Programs Offered

Post Graduate Programmes (PG)

- Master of Business Administration
- Master of Computer Applications
- Master of Commerce (Financial Management / Financial Technology)
- Master of Arts (Journalism and Mass Communication)
- Master of Arts (Economics)
- Master of Arts (Public Policy and Governance)
- Master of Social Work
- Master of Arts (English)
- Master of Science (Information Technology) (ODL)
- Master of Science (Environmental Science) (ODL)

Diploma Programmes

- Post Graduate Diploma (Management)
- Post Graduate Diploma (Logistics)
- Post Graduate Diploma (Machine Learning and Artificial Intelligence)
- Post Graduate Diploma (Data Science)

Undergraduate Programmes (UG)

- Bachelor of Business Administration
- Bachelor of Computer Applications
- Bachelor of Commerce
- Bachelor of Arts (Journalism and Mass Communication)
- Bachelor of Arts (General / Political Science / Economics / English / Sociology)
- Bachelor of Social Work
- Bachelor of Science (Information Technology) (ODL)

AMITY UNIVERSITY ONLINE
Amity Helpline: (Toll free) 18001023434
For Student Support: +91 - 8826334455
Support Email id: studentsupport@amityonline.com | https://amityonline.com

Product code
81030112000628

AMITY

Data Structures and Algorithm Design

Data Structures and Algorithm Design

Amity University Online

AMITY UNIVERSITY ONLINE

Data Structures and Algorithm Design

AMITY UNIVERSITY ONLINE

Published by Amity University Press for exclusive use of Amity Directorate of Distance and Online Education, Amity University, Noida-201313

Contents

Page No.

Module -I: Overview of Data Structures 01

1.1 Big Data Structures
- 1.1.1 Stack
- 1.1.2 Evaluation of Postfix Expression
- 1.1.3 Tower of Hanoi Problem
- 1.1.4 Queue
- 1.1.5 Linked List
- 1.1.6 Double Linked List

1.2 Algorithm
- 1.2.1 Algorithm and Characteristics
- 1.2.2 Asymtotic Notations
- 1.2.3 Algorithm Time Complexity
- 1.2.4 Recursive Program Analysis Strategies
- 1.2.5 Master Theorem

1.3 Divide and Conquer Paradigm
- 1.3.1 Divide-Conquer Recurrence Equations Solutions

1.4 Sorting Techniques
- 1.4.1 Bubble Sort
- 1.4.2 Insertion Sort
- 1.4.3 Selection Sort
- 1.4.4 Merge Sort
- 1.4.5 Heap Sort
- 1.4.6 Quick Sort

Module - II: Tree and Graphs 55

2.1 Tree
- 2.1.1 Basic Terminology and Applications
- 2.1.2 Binary Trees
- 2.1.3 Binary Search Trees
- 2.1.4 Optimal Binary Search Tree
- 2.1.5 Red-Black Trees
- 2.1.6 AVL Trees
- 2.1.7 B Trees
- 2.1.8 Spanning Trees
- 2.1.9 Prim's Algorithm
- 2.1.10 Kruskal's Algorithm

2.2 Graphs
- 2.2.1 Terminology, Representations, Traversals
- 2.2.2 Depth-First Search (DFS)

2.2.3 Breadth-First Search (BFS)
2.2.4 Single-Source Shortest Path
2.2.5 All-Pair Shortest Path

Module - III: Dynamic and Greedy Approach **107**

3.1 Dynamic Programming
3.1.1 Dynamic Programming Approach
3.1.2 Principle of Optimality
3.1.3 Strassens Matrix Multiplication
3.1.4 Matrix Chain Multiplication
3.1.5 Longest Common subsequence
3.1.6 0/1 Knapsack Problem
3.2 Greedy Approach
3.2.1 Introduction of Greedy Approach
3.2.2 Activity Selection Problem
3.2.3 Huffman Codes
3.2.4 Fractional Knapsack Problem
3.3 Branch and Bound Approach
3.3.1 Traveling Salesman Problem
3.3.2 Knapsack Problem
3.3.3 Job Sequencing
3.3.4 Job Sequencing with Deadline

Module - IV: E String Matching Algorithms and Approximation Algorithms **141**

4.1 String Matching Algorithm
4.1.1 Naïve String Matching Algorithm
4.1,2 String Searching Algorithm: Knuth Morris and Pratt
4.1.3 String Matching Algorithm: Rabin Karp
4.2 Approximation Algorithm
4.2.1 Vertex-Cover Algorithm
4.2.2 Set Covering Problem
4.2.3 Randomisation and Linear Programming
4.2.4 The Subset-sum Problem

Module -V: NP-Completeness **172**

5.1 Polynomial Time
5.1.1 Green Technology
5.1.2 Hamiltonian Cycles
5.2 NP-Completeness
5.2.1 NP-completeness Proofs
5.2.2 NP-completeness Problems
5.2.3 NP-Hard and SAT Problems
5.2.4 NP-completeness and Reducibility

Module - I: Overview of Data Structures

Learning Objectives:

At the end of this module, you will be able to:

- Understand the characteristics and applications of big data structures.
- Identify different types of big data structures and their use cases.
- Differentiate between various algorithm design techniques.
- Implement divide and conquer algorithms for common computational problems.
- Compare the efficiency of different sorting algorithms.
- Implement and apply sorting algorithms to organise data effectively.

Introduction

A data structure is a method of organising and storing datain on a computer, enabling efficient access and usage. It defines the relationship between data elements and the operations that can be performed on them.

1.1 Big Data Structures

Importance of Data Structures

Data structures are crucial for several reasons:

- Efficient Data Management: They allow for efficient storage and retrieval of data, reducing processing time and enhancing performance.
- Data Organization: They logically organise data, making it easier to comprehend and access.
- Data Abstraction: They conceal the implementation details of data storage, letting programmers focus on logical data manipulation.
- Reusability: Common data structures can be reused across multiple applications, saving development time and effort.
- Algorithm Optimization: Choosing the right data structure can greatly improve the efficiency of algorithms operating on the data.

Classification of Data Structures

Data structures are broadly classified into two categories:

- Linear Data Structures: These store data in a sequential order, allowing easy insertion and deletion operations. Examples include arrays, linked lists, and queues.
- Non-Linear Data Structures: These store data in a hierarchical or interconnected manner, facilitating more complex relationships between data elements. Examples include trees, graphs, and hash tables.

Types of Data Structures

Data structures are divided into two primary categories:

- Linear Data Structures:
  - ❖ Array: A collection of elements of the same type stored in contiguous memory locations.

Notes

- ❖ Linked List: A collection of elements linked together by pointers, allowing dynamic insertion and deletion.
- ❖ Queue: A First-In-First-Out (FIFO) structure where elements are added at the end and removed from the beginning.
- ❖ Stack: A Last-In-First-Out (LIFO) structure where elements are added and removed from the top.

- Non-Linear Data Structures:
  - ❖ Tree: A hierarchical structure where each node can have multiple child nodes.
  - ❖ Graph: A collection of nodes connected by edges, representing relationships between data elements.
  - ❖ Hash Table: A data structure that uses a hash function to map keys to values, allowing for fast lookup and insertion.

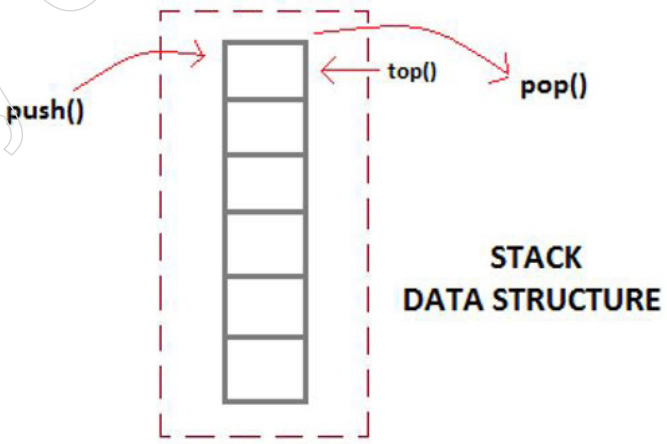
Applications of Data Structures

Data structures are widely used in various fields, including:

- Database Management Systems: To store and manage large amounts of structured data.
- Operating Systems: To manage memory, processes, and files.
- Compiler Design: To represent source code and intermediate code.
- Artificial Intelligence: To represent knowledge and perform reasoning.
- Graphics and Multimedia: To store and process images, videos, and audio data.

1.1.1 Stack

A stack is an abstract data type with limited, predefined capabilities. It is a fundamental data structure that allows elements to be added and removed in a specific order. Whenever an element is added, it goes to the top of the stack, and only the element at the top can be removed, similar to a stack of objects.



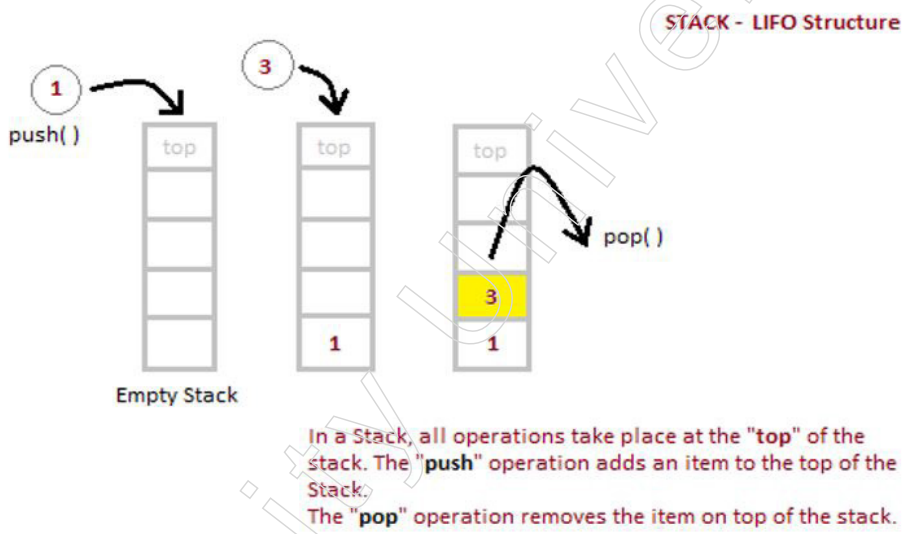
1. Stack's Basic Characteristics
   - ❖ A stack is an ordered list of similar types of data.
   - ❖ It follows the LIFO (Last In First Out) or FILO (First In Last Out) principle.
   - ❖ The function push() is used to insert new elements into the stack.
   - ❖ The function pop() is used to delete elements from the stack.

Notes

- Both insertion and deletion operations are allowed only at one end of the stack, called the top.
- When the stack is full, it is said to be in an overflow state.
- When the stack is empty, it is said to be in an underflow state.

Stack's Applications

Reversing a term is a simple application of a stack. You push a word into a stack, letter by letter. Then, you pop the letters out of the stack to reverse the word.

1. Other uses of stacks include:
   - Parsing expressions.
   - Converting expressions between different notations (e.g., Infix to Postfix, Postfix to Prefix).
2. Implementation of Stack Data Structure
   - Stacks can be implemented using either an Array or a Linked List.
   - Arrays are fast but have a limited size.
   - Linked Lists have overhead but are not limited in size.
   - Here, we'll implement a Stack using an array.
3. 

4. Fundamental Activities
   - Fundamental stack operations include initialization, usage, and de-initialization.
   - The two primary operations are:
     - Push(): Adding an element to the stack.
     - Pop(): Removing (accessing) an element from the stack.
   - Additional functionality for verifying the stack's state includes:
     - Peek(): Retrieving the top element of the stack without removing it.
     - IsFull(): Checking if the stack is full.
     - IsEmpty(): Checking if the stack is empty.

Notes

- A pointer is maintained to the last pushed data on the stack, which represents the top of the stack. This pointer is referred to as the “top” pointer, providing access to the top value without removing it.

1.1.2 Evaluation of Postfix Expression

The virtue of postfix notation is that it enables easy evaluation of expressions. To begin with, the need for parentheses is eliminated. Secondly, the priority of the operators is no longer relevant. The expression can be evaluated by making a left to right scan, stacking operands, and evaluating operators using as operands the correct elements from the stack and finally placing the result onto the stack. This evaluation is much simpler than attempting a direct evaluation of infix notation. Let us now see a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

#define MAX 50
struct postfix
{
int stack[MAX]
int top, nn;
char *s;
};
void initpostfix (struct postfix *);
void setexpr (struct postfix *, char *);
void push (struct postfix *, int);
int pop (struct postfix *);
void calculate (struct postfix *);
void show (struct postfix );
void main()
{
struct postfix q;
char expr[MAX];
clrscr():
initpostfix (&q);
printf ("\n Enter postfix expression to be evaluated: ");
gets (expr);
```

Notes

```
setexpr (&q, expr);
calculate (&q);
show (q);

getch();
}
/* initializes structure elements */
void initpostfix (struct postfix *p)
{
p->top = -1;
}

/* sets s to point to the given expr. */
void setexpr (struct postfix *p, char *str)
{
p->s = str;
}
/* adds digit to the stack */
void push (struct postfix *p, int item)
{
if(p-> top == MAX-1)
printf ("\nStack is full.");

else
{
 p> top++;
 p-> stack[p-> top] = item;
}
/* pops digit from the stack */
int pop (struct postfix *p)
{
int data;
if(p-> top ==-1)
{
printf ("\n Stack is empty.");
return NULL;
}
data = p-> stack[p-> top];
p-> top--;

  return data;
```

Notes

```
}
/* evaluates the postfix expression */
void calculate( struct postfix *p)
{
int n1, n2, n3;
while (*(p->s))
{
/* skip whitespace, if any */
if(*(p->s) == ' ' || *(p->s) == '\t')
{
p->s++;
continue;
}
/*if digit is encountered */
if (isdigit (*(p->s )))
{
p->nn = *(p->s)-'0';
push (p, p-> nn);
}
else
{
/*if operator is encountered */
n1 = pop(p);
n2 = pop(p);
switch (*(p->s))
{
case '+':
n3 = n2+n1;
break;
case '-':
n3 = n2-n1:
break;
case '/':
n3 = n2/n1:
break;
case '*':
n3 = n2 * n1;
break;
case '%:
```

Notes

```
n3 = n2 % n1;
break;
case '$':
n3 = pow(n2 , n1);
break;
default:
printf ("Unknown operator");
exit(1);
}
push(p, n3);
}
P -> s++;
}
}
/* displays the result*/
void show (struct postfix p)
{
p.nn = pop (&p);
printf ("Result is: %d", p.nn);
}
```

Output:

Enter postfix expression to be evaluated 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Result is: 46

In this program the structure postfix contains an integer array stack, to store the intermediate results of the operations and top to store the position of the topmost element in the stack. The evaluation of the expression gets performed in the calculate() function.

During the course of execution, the user enters an arithmetic expression in the postfix form. In the calculate() function, this expression would get scanned character by character. If the character scanned is an operand, then first it is converted to a digit form (from string form), and then it is pushed onto the stack. If the character scanned is a blank space, then it is skipped. If the character scanned is an operator, then the top two elements from the stack are retrieved.

An arithmetic operation is performed between the two operands. The type of arithmetic operation i.e., addition, subtraction or multiplication etc. depends on the operator scanned from the sting s. The result is then pushed back onto the stack. These steps are repeated as long as the strings is are not exhausted. The show() function displays the final result. These steps can be better understood if you go through the evaluation of a sample postfix expression shown in the below Table.

Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

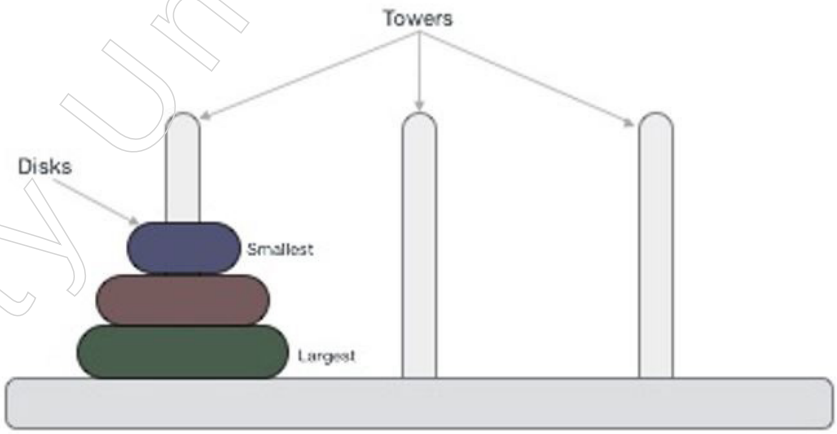
Notes

Char. Scanned	Stack Contents
4	4
2	4 , 2
$	16
3	16 , 3
*	48
3	48 , 3
-	45
8	45 , 8
4	45 , 8 , 4
/	45, 2
1	45, 2, 1
1	45, 2, 1, 1
+	45, 2, 2
/	45 , 1
+	46 (Results)

Evaluation of Postfix expression

1.1.3 Towers of Hanoi Problem

The Tower of Hanoi is a mathematical puzzle which consists of three towers (pegs) and more than one ring. At first, every one of the circles are set on one rod or tower, one over the other in a rising request of size like a cone-shaped tower. It is depicted as –



These rings are of various sizes and stacked upon in a climbing request, for example the more modest one sits over the bigger one. There are different varieties of the riddle where the quantity of disks increment, yet the tower count remains as before.

Rules

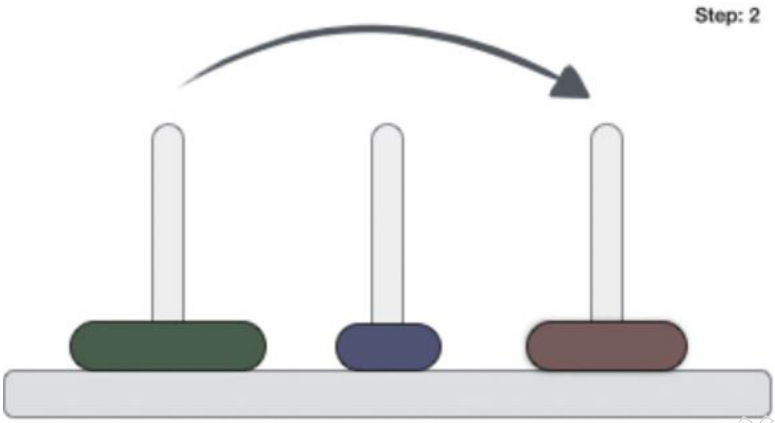
The mission is to move every one of the disks to some other tower without disregarding the succession plan. A couple of rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Just the “top” disk can be taken out.

Notes

- No large disk can sit over a little disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



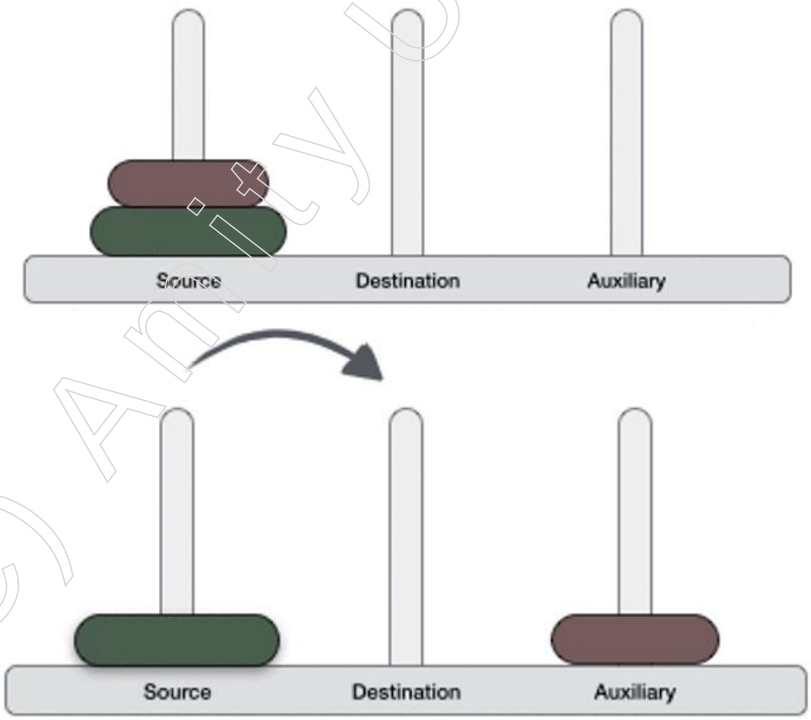
The Tower of Hanoi puzzle with n disks can be solved in a minimum of 2n−1 steps. This presentation shows that a puzzle with 3 disks has taken 23 - 1 = 7 steps.

Algorithm

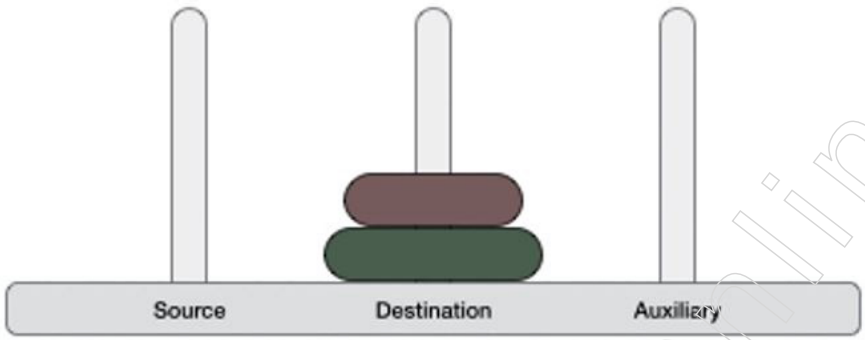
To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser number of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to the destination peg.
- And finally, we move the smaller disk from aux to destination peg.



Notes



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks into two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine applying the same in a recursive way for all given sets of disks.

The steps to follow are –

- Step 1 − Move n-1 disks from source to aux
- Step 2 − Move nth disk from source to dest
- Step 3 − Move n-1 disks from aux to dest

```
   A recursive calculation for Tower of Hanoi can be driven
as follows −
   START
   Procedure Hanoi(disk, source, dest, aux)
   IF disk == 1, THEN
   move disk from source to dest
   ELSE
   Hanoi(disk - 1, source, aux, dest)     // Step 1
   move disk from source to dest          // Step 2
   Hanoi(disk - 1, aux, dest, source)     // Step 3
   END IF
   END Procedure
   STOP.
```

1.1.4 Queue

Like stack data structure, Queue is either an abstract data form or a linear data structure in which the first element is added from one end called REAR (also called the tail) and the existing element is removed from the other end called FRONT (also called the head).

This generates a queue as the data structure of FIFO(First in First Out), which ensures that the variable inserted first will be removed first.

Notes

This is exactly how the queue structure in the real-world functions. If you go to the movie ticket counter to purchase movie tickets, and you're in the queue first, you're going to be the first one to get the tickets. Huh? Right? The same is the case for the structure of queue data. The data that is entered first will first exit the queue.

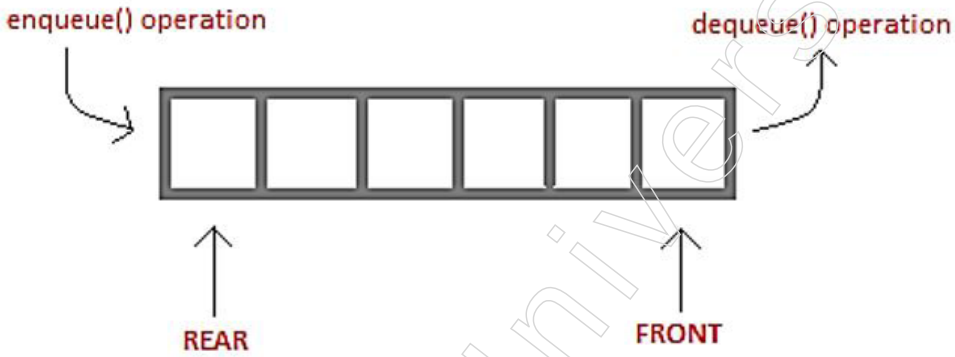
The process by which an element is inserted into the queue is called Enqueue, and the process by which an element is removed from the queue is called Dequeue.

Fig below shows the Queue Operation:

We will try to explain the fundamental operations associated with queues here.

Enqueue() − to add an object (store) to the queue.

Dequeue() − delete an object from the queue (access).



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

1. Queue's Basic Features
    1. The queue is often an ordered list of elements of related data types, like a stack.
    2. A FIFO(First In First Out) system is a queue.
    3. Once a new element is inserted in the queue, it is important to remove all the elements inserted in the queue before the new element is inserted, to delete the new element.
    4. The peek() function is often used to return, without dequeuing, the value of the first element.
2. Queue Applications

    Queue, as the name indicates, is used if we need to treat any group of objects in an order in which the first one comes in, while the others wait for their turn, also gets out first, as in the following scenarios:

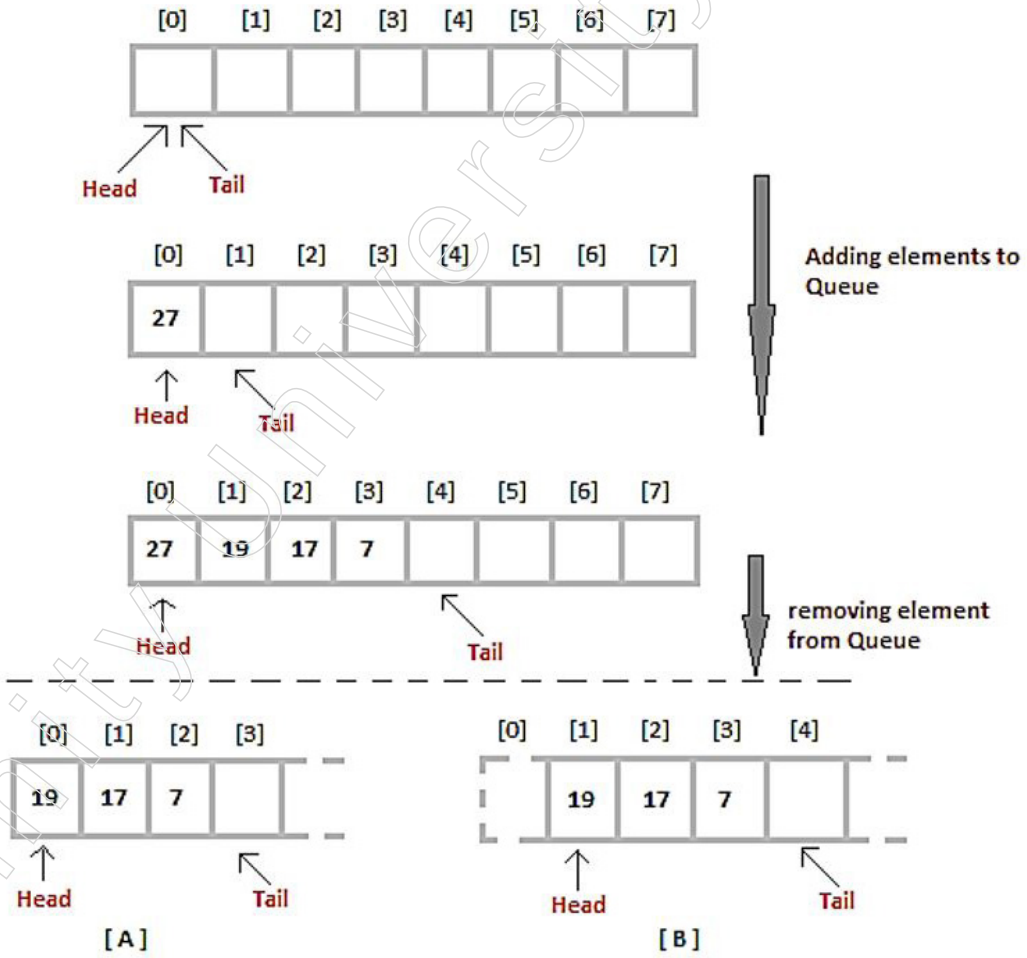
Notes

1. Serving demands, such as a printer, CPU task scheduling, etc., on a single shared resource
2. Call Center telephone networks use lines in real-life situations to keep people calling them in order until a service representative is ready.
3. Interrupt management of real-time applications. The interrupts are treated in the same order, i.e., first come first served, as they arrive.

3. Queue Data Structure Implementation

An Array, Stack, or Linked List may be used to enforce the queue. Using an array is the best way of implementing a queue.

At the first index of the array, initially the head(FRONT) and the tail(REAR) of the queue points (starting the index of an array from 0). The tail keeps going forward as we add elements to the queue, always pointing to the place where the next element will be added, while the head stays on the first index.



We can follow two possible approaches when we remove an element from the queue (mentioned [A] and [B] in the above diagram). We delete the element at the head position in the [A] approach and then transfer all the other elements one by one into the forward position.

We remove the element from the head position in Approach[B] and then move the head to the next position.

Notes

There is an overhead in approach[A] of moving the elements one position forward each time we remove the first element.

There is no such overhead in approach [B], but whenever we move head one position ahead, after removing the first variable, each time the size of the queue is reduced by one space.

1. Algorithm for running ENQUEUE
   - ❖ Check to see whether or not the queue is complete.
   - ❖ If the queue is complete, then the program will print an overflow error and exit.
   - ❖ If the queue is not complete, raise the queue and add an element to it.
2. Algorithm for working with DEQUEUE
   - ❖ Please check whether or not the queue is empty.
   - ❖ If the queue is empty, then the program will print an underflow error and exit.
   - ❖ If the queue is not empty, print the head element and increase the head value.

```
/* Below program is written in C++ language */
#include<iostream>
using namespace std;
#define SIZE 10
class Queue
{
int a[SIZE];
int rear;   //same as tail
int front;  //same as head
public:
{
rear = front = -1;
}
 //declaring enqueue, dequeue and display functions
void enqueue(int x);
int dequeue();
void display();
};
// function enqueue - to add data to queue
void Queue :: enqueue(int x)
{
if(front == -1) {
front++;
}
if( rear == SIZE-1)
{
```

Notes

```
    cout<< "Queue is full";
    }
    else
    {
    a[++rear] = x;
    }
    }
    // function dequeue - to remove data from queue
    int Queue :: dequeue()
    {
    return a[++front]; // following approach [B], explained
above
    }
    // function to display the queue elements
    void Queue :: display()
    {
    int i;
    for( i = front; i<= rear; i++)
    {
    cout<< a[i] <<endl;
    }
    }
    // the main function
    int main()
    {
    Queue q;
    q.enqueue(10);
    q.enqueue(100);
    q.enqueue(1000);
    q.enqueue(1001);
    q.enqueue(1002);
    q.dequeue();
    q.enqueue(1003);
    q.dequeue();
    q.dequeue();
    q.enqueue(1004);
    q.display();
    return 0;
    }
```

Notes

