 0

A^1 (2, 3) = min {(A^o (2, 1) + A^o (1, 3)), c (2, 3)} = min {(6 + 11), 2} = 2

A^1 (3, 1) = min {(A^o (3, 1) + A^o (1, 1)), c (3, 1)} = min {(3 + 0), 3} = 3

A^1 (3, 2) = min {(A^o (3, 1) + A^o (1, 2)), c (3, 2)} = min {(3 + 4), $\propto$} = 7

A^1 (3, 3) = min {(A^o (3, 1) + A^o (1, 3)), c (3, 3)} = min {(3 + 11), 0} = 0

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

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

$A^2(1, 1) = \min\{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min\{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min\{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min\{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min\{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min\{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min\{(A(2, 2) + A(2, 1), c(2, 1)\} = \min\{(0 + 6), 6\} = 6$

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

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

$A^2(3, 1) = \min\{(A(3, 2) + A(2, 1), c(3, 1)\} = \min\{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min\{(A(3, 2) + A(2, 2), c(3, 2)\} = \min\{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min\{(A(3, 2) + A(2, 3), c(3, 3)\} = \min\{(7 + 2), 0\} = 0$

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

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

$A^3(1, 1) = \min\{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min\{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min\{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min\{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min\{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min\{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min\{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min\{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min\{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min\{(2 + 7), 0\} = 0$

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

$A^3(3, 1) = \min\{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min\{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min\{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min\{(0 + 7), 7\} = 7$

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

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

TRAVELLING SALESPERSON PROBLEM:

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 $< i, j> \notin E$. 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:

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

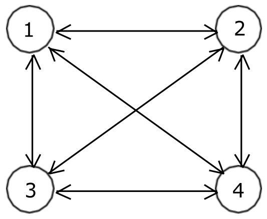
Generalizing equation 1, we obtain (for $i \notin S$)

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

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

Example :

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:

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

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \qquad - \qquad (2)$$

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:

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

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

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

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

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.

0/1 – KNAPSACK:

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 $x_1, x_2, \ldots, x_n$. 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 $x_n, x_{n-1},$ 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}, , x_1 must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, x_n, , x_1 will not be optimal. Hence, the principal of optimality holds.

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

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \qquad -- \qquad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_o(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then $f_1, f_2, \ldots f_n$ 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 $\Theta(m\ n)$ time to compute f_n. When the wi'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 $\Theta(m\ n)$ 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, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; 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:

$$S^i_1 = \{(P, W) \mid (P - p_i, W - w_i) \varepsilon S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i to$_1$gether. 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, $f_o(x) = 0$, for all x and $f_i(x) = - \propto$ if $x < 0$.

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

$F_3(6) = \max (f_2(6), f_2(6 – 4) + 5\} = \max \{f_2(6), f_2(2) + 5\}$

$F_2(6) = \max (f_1(6), f_1(6 – 3) + 2\} = \max \{f_1(6), f_1(3) + 2\}$

$F_1(6) = \max (f_0(6), f_0(6 – 2) + 1\} = \max \{0, 0 + 1\} = 1$

$F_1(3) = \max (f_0(3), f_0(3 – 2) + 1\} = \max \{0, 0 + 1\} = 1$

Therefore, $F_2(6) = \max (1, 1 + 2\} = 3$

$F_2(2) = \max (f_1(2), f_1(2 – 3) + 2\} = \max \{f_1(2), - \propto + 2\}$

$F_1(2) = \max (f_0(2), f_0(2 – 2) + 1\} = \max \{0, 0 + 1\} = 1$

$F_2(2) = \max \{1, - \propto + 2\} = 1$

Finally, $f_3(6) = \max \{3, 1 + 5\} = 6$

Other Solution:

For the given data we have:

$S^0 = \{(0, 0)\}$; $S^0 = \{(1_1, 2)\}$

$S^1 = (S^0 \cup S^0{}_1) = \{(0, 0), (1, 2)\}$

```
X - 2 = 0  => x  = 2.        y – 3 = 0  => y = 3
X - 2 = 1  => x  = 3.        y – 3 = 2  => y = 5
```

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

$S^2 = (S^1 \cup S^1{}_1) = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$

```
X – 5 = 0  => x  = 5.        y – 4 = 0  => y = 4
X – 5 = 1  => x  = 6.        y – 4 = 2  => y = 6
X – 5 = 2  => x  = 7.        y – 4 = 3  => y = 7
X – 5 = 3  => x  = 8.        y – 4 = 5  => y = 9
```

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

$S^3 = (S^2 \cup S^2{}_1) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$

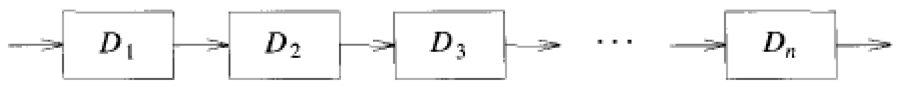
By applying Dominance rule,

$S^3 = (S^2 \cup S^2{}_1) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)\}$

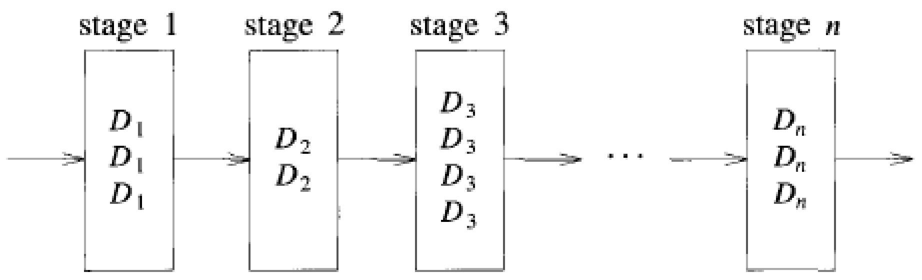
From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

Reliability Design:

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 $\Pi\ r_i$. 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 $r_i = 0.99$, $i \le i \le 10$, then $\Pi\ r_i = .904$. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.



n devices D_i, $1 \le i \le n$, connected in series



Multiple devices connected in parallel in each stage

If stage i contains mi 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:

Maximize $\underset{1 \le i \le n}{\pi} \phi_i(m_i)$

Subject to $\sum_{1 \le i \le n} C_i\ m_i < C$

$m_i \ge 1$ and interger, $1 \le i \le n$

Assume each $C_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \ge 1$

An optimal solution $m_1, m_2 \ldots\ldots\ldots m_n$ is the result of a sequence of decisions, one decision for each m_i.

Let $f_i(x)$ represent the maximum value of $\underset{1 \le j \le i}{\pi} \phi(m_J)$

Subject to the constrains:

$$\sum_{1 \le J \le i} C_J m_J \le x \quad \text{and } 1 \le m_j \le u_J,\ 1 \le j \le i$$

Example :

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 if stage I has mi devices of type i in parallel, then $\phi_i\ (m_i) = 1 - (1-r_i)^{mi}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

We use $S_j^i \rightarrow i\text{:}\mathit{stage\ number\ and\ } J\text{: } \mathit{no.\ of\ devices\ in\ stage\ } i = m_i$

$S^o = \{f_o(x), x\}$ *initially* $f_o(x) = 1$ *and* $x = 0$, *so*, $S^o = \{1, 0\}$

Compute S^1, S^2 and S^3 as follows:

S^1 = depends on u_1 value, as $u_1 = 2$, so

$$S^1 = \{S^1_1, S^1_2\}$$

S^2 = depends on u_2 value, as $u_2 = 3$, so

$S^2 = \left\{S_1^2, S_2^2, S_3^2\right\}$

S^3 = depends on u_3 value, as $u_3 = 3$, so

$S^3 = \left\{S_1^3, S_2^3, S_3^3\right\}$

Now find, $S^1_1 = \left\{\left(f_1(x), x\right)\right\}$

$f_1(x) = \{\phi_1(1) f_o(\,), \phi_1(2) f_0()\}$ With devices $m_1 = 1$ and $m_2 = 2$

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_i(mi)) = 1 - (1 - r_i)^{mi}$

