Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- DataBase Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

- Compiled Books: https://www.tutorialsduniya.comcompiled-books
- Programs: https://www.tutorialsduniya.comprograms
- Question Papers: https://www.tutorialsduniya.comquestion-papers
- Python Notes: https://www.tutorialsduniya.compython
- Java Notes: https://www.tutorialsduniya.comjava
- JavaScript Notes: https://www.tutorialsduniya.comjavascript
- JSP Notes: https://www.tutorialsduniya.comjsp
- Microprocessor Notes: https://www.tutorialsduniya.commicroprocessor
- OR Notes: https://www.tutorialsduniya.comoperational-research
1. Introduction:

Algorithm: The word algorithm came from the name of a Persian mathematician Abu Jafar Mohammed Ibn Musa Al Khowarizmi (ninth century). An algorithm is simply a set of rules used to perform some calculations either by hand or more usually on a machine (computer).

Definition: An algorithm is a finite set of instructions that accomplishes a particular task. Another definition is a sequence of unambiguous instructions for solving a problem i.e., for obtaining a required output for any legitimate (genuine) input in a finite amount of time.

In addition, all algorithms must satisfy the following criteria (characteristics).

1. Input: zero or more quantities are externally supplied as input.

Consider Fibonacci numbers program, here, the aim of the problem is to display ten Fibonacci numbers. No input is required; in the problem itself, this is clearly mentioned as ten Fibonacci values. So zero items required for input.

Another problem is displaying given numbers of evens, so user should accept how many evens required. Based on the user input, the number of evens is to be displayed. So, one data item is required as input.

2. Output: At least one quantity is produced by given algorithm as output.

In the case of Fibonacci numbers program after executing the program, first ten Fibonacci values displayed as output.

In second case, based on user input, it should display given number of evens. An input of negative number is incorrect, should display proper error message as output. So this program displays at least one output as error message, or number if outputs that show given number of steps.

3. Definiteness: Each instruction is clear and unambiguous i.e., each step must be easy to understand and convey only a single meaning.

4. Effectiveness: Each instruction must be very basic, so that it can be carried out by a person using only pencil and paper.

This step is common in both Fibonacci and primes. For example, if user enters a negative numbers as input in evens, if you have a step like

Step: If N < 0 then
    Go to ERROR

A wrong instruction given as go to ERROR, those kinds of instructions should not be there in an algorithm.

5. Finiteness: If we can trace out the instructions of an algorithm then for all cases, the algorithm terminates after a finite number of steps.

Either in the case of Fibonacci or even numbers problem should be solved in some number of steps. For example, continuous display of Fibonacci series without termination leads to abnormal termination.

Criteria for Algorithms

<table>
<thead>
<tr>
<th>Input: Zero or more inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: At least one output.</td>
</tr>
<tr>
<td>Finiteness: N number of steps.</td>
</tr>
<tr>
<td>Definiteness: Clear algorithm step.</td>
</tr>
<tr>
<td>Effectiveness: A carried out step.</td>
</tr>
</tbody>
</table>
2. **Process for Design and analysis of algorithms:**

![Algorithm Design Technique](image)

1. **Understand the problem:** This is very crucial phase. If we did any mistake in this step the entire algorithm becomes wrong. So before designing an algorithm is to understand the problem first.

2. **Solution as an algorithm** (*Exact vs approximation solving*): Solve the problem exactly if possible. Even though some problems are solvable by exact method, but they are not faster when compared to approximation method. So in that situation we will use approximation method.

3. **Algorithm techniques:** In this we will use different design techniques like,
   
   *i)* Divide-and-conquer
   *ii)* Greedy method
   *iii)* Dynamic programming
   *iv)* Backtracking
   *v)* Branch and bound… etc.,

4. **Prove correctness:** once algorithm has been specified, next we have to prove its correctness. Usually testing is used for proving correctness.

5. **Analyze an algorithm:** Analyzing an algorithm means studying the algorithm behavior i.e., calculating the time complexity and space complexity. If the time complexity of algorithm is more then we will use one more designing technique such that time complexity should be minimum.

6. **Coding an algorithm:** after completion of all phases successfully then we will code an algorithm. Coding should not depend on any program language. We will use general notation (pseudo-code) and English language statement. Ultimately algorithms are implemented as computer programs.

3. **Types of Algorithms:**
   
   There are four types of algorithms
   
   1. Approximate algorithm.
   2. Probabilistic algorithm.
   3. Infinite algorithm.
1. **Approximate Algorithm**: An algorithm is said to approximate if it is infinite and repeating.

   \[ \sqrt{2} = 1.414 \]
   \[ \sqrt{3} = 1.713 \]
   \[ \pi = 3.14 \text{ etc...} \]

2. **Probabilistic algorithm**: If the solution of a problem is uncertain then it is called as probabilistic algorithm. **Ex**: Tossing of a coin.

3. **Infinite Algorithm**: An algorithm which is not finite is called as infinite algorithm. **Ex**: A complete solution of a chessboard, division by zero.

4. **Heuristic algorithm**: Giving fewer inputs getting more outputs is called the Heuristic algorithms. **Ex**: All Business Applications.

4. **Criteria’s (or) Issues for algorithm**:

   There are various issues in the study of algorithms;
   1. **How to devise algorithms**: The creation of an algorithm is a logical activity which may never be fully automated.
   2. **How to express algorithms**: We shall express all of our algorithms using the best principles of structuring.
   3. **How to validate algorithms**: After creation of algorithms is to validate algorithms. The process of checking an algorithm computes the correct answer for all possible legal inputs is called algorithm validation. The purpose of validation of algorithm is to find whether algorithm works properly without being dependent upon programming languages.
   4. **How to analyze algorithms**: Analysis of algorithm is a task of determining how much computing time and storage is required by an algorithm. Analysis of algorithms is also called performance analysis. The behavior of algorithm in best case, worst case and average case needs to be obtained.
   5. **How to test a program**: Testing a program really consists of two phases:
      i). **Debugging**: While debugging a program, it is checked whether program produces faulty results for valid set of input and if it is found then the program has to be corrected.
      ii). **Profiling or performance measuring**: Profiling is a process of measuring time and space required by a corrected program for valid set of inputs.

5 **Specification of algorithm**:

   There are various ways by which we can specify an algorithm.

   ![Diagram of Algorithm Specification](source)

   Using natural language
   Pseudocode
   Flow chart
   Program (Using programming language)

   It is very easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear, and may require brief description.

   Example: Write an algorithm to perform addition of two numbers.
   
   Step 1: Read the first number, say ‘a’.
   Step 2: Read the second number, say ‘b’.
   Step 3: Add the two numbers and store the result in a variable ‘c’.
   Step 4: Display the result.
Such a specification creates difficulty, while actually implementing it (difficulty in converting into source code). Hence many programmers prefer to have specification of algorithm by means of pseudo-code.

Another way of representing the algorithm is by flow chart. Flow chart is a graphical representation of an algorithm, but flowchart method work well only if the algorithm is small and simple.

**Pseudo-Code for expressing Algorithms**

Based on algorithm there are two more representations used by programmer and those are flow chart and pseudo-code. Flowchart is a graphical representation of an algorithm. Similarly pseudo-code is a representation of algorithm in which instruction sequence can be given with the help of programming constructs. It is not a programming language since no pseudo language compiler exists.

The general procedure for writing the pseudo-code is presented below-

1. Comments begin with // and continue until the end of line
2. A block of statements (compound statement) are represented using { and } for example if statement, while loop, functions etc.,

   **Example**
   
   ```
   {
   Statement 1;
   Statement 2;
   ........
   ........
   }
   ```

3. The delimiters [;] are used at the end of the each statement.
4. An identifier begins with a letter. *Example*: sum, sum5, a; but not in 5sum, 4a etc.,
5. Assignment of values to the variables is done using the assignment operators as := or .
6. There are two Boolean values TRUE and FALSE.
   Logical operators: AND, OR, NOT.
   Relational operators: <, >, ≥, ≤, =, ≠.
   Arithmetic operators: +, -, *, /, %;
7. The conditional statement if-then or if-then-else is written in the following form.
   If (condition) then (statement)
   If (condition) then (statement-1) else (statement-2)
   ‘If’ is a powerful statement used to make decisions based as a condition. If a condition is true the particular block of statements are execute.

   **Example**
   ```
   if(a>b) then
   {
   write("a is big");
   }
   else
   {
   write("b is big");
   }
   ```
8. Case statement

```
case
{
    :(condition -1): (statement-1)
    :(condition -2): (statement-2)
    :(condition -n): (statement-n)
    ............
    ............
    else     :(statement n+1);
}
```

If condition -1 is true, statement -1 executed and the case statement is exited. If statement -1 is false, condition -2 is evaluated. If condition -2 is true, statement-2 executed and so on. If none of the conditions are true, statement -(n+1) is executed and the case statement is exited. The else clause is optional.

9. Loop statements:

**For loop:**

i). The general form of the for loop is

```
for variable:=value 1 to value n step do
{
    Statement -1;
    Statement -1;
    ........
    ........
    Statement -n;
}
```

Example:

```
for i:=1 to 10 do
{
    write(i); //displaying numbers from 1 to 10
    i:=i+1;
}
```

ii). While loop:

The general form of the while loop is

```
while <condition> do
{
    <statement 1>
    <statement 2>
    ........
    ........
    <statement n>
}
```

Example:

```
i:=1;
while(i<=10)do
{
    write (i);//displaying numbers from 1 to 10
    i:=1+1;
}
```

Note that the statements of while loop are executed as long as <condition> is true.

iii). Repeat-until loop:

The general form of repeat-until is-

```
repeat
{
    <statement 1>
    <statement 2>
    ......
    ......
    <statement n>
}
until <condition>
```

Example:

```
i:=1;
repeat
{
    write (i);
    i:=1+1;
}
until (i>10);
```

Note that the statements are executed as long as <condition> is false.
10. Break: this statement is exit from the loop.
11. Elements of array are accessed using [ ].

For example, if A is an one-dimensional array, then \( i^{th} \) element can be accessed using \( A[i] \). If A is two-dimensional array, then \( (i, j)^{th} \) element can be accessed using \( A[i,j] \).

12. Procedures (functions): There is only one type of procedure:

An algorithm consists of a heading and a body.

\[
\begin{align*}
\text{procedure } & \quad \text{name of the procedure} \\
\text{Algorithm Name (} & \quad \text{<parameter list>}) \\
\{ & \quad \text{body of the procedure} \\
\}
\end{align*}
\]

13. Compound data-types can be formed with records

\[
\text{Syntax: } \\
\text{Name} = \text{record} \\
\{ & \quad \text{data-type -1 data 1;} \\
\text{data-type -2 data 2;} & \quad \text{data-type -n data n;} \\
\}
\]

**Example**

\[
\text{Employee} = \text{record} \\
\{ & \quad \text{int no;} \\
\text{char name[10];} & \quad \text{float salary;} \\
\}
\]

**Example 1:** Write an algorithm to find the sum of \( n \) numbers.

**Algorithm** `sum(n)`

```
{ 
   total:=0;
   for i:=1 to n do 
   total:= total + i;
   i:=i+1;
}
```

**Example 2:** Write an algorithm to perform Multiplication of two matrices.

**Algorithm** `Multiplication (A, B, n)`

```
{ 
   for i:=1 to n do 
   for j:=1 to n do 
   for k:=1 to n do 
   C[i,j]:=C[i,j]+A[i,k]*B[k,j];
}
```

6. **Performance Analysis:**

Performance analysis or analysis of algorithms refers to the task of determining the efficiency of an algorithm i.e how much computing time and storage an algorithm requires to run (or execute). This analysis of algorithm helps in judging the value of one algorithm over another.

To judge an algorithm, particularly two things are taken into consideration

1. Space complexity
2. Time complexity.

**Space Complexity:** The space complexity of an algorithm (program) is the amount of memory it needs to run to completion. The space needed by an algorithm has the following components.
INTRODUCTION TO ALGORITHMS

UNIT-1

1. Instruction Space.
2. Data Space.
3. Environment Stack Space.

**Instruction Space:** Instruction space is the space needed to store the compiled version of the program instructions. The amount of instruction space that is needed depends on factors such as:

i). The compiler used to compile the program into machine code.

ii). The compiler options in effect at the time of compilation.

iii). The target computer, i.e computer on which the algorithm run.

Note that, one compiler may produce less code as compared to another compiler, when the same program is compiled by these two.

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

i). Space needed by constants, for example 0, 1, 2.134.

ii). Space needed by dynamically allocated objects such as arrays, structures, classes.

**Environmental Stack Space:** Environmental stack space is used during execution of functions. Each time function is involved the following data are saved as the environmental stack.

i). The return address.

ii). Value of local variables.

iii). Value of formal parameters in the function being invoked.

Environmental stack space is mainly used in recursive functions. Thus, the space requirement of any program may therefore be written as

**Space complexity S(P) = C + Sp (Instance characteristics).**

This equation shows that the total space needed by a program is divided into two parts.

- Fixed space requirements(C) is independent of instance characteristics of the inputs and outputs.
  - Instruction space
  - Space for simple variables, fixed-size structure variables, constants.

- A variable space requirements (SP(1)) dependent on instance characteristics 1.
  - This part includes dynamically allocated space and the recursion stack space.

Example of instance character is:

**Examples: 1**

Algorithm NEC (float x, float y, float z)
{
    Return (X + Y + Y * Z + (X + Y + Z)) / (X + Y) + 4.0;
}

In the above algorithm, there are no instance characteristics and the space needed by X, Y, Z is independent of instance characteristics, therefore we can write,

S(XYZ) = 3+0 = 3

One space each for X, Y and Z

:. Space complexity is O(1).

**Examples: 2**

Algorithm ADD (float [], int n)
{
    sum = 0.0;
    for i=1 to n do
        sum=sum+X[i];
    return sum;
}
Here, at least \( n \) words since \( X \) must be large enough to hold the \( n \) elements to be summed. Here the problem instances is characterized by \( n \), the number of elements to be summed. So, we can write,

\[
S(ADD) = 3 + n
\]

3-one each for \( n \), \( I \) and sum

Where \( n \) is for array \( X[] \),

\[
\therefore \text{Space complexity is } O(n).
\]

**Time Complexity**

The time complexity of an algorithm is the amount of compile time it needs to run to completion. We can measure time complexity of an algorithm in two approaches

1. Priori analysis or *compile time*
2. Posteriori analysis or *run (execution) time*.

**In priori analysis** before the algorithm is executed we will analyze the behavior of the algorithm. A priori analysis concentrates on determining the order if execution of statements.

**In Posteriori analysis** while the algorithm is executed we measure the execution time. Posteriori analysis gives accurate values but it is very costly.

As we know that the compile time does not depend on the size of the input. Hence, we will confine ourselves to consider only the run-time which depends on the size of the input and this run-time is denoted by \( TP(n) \). Hence

**Time complexity** \( T(P) = C + TP(n) \).

The time \( (T(P)) \) taken by a program \( P \) is the sum of the compile time and execution time. The compile time does not depend on the instance characteristics, so we concentrate on the runtime of a program. This runtime is denoted by \( tp \) (instance characteristics).

The following equation determines the number of addition, subtraction, multiplication, division compares, loads stores and so on, that would be made by the code for \( p \).

\[
tp(n) = CaADD(n)+ CsSUB(n)+ CmMUL(n)+ CdDIV(n)+ \ldots \ldots \ldots \ldots \ldots
\]

where \( n \) denotes instance characteristics, and \( Ca, Cs, Cm, Cd \) and so on.....

As denote the time needed for an addition, subtraction, multiplication, division and so on, and \( ADD, SUB, MUL, DIV \) and so on, are functions whose values are the number of additions, subtractions, multiplications, divisions and so on. But this method is an impossible task to find out time complexity.

Another method is step count. By using step count, we can determine the number if steps needed by a program to solve a particular problem in 2 ways.

**Method 1:** introduce a global variable “count”, which is initialized to zero. So each time a statement in the signal program is executed, count is incremented by the step count of that statement.
Example:
Algorithm Sum(a, n)
{ 
s:=0;
for i:=1 to n do
{ 
s:=s+a[i];
}
return s;
}

Algorithm sum with count statement added
count:=0;
Algorithm Sum(a,n)
{ 
s:=0; count:=count+1;
for i:=1 to n do 
{ 
count:=count +1;
s:=s+a[i]; count:=count+1;
}
count:=count+1; //for last time of for loop
count:=count+1; //for return statement
return s;
}

Thus the total number of steps are 2n+3

Method 2: The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

<table>
<thead>
<tr>
<th>Statement</th>
<th>S/e</th>
<th>Frequency</th>
<th>Total steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Algorithm Sum(a, n)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2. {</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3. s:=0;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4. for i:=1 to n do</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>5. s:=s+a[i];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>6. return s;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7. }</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2n+3 steps</td>
</tr>
</tbody>
</table>

The S/e (steps per execution) of a statement is the amount by which the count changes as a result of the execution of that statement. The frequency determines the total number of times each statement is executed.