```
    return a[0];//returning first element
    You simply need to change the dequeue method to implement
approach[A] and add a for loop that will change all the
remaining elements by one location.
    for (i = 0; i< tail-1; i++) //shifting all other elements
    {
    a[i] = a[i+1];
    tail--;
    }
```

1.1.5 Linked List

Introduction to Linked Lists

The Linked List is a linear data structure consisting of a set of nodes in a sequence that is very widely used. Each node contains its data and the address of the next node, thus creating a structure like a chain. To build trees and graphs, connected lists are used.

Advantages of Linked Lists

1. They are dynamic in nature which, when needed, allocates the memory.
2. It is possible to quickly execute insertion and deletion operations.
3. Stacks and queues can be executed conveniently.
4. The Linked List reduces the time for entry.

Linked Lists Drawbacks

1. As pointers require extra memory for storage, the memory is wasted.
2. No element can be randomly accessed; each node must be sequentially accessed.
3. In the linked list, Reverse Traversing is hard.

Related Linked Lists Program

1. For implementing stacks, queues, graphs, etc., linked lists are used.
2. At the beginning and end of the list, connected lists let you insert elements.
3. We do not need to know the size in advance for Linked Lists.

Types of Linked Lists

There are 3 different Linked List implementations available, and they are:

1. Linked Single List.
2. Doubly LinkedList.
3. Circular Linked List.

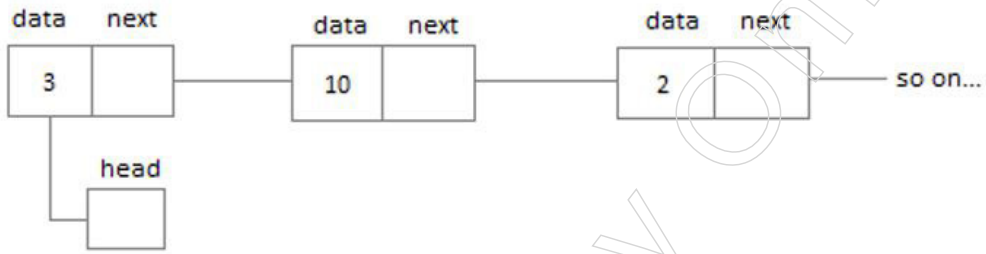
Notes

Let us get to know more about them and how different they are from each other.

Single Linked List

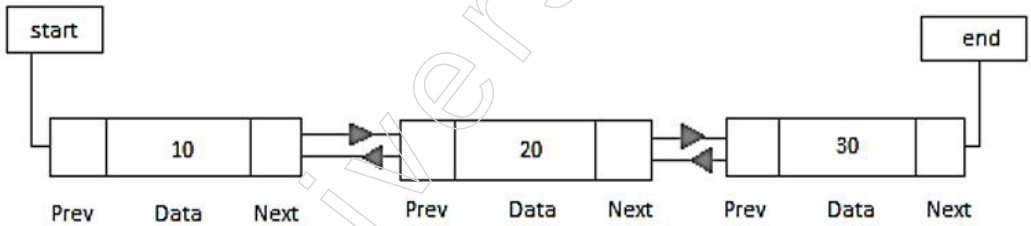
Single linked lists contain nodes that have a part of the data as well as a part of the address, i.e. the next, referring to the next node in the node chain.

Insertion, deletion, and traversal are the operations we can perform on independently bound lists.



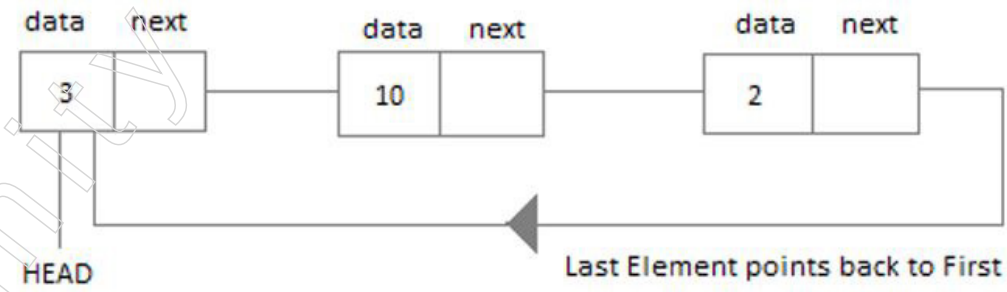
Doubly Linked List

Each node contains a data component and two addresses in a doubly-linked list, one for the previous node and one for the next node.



Circular Linked List

The last node of the list in a circular connected list holds the address of the first node, thus creating a circular chain.



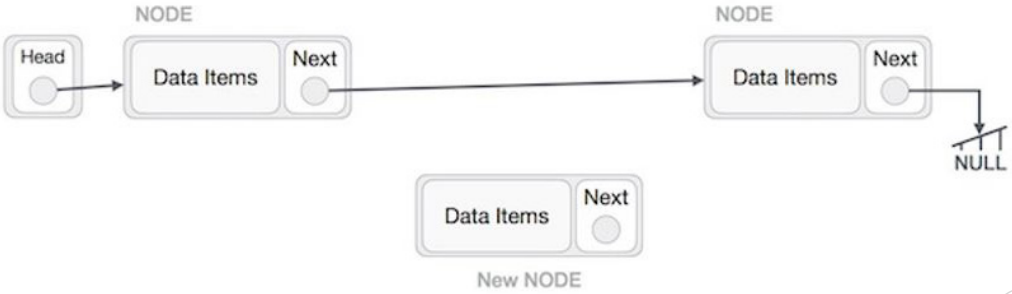
Fundamental Operations

The basic operations supported by a list are then followed.

- Insertion − At the beginning of the list, add an element.
- Deletion − Deletes an item at the beginning of the list.
- Display − Shows the list in full.
- Search: Use the given key to search for an element.
- Delete − Using the given key, delete an element.

Notes

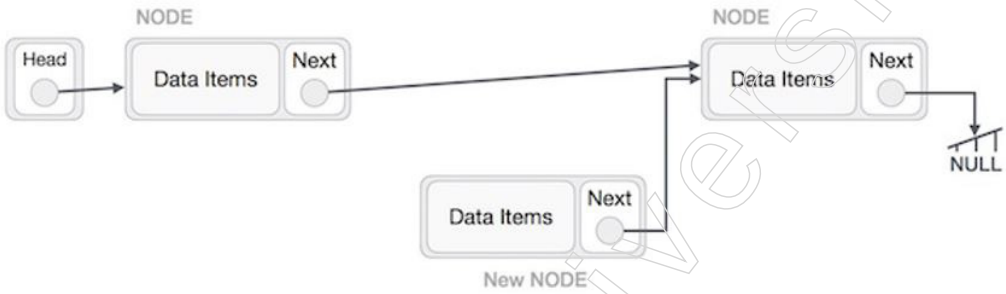
Operation Insertion



Adding a new node to the linked list is a multi-step activity. With diagrams here, we shall learn this. First, using the same structure, create a node and find the place where it has to be inserted.

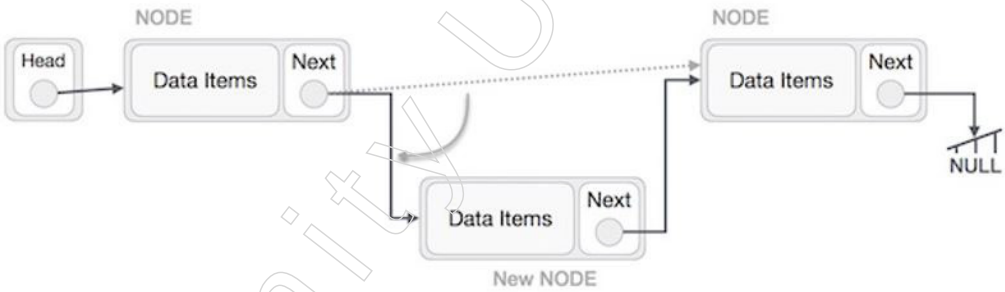
Imagine inserting a B (NewNode) node between A (LeftNode) and C (RightNode). Point B.next to C, then −

NewNode.next −>RightNode;



Now, the next node should point to the new node on the left.

LeftNode.next +>NewNode;



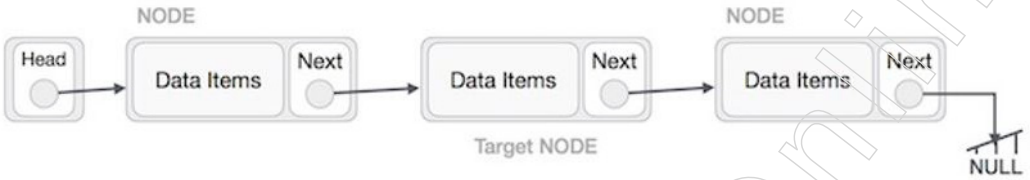
This is going to position the new node in the middle of the two. The updated list ought to look like the figure below

Similar steps should be taken if the node at the beginning of the list is added. The second last node of the list should point to a new node when adding it at the top, and the new node should point to NULL.

Notes

Operation Deletion

Deletion is often a process involving more than one Phase. With pictorial representation, we will understand. First, by using search algorithms, find the target node that you want to delete.



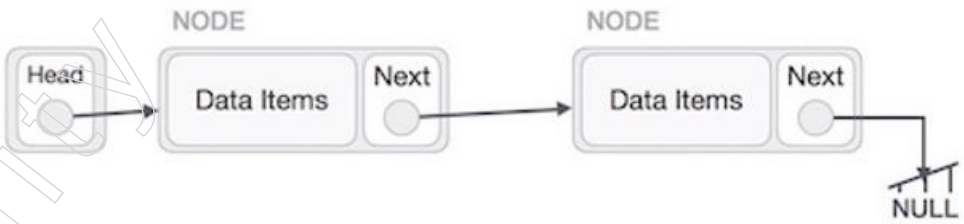
The target node's left (previous) node should now point to the target node's next node −

LeftNode.next =>TargetNode.next;

This will remove the connection to the target node that was pointing to it. Now we're going to remove what the target node is pointing to, using the following code.

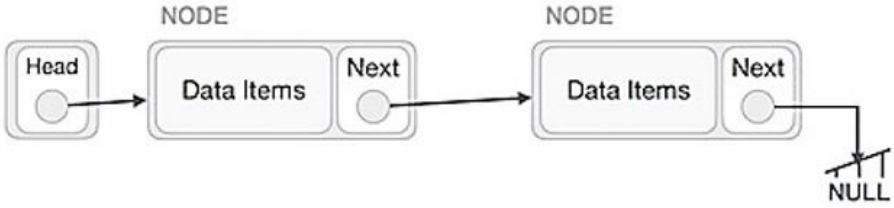
TargetNode.next => NULL;

We need to use the node that was deleted. Otherwise, we can simply allocate memory and wipe off the target node entirely. We can keep that in memory.



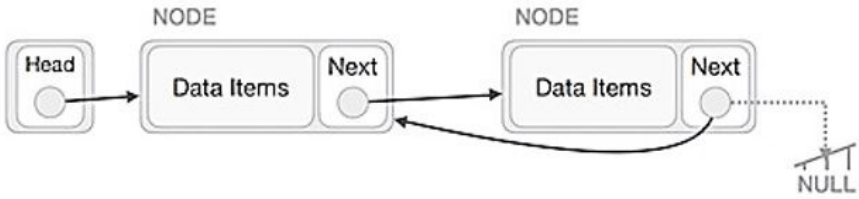
Operation Reverse

This operation is an exhaustive one. We need to render the last node that the head node is pointing at and reverse the entire linked list.

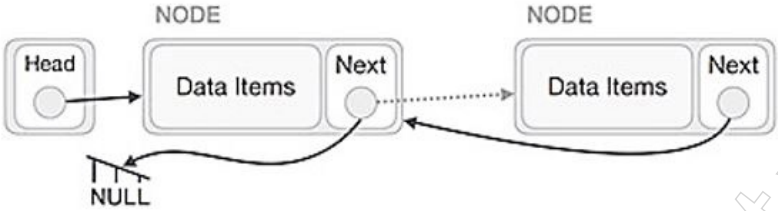


Next, we're going to the end of the page. That should point to NULL. Now, we will point it to its previous node,

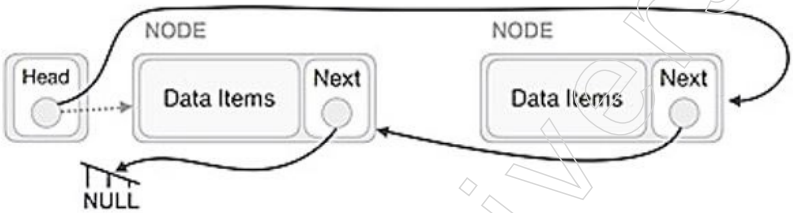
Notes



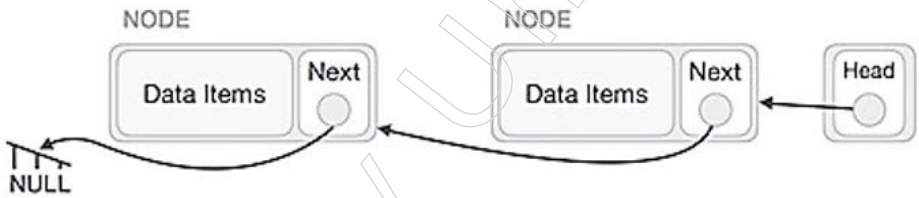
We've got to make sure the last node isn't the last one. So, we're going to have a temp node, which looks like a head node that points to the last node. Now, one by one, we are going to make all left side nodes point to their previous nodes.



All nodes should point to their predecessor, excluding the node (first node) pointed to by the head node, making them their current successor. NULL will be referred to by the first node.



Using the temp node, we will make the head node point to the first new node.



The list linked is now reversed.

1.1.6 Double Linked List

The Doubly Linked List is a form of linked list in which there are two links to each node apart from storing its data. The first connection points to the previous node in the list, while the second connection points to the next node in the list. The list's first node has its previous link pointing to NULL The last node in the list, likewise, has its next node pointing to NULL.

The relevant words for understanding the notion of a doubly connected list are below.

Link − A data called an element can be stored by each link of a linked list.

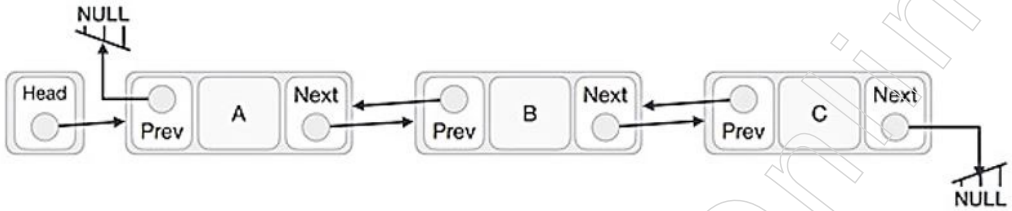
Next − Each link in a linked list includes a link named Next to the next link.

Prev − Each link on a linked list includes a link named Prev to the previous link.

Notes

Linked List- The relation link to the first link named First and to the last link called Last is found in − A Linked List.

Representation of Doubly Linked List:



As per the above example, the essential points to be considered are below.

- There is a link feature named first and last in the Doubly Linked List.
- Each link has a field(s) of data and two fields of links called next and prev.
- Using his next link, each link is connected to his next link.
- Using its previous link, each link is associated with its previous link.
- To mark the end of the list, the last link carries a null link.

Fundamental Activities

- The basic operations provided by a list are then followed.
- Insertion − At the beginning of the list, add an element.
- Deletion eliminates an item from the top of the list.
- Insert Last − The element at the end of the list is inserted.
- Delete Last − Deletes an element from the list's edge.
- Insert After − Adds an element to the list after an object.
- Delete − Deletes an element using a key from the list.
- Show forward: Shows in a forward manner the full list.
- Show backward-Shows in a backward way the complete list.

Operation Insertion

The following code indicates the insertion operation at the start of a double-linked list.

```
   //insert link at the first location
   voidinsertFirst(int key,int data){

   //create a link
   struct node *link =(struct node*)malloc(sizeof(struct
node));
   link->key = key;
   link->data = data;

   if(isEmpty()){
   //make it the last link
```

Notes

```
last= link;
}else{
//update first prev link
head->prev= link;
}

//point it to old first link
link->next= head;

//point first to new first link
head = link;
}
```

Operation Deletion

The following code indicates the deletion operation at the start of a double-linked list.

```
//delete first item
struct node* deleted first(){

//save reference to first link
struct node *tempLink= head;

//if only one link
if(head->next== NULL){
last= NULL;
}else{
head->next->prev= NULL;
}

head = head->next;

//return the deleted link
returntempLink;
}
```

Insertion At the End of an Operation

```
//insert link at the last location
void insertlast(int key,int data){

//create a link
```

Notes

```
    struct node *link =(struct node*)malloc(sizeof(struct
node));
    link->key = key;
    link->data = data;

    if(isEmpty()){
    //make it the last link
    last= link;
    }else{
    //make link a new last link
    last->next= link;

    //mark old last node as prev of new link
    link->prev=last;
    }

    //point last to new last node
    last= link;
    }
```

1.2 Algorithm

Data structures and algorithms are intricately linked. A data structure's utility is enhanced when paired with efficient algorithms, and conversely, algorithms rely on appropriate data structures for optimal performance.

- Efficiency: The effectiveness of both data structures and algorithms lies in their efficiency. Efficient storage, retrieval, and manipulation of data are key goals of DSA.
- Problem Solving: DSA equips programmers with the tools to approach problem-solving systematically. By selecting suitable data structures and algorithms, complex problems can be tackled effectively.

Benefits of Understanding DSA

- Informed Decision Making: Proficiency in DSA enables informed decisions regarding the selection of the most appropriate data structure or algorithm for a given scenario.
- Performance Optimization: Mastery of DSA facilitates the creation of programs that run faster and consume less memory, contributing to enhanced performance.
- Systematic Problem Solving: DSA provides a structured framework for problem-solving, empowering individuals to tackle intricate problems methodically.

1.2.1 Algorithm and Characteristics

To accomplish a certain predefined task, an algorithm is a finite set of instructions or logic written in order. An algorithm is not the complete code or program, it is just a

Notes

problem's core logic(solution), which can be expressed either as pseudo code or using a flowchart as an informal high-level description.

The Following Properties Must Be Satisfied By Each Algorithm:

- Input- An algorithm should have 0 or more inputs supplied externally.
- Output- A minimum of 1 output should be obtained.
- Definiteness- The algorithm's every step should be clear and well defined.
- Finiteness- A finite number of steps should be in the algorithm.
- Correctness- Each algorithm step must produce correct output.

If it takes less time to execute and consumes less memory space, an algorithm is said to be efficient and fast.

The Algorithm's Performance Is Measured based on The Following Properties:

1. Complexity in Time

   Time Complexity is a way of representing the amount of time the program requires to run until it is completed. In general, trying to keep the time required minimum is a good practice, so that our algorithm completes its execution in the minimum possible time. In more detail, we will study Time Complexity in later sections.

   NOTE: You should have a good understanding of programming either in C or in C++ or Java or Python etc before going deep into the data structure.

2. Complexity in Space

   This is the amount of memory space that the algorithm requires during its execution. For multi-user systems and in situations where limited memory is available, space complexity has to be taken seriously.

   For the following components, an algorithm generally requires space:

   - ❖ Instruction space: This is the space required to store the program's executable version. This space is fixed but varies depending on the program's number of code lines.
   - ❖ Data Space: This is the space needed to store the value of all constants and variables (including temporary variables).
   - ❖ Environment Space: It is the space needed to store the information needed to resume the suspended function of the environment.

Applications

Stack's Applications

Reversing a term is the easiest use of a stack. You push a word into a stack, letter by letter, and then pop letters out of the stack. Other uses still exist, such as Parsing Speech Conversion (Infix to Postfix, Postfix to Prefix, etc.)

Queue Applications

Queue, as the name indicates, is used if we need to treat any group of objects in an order in which the first one comes in, while the others wait for their turn, also gets out first, as in the following scenarios:

1. Serving demands, such as a printer, CPU task scheduling, etc., on a single shared resource

Notes

2. Call Center telephone networks use lines in real-life situations to keep people calling them in order until a service representative is ready.
3. Interrupt management of real-time applications. The interrupts are treated in the same order, i.e., first come first served, as they arrive.

Analysis and Efficiency of Algorithms

An algorithm's efficiency can be evaluated at two separate levels, before and after implementation. These are the following −

A Priori Analysis- This is an algorithm theoretical analysis. An algorithm's efficiency is calculated by assuming that all other variables are constant and have no effect on the implementation, such as processor speed.

A Posterior Analysis- This is an algorithm's empirical analysis. A programming language is used to implement the chosen algorithm. This is executed on the target computer machine afterward. In this analysis, actual statistics are collected, such as the necessary running time and space.

We will think about testing the algorithm a priori. The study of algorithms deals with the execution or running time of different operations involved. You may describe the running time of an operation as the number of machine instructions per operation performed.

Complexity of Algorithms

If X is an algorithm and n is the input data size, the time and space used by the X algorithm are the two key factors that determine X's performance.

- Time Factor: Time is calculated by counting the number of main operations in the sorting algorithm, such as comparisons.
- Space Factor: Space is determined by counting the algorithm's maximum memory space available.

The complexity of an algorithm f(n) gives the algorithm's necessary runtime and/or storage space in terms of n as the input data size.

Complexity in Space

The algorithm's space complexity reflects the amount of memory space needed by the algorithm throughout its life cycle. The space needed by an algorithm is equal to the sum of the two components that follow,

- A fixed component is a space needed to store some information and variables that are independent of the problem's size. Easy variables and constants, for instance, used, program size, etc.
- A part of a variable is the space needed by variables, the size of which depends on the problem's size. For instance, allocating dynamic memory, recursion stack space, etc.
- Complexity in space Of any algorithm, S(P) P is S(P) = C + SP(I), where C is the fixed part and S(I) is the algorithm's variable part, which depends on the characteristic of the instance I.

The following is a brief example that helps to illustrate the theory.

SUM: algorithm (A, B)

Notes

Phase 1 - Beginning

Phase 2 - C > A + B + 10

Phase 3 - Avoiding

We have three variables here, namely A, B, and C, and one constant. Therefore, S(P)= 1 + 3. Now, space depends on and will be multiplied accordingly by data types of given variables and constant types.

Analysis of Algorithm

In the theoretical study of algorithms, it is usual to estimate the complexity function in an asymptotic sense, that is, for arbitrarily large input. Donald Knuth coined the term "the study of algorithms."

Algorithm analysis is a vital aspect of computational complexity theory since it provides a theoretical estimate of how much time and energy an algorithm would take to solve a given problem. The majority of algorithms are made to work with inputs of any length. Algorithm analysis is the calculation of the amount of time and space resources necessary for it to be performed.

Usually, an algorithm's efficiency or running time is expressed as a function comparing the input length to the number of steps (time complexity) or the volume of memory (space complexity).

The Importance of Analysis

The evaluation of algorithms and the selection of the most suitable algorithm for a given problem are crucial due to the existence of multiple algorithms that can solve the same computational problem.

- Pattern Recognition Enhancement: Assessing algorithms for specific problems aids in improving pattern recognition, enabling the utilisation of these algorithms for solving similar types of problems.
- Algorithm Variance: Although algorithms may differ significantly, they often share the same objective. For example, various algorithms can be employed to sort a set of numbers, each differing in the number of comparisons performed and thus affecting time complexity.
- Time and Space Complexity: Algorithm analysis involves evaluating the problem-solving capabilities of algorithms based on their time and space requirements. This includes measuring the memory space required for implementation.
- Methodology of Analysis: The primary focus of algorithm analysis is on the necessary time or efficiency. Various types of analysis are typically conducted, including:
  - ❖ Worst-case-the maximum number of steps taken in an example of a scale.
  - ❖ The smallest number of steps taken on any instance of size and in the best-case scenario.
  - ❖ An average number of steps taken on any instance of size and in the average case.

Understanding Amortised Concepts

Definition: Amortised refers to a concept where a sequence of operations applied to a specific input size is averaged over time.

Notes

Considering Time and Space Complexity

Dual Consideration: Solving a problem necessitates evaluating both time and space complexity, as programs may operate on devices with varied memory and processing constraints.

Bubble Sort vs. Merge Sort

- Memory Usage: Bubble sort requires no additional memory, while merge sort necessitates extra space for its operations.
- Time Complexity: Merge sort typically outperforms bubble sort in terms of time complexity.
- Memory Constraints: In scenarios where memory is limited but sufficient processing power is available, bubble sort may be a preferable choice despite its higher time complexity.

Analysis Method – Asymptotic Analysis

An algorithm's asymptotic analysis refers to the concept of its run-time performance's mathematical foundation/framing. We can very well infer the best case, average case, and worst-case scenario of an algorithm using asymptotic analysis.

Asymptotic analysis is bound by input, i.e. it is concluded to operate in a constant time if there is no input to the algorithm. All other variables are considered stable, other than the input.

In mathematical units of computation, asymptotic analysis refers to calculating the running time of any operation. E.g., one operation's running time is computed as f(n), while another operation's running time is computed as g. (n2). This means that as n increases, the first operation's running time will increase linearly, while the second operation's running time will increase exponentially. Similarly, if n is substantially small, the runtime of both operations would be almost the same.

An algorithm's time requirement typically falls into one of three groups.

- ❖ In the best-case scenario, the program execution time is as short as possible.
- ❖ Average Case Average time needed for execution of the program.
- ❖ Bad Case-Maximum time needed for implementation of the program

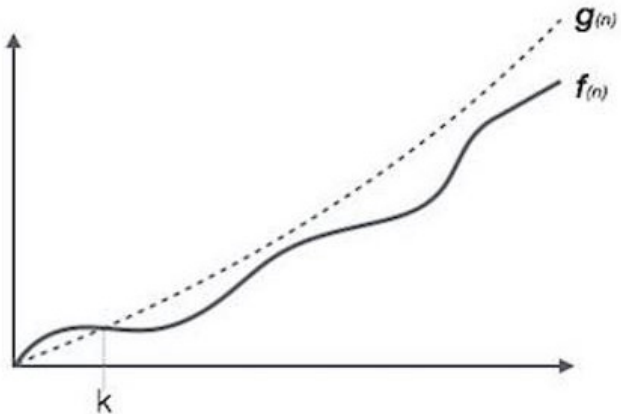
1.2.2 Asymptotic Notations

The widely used asymptotic notations to measure an algorithm's running time complexity are as follows.

- O Notation
- Ω Notation
- θ Notation
- Big Oh Notation, O

The formal way to express the upper bound of an algorithm's running time is to use the notation (n). It calculates the worst-case time complexity, or the maximum time an algorithm will take to complete.

Notes

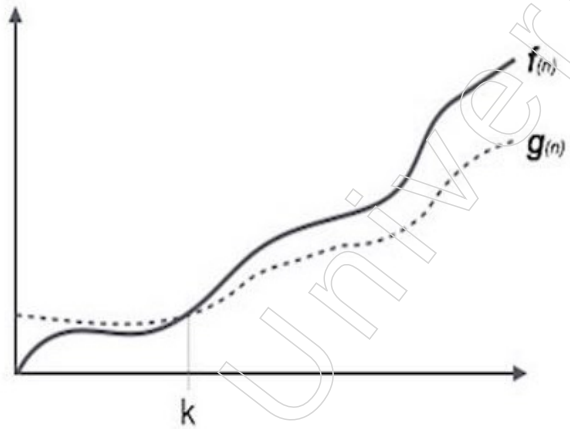


For example, for a function f(n)

O(f(n)) = { g(n) : there exists c > 0 and n0 such that f(n) ≤ c.g(n) for all n > n0. }

Omega Notation, Ω

The notation Ω (n) is the formal way of describing the lower limit of the running time of an algorithm. It calculates the best-case time complexity, or the shortest time an algorithm will take to complete.

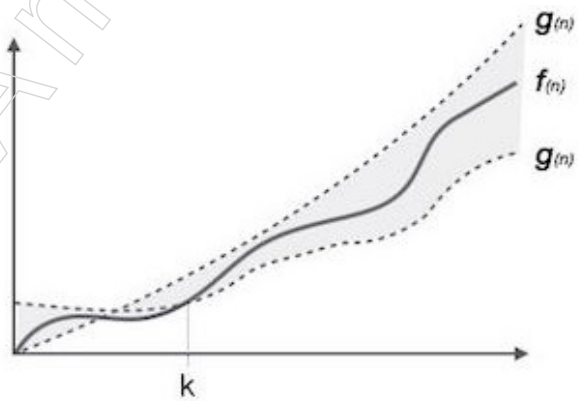


For a function f(n)

Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Theta Notation θ

The notation θ(n) is the official way to express both the lower bound and the upper bound of the running time of an algorithm. The following is a representation of it:



Notes

$\theta(f(n))$ = { g(n) if and only if g(n) = $O(f(n))$ and g(n) = $\Omega(f(n))$ for all $n > n_0$. }

Asymptotic Notations Used Frequently