$\phi_1(1) = 1 - (1 - r_1)^{m\,1} = 1 - (1 - 0.9)^1 = 0.9$

$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$

$S_1^1 = \{f_1(x), x\} = = (0.9, 30)$

$S_2^1 = \{0.99, 30 + 30\} = (0.99,$

$60)$ Therefore, $S^1 = \{(0.9, 30),$

$(0.99, 60)\}$

Next find $S_1^2 = \left\{\left(f_2(x), x\right)\right\}$

$f_2(x) = \{\phi_2(1) * f_1(\,), \phi_2(2) * f_1(\,), \phi_2(3) * f_1(\,)\}$

$\phi_2(1) = 1 - (1 - r_I)_{mi} = 1 - (1 - 0.8)^1 = 1 - 0.2 = 0.8$

$\phi_2(2) = 1 - (1 - 0.8)^2 = 0.96$

$\phi_2(3) = 1 - (1 - 0.8)^3 = 0.992$

$S_1^2 = \{(0.8(0.9), 30 + 15), (0.8(0.99), 60 + 15)\} = \{(0.72, 45), (0.792, 75)\}$

$S_2^2 = \{(0.96(0.9), 30 + 15 + 15), (0.96(0.99), 60 + 15 + 15)\}$
$= \{(0.864, 60), (0.9504, 90)\}$

$S_3^2 = \{(0.992(0.9), 30 + 15 + 15 + 15), (0.992(0.99), 60 + 15 + 15 + 15)\}$
$= \{(0.8928, 75), (0.98208, 105)\}$

$S^2 = \left\{S_1^2, S_2^2, S_3^2\right\}$

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 $f_1 \underline{>} f_2$ and $x_1 < x_2$ the discard

(f_2, x_2)Case 3: otherwise simply write (f_1, x_1)

$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$\phi_3\ (1) = 1 - (1 - r_I)^{mi} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$

$\phi_3\ (2) = 1 - (1 - 0.5)^2 = 0.75$

$\phi_3\ (3) = 1 - (1 - 0.5)^3 = 0.875$

$S_1^3 = \{(0.5\ (0.72), 45 + 20), (0.5\ (0.864), 60 + 20), (0.5\ (0.8928), 75 + 20)\}$

$S_1^3 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.75\ (0.72), 45 + 20 + 20), (0.75\ (0.864), 60 + 20 + 20),$
$(0.75\ (0.8928), 75 + 20 + 20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

$S_3^3 = \{\ (0.875\ (0.72), 45 + 20 + 20 + 20), (0.875\ (0.864), 60 + 20 + 20 + 20),$
$(0.875\ (0.8928), 75 + 20 + 20 + 20)\ \}$

$S_3^3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$

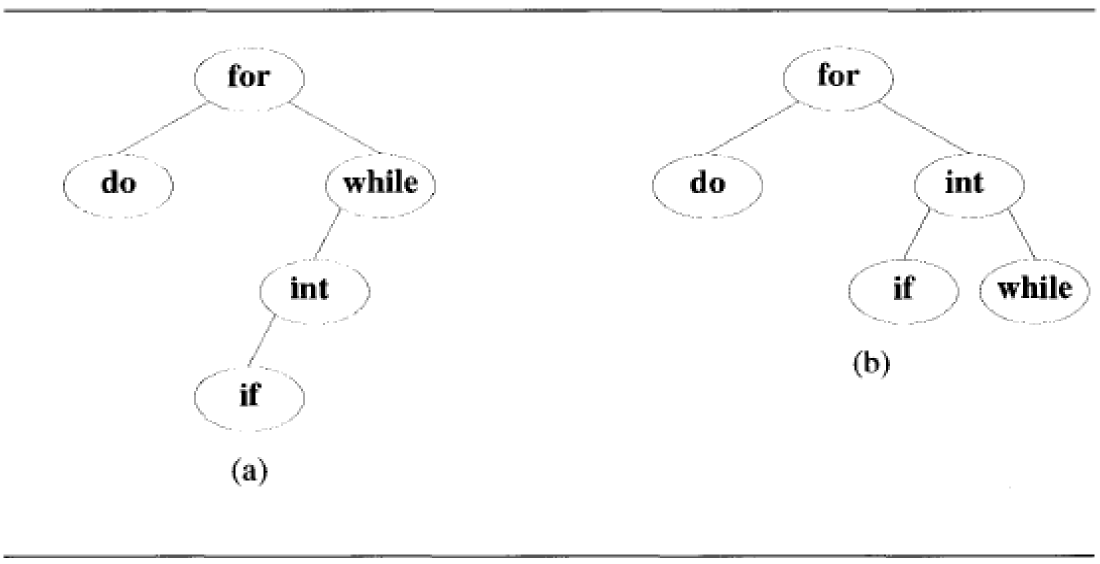
If cost exceeds 105, remove that tuples

$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$

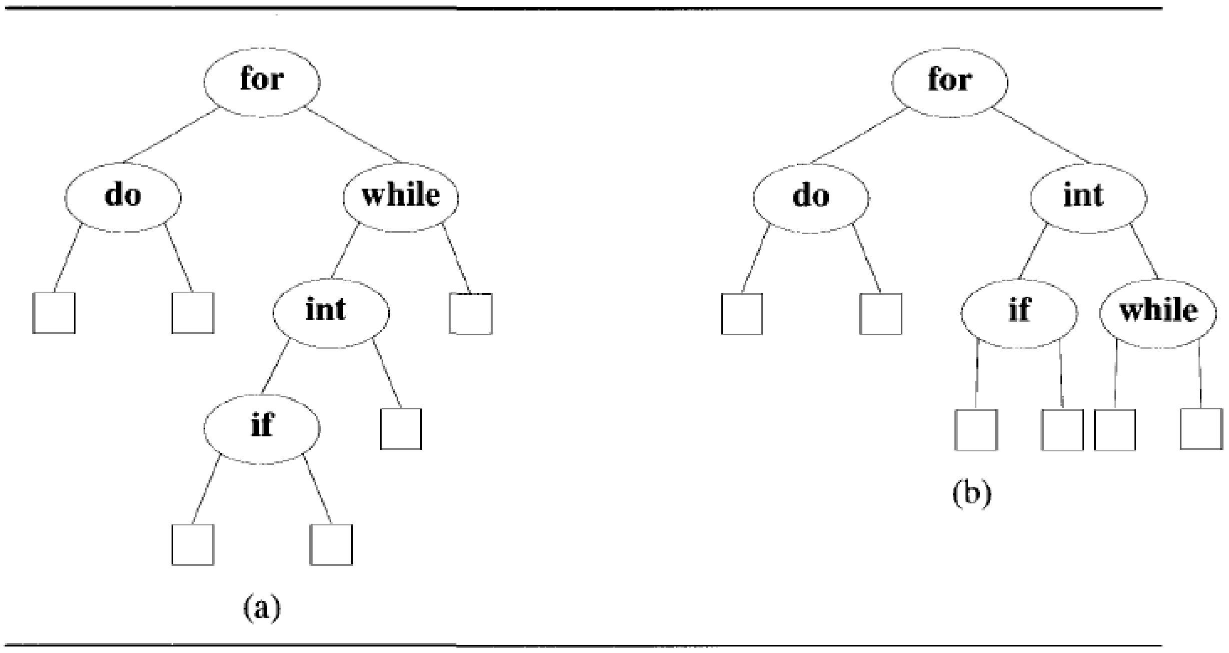
The best design has a reliability of 0.648 and a cost of 100. Tracing back forthesolution through S^i 's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Optimal Binary Search Tree:

In computer science, an optimal binary search tree (Optimal BST), sometimes called a weight-balanced binary tree,[1] is a binary search tree which provides the smallest possible search time (or expected searchtime) for a given sequence of accesses (or access probabilities).