Complexity of Algorithms:
1. Best Case: Inputs are provided in such a way that the minimum time is required to process them.
2. Average Case: The amount of time the algorithm takes on an average set of inputs.
3. Worst Case: The amount of time the algorithm takes on the worst possible set of inputs.

Example: Linear Search

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Best Case: If we want to search an element 3, whether it is present in the array or not. First, A(1) is compared with 3, match occurs. So the number of comparisons is only one. It is observed that search takes minimum number of comparisons, so it comes under best case.

Time complexity is O(1).
Average Case: If we want to search an element 7, whether it is present in the array or not. First, A(1) is compared with 7 i.e., (3=7), no match occurs. Next, compare A(2) and 7, no match occurs. Compare A(3) and A(4) with 7, no match occurs. Up to now 4 comparisons takes place. Now compare A(5) and 7 (i.e., 7=7), so match occurs. The number of comparisons is 5. It is observed that search takes average number of comparisons. So it comes under average case.

Note: If there are n elements, then we require n/2 comparisons.

\[ \therefore \text{Time complexity is } O\left(\frac{n}{2}\right) = O(n) \] (we neglect constant)

Worst Case: If we want to search an element 15, whether it is present in the array or not. First, A(1) is compared with 15 (i.e., 3=15), no match occurs. Continue this process until either element is found or the list is exhausted. The element is found at 9th comparison. So number of comparisons are 9.

\[ \therefore \text{Time complexity is } O(n). \]

Note: If the element is not found in array, then we have to search entire array, so it comes under worst case.

7. Asymptotic Notation:

Accurate measurement of time complexity is possible with asymptotic notation. Asymptotic complexity gives an idea of how rapidly the space requirement or time requirement grow as problem size increase. When there is a computing device that can execute 1000 complex operations per second. The size of the problem is that can be solved in a second or minute or an hour by algorithms of different asymptotic complexity. In general asymptotic complexity is a measure of algorithm not problem. Usually the complexity of an algorithm is as a function relating the input length to the number of steps (time complexity) or storage location (space complexity). For example, the running time is expressed as a function of the input size ‘n’ as follows.

\[ f(n) = n^4 + 100n^2 + 10^n + 50 \] (running time)

There are four important asymptotic notations.

1. Big oh notation (O)
2. Omega notation (Ω).
3. Theta notation (θ)

Let \( f(n) \) and \( g(n) \) are two non-negative functions.

**Big oh notation**

Big oh notation is denoted by ‘O’. it is used to describe the efficiency of an algorithm. It is used to represent the upper bound of an algorithms running time. Using Big O notation, we can give largest amount of time taken by the algorithm to complete.

**Definition:** Let \( f(n) \) and \( g(n) \) be the two non-negative functions. We say that \( f(n) \) is said to be \( O(g(n)) \) if and only if there exists a positive constant ‘c’ and ‘n₀’ such that,

\[ f(n) \leq c \times g(n) \] for all non-negative values of \( n \), where \( n \geq n₀ \).

Here, \( g(n) \) is the upper bound for \( f(n) \).
Ex: Let \( f(n) = 2n^4 + 5n^2 + 2n + 3 \)
\[ \leq 2n^4 + 5n^4 + 2n^4 + 3n^4 \]
\[ \leq (2+5+2+3)n^4 \]
\[ \leq 12n^4. \]
\[ \therefore f(n) = 12n^4. \]

This implies \( g(n) = n^4, \ n \geq 1 \)
\[ \therefore c = 12 \text{ and } n_0 = 1 \]
\[ \therefore f(n) = O(n^4). \]

The above definition states that the function ‘f’ is almost ‘c’ times the function ‘g’ when ‘n’ is greater than or equal to \( n_0 \).

This notion provides an upper bound for the function ‘f’ i.e, the function \( g(n) \) is an upper bound on the value of \( f(n) \) for all \( n \), where \( n \geq n_0 \).

**Big omega notation**

Big omega notation is denoted by ‘\( \Omega \)’. It is used to represent the lower bound of an algorithms running time. Using big omega notation we can give shortest amount of time taken by the algorithm to complete.

**Definition:** The function \( f(n) = \Omega(g(n)) \) (read as for of \( n \) is omega of \( g \) of \( n \)) if and only if there exist positive constants ‘c’ and ‘\( n_0 \)’ such that,
\[ f(n) \geq c \cdot g(n) \text{ for all } n, \ n \geq n_0 \]

**Example:**

Let \( f(n) = 2n^4 + 5n^2 + 2n + 3 \)
\[ \geq 2n^4 \text{ (for example as } n \rightarrow \infty, \text{ lower order terms are insignificant)} \]
\[ \therefore f(n) \geq 2n^4, \ n \geq 1 \]
\[ \therefore g(n) = n^4, \ c = 2 \text{ and } n_0 = 1 \]
\[ \therefore f(n) = \Omega(n^4) \]

**Big Theta notation**

The big theta notation is denoted by ‘\( \Theta \)’. It is in between the upper bound and lower bound of an algorithms running time.

**Definition:** Let \( f(n) \) and \( g(n) \) be the two non-negative functions. We say that \( f(n) \) is said to be \( \Theta(g(n)) \) if and only if there exists a positive constants ‘\( c_1 \)’ and ‘\( c_2 \)’, such that,
\[ c_1g(n) \leq f(n) \leq c_2g((n) \text{ for all non-negative values } n, \text{ where } n \geq n_0. \]

The above definition states that the function \( f(n) \) lies between ‘\( c_1 \)’times the function \( g(n) \) and ‘\( c_2 \)’, times the function \( g(n) \) where ‘\( c_1 \)’ and ‘\( c_2 \)’ are positive constants.

This notation provides both lower and upper bounds for the function \( f(n) \) i.e, \( g(n) \) is both lower and upper bounds on the value of \( f(n) \), for large \( n \). in other words theta notation says that \( f(n) \) is both \( O(g(n)) \) and \( \Omega(g(n)) \) for all \( n \), where \( n \geq n_0. \)
This function $f(n) = \Theta(g(n))$ iff $g(n)$ is both upper and lower bound an $f(n)$.

**Example:**

$f(n) = 2n^4 + 5n^2 + 2n + 3$

$\Rightarrow 2n^4 \leq f(n) \leq 12n^4$

$\Rightarrow 2n^4 \leq f(n) \leq 12n^4, \ n \geq 1$

$\therefore g(n) = n^4$

$\therefore c1=2, \ c2=12 \text{ and } n0=1$

$\therefore f(n) = (n^4)$

---

**Little ‘oh’ notation**

Little oh notation is denoted by “$o$”. the asymptotic upper bound provided by $O$-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n=O(n^2)$ is not. We use $o$-notation to denote an upper bound that is not asymptotically tight.

**Definition:** $f(n) = o(g(n))$, iff $f(n) < c \cdot g(n)$ for any positive constants $c > 0, \ n0 > 0$ and $n > n0$.

Or

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

Find the time complexity for sum $f$ given array elements

The asymptotic complexity of sum is as follows

<table>
<thead>
<tr>
<th>Ex:</th>
<th>Statement</th>
<th>S/e</th>
<th>Frequency</th>
<th>Total steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Algorithm Sum(a, n)</td>
<td>0</td>
<td>-</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>2.</td>
<td>{</td>
<td>0</td>
<td>-</td>
<td>$\Theta(0)$</td>
</tr>
<tr>
<td>3.</td>
<td>s:=0;</td>
<td>1</td>
<td>1</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>4.</td>
<td>for i:=1 to n do</td>
<td>1</td>
<td>n+1</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>5.</td>
<td>s:=s+a[i];</td>
<td>1</td>
<td>n</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>6.</td>
<td>return s;</td>
<td>1</td>
<td>1</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>7.</td>
<td>}</td>
<td>0</td>
<td>-</td>
<td>$\Theta(0)$</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

The time complexity can be calculated as follows-

First find out the basic operation of the above algorithm is-

$S := S + a[i]$ i.e., addition in the loop is the basic operation.
The basic operation is executed every time the loop is executed.
Thus time complexity

\[ T(n) = \sum_{i=1}^{n} 1 = 1 + 1 + 1 + 1 \ldots \ldots \ldots n = n \]

\[ \therefore T(n) = O(n) \]

8 **Probabilistic Analysis:**

Probabilistic analysis of algorithms is an approach to estimate the complexity of an algorithm. It uses the probability in the analysis of problems. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to compute an expected running time of a known algorithm.

The following is the simple example as probabilistic average case analysis.

**Example:** Consider linear search algorithm which searches a target element say \( x \), in the given list of size \( n \). In the worst case, the algorithm will examine all \( n \) elements in the list before terminating.

For a probabilistic average-case analysis, it is generally assumed that all possible terminations are equally likely—that is, the probability that \( x \) will be found at position 1 is \( 1/n \) at position 2 is \( 1/n \) and so on.

The average search cost is therefore the sum of all possible search costs each multiplied by their associated probability.

For example, if \( n=5 \), we would have

Average search cost = \( 1/5(1 + 2 + 3 + 4 + 5) = 3 \).

In general case we have

Average search cost = \( 1/n(n+1)/2 = (n+1)/2 \)

Probabilistic analysis is mainly useful in estimate running time of an algorithm, calculating search costs in a searching algorithm etc.

9. **Amortized Analysis:**

Amortized analysis refers to finding the average running time per operation, over a worst case sequence of operations. That is the main goal of amortized analysis is to analyze the time per operation for a series of operations. Sometimes single operation might be expensive; in that case amortized analysis specifies average over a sequence of operations. Amortized cost per operation for a sequence of \( n \) operations is the total cost of operations divided by \( n \).

For example, if we have 100 operations at cost 1, followed by one operation at cost 100, then amortized cost per operation is \( 200/101 < 2 \). Amortized analysis does not allow random selection of input.

The average case analysis and amortized analysis are different. In average case analysis, we are averaging over all possible inputs whereas in amortized analysis we are averaging over a sequence of operations.

Amortized analysis does not allow random selection of input.

There are several techniques used in amortized analysis.

1. **Aggregate Analysis:** In this type of analysis upper bound \( T(n) \) on the total cost of a sequence of \( n \) operations is decided, then the average cost is calculated as \( T(n)/n \).

2. **Accounting Method:** In this method the individual cost of each operation is determined, by combining immediate execution time and its influence on the running time of future operations.

3. **Potential Method:** It is like the accounting method, but overcharges operations early to compensate for undercharges later.
Divide-and-conquer method: Divide-and-conquer are probably the best known general algorithm design technique. The principle behind the Divide-and-conquer algorithm design technique is that it is easier to solve several smaller instance of a problem than the larger one.

The “divide-and-conquer” technique involves solving a particular problem by dividing it into one or more cub-problems of smaller size, recursively solving each sub-problem and then “merging” the solution of sub-problems to produce a solution to the original problem.

Divide-and-conquer algorithms work according to the following general plan.
1. **Divide**: Divide the problem into a number of smaller sub-problems ideally of about the same size.
2. **Conquer**: The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straightforward manner.
3. **Combine**: If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem.

The following figure shows-

![Divide and Conquer Typical Case](image)

Control abstraction for divide-and-conquer technique:
Control abstraction means a procedure whose flow of control is clear but whose primary operations are satisfied by other procedure whose precise meanings are left undefined.

**Algorithm** DandC(p)
```
if small (p) then
    return S(p)
else
    Divide P into small instances P1, P2, P3, ...., Pk, k≥1;
    Apply DandC to each of these sub-problems;
    return combine (DandC(P1), DandC(P2), ...., DandC(Pk));
```


Algorithm: Control abstraction for divide-and-conquer

DandC(p) is the divide-and-conquer algorithm, where P is the problem to be solved. Small(p) is a Boolean valued function (i.e., either true or false) that determines whether the input size is small enough that the answer can be computed without splitting. If this is so the function S is invoked. Otherwise the problem P is divided into smaller sub-problems. These sub-problems $P_1, P_2, P_3, \ldots, P_k$ are solved by receive applications of DandC.

Combine is a function that combines the solution of the K sub-problems to get the solution for original problem ‘P’.

Example: Specify an application that divide-and-conquer cannot be applied.

Solution: Let us consider the problem of computing the sum of n numbers $a_0, a_1, \ldots, a_{n-1}$. If $n>1$, we divide the problem into two instances of the same problem. That is to compute the sum of the first $[n/2]$ numbers and to compute the sum of the remaining $[n/2]$ numbers. Once each of these two sum is compute (by applying the same method recursively), we can add their values to get the sum in question:

$$a_0 + a_1 + \ldots + a_{n-1} = (a_0 + a_1 + \ldots + a_{[n/2]-1}) + (a_{[n/2]} + \ldots + a_{n-1}).$$

For example, the sum of 1 to 10 numbers is as follows-

$$(1+2+3+4+\ldots+10) = (1+2+3+4+5) + (6+7+8+9+10)$$

$$= [(1+2) + (3+4+5)] + [(6+7) + (8+9+10)]$$

$$= \ldots$$

$$= \ldots$$

$$= (1) + (2) + \ldots + (10).$$

This is not an efficient way to compute the sum of n numbers using divide-and-conquer technique. In this type of problem, it is better to use brute-force method.

Applications of Divide-and-Conquer: The applications of divide-and-conquer methods are-

1. Binary search.
2. Quick sort
3. Merge sort.

Binary Search:

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and-conquer technique.

The process of binary search is as follows:

The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list. If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500.

Trace the binary search algorithm searching for the element -1.
**UNIT-2**  
**DIVIDE AND CONQUER**

**Sol:** The given list of elements are:

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Searching key '-1':

Here the key to search is '-1'.

First calculate mid;

$$\text{Mid} = \frac{\text{low} + \text{high}}{2}$$

$$= \frac{0 + 14}{2} = 7$$

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>Mid</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now mid = \(\frac{0 + 6}{2}\)

=3.

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>Mid</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the search key -1 is less than the middle element (5) in the list. So the search process continues with the first half of the list.

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>Mid</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now mid = \(\frac{0 + 2}{2}\)

=1.

<table>
<thead>
<tr>
<th>Low</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>27</td>
<td>32</td>
<td>33</td>
<td>98</td>
<td>147</td>
<td>154</td>
<td>198</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>Mid</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the search key -1 is found at position 1.
The following algorithm gives the *iterative binary Search Algorithm*
Algorithm BinarySearch(a, n, key)
{
    // a is an array of size n elements
    // key is the element to be searched
    // if key is found in array a, then return j, such that
    //key = a[i]
    //otherwise return -1.
    Low: = 0;
    High: = n-1;
    While (low ≤ high) do
    {
        Mid: = (low + high)/2;
        If ( key = a[mid]) then
            Return mid;
        Else if (key < a[mid])
        {
            High: = mid +1;
        }
        Else if ( key > a[mid])
        {
            Low: = mid +1;
        }
    }
}

The following algorithm gives *Recursive Binary Search*
Algorithms Binsearch ( a, n, key, low, high)
{
    // a is array of size n
    // Key is the element to be searched
    // if key is found then return j, such that key = a[i].
    //otherwise return -1
    If ( low ≤ high) then
    {
        Mid: = (low + high)/2;
        If ( key = a[mid]) then
            Return mid;
        Else if (key < a[mid])
        {
            Binsearch ( a, n, key, low, mid-1);
        }  
        Else if (key > a[mid])
        {
            Binsearch ( a, n, key, mid+1, high);
        }  
        } 
        Return -1;
    }

*Advantages of Binary Search:* The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method.
**Disadvantages of Binary Search:** The disadvantage of binary search is that it can be applied to only a sorted list of elements. The binary search is unsuccessful if the list is unsorted.

**Efficiency of Binary Search:** To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.

**Best Case:** The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is $O(1)$.

**Ex:** Let the given list is: 1, 5, 10, 11, 12.

<table>
<thead>
<tr>
<th>Low</th>
<th>Mid</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Let key = 10.
Since the key is the middle element and is found at our first attempt.

**Worst Case:** Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

<table>
<thead>
<tr>
<th>Items left to search</th>
<th>Comparisons so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if $n$ is the size of the list and $c$ is the number of comparisons, then

$$C = \log_2 n$$

$\therefore$ Efficiency in worst case $= O(\log n)$

**Average Case:** In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as $O(\log n)$.

$\therefore$ Efficiency in average case $= O(\log n)$.

<table>
<thead>
<tr>
<th>Binary Search</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Average Case</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Worst Case</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Space Complexity is $O(n)$

**Quick Sort:**

The quick sort is considered to be a fast method to sort the elements. It was developed by CAR Hoare. This method is based on divide-and-conquer technique i.e. the entire list is divided into various partitions and sorting is applied again and again on these partitions. This method is also called as partition exchange sorts.

The quick sort can be illustrated by the following example

12 6 18 4 9 8 2 15
UNIT-2

DIVIDE AND CONQUER

The reduction step of the quick sort algorithm finds the final position of one of the numbers. In this example, we use the first number, 12, which is called the pivot (rotate) element. This is accomplished as follows-

Let ‘i’ be the position of the second element and ‘j’ be the position of the last element. i.e. $i=2$ and $j=8$, in this example.