A list of some common asymptotic notations is given below.

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{o(1)}$
exponential	–	$2^{0(n)}$

1.2.3 Algorithm Time Complexity

An algorithm's time complexity reflects the amount of time needed for the algorithm to run to completion. Time requirements can be defined as a T(n) numeric function, where T(n) can be calculated as the number of steps, given that constant time is consumed by each step.

For instance, n steps are used to add two n-bit integers. Therefore, the total computational time is T(n) = c * n, where c is the time taken for two bits to be inserted. Here, as the input size increases, we note that T(n) grows linearly.

1.2.4 Recursive Program Analysis Strategies

Many algorithms are necessarily recursive. We get a recurrence relation for time complexity when we evaluate them. We can measure the running time for an input of size n as a function of n, as well as the running time for smaller inputs. To sort a given array in Merge Sort, for example, we divide it into two halves and recursively repeat the process for each half. Finally, we integrate the effects. The Merge Sort's time complexity can be written as T(n) = 2T(n/2) + cn. There are many other algorithms, such as Binary Search, Hanoi Tower, etc.

Three methods of solving recurrences are mainly available.

1. Substitution Method: We make an informed guess at the answer and then use mathematical inference to show whether or not our guess is correct.

   Consider, for instance, the recurrence T(n) = 2T(n/2) + n

   We think of the solution as T(n) = O (N log N). Now, to prove our guess, we are using induction.

   We've got to show T(n) <= cnLogn. For values smaller than n, we can assume that it is valid.

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

   <= 2cn/2Log(n/2) + n

   = cnLogn - cnLog2 + n

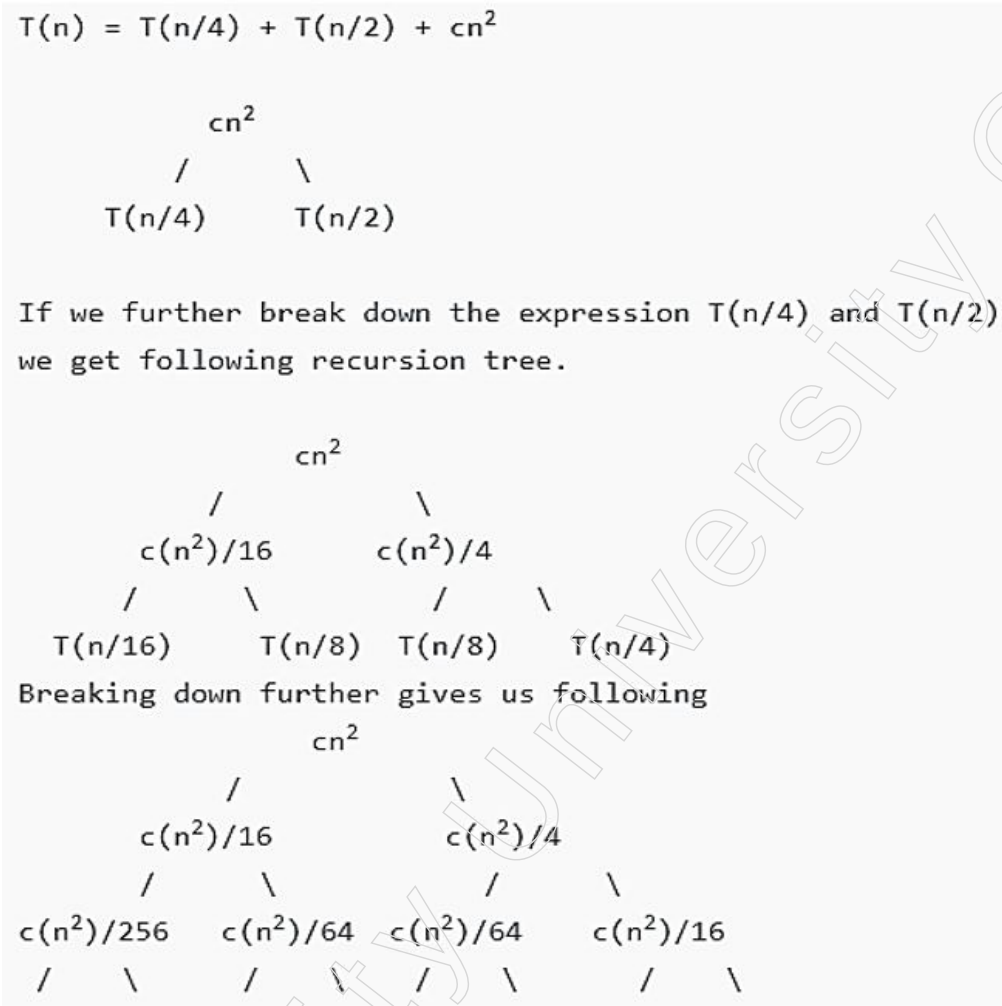
   = cnLogn - cn + n

   <= cnLogn

Notes

2. Recurrence Tree Method: We draw a recurrence tree in this method and measure the time taken by every tree stage. Finally, we add up all of the work we have done so far. To draw the recurrence tree, we begin with the given recurrence and continue drawing until a pattern emerges among the levels. Typically, the pattern is an arithmetic or geometric sequence.

   Clipping Below Shows an example of a recurrence relation.



To find the value of T(n), we must add the sums of the tree nodes at each step. If we sum the tree level above by level, we get the series below

T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + _____

Geometrical progression with a ratio of 5/16 is the sequence above.

We can sum the infinite series up to get an upper bound.

The sum is given as $(n^2)/(1 - 5/16)$, which is O (n^2)

1.2.5 Master Theorem

A straightforward way to get the solution is the Master Method. The master method only works with the following type of recurrence or recurrence that can be translated to the next type.

Notes

T(n) = aT(n/b) + f(n), where a and b are both greater than one.(a & b >1)

The following are three scenarios:

If f(n) = Θ(nc) & c <Logba then T(n) = Θ(nLogba)

If f(n) = Θ(nc) & c = Logba then T(n) = Θ(ncLog n)

If f(n) = Θ(nc) & c >Logba then T(n) = Θ(f(n))

How is this functioning?

The recurrence tree method is the key source of inspiration for the master method. If we draw T(n)= aT(n/b) + f(n) recurrence tree, we can see that the root word is f(n) and the work performed on all leaves is al(NC) where c is Logba. And the recurrence tree's height is log in.

1.3 Divide and Conquer Paradigm

The Divide and Conquer Algorithm is a problem-solving approach that involves breaking down a main problem into smaller subproblems, solving them independently, and then merging the solutions to obtain the solution for the original problem.

Benefits and Applications

- Efficiency: Divide and Conquer Algorithm aids in solving complex problems efficiently by breaking them down into manageable subproblems.
- Versatility: This algorithmic technique can be applied to a wide range of problems across various domains, including computer science, mathematics, and engineering.

Problem-Solving Process

- Divide: The main problem is divided into smaller, more manageable subproblems.
- Conquer: Each subproblem is solved independently using appropriate techniques or algorithms.
- Merge: The solutions to the subproblems are combined or merged to obtain the solution for the original problem.

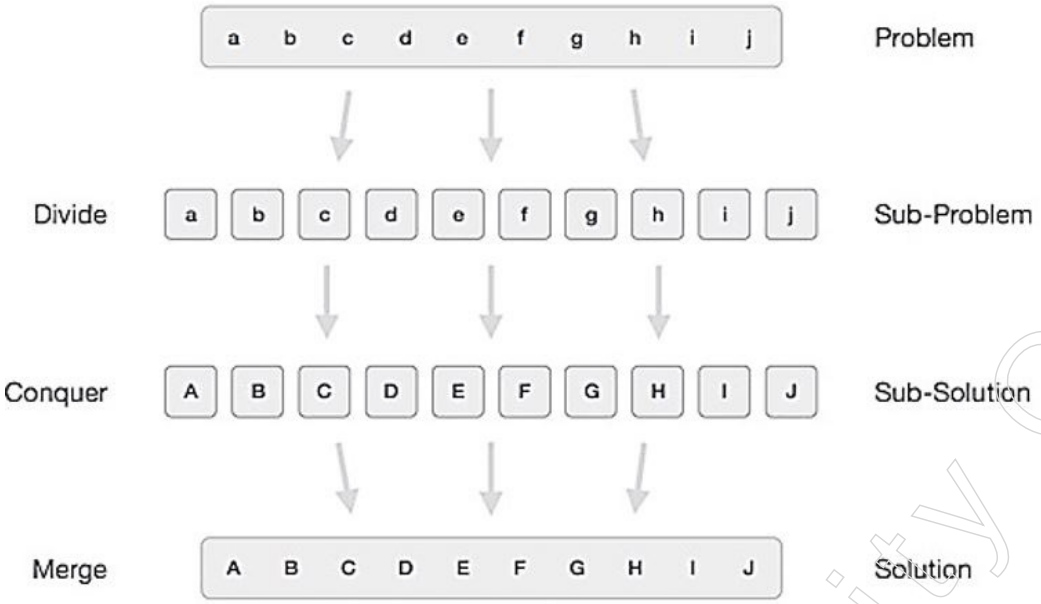
Examples of Usage

- Sorting Algorithms: Divide and Conquer techniques are utilised in sorting algorithms such as Merge Sort and Quick Sort.
- Searching Algorithms: Techniques like Binary Search employ Divide and Conquer to efficiently search through a sorted array or list.

1.3.1 Divide-Conquer Recurrence Equations Solutions

The issue at hand is divided into smaller subproblems in the divide and conquer method, and then each issue is solved separately. We can eventually reach a point where no further division is possible if we continue to split the subproblems into smaller subproblems. The smallest possible "atomic" sub-problem (fractions) is solved. Finally, the solution of all sub-problems is combined to achieve the solution of an initial problem.

Notes



The divide-and-conquer strategy can be broken down into three stages.

Divide

Breaking the problem down into smaller subproblems is the next step. Sub-problems should be a part of the original topic. In general, this move uses a recursive method to divide the issue until no sub-problem is further divisible. Sub-problems become atomic in nature at this stage, but they still reflect a portion of the larger problem.

Conquering/Solving

This process requires several smaller subproblems to be tackled. Generally, the issues are deemed 'solved' on their own at this stage.

Combine/Merge

This stage recursively combines them as the smaller subproblems are solved, before they formulate a solution to the original problem. This algorithmic approach is recursive, and the conquer and merge phases are so similar together that they appear to be one.

The following computer algorithms are based on the programming method of divide and conquer.

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

There are several approaches to solving any computer problem, but the ones mentioned above are a good example of a divide and conquer strategy.

Notes

Dynamic Programming Approach

In breaking down the problem into smaller and smaller potential sub-problems, the dynamic programming approach is similar to divide and conquer. But these sub-problems, unlike divide and conquer, are not individually solved. Instead, for related or overlapping subproblems, the outcomes of these smaller subproblems are remembered and used.

Where we have problems, dynamic programming is used, which can be broken into related sub-problems such that their results can be reused. For optimization, mainly, these algorithms are used. The dynamic algorithm will try to analyse the outcomes of the previously solved sub-problems before solving the in-hand sub-problem. Sub-problem solutions are mixed to find the best solution.

But we should say that...

- The issue can be divided into smaller subproblems that overlap.
- Using the optimal solution of smaller sub-problems, an optimum solution can be obtained.
- Memorization is used in complex algorithms.

Comparison

Dynamic algorithms are driven to optimise the problem overall, as opposed to greedy algorithms, where local optimization is discussed.

Dynamic algorithms use the output of a smaller sub-problem and then try to maximise a bigger sub-problem, in comparison to divide and conquer algorithms, where solutions are combined to achieve an overall solution. Memorization is used by dynamic algorithms to recall the performance of sub-problems already solved.

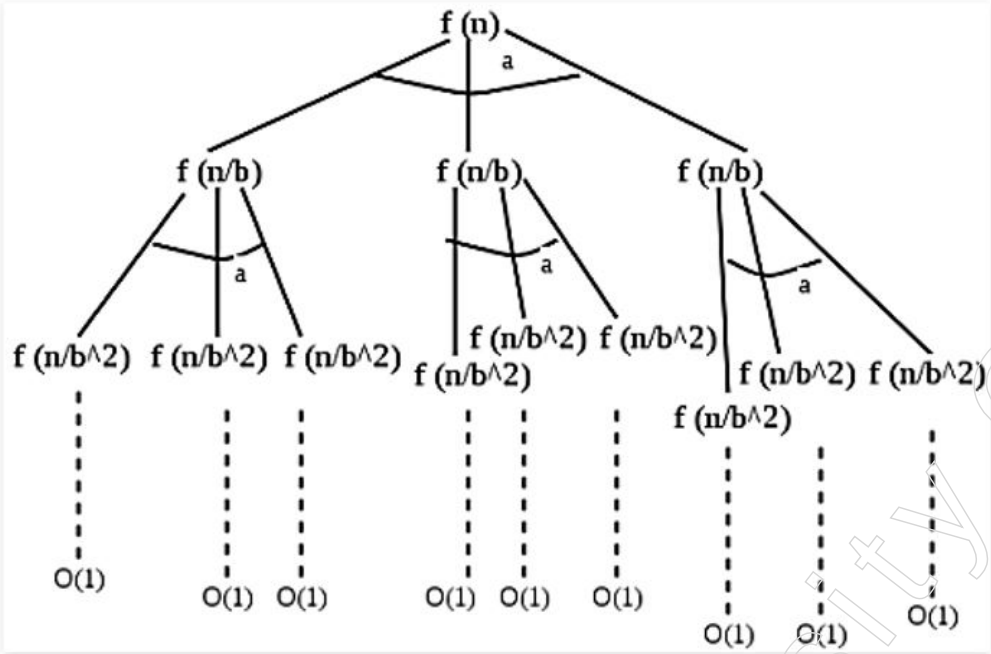
As an example

The dynamic programming methodology can be used to solve the following computer problems:

- Series of Fibonacci numbers
- Knapsack problem
- The Hanoi Tower
- Floyd-Warshall found the shortest path for all pairs.
- The shortest Dijkstra route
- Timetables for programs

Both top-down and bottom-up approaches can be used for dynamic programming. And, of course, much of the time, in terms of CPU cycles, referring to previous solution performance is cheaper than recomputing.

Notes



We measure the total work done in the recurrence tree process. If the work done at the leaves is polynomially more, leaves are the dominant component, and our result is the work done at the leaves (Case 1). If the work done at the leaves and root is asymptotically the same, the result is height multiplied by the work done at either step (Case 2). If the sum of work done at the root is asymptotically greater, our result is work done at the root (Case 3).

Examples of some traditional algorithms whose time complexity can be calculated using the Master theorem.

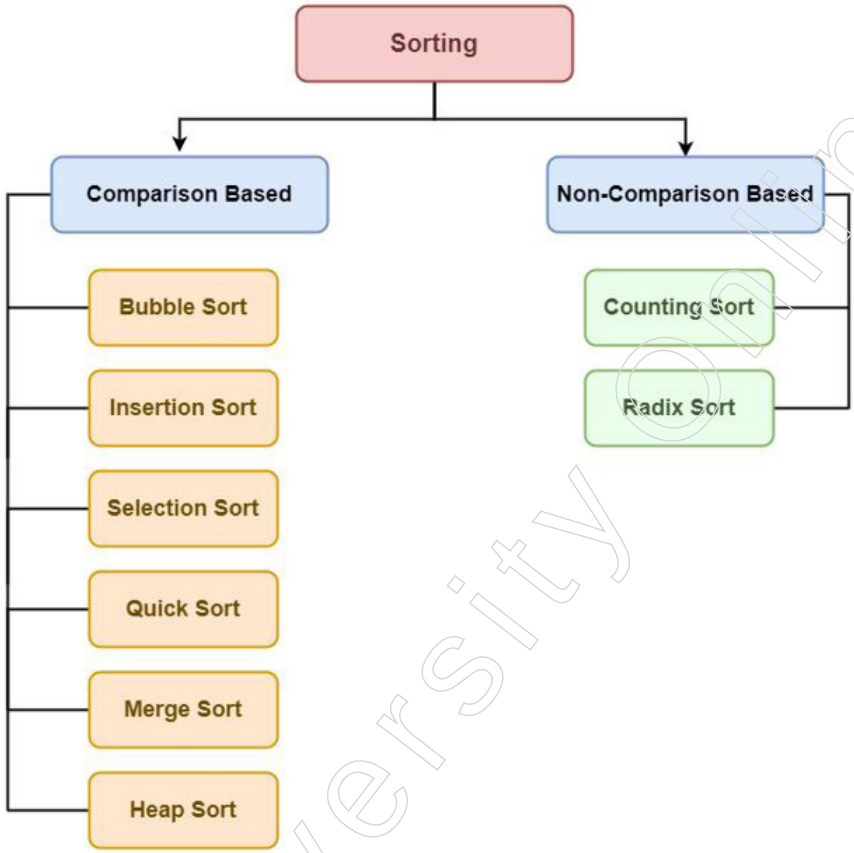
1.4 Sorting Techniques

Sorting involves reordering a given array or list of elements based on a defined comparison operator, determining the new arrangement of elements within the data structure.

Types of Sorting Techniques

1. Comparison-based Sorting Algorithms:In comparison-based sorting algorithms, elements are compared to each other based on a defined comparison operator. The relative order of elements is determined through these comparisons.
2. Non-comparison-based Sorting Algorithms:Non-comparison-based sorting algorithms do not rely on direct comparisons between elements to determine their order. Instead, alternative techniques are employed to arrange the elements according to specific criteria, often exploiting properties of the data itself rather than comparing individual elements.

Notes



1.4.1 Bubble Sort

Bubble sorting is a basic algorithm for sorting. This sorting algorithm is a comparison-based algorithm that compares each pair of adjacent elements and, if they are not in order, swaps the elements. For large data sets, this algorithm is not suitable because its average and worst-case complexity are O(n2), where n is the number of products.

What is the method for sorting bubbles?

As an example, we will use an unsorted array. Bubble sorting takes μ(n2) time, so we keep it fast and accurate.

The first two elements in the bubble type are compared to see which one is greater.

Value 33 is greater than 14 in this situation because it is already in the sorted positions. After that, we equate 33 to 27.

We discover that 27 is less than 33, so the two numbers must be switched.

Notes

The new array ought to look like this,

After that, we'll compare 33 and 35. We learn that both have already been sorted.

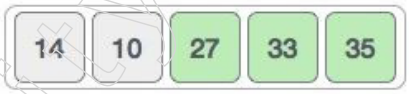
After that, we'll look at the next two numbers, 35 and 10.

Therefore, we know that 10 is less than 35. As a result, they are not sorted.

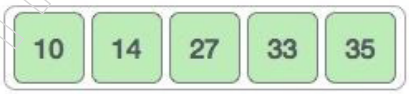
These values are swapped. We conclude that we have reached the end of the array. The array should look like this after one iteration.

To be more accurate, we are now illustrating how an array should appear after each iteration. It should look like this after the second iteration:

It is worth remembering that at the end of each iteration, at least one value shifts.



And bubble sorts learn that an array is fully sorted when there is no swap needed.



Now it is time to look at some of the practical aspects of bubble sorting.

The Algorithm

We assume that the list is an array of elements with n. We also assume that the swap function swaps the values of the array elements passed to it.

start sorting bubbles (list)

for all list elements

If list[i] > list[i+1] is defined,

Notes

Exchange(list[i], list[i+1])

if it's over

End for

Return List

Terminate the Bubble Sort

Pseudocode for Bubble Sort

We note in the algorithm that, unless the entire array is completely sorted in ascending order, Bubble Sort compares each pair of array elements. This can cause a few problems of complexity, such as what if the array no longer needs to be switched as all the elements are already raising.

We use a swapped flag variable to ease the problem, which will help us see whether any swap has occurred or not. It will exit the loop if no swap has occurred, i.e. the array needs no further processing to be sorted. The pseudo-code for the Bubble Sort algorithm is as follows:

The bubble Sort procedure (list: An array of items)

list. Count; loop = list. Count;

To loop-1 for I = 0, do:

Correct = switched

Do the following for j = 0 to loop-1:

/* Compare the elements next to each other */

If list[j] is greater than list[j+1], then

/* turn them around */

Exchange(list[j], list[j+1])

Swapped = Actual

if it's over

End for the

/*If no numbers were swapped, it means that

Break the loop now that the list has been sorted.

Then if(not swapped)--? Split in a break? if it's over? End for

Return list at the end of the treatment

Putting it Into Effect

Another problem we did not answer in our original algorithm and its improvised pseudo code is that the highest values settle at the end of the list after each iteration. As a consequence, the next iteration does not need to contain elements that have already been sorted. For this reason, we restrict the inner loop in our implementation to avoid already sorted values.

1.4.2 Insertion Sort

This is a sorting algorithm that uses in-place comparisons. Here, a sub-list that is always sorted is preserved. The lower part of an array, for example, is held to be sorted.

Notes

An element to be 'inserted' in this sorted sub-list must find its suitable position and then it must be inserted there. As a result, the word "insertion type" was coined.

The collection is sequentially scanned, and unsorted objects are transferred to the sorted sub-list and inserted (in the same array). For large data sets, this algorithm is not suitable since its average and worst-case complexity is O(n2), where the number of items is n.

What Is Insertion Sort and How Does It Work?

In our example, we take an unsorted array.

Array unsorted

The first two elements are compared in insertion sort.

It discovers that 14 and 33 are already grouped in ascending order. For the time being, 14 is in the sorted sub-list.

Insertion sort advances to the next stage, comparing 33 to 27.

And he learns that 33 is not in the right place.

Swap 33 for 27. It also double-checks all of the elements in the sorted sub-list. Here we see that only one element of the sorted sub-list is 14, and 27 is greater than 14. Hence, after switching, the sorted sub-list remains sorted.

In the sorted sub-list, we now have 14 and 27. Next, you equate 33 to 10.

These numbers aren't grouped in any specific order.

But we're exchanging them.

Notes

Swapping makes 27 and 10 unsorted, however.

We also switch them, thus.

We'll find 14 and 10 again in an unsorted order.

We turn them around once more. We have a sorted sub-list of four items at the end of the third iteration.

This method continues until a sorted sub-list includes all of the unsorted values. We'll now look at some programming aspects of the insertion type.

Algorithm

Now we have a bigger picture of how this sorting process operates, so we can take easy steps to accomplish the form of insertion.

Phase 1: It is already sorted if it is the first element. 1 is returned;

Phase 2: Choose the next feature

Step 3: Compare all the elements in the sub-list of the sorted items

Phase 4: Move all elements in the sorted sub-list that are greater than the threshold.

A value that should be sorted

Phase 5: Fill in the value

Phase 6: Proceed until the list is sorted.

Pseudocode for Insertion

SortSorting procedure (A: an array of items)

Int HolePositionLocation

ValueToInsert int

For I = 1 to inclusive length(A) do:

/* Select the attribute to be added */

ToInsertValue = A[i]

HolePosition = I HolePosition= I

/*find the hole location for the inserted part */

If holePosition> 0 and A[holePosition-1] >ToInsertValue do:

A[holePosition] = A[holePosition-1] A[holePosition]

holePosition = holePosition -1; holePosition = holePosition -1; holePosition

Notes

End While Finishing

/* Place the number in the appropriate hole location */

A[holePosition] = valueToInsert Value

End for the

End Method Procedure

1.4.3 Selection Sort

Selection sort is a basic algorithm for sorting. This sorting algorithm is a comparison-based in-place algorithm in which the list is split into two parts, the left-end sorted part, and the right-end unsorted part. The sorted part is initially empty, while the unsorted part contains the entire list.

The unsorted array's smallest element is chosen and switched with the leftmost element, resulting in that element being a part of the sorted array. This process continues to shift the unsorted array boundary to the right by one portion.

Since the average and worst-case complexities of this algorithm are O(n2), where n is the number of objects, it is not suitable for large data sets.

How Does the Selection Sort Work?

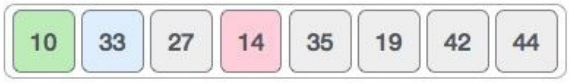
As an example, consider the array depicted below.

The entire sorted list is searched sequentially for the first place. We scan the entire list and find that 10 is the lowest value in the first position where 14 is currently stored.

And so we're replacing 14 with 10. After one iteration of 10, which is the minimum value of the list, the first position of the sorted list is shown.

We begin scanning the remainder of the list linearly for the second location, where 33 resides.

We find that the second lowest value in the list is 14 and that it should appear in the second position. We are swapping these qualities.

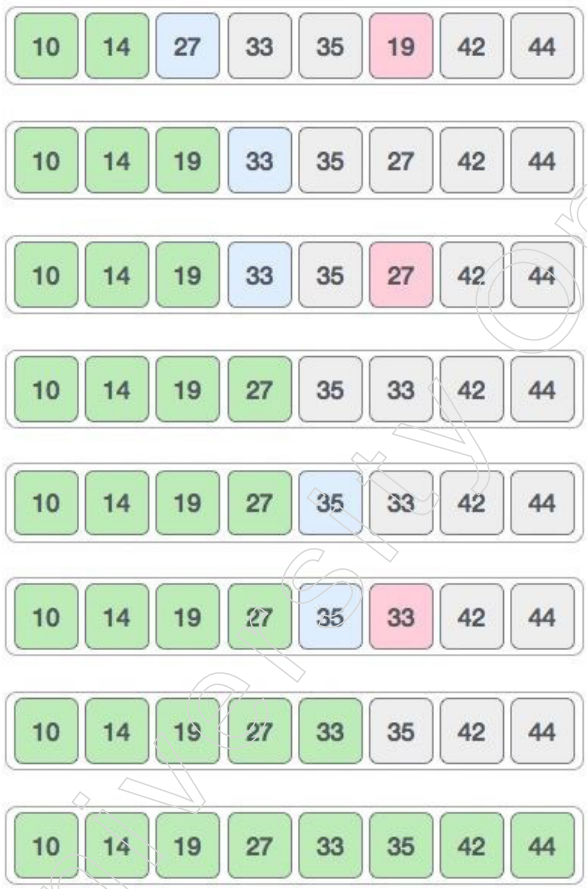


After two iterations, the two least values are sorted and put at the beginning.

Notes

The remainder of the objects in the collection are handled in the same way.

The following is a diagram of the entire sorting procedure.



Algorithm

Set MIN to 0 in the first step.

Phase 2: Check in the list for a minimum element

Phase 3 Switch at the MIN position with value

Phase 4: Increase the MIN to indicate the next factor

Phase 5: Proceed until the list is sorted.

Selection of Pseudocode Procedures List of kinds: The Object Collection

N: list size for I = 1 to n - 1

/* set the current element to a minimum of */

min = I /* check the element to be a minimum of */

for j = i+1 to n if list[j] < list[min] then min = j; end when /* ends, swap the minimum element with the current element*/

if indexMin != I swap list[min] and list[i] end when list[min] and list[i] ends.

End Method Procedure

1.4.4 Merge Sort

Merge sort is a sorting system based on the strategy of divide and conquer. It is one of the most respected algorithms, with the worst-case time complexity being O(n

log n). Merge sort splits the list into equal halves before merging them in sorted order.

Notes

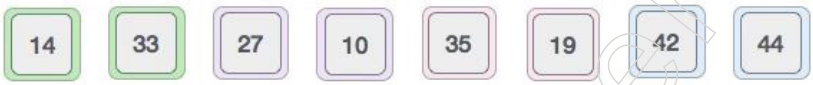
What Is Merge Sort and How Does It Work?

To understand merge sort, consider the following unsorted array:

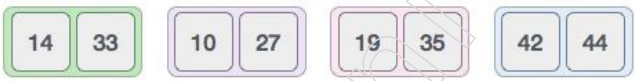
We know that the merging sort first iteratively splits the entire array into equal halves before the atomic values are reached. We can see that an 8-item array is split into two 4-item arrays.

The sequence of appearance of objects in the original does not alter this. We are breaking these two arrays into halves now.

We split these arrays further and atomic value is achieved, which can no longer be divided.



Now we put them back together in the same order as they were broken down. Please notice the color codes that are given for these lists.



We compare the elements in each list first, then combine them in a sorted manner into another list. We can see that 14 and 33 are in the right order. We compare 27 and 10, and we put 10 first in the goal list of two values, followed by 27. We modify the order of 19 and 35, while 42 and 44 are sequentially positioned.

We compare lists of two data values in the next iteration of the combining process and combine them into a list of found data values, putting them all in sorted order.

The list should look like this after the final merger.

We should now review some merge sorting programming aspects.

Algorithm

Merge sort splits the list into identical halves until it can't be divided any longer. By definition, if there is only one element in the list, then that element is sorted. Merge sort then joins the smaller sorted lists together, keeping the new list sorted as well.

Phase 1: Return if there is only one element in the list that has already been sorted.

Notes

Phase 2: split the list into two halves, recursively, until it can no longer be divided.

Phase 3: Organise the smaller lists and combine them into a new list.

Pseudocode

We shall now see the pseudo-codes for merge sort functions. As our algorithms mean two main functions − divide & merge.

Merge sort works with recursion which we shall see our implementation within an equivalent way.