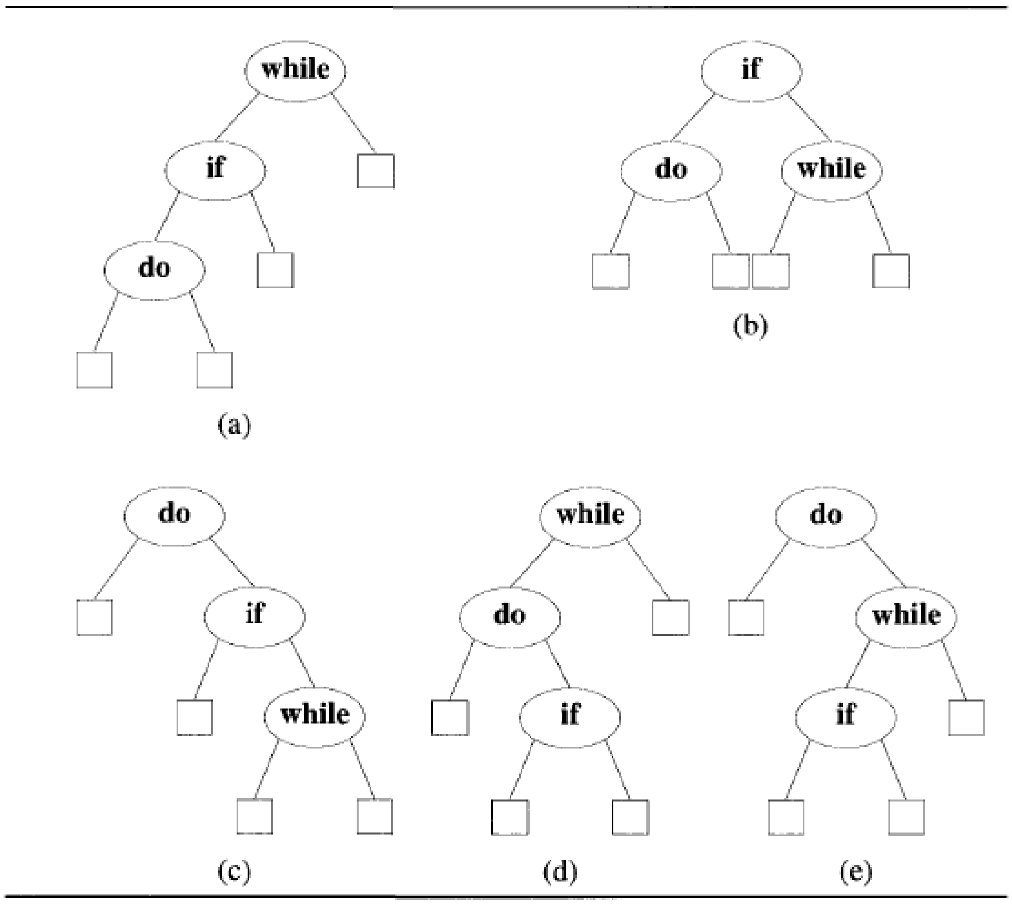


Two possible binary search trees



Binary search trees of Figure ⇧ with external nodes added

The no of external nodes are same in both trees.



Possible binary search trees for the identifier set {**do**, **if**, **while**}

The C (i, J) can be computed as:

$$C(i, J) = \min_{i<k\leq J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K,J)\}$$

$$= \min_{i<k\leq J} \{C(i, K-1) + C(K, J)\} + w(i, J) \qquad \text{--} \qquad (1)$$

Where $W(i, J) = P(J) + Q(J) + w(i, J-1)$ -- (2)

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 \leq i \leq n$.

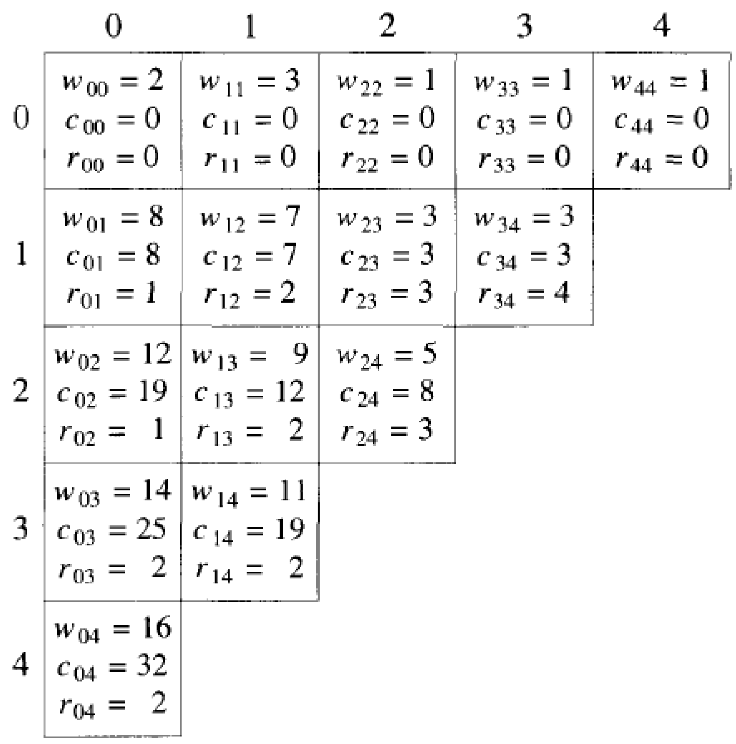
C (i, J) is the cost of the optimal binary search tree 'T_{ij}' during computation we record the root R (i, J) of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n),C (0, n) and R (0, n) are obtained.

Example Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ (**do, if, int, while**). Let $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i) = q(i), c(i,i) = 0$ and $r(i,i) = 0, 0 \le i \le 4$. the observation $w(i,j) =$ $p(j) + q(j) + w(i,j-1)$, we get

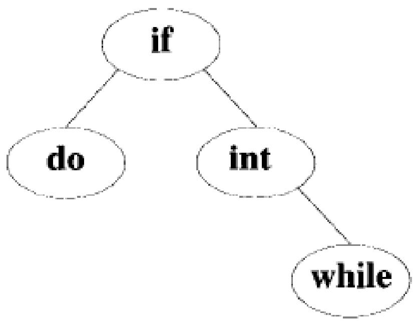
$$\begin{array}{rcl}
w(0,1) & = & p(1) + q(1) + w(0,0) = 8 \\
c(0,1) & = & w(0,1) + \min\{c(0,0) + c(1,1)\} \;=\; 8 \\
r(0,1) & = & 1 \\
w(1,2) & = & p(2) + q(2) + w(1,1) \;=\; 7 \\
c(1,2) & = & w(1,2) + \min\ \{c(1,1) + c(2,2)\} \;=\; 7 \\
r(0,2) & = & 2 \\
w(2,3) & = & p(3) + q(3) + w(2,2) \;=\; 3 \\
c(2,3) & = & w(2,3) + \min\ \{c(2,2) + c(3,3)\} \;=\; 3 \\
r(2,3) & = & 3 \\
w(3,4) & = & p(4) + q(4) + w(3,3) \;=\; 3 \\
c(3,4) & = & w(3,4) + \min\ \{c(3,3) + c(4,4)\} \;=\; 3 \\
r(3,4) & = & 4
\end{array}$$

Knowing $w(i,i+1)$ and $c(i,i+1)$, $0 \le i < 4$, we can again use Equation 5.12 to compute $w(i,i+2)$, $c(i,i+2)$, and $r(i,i+2)$, $0 \le i < 3$. This process can be repeated until $w(0,4)$, $c(0,4)$, and $r(0,4)$ are obtained.



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

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



Optimal search tree for Example

Matrix chain multiplication

The problem

Given a sequence of matrices $A_1, A_2, A_3, \ldots, A_n$, find the best way (using the minimal number of multiplications) to compute their product.

- Isn't there only one way? $((\cdots((A_1 \cdot A_2) \cdot A_3) \cdots) \cdot A_n)$
- No, matrix multiplication is *associative*.
  e.g. $A_1 \cdot (A_2 \cdot (A_3 \cdot (\cdots(A_{n-1} \cdot A_n) \cdots)))$ yields the same matrix.
- Different multiplication orders do not cost the same:
  - Multiplying $p \times q$ matrix A and $q \times r$ matrix B takes $p \cdot q \cdot r$ multiplications; result is a $p \times r$ matrix.
  - Consider multiplying 10×100 matrix A_1 with 100×5 matrix A_2 and 5×50 matrix A_3.
    - $(A_1 \cdot A_2) \cdot A_3$ takes $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ multiplications.
    - $A_1 \cdot (A_2 \cdot A_3)$ takes $100 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 100 = 75000$ multiplications.