Assume that $a[n+1]=\infty$, where ‘$a$’ is an array of size $n$.

\[
\begin{array}{cccccccccc}
12 & 6 & 18 & 4 & 9 & 8 & 2 & 15 & \alpha & 2 & 8 \\
\end{array}
\]

First scan the list from left to right (from $i$ to $j$) can compare each and every element with pivot. This process continues until an element found which is greater than or equal to pivot element. If such an element found, then that element position becomes the value of ‘$i$’.

Now scan the list from right to left (from $j$ to $i$) and compare each and every element with the pivot. This process continues until an element found which is less than or equal to pivot element. If such an element finds then that element’s position become ‘$j$’ value.

Now compare ‘$i$’ and ‘$j$’. If $i<j$, then swap $a[i]$ and $a[j]$. Otherwise swap pivot element and $a[j]$.

Continue the above process the entire list is sorted.

\[
\begin{array}{cccccccccc}
12 & 6 & 18 & 4 & 9 & 8 & 2 & 15 & \alpha & 2 & 8 \\
\end{array}
\]

Since $i=7\leq j=6$, then swap pivot element and 6th element ( jth element), we get

\[
\begin{array}{cccccccccc}
8 & 6 & 2 & 4 & 9 & 12 & 18 & 15 & \_ & \_ & \_ \\
8 & 6 & 2 & 4 & 9 & 12 & 18 & 15 & \alpha & 3 & 7 \\
\end{array}
\]

Thus pivot reaches its original position. The elements on left to the right pivot are smaller than pivot (12) and right to pivot are greater pivot (12).

\[
\begin{array}{cccccccccc}
8 & 6 & 2 & 4 & 9 & 12 & 18 & 15 & \_ & \_ & \_ \\
Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 \\
\end{array}
\]

Now take sub-list1 and sub-list2 and apply the above process recursively, at last we get sorted list.

**Ex 2:** Let the given list is-

\[
\begin{array}{cccccccccc}
8 & 18 & 56 & 34 & 9 & 92 & 6 & 2 & 64 & \_ & \_ \\
8 & 18 & 56 & 34 & 9 & 92 & 6 & 2 & 64 & \alpha & 98 \\
8 & 18 & 56 & 34 & 9 & 92 & 6 & 2 & 64 & \alpha & 3 & 7 \\
8 & 2 & 56 & 34 & 9 & 92 & 6 & 18 & 64 & \alpha & 4 & 3 \\
8 & 2 & 6 & 34 & 9 & 92 & 56 & 18 & 64 & \alpha & 2 & 8 \\
\end{array}
\]

Since $i<j$, then swap jth element, and pivot element, we get

\[
\begin{array}{cccccccccc}
6 & 2 & 8 & 34 & 9 & 92 & 56 & 18 & 64 & \_ & \_ \\
Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 & Sublist 1 & Sublist 2 \\
\end{array}
\]
UNIT-2  
DIVIDE AND CONQUER

Now take a sub-list that has more than one element and follow the same process as above. At last, we get the sorted list that is, we get

\[
\begin{align*}
2 & \quad 6 & \quad 8 & \quad 9 & \quad 18 & \quad 34 & \quad 56 & \quad 64 & \quad 92
\end{align*}
\]

The following algorithm shows the quick sort algorithm-

**Algorithm** Quicksort(i, j)

```
Algorithm Quicksort(i, j)
{
   // sorts the array from a[i] through a[j]
   If ( i < j) then   //if there are more than one element
   {
      //divide P into two sub-programs
      K: = partition (a, i, j+1);
      //Here K denotes the position of the partitioning element
      //solve the sub problems
      Quicksort(i, K-1);
      Quicksort(K+1, j);
      // There is no need for combining solution
   }
}
```

**Algorithm** Partition (a, left, right)

```
Algorithm Partition (a, left, right)
{
   // The element from a[left] through a[right] are rearranged in such a manner that if initially
   // pivot =a[left] then after completion a[j]= pivot, then return. Here j is the position where
   // pivot partition the list into two partitions. Note that a[right]= \infty.
   pivot: a[left];
   i:= left;  j:=right;
   repeat
   {
      repeat
      i: =i+1;
      until (a[i] \geq pivot);
      repeat
      j: =j-1;
      until (a[j] < pivot);
      if( i<j) then
      Swap (a, i, j);
   }until (i \geq j);
   a[left]: = a[j];
a[j]: = pivot;
return j;
}
```

Algorithm Swap (a, i, j)

```
Algorithm Swap (a, i, j)
{
   //Example a[i] with a[j]
   temp:= a[i];
a[i]: = a[j];
a[j]:= temp;
}
```
Advantages of Quick-sort: Quick-sort is the fastest sorting method among all the sorting methods. But it is somewhat complex and little difficult to implement than other sorting methods.

Efficiency of Quick-sort: The efficiency of Quick-sort depends upon the selection of pivot element.

Best Case: In best case, consider the following two assumptions-
1. The pivot, which we choose, will always be swapped into the exactly the middle of the list. And also consider pivot will have an equal number of elements both to its left and right.
2. The number of elements in the list is a power of 2 i.e. \( n = 2^y \).

This can be rewritten as, \( y = \log_2 n \).

<table>
<thead>
<tr>
<th>Pass</th>
<th>Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>2</td>
<td>( 2 \times (n/2) )</td>
</tr>
<tr>
<td>3</td>
<td>( 4 \times (n/4) )</td>
</tr>
<tr>
<td>( y )</td>
<td>( x \times (n/x) )</td>
</tr>
</tbody>
</table>

Thus, the total number of comparisons would be
\[
O(n) + O(n) + O(n) + \ldots \ldots (y \text{ terms})
\]
\[
= O(n \times y).
\]

\[\therefore\] Efficency in best case \( O(n \log n) \) (\( \therefore y = \log_2 n \))
**UNIT-2**  
**DIVIDE AND CONQUER**

**Worst Case:** In worst case, assume that the pivot partition the list into two parts, so that one of the partition has no elements while the other has all the other elements.

![Subproblem sizes and total partitioning time diagram]

\[
\begin{align*}
\text{Subproblem sizes} & \quad \text{Total partitioning time for all subproblems of this size} \\
\vdots & \quad \vdots \\
2 & 2c \\
0 & 0 \\
n & cn \\
n-1 & c(n-1) \\
n-2 & c(n-2) \\
n-3 & c(n-3) \\
& \vdots \\
0 & 0
\end{align*}
\]

\[
\therefore \quad \text{Total number of comparisons will be-} \\
(n - 1) + (n - 2) + (n - 3) + \ldots \ldots + 2 - 1 = \frac{n(n - 1)}{2} = O(n^2)
\]

\[
\therefore \quad \text{Thus, the efficiency of quick-sort in worst case is } O(n^2).
\]

**Average Case:** Let \(cA(n)\) be the average number of key comparisons made by quick-sort on a list of elements of size \(n\) assuming that the partitions split can happen in each position \(k(1 \leq k \leq n)\) with the same probability \(1/n\), we get the following recurrence relation.

![Subproblem size and total partitioning time diagram]

\[
\begin{align*}
\text{Subproblem size} & \quad \text{Total partitioning time for all subproblems of this size} \\
\log_4 n & \quad cn \\
\log_{16} n & \quad cn \\
\log_{64} n & \quad cn \\
1 & \quad cn < cn \\
\end{align*}
\]
The left child of each node represents a sub-problem size 1/4 as large, and the right child represents a sub-problem size 3/4 as large.

There are $\log_{4/3} n$ levels, and so the total partitioning time is $O(n \log_{4/3} n)$. Now, there's a mathematical fact that

$$\log_a n = \frac{\log_b n}{\log_b a}$$

for all positive numbers $a$, $b$, and $n$. Letting $a=4/3$ and $b=2$, we get that

$$\log_{4/3} n = \frac{\log n}{\log(4/3)}$$

### Quick Sort

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Case</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Average Case</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Space Complexity** $O(n)$

### Merge Sort:

Merge sort is based on divide-and-conquer technique. Merge sort method is a two phase process-

1. Dividing
2. Merging

**Dividing Phase:** During the dividing phase, each time the given list of elements is divided into two parts. This division process continues until the list is small enough to divide.

**Merging Phase:** Merging is the process of combining two sorted lists, so that, the resultant list is also the sorted one. Suppose $A$ is a sorted list with $n$ element and $B$ is a sorted list with $n_2$ elements. The operation that combines the elements of $A$ and $B$ into a single sorted list $C$ with $n = n_1 + n_2$ elements is called merging.

**Algorithm-(Divide algorithm)**

**Algorithm** Divide (a, low, high)

```java
// a is an array, low is the starting index and high is the end index of a

If (low < high) then
{
    Mid = (low + high)/2;
    Divide(a, low, mid);
    Divide(a, mid + 1, high);
    Merge(a, low, mid, high);
}
```
The merging algorithm is as follows-

Algorithm Merge( a, low, mid, high)
{
    L:= low;
    H:= high;
    J:= mid +1;
    K:= low;

    While (low ≤ mid AND j ≤ high) do
    {
        If (a[low < a[j]) then
        {
            B[k] = a[low];
            K:= k+1;
            Low:= low+1;
        }
        Else
        {
            B[k]= a[j];
            K:= k+1;
            J:= j+1;
        }
    }

    While (low ≤ mid) do
    {
        B[k]=a[low];
        K:= k+1;
        Low:= low + 1;
    }

    While (j ≤ high) do
    {
        B[k]=a[j];
        K:= k+1;
        j:= j + 1;
    }

    //copy elements of b to a
    For i := 1 to n do
    {
        A[i]: =b[i];
    }
}
**UNIT-2**

**DIVIDE AND CONQUER**

Ex: Let the list is: - 500, 345, 13, 256, 98, 1, 12, 3, 34, 45, 78, 92.

<table>
<thead>
<tr>
<th>500</th>
<th>345</th>
<th>13</th>
<th>256</th>
<th>98</th>
<th>1</th>
<th>12</th>
<th>3</th>
<th>34</th>
<th>45</th>
<th>78</th>
<th>92</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>345</td>
<td>13</td>
<td>256</td>
<td>98</td>
<td>1</td>
<td>12</td>
<td>3</td>
<td>34</td>
<td>45</td>
<td>78</td>
<td>92</td>
</tr>
<tr>
<td>500</td>
<td>345</td>
<td>13</td>
<td>256</td>
<td>98</td>
<td>1</td>
<td>12</td>
<td>3</td>
<td>34</td>
<td>45</td>
<td>78</td>
<td>92</td>
</tr>
</tbody>
</table>

The merge sort algorithm works as follows-

**Step 1:** If the length of the list is 0 or 1, then it is already sorted, otherwise,

**Step 2:** Divide the unsorted list into two sub-lists of about half the size.

**Step 3:** Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

**Step 4:** Apply merging to each sub-list and continue this process until we get one sorted list.

**Efficiency of Merge List:** Let ‘n’ be the size of the given list/ then the running time for merge sort is given by the recurrence relation.

\[
T(n) = \begin{cases} 
  a & \text{if } n=1, \text{ } a \text{ is a constant} \\
  2T(n/2) + Cn & \text{if } n>1, \text{ } C \text{ is constant} 
\end{cases}
\]

Assume that ‘n’ is a power of 2 i.e. \(n=2^k\).

This can be rewritten as \(k=\log_2 n\).

Let \(T(n) = 2T(n/2) + Cn\) ——— 1

We can solve this equation by using successive substitution.
UNIT-2  
DIVIDE AND CONQUER

Replace \( n \) by \( n/2 \) in equation, (1), we get

\[
T(n/2) = 2T(n/4) + \frac{Cn}{2} \tag{2}
\]

Thus, \( T(n) = 2 \left( 2T(n/4) + \frac{Cn}{2} \right) + Cn \)

\[
= 4T(n/4) + 2Cn
\]

\[
= 4T\left( 2T(n/8) + \frac{Cn}{4} \right) + 2Cn
\]

\[
\vdots
\]

\[
= 2^k T(1) + KCn \quad (\because \, k = \log_2 n)
\]

\[
= a_n + Cn \log n
\]

\[
\therefore T(n) = O(n \log n)
\]

<table>
<thead>
<tr>
<th></th>
<th>( O(n \log n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Best case</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Average case</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>( O(2n) )</td>
</tr>
</tbody>
</table>
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
1. GENERAL METHOD

**Greedy method:** It is most straight forward method. It is popular for obtaining the optimized solutions.

**Optimization Problem:** An optimization problem is the problem of finding the best solution (optimal solution) from all the feasible solutions (practicable of possible solutions). In an optimization problem we are given a set of constraints and an optimization functions. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called optimal solution.

**Ex:** Problem: Finding a minimum spanning tree from a weighted connected directed graph G.

**Constraints:** Every time a minimum edge is added to the tree and adding of an edge does not form a simple circuit.

**Feasible solutions:** The feasible solutions are the spanning trees of the given graph G.

**Optimal solution:** An optimal solution is a spanning tree with minimum cost i.e. minimum spanning tree.

**Q:** Find the minimum spanning tree for the following graph.

![Graph G](image)

The feasible solutions are the spanning tree of the graph G. Those are

1. Total Weights=6
2. Total Weights=6
3. Total Weights=7
4. Total Weights=5

From the above spanning tree the figure 4 gives the optimal solution, because it is the spanning tree with the minimum cost i.e. it is a minimum spanning tree of the graph G.

The greedy technique suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far until a complete solution to the problem is reached to each step, the choice made must be feasible, locally optimal and irrecoverable.

Feasible: The choice which is made has to be satisfying the problems constraints.
Locally optimal: The choice has to be the best local choice among all feasible choices available on that step.
Irrecoverable: The choice once made cannot be changed on subsequent steps of the algorithm (Greedy method).
Control Abstraction for Greedy Method:

Algorithm GreedyMethod (a, n)
{
    // a is an array of n inputs
    Solution: = Ø;
    for i: =0 to n do
    {
        s: = select (a); // s is a solution from the given input domain
        if (feasible (Solution, s)) then
        {
            Solution: = union (Solution, s);
        }
        else
        {
            reject (); // if solution is not feasible reject it.
        }
    }
    return solution;
}

In greedy method there are three important activities.
1. A selection of solution from the given input domain is performed, i.e. \( s = \text{select}(a) \).
2. The feasibility of the solution is performed, by using feasible \( (\text{solution, s}) \) and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution.

Q: A child buys a candy 42 rupees and gives a 100 note to the cashier. Then the cashier wishes to return change using the fewest number of coins. Assume that the cashier has Rs. 1, Rs. 5 and Rs. 10 coins.

This problem can be solved using the greedy method.

2. APPLICATION - JOB SEQUENCING WITH DEADLINES

This problem consists of \( n \) jobs each associated with a deadline and profit and our objective is to earn maximum profit. We will earn profit only when job is completed on or before deadline. We assume that each job will take unit time to complete.

Points to remember:

- In this problem we have \( n \) jobs \( j_1, j_2, \ldots j_n \), each has an associated deadlines are \( d_1, d_2, \ldots d_n \) and profits are \( p_1, p_2, \ldots p_n \).
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.
Example: Consider the following 5 jobs and their associated deadline and profit.

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>j1</td>
<td>j2</td>
<td>j3</td>
<td>j4</td>
<td>j5</td>
</tr>
<tr>
<td>DEADLINE</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>PROFIT</td>
<td>60</td>
<td>100</td>
<td>20</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

Sort the jobs according to their profit in descending order.
Note! If two or more jobs are having the same profit then sorts them as per their entry in the job list.

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>j2</td>
<td>j1</td>
<td>j4</td>
<td>j3</td>
<td>j5</td>
</tr>
<tr>
<td>DEADLINE</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>PROFIT</td>
<td>100</td>
<td>60</td>
<td>40</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Find the maximum deadline value
Looking at the jobs we can say the max deadline value is 3. So, dmax = 3
As dmax = 3 so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

<table>
<thead>
<tr>
<th>time slot</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>EMPTY</td>
<td>EMPTY</td>
<td>EMPTY</td>
</tr>
</tbody>
</table>

Total number of jobs is 5. So we can write n = 5.
Note!
If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.
Similarly, if we look at job j1 it has a deadline 2. This means we have to complete job j1 on or before time slot 2 in order to earn its profit.
Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit.
Our objective is to select jobs that will give us higher profit.

<table>
<thead>
<tr>
<th>time slot</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>J1</td>
<td>J2</td>
<td>J4</td>
</tr>
<tr>
<td>Profit</td>
<td>100</td>
<td>60</td>
<td>20</td>
</tr>
</tbody>
</table>

Total Profit is 180
3. APPLICATION - KNAPSACK PROBLEM

In this problem the objective is to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.