```
procedure mergesort( var a as array ) if ( n == 1 ) return a
var l1 as array = a[0] ... a[n/2]
var l2 as array = a[n/2+1] ... a[n]
l1 = mergesort( l1 )
l2 = mergesort( l2 )
return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
var c as array while ( a and b have elements )
if ( a[0] > b[0] )
add b[0] to the highest of c
remove b[0] from b
else add a[0] to the highest of c
remove a[0] from a
end if
end while
while ( a has elements ) add a[0] to the highest of c
remove a[0] from a
end while
while ( b has elements )
add b[0] to the highest of c
remove b[0] from b
end while
return c
end procedure
```

1.4.5 Heap sort

Heap sort can be understood as the improved version of the binary search tree. It does not create a node as in the case of a binary search tree; instead it builds the heap by adjusting the position of elements within the array itself. In which method a tree structure called heap is used where a heap is a type of binary tree.

In this method, a tree structure called heap is used. A heap is a type of a binary tree. An ordered balanced binary tree is called a min heap where the value at the root of any sub-tree is less than or equal to the value of either of its children. An ordered balanced binary tree is called a max-heap when the value at the root of any sub-tree is

Notes

more than or equal to the value of either of its children. It is not necessary that the two children must be in some order. Sometimes the value in the left child may be more than the value in the right child and some other times it may be the other way round.

Heap sort is basically an improvement over the binary tree sort. It does not create nodes as in the case of binary tree sort. Instead, it builds a heap by adjusting the position of elements within the array itself. Basically, there are two phases involved in sorting the elements using a heap sort algorithm. They are as follows:

a) Construct a heap by adjusting the array elements.

b) Repeatedly eliminate the root element of the heap by shifting it to the end of the array and then restore the heap structure with remaining elements.

The root element of a max-heap is always the largest element. The sorting ends when all the root element of each successive heap has been moved to the end of the array (i.e., when the tree is exhausted). The resulting array now contains a sorted list.

Suppose an array consists of n number of distinct elements in memory, then the heap sort algorithm works as follows:

a) To begin with, a heap is built by moving the elements to proper positions within the array.

b) In the second phase the root element is eliminated from the heap by moving it to the end of the array.

c) The balance elements may not be a heap. So again steps (a) and (b) are repeated for the balance elements. The procedure is continued till all the elements are eliminated.

Note that for eliminating the element from the heap we need to merely decrement the maximum index value of the array by one. The elements are eliminated in decreasing order for a max-heap and in increasing order for a min-heap.

Let us now understand this procedure with the help of an example. Suppose the array k that is to be sorted contains the following elements:

11, 2, 9, 13, 57, 25, 17, 1, 90, 3

The first step now is to create a heap from the array elements. For this imagine a binary tree that can be built from the array elements (Figure 1.4.5). The zeroth element would be the root element and left and right child of any element k[i] would be at k [2*i + 1] and k[2*i + 2] respectively. Note that while writing the program we do not physically construct this binary tree by establishing the link between the nodes. Instead, we imagine this tree and then readjust the array elements to form a heap.

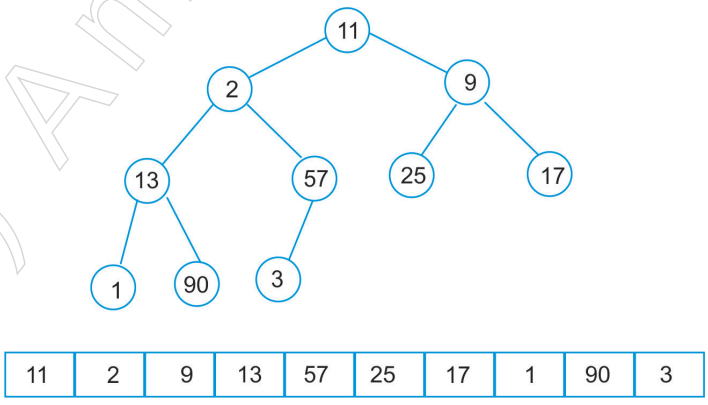


Figure: Array and its equivalent binary tree

Notes

Now the heap is built from the binary tree. The heap and the array are shown in Figure 1.3.

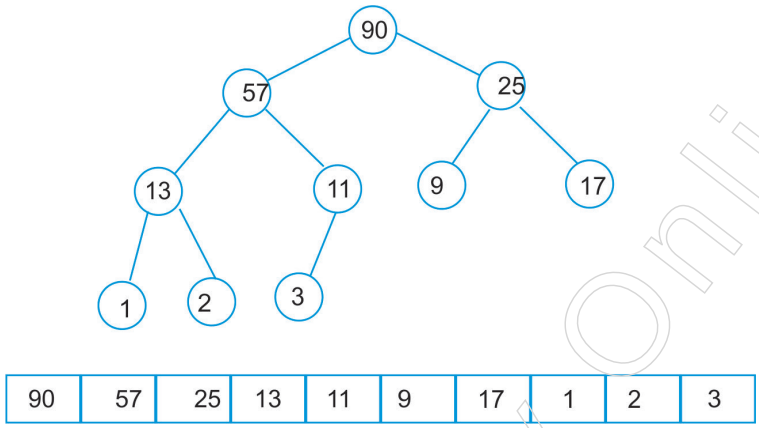


Figure: Heap build from a binary tree

Now the root element 90 is moved to the last location by exchanging it with 3. Finally, 90 is eliminated from the heap by reducing the maximum index value of the array by 1. The balance elements are then rearranged into heap. The heap and array are shown in Figure 1.4.

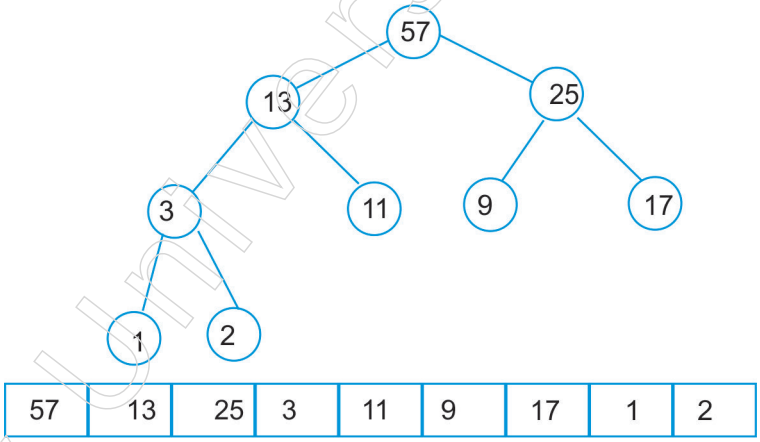


Figure: Heap after eliminating root element 90.

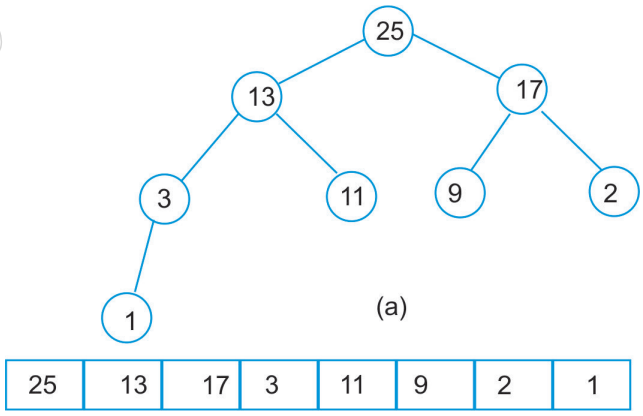
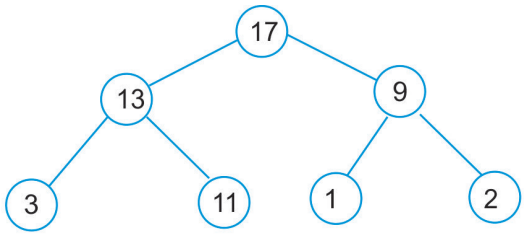


Figure: Heap after eliminating root element 57

Similarly, one by one the root element of the heap is eliminated Figure (a) to (h) show the heap and the array after each elimination.

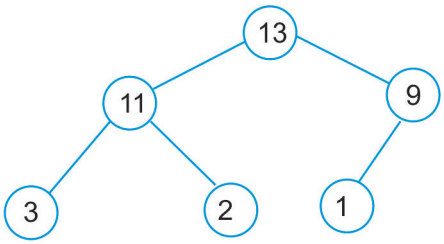
Notes



(b)

17	13	9	3	11	1	2	25	57	90

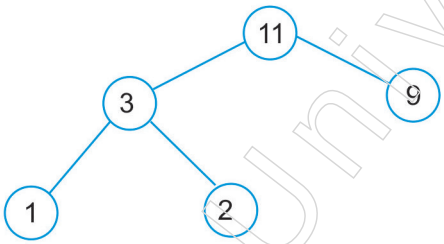
Figure (b)



©

13	11	9	3	2	1	17	25	57	90

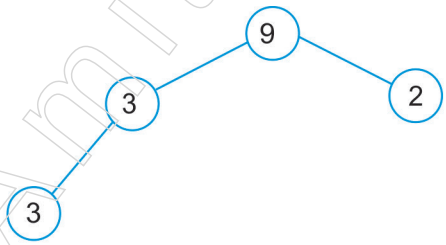
Figure (c)



(d

11	3	9	1	2	13	17	25	57	90

Figure (d)

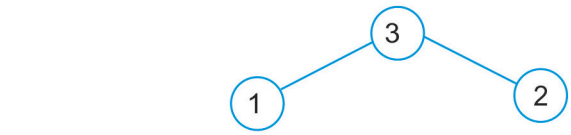


(e)

9	3	2	1	11	13	17	25	57	90

Figure (e)

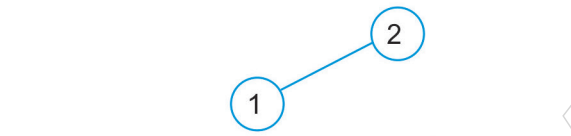
Notes



(f)

3	1	2	9	11	13	17	25	57	90

Figure (f)



(g)

2	1	3	9	11	13	17	25	57	90

Figure (g)

(h)

1	2	3	9	11	13	17	25	57	90

Figure (h)

Figure : Heap on eliminating successive root elements.

Complexity

The complexity of the heap sort algorithm is tabulated in the below mentioned Table:

Algorithm	Worst Case	Average Case	Best Case
Heap Sort	O (n log n)	O (n log n)	O (n log n)

Complexity of heap sort

The following program implements the heap sort algorithm:

```
#include <stdio.h>
#include <conio.h>
void makeheap (int [ ], int);
void heapsort (int [ ], int );
void main()
{
int k[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3};
int i;
```

Notes

```
clrscr();
printf ("\nHeap Sort");
makeheap (k, 10);
printf ("\nBefore Sorting:\n");
for (i=0; i<= 9; i++)
    printf ("%d\t", k[i]);
heapsort (k, 10);
printf ("\nAfter Sorting:\n");
for (i=0; i<= 9; i++)
    printf ("%d\t", k[i]);
getch();
}
/*creates heap from the tree*/
void makeheap (int x[ ], int n)
{
    int i, val, s, f;
    for (i=1; i<n; i++)
   {
val=x[i]:
s = i;
f=(s-1)/2;
       while (s> 0 && x[f] < val)
         {
x[s] = x[f];
s=f;
f=(s-1)/2;
         }
x[s] = val;
    }
}
/* sorts heap */
void heapsort (int x[ ], int n)
int i, s, f, ivalue;
for (i =n-1; i >0; i--)
{
  ivalue = x[i];
   x[i] = x[0];
   f = 0;
if (i == 1)
    s = -1;
else
```

Notes

```
    s = 1;
  if (i > 2 && x[2] > x[1])
     s=2;
  while (s>= 0 && ivalue < x[s])
  {
     x[f] =x[s];
     f=s;
     s=2*f+1;
  if (s+1 <i-1 && x[s] <x[s +1])
     s++;
  if (s> i-1)
    s = -1;
  }
        x[f] = ivalue;
     }
  }
```

Output:

Heap Sort.

Before Sorting:

90 57 25 13 11 9 17 1 2 3

After Sorting:

1 2 3 9 11 13 17 25 57 90

Here, from main() we have called two functions-makeheap() and heapsort(). As the name suggests, makeheap () function is used to create the heap from the tree that can be built from the array k. It receives two parameters: the base address of the array and the number of elements present in the array. Here, data of each element is compared with its child's data and parent and the child data is swapped if required. The function heapsort () is called by passing two arguments: the base address of the array and the number of elements present in the heap. This function sorts the heap by eliminating the root elements one by one and moving them to the end of the array.

1.4.6 Quick Sort

Quick sort is a very popular sorting method. The name comes from the fact that, in general, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small arrays than one larger one. The basic strategy of quick sort is to divide and conquer.

If you were given a large stack of papers bearing the names of the students to sort them by name, you might use the following approach. Pick a splitting value, say L (known as pivot element) and divide the stack of papers into two piles, A-L and M-Z (note that the two piles will not necessarily contain the same number of papers). Then take the first pile and sub-divide it into two piles, A-F and G-L. The A-F pile can

Notes

be further broken down into A-C and D-F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M-Z pile. Eventually all the small sorted piles can be stacked one on top of the other to produce an ordered set of papers. This strategy is based on recursion-on each attempt to sort the stack of papers the pile is divided and then the same approach is used to sort each smaller pile (a smaller case).

Quick sort is also known as partition exchange sort. The quick sort procedure can be explained with the help of Figure 1.1. In Figure 1.1 the element that is indicated by '*'is the pivot element and the element that is indicated by '-' is the element whose position is finalised.

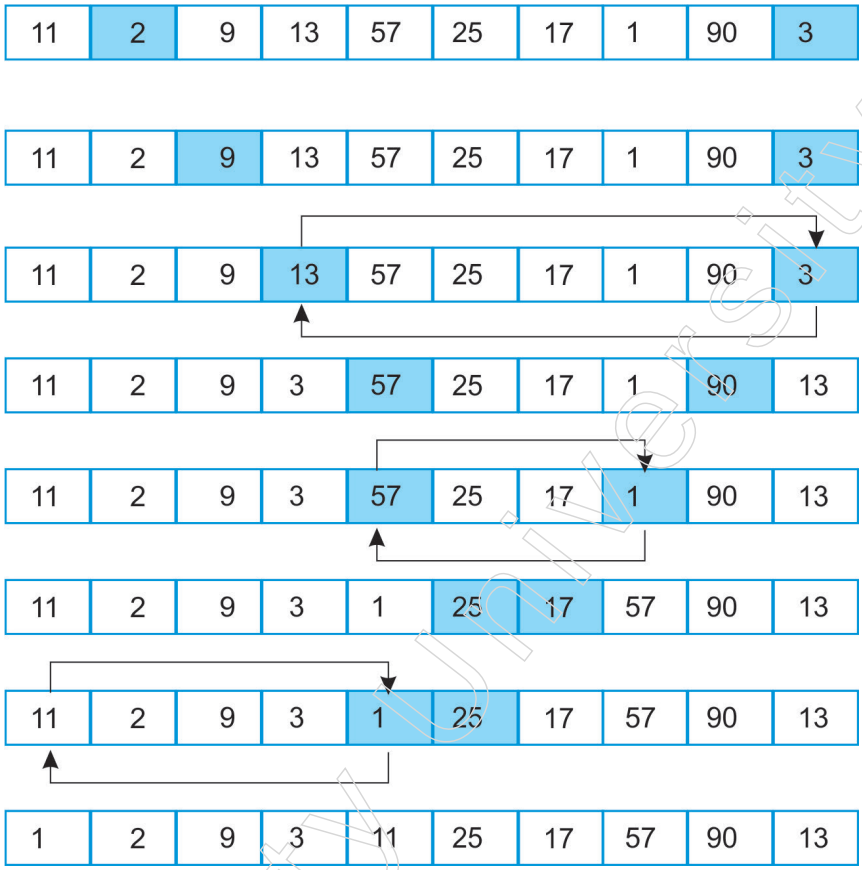


Figure : Quick sort

Suppose an array k consists of 10 distinct elements. The quick sort algorithm works as follows:

a) In the first iteration, we will place the 0th element 11 at its final position and divide the array. Here, 11 is the pivot element. To divide the array, two index variables, p and q, are taken. The indexes are initialised in such a way that, p refers to the 1st element 2 and q refers to the (n-1)th element 3.

b) The job of index variable p is to search for an element that is greater than the value at 0th location. So, p is incremented by one till the value stored at p is greater than 0h element. In our case it is incremented till 13, as 13 is greater than 11.

c) Similarly, q needs to search for an element that is smaller than the 0th element. So, q is decremented by one till the value stored at q is smaller than the value at 0th location. In our case q is not decremented because 3 is less than 11.

Notes

d) When these elements are found they are interchanged. Again, from the current positions p and q are incremented and decremented respectively and exchanges are made appropriately if desired.

e) The process ends whenever the index pointers meet or crossover. In our case, they are crossed at the values 1 and 25 for the indexes q and p respectively. Finally, the 0th element 11 is interchanged with the value at index q, i.e., 1. The position q is now the final position of the pivot element 11.

f) As a result, the whole array is divided into two parts. Where all the elements before 11 are less than 11 and all the elements after 11 are greater than 11.

g) Now the same procedure is applied for the two sub-arrays. As a result, at the end when all sub-arrays are left with one element, the original array becomes sorted.

Here, it is not necessary that the pivot element whose position is to be finalized in the first iteration must be the 0th element. It can be any other element as well.

Complexity

The complexity of the quick sort algorithm is tabulated in the below mentioned Table:

Algorithm	Worst Case	Average Case	Best Case
Quick Sort	O(n2)	Log2 n	Log2 n

Complexity of quick sort

The program given below implements the Quick sort algorithm:

```
#include <stdio.h>
#include <conio.h>
void quicksort(int*, int, int);
int split (int*, int, int);
void main()
{
int k[10] = {11, 2, 9, 13, 57, 25, 17, 1, 90,3};
int i;
clrscr();
printf ("Quick sort.\n");
printf ("Array before sorting:\n");
for (i=0; i<=9; i++)
printf("%d\t",k[i]);
quicksort (k,0,9);
printf ("\nArray after sorting:\n");
for (i=0;i<= 9; i++)
printf ("%d\t", k[i]);
getch();
}
void quicksort (int a[ ], int lower, int upper)
{
```

Notes