Notation

- In general, let A_i be $p_{i-1} \times p_i$ matrix.
- Let $m(i, j)$ denote minimal number of multiplications needed to compute $A_i \cdot A_{i+1} \cdot \cdot \cdot A_j$
- We want to compute $m(1, n)$.

Recursive algorithm

- Assume that someone tells us the position of the **last** product, say k. Then we have to compute recursively the best way to multiply the chain from i to k, and from $k + 1$ to j, and add the cost of the final product. This means that

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$$

- If noone tells us k, then we have to try all possible values of k and pick the best solution.
- Recursive formulation of $m(i, j)$:

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j}\{m(i,k)+m(k+1,j)+p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

- To go from the recursive formulation above to a program is pretty straightforward:

```
Matrix-chain($i, j$)
    IF $i = j$ THEN return 0
    $m = \infty$
    FOR $k = i$ TO $j - 1$ DO
        q = Matrix-chain($i, k$) + Matrix-chain($k + 1, j$) $+p_{i-1} \cdot p_k \cdot p_j$
        IF $q < m$ THEN $m = q$
    OD
    Return $m$
END  Matrix-chain

Return Matrix-
```

- Running time:

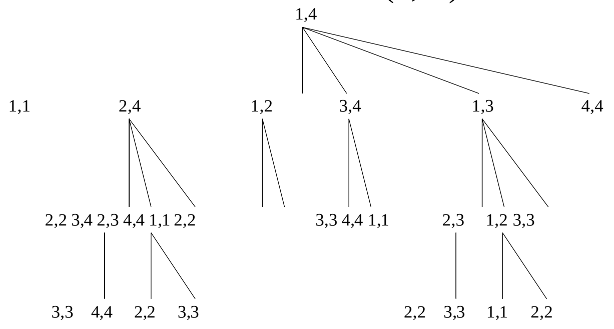
$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k) + O(1))$$

$$= 2 \cdot \sum_{k=1}^{n-1} T(k) + O(n)$$

$$\geq 2 \cdot T(n-1)$$
$$\geq 2 \cdot 2 \cdot T(n-2)$$
$$\geq 2 \cdot 2 \cdot 2 \ldots$$
$$= 2^n$$

- Exponential is ... SLOW!

- Problem is that we compute the same result over and over again.
  - Example: Recursion tree for Matrix-chain(1, 4)



For example, we compute Matrix-chain(3, 4) twice.

Dynamic programming with a table and recursion

- Solution is to "remember" the values we have already computed in a table. This is called *memoization*. We'll have a table T[1..n][1..n] such that T[i][j] stores the solution to problem Matrix-CHAIN(i,j). Initially all entries will be set to ∞.

```
FOR $i$ = 1 to $n$
      DO FOR $j = i$
      to $n$ DO
            $T[i][j] = \infty$
      OD
OD
```

- The code for MATRIX-CHAIN(i,j) stays the same, except that it now uses the table. The first thing MATRIX-CHAIN(i,j) does is to check the table to see if $T[i][j]$ is already computed. Is so, it returns it, otherwise, it computes it and writes it in the table. Below is the updated code.

```
Matrix-chain($i, j$)
      IF $T[i][j] < \infty$ THEN return $T$
      $[i][j]$ IF $i = j$ THEN $T[i][j] = 0$,
      return 0 $m = \infty$
      FOR $k = i$ to $j - 1$ DO
            q = Matrix-chain($i, k$) + Matrix-chain($k + 1, j$) $+ p_{i-1} \cdot p_k \cdot p_j$
            IF $q < m$ THEN $m = q$
      OD
      $T[i][j] = m$
      return $m$
END Matrix-chain

return Matrix-
```

- The table will prevent a subproblem MATRIX-CHAIN(i,j) to be computed more thanonce.
- Running time:
  - $\Theta(n^2)$ different calls to matrix-chain(i, j).
  - The first time a call is made it takes $O(n)$ time, *not* counting recursive calls.
  - When a call has been made once it costs $O(1)$ time to make it again.

  $O(n^3)$ time
  - Another way of thinking about it: $\Theta(n^2)$ total entries to fill, it takes $O(n)$ to fill one.

UNIT IV
Backtracking: General method, Applications- n-queue problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

BACKTRACKING

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 (x1, , x_n) where each x_i Є 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, , 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 Iof 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 whereby, 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.

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.

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.

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.

N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 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 $(x_1, \ldots, x_8)$, 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 45^0 diagonal if:

$i - j = k - l.$

This implies, $j - l = i - k$

- Diag 135 conflict:

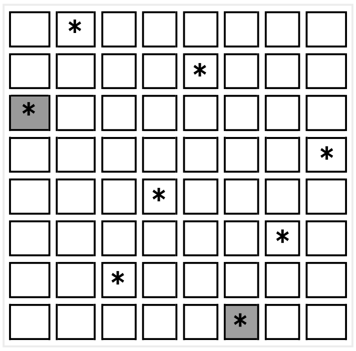
  $i + j = k + l$. This implies, $j - l = k - i$

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 ith queen and l be the column of object in row 'k' for the kth 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

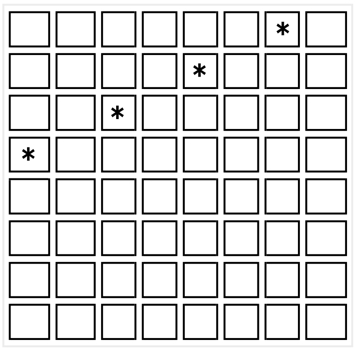
Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$
$\Rightarrow 5 = 5$

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 then 8, repeat Step 1.

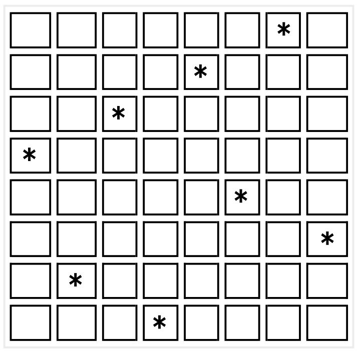
Step 3:

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.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Remarks**
7	5	3	1					
7	5	3	1*	2*				|j - l| = |1 – 2| = 1 |i – k | = |4 – 5| = 1
7	5	3	1	4				
7*	5	3	1	4	2*			|j - l| = |7 – 2| = 5 |i – k | = |1 – 6| = 5
7	5	3*	1	4	6*			|j - l| = |3 – 6| = 3 |i – k | = |3 – 6| = 3
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		|j - l| = |4 – 2| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4*	8	6*		|j - l| = |4 – 6| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			|j - l| = |1 – 2| = 1 |i – k | = |7 – 6| = 1
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	|j - l| = |3 – 8| = 5 |i – k| = |3 – 8| = 5
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

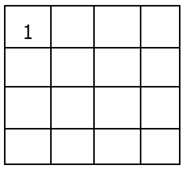
* indicates conflicting queens.