**Points to remember:**

In this problem we have a Knapsack that has a weight limit \( W \)
There are items \( i_1, i_2, \ldots, \) in each having weight \( w_1, w_2, \ldots, w_n \) and some benefit (value or profit) associated with it \( v_1, v_2, \ldots, v_n \)

Our objective is to maximise the benefit such that the total weight inside the knapsack is at most \( W \). And we are also allowed to take an item in fractional part.
**UNIT-III**

**GREEDY METHOD**

\[
\begin{align*}
\text{max } & \sum_{0 \leq i \leq n} p_i x_i \\
\text{s.t.} & \quad \sum_{0 \leq i \leq n} w_i x_i \leq M \\
& \quad 0 \leq x_i \leq 1 \\
& \quad p_i \geq 0, w_i \geq 0, 0 \leq i < n
\end{align*}
\]

**Example:** Assume that we have a knapsack with max weight capacity, \( W = 16 \). Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

<table>
<thead>
<tr>
<th>ITEM</th>
<th>WEIGHT</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>i2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>i3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>i4</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>i5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>i6</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Steps
1. Calculate value per weight for each item (we can call this value density)
2. Sort the items as per the value density in descending order
3. Take as much item as possible not already taken in the knapsack

Compute density = \( \frac{\text{value}}{\text{weight}} \)

<table>
<thead>
<tr>
<th>ITEM</th>
<th>WEIGHT</th>
<th>VALUE</th>
<th>DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>6</td>
<td>6</td>
<td>1.000</td>
</tr>
<tr>
<td>i2</td>
<td>10</td>
<td>2</td>
<td>0.200</td>
</tr>
<tr>
<td>i3</td>
<td>3</td>
<td>1</td>
<td>0.333</td>
</tr>
<tr>
<td>i4</td>
<td>5</td>
<td>8</td>
<td>1.600</td>
</tr>
<tr>
<td>i5</td>
<td>1</td>
<td>3</td>
<td>3.000</td>
</tr>
<tr>
<td>i6</td>
<td>3</td>
<td>5</td>
<td>1.667</td>
</tr>
</tbody>
</table>

Sort the items as per density in descending order

<table>
<thead>
<tr>
<th>ITEM</th>
<th>WEIGHT</th>
<th>VALUE</th>
<th>DENSITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>i5</td>
<td>1</td>
<td>3</td>
<td>3.000</td>
</tr>
</tbody>
</table>
UNIT-III

GREEDY METHOD

<table>
<thead>
<tr>
<th>ITEM</th>
<th>WEIGHT</th>
<th>VALUE</th>
<th>TOTAL WEIGHT</th>
<th>TOTAL BENEFIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>i6</td>
<td>3</td>
<td>5</td>
<td>1.667</td>
<td></td>
</tr>
<tr>
<td>i4</td>
<td>5</td>
<td>8</td>
<td>1.600</td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td>6</td>
<td>6</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>i3</td>
<td>3</td>
<td>1</td>
<td>0.333</td>
<td></td>
</tr>
<tr>
<td>i2</td>
<td>10</td>
<td>2</td>
<td>0.200</td>
<td></td>
</tr>
</tbody>
</table>

Now we will pick items such that our benefit is maximum and total weight of the selected items is at most W.

Our objective is to fill the knapsack with items to get maximum benefit without crossing the weight limit W = 16.

How to fill Knapsack Table?
- is WEIGHT(i) + TOTAL WEIGHT <= W
- if its YES
- then we take the whole item

How to find the Benefit?
- If an item value is 10 and weight is 5
- And if you are taking it completely
- Then,
- benefit = (weight taken) x (total value of the item / total weight of the item)

weight taken = 5 (as we are taking the complete (full) item, no fraction)
total value of the item = 10
total weight of the item = 5

So, benefit = 5 x (10/5) = 10

On the other hand if you are taking say, 1/2 of the item
Then,
weight taken = 5 x (1/2) = 5/2 (as we are taking 1/2 item)
So, benefit = (weight taken) x (total value of the item / total weight of the item)

= (5/2) x (10/5)
= 5

Values after calculation

<table>
<thead>
<tr>
<th>ITEM</th>
<th>WEIGHT</th>
<th>VALUE</th>
<th>TOTAL WEIGHT</th>
<th>TOTAL BENEFIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>i5</td>
<td>1</td>
<td>3</td>
<td>1.000</td>
<td>3.000</td>
</tr>
<tr>
<td>i6</td>
<td>3</td>
<td>5</td>
<td>4.000</td>
<td>8.000</td>
</tr>
</tbody>
</table>
UNIT-III

<table>
<thead>
<tr>
<th>i4</th>
<th>5</th>
<th>8</th>
<th>9.000</th>
<th>16.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>6</td>
<td>6</td>
<td>15.000</td>
<td>22.000</td>
</tr>
<tr>
<td>i3</td>
<td>1</td>
<td>0.333</td>
<td>16.000</td>
<td>22.333</td>
</tr>
</tbody>
</table>

So, total weight in the knapsack = 16 and total value inside it = 22.333336

\[
n=3, \ m=20, (p_1, p_2, p_3)=(25, 24, 15), (w_1, w_2, w_3)=(18, 15, 10)
\]

\[
(x_1, x_2, x_3) = \sum w_i x_i, \sum p_i x_i
\]

1. \((1/2, 1/3, 1/4)\) \(16.5 \quad 24.25\)
2. \((1, 2/15, 0)\) \(20 \quad 28.2\)
3. \((0, 2/3, 1)\) \(20 \quad 31\)
4. \((0, 1, 1/2)\) \(20 \quad 31.5\)

Algorithm:

```c
void GreedyKnapsack(float m, int n)
{
    // p[1:n] and w[1:n] contain the profits and weights
    // respectively of the n objects ordered such that
    // p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
    // size and x[1:n] is the solution vector.
    // Initialize x.
    float U = m;
    for (i=1; i<=n; i++)
    {
        if (w[i] > U) break;
        x[i] = 1.0;
        U -= w[i];
    }
    if (i <= n) x[i] = U/w[i];
}
```

Time Complexity = \(O(n^2)\)

4. APPLICATION - MINIMUM SPANNING TREE

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.

Note: Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.
We found three spanning trees off one complete graph. A complete undirected graph can have maximum $n^{n-2}$ number of spanning trees, where $n$ is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

**General Properties of Spanning Tree**

- 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.

**Mathematical Properties of Spanning Tree**

- 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 $n^{n-2}$ 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.

**Application of Spanning Tree**

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

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

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

**Minimum Spanning Tree (MST)**

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms (greedy algorithms):
UNIT-III  

GREEDY METHOD

1. Kruskal's Algorithm
2. Prim's Algorithm

i. Kruskal's Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example:

Step 1 - Remove all loops and Parallel Edges
Remove all loops and parallel edges from the given graph.

In case of parallel edges, keep the one which has the least cost associated and remove all others.
Step 2 - Arrange all edges in their increasing order of weight
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Step 3 - Add the edge which has the least weightage
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

Next cost is 3, and associated edges are A,C and C,D. We add them again –

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.
ii. **Prim’s Algorithm**

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example.

![Graph](https://www.tutorialsduniya.com)

**Step 1 - Remove all loops and parallel edges**
Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

Step 2 - Choose any arbitrary node as root node

In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.
After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

We may find that the output spanning tree of the same graph using two different algorithms is same.
5. APPLICATION - SINGLE SOURCE SHORTEST PATH PROBLEM

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Time Complexity = O (n^2)
Example:

![Graph](image)

Algorithm:
```
ShortestPaths(int v, float cost[][SIZE], float dist[], int n)
{
    int u; bool S[SIZE];
    for (int i=1; i<=n; i++) { // Initialize S.
        S[i] = false; dist[i] = cost[v][i];
    }
    S[v]=true; dist[v]=0.0; // Put v in S.
    for (int num = 2; num < n; num++) {
        // Determine n-1 paths from v.
        choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] = true; // Put u in S.
        for (int w=1; w<=n; w++) //Update distances.
            if ((( S[w]=false) && (dist[w] > dist[u] + cost[u][w]))
                dist[w] = dist[u] + cost[u][w];
    }
}
```

Time Complexity = O (n²)

<table>
<thead>
<tr>
<th>GREEDY APPROACH</th>
<th>DIVIDE AND CONQUER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Many decisions and sequences are guaranteed and all the overlapping subinstances are considered.</td>
<td>1. Divide the given problem into many subproblems. Find the individual solutions and combine them to get the solution for the main problem</td>
</tr>
<tr>
<td>2. Follows Bottom-up technique</td>
<td>2. Follows top-down technique</td>
</tr>
<tr>
<td>3. Split the input at every possible points rather than at a particular point</td>
<td>3. Split the input only at specific points (midpoint), each problem is independent.</td>
</tr>
<tr>
<td>4. Subproblems are dependent on the main Problem</td>
<td>4. Subproblems are independent on the main Problem</td>
</tr>
<tr>
<td>5. Time taken by this approach is not that much efficient when compared with DAC.</td>
<td>5. Time taken by this approach is efficient when compared with GA.</td>
</tr>
<tr>
<td>6. Space requirement is less when compared DAC approach.</td>
<td>6. Space requirement is very much high when compared GA approach.</td>
</tr>
</tbody>
</table>
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research
Dynamic Programming:- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub problems. Dynamic programming is applicable when the sub problems are not independent, that is, when sub problems share sub problems. A dynamic-programming algorithm solves every sub sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub sub-problem is encountered. Dynamic programming is typically applied to optimization problems. The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

The dynamic programming technique was developed by Bellman based upon the principle known as principle of optimality. Dynamic programming uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to sub problems and, having solved the sub problems, we find an optimal solution to the problem.

Application-I: Matrix chain multiplication:- Matrix chain multiplication is an example of dynamic programming. We are given a sequence (chain) \( A_1, A_2, ..., A_n \) of matrices to be multiplied, and we wish to compute the product \( A_1 \times A_2 \times A_3 \times ... \times A_n \). We can evaluate the expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Matrix multiplication is associative, and so all parenthesize yield the same product. For example, if the chain of matrices is \( A_1, A_2, A_3, A_4 \), the product \( A_1 \times A_2 \times A_3 \times A_4 \) can be fully parenthesized in five distinct ways:

\[
(A_1 (A_2 (A_3 A_4))) ,
(A_1 (((A_2 A_3) A_4))),
((A_1 A_2) (A_3 A_4)),
((A_1 (A_2 A_3)) A_4),
(((A_1 A_2) A_3) A_4).
\]

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B. If A is a \( m \times n \) matrix and B is a \( p \times q \) matrix, the resulting matrix C is a \( m \times q \) matrix. The standard algorithm is given below.

```
Algorithm Matrix_Mul(A, B)
{
    if (n ≠ P) then
        Error "incompatible dimensions"
    else
        for i ← 1 to m do
            for j ← 1 to q do
                { 
                    C[i, j] ← 0
                    for k ← 1 to n do
                }
        return C
    }
```

The time to compute C is the number of multiplications which is \( mnq \) or \( mpq \).
UNIT-IV  
DYNAMIC PROGRAMMING

Example 1:- To illustrate the different costs incurred by different parenthesization of a matrix product, consider the problem to find the product of three matrices $A_1, A_2, A_3$ i.e. $A_1 * A_2 * A_3$ of three matrices. Suppose that the dimensions of the matrices are $10 \times 100$, $100 \times 5$, and $5 \times 50$, respectively. If we multiply according to the parenthesization:

$((A_1 A_2) A_3) = 10 \times 100 \times 5 + 100 \times 5 \times 50 = 7500$

Thus, computing the product according to the first parenthesization is 10 times faster.

Definition:- The matrix-chain multiplication problem can be stated as follows: given a chain $A_1, A_2, ..., A_n$ of $n$ matrices, where for $i = 1, 2, ..., n$, matrix $A_i$ has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1 A_2 A_3$ in a way that minimizes the number of scalar multiplications.

Note:- In the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Solving the matrix-chain multiplication problem by dynamic programming

Step 1: The structure of an optimal parenthesization. Our first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. For the matrix-chain multiplication problem, we can perform this step as follows. For example any parenthesization of the product $A_i A_{i+1} A_j$ must split the product between $A_k$ and $A_{k+1}$ for some integer $k$ in the range $i \leq k < j$. That is, for some value of $k$, we first compute the matrices $A_{i,k}$ and $A_{k+1,j}$ and then multiply them together to produce the final product $A_{i,j}$. The cost of this parenthesization is thus the cost of computing the matrix $A_{i,k}$, plus the cost of computing $A_{k+1,j}$, plus the cost of multiplying them together.

Step 2: A recursive solution. Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, We can define $m[i, j]$ recursively as follows. If $i = j$, the matrix $A_{i,j} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $M_{i,j} = 0$ for $i = j$

$$M_{i,j} = \min \{ M_{i,k} + M_{k+1,j} + P_i P_k P_j \} \quad \text{for } i < j$$

Step 3: Computing the optimal costs. We perform the third step of the dynamic-programming paradigm and compute the optimal cost by using a tabular, bottom-up approach.

Step 4: Constructing an optimal solution. In the first level we compare $M_{12}$ and $M_{23}$. When $M_{12} < M_{23}$, we parenthesize the $A_1 A_2$ in the product $A_1 A_2 A_3$ i.e. $(A_1 A_2) A_3$ and parenthesize the $A_2 A_3$ in the product $A_1 A_2 A_3$ i.e., $A_1 (A_2 A_3)$ when $M_{12} > M_{23}$. This process is repeated until the whole product is parenthesized. The top entry in the table i.e $M_{13}$ gives the optimum cost of matrix chain multiplication.

Example:- Find an optimal parenthesization of a matrix-chain product whose dimensions are given in the table below.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$5 \times 4$</td>
</tr>
<tr>
<td>Q</td>
<td>$4 \times 6$</td>
</tr>
<tr>
<td>R</td>
<td>$6 \times 2$</td>
</tr>
<tr>
<td>T</td>
<td>$2 \times 7$</td>
</tr>
</tbody>
</table>
Solution: Given 
\[ P_0 = 5, \quad P_1 = 4, \quad P_2 = 6, \quad P_3 = 2, \quad P_4 = 7 \]
The Bottom level of the table is to be initialized.
\[ M_{ij} = 0 \text{ where } i = j \]

To compute \( M_{ij} \) when \( i < j \),
\[ M_{ij} = \min \{ M_{i,k} + M_{k,j} + P_{i-1}P_{j} \} \quad \text{for } i < j \quad \text{i} \leq k < j \]

Thus \( M_{12} = \min \{ M_{11} + M_{22} + P_0P_1P_2 \} = 0 + 0 + 5 \times 4 \times 6 = 120 \)
\( M_{23} = \min \{ M_{22} + M_{33} + P_1P_2P_3 \} = 0 + 0 + 4 \times 6 \times 2 = 48 \)
\( M_{34} = \min \{ M_{33} + M_{44} + P_2P_3P_4 \} = 0 + 0 + 6 \times 2 \times 7 = 84 \)
\( M_{13} = \min \{ M_{11} + M_{23} + P_0P_1P_3, \quad M_{12} + M_{33} + P_0P_2P_3 \} \)
\[ = \min\{0 + 48 + 5 \times 4 \times 2, \quad 120 + 0 + 5 \times 6 \times 2\} = \min\{88, 180\} = 88 \]
\( M_{24} = \min \{ M_{22} + M_{34} + P_0P_2P_4, \quad M_{23} + M_{44} + P_1P_4 \} \)
\[ = \min\{0 + 84 + 4 \times 6 \times 7, \quad 48 + 0 + 4 \times 2 \times 7\} = \min\{252, 104\} = 104 \]
\( M_{14} = \min \{ M_{11} + M_{24} + P_0P_3P_4, \quad M_{12} + M_{34} + P_1P_2P_4, \quad M_{13} + M_{44} + P_0P_3P_4 \} \)
\[ = \min\{0 + 104 + 5 \times 4 \times 7, \quad 120 + 84 + 5 \times 6 \times 7, \quad 88 + 0 + 5 \times 2 \times 7\} \]
\[ = \min\{244, 414, 158\} = 158 \]

In the first level when we compare \( M_{12} \), \( M_{23} \) and \( M_{34} \). As \( M_{23} = 48 \) is minimum among three we parenthesize the QR in the product PQRT i.e \( P(QR)T \). In the second level when we compare \( M_{13} \) and \( M_{24} \). As \( M_{13} = 88 \) is minimum among the two we parenthesize the P and QR in the product PQRT i.e \( (P(QR))T \). Finally we parenthesize the whole product i.e \( ((P(QR))T) \). The top entry in the table i.e \( M_{14} \) gives the optimum cost of \( ((P(QR))T) \).