On a chessboard, the **solution** will look like:

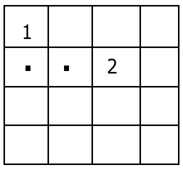


4 – Queens Problem:

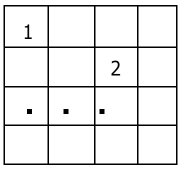
Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 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:



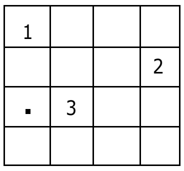
(a)



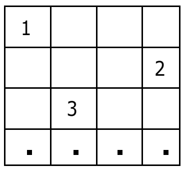
(b)



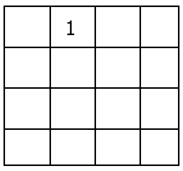
(c)



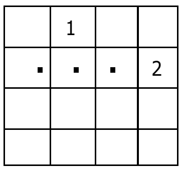
(d)



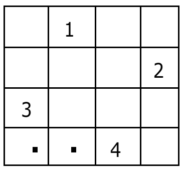
(e)



(f)



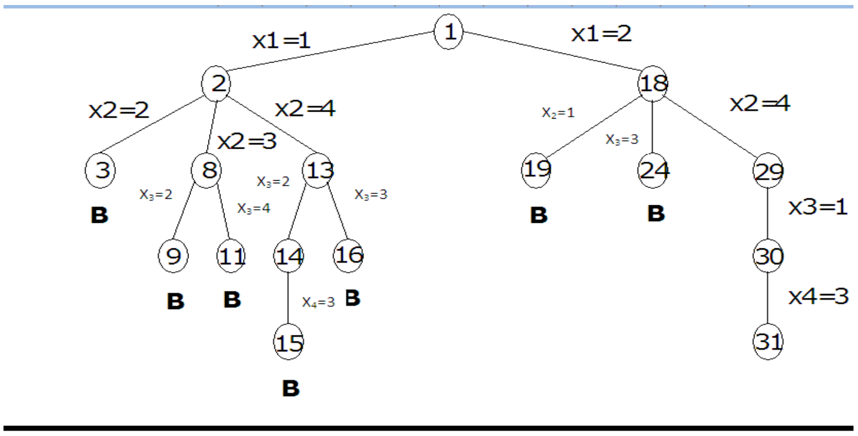
(g)



(h)

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

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

Algorithm 7.4 Can a new queen be placed?

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

Algorithm 7.5 All solutions to the n-queens problem

Complexity Analysis:

$$1 + n + n^2 + n^3 + \dots\dots\dots\dots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains:

$$\frac{8^{8+1}-1}{8-1} = 19,173,961 \text{ nodes}$$

Sum of Subsets:

Given positive numbers wi, 1 ≤ i ≤ n, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, 1 ≤ k ≤ n.

Explicit constraints:

- x_i **Є** {j | j is an integer and 1 ≤ j ≤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 ≤ 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).

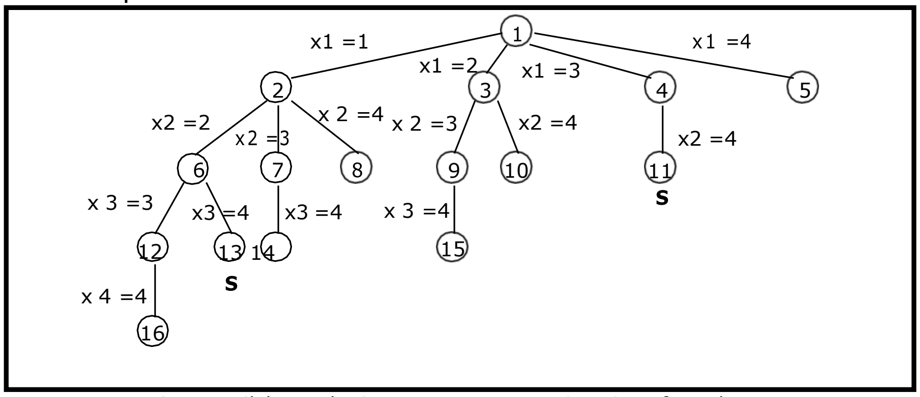
A better formulation of the problem is where the solution subset is represented by an n-tuple (x_1, , x_n) such that x_i **Є** {0, 1}.

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 distincttuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are(11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



A possible solution space organisation for the sum of the subsets problem.

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 mot 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.

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

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

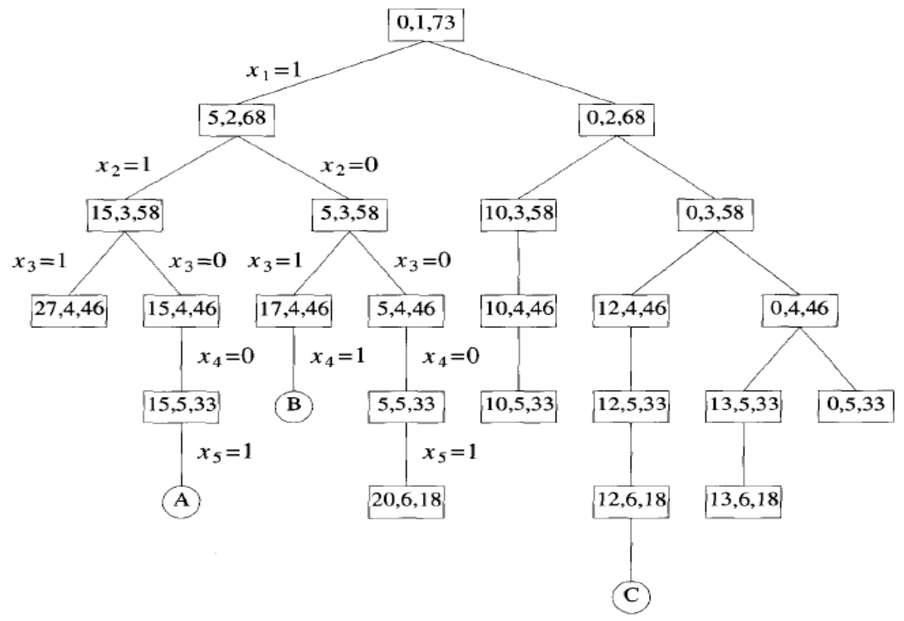


Figure 7.10 Portion of state space tree generated by SumOfSub

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-colorabiltiy 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, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., 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);

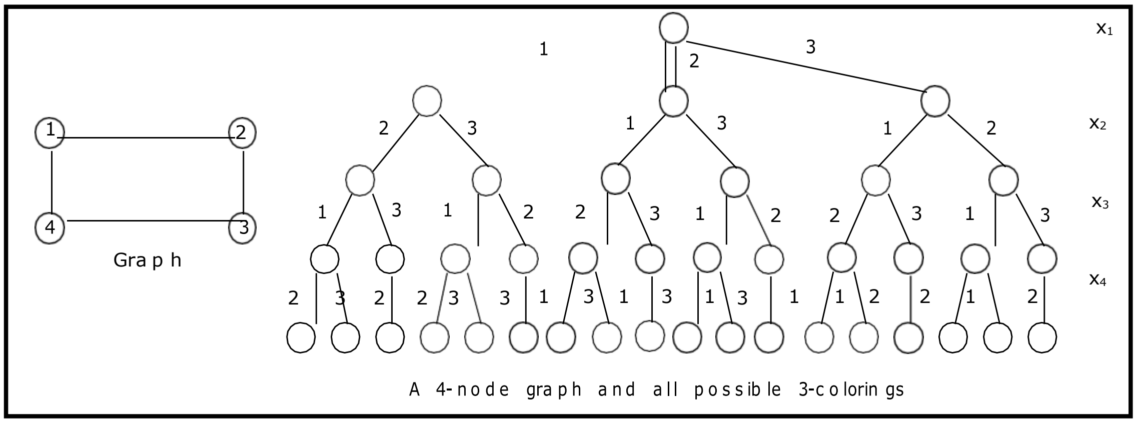
```
Algorithm mcoloring (k)
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, ........ , m to the vertices of the graph such that adjacent vertices areassigned
// distinct integers 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.
    }
```

Example:

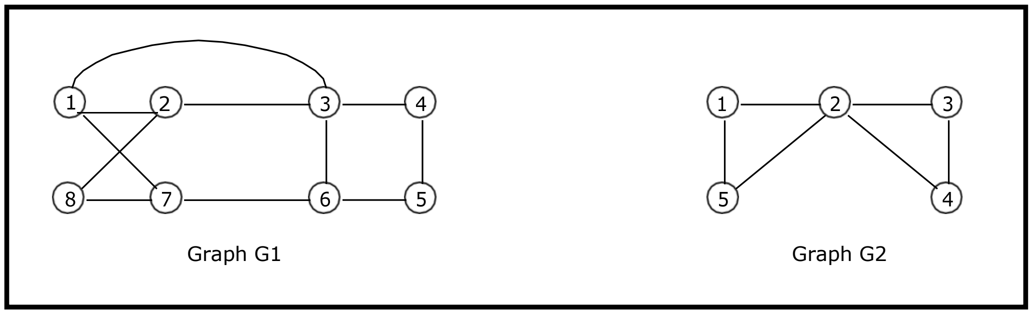
Color the graph given below with minimum number of colors by backtracking using state space tree



A 4-node graph and all possible 3-colorings

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 $v_1, v_2, \ldots\ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect 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 $(x_1, \ldots\ldots x_n)$ 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 $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-1}. The

UNIT V:

Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

NP-Hard and NP-Complete problems: Basic concepts, non deterministic algorithms, NP - Hard and NP Complete classes, Cook's theorem.