Verification: The chain of matrices is P, Q, R, T, the product P x Q x R x T can be fully parenthesized in five distinct ways:

1. \( (P(Q(RT))) \)
2. \( (P((QR)T)) \)
3. \( (((PQ)(RT))) \)
4. \( ((P(QR))T) \)
5. \( (((PQ)R)T) \)

Cost of \( (P(Q(RT))) \) = \( 5 \times 4 \times 7 + 4 \times 6 \times 7 + 6 \times 2 \times 7 = 392 \)
Cost of \( (P((QR)T)) \) = \( 5 \times 4 \times 7 + 4 \times 6 \times 2 + 4 \times 2 \times 7 = 244 \)
Cost of \( (((PQ)(RT))) \) = \( 5 \times 4 \times 6 + 6 \times 2 \times 7 + 5 \times 6 \times 7 = 414 \)
Cost of \( ((P(QR))T) \) = \( 5 \times 4 \times 2 + 4 \times 6 \times 2 + 5 \times 2 \times 7 = 158 \)
Cost of \( (((PQ)R)T) \) = \( 5 \times 4 \times 6 + 5 \times 6 \times 2 + 5 \times 2 \times 7 = 250 \)
From the above manual method also we find the optimal cost is 158 and the order of matrix multiplication is ((PQR)T)

Algorithm Matrix_Chain_Mul(p)
{
    for i = 1 to n do
        M[i, i] = 0
    for len = 2 to n do
    {
        for i = 1 to n - len + 1 do
        {
            j ← i + l - 1
            M[i, j] ← ∞
            for k = i to j - 1 do
                q = M[i, k] + M[k + 1, j] + P_i-1P_kP_j
                if q < M[i, j] then
                {
                    M[i, j] ← q
                }
        }
    }
    return m
}

*Time Complexity:* Algorithm Matrix_Chain_Mul uses first For loop to initialize M[i,j] which takes O(n).
M[i, j] value is computed using three For loops which takes O(n^3). Thus the overall time complexity of Matrix_Chain_Mul is O(n^3).

**Application-II: OBST**

**Binary Search Trees:**
- A binary search tree T is a binary tree, either it is empty or each node in the tree contains an identifier and,
  1. All identifiers in the left subtree of T are less than the identifier in the root node T.
  2. All identifiers in the right subtree are greater than the identifier in the root node T.
  3. The left and right subtree of T are also binary search trees.

**Optimal Binary Search Tree problem:** Given a sequence K = {k1, k2, ..., kn } of n distinct keys in sorted order (so that k1 < k2 < ... < kn), and we wish to build a binary search tree from these keys such that the cost of the binary search tree is minimum. For each key ki, we have a probability pi that a search will be for ki. Some searches may be for values not in K, and so we also have n + 1 "dummy keys" E0, E1, E2, ..., En representing values not in K. In particular, E0 represents all values less than k1, En represents all values greater than kn, and for i = 1, 2, ..., n -1, the dummy key Ei represents all values between ki and ki+1. For each dummy key Ei, we have a probability qi that a search will correspond to Ei.

Number of possible binary search trees for n keys is given by the following formula.

\[
\frac{1}{n+1} 2nC_n
\]

Cost of binary search tree is calculated as follows

\[
\sum_{i=1}^{n} p(i) \times \text{level}(a_i) + \sum_{i=0}^{n} q(i) \times (\text{level}(E_i) - 1)
\]
Solving the Optimal Binary Search Tree problem by dynamic programming

**Step 1: The structure of an optimal binary search tree**

Construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i$, ..., $k_j$, one of these keys, say $k_r$ ($i \leq r \leq j$), will be the root of an optimal subtree containing these keys. The left subtree of the root $k_r$ will contain the keys $k_i$, ..., $k_{r-1}$ (and dummy keys $E_{i-1}$, ..., $E_{r-1}$), and the right subtree will contain the keys $k_{r+1}$, ..., $k_j$ (and dummy keys $E_r$, ..., $E_j$).

**Step 2: A recursive solution**

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the optimal binary search tree problem, We can define $W[i, j]$, $C[i, j]$, and $r[i, j]$ recursively as follows.

\[
\begin{align*}
W[i,j], C[i,j], r[i,j] &= \{ q_i, 0, 0 \} \\
W[i, j] &= \{ W[i, j-1] + q_j + P_j \} \\
C[i, j] &= \min \{ C[i, k-1] + C[k, j] + W[i, j] \} \\
r[i, j] &= k
\end{align*}
\]

for $i < j$

\[
\begin{align*}
W[i, j] &= q_i + q_j + P_j \\
C[i, j] &= \min \{ C[i, k-1] + C[k, j] + W[i, j] \} \\
r[i, j] &= k
\end{align*}
\]

for $i < j$

**Step 3: Computing the expected search cost of an optimal binary search tree**

Computing the optimal costs, we perform the third step of the dynamic-programming paradigm and compute the optimal cost by using a tabular, bottom-up approach.

**Algorithm OBST(p, q, n)**

\[
\begin{align*}
&\text{for } i = 0 \text{ to } n-1 \text{ do} \\
&\quad \{ \\
&\quad \quad w[i,i] = q[i]; c[i,i] = 0; r[i,i] = 0; \\
&\quad \quad w[i,i+1] = q[i] + q[i+1] + p[i+1]; \\
&\quad \quad r[i,i+1] = i+1; \\
&\quad \quad c[i,i+1] = q[i] + q[i+1] + p[i+1]; \\
&\quad \}
\end{align*}
\]

\[
\begin{align*}
&\text{for } m = 2 \text{ to } n \text{ do} \\
&\quad \{ \\
&\quad \quad \text{for } i = 0 \text{ to } n - m \text{ do} \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad j = i + m; \\
&\quad \quad \quad \}\}
\end{align*}
\]
\[ w[i,j] = w[i,j-1]+p[j]+q[j]; \]
\[ k = \text{Find}(c, r, i, j); \]
\[ c[i,j] = c[i,k-1]+c[k,j]+w[i,j]; \]
\[ r[i,j] = k; \]

\text{Write}(c[0,n], w[0,n], r[0,n]);

\textbf{Algorithm Find}(c, r, i, j)

\{
\text{min} = \infty
\text{for } m = r[i,j-1] \text{ to } r[i+1, j] \text{ do}
\text{if } (c[i,m-1] + c[m,j]) < \text{min} \text{ then}
\{
\text{min} = c[i,m-1] + c[m,j];
\text{l} = m;
\}
\text{return l;}
\}

\textbf{Time Complexity}: The computation of each } c[i, j] \text{ requires to find the minimum of } m \text{ quantities. Hence each such } c[i, j] \text{ can be computed in time } O(m). \text{ The total time for all } c[i,j]'s \text{ is } O(n-m) \ast O(m) = O(nm-m^2).

\textbf{Example}:- \text{Given } n=4 \text{ and } \{a_1, a_2, a_3, a_4\} = \{ \text{do, if, int, while} \}, \text{ } p\{1:4\} = \{3,3,1,1\} \text{ and } q\{0:4\} = \{2, 3, 1, 1, 1\}. \text{ Construct an optimal binary search tree and find its cost.}

\textbf{Solution}:- \text{Initially we have } \{W[i,j], C[i,j], r[i,j]\} = \{ q_i, 0, 0 \} \text{ which initializes the bottom row of the table.}

\text{The remaining values of the table are calculated using the below formulas}

\[ W[i,j] = \{ W[i,j-1] + p[j] + q[j] \} \text{ for } i < j \]
\[ C[i,j] = \min \{ C[i,k-1] + C[k,j] \} + W[i,j] \text{ for } i < j \]
\[ r[i,j] = k \text{ for } i < j \]

\text{After calculating all the values the table is as below}

\begin{figure}
\centering
\includegraphics[scale=0.5]{binary_search_tree.png}
\end{figure}
To compute $W_{ij}$, $C_{ij}$, $r_{ij}$ when $i = j$
Thus, $W_{00} = q_0 = 2$, $C_{00} = 0$, $r_{00} = 0$
$W_{11} = q_1 = 3$, $C_{11} = 0$, $r_{11} = 0$
$W_{22} = q_2 = 1$, $C_{22} = 0$, $r_{22} = 0$
$W_{33} = q_3 = 1$, $C_{33} = 0$, $r_{33} = 0$
$W_{44} = q_4 = 1$, $C_{44} = 0$, $r_{44} = 0$
When $i < j$
$W_{01} = \{W[0,0] + p_1 + q_1\} = (2 + 3 + 3) = 8$
$C_{01} = \min\{c[0,0] + c[1,1]\} = (0 + 0 + 8) = 8$
$R_{01} = k = 1$
$W_{12} = \{W[1,1] + p_2 + q_2\} = (3 + 3 + 1) = 7$
$C_{12} = \min\{C[1,1] + C[2,2]\} = (0 + 0 + 7) = 7$
$R_{12} = k = 2$
$W_{23} = \{W[2,2] + p_3 + q_3\} = (1 + 1 + 1) = 3$
$C_{23} = \min\{C[2,2] + C[3,3]\} = (0 + 0 + 3) = 3$
$R_{23} = k = 3$
$W_{34} = \{W[3,3] + p_4 + q_4\} = (1 + 1 + 1) = 3$
$C_{34} = \min\{C[3,3] + C[4,4]\} = (0 + 0 + 3) = 3$
$R_{34} = k = 4$
$W_{02} = \{W[0,1] + p_2 + q_2\} = (8 + 3 + 1) = 12$
$C_{02} = \min\{c[0,0] + c[1,2]\} = (0 + 0 + 12) = 12$
$R_{02} = k = 1$
$W_{13} = \{W[1,2] + p_3 + q_3\} = 9$
$C_{13} = \min\{c[1,1] + c[2,3]\} = (0 + 0 + 9) = 9$
$R_{13} = k = 2$
$W_{24} = \{W[2,3] + p_4 + q_4\} = 5$
$C_{24} = \min\{c[2,2] + c[3,4]\} = (0 + 0 + 5) = 5$
$R_{24} = k = 3$
Repeat the same procedure for the upper layers
Now consider the last cell i.e 0th row and 4th column i.e. $T_{04}$. The $r_{04}$ in this cell specifies the root i.e $r_{04} = k = 2$ then 2nd element will be the root. Then the left sub tree is $T_{i,k-1}$ i.e. $T_{01}$ and into right sub tree i.e $T_{k,j}$ i.e. $T_{24}$.

Now consider the cell which contains $r_{01}$. The $r_{01}$ specifies the root of left sub tree i.e $r_{01} = k = 1$ then 1st element will be the root of left subtree. Then the left sub tree is $T_{0,1}$ i.e. $T_{00}$. AS $i = j$ the left subtree is a leaf node and the right sub tree of $T_{01}$ is $T_{11}$ which is also a leaf node.
Now consider the cell which contains $r_{24}$. The $r_{24}$ specifies the root of right sub tree i.e. $r_{24} = k = 3$ then 3rd element will be the root of right subtree. Then the left sub tree of $T_{24}$ i.e. $T_{22}$ is a leaf node. The right sub tree of $T_{24}$ is $T_{34}$ which is a root of the right subtree.

As all nodes $T_{00}, T_{11}, T_{22}$ are leaf nodes. Now consider the cell which contains $r_{34}$. The $r_{34}$ specifies the root i.e. $r_{34} = k = 4$ then 4th element will be the root. Then the left sub tree of $T_{34}$ i.e. $T_{33}$ is a leaf node. The right sub tree of $T_{34}$ is $T_{44}$ which is a leaf node.

Representing the above tree structure with given elements we have

Representing the above tree structure with given identifiers we have
Verification: Number of possible binary search trees for 4 keys is given as.

\[
\binom{2n}{n} = \binom{2*4}{4+1} = 14
\]

Similarly we will construct 14 possible binary search trees and compute the cost of each tree. But we find tree T2 has the minimum cost of 32, so it is the optimal binary search tree.
Application-III: 0/1 knapsack problem: The 0–1 knapsack problem is posed as follows. A thief robbing a store finds n items; the ith item gives profit of pi dollars and weighs wi pounds, where pi and wi are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. Which items should he take? This is called the 0–1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.

Solving the 0/1 Knapsack problem by dynamic programming

Step 1: Generate the set S' where the set contains the possible elements (p,w) that can be added to the set.

Step 2: A recursive solution

Initially the bag contains no items

\[ S^0 = \{(0,0)\} \quad \text{for} \quad i = 0 \]

\[ S^1 = S^{i-1} + (P_i,W_i) \quad \text{- + means addition} \]

\[ S^i = S^{i-1} + S_{i-1} \quad \text{- + means merging} \]

Purging rule(Dominance Rule): If \( S^i \) has a pair (Pj,Wj) and other pair (Pk,Wk) and Pj \(<=\) Pk but Wj \(>\) Wk then the pair (Pj,Wj) is discarded from the set \( S^i \)

This process is repeated after every generation of \( S^i \) where i = 1,2,…n

Step 3: Generating the sets \( S^1, S^2, S^3 \ldots \) \( S^n \) using the above formulae.

Step 4: After generating \( S^n \) Select the element (Pk,Wk) such that Wk = capacity of the bag.

If (Pk,Wk) \( \in \) \( S^n \) and (Pk,Wk) \( \notin \) \( S^{n-1} \) then \( X_{n}=1 \) otherwise \( X_{n}=0 \).

When \( X_{n}=1 \) find another element (Pj,Wj) such that Pj=Pk and Wj=Wk-\( W_n \)

Check If (Pj,Wj) \( \in \) \( S^{n-1} \) and (Pk,Wk) \( \notin \) \( S^{n-2} \) then \( X_{n-1}=1 \) otherwise \( X_{n-1}=0 \).

This process is repeated until we find all \( X_i \) where i = 1,2,…n

Example: Consider the knapsack instance n=3, (W1,W2,W3)=(2, 3, 4), (P1,P2,P3)=(1, 2, 5) and m=6. Generate the \( S^i \) sets containing the pair (Pi,Wi) and thus find the optimal solution.

Solution: Initially \( S^0 = \{(0,0)\} \) as nothing is added to the bag.

\[ S^1 = S^0 + (P_1,W_1) = \{(0,0)\} + (1, 2) = \{(1, 2)\} \]

\[ S^2 = S^1 + (P_2,W_2) = \{(0,0), (1, 2)\} + (2, 3) = \{(2, 3), (3, 5)\} \]

\[ S^3 = S^2 + (P_3,W_3) = \{(0,0), (1, 2), (2, 3), (3, 5)\} + (5, 4) \]

\[ = \{(5,4), (6,6), (7,7), (8,9)\} \]

In \( S^3 \) has a pair (3, 5) and other pair (5,4) as 3\( <=\) 5 and 5 \(>\) 4 we discard the element (3, 5) from the set \( S^3 \) due to purging rule.

Select (6,6) from \( S^3 \) as maximum capacity of the bag is 6.

As (6,6) is an element of \( S^3 \) but it is not the element of \( S^2 \), So \( X_3=1 \).

Subtract P3,W3 from 6,6, i.e (6,6)-(5,4) = (1,2).

As (1, 2) is an element of \( S^2 \) but it is also the element of \( S^1 \), So \( X_2=0 \).

As 2nd item is not added to the bag nothing is subtracted from the element (1,2).

As (1, 2) is an element of \( S^1 \) but it is not the element of \( S^0 \), So \( X_1=1 \).

Thus (\( X_3,X_2,X_1 \)) = (1, 0, 1).

Profit obtained by placing the above items in the bag is

\[ = \sum_{i=1}^{n}(PiXi) = 1*1 + 2*0 + 5*1 = 6 \]
Algorithm DKP(p,w,n,m)
{
    S^0 = {(0,0)};
    for i = 1 to n-1 do
    {
        S_{i+1} = S_i + (P_i,W_i);
        S_i = MergePurge(S_{i+1} + S_1);
    }
    (PX,WX) = last pair in S^{n-1};
    (PY,WY) = (P^1+P_n, W^1+W_n) where W^i is the largest W in any pair in S^{n-1} such that W^1+W_n <=m;
    // Trace back for X_n, X_{n-1}, X_{n-2}......X_1;
    if (PX > PY) then
    X_n = 0;
    else
    X_n = 1;
    Trace back for X_n, X_{n-1}, X_{n-2}......X_1;
}

Time Complexity: Time complexity of 0/1 knapsack problem is O(2^{n/2}).

Application-IV: All pairs shortest path problem: All-pairs shortest-paths problem is to find a shortest path from u to v for every pair of vertices u and v. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster using the dynamic programming technique.