Branch and Bound

General method:

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-an-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 in 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 $\bar{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$\bar{c}(x) = f(h(x)) + \bar{g}(x)$

where, $\bar{c}(x)$ is the cost of x.

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

$\bar{g}(x)$ is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function $\bar{c}(x) = f(h(x) + \bar{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\bar{c}(.)$ is called a LC–search (Least Cost search)

BFS and D-search are special cases of LC-search. If $\bar{g}(x) = 0$ and f(h(x)) = 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 $\bar{g}(x) > \bar{g}(y)$ 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 $\bar{c}(x)$ of c(x).

Control Abstraction for LC-Search:

Let t be a state space tree and c() 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 $\bar{c}(.)$ is used to estimate c(). 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 $c(x) = \bar{c}(x)$.

LC-search uses $\bar{c}$ 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 c(). 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 $\hat{c}(.)$ 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 $c(x) \geq \hat{c}(x) >$ 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(.), 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(.).

- For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.
- For nodes representing infeasible solutions, $c(x) = \propto$.
- 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.

FIFO Branch and Bound:

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = $\propto$ 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 $\hat{c}(4)$ and $\hat{c}(5)$ 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 $\hat{c}(7) = 10 >$ 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 $\hat{c}(10) = 11 >$ upper, and this nodeis 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 $\hat{c}(x)$ > 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 $\hat{c}(x)$ > upper are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and u(.) is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper = ∝ 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 $\hat{c}(4)$ > upper and $\hat{c}(5)$ > 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 $\hat{c}(7) = 10$ > 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 answernode.

The path = $1 \rightarrow \overset{2}{3} \rightarrow \overset{3}{9} = 5 + 3 = 8$

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 <i, j>. c_{ij} = ∝ if <i, j> ∉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, π, 1 | π is a permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that (1, i_1, i_2, ... i_{n-1}, 1) $\in$ S iff <i_j, i_{j+1}> $\in$ E, $0 \le j \le n - 1$ 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 $\propto$ 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 <i, j> 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 $\propto$.

   b) Set A (j, 1) to $\propto$.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only $\propto$. Let r is the total amount subtracted to reduce the matrix.

   c) Find $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $\hat{c}(R)$ indicates the lower bound of the ith node in (i, j) path and $\hat{c}(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

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

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix $\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$

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

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) = $\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$

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

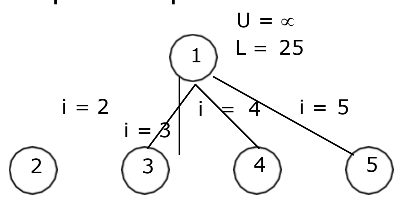
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced costmatrix.

The lower bound for node is 25 and upper bound is $\propto$.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1,5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

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

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

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∝ and also set A(3, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 3) + r$
$\hat{c}(S)$ = 25 + 17 + 11 = 53

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∝ and also set A(4, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

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

$\hat{c}(S) = 25 + 0 + 0 = 25$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∝ and also set A(5, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

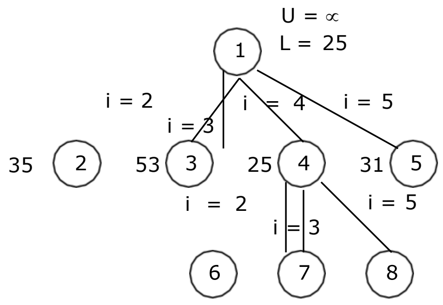
Then the resultant matrix is $$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

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

$\hat{c}(S) = 25 + 1 + 5 = 31$

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

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

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to $\propto$ and also set A(2, 1) to $\propto$.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

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

$\hat{c}(S)$ = 25 + 3 + 0 = 28

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to $\propto$ and also set A(3, 1) to $\propto$.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

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

$\hat{c}(S)$ = 25 + 12 + 13 = 50

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to $\propto$ and also set A(5, 1) to $\propto$.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely $\propto$.

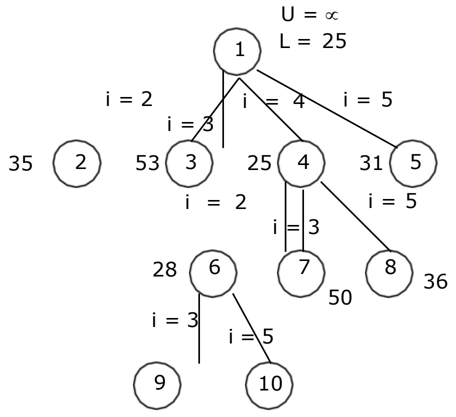
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

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

$\hat{c}(S)$ = 25 + 0 + 11 = 36

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

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

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∝ and also set A(3, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

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

$\hat{c}\,(S\,)$ = 28 + 11 + 13 = 52

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∝ and also set A(5, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

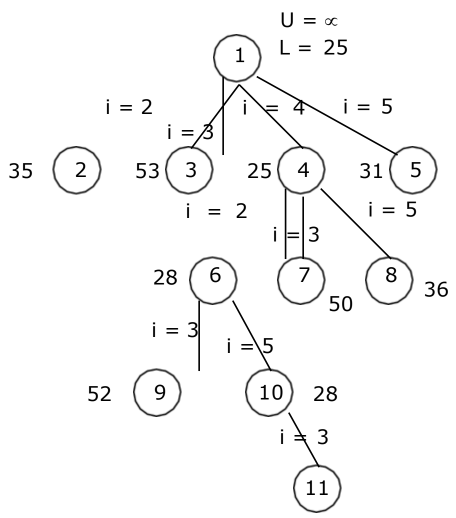
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

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

$\hat{c}\,(S\,)$ = 28 + 0 + 0 = 28

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

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

The new possible paths is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∝ and also set A(3, 1) to ∝. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

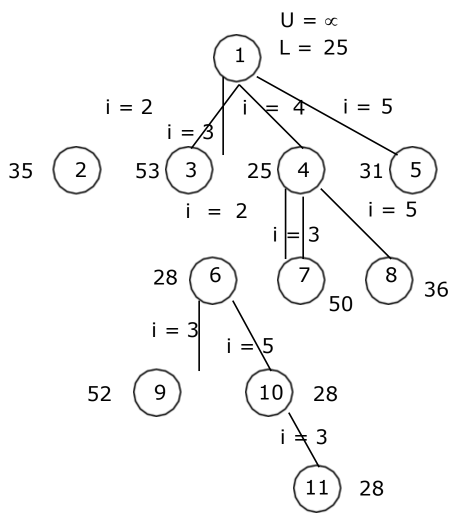
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

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

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The overall tree organization is as follows:



The path of traveling sale person problem is:

$1 \longrightarrow 4 \longrightarrow 2 \longrightarrow 5 \longrightarrow 3 \longrightarrow 1$

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

0/1 Knapsack Problem

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.

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

So, Upper bound = 32

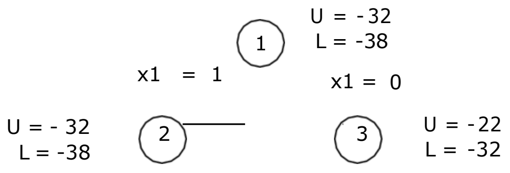
To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound $= 10 + 10 + 12 + (\frac{3}{9} X\ 18) = 32 + 6 = 38$

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.

Therefore, Upper bound (U) = -32
Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, x_1= 1, means we should place first item in the knapsack.

U = 10 + 10 + 12 = 32, make it as -32

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, \text{ make it as } -38$$

For node 3, x_1 = 0, means we should not place first item in the knapsack.