Solving All pairs shortest path problem by dynamic programming
Step 1: - Optimal substructure of a shortest path
Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
Step 2: - A recursive solution

\[
D^k[i,j] = \begin{cases} 
    C[i,j] & \text{for } k=0 \\
    \min \left\{ D^{k-1}[i,j], D^{k-1}[i,k]+D^{k-1}[k,j] \right\} & \text{for } k > 0 
\end{cases}
\]

Where C[i,j] is the cost matrix of the given graph.
Step 3: - Computing the distance matrices D^k where k= 1, 2, ...., n.
Step 4: - Finally D^n matrix gives the shortest distance form every vertex I to every other vertex j.
Example: - Find the shortest path between all pair of nodes in the following graph.
Solution: The cost matrix of the given graph is as follows

\[
\begin{bmatrix}
0 & 4 & 11 \\
6 & 0 & 2 \\
3 & \infty & 0
\end{bmatrix}
\]

We know \(D^0_{i,j} = C_{i,j}\)

Now we have to calculate \(D^1_{i,j}\)

\[
\begin{align*}
D^1_{1,1} &= \min\{D^0_{1,1}, D^0_{1,1} + D^0_{1,1}\} = \min\{0, 0+0\} = 0 \\
D^1_{1,2} &= \min\{D^0_{1,2}, D^0_{1,1} + D^0_{1,2}\} = \min\{4, 0+4\} = 4 \\
D^1_{1,3} &= \min\{D^0_{1,3}, D^0_{1,1} + D^0_{1,3}\} = \min\{11, 0+11\} = 11 \\
D^1_{2,1} &= \min\{D^0_{2,1}, D^0_{2,1} + D^0_{1,1}\} = \min\{6, 6+0\} = 6 \\
D^1_{2,2} &= \min\{D^0_{2,2}, D^0_{2,1} + D^0_{1,2}\} = \min\{0, 6+4\} = 0 \\
D^1_{2,3} &= \min\{D^0_{2,3}, D^0_{2,1} + D^0_{1,3}\} = \min\{2, 6+11\} = 2 \\
D^1_{3,1} &= \min\{D^0_{3,1}, D^0_{3,1} + D^0_{1,1}\} = \min\{3, 3+0\} = 3 \\
D^1_{3,2} &= \min\{D^0_{3,2}, D^0_{3,1} + D^0_{1,2}\} = \min\{\infty, 3+4\} = 7 \\
D^1_{3,3} &= \min\{D^0_{3,3}, D^0_{3,1} + D^0_{1,3}\} = \min\{0, 3+11\} = 0
\end{align*}
\]

Thus

\[
\begin{bmatrix}
0 & 4 & 11 \\
6 & 0 & 2 \\
3 & \infty & 0
\end{bmatrix}
\]

Similarly using the same procedure we get

\[
\begin{align*}
D^2_{i,j} &= \begin{bmatrix}
0 & 4 & 6 \\
6 & 0 & 2 \\
3 & 7 & 0
\end{bmatrix} \quad \text{and} \quad D^3_{i,j} = \begin{bmatrix}
0 & 4 & 6 \\
5 & 0 & 2 \\
3 & 7 & 0
\end{bmatrix}
\end{align*}
\]

As no of nodes in the given graph are 3, So \(D^3_{i,j}\) gives the shortest distance from every vertex \(i\) to every other vertex \(j\).

Algorithm AllPaths(cost, D, n)

```c
for i = 1 to n do
    for j = 1 to n do
        D[i, j] = cost[i, j];
    for k= 1 to n do
        for i = 1 to n do
            for j = 1 to n do
```
UNIT-IV

DYNAMIC PROGRAMMING

\[ D[i, j] = \min \{ D[i, j], D[i, k] + D[k, j] \}; \]

\[ D^k[i, j] = \begin{cases} C[i, j] & \text{for } k = 0 \\ \min \left\{ D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j] \right\} & \text{for } k > 0 \end{cases} \]

Time Complexity:- The time needed by AllPaths algorithm is especially easy to determine because the loop is independent of the data in the matrix \( D \). The \( D[i, j] \) is obtained after the statement is iterated \( n^3 \) times. So the time complexity of All pairs shortest paths algorithm is \( \Theta(n^3) \).

Application-V: Travelling sales person problem:- Travelling sales person problem is to find the route travelled by the salesman starting from one vertex and touching each vertex exactly once and returning back to the starting vertex. The main objective of this problem is to minimize the travelling cost of the sales person.

Solving Travelling sales person problem by dynamic programming

Step 1: - Travelling sales person problem typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

Step 2: - A recursive solution

\[ g(i, S) = \begin{cases} C[i, 1] & \text{for } S = \emptyset \\ \min_{j \in S} \left\{ C[i, j] + g(j, S - \{j\}) \right\} & \text{for } S \neq \emptyset \end{cases} \]

Where \( C[i, j] \) is the cost matrix of the given graph.

Step 3: - Computing the route until all the vertexes are added to the set \( S \).

Step 4: - Finally calculate \( g(i, S) \) where set \( S \) contains all vertexes other than the starting vertex which gives the optimal cost of travelling.

Example :- Find the shortest tour of a travelling sales person for the following graph using dynamic programming technique.

![Graph Image]

Solution:- The cost matrix of the given graph is as follows

\[
C[i, j] = \begin{bmatrix}
0 & 10 & 15 & 20 \\
5 & 0 & 9 & 10 \\
6 & 13 & 0 & 12 \\
8 & 8 & 9 & 0
\end{bmatrix}
\]
Initially Set \( S = \emptyset \) and \( g(i, S) = C[i, 1] \)

Thus \( g(1, \emptyset) = C[1, 1] = 0 \)

\( g(2, \emptyset) = C[2, 1] = 5 \)

\( g(3, \emptyset) = C[3, 1] = 6 \)

\( g(4, \emptyset) = C[4, 1] = 8 \)

Now computing \( g(i, S) \) where Set \( S \) contains a single element. As starting vertex is 1 vertex, we assume the second vertex that can be touched is 2, 3, & 4. So we calculate the cost for reaching all these vertices.

\[
g(2, \{3\}) = \min \left\{ C[i,j] + g(j, S - \{j\}) \right\} = \min \left\{ C[2,3] + g(3, S - \{3\}) \right\}
\]

\[
= 9 + 6 = 15
\]

\[
g(2, \{4\}) = \min \left\{ C[i,j] + g(j, S - \{j\}) \right\} = \min \left\{ C[2,4] + g(4, S - \{4\}) \right\}
\]

\[
= 10 + 8 = 18
\]

Similarly

\[
g(3, \{2\}) = \min \left\{ C[3,2] + g(2, S - \{2\}) \right\} = \min \left\{ C[3,2] + g(2, \emptyset) \right\}
\]

\[
= 13 + 5 = 18
\]

\[
g(3, \{4\}) = \min \left\{ C[3,4] + g(4, S - \{4\}) \right\} = \min \left\{ C[3,4] + g(4, \emptyset) \right\}
\]

\[
= 12 + 8 = 20
\]

\[
g(4, \{2\}) = \min \left\{ C[4,2] + g(2, S - \{2\}) \right\} = \min \left\{ C[4,2] + g(2, \emptyset) \right\}
\]

\[
= 8 + 5 = 13
\]

\[
g(4, \{3\}) = \min \left\{ C[4,3] + g(3, S - \{3\}) \right\} = \min \left\{ C[4,3] + g(3, \emptyset) \right\}
\]

\[
= 9 + 6 = 15
\]

Now computing \( g(i, S) \) where Set \( S \) contains a two elements.

\[
g(2, \{3,4\}) = \min \left\{ C[2,3] + g(3, S - \{3\}), C[2,4] + g(4, S - \{4\}) \right\}
\]

\[
= \min \left\{ C[2,3] + g(3, S - \{3\}), C[2,4] + g(4, S - \{4\}) \right\}
\]

\[
= \min \{9 + 20, 10 + 15 \} = 25
\]

\[
g(3, \{2,4\}) = \min \left\{ C[3,2] + g(2, S - \{2\}), C[3,4] + g(4, S - \{4\}) \right\}
\]

\[
= \min \left\{ C[3,2] + g(2, S - \{2\}), C[3,4] + g(4, S - \{4\}) \right\}
\]

\[
= \min \{9 + 18, 12 + 13 \} = 25
\]

\[
g(4, \{2,3\}) = \min \left\{ C[4,2] + g(2, S - \{2\}), C[4,3] + g(3, S - \{3\}) \right\}
\]

\[
= \min \left\{ C[4,2] + g(2, S - \{2\}), C[4,3] + g(3, S - \{3\}) \right\}
\]

\[
= \min \{8 + 15, 9 + 18 \} = 23
\]

Finally

\[
g(1, \{2, 3, 4\}) = \min \{ C[1,2] + g(2, S - \{2\}), C[1,3] + g(3, S - \{3\}), C[1,4] + g(4, S - \{4\}) \}
\]

\[
= \min \{ C[1,2] + g(2, S - \{2\}), C[1,3] + g(3, S - \{3\}), C[1,4] + g(4, S - \{4\}) \}
\]

\[
= \min \{10 + 25, 15 + 25, 20 + 22 \} = 35
\]

The optimal cost to tour through all the vertices is 35.

As from \( g(1, \{2, 3, 4\}) \) the minimum cost is obtained when \( j=2 \). So after touching 1 vertex we reach to node 2 i.e. \( 1 \rightarrow 2 \)

As from \( g(2,\{3,4\}) \) the minimum cost is obtained when \( j=4 \). So after touching 2 vertex we reach to node 4 i.e. \( 1 \rightarrow 2 \rightarrow 4 \)

The remaining vertex untouched is 3 so we reach to node 3 after touching 4th vertex i.e. \( 1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \)

As we have to return to starting vertex i.e. 1 so we reach to node 1 after touching 3rd vertex i.e. \( 1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1 \)

**Reliability design problem**: Reliability design problem is to design a system which is composed of several devices connected in series. Let \( r_i \) be the reliability of device \( D_i \) then the reliability of entire system is \( \pi r_i \).

Our problem is to use device duplication to maximize reliability under cost constraint.

**Solving Reliability design problem by dynamic programming**
**Step 1:** Reliability design problem typically rely on the property that less reliable devices are more duplicated than the more reliable devices.

**Step 2:** A recursive solution

\[ U_i = \text{Upper bound} = (C + C_i - \sum_{j=1}^{n} C_j) / C_i \]

\[ \Phi(j) = 1 - (1 - r_j)^j \text{ where } j=1,2,.., U_i \]

Generate the set \( S^1 \) where the set contains the possible elements \((r, c)\) that can be added to the system.

Initially no devices are added to the system

\[ S^0 = \{ (1,0) \} \text{ for } i = 0 \]

\[ S_i^1 = S_i^{i-1} + (\Phi(m_i), j^i c_i) \]

\[ S_i = \text{Union of } S_i^1 \text{ where } j = 1, 2, .., U_i \]

**Purging rule (Dominance Rule):** If \( S_i \) has a pair \((r_j, c_j)\) and other pair \((r_k, c_k)\) and \( r_j < r_k \) but \( c_j > c_k \) then the pair \((r_j, c_j)\) is discarded from the set \( S_i \).

This process is repeated after every generation of \( S_i \) where \( i = 1, 2, .., n \)

**Step 3:** Generating the sets \( S^1, S^2, S^3 \ldots S^n \) using the above formulae.

**Step 4:** After generating \( S^n \) Select the element \( (r_k, c_k) \) such that \( c_k = \text{cost constraint} \).

If \((r_i, c_i) \in S^n \) and \((r_k, c_k) \in S^n \) then \( D_i = j \).

Find another element \((r_j, c_j)\) such that \( r_j = r_k - \Phi(m_i) \) and \( c_j = c_k - j^i * c_n \)

Check if \((r_j, c_j) \in S^i \) and \((r_j, c_j) \in S^i \) then \( D_{n-i} = j \).

This process is repeated until we find all \( D_i \) where \( i = 1, 2, .., n \)

**Example:** Design a three stage system with device types \( D_1, D_2, D_3 \). 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 type is 0.9, 0.8, 0.5 respectively.

**Solution:** Given \( C = \$105, C_1 = \$30, C_2 = \$15, C_3 = \$20, r_1 = 0.9, r_2 = 0.8, r_3 = 0.5 \)

First compute \( U_i \) where \( i = 1, 2, .., n \)

\[ U_i = \text{Upper bound} = (C + C_i - \sum_{j=1}^{n} C_j) / C_i \]

\[ U_1 = (C + C_1 - (C_1 + C_2 + C_3)) / C_1 = (105 + 30 - 65) / 30 = 2.33 = 2 \]

\[ U_2 = (C + C_2 - (C_1 + C_2 + C_3)) / C_2 = (105 + 15 - 65) / 15 = 3.66 = 3 \]

\[ U_3 = (C + C_3 - (C_1 + C_2 + C_3)) / C_3 = (105 + 20 - 65) / 20 = 3 \]

Hence \((U_1, U_2, U_3) = (2, 3, 3)\)

Initially no devices are added to the system

\[ S^0 = \{ (1,0) \} \]

As \( U_1 = 2 \) we have to calculate \( S_i^1 \) where \( i = 1 \) and \( j = 1, 2, .., U_1 \)

\[ \Phi(1) = 1 - (1 - 1)^1 = 1 - (1 - 0.9)^1 = 0.9 \]

\[ S_1^1 = S_1^{i-1} + (\Phi(1), 1^i c_1) = \{ (1,0) \} + (0.9, 30) = \{ (0.9, 30) \} \]

\[ \Phi(2) = 1 - (1 - r_1)^2 = 1 - (1 - 0.9)^2 = 0.99 \]

\[ S_2^1 = S_1^{i-1} + (\Phi(2), 1^i c_1) = \{ (1,0) \} + (0.99, 2^i 30) = \{ (0.99, 60) \} \]

\[ S_i = \text{Union of } S_i^1 \text{ where } j = 1, 2, .., U_i \]

Thus \( S_1^1 \cup S_2^1 = \{ (0.9, 30), (0.99, 60) \} \cup \{ (0.9, 30), (0.99, 60) \} \)

As \( U_2 = 3 \) we have to calculate \( S_i^2 \) where \( i = 2 \) and \( j = 1, 2, .., U_2 \)

\[ \Phi(1) = 1 - (1 - r_2)^1 = 1 - (1 - 0.8)^1 = 0.8 \]

\[ S_2^2 = \{ (0.9, 30), (0.99, 60) \} + (0.8, 15) = \{ (0.72, 45), (0.792, 75) \} \]

\[ \Phi(2) = 1 - (1 - r_2)^2 = 1 - (1 - 0.8)^2 = 0.96 \]

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

\[ \Phi(3) = 1 - (1 - r_2)^3 = 1 - (1 - 0.8)^3 = 0.992 \]

\[ S_2^2 = \{ (0.9, 30), (0.99, 60) \} + (0.992, 3^i 15) = \{ (0.8928, 75), (0.9828, 105) \} \]

Thus \( S_2^2 \cup S_3^2 = \{ (0.72, 45), (0.792, 75) \} \cup \{ (0.864, 60), (0.9504, 90) \} \cup \{ (0.8928, 75), (0.98, 105) \} \)
Applying Purging rule (0.792, 75) is removed form \( S^2 \)

Thus

\[ S^2 = \{ (0.72, 45), (0.864, 60), (0.9504, 90), (0.8928, 75), (0.98, 105) \} \]

Applying Purging rule (0.9504, 90) is removed form \( S^2 \)

Thus

\[ S^2 = \{ (0.72, 45), (0.864, 60), (0.8928, 75), (0.98, 105) \} \]

As \( U_3 = 3 \) we have to calculate \( S^i_j \) where \( i = 2 \) and \( j = 1, 2 \ldots U_3 \)

\[ \Phi_3(1) = 1 - (1 - r_3)^1 = 1 - (1 - 0.5) = 0.5 \]

\[ S^3_1 = \{ (0.72, 45), (0.864, 60), (0.8928, 75), (0.98, 105) \} + (0.5, 20) \]

\[ = \{ (0.36, 65), (0.432, 80), (0.4464, 95) \} \] [Remaining elements are not included as cost exceeds 105]

\[ \Phi_3(2) = 1 - (1 - r_3)^2 = 1 - (1 - 0.5)^2 = 0.75 \]

\[ S^3_2 = \{ (0.72, 45), (0.864, 60), (0.8928, 75), (0.98, 105) \} + (0.75, 40) \]

\[ = \{ (0.54, 85), (0.648, 100) \} \]

\[ \Phi_3(3) = 1 - (1 - r_3)^3 = 1 - (1 - 0.5)^3 = 0.875 \]

\[ S^3_3 = \{ (0.72, 45), (0.864, 60), (0.8928, 75), (0.98, 105) \} + (0.875, 60) \]

\[ = \{ (0.63, 105) \} \]

Thus \( S^3 = S^3_1 \cup S^3_2 \cup S^3_3 = \)

\[ = \{ (0.36, 65), (0.432, 80), (0.4464, 95) \} \cup \{ (0.54, 85), (0.648, 100) \} \cup \{ (0.63, 105) \} \]

Applying Purging rule (0.4464, 95) is removed from \( S^3 \)

Thus

\[ S^3 = \{ (0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100), (0.63, 105) \} \]

Applying Purging rule (0.63, 105) is removed from \( S^3 \)

Thus \( S^3 = \{ (0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100) \} \)

After generating \( S^3 \) Select the element (0.648, 100).

If (0.648, 100) \( \in S^3 \) and (0.648, 100) \( \in S^3_2 \) then Duplication of \( D_3 = 2 \).

Now 100 – 40 = 60. Now the cost constraint for \( D_2 \) is 60. So Select the element with cost equal to 60 in \( S^2 \).

i.e. (0.864, 60)

If (0.864, 60) \( \in S^2 \) and (0.864, 60) \( \in S^3_2 \) then Duplication of \( D_2 = 2 \).

Now 60 – 30 = 30. Now the cost constraint for \( D_1 \) is 30. So Select the element with cost equal to 30 in \( S^1 \).

i.e. (0.9, 30)

If (0.9, 30) \( \in S^1 \) and (0.9, 30) \( \in S^1 \) then Duplication of \( D_1 = 1 \).

Thus \( (J_1, J_2, J_3) = (1, 2, 2) \).