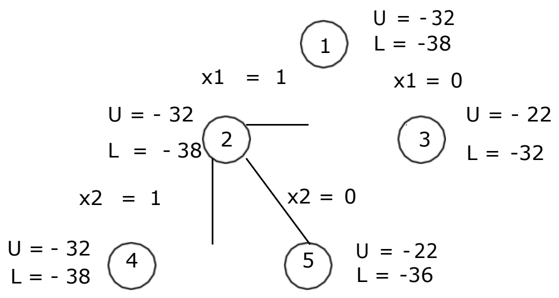
U = 10 + 12 = 22, make it as -22

$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, \text{ make it as } -32$$

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

For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

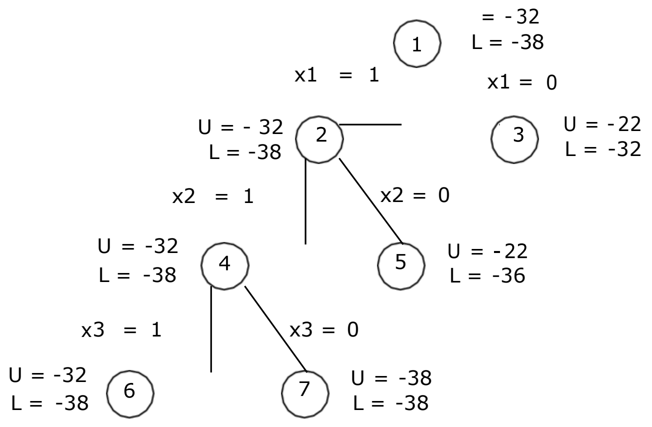
Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14

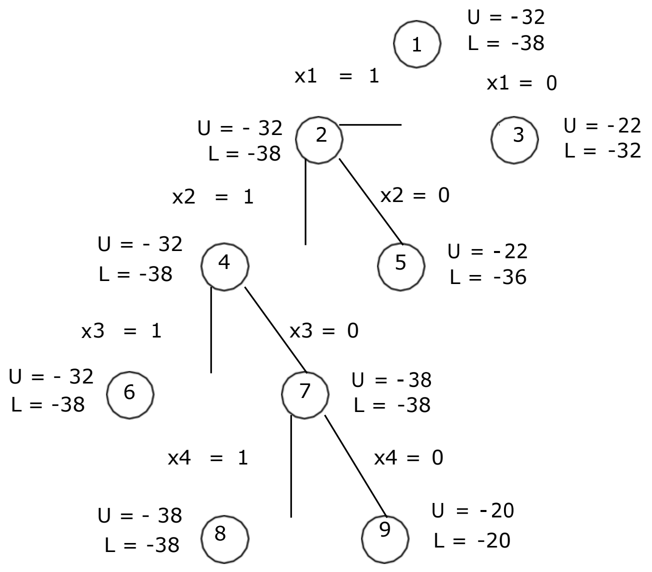
Choose node 4, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6
For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 = 0
For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from $1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8$

$X_1 = 1$

$X_2 = 1$

$X_3 = 0$

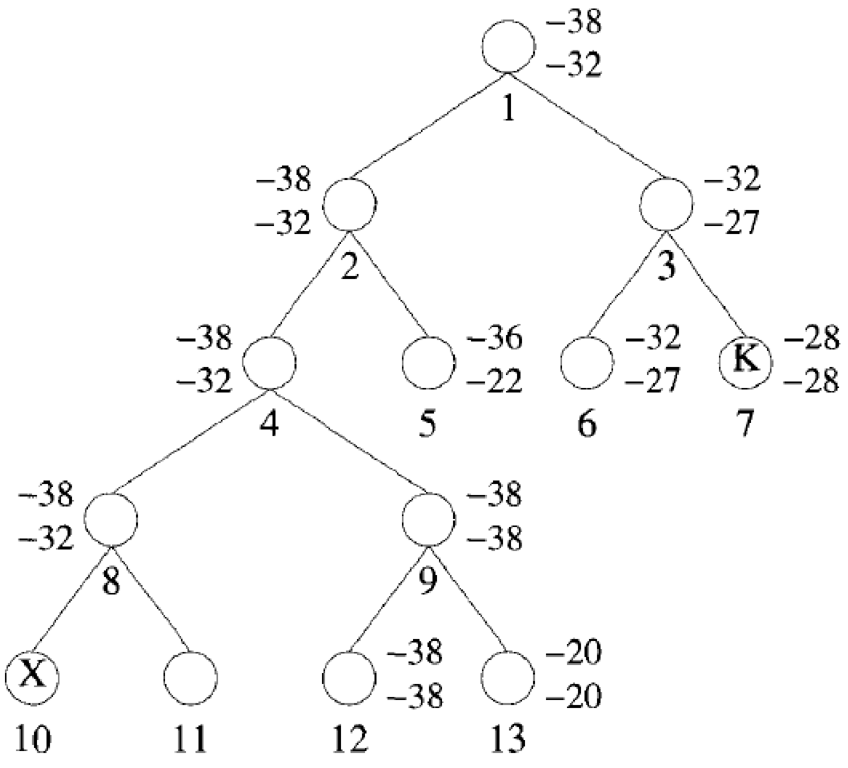
$X_4 = 1$

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

Maximum profit is:

$$\sum P_i\, x_i = 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1$$
$$= 10 + 10 + 18 = 38.$$

Portion of state space tree using FIFO Branch and Bound for above problem: As follows:



upper number = $\hat{c}$
lower number = u

NP-Hard and NP-Complete problems

Deterministic and non-deterministic algorithms

Deterministic: The algorithm in which every operation is uniquely defined is called deterministic algorithm.

Non-Deterministic: The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such an algorithm is called non-deterministic algorithm.

The non-deterministic algorithms use the following functions:

1. Choice: Arbitrarily chooses one of the element from given set.
2. Failure: Indicates an unsuccessful completion
3. Success: Indicates a successful completion

A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. Whenever, there is a set of choices that leads to a successful completion, then one such set of choices is selected and the algorithm terminates successfully.

In case the successful completion is not possible, then the complexity is O(1). In case of successful signal completion then the time required is the minimum number of steps needed to reach a successful completion of O(n) where n is the number of inputs.

The problems that are solved in polynomial time are called tractable problems and the problems that require super polynomial time are called non-tractable problems. All deterministic polynomial time algorithms are tractable and the non-deterministic polynomials are intractable.

```
Algorithm NSort($A$, $n$)
// Sort $n$ positive integers.
{
    for $i := 1$ to $n$ do $B[i] := 0$; // Initialize $B[\,]$.
    for $i := 1$ to $n$ do
    {
        $j :=$ Choice$(1, n)$;
        if $B[j] \neq 0$ then Failure();
        $B[j] := A[i]$;
    }
    for $i := 1$ to $n - 1$ do  // Verify order.
        if $B[i] > B[i + 1]$ then Failure();
    write ($B[1 : n]$);
    Success();
}
```

Nondeterministic sorting

```
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();
}
```

Nondeterministic knapsack algorithm

Satisfiability Problem:
The satisfiability is a boolean formula that can be constructed using the following literals and operations.

1. A literal is either a variable or its negation of the variable.
2. The literals are connected with operators $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$
3. Parenthesis

The satisfiability problem is to determine whether a Boolean formula is true for some assignment of truth values to the variables. In general, formulas are expressed in Conjunctive Normal Form (CNF).

A Boolean formula is in conjunctive normal form iff it is represented by
$(x_i \vee x_j \vee x_k^1) \wedge (x_i \vee x^1_j \vee x_k)$

A Boolean formula is in 3CNF if each clause has exactly 3 distinct literals.

Example:

The non-deterministic algorithm that terminates successfully iff a given formula E(x1,x2,x3) is satisfiable.

```
Algorithm Eval(E, n)
// Determine whether the propositional formula E is
// satisfiable. The variables are x_1, x_2, ..., x_n.
{
    for i := 1 to n do // Choose a truth value assignment.
        x_i := Choice(false, true);
    if E(x_1, ..., x_n) then Success();
    else Failure();
}
```

Nondeterministic satisfiability

Reducability:
A problem Q1 can be reduced to Q2 if any instance of Q1 can be easily rephrased as an instance of Q2. If the solution to the problem Q2 provides a solution to the problem Q1, then these are said to be reducable problems.

Let L1 and L2 are the two problems. L1 is reduced to L2 iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time and is denoted by L1α L2.
If we have a polynomial time algorithm for L2 then we can solve L1 in polynomial time.
Two problems L1 and L2 are said to be polynomially equivalent iff L1α L2 and L2 α L1.

Example: Let P1 be the problem of selection and P2 be the problem of sorting. Let the input have n numbers. If the numbers are sorted in array A[ ] the i^{th} smallest element of the input can be obtained as A[i]. Thus P1 reduces to P2 in O(1) time.

Decision Problem:
Any problem for which the answer is either yes or no is called decision problem. The algorithm for decision problem is called decision algorithm.
Example: Max clique problem, sum of subsets problem.

Optimization Problem: Any problem that involves the identification of an optimal value (maximum or minimum) is called optimization problem.
Example: Knapsack problem, travelling salesperson problem.
In decision problem, the output statement is implicit and no explicit statements are permitted.
The output from a decision problem is uniquely defined by the input parameters and algorithm specification.

Many optimization problems can be reduced by decision problems with the property that a decision problem can be solved in polynomial time iff the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time then the optimization problem cannot be solved in polynomial time.

Class *P*:

P: the class of decision problems that are solvable in O(*p*(*n*)) time, where *p*(*n*) is a polynomial of problem's input size *n*

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing

Class *NP*

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true? This problem is in *NP*.

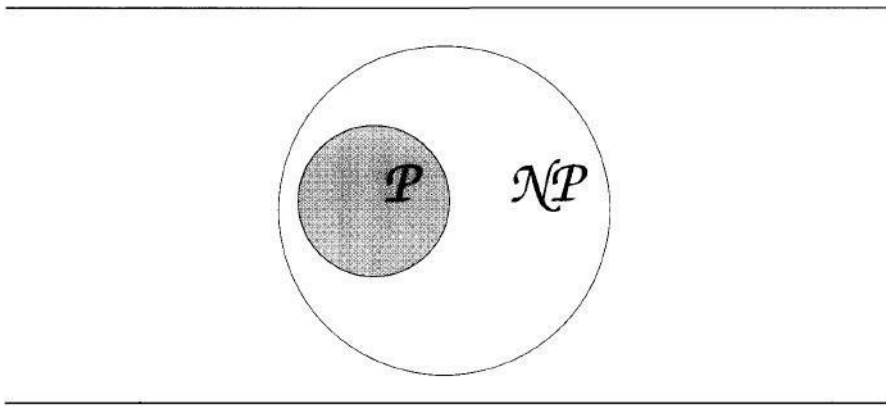
Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

What problems are in *NP*?

- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of *n* integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

  $P \subseteq NP$
- Big question: $P = NP$?



Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

NP HARD AND NP COMPLETE

Polynomial Time algorithms

Problems whose solutions times are bounded by polynomials of small degree are called polynomial time algorithms

Example: Linear search, quick sort, all pairs shortest path etc.

Non- Polynomial time algorithms

Problems whose solutions times are bounded by non-polynomials are called non-polynomial time algorithms

Examples: Travelling salesman problem, 0/1 knapsack problem etc

It is impossible to develop the algorithms whose time complexity is polynomial for non-polynomial time problems, because the computing times of non-polynomial are greater than polynomial. A problem that can be solved in polynomial time in one model can also be solved in polynomial time.

NP-Hard and NP-Complete Problem:

Let P denote the set of all decision problems solvable by deterministic algorithm in polynomial time. NP denotes set of decision problems solvable by nondeterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of nondeterministic algorithms, P ⊆ NP. The nondeterministic polynomial time problems can be classified into two classes. They are

1. NP Hard and
2. NP Complete

NP-Hard: A problem L is NP-Hard iff satisfiability reduces to L i.e., any nondeterministic polynomial time problem is satisfiable and reducable then the problem is said to be NP-Hard.

Example: Halting Problem, Flow shop scheduling problem

NP-Complete: A problem L is NP-Complete iff L is NP-Hard and L belongs to NP (nondeterministic polynomial).

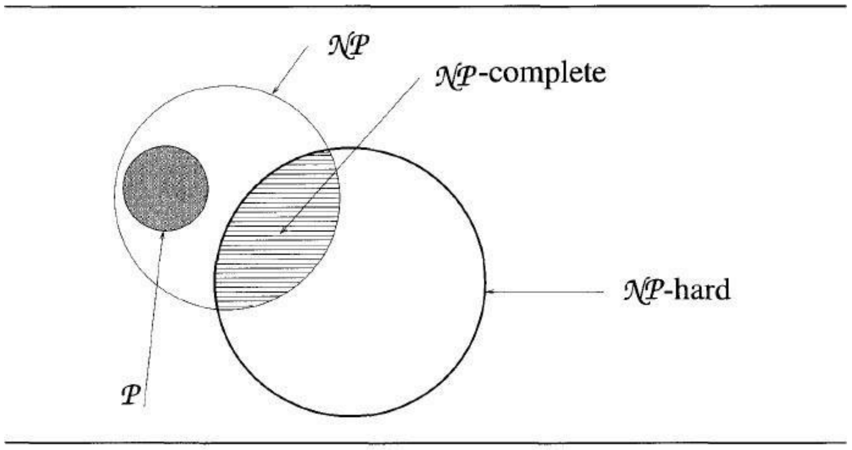
A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time. (NP=P)

If an 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.
Normally the decision problems are NP-complete but the optimization problems are NP-Hard.

However if problem L1 is a decision problem and L2 is an optimization problem, then it is possible that L1α L2.
Example: Knapsack decision problem can be reduced to knapsack optimization problem.
There are some NP-hard problems that are not NP-Complete.

Relationship between P,NP,NP-hard, NP-Complete

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-complete respectively. Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:



Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems

Problem conversion

A decision problem D1 can be converted into a decision problem D2 if there is an algorithm which takes as input an arbitrary instance I1 of D1 and delivers as output an instance I2 of D2such that I2 is a positive instance of D2 if and only if I1 is a positive instance of D1. If D1 can be converted into D2, and we have an algorithm which solves D2, then we thereby have an algorithm which solves D1. To solve an instance I of D1, we first use the conversion algorithm to generate an instance I0 of D2, and then use the algorithm for solving D2 to determine whether or not I0 is a positive instance of D2. If it is, then we know that I is a positive instance of D1, and if it is not, then we know that I is a negative instance of D1. Either way, we have solved D1 for that instance. Moreover, in this case, we can say that the computational complexity of D1 is at most the sum of the computational complexities of D2 and the conversion algorithm. If the conversion algorithm has polynomial complexity, we say that D1 is at most polynomially harder than D2. It means that the amount of computational work we have to do to solve D1, over and above whatever is required to solve D2, is polynomial in the size of the problem instance. In such a case the conversion algorithm provides us with a feasible way of solving D1, given that we know how to solve D2.

Given a problem X, prove it is in NP-Complete.

1. Prove X is in NP.
2. Select problem Y that is known to be in NP-Complete.
3. Define a polynomial time reduction from Y to X.
4. Prove that given an instance of Y, Y has a solution iff X has a solution.

Cook’s theorem:

Cook’s Theorem implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. This would have huge practical repercussions, since many frequently encountered problems which are so far believed to be intractable are NP. This special property of SAT is called NP-completeness. A decision problem is NP-complete if it has the property that any NP problem can be converted into it in polynomial time. SAT was the first NP-complete problem to be recognized as such (the theory of NP-completeness having come into existence with the proof of Cook’s Theorem), but it is by no means the only one. There are now literally thousands of problems, cropping up in many different areas of computing, which have been proved to be NP- complete.

In order to prove that an NP problem is NP-complete, all that is needed is to show that SAT can be converted into it in polynomial time. The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial.

Suppose SAT can be converted to problem D in polynomial time. Now take any NP problem D0. We know we can convert it into SAT in polynomial time, and we know we can convert SAT into D in polynomial time. The result of these two conversions is a polynomial-time conversion of D0 into D. since D0 was an arbitrary NP problem, it follows that D is NP-complete