<table>
<thead>
<tr>
<th>Divide &amp; Conquer</th>
<th>Dynamic Programming</th>
</tr>
</thead>
</table>
| 1. The divide-and-conquer paradigm involves three steps at each level of the recursion:  
- Divide the problem into a number of subproblems.  
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.  
- Combine the solutions to the subproblems into the solution for the original problem. | 1. The development of a dynamic-programming algorithm can be broken into a sequence of four steps:  
- Characterize the structure of an optimal solution.  
- Recursively define the value of an optimal solution.  
- Compute the value of an optimal solution in a bottom-up fashioned. Construct an optimal solution from computed information |
<table>
<thead>
<tr>
<th></th>
<th>UNIT-IV</th>
<th>DYNAMIC PROGRAMMING</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>They call themselves recursively one or more times to deal with closely related sub problems.</td>
<td>2. Dynamic Programming is not recursive.</td>
</tr>
<tr>
<td>3.</td>
<td>D&amp;C do more work on the sub-problems and hence has more time consumption.</td>
<td>3. DP solves the sub problems only once and then stores it in the table.</td>
</tr>
<tr>
<td>4.</td>
<td>In D&amp;C the sub problems are independent of each other.</td>
<td>4. In DP the sub-problems are not independent.</td>
</tr>
<tr>
<td>5.</td>
<td>Example: Merge Sort, Binary Search</td>
<td>5. Example : Matrix chain multiplication</td>
</tr>
</tbody>
</table>
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

# Compiled Books: [https://www.tutorialsduniya.com/compiled-books](https://www.tutorialsduniya.com/compiled-books)
# Programs: [https://www.tutorialsduniya.com/programs](https://www.tutorialsduniya.com/programs)
# Question Papers: [https://www.tutorialsduniya.com/question-papers](https://www.tutorialsduniya.com/question-papers)
# Python Notes: [https://www.tutorialsduniya.com/python](https://www.tutorialsduniya.com/python)
# Java Notes: [https://www.tutorialsduniya.com/java](https://www.tutorialsduniya.com/java)
# JavaScript Notes: [https://www.tutorialsduniya.com/javascript](https://www.tutorialsduniya.com/javascript)
# JSP Notes: [https://www.tutorialsduniya.com/jsp](https://www.tutorialsduniya.com/jsp)
# Microprocessor Notes: [https://www.tutorialsduniya.com/microprocessor](https://www.tutorialsduniya.com/microprocessor)
# OR Notes: [https://www.tutorialsduniya.com/operational-research](https://www.tutorialsduniya.com/operational-research)
**UNIT-V**

**BACKTRACKING**

**Introduction:**

In back tracking technique, we will solve problems in an efficient way, when compared to other methods like greedy method and dynamic programming. The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function \( P(x_1, \ldots, x_n) \). Form a solution at any point seems not promising, ignore it. All possible solutions require a set of constraints divided into two categories:

1. **Explicit Constraint:** Explicit constraints are rules that restrict each \( x_i \) to take on values only from a given set. Ex: \( x_n = 0 \) or 1.
2. **Implicit Constraint:** Implicit Constraints are rules that determine which of the tuples in the solutions space of \( T \) satisfy the criterion function.

Implicit constraints for this problem are that no two queens can be on the same diagonal.

Back tracking is a modified depth first search tree. Backtracking is a 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. State space tree exists implicitly in the algorithm because it is not actually constructed.

Terminologies which is used in this method:

1. **Solution Space:** All tuples that satisfy the explicit constraints define a possible solution space for a particular instance \( T \) of the problem.

   Example:

   ![Tree organization of a solution space](image)

   Fig: Tree organization of a solution space

2. **Problem State:** A problem state is the state that is defined by all the nodes within the tree organization.

   Example:

   ![Problem State](image)

   Fig: Problem State

3. **Solution States:** These are the problem states \( S \) for which the path form the root to \( S \) defines a tuple in the solution space.

   Here, square nodes (□) indicate solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e., \((ghk-,B,D),(A,C,F)\) and \((A,C,G)\) are the solution states.

   Example:

   ![Solution State](image)

   Fig: Solution State

4. **State Space Tree:** Is the set of paths from root node to other nodes. State space tree is the tree organization of the solution of the solution space.

   Example: State space tree of a 4-queen problem.
In the above figure nodes are numbered as in depth first search. Initially, \( x_1 = 1 \) or 2 or 3 or 4, it means we can place first queen in either first, second, third or fourth column. If \( x_1 = 1 \) then \( x_2 \) can be placed in either 2nd, 3rd or 4th column. If \( x_1 = 2 \) then \( x_2 \) can be placed either in 3rd, or 4th column. If \( x_3 = 3 \), then \( x_4 = 4 \). So nodes 1-2-3-4-5 is one solution in solution space. It may not be a feasible solution.

5. **Answer States:** These solution states \( S \), for which the path from the root node to \( S \) defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem.

![Tree organization of the 4-queens solution space](image)

Here are \( C, D \) are answer states. \((A, C)\) and \((A, C, D)\) are solution states.

6. **Live Node:** A node which has been generated but whose children have not yet been generated is live node.

**Example 1:**

This node 1 is called as live node since the children of node 1 have not been generated.

**Example 2:**

In this, node 1 is not a live node but node 2, node 3 are live nodes.
UNIT-V

BACKTRACKING

Example 3:

Here, 4, 5, 3 are live nodes because the children of these nodes not yet been generated.

7. **E-Node:** The live nodes whose children are currently being generated is called the E-node (node being expanded).

   Example 1:  

   This node 1 is live node and its children are currently being generated (expanded).

Example 2:

Here, node 2 is E-node.

8. **Dead Node:** It is generated node, that is either not to be expanded further or one for which all of its children has been generated.

Nodes 1, 2, 3, are dead nodes. Since node 1’s children generated and node 2, 3 are not expanded.

Assumed that node 2 generated one more node, So, 1, 3, 4 are dead nodes.

**General Method:**

- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e., set of all feasible solutions) for the given problem.
- Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.
Algorithm Backtrack(k)
1  // This schema describes the backtracking process using
2  // recursion. On entering, the first k - 1 values
3  // x[1], x[2], ..., x[k - 1] of the solution vector
4  // x[1 : n] have been assigned. x[] and n are global.
5  {
6        for (each x[k] \in T(x[1], ..., x[k - 1]) do
7            if (B_k(x[1], x[2], ..., x[k]) \neq 0) then
8                if (x[1], x[2], ..., x[k] is a path to an answer node)
9                    then write [x[1 : k]];
10                   if (k < n) then Backtrack(k + 1);
11                }
12  }
13
Algorithm: Recursive backtracking

Applications of Backtracking

Backtracking is an algorithm design technique that can effectively solve the larger instances of combinational problems. It follows a systematic approach for obtaining solution to a problem. The applications of backtracking include,

1) N-Queens Problem: This is generalization problem. If we take n=8 then the problem is called as 8 queens problem. If we take n=4 then the problem is called 4 queens problem. A classic combinational problem is to place n queens on a n*n chess board so that no two attack, i.e no two queens are on the same row, column or diagonal.

Algorithm of n-queens problem is given below:

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

// Returns true if a queen can be placed in $k$th row and
// $i$th 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;

4-Queens problem:
Consider a 4*4 chessboard. Let there are 4 queens. The objective is place there 4 queens on
4*4 chessboard in such a way that no two queens should be placed in the same row, same column or
diagonal position.

The explicit constraints are 4 queens are to be placed on 4*4 chessboards in 44 ways.
The implicit constraints are no two queens are in the same row column or diagonal.
Let{$x_1, x_2, x_3, x_4$} be the solution vector where $x_1$ column on which the queen $i$ is placed.

First queen is placed in first row and first column.

![Chessboard](a)

The second queen should not be in first row and second column. It should be placed in second row and in second, third or fourth column. It we place in second column, both will be in same diagonal, so place it in third column.

![Chessboard](b)

![Chessboard](c)

We are unable to place queen 3 in third row, so go back to queen 2 and place it somewhere else.

![Chessboard](d)

![Chessboard](e)
UNIT-V BACKTRACKING

Now the fourth queen should be placed in 4th row and 3rd column but there will be a diagonal attack from queen 3. So go back, remove queen 3 and place it in the next column. But it is not possible, so move back to queen 2 and remove it to next column but it is not possible. So go back to queen 1 and move it to next column.

Fig: Example of Backtrack solution to the 4-queens problem

Hence the solution of to 4-queens’s problem is $x_1=2$, $x_2=4$, $x_3=1$, $x_4=3$, i.,e first queen is placed in 2nd column, second queen is placed in 4th column and third queen is placed in first column and fourth queen is placed in third column.

Fig: Portion of the tree that is generated during Backtracking

8-queens problem

A classic combinatorial problem is to place 8 queens on a 8*8 chess board so that no two attack, i.,e no two queens are to the same row, column or diagonal.

Now, we will solve 8 queens problem by using similar procedure adapted for 4 queens problem. The algorithm of 8 queens problem can be obtained by placing $n=8$, in N queens algorithm. We observe that, for every element on the same diagonal which runs from the upper left to the lower right, each element has the same “row-column” value. Also every element on the same diagonal which goes from upper right to lower left has the same “row+column” value.
If two queens are placed at positions (i,j) and (k,l). They are on the same diagonal only if
\[ i-j=k-l \quad \ldots \ldots \ldots \ldots \ldots (1) \]
or
\[ i+j=k+l \quad \ldots \ldots \ldots \ldots \ldots (2). \]
From (1) and (2) implies
\[ j-l=i-k \]
\[ j+l=k+i \]
Two queens lie on the same diagonal iff
\[ |j-l|=|i-k| \]
But how can we determine whether more than one queen is lying on the same diagonal? To answer this question, a technique is devised. Assume that the chess board is divided into rows and columns say A:

This can be diagrammatically represented as follows

Now, assume that we had placed a queen at position (3,2).

Now, its diagonal cells includes (2,1)(4,3)(5,4)…(if we traverse from upper left to lower right). If we subtract values in these cells say 2-1=1, 4-3=1, 5-4=1, we get same values, also if we traverse from upper right to lower left say (2,3) (1,4)(4,1)….we get common values when we add the bits of these cells i.e 2+3=5, 1+4=5, 4+1=5. Hence, we say that, on traversing from upper left to lower right, if (m,n)(a,b) are the diagonal elements(of a cell) than m-n=a-b or on traversing from upper right to lower left if (m,n)(a,b) are the diagonal elements(of a cell) then m+n=a+b.

The solution of 8 queens problem can be obtained similar to the solution of 4 queens problem. X1=3, X2=6, X3=2, X4=7, X5=1, X6=4, X7=8, X8=5,

The solution can be shown as
**UNIT-V**

**BACKTRACKING**

**Time complexity:** The solution space tree of 8-queens problem contains $8^8$ tuples. After imposing implicit constraints, the size of solution space is reduced to $8!$ tuples.

The state space tree for the above solution is given

---

**2) Sum of Subsets Problem**

Given a set of n objects with weights $(w_1,w_2,\ldots,w_3)$ and a positive integer $M$. We have to find a subset $S'$ of the given set $S$, such that

- $S' \subseteq S$
- Sum of the elements of subset $S'$ is equal to $M$.

For example, if a given set $S=(1,2,3,4)$ and $M=5$, then there exists sets $S'=(3,2)$ and $S'=(1,4)$ whose sum is equal to $M$.

It can also be noted that some instance of the problem does not have any solution.

For example, if a given set $S=(1,3,5)$ and $M=7$, then no subset occurs for which the sum is equal to $M=7$.

The sum of subsets problem can be solved by using the back tracking approach. In this implicit tree is created, which is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that, the elements of the given set are arranged increasing order.
The left child of the root node indicates that, we have to include the first element and right child of the root node indicates that, we have to exclude the first element and so on for other nodes. Each node stores the sum of the partial solution element. If at any stage, the number equals to ‘M’ then the search is successful. At this time search will terminate or continues if all the possible solutions need to be obtain. The dead end in the tree occurs only when either of the two inequalities exists.

The sum of $S'$ is too large.
The sum of $S'$ is too small.
Thus we take back one step and continue the search.

**ALGORITHM:**

```plaintext
Algorithm SumOfSub(s, k, r)
1 // Find all subsets of $w[1 : n]$ that sum to $m$. The values of $x[j]$,
2 // $1 \leq j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] \cdot x[j]$
3 // and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
4 // It is assumed that $w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$.
5 {
6     // Generate left child. Note: $s + w[k] \leq m$ since $B_{k-1}$ is true.
7     $x[k] := 1$
8     if ($s + w[k] = m$) then write $(x[1 : k])$; // Subset found
9     else if ($s + w[k] + w[k + 1] \leq m$)
10         then SumOfSub($s + w[k], k + 1, r - w[k]$);
11     // Generate right child and evaluate $B_k$.
12     if (($s + r - w[k] \geq m$) and $(s + w[k + 1] \leq m$)) then
13         {
14             $x[k] := 0$
15             SumOfSub($s, k + 1, r - w[k]$);
16         }
17     }
18 }
```

Algorithm: Recursive backtracking algorithm for sum of subsets

Example:

Let $m=31$ and $w=\{7, 11, 13, 24\}$ draw a portions of state space tree.

Solution: Initially we will pass some subset (0, 1, 55). The sum of all the weights from $w$ is 55, i.e., $7+11+13+24=55$. Hence the portion of state-space tree can be

Here Solution A={1, 1, 1, 0} i.e., subset {7, 11, 13}
And Solution B={1, 0, 0, 1} i.e. subset {7, 24}
Satisfy given condition=31;

Example: Consider a set $S=\{5, 10, 12, 13, 15, 18\}$ and $N=30$. 

3) Graph Coloring

Let G be a graph and m be a given positive integer. The graph coloring problem is to find if the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorability decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.
Algorithm $m$Coloring\((k)\)

1.  
   \begin{algorithmic}[1]
     \Repeat
       \{ // Generate all legal assignments for $x[k]$.
       \State NextValue\((k)\); // Assign to $x[k]$ a legal color.
       \If\((x[k] = 0)\) \Return; // No new color possible
       \EndIf
       \If\((k = n)\) \Return; // At most $m$ colors have been
       \Else \mColoring\((k + 1)\);
       \EndIf
     \Until\((\text{false})\);
   \end{algorithmic}

Algorithm: Finding all $m$-colorings of graph

Figure. An Ex. Of graph coloring
UNIT - V

BACKTRACKING

Algorithm: Finding next color

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

Algorithm: Finding next color

Fig. A map and it’s planar graph representation

To color the above graph chromatic number is 4. And the order of coloring is X1=1, X2=2, X3=3, X4=4, X5=1

Time Complexity: At each internal node O(mn) time is spent by Nextcolor to determine the children corresponding to legal coloring. Hence the total time is bounded by,

\[ \sum_{i=1}^{n} m^n = n(m + m^2 + \ldots + m^n) \]

= \[ n \cdot m^n \]

= \[ O(n \cdot m^n) \]
4) Hamiltonian Cycle

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

Given a graph $G=(V,E)$ we have to find the Hamiltonian circuit using backtracking approach, we start our search from any arbitrary vertex, say $x$. This vertex $x$ becomes the root of our implicit tree. The next adjacent vertex is selected on the basis of alphabetical / or numerical order. If at any stage an arbitrary vertex, say $y$ makes a cycle with any vertex other than vertex $x$ then we say that dead end is reached. In this case we backtrack one step and again the search begins by selecting another vertex. It should be noted that, after backtracking the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

Example: Consider a graph $G=(V,E)$, we have to find the Hamiltonian circuit using backtracking method.

![Graph G](image)

Solution: Initially we start out search with vertex ‘1’ the vertex ‘1’ becomes the root of our implicit tree.

Next we choose vertex ‘2’ adjacent to ‘1’, as it comes first in numerical order (2, 3, 4).

Next vertex ‘3’ is selected which is adjacent to ‘2’ and which comes first in numerical order (3, 5).

Next we select vertex ‘4’ adjacent to ‘3’ which comes first in numerical order (4, 5).
Next vertex ‘5’ is selected. If we choose vertex ‘1’ then we do not get the Hamiltonian cycle.

The vertex adjacent to 5 is 2, 3, 4 but they are already visited. Thus, we get the dead end. So, we backtrack one step and remove the vertex ‘5’ from our partial solution.

The vertex adjacent to ‘4’ are 5, 3, 1 from which vertex ‘5’ has already been checked and we are left with vertex ‘1’ but by choosing vertex ‘1’ we do not get the Hamiltonian cycle. So, we again backtrack one step.

Hence we select the vertex ‘5’ adjacent to ‘3’.

The vertex adjacent to ‘5’ are (2, 3, 4) so vertex 4 is selected.
The vertex adjacent to ‘4’ are (1, 3, 5) so vertex ‘1’ is selected. Hence we get the Hamiltonian cycle as all the vertex other than the start vertex ‘1’ is visited only once, 1-2-3-5-4-1.

The final implicit tree for the Hamiltonian circuit is shown below. The number above each node indicates the order in which these nodes are visited.
Algorithm Hamiltonian($k$)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix $G[1 : n, 1 : n]$. All cycles begin at node 1.
{
  repeat
    { // Generate values for $x[k]$.
      NextValue($k$); // Assign a legal next value to $x[k]$.
      if ($x[k] = 0$) then return;
      if ($k = n$) then write ($x[1 : n]$);
      else Hamiltonian($k + 1$);
    } until (false);
}

Algorithm: Finding all Hamiltonian cycles
UNIT-V

Algorithm NextValue($k$)

1  // $x[1 : k-1]$ is a path of $k-1$ distinct vertices. If $x[k] = 0$, then
2  // no vertex has as yet been assigned to $x[k]$. After execution,
3  // $x[k]$ is assigned to the next highest numbered vertex which
4  // does not already appear in $x[1 : k-1]$ and is connected by
5  // an edge to $x[k-1]$. Otherwise $x[k] = 0$. If $k = n$, then
6  // in addition $x[k]$ is connected to $x[1]$.
7
8  {
9     repeat
10     {
11         $x[k] := (x[k] + 1) \mod (n + 1)$; // Next vertex.
12         if ($x[k] = 0$) then return;
13         if ($G[x[k-1], x[k]] \neq 0$) then
14             {
15                 for $j := 1$ to $k-1$ do if ($x[j] = x[k]$) then break;
16                     // Check for distinctness.
17                 if ($j = k$) then // if true, then the vertex is distinct.
18                     if (($k < n$) or (($k = n$) and $G[x[n], x[1]] \neq 0$))
19                         then return;
20             }
21         } until (false);
22     }

Algorithm: Generating a next vertex
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- Database Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research
Introduction:

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.
- 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 list.

**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 anode that is not be expanded or explored any further. All children of a dead node have already been expanded.

*Here we will use 3 types of search strategies:*
1. FIFO (First In First Out)
2. LIFO (Last In First Out)
3. LC (Least Cost) Search

**FIFO Branch and Bound Search:**

For this we will use a data structure called Queue. Initially Queue is empty.

Example:

Assume the node 12 is an answer node (solution)  
In FIFO search, first we will take E-node as a node 1. Next we generate the children of node 1. We will place all these live nodes in a queue.

```
2 3 4
```

Now we will delete an element from queue, i.e. node 2, next generate children of node 2 and place in this queue.

```
3 4 5 6
```
Next, delete an element from queue and take it as E-node, generate the children of node 3, 7, 8 are children of 3 and these live nodes are killed by bounding functions. So we will not include in the queue.

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Again delete an element an from queue. Take it as E-node, generate the children of 4. Node 9 is generated and killed by boundary function.

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Next, delete an element from queue. Generate children of nodes 5, i.e., nodes 10 and 11 are generated and by boundary function, last node in queue is 6. The child of node 6 is 12 and it satisfies the conditions of the problem, which is the answer node, so search terminates.

**LIFO Branch and Bound Search**

For this we will use a data structure called stack. Initially stack is empty.

**Example:**

Generate children of node 1 and place these live nodes into stack.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

Remove element from stack and generate the children of it, place those nodes into stack. 2 is removed from stack. The children of 2 are 5, 6. The content of stack is,

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

stack
Again remove an element from stack, i.e node 5 is removed and nodes generated by 5 are 10, 11 which are killed by bounded function, so we will not place 10, 11 into stack.

Delete an element from stack, i.e node 6. Generate child of node 6, i.e 12, which is the answer node, so search process terminates.

**LC (Least Cost) Branch and Bound Search**

In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.

In this we will use ranking function or cost function. We generate the children of E-node, among these live nodes; we select a node which has minimum cost. By using ranking function we will calculate the cost of each node.

Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate the cost of 2, 3, 4 nodes is ĉ =2, ĉ =3, ĉ =4 respectively. Now we will select a node which has minimum cost i.e node 2. For node 2, the children are 5, 6. Between 5 and 6 we will select the node 6 since its cost minimum. Generate children of node 6 i.e 12 and 13. We will select node 12 since its cost (ĉ =1) is minimum. More over 12 is the answer node. So, we terminate search process.

**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 sub tree with root \( x \). Thus, \( c(t) \) is the cost of a minimum-cost answer node in \( t \).

LC search uses \( \hat{c} \) to find an answer node. The algorithm uses two functions.
UNIT-V1

BRANCH AND BOUND

1. Least-cost()
2. Add_node()

Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add_node() to delete and add a live node from or to the list of live nodes.

Add_node(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

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.
- A good bounding helps to prune (reduce) efficiently the tree, leading to a faster exploration of the solution space. Each time a new answer node is found, the value of upper can be updated.
- Branch and bound algorithms are used for optimization problem 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.

APPLICATION: 0/1 KNAPSACK PROBLEM (LCBB)

There are n objects given and capacity of knapsack is M. Select some objects to fill the knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is maximization problem. It means we will always seek for maximum \( p_i x_i \) (where \( p_i \) represents profit of object \( x_i \)).

A branch bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch bound deals only the minimization problems. We modify the knapsack problem to the minimization problem. The modifies problem is,

\[
\text{minimize} \quad - \sum_{i=1}^{n} p_i x_i \\
\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq m \\
x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n
\]
**Algorithm** Reduce\( (p, w, n, m, I1, I2) \)

```
// Variables are as described in the discussion.
// \( p[i]/w[i] \geq p[i + 1]/w[i + 1], \ 1 \leq i < n. \)
```

\[
\begin{align*}
I1 & := I2 := \emptyset; \\
q & := \text{Lbb}(\emptyset, \emptyset); \\
k & := \text{largest } j \text{ such that } w[1] + \cdots + w[j] < m; \\
\text{for } i := 1 \text{ to } k \text{ do} \\
& \{ \\
& \quad \text{if } (\text{Ubb}(\emptyset, \{i}\}) < q \text{ then } I1 := I1 \cup \{i\}; \\
& \quad \text{else if } (\text{Lbb}(\emptyset, \{i}\}) > q \text{ then } q := \text{Lbb}(\emptyset, \{i\}); \\
& \} \\
\text{for } i := k + 1 \text{ to } n \text{ do} \\
& \{ \\
& \quad \text{if } (\text{Ubb}(\{i\}, \emptyset) < q \text{ then } I2 := I2 \cup \{i\}; \\
& \quad \text{else if } (\text{Lbb}(\{i\}, \emptyset) > q \text{ then } q := \text{Lbb}(\{i\}, \emptyset); \\
& \}
\end{align*}
```

**Algorithm: KNAPSACK PROBLEM**

**Example:** Consider the instance \( M=15, \ n=4, \ (p_1, p_2, p_3, p_4) = 10, 10, 12, 18 \) and \( (w_1, w_2, w_3, w_4) = (2, 4, 6, 9) \).

**Solution:** 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.

Arrange the item profits and weights with respect of profit by weight ratio. After that, place the first item in the 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 fraction 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+ (3/9*18)=32+6=38

Knapsack is maximization problem but branch 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 minimized 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
UNIT-V1
BRANCH AND BOUND

For node 2, \( x_1 = 1 \), means we should place first item in the knapsack.
\[ U = 10 + 10 + 12 = 32, \text{ make it as } -32 \]
\[ L = 10 + 10 + 12 + \left( \frac{3}{9} \times 18 \right) = 32 + 6 = 38, \text{ we make it as } -38 \]
For node 3, \( x_1 = 0 \), means we should not place first item in the knapsack.
\[ U = 10 + 12 = 22, \text{ make it as } -22 \]
\[ L = 10 + 12 + \left( \frac{5}{9} \times 18 \right) = 10 + 12 + 10 = 32, \text{ we make it as } -32 \]

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 6 and 7. Calculate difference of lower and upper bound of nodes 6 and 7.

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

Choose node 7, since it has minimum difference value of 0.
UNIT-V1

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 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:
\[
\sum p_i x_i = 10*1 + 10*1 + 12*0 + 18*1
\]
\[= 10 + 10 + 18 = 38.\]

**FIFO Branch-and-Bound Solution**

Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in above Example. Initially the root node, node 1 of following Figure, is the E-node and the queue of live nodes is empty. Since this is not a solution node, upper is initialized to \( u(1) = -32 \). We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of upper remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue.

Node 3, the next-node, is expanded. Its children nodes are generated; Node 6 gets added to the queue. Node 7 is immediately killed as \( L(7) > \text{upper} \). Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then Upper is updated to \( u(9) = -38 \), Nodes 5 and 6 are the next two nodes to become B-nodes. Neither is expanded as for each, \( L > \text{upper} \). Node 8 is the next E-node. Nodes 10 and 11 are generated; Node 10 is infeasible and so killed. Node 11 has \( L(11) > \text{upper} \) and so is also killed. Node 9 is expanded next.

When node 12 is generated, 'Upper and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as \( L(13) > \text{upper} \). The only remaining live node is node 12. It has no children and the search terminates. The value of upper and the path from node 12 to the root is output. So solution is \( X_1 = 1, X_2 = 1, X_3 = 0, X_4 = 1 \).
UNIT-V1

BRANCH AND BOUND

APPLICATION: TRAVELLING SALES PERSON PROBLEM

Let G = (V', E) be a directed graph defining an instance of the traveling salesperson problem. Let C_{ij} equal the cost of edge (i, j), C_{ij} = \infty if (i, j) \notin E, and let \#V = n, without loss of generality, we can assume that every tour starts and ends at vertex 1.

Procedure for solving travelling sales person problem

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. This can be done as follows:

   **Row Reduction:**
   a) 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:**
   d) 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.
   e) Find the sum of elements, which were subtracted from columns.
   f) 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 \( \alpha \) as upper bound.
2. Calculate the reduced cost matrix for every node.
   a) If path (i,j) is considered then change all entries in row i and column j of A to $\alpha$.
   b) Set $A(j,1)$ to $\alpha$.
   c) Apply row reduction and column reduction except for rows and columns containing only $\alpha$. Let $r$ is the total amount subtracted to reduce the matrix.
   d) Find $\hat{c}(S) = \hat{c}(R) + A(i,j) + r$.

Repeat step 2 until all nodes are visited.

Example: Find the LC branch and bound solution for the travelling sales 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 =

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

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 2nd column.

The resulting column wise reduced cost matrix ($A$) =

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

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 cost matrix.
The lower bound for node is 25 and upper bound is $\infty$.

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), (1,5).

The tree organization up to this as follows;
Variable i indicate the next node to visit.
Step 2:

Now consider the path (1, 2)

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

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

Apply row and column reduction for the rows and columns whose rows and column are not completely $\infty$. 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 $\hat{c}(S) = \hat{c}(R) + A(1,2) + r \Rightarrow \hat{c}(S) = 25 + 10 + 0 = 35$.

Now consider the path (1, 3)

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

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

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

\[
\begin{bmatrix}
\infty & \infty & \infty & \infty & \infty \\
1 & \infty & \infty & 2 & 0 \\
15 & \infty & 3 & \infty & 0 \\
11 & 0 & \infty & 12 & \infty \\
\end{bmatrix}
\]

Then the resultant matrix is

\[
\begin{bmatrix}
\infty & \infty & \infty & \infty & \infty \\
1 & \infty & \infty & 2 & 0 \\
15 & 3 & \infty & \infty & 0 \\
11 & 0 & \infty & 12 & \infty \\
\end{bmatrix}
\]

Row reduction sum = 0

Column reduction sum = 11

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

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

Now consider the path (1, 4)
UNIT-V1  

BRANCH AND BOUND

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty & \\
12 & \infty & 11 & 2 & 0 & \\
0 & 3 & \infty & 0 & \infty & \\
\infty & 3 & 12 & \infty & 0 & \\
11 & 0 & 0 & \infty & \infty & \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty & \\
12 & \infty & 11 & 2 & 0 & \\
0 & 3 & \infty & 0 & \infty & \\
\infty & 3 & 12 & \infty & 0 & \\
11 & 0 & 0 & \infty & \infty & \\
\end{array}
\]

Then the resultant matrix is =

\[
\begin{array}{cccccc}
0 & 3 & \infty & \infty & 2 & \\
\infty & 3 & 12 & \infty & 0 & \\
11 & 0 & 0 & \infty & \infty & \\
\end{array}
\]

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,4)+r
ĉ(S)= 25 + 0 +0 = 25.

**Now Consider the path (1, 5)**

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty & \\
12 & \infty & 11 & 2 & \infty & \\
0 & 3 & \infty & 0 & \infty & \\
15 & 3 & 12 & \infty & \infty & \\
\infty & 0 & 0 & 12 & \infty & \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty & \\
10 & \infty & 9 & 0 & \infty & \\
0 & 3 & \infty & 0 & \infty & \\
\infty & 0 & 0 & 12 & \infty & \\
\end{array}
\]

Then the resultant matrix is =

\[
\begin{array}{cccccc}
0 & 3 & \infty & \infty & 0 & \\
12 & 0 & 9 & \infty & \infty & \\
\infty & 0 & 0 & 12 & \infty & \\
\end{array}
\]

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction(r) = 5 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,5)+r
ĉ(S)= 25 + 1 +5 = 31.

*The tree organization up to this as follows:

```
25
 /   
/     
35 2  53 3  4 25  5 31
```

Numbers outside the node are ĉ values
The cost of the between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25, (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.

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
12 & 11 & \infty & 0 \\
0 & 3 & \infty & \infty & 2 \\
\infty & 3 & 12 & \infty & 0 \\
11 & 0 & 0 & \infty & \infty \\
\end{array}
\]

A

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

*Now consider the path (4, 2)*

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & 11 & 0 \\
0 & \infty & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
11 & \infty & 0 & \infty & \infty \\
\end{array}
\]

Apply row and column reduction for the rows and columns whose rows and column are not completely \(\infty\)

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & 11 & 0 \\
0 & \infty & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
11 & \infty & 0 & \infty & \infty \\
\end{array}
\]

Then the resultant matrix is

\[
\begin{array}{cccccc}
0 & \infty & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
11 & \infty & 0 & \infty & \infty \\
\end{array}
\]

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as \(c(S)=c(R)+A(4,2)+r\)
\[c(S)= 25 + 3 +0 = 28.\]

*Now consider the path (4, 3)*

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
12 & \infty & \infty & \infty & 0 \\
\infty & 3 & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
11 & 0 & \infty & \infty & \infty \\
\end{array}
\]

Apply row and column reduction for the rows and columns whose rows and column are not completely \(\infty\)

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
1 & \infty & \infty & \infty & 0 \\
\infty & \infty & \infty & \infty & \infty \\
0 & 0 & \infty & \infty & \infty \\
\end{array}
\]

Then the resultant matrix is

\[
\begin{array}{cccccc}
0 & \infty & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
0 & 0 & \infty & \infty & \infty \\
\end{array}
\]

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction(r) = 2 +11=13
Therefore, as \( \hat{c}(S) = \hat{c}(R) + A(4,3) + r \)
\[ \hat{c}(S) = 25 + 12 + 13 = 50. \]

*Now consider the path (4, 5)*
Change all entries of row 4 and column 5 of A to \( \infty \) and also set \( A(5,1) \) to \( \infty \).

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
12 & \infty & 11 & \infty & \infty \\
0 & 3 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 0 & \infty & \infty \\
\end{array}
\]

Apply row and column reduction for the rows and columns whose rows and column are not completely \( \infty \)

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
1 & \infty & 0 & \infty & \infty \\
0 & 3 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 0 & \infty & \infty \\
\end{array}
\]

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 as follows:*

![Tree Diagram]

**Numbers outside the node are \( \hat{c} \) values**

The cost of the between (4, 2) = 28, (4, 3) = 50, (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.

\[
A = \begin{bmatrix}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & 11 & \infty & 0 \\
0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
11 & \infty & 0 & \infty & \infty \\
\end{bmatrix}
\]

The new possible paths are (2, 3) and (2, 5).
Now Consider the path (2, 3):
Change all entries of row 2 and column 3 of A to ∞ and also set A(3,1) to ∞.

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 2 \\
\infty & \infty & \infty & \infty & \infty \\
11 & \infty & \infty & \infty & \infty \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

Then the resultant matrix is

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

Row reduction sum =13
Column reduction sum = 0
Cumulative reduction(r) = 13 +0=13
Therefore, as \( \hat{c}(S) = \hat{c}(R)+A(2,3)+r \)
\[
\hat{c}(S) = 28 + 11 +13 = 52.
\]

Now Consider the path (2, 5):
Change all entries of row 2 and column 5 of A to ∞ and also set A(5,1) to ∞.

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

Then the resultant matrix is

\[
\begin{array}{cccccc}
0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
\end{array}
\]

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as \( \hat{c}(S) = \hat{c}(R)+A(2,5)+r \) \( \Rightarrow \)
\[
\hat{c}(S) = 28 + 0 +0 = 28.
\]

The tree organization up to this as follows:
The cost of the 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.

The new possible path is (5, 3).

Now 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 column are not completely ∞

Then the resultant matrix is =

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(5,3)+r
ĉ(S)= 28 + 0 +0 = 28.
The path travelling sales person problem is:
1→4→2→5→3→1:
The minimum cost of the path is: 10+2+6+7+3=28.
The overall tree organization is as follows:

Numbers outside the node are $c$ values.
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research