```
int i;
 if (upper > lower)
{
i = split (a, lower, upper);
quicksort (a, lower, i-1);
quicksort (a, i+1, upper);
}
}
int split (int a[ ], int lower, int upper)
{
int i, p, q, t;
p = lower + 1;
q=upper;
i = a[lower];
while (q>= p)
{
  while (a[p] <i)
   p++;
  while (a[q]> i)
   q++;
  if (q>p)
  {
t = a[p];
   a[p] = a[q];
   a[q] = t;
   }
}
    t= a[lower];
   a[lower] = a[q];
   a[q] = t;
   return q;
}
```

Output:

Quick sort.

Array before sorting:

11 2 9 13 57 25 17 1 90 3

Array after sorting:

1 2 3 9 11 13 17 25 57 90

The arguments being passed to the function quicksort() would reflect the part of the array that is being currently processed. We will pass the first and last indexes that

Notes

define the part of the array to be processed on this call. The initial call to quicksort() would contain the arguments 0 and 9, since there are 10 integers in our array.

In the capacity quicksort(), a condition is checked whether upper is more noteworthy than lower. Assuming the condition is fulfilled, just the exhibit will be parted into two sections, in any case, the control will essentially be returned. To part the cluster into two sections the capacity split() is called.

In the function split(), to start with the two variables p and q are taken which are assigned with the values lower + 1 and upper. Then a while loop is executed that checks whether the indexes p and q cross each other. If they are not crossed then inside the while loop two more nested while loops are executed to increase the index p and decrease the index q to their appropriate places. Then it is checked whether q is greater than p. If so, then the elements present at p^{th} and q^{th} positions are interchanged.

Finally, when the control returns to the function quicksort() two recursive calls are made to function quicksort(). This is done to sort the two split sub-arrays. As a result after all the recursive calls when the control reaches the function main() the arrays become sorted.

The efficiency of the quick sort method can be improved by:

i) Choosing a better pivot element.

ii) Using a better algorithm for small sub-lists.

iii) Eliminating recursion.

Summary

- A data structure is a method of organising and storing data in a computer, enabling efficient access and usage. It defines the relationship between data elements and the operations that can be performed on them.
- A stack is an abstract data type with limited, predefined capabilities. It is a fundamental data structure that allows elements to be added and removed in a specific order.
- The Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one ring. At first, every one of the circles are set on one rod or tower, one over the other in a rising request of size like a cone-shaped tower.
- Like stack data structure, Queue is either an abstract data form or a linear data structure in which the first element is added from one end called REAR (also called the tail) and the existing element is removed from the other end called FRONT (also called the head).
- At the first index of the array, initially the head(FRONT) and the tail(REAR) of the queue points (starting the index of an array from 0).
- The Doubly Linked List is a form of linked list in which there are two links to each node apart from storing its data.
- Time Complexity is a way of representing the amount of time the program requires to run until it is completed.
- The Divide and Conquer Algorithm is a problem-solving approach that involves breaking down a main problem into smaller subproblems, solving them independently, and then merging the solutions to obtain the solution for the original problem.
- Sorting involves reordering a given array or list of elements based on a defined comparison operator, determining the new arrangement of elements within the data structure.

Notes

- Bubble sorting is a basic algorithm for sorting. This sorting algorithm is a comparison-based algorithm that compares each pair of adjacent elements and, if they are not in order, swaps the elements.
- Selection sort is a basic algorithm for sorting. This sorting algorithm is a comparison-based in-place algorithm in which the list is split into two parts, the left-end sorted part, and the right-end unsorted part.

Glossary

- Array: A collection of elements of the same type stored in contiguous memory locations.
- Linked List: A collection of elements linked together by pointers, allowing dynamic insertion and deletion.
- Queue: A First-In-First-Out (FIFO) structure where elements are added at the end and removed from the beginning.
- Stack: A Last-In-First-Out (LIFO) structure where elements are added and removed from the top.
- Tree: A hierarchical structure where each node can have multiple child nodes.
- Graph: A collection of nodes connected by edges, representing relationships between data elements.
- Hash Table: A data structure that uses a hash function to map keys to values, allowing for fast lookup and insertion.
- Push(): Adding an element to the stack.
- Pop(): Removing (accessing) an element from the stack.
- Peek(): Retrieving the top element of the stack without removing it.
- A Priori Analysis- This is an algorithm theoretical analysis. An algorithm's efficiency is calculated by assuming that all other variables are constant and have no effect on the implementation, such as processor speed.
- A Posterior Analysis- This is an algorithm's empirical analysis. A programming language is used to implement the chosen algorithm.
- Time Factor: Time is calculated by counting the number of main operations in the sorting algorithm, such as comparisons.
- Space Factor: Space is determined by counting the algorithm's maximum memory space available.
- Memory Usage: Bubble sort requires no additional memory, while merge sort necessitates extra space for its operations.
- Memory Constraints: In scenarios where memory is limited but sufficient processing power is available, bubble sort may be a preferable choice despite its higher time complexity.

Check Your Understanding

1. What is the primary characteristic of a stack data structure?

   a) FIFO (First In First Out) b) LIFO (Last In First Out)

   c) FILO (First In Last Out) d) Ordered Insertion

2. In the Tower of Hanoi problem, how many steps are required to solve a puzzle with 3 disks?

Notes

a) 6 b) 7 c) 8 d) 9

3. Which type of linked list allows traversal in both forward and backward directions?
   a) Single Linked List b) Doubly Linked List
   c) Circular Linked List d) Linear Linked List
4. What does the Big O notation represent in algorithm analysis?
   a) Upper bound of running time b) Lower bound of running time
   c) Both upper and lower bounds of running time
   d) Average case running time
5. Which method is used to solve recurrences by making an educated guess and then verifying it mathematically?
   a) Recurrence Tree Method b) Master Theorem
   c) Substitution Method d) Dynamic Programming

Exercise

1. Describe the role of indexing in big data structures.
2. Explain the concept of algorithm complexity and its importance.
3. What are the three main steps involved in a divide and conquer algorithm?
4. What is the time complexity of bubble sort in the worst case?
5. Explain how heap sort organises data during the sorting process.
6. Why is it important to choose the right sorting algorithm for a given problem?

Learning Activities

1. Organise a workshop where students can work in groups to implement and analyse different big data structures (e.g., Hadoop HDFS, Apache Spark RDD). Each group will present their findings, focusing on the efficiency and practical applications of the data structures they studied.
2. Conduct a coding challenge where students are given a set of problems that can be solved using the divide and conquer approach. Problems might include implementing algorithms such as MergeSort, QuickSort, and solving the Closest Pair of Points problem. Students will code their solutions and then explain the efficiency and logic behind their implementations.

Check Your Understanding Answer

1. b) 2. b) 3. b) 4. a)
5. c)

Further Readings And Bibliography

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Aggarwal, C. C. (2016). Data Mining: The Textbook. Springer.
3. Goodrich, M. T., & Tamassia, R. (2014). Algorithm Design and Applications. Wiley.
4. Heidari, M. (2020). Big Data Structures and Algorithms for Massive Data. CRC Press.
5. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Module - II: Tree and Graphs

Learning Objectives

At the end of this module, you will able to:

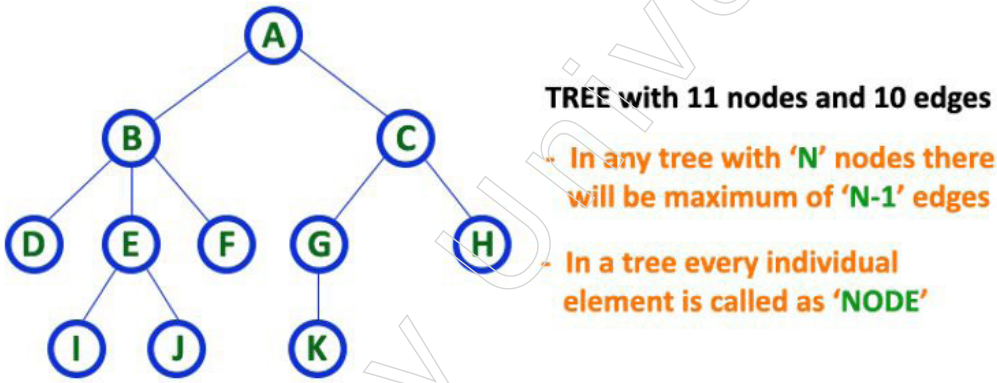
- Implement various tree traversal algorithms (in-order, pre-order, post-order).
- Analyse the performance and complexity of tree operations (insertion, deletion, searching).
- Explore different types of graphs (directed, undirected, weighted, unweighted).
- Implement and understand graph traversal algorithms (BFS, DFS).
- Analyse and solve problems using graph algorithms (shortest path, spanning tree).

Introduction

A tree is an assortment of components called nodes. Every node contains some worth or component. We will use the term nodes, instead of the vertex with a binary tree.

2.1 Tree

Nodes are the primary segment of any tree structure. It stores the real information alongside connections to different nodes.



2.1.1 Basic Terminology and Applications

- Path: A path is an arrangement of edges between nodes.
- Root: The root is the unique node from which all other nodes "descend." Each tree has a single root node.
- Parent of Node n: The parent of node n is the unique node with an edge to node n and which is the first node on the path from n to the root.
- Note: The root is the only node with no parent. Except for the root, every node has exactly one parent.
- Offspring of Node n: An offspring of node n is a node for which node n is the next node on the path to the root node. More formally, if the last edge on the path from root r of tree T to node x is (y, x), then node y is the parent of node x and node x is an offspring of node y.

Notes

Note: Every node may have at least 0 children.

- Siblings: Nodes with the same parent are siblings.
- Progenitor of Node n: Any node y on the (unique) path from root r to node n is a progenitor of node n. Every node is a progenitor of itself.
- The Immediate Progenitor of n: A valid progenitor of n is any node y such that node y is a progenitor of node n and y is not the same node as n.
- Descendant of Node n: Any node y for which n is a progenitor of y. Every node is a descendant of itself.
- Valid Descendant of Node n: A valid descendant of n is any node y for which n is a progenitor of y and y is not the same node as n.
- Subtrees of Node n: Subtrees of node n are the trees whose roots are the children of n.
- Level of Node n: The level of node n is the number of children node n has.
- Leaf Node: A leaf node is a node without any children.
- Outer Node: An outer node is a node without any children (same as a leaf node).
- Inner Node: An inner node is a non-leaf node.
- The Depth of Node n: The depth of node n is the length of the path from root to node n.
- Level d: Level d is the nodes at depth d.
- The Height of Tree T: The height of tree T is the largest depth of any node in T. We generally consider height relating to trees and depth relating to nodes, however, for full trees, the terms are sometimes used interchangeably.
- Ordered Tree: An ordered tree is a tree in which the children of each node are ordered.
- Full k-ary Tree: A full k-ary tree is a tree where all leaves have the same depth and all inner nodes have degree k. (The Cormen et al. Algorithm text calls this complete.)

Terminology

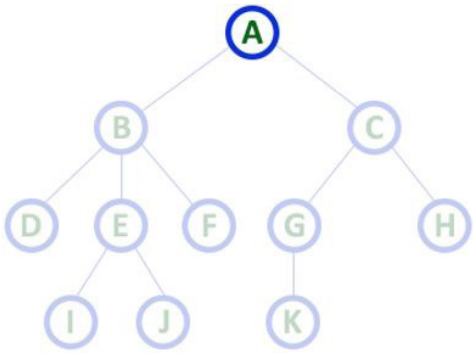
In a tree data structure, we use the following terminology:

1. Root

   In a tree information structure, the primary hub is called Root Node. Each tree should have a root hub. We can say that the root hub is the beginning of the tree information structure. In any tree, there should be just one root hub. We never have numerous root hubs in a tree.

   A root node is very much like any node, in that it is important for a data structure that comprises at least one field with connections to a different node and contains an information field; it just turns out to be the principal node. In such a manner, any node can be a root node corresponding to itself and its youngsters if that part of the tree is impartially chosen.

Notes

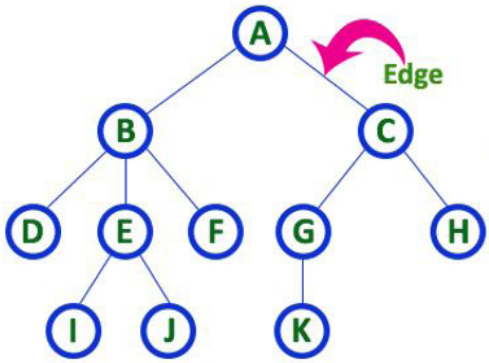


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

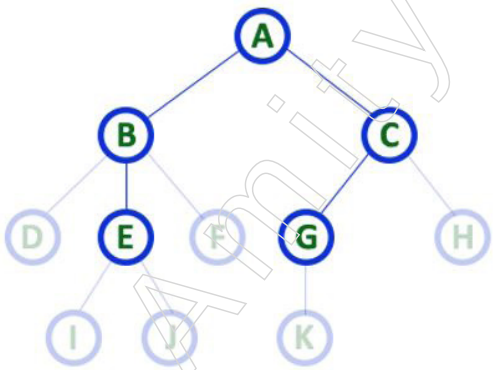
In a tree information structure, the associating join between any two hubs is called EDGE. In a tree with 'N' number of hubs, there will be a limit of 'N-1' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree information structure, the hub which is an archetype of any hub is called a PARENT NODE. In straightforward words, the hub which has a branch from it to some other hub is known as a parent hub. Parent hub can likewise be characterised as "The hub which has kid/kids".



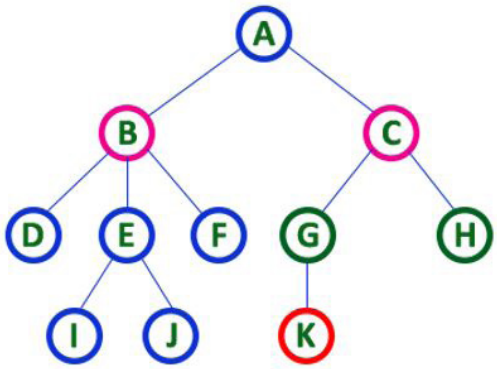
Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree information structure, the hub which is an archetype of any hub is called a PARENT NODE. In straightforward words, the hub which has a branch from it to some other hub is known as a parent hub. Parent hub can likewise be characterised as "The hub which has kid/kids".

Notes

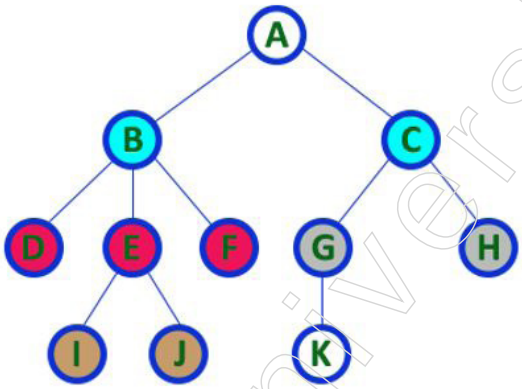


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree information structure, hubs that have a place with the same Parent are called SIBLINGS. In straightforward words, the hubs with similar parents are called Sibling hubs.



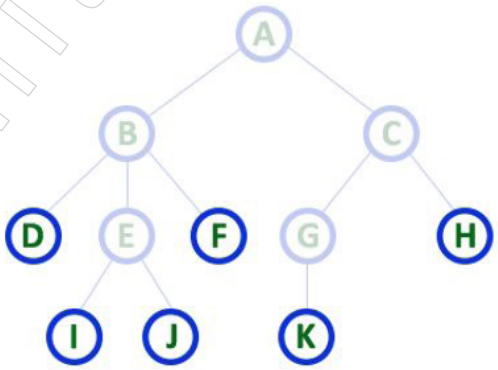
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree information structure, the hub which doesn't have a youngster is called LEAF Node. In basic words, a leaf is a hub with no kid.

In a tree information structure, the leaf hubs are likewise called External Nodes. The outer hub is additionally a hub with no kid. In a tree, leaf hub is additionally called as 'Terminal' hub



Here D, I, J, F, K & H are Leaf nodes

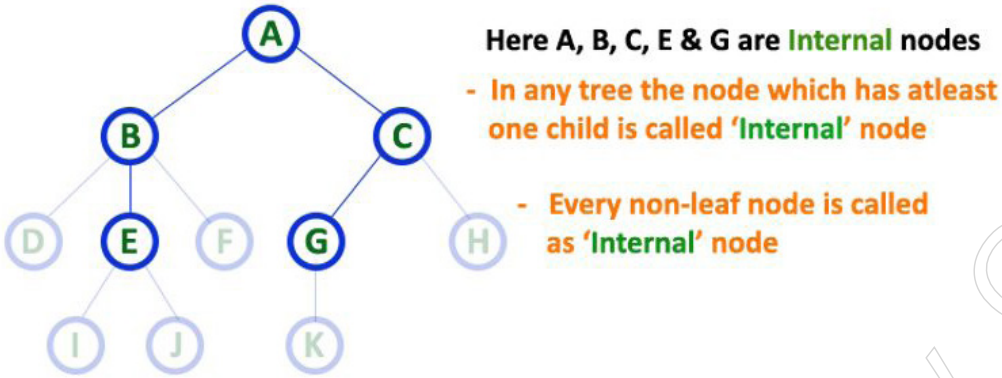
- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

In a tree information structure, the hub which has at least one youngster is called INTERNAL Node. In basic words, an inward hub is a hub with at least one kid.

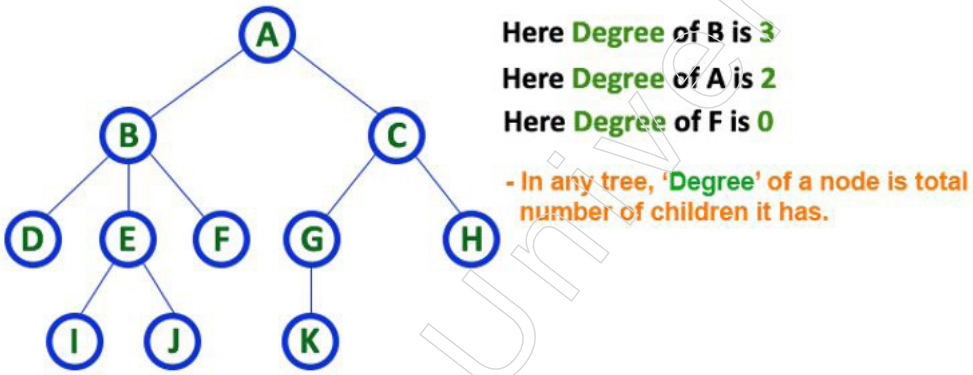
Notes

In a tree information structure, hubs other than leaf hubs are called Internal Nodes. The root hub is additionally supposed to be an Internal Node if the tree has more than one hub. Inside hubs are likewise called 'Non-Terminal' hubs.



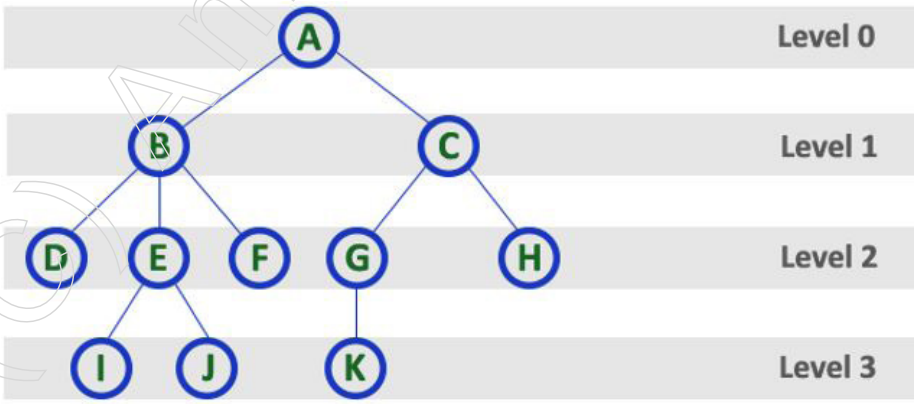
8. Degree

In a tree information structure, the complete number of offspring of a hub is called the DEGREE of that Node. In basic words, the Degree of a hub is all outnumber of youngsters it has. The most extensive level of a hub among all the hubs in a tree is called as 'Level of Tree'



9. Level

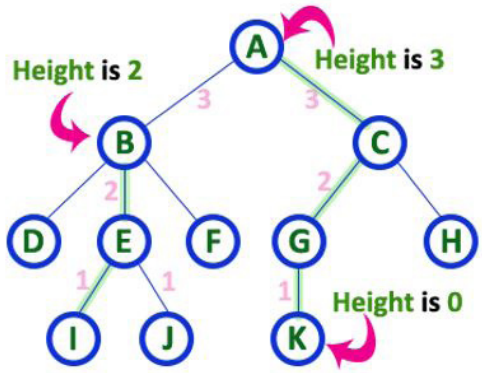
In a tree information structure, the root hub is supposed to be at Level 0 and the offspring of the root hub is at Level 1 and the offspring of the hubs which are at Level 1 will be at Level 2, etc... In basic words, in a tree, each progression through and through is called a Level and the Level tally begins with '0' and increases by one at each level (Step).



Notes

10. Height

In a tree information structure, the complete number of edges from the leaf hub to a specific hub in the longest way is called the HEIGHT of that Node. In a tree, the stature of the root hub is supposed to be the tallness of the tree. In a tree, the tallness of all leaf hubs is '0'.

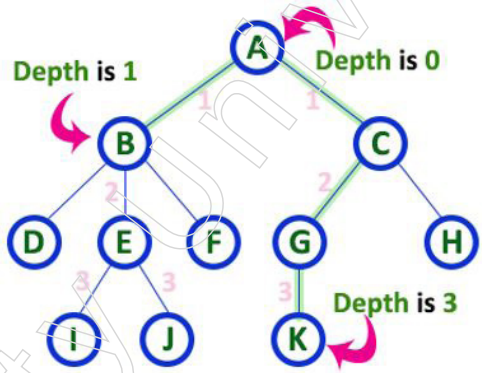


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree information structure, the complete number of edges from the root hub to a specific hub is called the DEPTH of that Node. In a tree, the absolute number of edges from the root hub to a leaf hub in the longest way is supposed to be the Depth of the tree. In basic words, the most noteworthy profundity of any leaf hub in a tree is supposed to be the profundity of that tree. In a tree, the profundity of the root hub is '0'

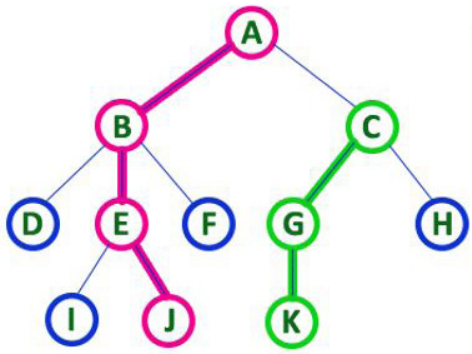


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree information structure, the succession of Nodes and Edges starting with one hub then onto the next hub is called PATH between those two Nodes. Length of a Path is the absolute number of hubs in that way. In the underneath model the way A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

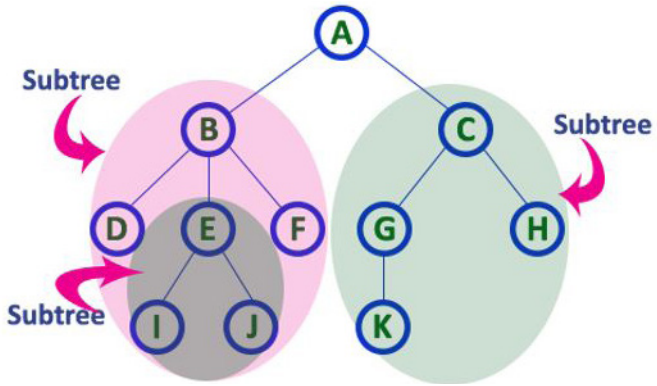
Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

Notes

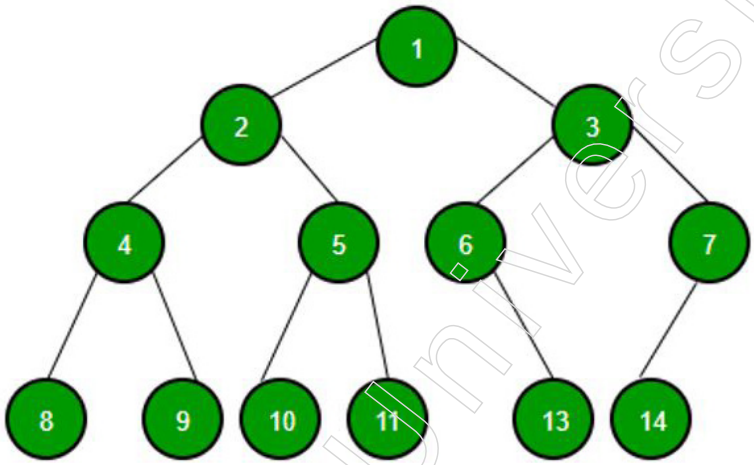
13. Sub Tree

In a tree information structure, every youngster from a hub shapes a subtree recursively. Each kid hub will shape a subtree on its parent hub.



Application of Tree:

Trees are hierarchical structures having a single root node.



Some applications of the trees are:

- XML Parser utilises a tree algorithm.
- The choice-based calculation is utilised in AI which works upon the calculation of trees.
- Information bases likewise utilise tree information structures for ordering.
- Area Name Server(DNS) likewise utilises tree structures.
- Record pilgrim/my PC of versatile/any PC
- BST utilised in PC Graphics
- Posting inquiries on sites like Quora, the remarks are offspring of inquiries

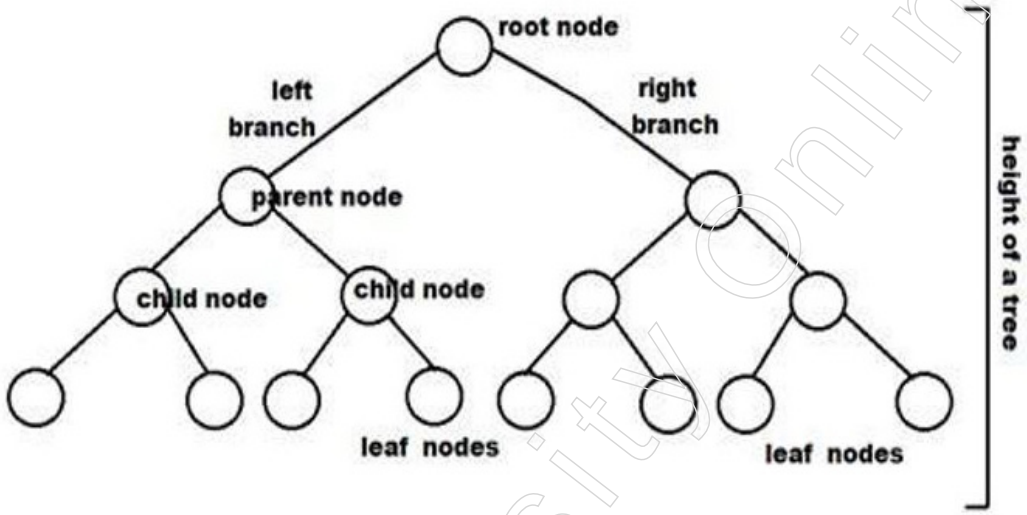
2.1.2: Binary Trees

What is Binary Tree Data Structure?

A binary tree is a tree-type non-straight information structure with a limit of two kids for each parent. Each hub in a binary tree has a left and right reference alongside the information component. The hub at the highest point of the order of a tree is known as the root hub. The hubs that hold other sub-hubs are the parent hubs.

Notes

A parent hub has two youngster hubs: the left kid and the right kid. Hashing, directing information for network traffic, information pressure, planning binary stacks, and twofold hunt trees are a portion of the applications that utilise a binary tree.



Terminologies Associated with Binary Trees and Types of Binary Trees

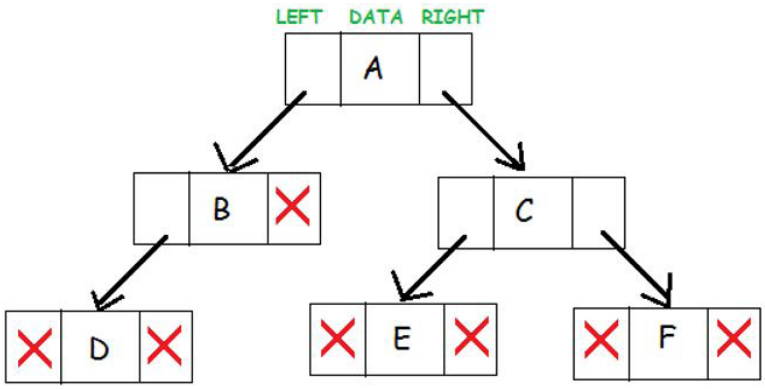
- Hub: It addresses an endpoint in a tree.
- Root: A tree's highest hub.
- Parent: Each hub (aside from the root) in a tree that has in any event one sub-hub of its own is known as a parent hub.
- Kid: A hub that straightway came from a parent hub while moving away from the root is the youngster hub.
- Leaf Node: These are outside hubs. They are the hubs that have no youngsters.
- Inside Node: As the name proposes, these are inward hubs with at any rate one youngster.
- The profundity of a Tree: The quantity of edges from the tree's hub to the root is.
- The stature of a Tree: It is the number of edges from the hub to the most profound leaf. The tree stature is additionally viewed as the root tallness.
- As you are currently acquainted with the phrasings related to the binary tree and the sorts of the binary tree, the time has come to comprehend the binary tree parts.

Binary Tree Components

There are three binary tree parts. Each twofold tree hub has these three parts related to it. It turns into a fundamental idea for developers to comprehend these three binary tree parts:

- Information component
- Pointer to left subtree
- Pointer to right subtree
- kinds of the binary tree model

Notes



Source

These three twofold tree parts address a hub. The information lives in the centre. The left pointer focuses on the kid hub, framing the left sub-tree. The correct pointer focuses on the kid hub at its right, making the privilege subtree.

Peruse: Top Guesstimate Questions and Informative Methods for Data Science

Kinds of Binary Trees

There are different sorts of binary trees, and every one of these binary tree types has remarkable qualities. Here is every one of the binary tree types in detail:

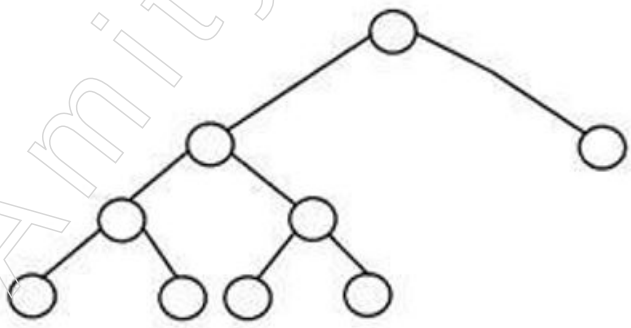
1. Full Binary Tree

   It is an exceptional sort of a binary tree that has either zero youngsters or two kids. It implies that all the hubs in that binary tree ought to either have two kid hubs of its parent hub or the parent hub is itself the leaf hub or the outer hub.

   As such, a full binary tree is a remarkable binary tree where each hub aside from the outside hub has two youngsters. At the point when it holds a solitary youngster, a particularly binary tree won't be a full binary tree. Here, the amount of leaf hubs is equivalent to the number of inner hubs in addition to one. The condition resembles L=I+1, where L is the number of leaf hubs, and I is the number of interior hubs.

   Here is the design of a full binary tree:

   sorts of binary tree - full binary tree

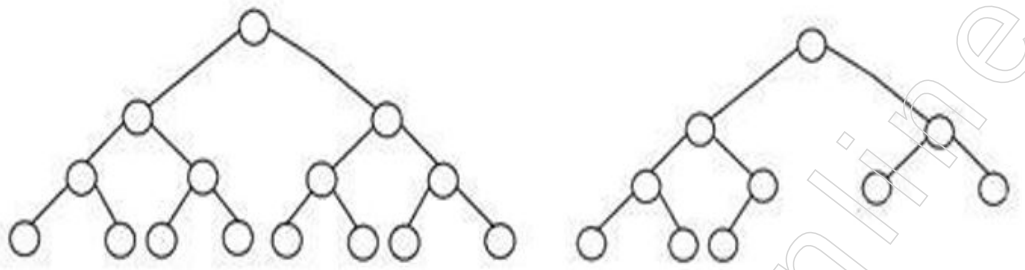


2. Complete Binary Tree

   A total binary tree is another particular kind of twofold tree where all the three levels are filled with hubs, aside from the most reduced level of the tree. Additionally, in the last or the most reduced level of this binary tree, each hub ought to perhaps dwell on the left side. Here is the design of a total binary tree:

Notes

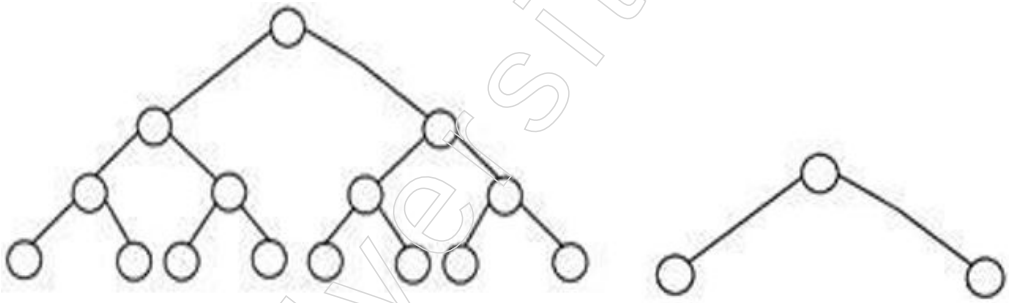
sorts of twofold tree - complete binary tree



3. Wonderful Binary Tree

A binary tree is supposed to be 'awesome' if all the inward hubs have carefully two youngsters, and each outer or leaf hub is at a similar level or same profundity inside a tree. An ideal binary tree having tallness 'h' has 2h – 1 hub. Here is the design of an ideal binary tree:

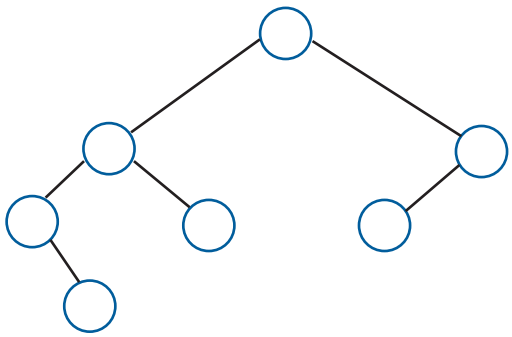
kinds of binary tree - amazing tree



4. Adjusted Binary Tree

A binary tree is supposed to be 'adjusted' if the tree tallness is O(logN), where 'N' is the number of hubs. In a reasonable twofold tree, the stature of the left and the privilege subtrees of every hub ought to fluctuate by all things considered one. An AVL Tree and a Red-Black Tree are some regular instances of information structure that can produce a fair binary inquiry tree. Here is an illustration of a fair binary tree:

kinds of binary tree - adjusted binary tree

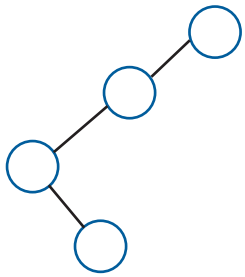


5. Degenerate Binary Tree

A binary tree is supposed to be a savage twofold tree or obsessive binary tree if each inside hub has just a solitary kid. Such trees are like a connected rundown execution astute. Here is an illustration of a savage binary tree:

degenerate binary tree

Notes



Advantages of a Binary Tree

Trees are so valuable and as often as possible utilised because they have some intense benefits:

- Trees show structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minimum effort.
- The inquiry activity in a twofold tree is quicker when contrasted with different trees
- Just two crossings are sufficient to give the components in arranged request
- It is not difficult to get the greatest and least components
- Diagram crossing likewise utilises binary trees
- Changing over various postfix and prefix articulations are conceivable utilising twofold trees

The doubletree is quite possibly the most broadly utilised tree in the information structure. Every one of the paired tree types has its one-of-a-kind highlights. These information structures have explicit prerequisites in applied software engineering. We trust this article about kinds of parallel trees was useful. upgrade offers different courses in information science, AI, large information, and then some.

2.1.3 Binary Search Trees

What is a Binary Search Tree?

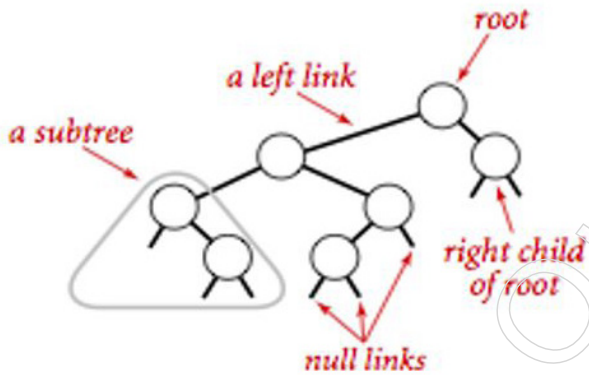
A BST is viewed as an information structure composed of hubs, as Linked Lists. These hubs are either invalid or have references (joins) to different hubs. These 'other' hubs are youngster hubs, called left hub and right hub. Hubs have values. These qualities figure out where they are put inside the BST.

Additionally to a connected rundown, every hub is referred to by just a single hub, its parent (except the root hub). So we can say that every hub in a BST is in itself a BST. From further down the tree, we arrive at another hub and that hub has a left and a right. At that point contingent upon what direction we go, that hub has a left and a privilege, etc.

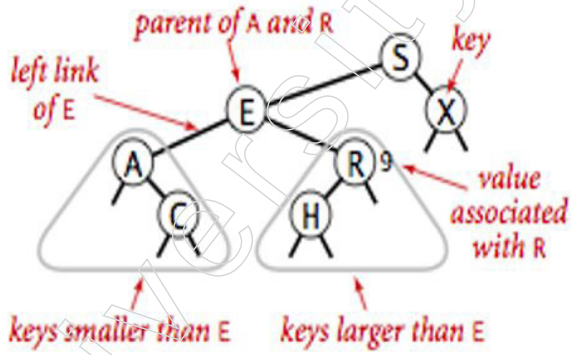
1. The left hub is consistently more modest than its parent.
2. The correct hub is consistently more noteworthy than its parent.
3. A BST is viewed as adjusted if each level of the tree is completely loaded up except for the last level. On the last level, the tree is filled left to right.

Notes

4. A Perfect BST is one in which it is both complete (all youngster hubs are on a similar level and every hub has a left and a correct kid hub).



Anatomy of a binary tree



Anatomy of a binary search tree

Why do we need a Binary Search Tree?

The two central points that make a twofold hunt tree an ideal answer for any genuine issues are Speed and Accuracy.

Because of the way that the twofold hunt is in a branch-like organisation with parent-kid relations, the calculation knows in which area of the tree the components should be looked at. This abates the quantity of key-esteem examinations the program needs to make to find the ideal component.

Moreover, if the component to be looked through is more noteworthy or not exactly the parent hub, the hub realises which tree side to look for. The explanation is, the left sub-tree is consistently lesser than the parent hub, and the correct sub-tree has values consistently equivalent to or more prominent than the parent hub.

BST is regularly used to actualize complex quests, powerful game rationales, auto-complete exercises, and designs.

The calculation effectively underpins tasks like hunt, embed, and erase.

2.1.4 Optimal Binary Search Tree

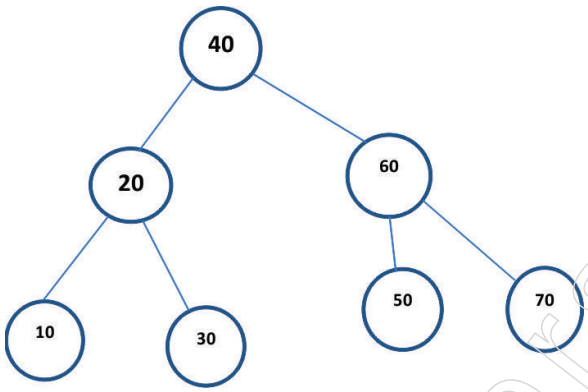
As we realise that in the binary search tree, the hubs in the left subtree have lesser worth than the root hub and the hubs in the privilege subtree have more noteworthy worth than the root hub.

Notes

We know the vital upsides of every hub in the tree, and we likewise know the frequencies of every hub as far as looking through implies how long is needed to look through a hub. The recurrence and key worth decide the general expense of looking through a hub. The expense of looking is a vital factor in different applications. The general expense of looking through a hub ought to be less. The time needed to look through a hub in BST is more than the fair parallel hunt tree as a reasonable twofold inquiry tree contains a lesser number of levels than the BST. There is one way that can diminish the expense of a double inquiry tree is known as an ideal binary search tree.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

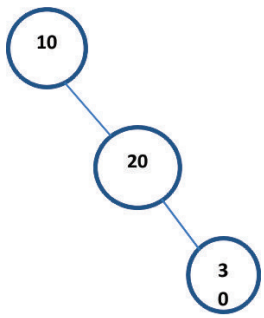
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

At the point when we utilise the above equation, at that point it is tracked down that a complete 5 number of trees can be made.

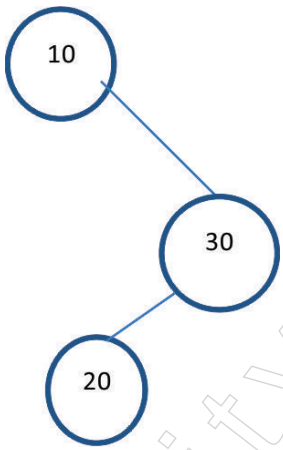
The expense needed for looking through a component relies upon the correlations with be made to look through a component. Presently, we will ascertain the normal expense of season of the above parallel pursuit trees.



Notes

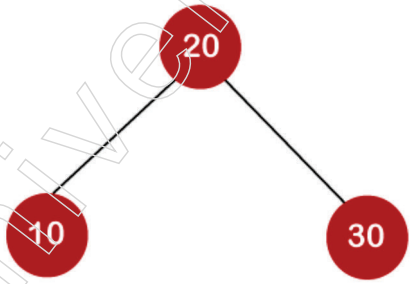
In the above tree, a total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



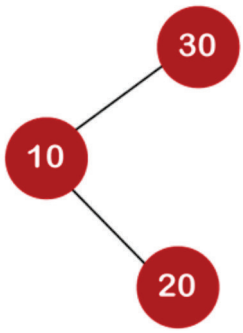
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the average number of comparisons that can be made as:

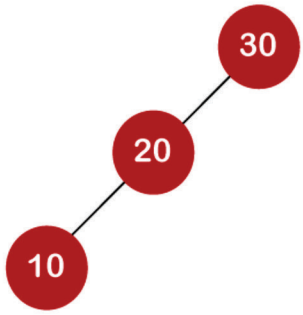
$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

Notes



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3

i \ j	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where j-i is equal to zero.

- When i=0, j=0, then j-i = 0
- When i = 1, j=1, then j-i = 0
- When i = 2, j=2, then j-i = 0
- When i = 3, j=3, then j-i = 0

Notes

- When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

- When j=1, i=0 then j-i = 1
- When j=2, i=1 then j-i = 1
- When j=3, i=2 then j-i = 1
- When j=4, i=3 then j-i = 1

Now to calculate the cost, we will consider only the jth value.

- The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).
- The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).
- The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)
- The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

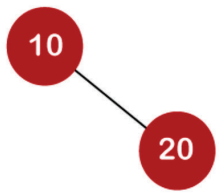
	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2

- When j=2, i=0 then j-i = 2
- When j=3, i=1 then j-i = 2
- When j=4, i=2 then j-i = 2

In this case, we will consider two keys.

- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



- In the first binary tree, cost would be: 4*1 + 2*2 = 8
- In the second binary tree, cost would be: 4*2 + 2*1 = 10
- The minimum cost is 8; therefore, c[0,2] = 8

Notes

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

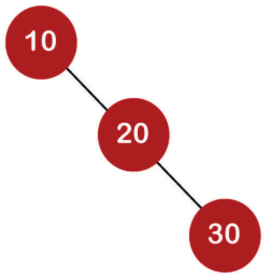
- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:
  - In the first binary tree, cost would be: 1*2 + 2*6 = 14
  - In the second binary tree, cost would be: 1*6 + 2*2 = 10
  - The minimum cost is 10; therefore, c[1,3] = 10
- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:
  - In the first binary tree, cost would be: 1*6 + 2*3 = 12
  - In the second binary tree, cost would be: 1*3 + 2*6 = 15
  - The minimum cost is 12, therefore, c[2,4] = 12

i \ 1	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

Now we will calculate the values when j-i = 3

- When j=3, i=0 then j-i = 3
- When j=4, i=1 then j-i = 3
- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

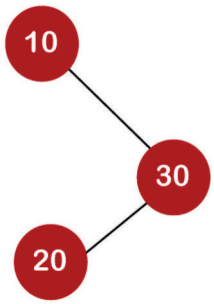
The following are the trees that can be made if 10 is considered as a root node.



Notes

In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

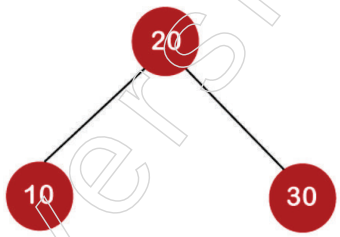
Cost would be: 1*4 + 2*2 + 3*6 = 26



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: 1*4 + 2*6 + 3*2 = 22

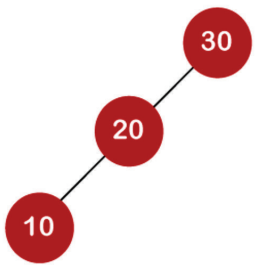
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

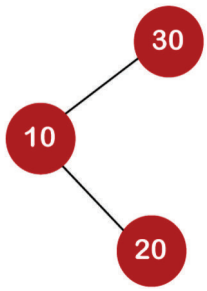
Cost would be: 1*2 + 4*2 + 6*2 = 22

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: 1*6 + 2*2 + 3*4 = 22



Notes

In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

- When i=1 and j=4 then we will consider the keys 20, 30, 40

  c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

  = min{0+12, 2+3, 10+0}+ 11

  = min{12, 5, 10} + 11

  The minimum value is 5; therefore, c[1,4] = 5+11 = 16

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- Now we will calculate the values when j-i = 4

  When j=4 and i=0 then j-i = 4

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then

C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]

= min {0 + 16} + 15= 31

If we consider 20 as the root node then

C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]

= min {8 + 3} + 15

= 26

If we consider 40 as the root node then,

C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]

= min{20 + 0} + 15

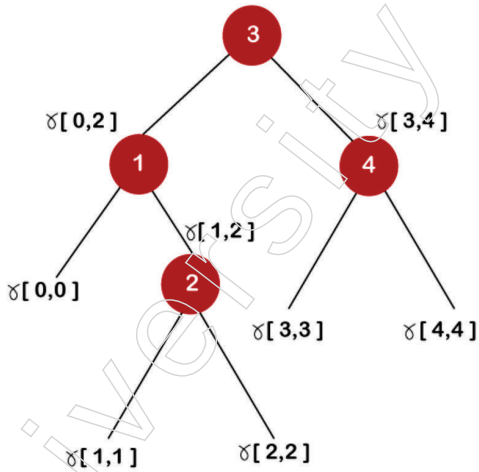
= 35

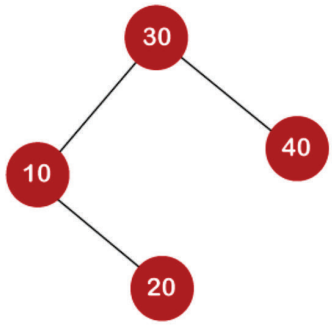
Notes

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary tree can be created as:





General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

2.1.5 Red-Black Trees

Introduction:

A red-dark tree is a sort of self-adjusting double hunt tree where every hub has an additional piece, and that piece is frequently deciphered as the tone (red or dark). These tones are utilised to guarantee that the tree stays adjusted during additions and cancellations. Although the equilibrium of the tree isn't great, it is adequate to decrease the looking through time and keep it around O(log n) time, where n is the absolute number of components in the tree. This tree was developed in 1972 by Rudolf Bayer.

Notes

It should be noticed that as every hub requires just 1 bit of room to store the shading data, these kinds of trees show indistinguishable memory impressions to the work of art (uncolored) parallel pursuit tree.

Decides That Every Red-Black Tree Follows:

- Each hub has a shading either red or dark.
- The base of the tree is consistently dark.
- There are no two neighbouring red hubs (A red hub can't have a red parent or red youngster).
- Each way from a hub (counting root) to any of its relative NULL hubs has a similar number of dark hubs.

Why Red-Black Trees?

The majority of the BST activities (e.g., search, max, min, embed, erase.. and so forth) take O(h) time where h is the tallness of the BST. The expense of these tasks may become O(n) for a slanted Binary tree. If we ensure that the tallness of the tree remains O(log n) after each inclusion and erasure, at that point we can ensure an upper bound of O(log n) for every one of these tasks. The stature of a Red-Black tree is consistently O(log n) where n is the number of hubs in the tree.

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

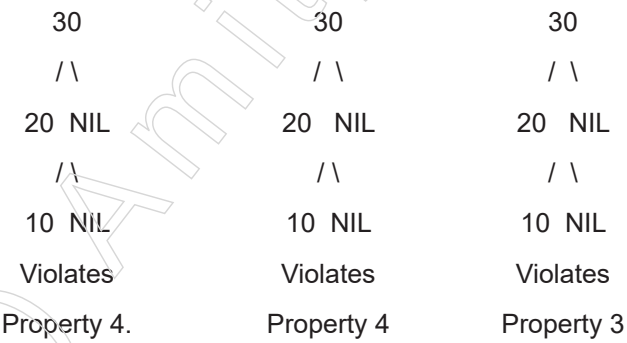
"n" is the total number of elements in the red-black tree.

How does a Red-Black Tree ensure balance?

A basic guide to comprehend adjusting is, a chain of 3 hubs is preposterous in the Red-Black tree. We can attempt any mix of tones and see every one of them disregard Red-Black tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are NOT Red-Black Trees



Following are different possible Red-Black Trees with the above 3 keys

Notes

/ \ / \ / \ / \

NIL NIL NIL NIL NIL NIL NIL NIL

Fascinating focuses about Red-Black Tree:

The dark stature of the red-dark tree is the number of dark hubs away from the root hub to a leaf hub. Leaf hubs are likewise considered dark hubs. In this way, a red-dark tree of stature h has dark tallness >= h/2.

Stature of a red-dark tree with n hubs is h<= 2 log2(n + 1).

All leaves (NIL) are dark.

The dark profundity of a hub is characterised as the number of dark hubs from the root to that hub i.e the number of dark predecessors.

Each red-dark tree is an extraordinary instance of a twofold tree.

Dark Height of a Red-Black Tree:

Dark stature is the number of dark hubs away from the root to a leaf. Leaf hubs are likewise tallied dark hubs. From the above properties 3 and 4, we can determine, a Red-Black Tree of stature h has dark tallness >= h/2.

Each Red Black Tree with n hubs has stature <= 2Log2(n+1)

This can be demonstrated utilising the accompanying realities:

For an overall Binary Tree, let k be the base number of hubs on all roots to NULL ways, at that point n >= 2k – 1 (Ex. If k is 3, n is at any rate 7). This articulation can likewise be composed as k <= Log2(n+1).

From property 4 of Red-Black trees or more cases, we can say in a Red-Black Tree with n hubs, there is a root to leaf way with at-most Log2(n+1) dark hubs.

From property 3 of Red-Black trees, we can guarantee that the quantity of dark hubs in a Red-Black tree is at any rate (n/2) where n is the complete number of hubs.

From the above focuses, we can finish up the way that Red Black Tree with n hubs has tallness <= 2Log2(n+1)

Search Operation in Red-dark Tree:

As each red-dark tree is an exceptional instance of a twofold tree , looking through the calculation of a red-dark tree is like that of a doubletree.

Algorithm:

```
searchElement (tree, val)
Step 1:
If tree -> data = val OR tree = NULL
   Return tree
Else
If val  data
      Return searchElement (tree -> left, val)
   Else
```

Notes

```
        Return searchElement (tree -> right, val)
    [ End of if ]
[ End of if ]

Step 2: END
```

2.1.6 AVL Trees

AVL tree is a tallness adjusted binary search tree. That implies, an AVL tree is likewise a paired hunt tree yet it is a decent tree. A twofold tree is supposed to be adjusted if the contrast between the statues of left and right subtrees of each hub in the tree is either - 1, 0, or +1. At the end of the day, a double tree is supposed to be adjusted if the tallness of the left and right offspring of each hub contrast by either - 1, 0 or +1. In an AVL tree, each hub keeps up additional data known as the equilibrium factor.

The AVL tree was introduced in 1962 by G.M. Adelson-Velsky and E.M. Landis.

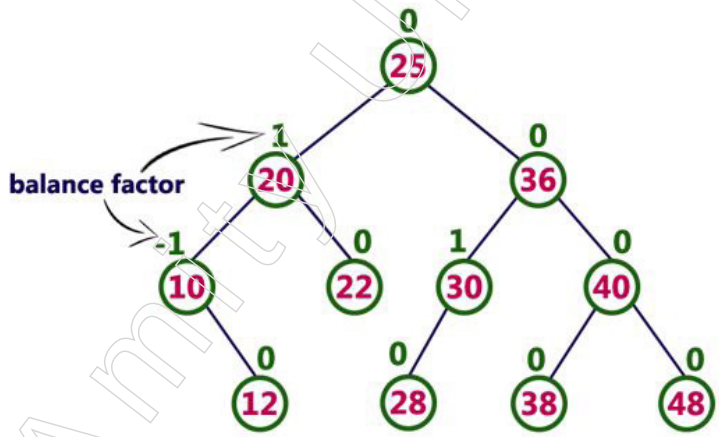
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, the balance factor of every node is either -1, 0 or +1.

Equilibrium factor of a hub is the contrast between the statues of the left and right subtrees of that hub. The equilibrium factor of a hub is determined by either tallness of left subtree - stature of right subtree (OR) tallness of right subtree - stature of left subtree. In the accompanying clarification, we ascertain as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree



The above tree is a binary search tree and every node satisfies the balance factor condition. So this tree is said to be an AVL tree.

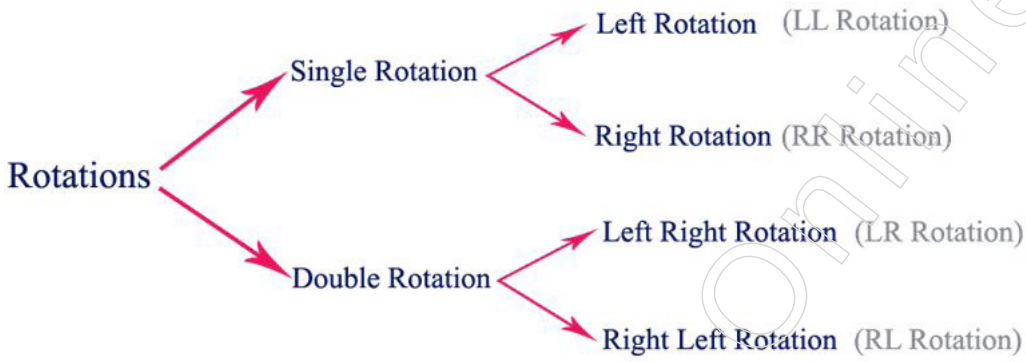
AVL Tree Rotations

In the AVL tree, subsequent to performing activities like inclusion and cancellation we need to check the equilibrium factor of each hub in the tree. On the off chance that each hub fulfils the equilibrium factor condition, we close the activity else we should make it adjusted. At whatever point the tree gets imbalanced because of any activity we use pivot tasks to make the tree adjusted.

Notes

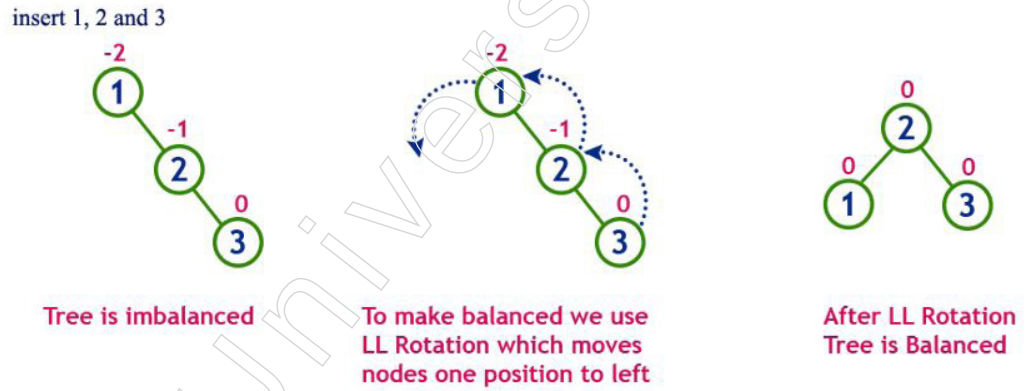
Pivot tasks are utilised to make the tree adjusted.

There are four rotations and they are classified into two types.



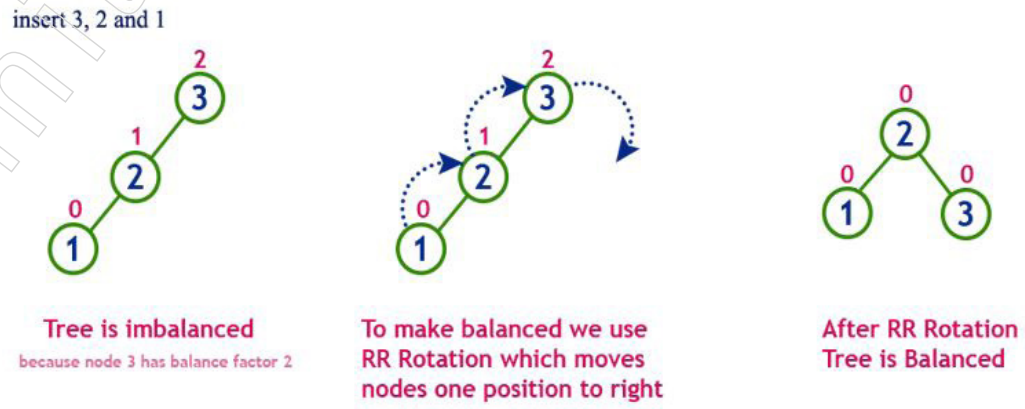
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to the left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

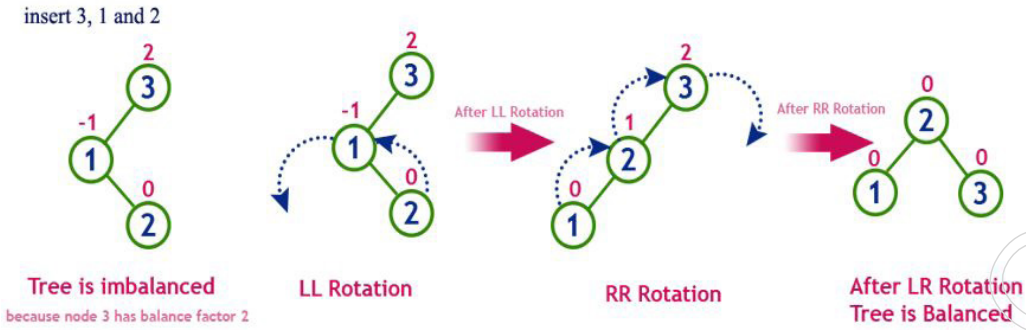


Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right

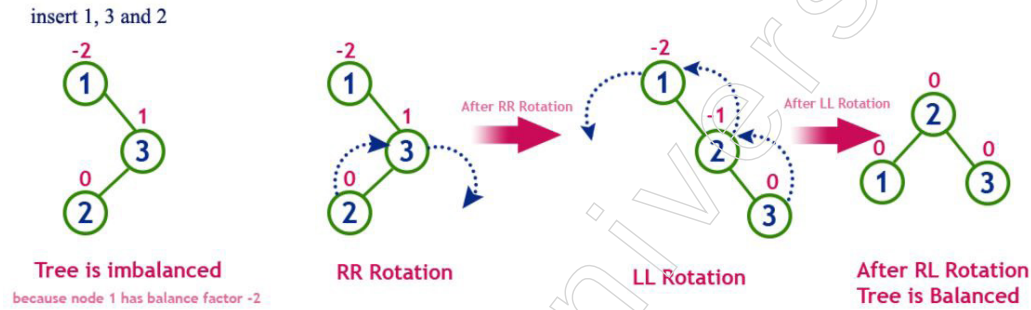
Notes

rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is a sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL trees...

Search

Insertion

Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search for an element in the AVL tree...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of the root node in the tree.
- Step 3 - If both are matched, then display “Given node is found!!!” and terminate the function
- Step 4 - If both are not matched, then check whether the search element is smaller or larger than that node value.
- Step 5 - If the search element is smaller, then continue the search process in the left subtree.

Notes

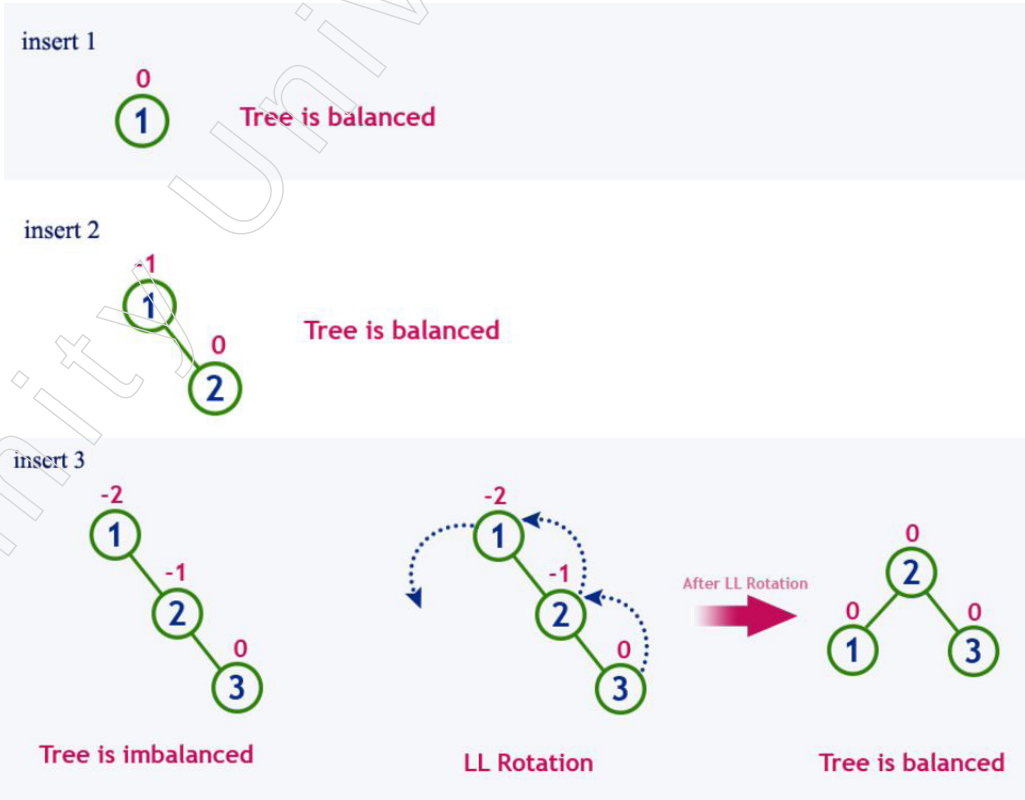
- Step 6 - If the search element is larger, then continue the search process in the right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- Step 8 - If we reach the node having the value equal to the search value, then display “Element is found” and terminate the function.
- Step 9 - If we reach the leaf node and if it is also not matched with the search element, then display “Element is not found” and terminate the function.

Insertion Operation in AVL Tree

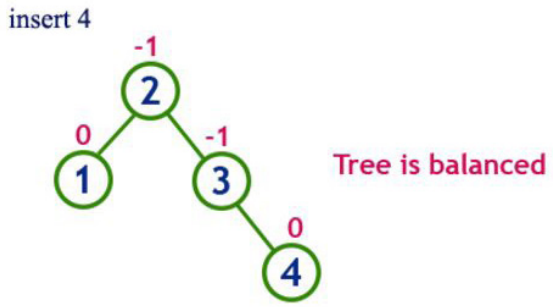
In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

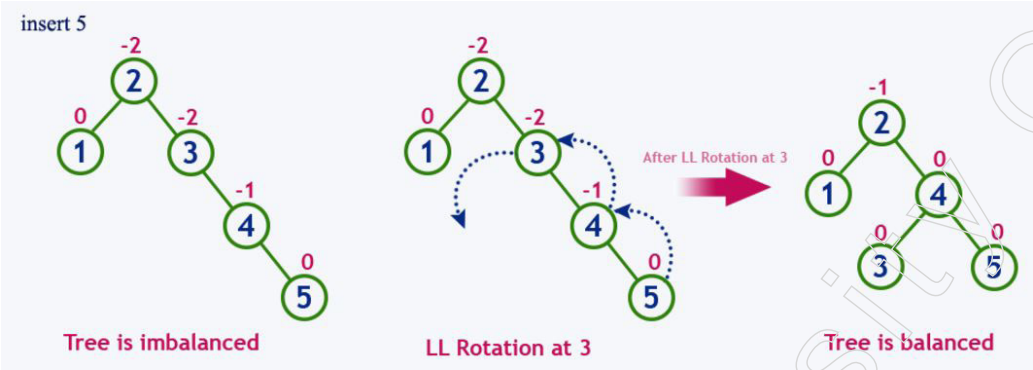
- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the Balance Factor of every node.
- Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for the next operation.
- Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform a suitable Rotation to make it balanced and go for the next operation.

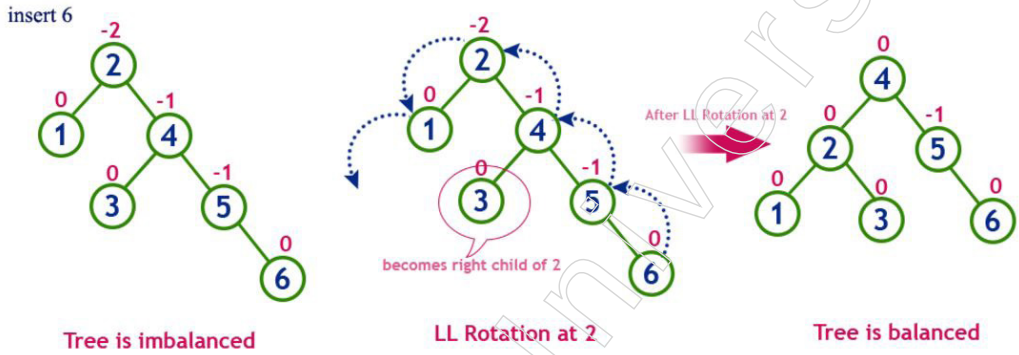
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

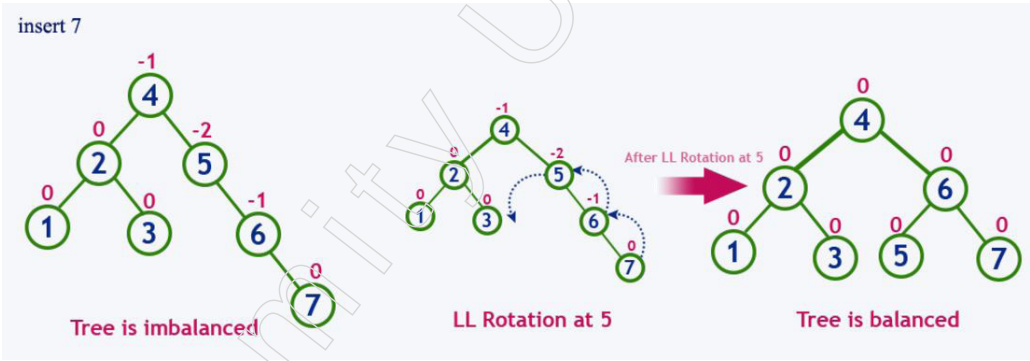


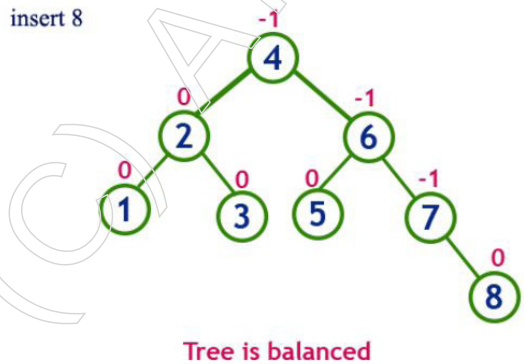
Notes











Notes

Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

2.1.7 B-Trees

B-Tree is a self-adjusting search tree. In the vast majority of the other self-adjusting search trees (like AVL and Red-Black Trees), it is expected that everything is in primary memory. To comprehend the utilisation of B-Trees, we should think about the immense measure of information that can't fit in fundamental memory. At the point when the quantity of keys is high, the information is perused from the circle as squares. Circle access time is extremely high in contrast with the principle memory access time. The primary thought of utilising B-Trees is to decrease the quantity of circle gets to. The vast majority of the tree tasks (search, embed, erase, max, min, ..and so on) require O(h) circle to get to where h is the stature of the tree. B-tree is a fat tree. The stature of B-Trees is kept low by placing the greatest potential keys in a B-Tree hub. For the most part, the B-Tree hub size is held equivalent to the circle block size. Since the stature of the B-tree is low so complete plate gets to for the greater part of the activities are diminished fundamentally contrasted with adjusted Binary Search Trees like AVL Tree, Red-Black Tree, ..and so on

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

"n" is the total number of elements in the B-tree.

Properties of B-Tree:

All leaves are at a similar level.

A B-Tree is characterised by the term least degree 't'. The estimation of t relies on plate block size.

Each hub except the root should contain in any event (ceiling)([t-1]/2) keys. The root may contain at least 1 key.

All hubs (counting root) may contain all things considered t – 1 key.

The number of offspring of a hub is equivalent to the number of keys in it in addition to 1.

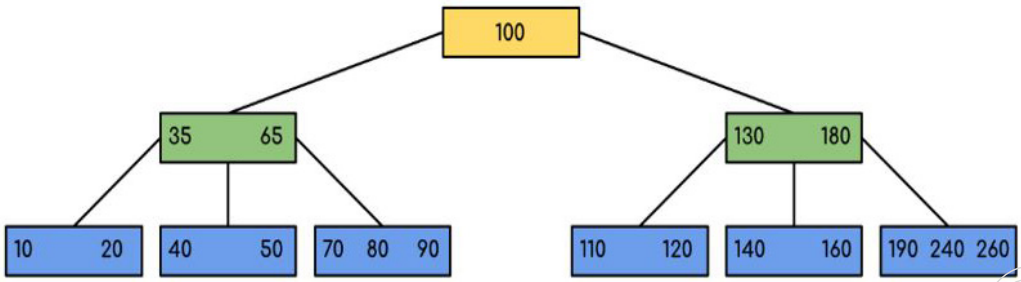
All keys of a hub are arranged in an expanding request. The youngster between two keys k1 and k2 contains all keys in the range from k1 and k2.

B-Tree develops and shrivels from the root which is not normal for Binary Search Tree. Parallel Search Trees become descending and shrivel from descending.

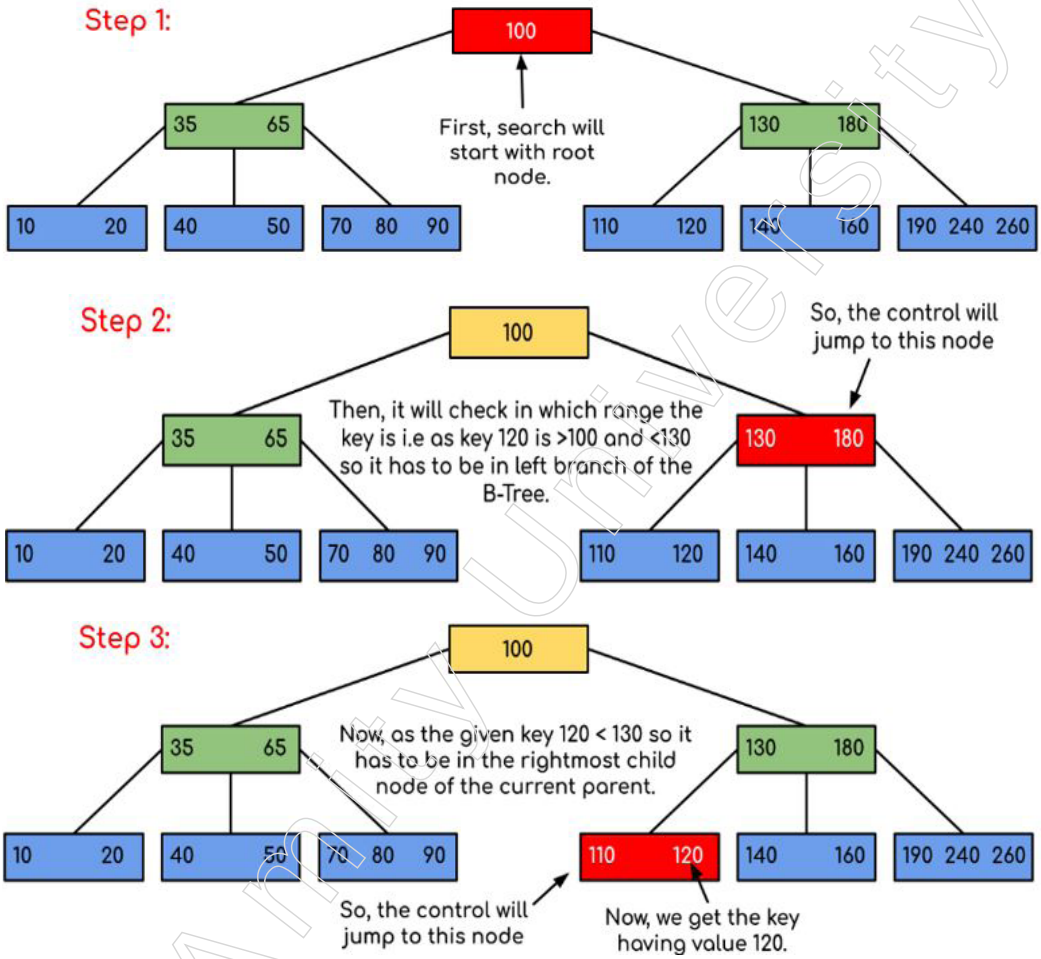
Like other adjusted Binary Search Trees, time intricacy to look, embed, and erase is O(log n).

Notes

Following is an illustration of B-Tree of least request 5. Note that in viable B-Trees, the estimation of the base request is considerably more than 5.



We can find in the above chart that all the leaf hubs are at a similar level and all non-leaf have no vacant sub-tree and have keys one not exactly the number of their kids.

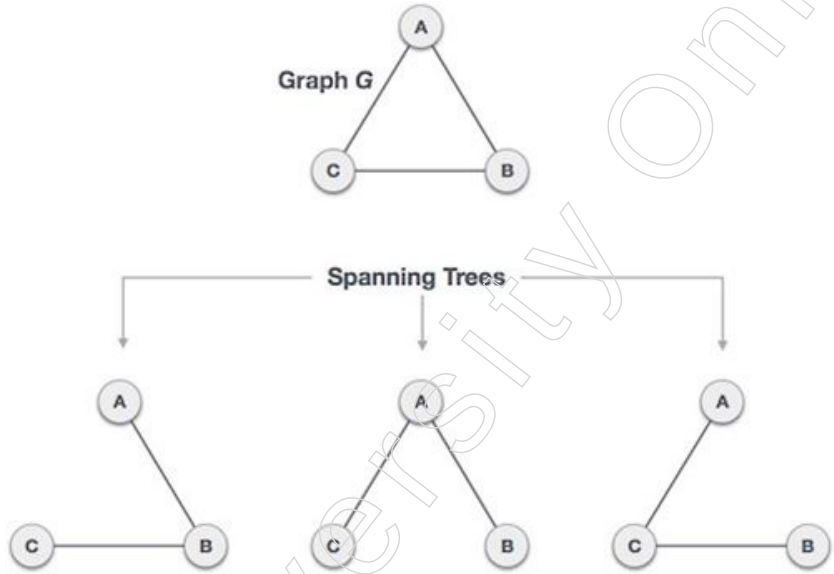


In this model, we can see that our pursuit was diminished simply by restricting the odds where the key containing the worth could be available. Likewise if inside the above model we've to search for 180, the control will stop at stage 2 because the program will locate that the key 180 is available inside the ebb and flow hub. Furthermore, correspondingly, on the off chance that it's to search out 90, as 90 < 100 so it'll go to one side subtree consequently, and along these lines the control stream will go also as demonstrated inside the above model.

Notes

2.1.8 Spanning Trees

A spanning tree is a subset of Graph G, which has all the vertices covered with the least conceivable number of edges. Consequently, a traversing tree doesn't have cycles and it can't be separated.. By this definition, we can make an inference that each associated and undirected Graph G has, in any event, one spanning over the tree. A detached chart doesn't have any crossing tree, as it can't be spread over to all its vertices.



We discovered three spanning over trees off one complete diagram. A total undirected chart can have the greatest nn-2 number spanning over trees, where n is the number of hubs. In the above tended to model, n is 3, subsequently, 33−2 = 3 traversing trees are conceivable.

General Properties of Spanning Tree

We presently comprehend that one chart can have more than one spanning over the tree. Following are a couple of properties of the crossing tree associated with diagram G −

- An associated diagram G can have more than one spanning over the tree.
- All conceivable spanning over trees of chart G have a similar number of edges and vertices.
- The crossing tree doesn't have any cycle (circles).
- Removing one edge from the spanning over the tree will make the diagram detached, for example, the traversing tree is negligibly associated.
- Adding one edge to the spanning over the tree will make a circuit or circle, for example, the crossing tree is maximally non-cyclic.

Numerical Properties of Spanning Tree

- The spanning tree has n-1 edges, where n is the number of hubs (vertices).
- From a total diagram, by eliminating the most extreme e - n + 1 edges, we can build a crossing tree.
- A complete diagram can have the greatest nn-2 number of spanning over trees.

Notes

Hence, we can presume that traversing trees are a subset of associated Graph G, and disengaged diagrams don't have to span over the tree

Minimum Spanning-Tree Algorithm

We shall learn about the two most important spanning tree algorithms here −

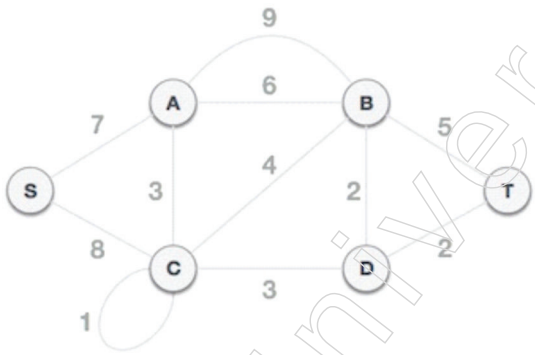
- Kruskal's Algorithm
- Prim's Algorithm

2.1.9 Prim's Algorithm

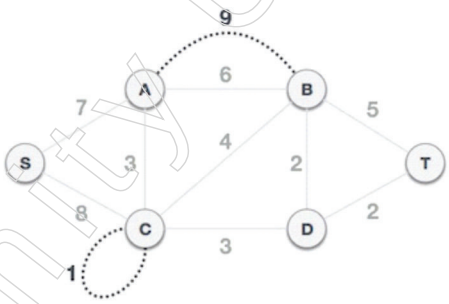
Prim's algorithm to discover the least expense traversing tree (as Kruskal's calculation) utilises the avaricious methodology. Tidy's calculation imparts comparability to the briefest way first algorithm.

Prim's algorithm, interestingly with Kruskal's calculation, regards the hubs as a solitary tree and continues adding new hubs to the crossing tree from the given chart.

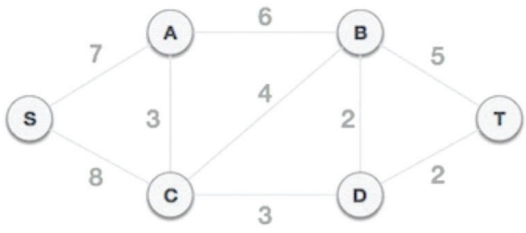
To stand out from Kruskal's calculation and to comprehend Prim's calculation better, we will utilise a similar model −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In the case of parallel edges, keep the one that has the least cost associated and remove all others.



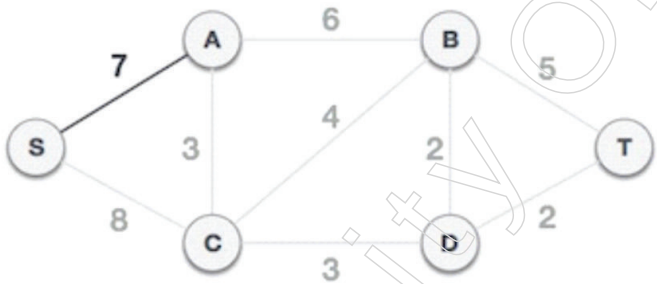
Step 2 - Choose any arbitrary node as the root node

Notes

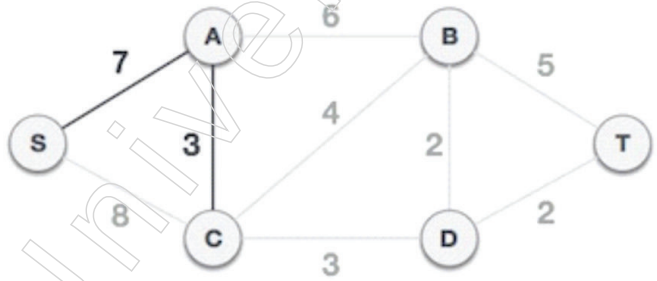
For this situation, we pick the S hub as the root hub of Prim's traversing tree. This hub is discretionarily picked, so any hub can be the root hub. One may ask why any video can be a root hub. So the appropriate response is, in the crossing tree all the hubs of a chart are incorporated and because it is associated then there should be at any rate one edge, which will go along with it to the remainder of the tree.

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

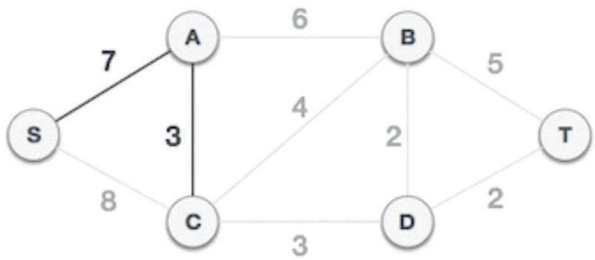
After choosing the root node S, we see that S, A, and S, C are two edges with weights 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



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.



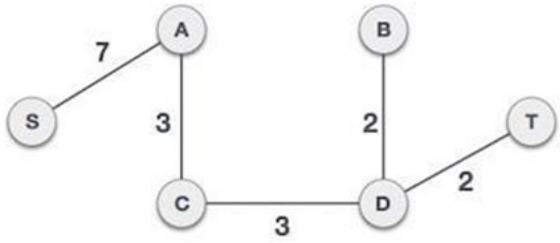
After this progression, the S-7-A-3-C tree is shaped. Presently we'll again regard it as a hub and will check all the edges once more. Be that as it may, we will pick just the smallest expense edge. For this situation, C-3-D is the new edge, which is not exactly other edges' expense 8, 6, 4, and so on



After adding hub D to the traversing tree, we presently have two edges leaving it having a similar expense, for example, D-2-T and D-2-B. In this manner, we can possibly add one. However, the subsequent stage will again yield edge 2 as the most minimal expense. Henceforth, we are showing a traversing tree with the two edges included. etc.

We may find that the output spanning tree of the same graph using two different algorithms is the same.

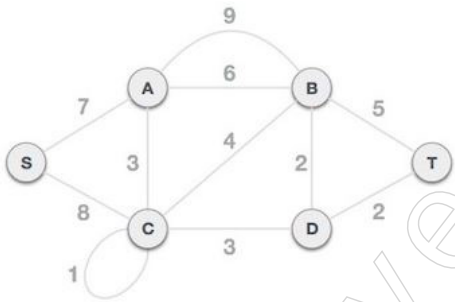
Notes



2.1.10 Kruskal's Algorithm

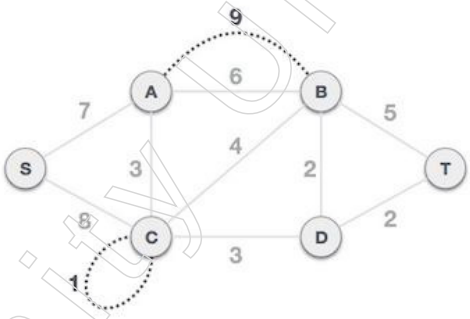
Kruskal's algorithm to track down the base expense spreading over trees utilises the eager methodology. This calculation regards the diagram as a wood and each hub it has as an individual tree. A tree interfaces with another just and just in the event that it has the smallest expense among every accessible choice and doesn't disregard MST properties.

To comprehend Kruskal's algorithm let us think about the accompanying model –

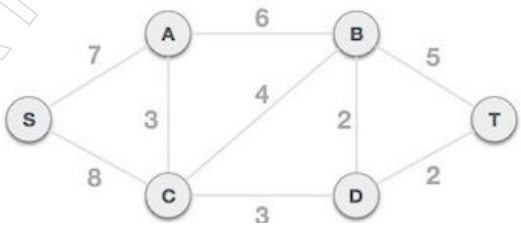


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In the case of parallel edges, keep the one that has the least cost associated and remove all others.



Step 2 - Arrange all edges in their increasing order of weight

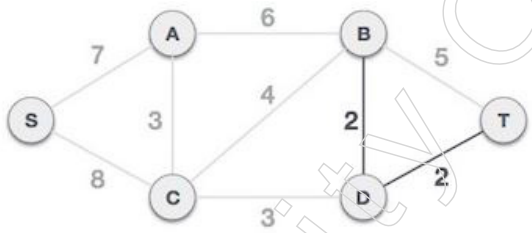
The following stage is to make a bunch of edges and weight and mastermind them in a climbing request of weightage (cost).

Notes

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

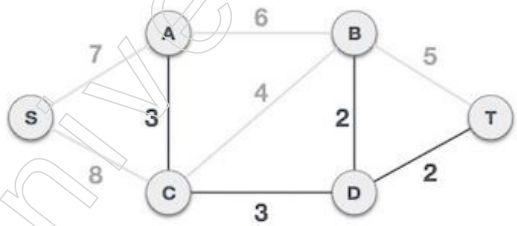
Step 3 - Add the edge which has the least weightage

Presently we begin adding edges to the diagram starting from the one which has the least weight. All through, we will continue watching that the spreading over properties stays unblemished. On the off chance that, by adding one edge, the traversing tree property doesn't hold then we will think about not remembering the edge for the diagram.

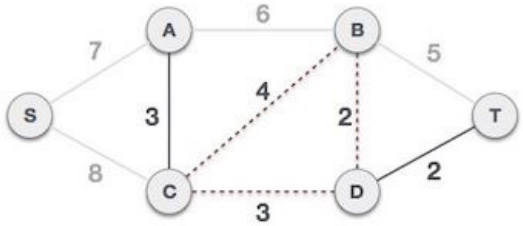


The most minimal expense is 2 and the edges included are B,D, and D,T. We add them. Adding them doesn't abuse traversing tree properties, so we proceed to our next edge determination.

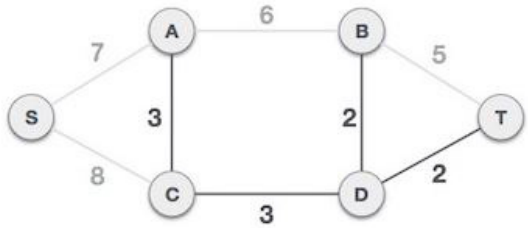
The next cost is 3, and the associated edges are A,C, and C,D. We add them again −



The next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −

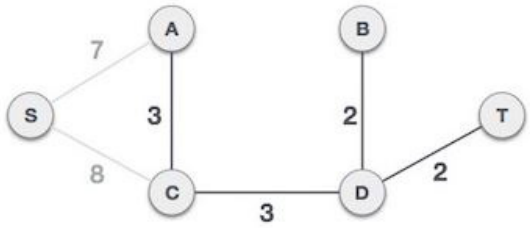


We ignore it. In the process, we shall ignore/avoid all edges that create a circuit.

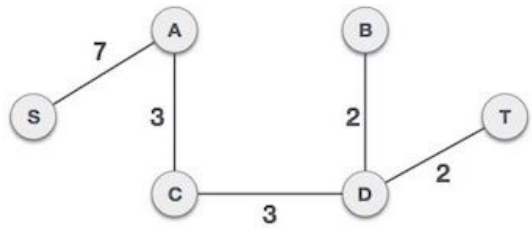


We observe that edges with costs 5 and 6 also create circuits. We ignore them and move on.

Notes



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,we have included all the nodes of the graph and we now have a minimum cost spanning tree.

2.2 Graphs

A graph is a non-linear data structure consisting of vertices and edges. The vertices, also referred to as nodes, are connected by edges, which are lines or arcs that link any two nodes in the graph. Formally, a graph is composed of a set of vertices V and a set of edges E. The graph is denoted by G(V,E).

Graph data structures are a powerful tool for representing and analysing complex relationships between objects or entities. They are particularly useful in various fields, such as:

- Social Network Analysis: Graphs can model relationships between individuals, identifying influential nodes and communities within the network.
- Recommendation Systems: Graphs help in modelling user-item interactions, enabling personalised recommendations based on the connectivity and similarity of nodes.
- Computer Networks: Graphs are used to represent and optimise network topologies, routing, and communication protocols.
- Sports Data Science: Graphs can analyse and understand the dynamics of team performance and player interactions on the field, providing insights into strategies and performance metrics.

2.2.1 Terminology, representations, traversals

Graphs Terminology

Terminology

A graph consists of:

- A set, V, of vertices (nodes)
- A collection, E, of pairs of vertices from V called edges (arcs)

Notes

Edges, also called arcs, are represented by (u, v) and are either:

Directed if the pairs are ordered (u, v)

u the origin

v the destination

Undirected if the pairs are unordered

Then a graph can be:

- A directed graph (digraph) if all the edges are directed

  Undirected graph (graph) if all the edges are undirected

  Mixed graph if edges are both directed or undirected

Illustrate terms on graphs

- The end-vertices of an edge are the endpoints of the edge.
- Two vertices are adjacent if they are endpoints of the same edge.
- An edge is an incident on a vertex if the vertex is an endpoint of the edge.
- Outgoing edges of a vertex are directed edges that the vertex is the origin.

Incoming edges of a vertex are directed edges that the vertex is the destination.

- The degree of a vertex, v, denoted deg(v) is the number of incident edges.

  Out-degree, outdeg(v), is the number of outgoing edges.

  In-degree, indeg(v), is the number of incoming edges.
- Parallel edges or multiple edges are edges of the same type and end-vertices

  Self-loop is an edge with the end vertices the same vertex

  Simple graphs have no parallel edges or self-loops

Graph Representation

Charts are numerical designs that address pairwise connections between objects. A chart is a stream structure that addresses the connection between different items. It tends to be pictured by utilising the accompanying two essential segments:

- Hubs: These are the main parts in any chart. Hubs are elements whose connections are communicated utilising edges. On the off chance that a diagram contains 2 hubs and an undirected edge between them, at that point, it communicates a bi-directional connection between the hubs and edge.
- Edges: Edges are the parts that are utilised to address the connections between different hubs in a chart. An edge between two hubs communicates a single direction or two-path connection between the hubs.

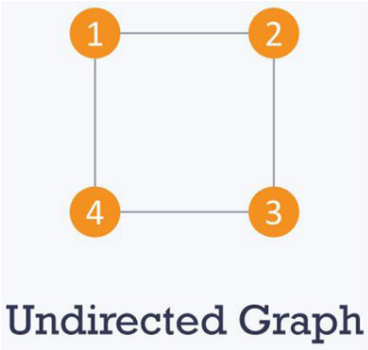
Types of Nodes

- Root hub: The root hub is the predecessor of any remaining hubs in a chart. It doesn't have any predecessor. Each chart comprises precisely one root hub. For the most part, you should begin navigating a chart from the root hub.
- Leaf hubs: In a chart, leaf hubs address the hubs that don't have any replacements. These hubs just have precursor hubs. They can have quite a few approaching edges however they won't have any friendly edges.

Notes

Types of Graphs

Undirected: An undirected chart is a diagram wherein all the edges are bi-directional for example the edges don't point a particular way.



Directed: A directed graph is a diagram where all the edges are unidirectional for example the edges point in a solitary way.

Graph Traversals

We often want to solve problems that are expressible in terms of a traversal or search over a graph. Examples include:

- Finding all reachable nodes (for garbage collection)
- Finding the best reachable node (single-player game search) or the minmax best reachable node (two-player game search)
- Finding the best path through a graph (for routing and map directions)
- Determining whether a graph is a DAG.
- Topologically sorting a graph.

The goal of a graph traversal, generally, is to find all nodes reachable from a given set of root nodes. In an undirected graph, we follow all edges; in a directed graph we follow only out-edges.

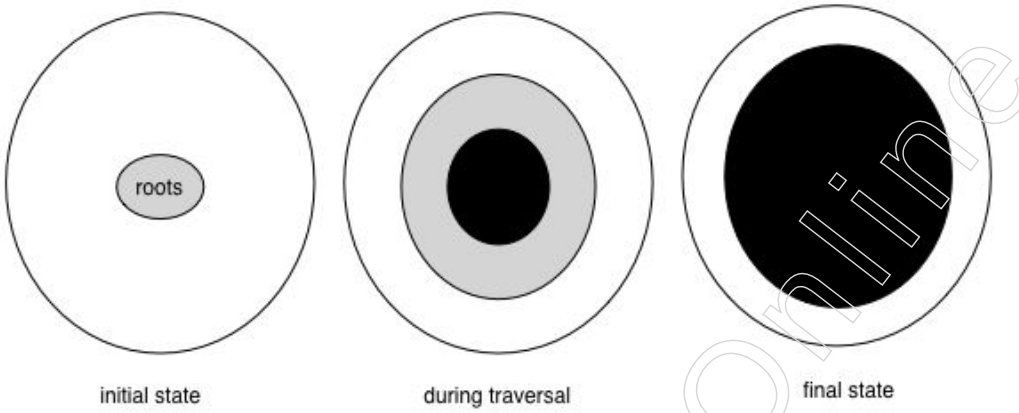
Tricolour Algorithm

Abstractly, graph traversal can be expressed in terms of the tricolour algorithm due to Dijkstra and others. In this algorithm, graph nodes are assigned one of three colours that can change over time:

- White nodes are undiscovered nodes that have not been seen yet in the current traversal and may even be unreachable.
- Black nodes are nodes that are reachable and that the algorithm is done with.
- Grey nodes are nodes that have been discovered but that the algorithm is not done with yet. These nodes are on a frontier between white and black.

The progress of the algorithm is depicted by the following figure. Initially, there are no black nodes and the roots are grey. As the algorithm progresses, white nodes turn into grey nodes and grey nodes turn into black nodes. Eventually, there are no grey nodes left and the algorithm is done.

Notes



The algorithm maintains a key invariant at all times: there are no edges from white nodes to black nodes. This is true initially, and because it is true at the end, we know that any remaining white nodes cannot be reached from the black nodes.

The algorithm pseudo-code is as follows:

1. Colour all nodes white, except for the root nodes, which are coloured grey.
2. While some grey node n exists:
   - ❖ colour some white successors of n grey.
   - ❖ if n has no white successors, optionally colour n black.

This algorithm is abstract enough to describe many different graph traversals. It allows the particular implementation to choose the node n from among the grey nodes; it allows choosing which and how many white successors to colour grey, and it allows delaying the colouring of grey nodes black. We say that such an algorithm is nondeterministic because its behaviour is not fully defined. However, as long as it does some work on each grey node that it picks, any implementation that can be described in terms of this algorithm will finish. Further, because the black-white invariant is maintained, it must reach all reachable nodes in the graph.

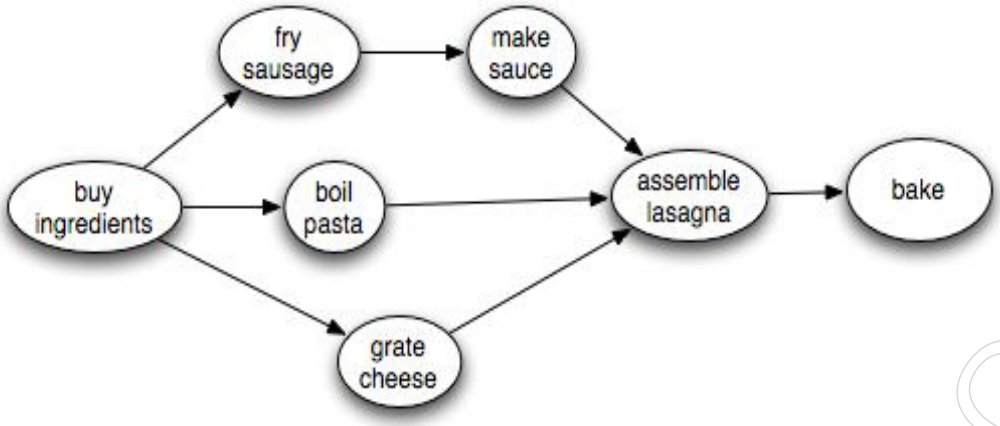
One value of defining graph search in terms of the tricolour algorithm is that the tricolour algorithm works even when grey nodes are worked on concurrently, as long as the black-white invariant is maintained. Thinking about this invariant therefore helps us ensure that whatever graph traversal we choose will work when parallelized, which is increasingly important.

Topological Sort

One of the most useful algorithms on graphs is topological sort, in which the nodes of an acyclic graph are placed in an order consistent with the edges of the graph. This is useful when you need to order a set of elements where some elements have no ordering constraint relative to other elements.

For example, suppose you have a set of tasks to perform, but some tasks have to be done before other tasks can start. In what order should you perform the tasks? This problem can be solved by representing the tasks as a node in a graph, where there is an edge from task 1 to task 2 if task 1 must be done before task 2. Then a topological sort of the graph will give an ordering in which task 1 precedes task 2. To topologically sort a graph, it cannot have cycles. For example, if you were making lasagna, you might need to carry out tasks described by the following graph:

Notes



There is some flexibility about what order to do things in, but clearly, we need to make the sauce before we assemble the lasagna. A topological sort will find some ordering that obeys this and the other ordering constraints. Of course, it is impossible to topologically sort a graph with a cycle in it.

The key observation is that a node finishes (is marked black) after all of its descendants have been marked black. Therefore, a node that is marked black later must come earlier when topologically sorted. A postorder traversal generates nodes in the reverse of a topological sort:

Algorithm:

Perform a depth-first search over the entire graph, starting anew with an unvisited node if previous starting nodes did not visit every node. As each node is finished (coloured black), put it on the head of an initially empty list. This takes time linear in the size of the graph: O(|V| + |E|).

For example, in the traversal example above, nodes are marked black in the order C, E, D, B, A. Reversing this, we get the ordering A, B, D, E, C. This is a topological sort of graph. Similarly, in the lasagna example, assuming that we choose successors top-down, nodes are marked black in the order: bake, assemble lasagna, make the sauce, fry sausage, boil pasta, and grate cheese. So the reverse of this ordering gives us a recipe for successfully making lasagna, even though successful cooks are likely to do things more in parallel!

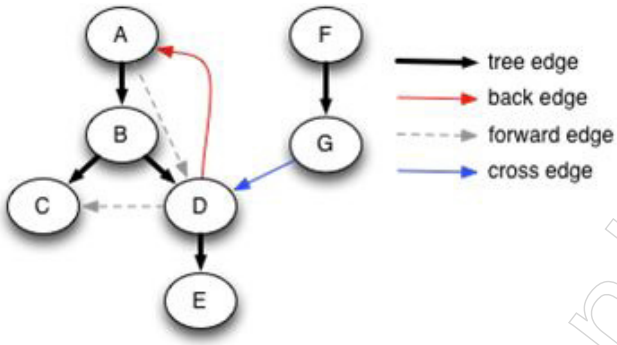
1. Detecting cycles

   Since a node finishes after its Descendents, a cycle involves a grey node pointing to one of its grey ancestors that haven't finished yet. If one of a node's successors is grey, there must be a cycle. To detect cycles in graphs, therefore, we choose an arbitrary white node and run DFS. If that completes and there are still white nodes left over, we choose another white node arbitrarily and repeat. Eventually, all nodes are coloured black. If at any time we follow an edge to a grey node, there is a cycle in the graph. Therefore, cycles can be detected in O(|V+E|) time.

2. Edge classifications

   We can classify the various edges of the graph based on the colour of the node reached when the algorithm follows the edge. Here is the expanded (A–G) graph with the edges coloured to show their classification.

Notes



Note that the classification of edges depends on what trees are constructed, and therefore depends on what node we start from and in what order the algorithm happens to select successors to visit.

When the destination node of a followed edge is white, the algorithm performs a recursive call. These edges are called tree edges, shown as solid black arrows. The graph looks different in this picture because the nodes have been moved to make all the tree edges go downward. We have already seen that tree edges show the precise sequence of recursive calls performed during the traversal.

When the destination of the followed edge is grey, it is a back edge, shown in red. Because there is only a single path of grey nodes, a back edge is looping back to an earlier grey node, creating a cycle. A graph has a cycle if and only if it contains a back edge when traversed from some node.

When the destination of the followed edge is coloured black, it is a forward edge or across the edge. It is a cross edge if it goes between one tree and another in the forest; otherwise, it is a forward edge.

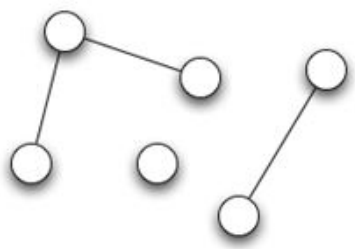
Detecting Cycles

It is often useful to know whether a graph has cycles. To detect whether a graph has cycles, we perform a depth-first search of the entire graph. If a back edge is found during any traversal, the graph contains a cycle. If all nodes have been visited and no back edge has been found, the graph is acyclic.

Connected Components

Graphs need not be connected, although we have been drawing connected graphs thus far. A graph is connected if there is a path between every two nodes. However, it is entirely possible to have a graph in which there is no path from one node to another node, even following edges backward. For connectedness, we don't care which direction the edges go in, so we might as well consider an undirected graph. A connected component is a subset S such that for every two adjacent vertices v and v', either v and v' are both in S or neither one is.

For example, the following undirected graph has three connected components:



Notes

The connected components problem is to determine how many connected components make up a graph, and to make it possible to find, for each node in the graph, which component it belongs to. This can be a useful way to solve problems. For example, suppose that different components correspond to different jobs that need to be done, and there is an edge between two components if they need to be done on the same day. Then to find out what is the maximum number of days that can be used to carry all the jobs, we need to count the components.

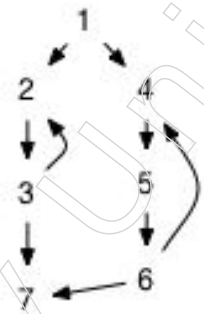
Algorithm:

Perform a depth-first search over the graph. As each traversal starts, create a new component. All nodes reached during the traversal belong to that component. The number of traversals done during the depth-first search is the number of components. Note that if the graph is directed, the DFS needs to follow both in- and out-edges.

For directed graphs, it is usually more useful to define strongly connected components. A strongly connected component (SCC) is a maximal subset of vertices such that every vertex in the set is reachable from every other. All cycles in a graph are part of the same strongly connected component, which means that every graph can be viewed as a DAG composed of SCCs. There is a simple and efficient algorithm due to Kosaraju that uses depth-first search twice:

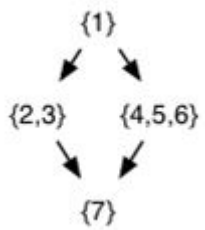
- Topologically sort the nodes using DFS. The SCCs will appear in sequence.
- Now traverse the transposed graph, but pick new (white) roots in topological order. Each new subtraversal reaches a distinct SCC.

For example, consider the following graph, which is not a DAG:



Running a depth-first traversal in which we happen to choose children left-to-right, and extracting the nodes in reverse postorder, we obtain the ordering 1, 4, 5, 6, 2, 3, 7. Notice that the SCCs occur sequentially within this ordering. The job of the second part of the algorithm is to identify the boundaries.

In the second phase, we start with 1 and find it has no predecessors, so {1} is the first SCC. We then start with 4 and find that 5 and 6 are reachable via backward edges, so the second SCC is {4,5,6}. Starting from 2, we discover {2,3}, and the final SCC is {7}. The resulting DAG of SCCs is the following:



Notes

Notice that the SCCs are also topologically sorted by this algorithm.

One inconvenience of this algorithm is that it requires being able to walk edges backward. Tarjan's algorithm for identifying strongly connected components is only slightly more complicated, yet performs just one depth-first traversal of the graph and only in a forward direction.

2.2.2 Depth-First Search (DFS)

What if we were to replace the FIFO queue with a LIFO stack? In that case, we get a completely different order of traversal. Assuming that successors are pushed onto the stack in reverse alphabetical order, the successive stack states look like this:

A

B D E

C D E

D E

E

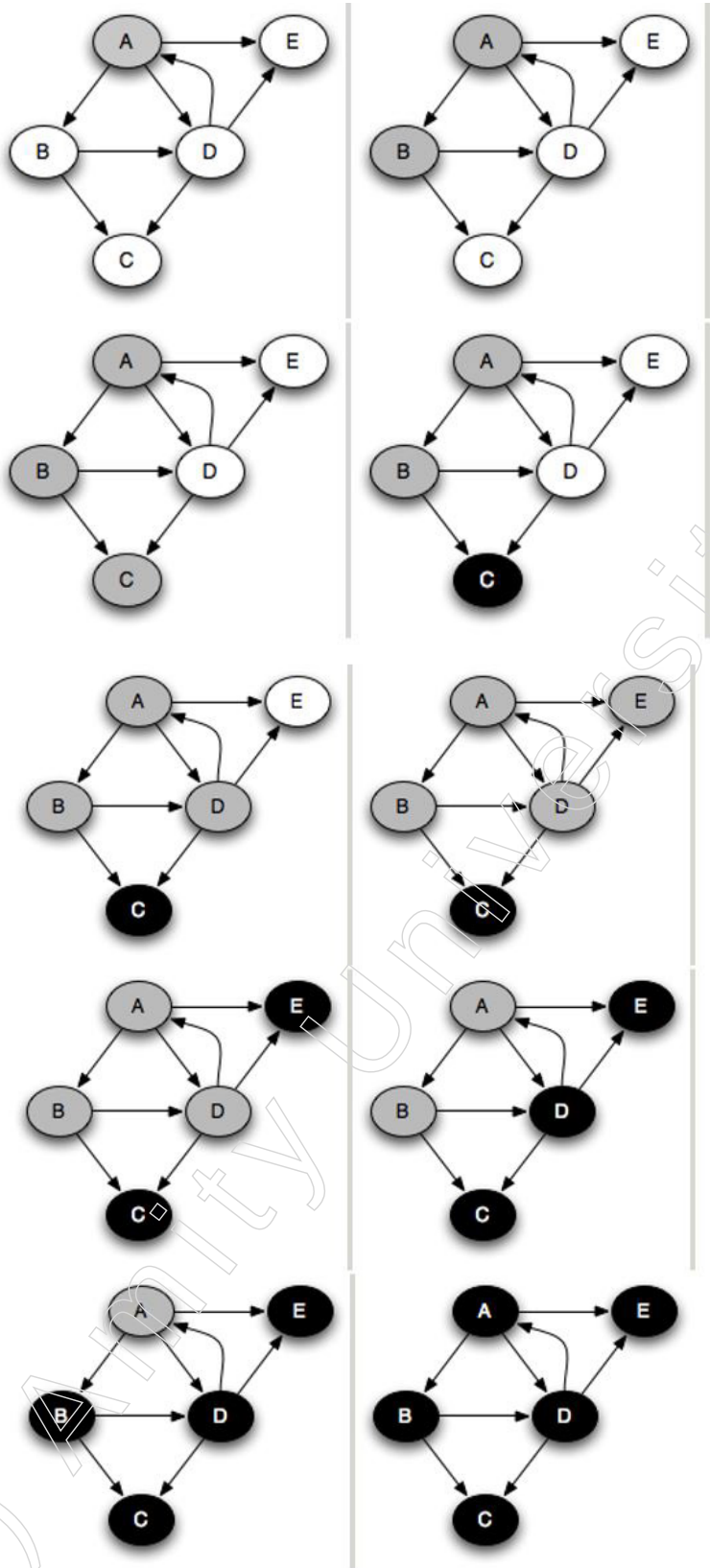
With a stack, the search will proceed from a given node as far as it can before backtracking and considering other nodes on the stack. For example, node E had to wait until all nodes reachable from B and D were considered. This is a depth-first search.

A more standard way of writing depth-first search is as a recursive function, using the program stack as the stack above. We start with every node white except the starting node and apply the function DFS to the starting node:

```
DFS(Vertex v) {
   mark v visited
   set colour of v to grey
   for each successor v' of v {
    if v' not yet visited {
       DFS(v')
    }
   }
   set colour of v to black
}
```

You can think of this as a person walking through the graph following arrows and never visiting a node twice except when backtracking when a dead-end is reached. Running this code on the graph above yields the following graph colorings in sequence, which are reminiscent of but a bit different from what we saw with the stack-based version:

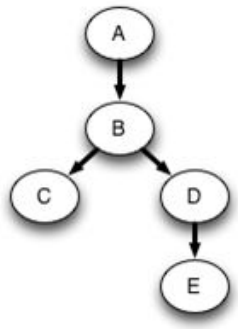
Notes



Notice that at any given time there is a single path of grey nodes leading from the starting node and leading to the current node v. This path corresponds to the stack in the earlier implementation, although the nodes end up being visited in a different order because the recursive algorithm only marks one successor grey at a time.

Notes

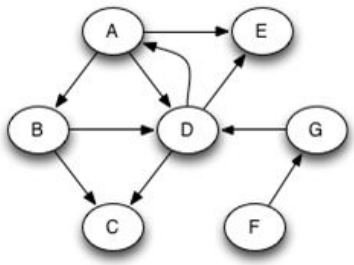
The sequence of calls to DFS forms a tree. This is called the call tree of the program, and in fact, any program has a call tree. In this case, the call tree is a subgraph of the original graph:



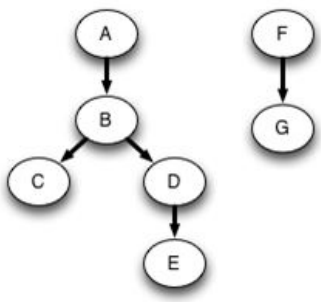
The algorithm maintains an amount of state that is proportional to the size of this path from the root. This makes DFS rather different from BFS, where the amount of state (the queue size) corresponds to the size of the perimeter of nodes at a distance k from the starting node. In both algorithms, the amount of state can be O(|V|). For DFS this happens when searching a linked list. For BFS this happens when searching a graph with a lot of branching, such as a binary tree, because there are 2k nodes at a distance k from the root. On a balanced binary tree, DFS maintains a state proportional to the height of the tree, or O(log |V|). Often the graphs that we want to search are more like trees than linked lists, and so DFS tends to run faster.

There can be at most |V| calls to DFS_visit. And the body of the loop on successors can be executed at most |E| times. So the asymptotic performance of DFS is O(|V| + |E|), just like for breadth-first search.

If we want to search the whole graph, then a single recursive traversal may not suffice. If we had started a traversal with node C, we would miss all the rest of the nodes in the graph. To do a depth-first search of an entire graph, we call DFS on an arbitrary unvisited node and repeat until every node has been visited. For example, consider the original graph expanded with two new nodes F and G:



DFS starting at A will not search all the nodes. Suppose we next choose F to start from. Then we will reach all nodes. Instead of constructing just one tree that is a subgraph of the original graph, we get a forest of two trees:



Notes

2.2.3 Breadth-First Search (BFS)

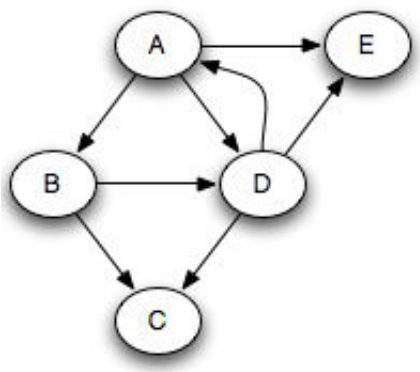
Breadth-first search (BFS) is a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node. Its pseudo-code looks like this:

```
// let s be the source node
frontier = new Queue()
mark root visited (set root.distance = 0)
frontier.push(root)
while frontier not empty {
Vertex v = frontier.pop()
for each successor v' of v {
if v' unvisited {
frontier.push(v')
mark v' visited (v'.distance = v.distance + 1)
}
}
}
```

Here the white nodes are those not marked as visited, the grey nodes are those marked as visited and that are in fronter, and the black nodes are visited nodes no longer in the frontier. Rather than having a visited flag, we can keep track of a node's distance in the field v.distance. When a new node is discovered, its distance is set to be one greater than its predecessor v.

When frontier is a first-in, first-out (FIFO) queue, we get breadth-first search. All the nodes on the queue have a minimum path length within one of each other. In general, there is a set of nodes to be popped off, at some distance k from the source, and another set of elements, later on, the queue, at distance k+1. Every time a new node is pushed onto the queue, it is at distance k+1 until all the nodes at distance k are gone, and k then goes up by one. Therefore newly pushed nodes are always at a distance at least as great as any other grey node.

Suppose that we run this algorithm on the following graph, assuming that successors are visited in alphabetic order from any given node: :



Notes

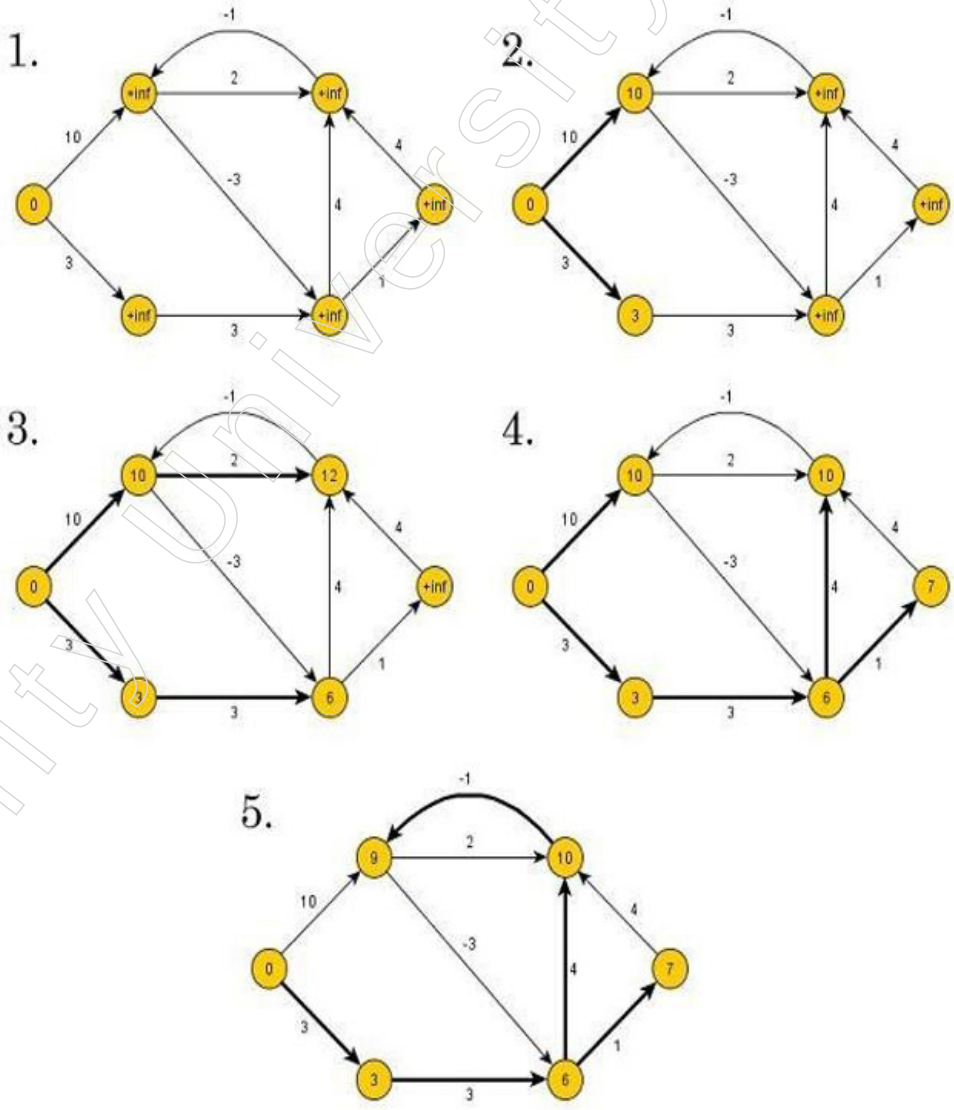
In that case, the following sequence of nodes passes through the queue, where each node is annotated by its minimum distance from the source node A. Note that we're pushing onto the right of the queue and popping from the left.

A0 B1 D1 E1 C2

Nodes are popped in distance order: A, B, D, E, C. This is very useful when we are trying to find the shortest path through the graph to something. When a queue is used in this way, it is known as a worklist; it keeps track of work left to be done.

2.2.4 Single-Source Shortest Path

The single source most limited way calculation (for subjective weight positive or negative) is likewise known as Bellman-Ford calculation is used to discover the least separation from the source vertex to some other vertex. The principal contrast between this calculation with Dijkstra's calculation is, in Dijkstra's calculation we can't deal with the negative weight, yet here we can deal with it without any problem.



Bellman-Ford calculation finds the distance in base up way. From the outset, it finds those distances which have just one edge in the way. After that expansion the way length to locate every conceivable arrangement.

Notes

Input − The cost matrix of the graph:

0 6 ∞ 7 ∞

∞ 0 5 8 -4

∞ -2 0 ∞ ∞

∞ ∞ -3 0 9

2 ∞ 7 ∞ 0

Output − Source Vertex: 2

Vert: 0 1 2 3 4

Dist: -4 -2 0 3 -6

Pred: 4 2 -1 0 1

The graph has no negative edge cycle

Algorithm

bellmanFord(dist, pred, source)

Input − Distance list, predecessor list, and the source vertex.

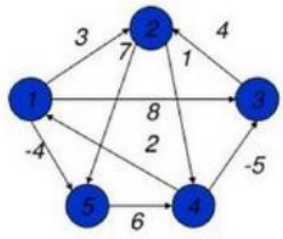
Output − True, when a negative cycle is found.

```
Begin
iCount := 1
maxEdge := n * (n - 1) / 2 //n is number of vertices
for all vertices v of the graph, do
dist[v] := ∞
pred[v] := ϕ
done
dist[source] := 0
eCount := number of edges present in the graph
create edge list named edgeList
while iCount < n, do
for i := 0 to eCount, do
if dist[edgeList[i].v] > dist[edgeList[i].u] + (cost[u,v]
for edge i)
dist[edgeList[i].v] > dist[edgeList[i].u] + (cost[u,v] for
edge i)
pred[edgeList[i].v] := edgeList[i].u
done
done
iCount := iCount + 1
for all vertices i in the graph, do
if dist[edgeList[i].v] > dist[edgeList[i].u] + (cost[u,v]
for edge i), then
return true
done
return false
End
```

Notes

2.2.5 All-Pair Shortest Path

They all pair briefest way calculation is otherwise called Floyd-Warshall calculation is utilised to discover all pair most limited way issues from a given weighted diagram. Because of this calculation, it will create a grid, which will address the base separation from any hub to any remaining hubs in the chart.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

From the start, the yield grid is the same as the given expense network of the chart. After that, the yield grid will be refreshed with all vertices k as the middle of the road vertex.

The time intricacy of this calculation is O(V3), where V is the number of vertices in the diagram.

Information − The expense grid of the chart

```
0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0
```

Output − Matrix of all pairs shortest path.

```
0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0
```

Algorithm

floydWarshall(cost)

Input − The cost matrix of the given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

Notes

```
Begin
for k := 0 to n, do
      for i := 0 to n, do
         for j := 0 to n, do
            if cost[i,k] + cost[k,j] < cost[i,j], then
               cost[i,j] := cost[i,k] + cost[k,j]
            done
         done
      done
  display the current cost matrix
End
```

Example

```
#include<iostream>
#include<iomanip>
#define NODE 7
#define INF 999
using namespace std;
//Cost matrix of the graph
int costMat[NODE][NODE] = {
   {0, 3, 6, INF, INF, INF, INF},
   {3, 0, 2, 1, INF, INF, INF},
   {6, 2, 0, 1, 4, 2, INF},
   {INF, 1, 1, 0, 2, INF, 4},
   {INF, INF, 4, 2, 0, 2, 1},
   {INF, INF, 2, INF, 2, 0, 1},
   {INF, INF, INF, 4, 1, 1, 0}
};
void floydWarshal(){
int cost[NODE][NODE]; //define to store shortest distance
from any node to any node
for(int i = 0; i<NODE; i++)
for(int j = 0; j<NODE; j++)
cost[i][j] = costMat[i][j]; //copy costMatrix to new matrix
for(int k = 0; k<NODE; k++){
for(int i = 0; i<NODE; i++)
for(int j = 0; j<NODE; j++)
if(cost[i][k]+cost[k][j] < cost[i][j])
```

Notes

```
cost[i][j] = cost[i][k]+cost[k][j];
}
cout << "The matrix:" << endl;
for(int i = 0; i<NODE; i++){
for(int j = 0; j<NODE; j++)
cout << setw(3) << cost[i][j];
cout << endl;
}
}
int main(){
floydWarshal();
}
```

Output

```
The matrix:
0 3 5 4 6 7 7
3 0 2 1 3 4 4
5 2 0 1 3 2 3
4 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0
```

Summary

- A tree is an assortment of components called nodes. Every node contains some worth or component.
- A binary tree is a tree-type non-straight information structure with a limit of two kids for each parent. Each hub in a binary tree has a left and right reference alongside the information component.
- A BST(Binary Search tree) is viewed as an information structure composed of hubs, as Linked Lists. These hubs are either invalid or have references (joins) to different hubs.
- A red-dark tree is a sort of self-adjusting double hunt tree where every hub has an additional piece, and that piece is frequently deciphered as the tone (red or dark).
- AVL tree is a tallness adjusted binary search tree. That implies, an AVL tree is likewise a paired hunt tree yet it is a decent tree.
- B-Tree is a self-adjusting search tree. In the vast majority of the other self-adjusting search trees (like AVL and Red-Black Trees), it is expected that everything is in primary memory.
- Prim's algorithm to discover the least expense traversing tree (as Kruskal's calculation)

Notes

utilises the avaricious methodology. Tidy's calculation imparts comparability to the briefest way first algorithm.

- Kruskal's algorithm to track down the base expense spreading over trees utilises the eager methodology. This calculation regards the diagram as a wood and each hub it has as an individual tree.
- A graph is a non-linear data structure consisting of vertices and edges. The vertices, also referred to as nodes, are connected by edges, which are lines or arcs that link any two nodes in the graph.
- Breadth-first search (BFS) is a graph traversal algorithm that explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node.
- The single source most limited way calculation (for subjective weight positive or negative) is likewise known as Bellman-Ford calculation is used to discover the least separation from the source vertex to some other vertex.

Glossary

- Path: A path is an arrangement of edges between nodes.
- Root: The root is the unique node from which all other nodes "descend." Each tree has a single root node.
- Parent of Node n: The parent of node n is the unique node with an edge to node n and which is the first node on the path from n to the root.
- Note: The root is the only node with no parent. Except for the root, every node has exactly one parent.
- Siblings: Nodes with the same parent are siblings.
- Progenitor of Node n: Any node y on the (unique) path from root r to node n is a progenitor of node n. Every node is a progenitor of itself.

Check Your Understanding

1. What is a binary tree?
   a) A tree with multiple children for each parent
   b) A tree with exactly two children for each parent
   c) A tree with no children
   d) A tree with only one child for each parent
2. What is the root of a tree?
   a) The leaf node
   b) The node with the most children
   c) The node with no children
   d) The topmost node
3. How is a perfect binary search tree defined?
   a) A binary search tree with a balanced number of nodes on each level
   b) A binary search tree with all leaf nodes at the same level
   c) A binary search tree with equal frequencies for all keys
   d) A binary search tree with only one child for each parent

Notes

4. What is the time complexity of the search operation in a red-black tree?
   a) O(n)
   b) O(log n)
   c) O(h)
   d) O(1)
5. What property distinguishes a spanning tree from other subsets of a graph?\
   a) It has the maximum number of edges
   b) It has the minimum number of vertices
   c) It covers all vertices with the least possible number of edges
   d) It contains cycles

Exercise

1. How does the in-order traversal algorithm work, and in what scenario is it used?
2. What are the properties of an AVL tree, and why is it useful?
3. Describe the Breadth-First Search (BFS) algorithm and its typical use case.
4. What is a weighted graph, and how does it differ from an unweighted graph?
5. Explain Dijkstra's algorithm for finding the shortest path in a graph.

Learning Activities

1. Students will be given a binary tree and asked to perform and visualise in-order, pre-order, and post-order traversals. They will write the code for each traversal and present the order of node visits.
2. Students will work in groups to solve shortest path problems using Dijkstra's algorithm. They will implement the algorithm and apply it to various graph examples, comparing their results and discussing the efficiency of their implementations.

Check Your Understanding Answer

1. b) 2. d) 3. b) 4. b)
5. c)

Further Readings and Bibliography

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Sedgewick, R. (2011). Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd ed.). Addison-Wesley Professional.
3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.
4. Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley Professional.

Module - III: Dynamic Programming

Learning Objectives

At the end of this module, you will be able to:

- Understand the fundamental principles of dynamic programming and its application to solve optimization problems.
- Comprehend the greedy approach and recognize problems where a greedy algorithm can provide an optimal solution.
- Learn the branch and bound approach for solving combinatorial optimization problems and understand its practical applications.
- Implement dynamic programming, greedy algorithms, and branch and bound algorithms to solve specific problems and analyse their performance.

Introduction

Dynamic Programming (DP) is a technique employed in both mathematics and computer science for tackling intricate problems by breaking them down into simpler subproblems.

3.1 Dynamic Programming

By addressing each subproblem just once and retaining the results, DP avoids repetitive computations, thereby facilitating more efficient solutions across a diverse range of problems.

How Does Dynamic Programming (DP) Operate?

- Identify Subproblems: Decompose the primary problem into smaller, independent subproblems.
- Store Solutions: Resolve each subproblem and retain the solution in a table or array.
- Construct Solutions: Utilise the stored solutions to construct the solution to the main problem.
- Prevent Redundancy: Through the storage of solutions, DP ensures that each subproblem is addressed only once, diminishing computation time.

3.1.1 Dynamic Programming Approach

The dynamic programming approach is like separation and vanquish in separating the issue into more modest but then more modest conceivable sub-issues. However, in contrast, to separate and vanquish, these sub-issues are not addressed freely. Or maybe, aftereffects of these more modest sub-issues are recalled and utilised for comparable or covering sub-issues.

Dynamic writing computer programs are utilised where we have issues, which can be separated into comparative sub-issues, so their outcomes can be re-utilised. For the most part, these algorithms are utilised for enhancement. Before tackling the close-by sub-issue, the dynamic calculation will attempt to inspect the consequences of the recently tackled sub-issues. The arrangements of sub-issues are joined to accomplish the best arrangement.

Notes

So we can say that –

The issue ought to have the option to be separated into a more modest covering sub-issue.

An ideal arrangement can be accomplished by utilising an ideal arrangement of more modest sub-issues.

Dynamic Algorithms use Memorization

Comparison

Rather than greedy algorithms, where neighbourhood improvement is tended to, dynamic algorithms are propelled for a general streamlining of the issue.

As opposed to an isolated and overcome algorithm, where arrangements are consolidated to accomplish a general arrangement, the dynamic algorithm utilises the yield of a more modest sub-issue and afterward attempts to improve a greater sub-issue. Dynamic algorithms use Memoization to recall the yield of effectively tackled sub-issues.

Example

The following computer problems can be solved using the dynamic programming approach −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be utilised in both top-down and base-up ways. What's more, the greater part of the occasions, alluding to the past arrangement yield is less expensive than recomputing regarding CPU cycles.

3.1.2 Principle of Optimality

To use dynamic programming, the problem must observe the principle of optimality that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the first decision .

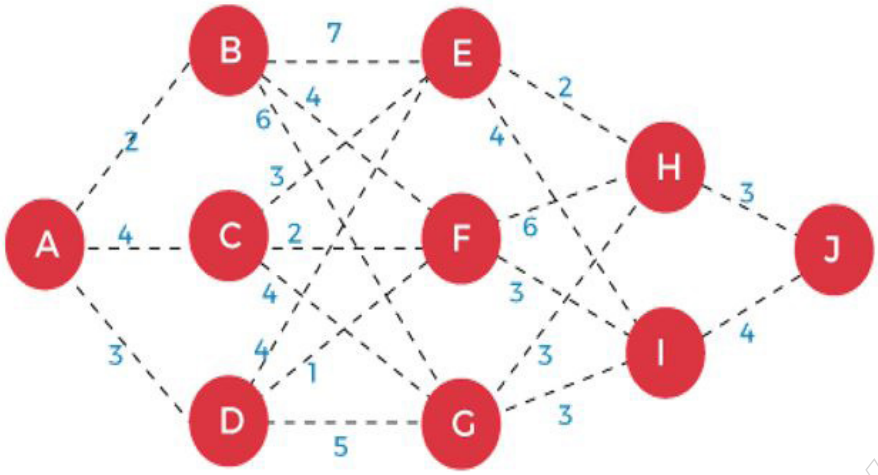
Combinatorial problems may have this property but may use too much memory time to be e client

Richard Bellman, a renowned scientist, introduced the concept of dynamic programming in 1957. One of the key concepts in dynamic programming is the Principle of Optimality, which can be understood through the Optimal Substructure Property.

Optimal Substructure Property

To understand the Optimal Substructure Property, let's use an example. Assume we want to find the minimum distance required to travel from a node X to a nodeJ in a graph. Let F(X) denote this minimum distance, and let F(J)=0 since the distance from J to itself is zero.

Notes



Consider the following distances:

- The distance from vertex H to vertex J is 3.
- The distance from vertex I to vertex J is 4.

Now, let's calculate the distance from vertex E to vertex J:

- F(E)=min{1+F(H),4+F(I)}
- F(E)=min{1+3,4+4}
- F(E)=min{4,8}

Therefore, the minimum distance from vertex E to vertex J is 4.

Next, we calculate the distance from vertex F to vertex J:

- F(F)=min{6+F(H),3+F(I)}
- F(F)=min{6+3,3+4}
- F(F)=min{9,7}

Therefore, the minimum distance from vertex F to vertex J is 7.

Now, let's calculate the distance from vertex G to vertex J:

- F(G)=min{3+F(H),3+F(I)}
- F(G)=min{3+3,3+4}
- F(G)=min{6,7}

Therefore, the minimum distance from vertex G to vertex J is 6.

Next, we calculate the distance from vertex B to vertex J:

- F(B)=min{7+F(E),4+F(F),6+F(G)}
- F(B)=min{7+4,4+7,6+6}
- F(B)=min{11,11,12}

Therefore, the minimum distance from vertex B to vertex J is 11.

Next, we calculate the distance from vertex C to vertex J:

- F(C)=min{3+F(E),2+F(F),4+F(G)}
- F(C)=min{3+4,2+7,4+6}
- F(C)=min{7,9,10}

Therefore, the minimum distance from vertex C to vertex J is 7.

Notes

Next, we calculate the distance from vertex D to vertex J:

- F(D)=min{4+F(E),1+F(F),5+F(G)}
- F(D)=min{4+4,1+7,5+6}
- F(D)=min{8,8,11}

Therefore, the minimum distance from vertex D to vertex J is 8.

Finally, we calculate the distance from vertex A to vertex J:

- F(A)=min{2+F(B),4+F(C),3+F(D)}
- F(A)=min{2+11,4+7,3+8}
- F(A)=min{13,11,11}

Therefore, the minimum distance from vertex A to vertex J is 11.

Hence, the minimum distance from A to J is 11. There are two ways to reach J from A through D:

1. A→D→F→I→J
2. A→D→E→H→J

By building an optimal solution using sub-solutions, we demonstrate the Optimal Substructure Property.

Proof by Contradiction

Let's prove the above solution using proof by contradiction. Let Ra..j be the optimal path from node A to node J. Assume this optimal path passes through node k.

The path can be split into R a..k and R k..j.

Assume there is a shorter path from A to K, denoted as Ra..k′.

If Ra..k′<Ra..k ,

then: Ra..k′+Rk..j<Ra..k+Rk..j

This relation cannot be true, as we already know Ra..k+Rk..j is an optimal solution. Therefore, Ra..k′, a shorter path from A to K, does not exist, and Ra..k+Rk..j is indeed the optimal solution.

3.1.3 Strassens Matrix Multiplication

Problem Statement

Let us consider two matrices X and Y. We want to calculate the resultant matrix Z by multiplying X and Y.

Naïve Method

In the first place, we will examine the innocent strategy and its intricacy. Here, we are ascertaining Z = X × Y. Utilising the Naïve technique, two frameworks (X and Y) can be increased if the request for these grids is p × q and q × r. Following is the calculation.

Algorithm: Matrix-Multiplication (X, Y, Z)

```
for i = 1 to p do
 for j = 1 to r do
 Z[i,j] := 0
```

Notes

for k = 1 to q do

Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]

Complexity

Here, we expect that number tasks take O(1) time. There are three for circles in this calculation and one is settled in another. Consequently, the calculation takes O(n3) time to execute.

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n.

Divide X, Y and Z into four (n/2)×(n/2) matrices as represented below −

Z=[IKJL]Z=[IJKL] X=[ACBD]X=[ABCD] and Y=[EGFH]Y=[EFGH]

Using Strassen's Algorithm compute the following −

M1:=(A+C)×(E+F)M1:=(A+C)×(E+F)

M2:=(B+D)×(G+H)M2:=(B+D)×(G+H)

M3:=(A−D)×(E+H)M3:=(A−D)×(E+H)

M4:=A×(F−H)M4:=A×(F−H)

M5:=(C+D)×(E)M5:=(C+D)×(E)

M6:=(A+B)×(H)M6:=(A+B)×(H)

M7:=D×(G−E)M7:=D×(G−E)

Then,

I:=M2+M3−M6−M7I:=M2+M3−M6−M7

J:=M4+M6J:=M4+M6

K:=M5+M7K:=M5+M7

L:=M1−M3−M4−M5L:=M1−M3−M4−M5

Analysis

T(n)={c7xT(n2)+dxn2ifn=1otherwiseT(n)={cifn=17xT(n2)+dxn2 otherwise where c and d are constants

Using this recurrence relation, we get T(n)=O(nlog7)T(n)=O(nlog7)

Hence, the complexity of Strassen's matrix multiplication algorithm is O(nlog7) O(nlog7)

Drawbacks of Divide and Conquer

We now discuss some bottlenecks of Strassen's algorithm (and Divide and Conquer algorithms in general).

- We haven't considered communication bottlenecks; in real life communication is expensive.

Notes

- Disk/RAM differences are a bottleneck for recursive algorithms, and
- PRAM assumes perfect scheduling.

Communication Cost Our PRAM model assumes zero communication costs between processors. The reason is because the PRAM model assumes a shared memory model, in which each processor has fast access to a single memory bank. Realistically, we never have efficient communication, since oftentimes in the real world we have clusters of computers, each with its own private bank of memory. In these cases, divide and conquer is often impractical. It is true that when our data is split across multiple machines, having an algorithm operate on blocks of data at a time can be useful. However, as Strassen's algorithm continues to chop up matrices into smaller and smaller chunks, this places a large communication burden on distributed setups because after the first iteration, it is likely that we will incur a shuffle cost as we are forced to send data between machines.

Caveat - Big O and Big Constants One last caveat specific to Strassen's Algorithm is that in practice, the O(n2) term requires 20 · n2 operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really, we can often only afford to pass through the entire data set at one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big O notation is great to get you started and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

When is Strassen's worth it?

If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if n ≥ 1, 000, assuming no communication costs. Higher communication costs drive up the n at which Strassen's becomes useful very quickly. Even at n = 1, 000, naive matrix-multiply requires 1e9 operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea.

Disk Vs. RAM Trade-off

What is the reason that we can only pass through our data once?

There is a big trade-off between having data in ram and having it on disk. If we have tons of data, our data is stored on disk. We also have an additional constraint that with respect to streaming data, as the data are coming in, they are being stored in memory, i.e. we have fast random access, but once we store the data to disk retrieving it again is expensive

3.1.4 Matrix Chain Multiplication

Given a succession of lattices, locate the most effective approach to duplicate these grids together. The issue isn't really to play out the increases, however, simply to choose in which request to play out the duplications.

We have numerous choices to increase the chain of lattices since framework duplication is affiliated. As such, regardless of how we parenthesised the item, the outcome will be the equivalent. For instance, if we had four lattices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

Notes

Nonetheless, the request where we parenthesised the item influences the number of basic number-crunching activities expected to figure the item or the effectiveness. For instance, assume A will be a 10 × 30 grid, B is a 30 × 5 network, and C is a 5 × 60 framework. At that point,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Unmistakably the principal parenthesization requires fewer tasks.

Given an exhibit p[] which addresses the chain of lattices with the end goal that the ith grid Ai is of measurement p[i-1] x p[i]. We need to compose a capacity MatrixChainOrder() that should restore the base number of duplications expected to increase the chain.

Input: p[] = {40, 20, 30, 10, 30}

Output: 26000

There are 4 networks of measurements 40x20, 20x30, 30x10 and 10x30.

Leave the information 4 grids alone A, B, C, and D. The base number of augmentations are gotten by placing enclosure in after manner

(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}

Output: 30000

There are 4 frameworks of measurements 10x20, 20x30, 30x40 and 40x30.

Leave the information 4 networks alone A, B, C, and D. The base number of augmentations are gotten by placing enclosure in after manner

((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, the cost of which is 10*20*30

1) Optimal Substructure:

   A basic arrangement is to put a bracket at all potential spots, compute the expense for every situation and return the base worth. In a chain of grids of size n, we can put the initial set of enclosures in n-1 different ways. For instance, if the given chain is of 4 networks. leave the chain alone ABCD, at that point, there are 3 different ways to put the initial set of enclosure on the external side: (A)(BCD), (AB)(CD), and (ABC)(D). So when we place a bunch of enclosures, we partition the issue into subproblems of more modest size. Hence, the issue has ideal base property and can be handily addressed utilising recursion.

   Least number of increase expected to duplicate a chain of size n = Minimum of all n-1 situations (these positions make subproblems of more modest size)

2) Overlapping Subproblems

   Following is a recursive usage that essentially follows the above ideal foundation property.

Notes

3.1.5 Longest Common Subsequence

If a bunch of successions is given, the longest normal aftereffect issue is to locate a typical aftereffect of the multitude of arrangements that is of maximal length.

The longest normal aftereffect issue is an exemplary software engineering issue, the premise of information correlation projects, for example, the diff-utility, and has applications in bioinformatics. It is additionally broadly utilised by modification control frameworks, like SVN and Git, for accommodating various changes made to a correction-controlled assortment of documents.

This is a genuine illustration of the procedure of dynamic programming, which is the accompanying basic thought: start with a recursive calculation for the issue, which might be wasteful because it calls itself consistently on few subproblems. Essentially recall the answer for each subproblem on the first occasion when you process it, at that point after that find it as opposed to recomputing it. The general time-bound at that point turns out to be (ordinarily) relative to the number of particular subproblems instead of the bigger number of recursive calls. We previously saw this thought momentarily in the principal address.

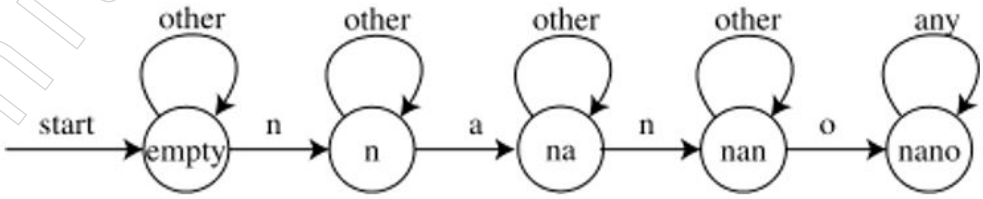
As we'll see, there are two different ways of doing dynamic programming, top-down and base up. The top-down (memoizing) technique is nearer to the first recursive calculation, so more clear, however, the base-up strategy is generally somewhat more productive.

Subsequence Testing

Before we characterise the longest regular aftereffect issue, we should begin with a simple warm up. Assume you're given a short string (design) and long string (text), as in the string coordinating issue. Be that as it may, presently you need to know whether the letters of the example show up altogether (however potentially isolated) in the content. On the off chance that they do, we say that the example is an aftereffect of the content.

For instance, is "nano" an aftereffect of "nematode information"? Indeed, and in just a single way. The most effortless approach to see this model is to underwrite the aftereffect: "NemAtode kNOwledge".

When all is said and done, we can test this before utilising a limited state machine. Draw circles and bolts as in the past, compared to fractional aftereffects (prefixes of the example), yet now there is no requirement for backtracking.



Equivalently, it is easy to write code or pseudocode for this:

```
subseq(char * P, char * T)
{
while (*T != '\0')
if (*P == *T++ && *++P == '\0')
return TRUE;
```

Notes

```
return FALSE;
}
```

Longest Common Subsequence Problem

Imagine a scenario where the example doesn't happen in the content. It bodes well to locate the longest aftereffect that happens both in the example and in the content. This is the longest regular aftereffect issue. Since the example and text have symmetric parts, from now into the foreseeable future we will not give them various names yet call them strings A and B. We'll use m to mean the length of An and n to indicate the length of B.

Note that the automata-hypothetical technique above doesn't tackle the issue - rather it gives the longest prefix of A that is an aftereffect of B. Be that as it may, the longest normal aftereffect of An and B isn't generally a prefix of A.

For what reason may we need to take care of the longest regular aftereffect issue? There are a few rousing applications.

Atomic science. DNA groupings (qualities) can be addressed as successions of four letters ACGT, compared to the four submolecular shaping DNA. At the point when scholars locate another grouping, they ordinarily need to understand what different successions it is generally like. One method of figuring out how comparable two groupings are to discover the length of their longest basic aftereffect.

Document examination. The Unix program "diff" is used to look at two changed adaptations of a similar record, to figure out what changes have been made to the document. It works by finding the longest normal aftereffect of the lines of the two records; any line in the aftereffect has not been changed, so what it shows is the leftover arrangement of lines that have changed. On this occasion of the difficulty, we should consider each line of a record being a solitary confounded character in a string.

Screen redisplay. Numerous word processors like "emacs" show part of a record on the screen, refreshing the screen picture as the document is changed. For moderate dial-in terminals, these projects need to send the terminal a couple of characters as conceivable to make it update its presentation effectively. It is conceivable to see the calculation of the base length grouping of characters expected to refresh the terminal similar to such a typical aftereffect issue (the basic aftereffect discloses to you the pieces of the showcase that are as of now right and don't should be changed).

(As an aside, it is normal to characterise a comparable longest basic substring issue, requesting the longest substring that shows up in two info strings. This issue can be settled in straight time utilising an information structure known as the addition tree however the arrangement is amazingly confounded.)

Recursive Solution

So, we want to solve the longest common subsequence problem by dynamic programming.

To do this, we first need a recursive arrangement. The unique programming thought doesn't reveal to us how to discover this, it simply gives us a method of making the arrangement more productive once we have.

We should begin with certain straightforward perceptions about the LCS issue. On the off chance that we have two strings, say "nematode information" and "void jug", we can address an aftereffect as a method of composing the two so certain letters line up:

Notes

n e m a t o d e k n o w l e d g e

| | | | | | |

e m p t y b o t t l e

If we draw lines interfacing the letters in the main string to the relating letters in the second, no two lines cross (the top and base endpoints happen in a similar request, the request for the letters in the aftereffect). Alternately any arrangement of lines drawn this way, without intersections, addresses an aftereffect.

From this we can notice the accompanying straightforward actuality: if the two strings start with a similar letter, it's consistently protected to pick that beginning letter as the principal character of the aftereffect. This is because, on the off chance that you have some other aftereffect, addressed as an assortment of lines as drawn above, you can "push" the furthest left line to the beginning of the two strings, without causing some other intersections, and get a portrayal of a similarly long aftereffect that begins thusly.

Then again, assume that similar to the model over, the two first character's contrast. At that point it isn't workable for the two of them to be important for a typical aftereffect - either (or perhaps both) should be eliminated.

Finally, see that whenever we've chosen how to manage the main characters of the strings, the leftover subproblem is again the longest regular after-effect issue, on two more limited strings. Accordingly, we can settle it recursively.

As opposed to finding the actual aftereffect, it ends up being more effective to discover the length of the longest aftereffect. At that point for the situation where the primary characters vary, we can figure out which subproblem gives the right arrangement by settling both and taking the maximum of the subsequent aftereffect lengths. When we transform this into a unique program we will perceive how to get the actual arrangement.

These perceptions give us the accompanying, wasteful, recursive calculation.

Recursive LCS:

```
int lcs_length(char * A, char * B)
{
if (*A == '\0' || *B == '\0') return 0;
else if (*A == *B) return 1 + lcs_length(A+1, B+1);
else return max(lcs_length(A+1,B), lcs_length(A,B+1));
}
```

This is the correct solution but it's very time-consuming. For example, if the two strings have no matching characters, so the last line always gets executed, the time bounds are binomial coefficients, which (if m=n) are close to 2^n.

3.1.6 0/1 Knapsack Problem

In the 0–1 Knapsack problem, we are given a bunch of things, each with a weight and a worth, and we need to decide the quantity of everything to remember for an assortment so the complete weight is not exactly or equivalent to a given cutoff and the absolute worth is pretty much as extensive as could be expected.

Notes

Please note that the items are indivisible; we can either take an item or not (0-1 property). For example,

Input:

```
value = [ 20, 5, 10, 40, 15, 25 ]
weight = [ 1, 2, 3, 8, 7, 4 ]
int W = 10

Output: Knapsack value is 60

value = 20 + 40 = 60
weight = 1 + 8 = 9 < W
```

Knapsack Problem:

The backpack is essentially an implying sack. A pack of given limits.

We need to pack n things in your baggage.

The ith thing is worth vi dollars and weight wi pounds.

Take as important a heap as could be expected yet can't surpass W pounds.

vi wi W are whole numbers.

1. W ≤ capacity
2. Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximise profit and minimise weight incapacity.

The knapsack problem is where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by Greedy Strategy whereas the 0 /1 problem is not.

It cannot be solved by the Dynamic Programming Approach.

0/1 Knapsack Problem:

This thing can't be broken, which implies criminals should accept the thing all in all or should leave it. That is the reason it is called the 0/1 backpack Problem.

Everything is taken or not taken.

Can't take a fragmentary measure of a thing taken or take a thing more than once.

It can't be tackled by the Greedy Approach since it is empowering to fill the rucksack.

Notes

The Greedy Approach doesn't guarantee an Optimal Solution.

Example of 0/1 Knapsack Problem:

Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$												
$w_2 = 2\ v_2 = 6$												
$w_3 = 5\ v_3 = 18$												
$w_4 = 6\ v_4 = 22$												
$w_5 = 7\ v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0											
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

V [i, j] = max {V [i - 1, j], vi + V [i - 1, j -wi]

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

V [3, 7] = max {V [3 - 1, 7], v3 + V [3 - 1, 7 - w3]

= max {V [2, 7], 18 + V [2, 7 - 5]}

= max {7, 18 + 6}

= 24

Notes

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

Algorithm of Knapsack Problem

KNAPSACK (n, W)

1. for w = 0, W
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if (wi≤ w & vi + V [i-1, w - wi]> V [i -1,W])
7. then V [i, W] ← vi + V [i - 1, w - wi]
8. else V [i, W] ← V [i - 1, w]

3.2. Greedy Approach

The Greedy Method is a strategy used in optimisation problems to make a series of decisions, each of which looks the best at the moment. Unlike some other strategies such as Divide and Conquer, which break down problems into smaller subproblems and solve each part recursively, the Greedy Method focuses on making a sequence of choices that seem optimal at the current step, without reconsidering those choices later.

Key Concepts

1. Optimization Problem: An optimization problem is one where the goal is to find the best solution from all feasible solutions. This could involve maximising or minimising some quantity, such as cost, distance, or time.
2. Feasible Solution: A feasible solution is one that satisfies all the problem's constraints. It is a valid solution but not necessarily the best.

Notes

3. Optimal Solution: The optimal solution is the best feasible solution according to the given criteria (e.g., the minimum cost, maximum profit).

Characteristics of the Greedy Method

- Local Optimum: The Greedy Method makes decisions based on the best choice available at the current step.
- Global Optimum: The method does not guarantee a globally optimal solution in all cases, but it often finds a good approximation.
- Simple and Straightforward: The method is generally easy to understand and implement.

3.2.1 Introduction of Greedy Approach

A calculation is intended to accomplish an ideal answer for a given issue. In the eager calculation approach, choices are produced using the given arrangement space. As being insatiable, the nearest arrangement that appears to give an ideal arrangement is picked.

Greedy is an algorithmic prototype that builds up a solution piece by piece, always choosing the next piece that offers the most apparent and immediate benefit. So, the problems were deciding locally optimal also leads to the global solution are best fit for Greedy.

Eager algorithms attempt to locate a restricted ideal arrangement, which may ultimately prompt worldwide upgraded arrangements. Notwithstanding, for the most part, the avaricious algorithm doesn't give all around the world improved arrangements

Counting Coins

This issue is total to an ideal incentive by picking the most un-potential coins and the insatiable methodology powers the calculation to pick the biggest conceivable coin. If we are given coins of ₹ 1, 2, 5, and 10 and we are approached to tally ₹ 18 then the insatiable method will be −

- 1 − Select one ₹ 10 coin, the remaining count is 8
- 2 − Then select one ₹ 5 coin, the remaining count is 3
- 3 − Then select one ₹ 2 coin, the remaining count is 1
- 4 − And finally, the selection of one ₹ 1 coin solves the problem

However, it is by all accounts turned out great, for this tally we need to pick just 4 coins. Be that as it may, if we marginally change the issue, a similar methodology will most likely be unable to deliver a similar ideal outcome.

For the money framework, where we have coins of 1, 7, 10 worth, tallying coins for esteem 18 will be ideal yet for a tally like 15, it might utilise a greater number of coins than needed. For instance, the ravenous methodology will utilise 10 + 1 + 1 + 1 + 1 + 1, complete 6 coins. While a similar issue could be addressed by utilising just 3 coins (7 + 7 + 1)

Henceforth, we may reason that the greedy methodology picks a prompt enhanced arrangement and may bomb where worldwide improvement is a significant concern.

Examples

Most networking algorithms use the greedy approach. Here is a list of a few of them −

Notes

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimum Spanning Tree Algorithm
- Dijkstra's Minimum Spanning Tree Algorithm
- Graph - Map Colouring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

There are lots of similar problems that use the greedy approach to find an optimum solution.

3.2.2 Activity Selection Problem

Greedy is an algorithmic worldview that develops an answer piece by piece, continually picking the following piece that offers the most evident and quick advantage. Ravenous algorithms are utilised for improvement issues. An enhancement issue can be tackled utilising Greedy if the issue has the accompanying property: At each progression, we can settle on a decision that takes a gander right now, and we get the ideal arrangement of the total issue.

If a Greedy Algorithm can tackle any issue, it for the most part turns into the best strategy to take care of that issue as the Greedy algorithm is as a rule more productive than different procedures like Dynamic Programming. However, the Greedy algorithm can't generally be applied. For instance, the Fractional Knapsack issue (See this) can be addressed utilising Greedy, however, 0-1 Knapsack can't be settled utilising Greedy.

What is the Activity Selection Problem?

How about we consider that you have n exercises with their beginning and finish times, the goal is to discover an arrangement set having the greatest number of non-clashing exercises that can be executed in a solitary period, accepting that just a single individual or machine is accessible for execution.

A few focuses to note here:

- It probably won't be conceivable to finish all the exercises, since their timings can fall.

  Two exercises, say i and j, are supposed to be non-clashing if si >= fj or sj >= fi where si and sj signify the beginning season of exercises I and j separately, and fi and fj allude to the completing season of the exercises I and j individually.
- The insatiable methodology can be utilised to discover the arrangement since we need to augment the tally of exercises that can be executed. This methodology will insatiably pick an action with the most punctual completion time at each progression, along these lines yielding an ideal arrangement.

Algorithm

The algorithm of Activity Selection is as follows:

Activity-Selection(Activity, start, finish)

Sort Activity by finish times stored in finish

Notes

Selected = {Activity[1]}

n = Activity.length

j = 1

for i = 2 to n:

if start[i] ≥ finish[j]:

Selected = Selected U {Activity[i]}

j = i

return Selected

Complexity

Time Complexity:

When activities are sorted by their finish time: O(N)

When activities are not sorted by their finish time, the time complexity is O(N log N) due to the complexity of sorting

Example

	0	1	2	3	4	5	6
START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[1]>=END[0], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[2]<END[1], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[3]<END[1], REJECTED

Notes

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[4] >=END[2], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[5]<END[4], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[6]>=END[4], SELECTED

In this example, we take the start and finish time of activities as follows:

start = [1, 3, 2, 0, 5, 8, 11]

finish = [3, 4, 5, 7, 9, 10, 12]

Sorted by their finish time, activity 0 gets selected. As activity 1 has a starting time that is equal to the finish time of activity 0, it gets selected. Activities 2 and 3 have a smaller starting time than the finish time of activity 1, so they get rejected. Based on similar comparisons, activities 4 and 6 also get selected, whereas activity 5 gets rejected. In this example, in all the activities 0, 1, 4, and 6 get selected, while others get rejected.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

- Step 1: Sort the given activities in ascending order according to their finishing time.
- Step 2: Select the first activity from sorted array act[] and add it to sol[] array.
- Step 3: Repeat steps 4 and 5 for the remaining activities in act[].
- Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to the sol[] array.
- Step 5: Select the next activity in the act[] array.
- Step 6: Print the sol[] array.

Activity Selection Problem Example

Let's try to trace the steps of the above algorithm using an example:

In the table below, we have 6 activities with the corresponding start and end time, the objective is to compute an execution schedule having a maximum number of non-conflicting activities:

Notes

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible solution would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus sol = {a2}.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity a3. Since the start time of a3 is greater than the finish time of a2 (i.e. s(a3) > f(a2)), we add a3 to the solution set. Thus sol = {a2, a3}.

B. Select a4. Since s(a4) < f(a3), it is not added to the solution set.

C. Select a5. Since s(a5) > f(a3), a5 gets added to solution set. Thus sol = {a2, a3, a5}

D. Select a1. Since s(a1) < f(a5), a1 is not added to the solution set.

E. Select a6. a6 is added to the solution set since s(a6) > f(a5). Thus sol = {a2, a3, a5, a6}.

Step 6: At last, print the array sol[]

Hence, the execution schedule of the maximum number of non-conflicting activities will be:

(1,2)

(3,4)

(5,7)

(8,9)

Notes

3.2.3 Huffman Code

A helpful application for avaricious calculations is for pressure—putting away pictures or words with the least measure of pieces.

1. Illustration of coding letters (wastefully)-

   A - > 00 (“code word”)

   B - > 01

   C - > 10

   D - > 11

   AABABACA is coded by:

   0000010001001000

   This is inefficient; a few characters may show up more frequently than others (for

   For example, if “A” is more regular than “D” in the English language or a coursebook that you are coding and wish to pack). Be that as it may, in the methodology most important characters are addressed with the equivalent number of pieces.

2. More productive: on the off chance that a few characters show up more regularly, we can code them with a more limited length in bits. Suppose A shows up more regularly and then B.

   A - > 0 (regular)

   B - > 10

   C - > 110

   D - > 111 (less regular)

   AABABACA is coded by:

   001001001100

We addressed similar grouping with fewer pieces = pressure. This is a variable length code.

This is for example an application for the English language (“a” more successive than “q”).

This sort of approach of coding more successive characters with fewer pieces is very helpful in pressure, and comparative methodologies likewise apply to the pressure of pictures.

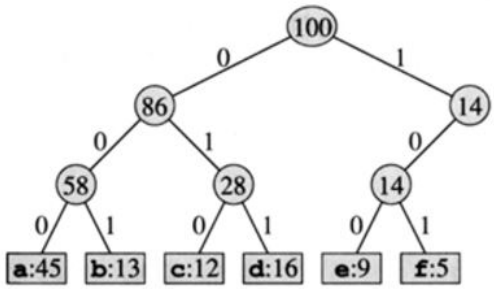
Prefix codes: We consider just codes in which no codeword is a prefix for the another one (= a beginning for the other one).

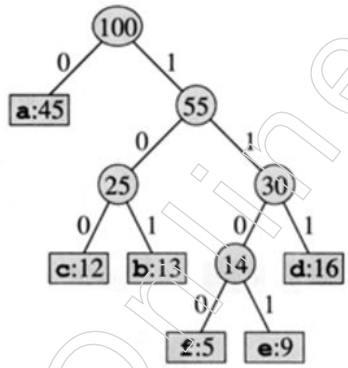
[so, we’re truly just considering non-prefix codes, albeit that is the word that is used]

Prefix codes are helpful because as we’ll see, they are simpler to interpret (go from 001001001100 to the characters).

Trees corresponding to the coding examples

Notes





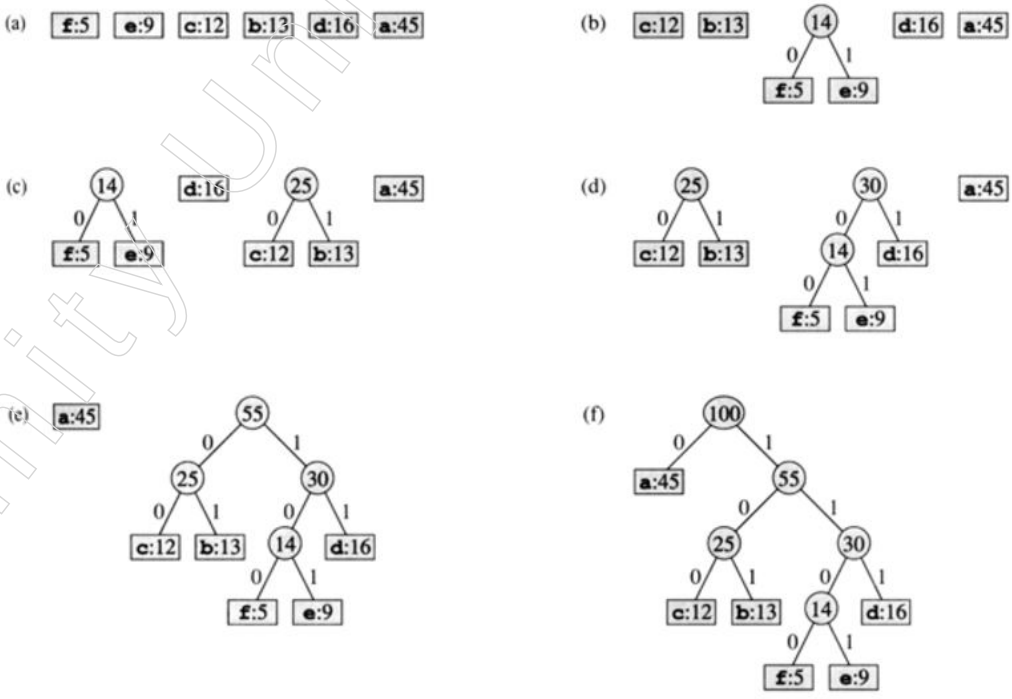
A few notes on the trees:

- Later: how to build trees and the recurrence hubs.
- We can without much of a stretch use trees for translating – continue to go down until arriving at a leaf as you experience 101 (b) 0 (a) 1101 (f) (in the variable length).
- Variable length is a full twofold tree (two youngsters for each hub until they reach leaves). Fixed length isn't.

Principle insatiable methodology for building the Huffman tree: Begins with a bunch of leaves, and each time distinguishes the two least continuous items to combine.

At the point when we combine the two articles, the outcome is currently an item whose total is the recurrence of the blended items.

Model building Huffman code tree:



The means of Huffman's calculation for the frequencies are given in above Figure. Each part shows the substance of the line arranged into an expanding request by recurrence. At each progression, the two trees with the most minimal frequencies are combined. Leaves have appeared as square shapes containing a character; what's

more, it's recurrence. Inside hubs have appeared as circles containing the amount of the frequencies of their youngsters. An edge interfacing an inward hub with its youngsters is named 0 if it is an edge to one side youngster and 1 if it is an edge to a correct kid. The codeword for a letter is the arrangement of names on the edges interfacing the root to the leaf for that letter. (a) The underlying arrangement of n D 6 hubs, one for each letter. (b)– (e) Intermediate stages. (f) The last tree

Pseudo code:

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

The principle thought Huffman pseudo-code: Here C is the characters. Over and again removes two lowest frequencies from a needed line Q (eg, a stack) and consolidates them as another hub in the line. Toward the end, restores the one hub left in the line, which is the ideal tree.

Run time: Huffman code:

For circle runs n-multiple times O(n)

Each extricating min requires O(log n)

Complete: O(n log n)

(the line additionally should be introduced with the characters and their frequencies, in any case, doesn't change the general run time)

3.2.4 Fractional Knapsack Problem

The Greedy algorithm could be seen very well with a notable issue alluded to as the Knapsack issue. Albeit a similar issue could be settled by utilising other algorithmic methodologies, the Greedy methodology takes care of the Fractional Knapsack issue sensibly in a happy time.

Given a bunch of things, each with a weight and a worth, decide a subset of things to remember for an assortment so the all-out weight is not exactly or equivalent to a given cutoff and the complete worth is just about as extensive as could really be expected.

The knapsack problem is a combinatorial advancement issue. It shows up as a subproblem in many, more mind-boggling numerical models of genuine issues. One general way to deal with troublesome issues is to recognise the most prohibitive requirement, disregard the others, take care of a backpack issue, and by one way or another change the answer to fulfil the overlooked limitations.

Notes

Applications

In many cases of resource allocation along with some constraints, the problem can be derived in a similar way to the Knapsack problem. Following is a set of examples.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A cheat is ransacking a store and can convey a maximal load of W into his rucksack. There are no things accessible in the store and the weight of ith thing is wi and its benefit is pi. What things should the cheat take?

In this unique circumstance, the things ought to be chosen so that the cheat will convey those things for which he will acquire the most extreme benefit. Henceforth, the goal of the hoodlum is to amplify the benefit.

Based on the nature of the items, Knapsack problems are categorised as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of ith item wi>0wi>0
- Profit for ith item pi>0pi>0 and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction xi of ith item.

0<xi<10<x<1

The ith item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to maximise∑n=1n(xi.pi) maximise∑n=1n(xi.pi)

subject to constraint,∑n=1n(xi.wi)<W∑n=1n(xi.wi)<W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by ∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwi..piwi . Here, x is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

for i = 1 to n

```
 do x[i] = 0
weight = 0
for i = 1 to n
 if weight + w[i] ≤ W then
 x[i] = 1
 weight = weight + w[i]
 else
 x[i] = (W - weight) / w[i]
 weight = W
 break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of piwi, then the while loop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Example

Let us consider that the capacity of the knapsack W = 60 and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (piwi)(piwi)	7	10	6	5

As the provided items are not sorted based on piwipiwi. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (piwi)(piwi)	10	7	6	5

Solution

After sorting all the items according to piwipiwi. First all of B is chosen as the weight of B is less than the capacity of the knapsack. Next, item A is chosen, as the available capacity of the knapsack is greater than the weight of A. Now, C is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of C.

Hence, fraction of C (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more items can be selected.

The total weight of the selected items is 10 + 40 + 20 * (10/20) = 60

Notes

And the total profit is 100 + 280 + 120 * (10/20) = 380 + 60 = 440

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

3.3 Branch & Bound Approach

The Branch and Bound Algorithm is a powerful method used to solve combinatorial optimization problems. It systematically searches for the optimal solution by dividing the main problem into smaller subproblems (branches) and then eliminating certain branches based on bounds on the optimal solution. This approach is particularly useful for solving problems where the solution space is too large to be explored exhaustively. It is commonly applied to problems such as the travelling salesman problem (TSP) and job scheduling.

Key Concepts

1. Branching: This involves dividing the problem into smaller subproblems. Each subproblem represents a partial solution, which will be further explored.
2. Bounding: This step involves calculating bounds (upper or lower limits) for the subproblems. If a bound of a subproblem indicates that it cannot produce a better solution than the current best, it is discarded (pruned).
3. Pruning: This process involves eliminating subproblems that do not have the potential to yield a better solution than the current best-known solution, thereby reducing the number of branches that need to be explored.

Steps in Branch and Bound Algorithm

1. Initialization: Start with the initial problem and set an initial bound. This could be an initial guess of the best solution.
2. Branching: Divide the problem into subproblems. This can be done recursively, and each subproblem represents a decision point in the solution space.
3. Bounding: For each subproblem, calculate a bound on the best possible solution that can be obtained.
4. Pruning: Compare the bound of each subproblem with the current best solution. If the bound is worse, prune the subproblem.
5. Selection: Select the next subproblem to explore based on a strategy (e.g., depth-first, breadth-first).
6. Update: If a better solution is found, update the current best solution and adjust the bounds of other subproblems if necessary.
7. Termination: The algorithm terminates when there are no more subproblems to explore or a predefined condition is met (e.g., a satisfactory solution is found).

3.3.1 Travelling Salesman Problem

As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

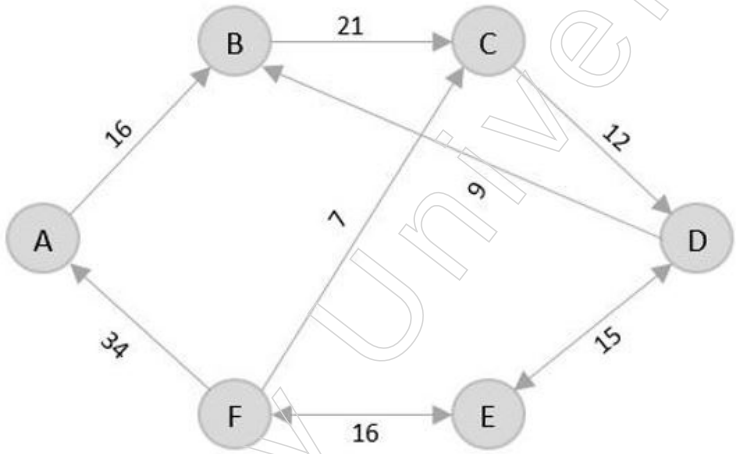
Notes

Algorithm

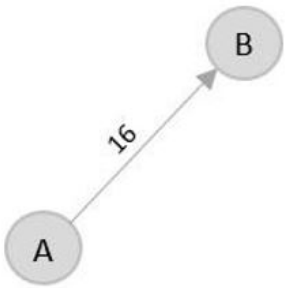
- Travelling salesman problem takes a graph G {V, E} as an input and declares another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples

Consider the following graph with six cities and the distances between them −

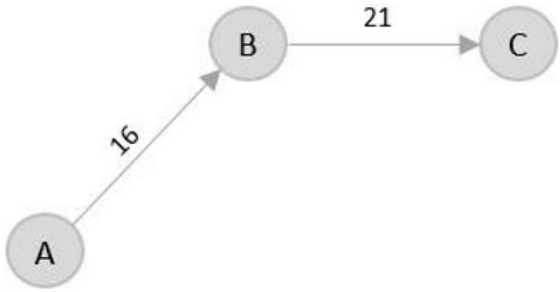


From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.

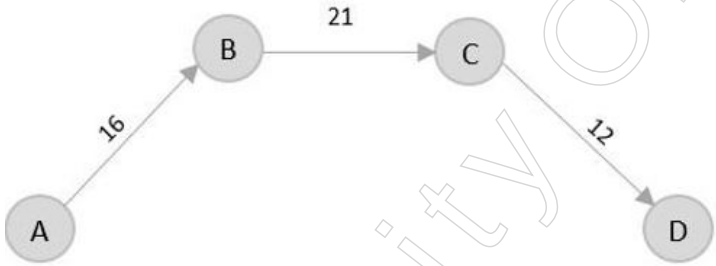


Then, B → C has the shortest and only edge between, therefore it is included in the output graph.

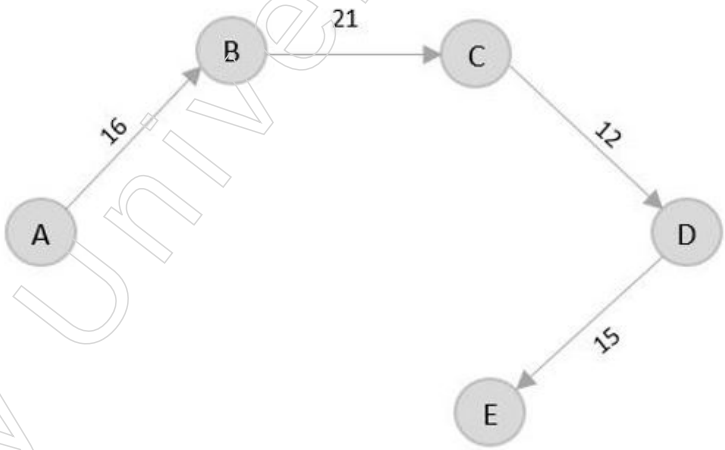
Notes



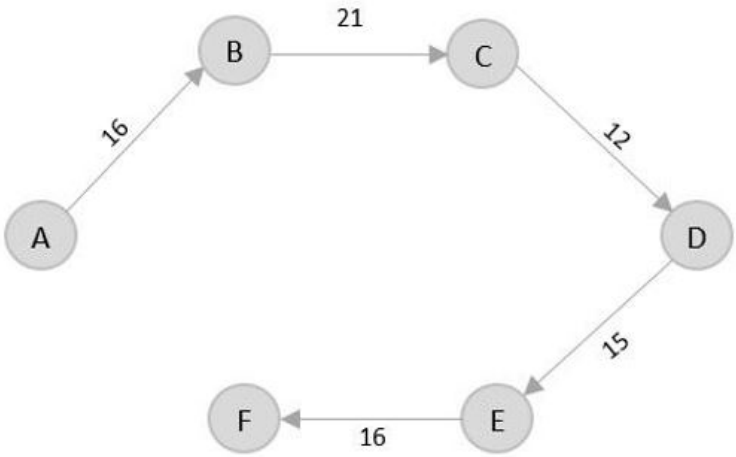
There's only one edge between C → D, therefore it is added to the output graph.



There's two outward edges from D. Even though D → B has a lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.

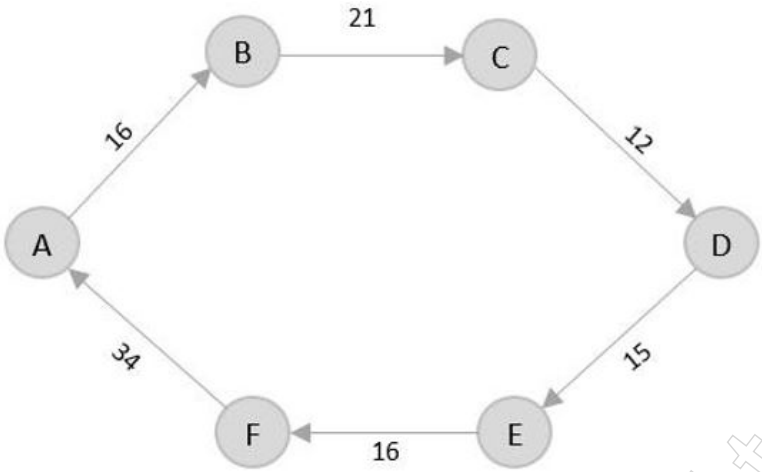


There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Notes

Again, even though F → C has a lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though the cost of a path could be decreased if it originates from other nodes, the question is not raised with respect to that.

3.3.2 Knapsack Problem

The backpack is essentially an implying sack. A pack of given limits. We need to pack n things in your baggage. The ith thing is worth vi dollars and weight wi pounds.

Take as important a heap as could be expected, yet can't surpass W pounds.

- vi wi W are whole numbers.
- W ≤ capacity
- Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output:

To maximise profit and minimise weight incapacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by Greedy Strategy whereas the 0 /1 problem is not.

   It cannot be solved by the Dynamic Programming Approach.

3.3.3 Job Sequencing

Given an array of jobs where each work has a cutoff time and related benefit if the

Notes

work is done before the cutoff time. It is likewise given that each work takes a solitary unit of time, so the base conceivable cutoff time for any work is 1. The most effective method to augment absolute benefit if just each work can be planned for turn.

Example

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	5

Output: Following is maximum profit sequence of jobs

c, a, e

A Simple Solution is to produce all subsets of a given arrangement of occupations and check singular subsets for the achievability of occupations around there. Monitor greatest benefit among every practical subset. The time intricacy of this arrangement is dramatic.

Following is the algorithm

1) Sort all jobs in decreasing order of profit.

2) Iterate on jobs in decreasing order of profit.For each job , do the following :

```
// Program to find the maximum profit job sequence from a
given array
// of jobs with deadlines and profits
#include<iostream>
#include<algorithm>
using namespace std;

// A structure to represent a job
struct Job
{
char id;  // Job Id
```

Notes

```
    int dead; // Deadline of job
    int profit; // Profit if job is over before or on deadline
    };

    // This function is used for sorting all jobs according to
profit
    bool comparison(Job a, Job b)
    {
    return (a.profit > b.profit);
    }

    // Returns minimum number of platforms required
    void printJobScheduling(Job arr[], int n)
    {
    // Sort all jobs according to decreasing order of profit
    sort(arr, arr+n, comparison);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n]; // To keep track of free time slots

    // Initialise all slots to be free
    for (int i=0; i<n; i++)
    slot[i] = false;

    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
    // Find a free slot for this job (Note that we start
    // from the last possible slot)
    for (int j=min(n, arr[i].dead)-1; j>=0; j--)
    {
    // Free slot found
    if (slot[j]==false)
    {
    result[j] = i; // Add this job to result
    slot[j] = true; // Make this slot occupied
    break;
    }
    }
    }

    // Print the result
    for (int i=0; i<n; i++)
```

Notes

```
if (slot[i])
cout << arr[result[i]].id << " ";
}

// Driver code
int main()
{
Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},
{'d', 1, 25}, {'e', 3, 15}};
int n = sizeof(arr)/sizeof(arr[0]);
cout << "Following is maximum profit sequence of jobs \n";

// Function call
printJobScheduling(arr, n);
return 0;
}
```

Output

Following is maximum profit sequence of jobs

c a e

The Time Complexity of the above solution is O(n2). It can be optimised using Disjoint Set Data Structure. Please refer to the below post for details.

3.3.4 Job Sequencing with Deadline

In the job sequencing issue, the goal is to get a grouping of lines of work, which is finished inside their cutoff times and gives the most extreme benefit.

Solution

Let us consider a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, the deadline of ith job Ji is di and the profit received from this job is pi. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, D(i)>0 for 1>i>n.

Initially, these jobs are ordered according to profit, i.e. p1>p2>p3>...>pn.

Algorithm:

```
Job-Sequencing-With-Deadline (D, J, n, k)
D(0) := J(0) := 0
k := 1
J(1) := 1 // means first job is selected
```

Notes

```
for i = 2 … n do
r := k
while D(J(r)) > D(i) and D(J(r)) ≠ r do
r := r – 1
if D(J(r)) ≤ D(i) and D(i) > r then
for l = k … r + 1 by -1 do
J(l + 1) := J(l)
J(r + 1) := i
k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n2).

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J2	J1	J4	J3	J5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J2, as it can be completed within its deadline and contributes maximum profit.

- Next, J1 is selected as it gives more profit compared to J4.
- In the next clock, J4 cannot be selected as its deadline is over, hence J3 is selected as it executes within its deadline.
- The job J5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J2, J1, J3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is 100 + 60 + 20 = 180.

Summary

- Dynamic Programming (DP) is a technique employed in both mathematics and computer science for tackling intricate problems by breaking them down into simpler subproblems.
- The dynamic programming approach is like separation and vanquish in separating

Notes

the issue into more modest but then more modest conceivable sub-issues. However, in contrast, to separate and vanquish, these sub-issues are not addressed freely.

- Richard Bellman, a renowned scientist, introduced the concept of dynamic programming in 1957. One of the key concepts in dynamic programming is the Principle of Optimality, which can be understood through the Optimal Substructure Property.
- The longest normal aftereffect issue is an exemplary software engineering issue, the premise of information correlation projects, for example, the diff-utility, and has applications in bioinformatics.
- In the 0–1 Knapsack problem, we are given a bunch of things, each with a weight and a worth, and we need to decide the quantity of everything to remember for an assortment so the complete weight is not exactly or equivalent to a given cutoff and the absolute worth is pretty much as extensive as could be expected.
- The Greedy Method is a strategy used in optimisation problems to make a series of decisions, each of which looks the best at the moment.
- The knapsack problem is a combinatorial advancement issue. It shows up as a subproblem in many, more mind boggling numerical models of genuine issues.
- The Branch and Bound Algorithm is a powerful method used to solve combinatorial optimization problems. It systematically searches for the optimal solution by dividing the main problem into smaller subproblems (branches) and then eliminating certain branches based on bounds on the optimal solution.
- In the job sequencing issue, the goal is to get a grouping of lines of work, which is finished inside their cutoff times and gives the most extreme benefit.

Glossary

- Optimization Problem: An optimization problem is one where the goal is to find the best solution from all feasible solutions. This could involve maximising or minimising some quantity, such as cost, distance, or time.
- Feasible Solution: A feasible solution is one that satisfies all the problem's constraints. It is a valid solution but not necessarily the best.
- Optimal Solution: The optimal solution is the best feasible solution according to the given criteria (e.g., the minimum cost, maximum profit).
- Branching: This involves dividing the problem into smaller subproblems. Each subproblem represents a partial solution, which will be further explored.
- Bounding: This step involves calculating bounds (upper or lower limits) for the subproblems. If a bound of a subproblem indicates that it cannot produce a better solution than the current best, it is discarded (pruned).
- Pruning: This process involves eliminating subproblems that do not have the potential to yield a better solution than the current best-known solution, thereby reducing the number of branches that need to be explored.

Check Your Understanding

1. What is one of the key strategies employed by Dynamic Programming (DP) to enhance efficiency?
   a) Randomising the solution process
   b) Storing solutions to subproblems

Notes

c) Ignoring subproblems

d) Solving each subproblem multiple times

2. What is the significance of the Principle of Optimality in dynamic programming?

a) It emphasises the importance of random selection in algorithm design

b) It ensures that each subproblem is solved only once to reduce computation time

c) It promotes the use of divide-and-conquer strategies for problem-solving

d) It advocates for the use of heuristic methods over exact algorithms

3. Which of the following problems can be solved using the Greedy Approach?

a) Fractional Knapsack Problem

b) 0/1 Knapsack Problem

c) Longest Common Subsequence Problem

d) Travelling Salesman Problem

4. Which step in the Branch and Bound Algorithm involves eliminating subproblems that cannot yield a better solution than the current best-known solution?

a) Initialization

b) Branching

c) Bounding

d) Pruning

5. What is the objective of the Job Sequencing problem?

a) To minimise the number of jobs scheduled

b) To maximise the total deadline of all jobs

c) To maximise the total profit by scheduling jobs within their deadlines

d) To minimise the total profit by scheduling jobs after their deadlines

Exercise

1. What are the key differences between dynamic programming and the divide and conquer approach?
2. Explain why a greedy algorithm might not always provide an optimal solution. Can you provide an example?
3. Describe the branch and bound approach and give an example of a problem where this method is particularly useful.
4. How does memoization work in dynamic programming, and how does it improve the efficiency of algorithms?
5. What is the greedy-choice property, and how is it used to design a greedy algorithm?

Learning Activities

1. Students will be given a set of classic dynamic programming problems (e.g., knapsack problem, longest common subsequence) to solve. They will write code to implement the solutions and then discuss the time complexity and space complexity of their implementations.

Notes

2. In this activity, students will be given a problem that can be solved using both a greedy algorithm and dynamic programming. They will implement both solutions, compare their results, and discuss which approach is more efficient and why.

Check Your Understanding Answer

1. b) 2. b) 3. a) 4. d)
5. c)

Further Readings and Bibliography

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Kleinberg, J., & Tardos, É. (2005). Algorithm Design. Pearson.
3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
4. Papadimitriou, C. H., & Steiglitz, K. (1998). Combinatorial Optimization: Algorithms and Complexity. Dover Publications.
5. Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman.
6. Mitzenmacher, M., & Upfal, E. (2005). Probability and Computing: Randomised Algorithms and Probabilistic Analysis. Cambridge University Press.

Module -IV: E String Matching Algorithms and Approximation Algorithms

Learning Objectives

At the end of this module, you will be able to:

- Understand the fundamental principles behind string matching algorithms and their significance in computer science.
- Learn about various exact string matching algorithms, including the Naive algorithm, Knuth-Morris-Pratt (KMP) algorithm, and Boyer-Moore algorithm.
- Explore approximate string matching algorithms such as the Rabin-Karp algorithm and the Aho-Corasick algorithm.
- Understand the concept of approximation algorithms and their role in solving NP-complete and optimization problems.
- Explore examples of approximation algorithms such as the Vertex Cover Problem, Travelling Salesman Problem, and Set Covering Problem.

Introduction

String matching algorithms play a crucial role in computer science, enabling efficient searching of patterns within larger texts. These algorithms have a wide range of applications, including database searching, network security, and text processing.

4.1 String Matching Algorithm

String matching algorithms can be broadly classified into two categories:

1. Exact String Matching Algorithms
2. Approximate String Matching Algorithms

1. Exact String Matching Algorithms

Exact string matching algorithms aim to find one or more occurrences of a defined pattern within a larger text such that each matching is perfect. In other words, all characters of the pattern must match the corresponding characters in the text exactly.

Categories of Exact String Matching Algorithms:

- Algorithms based on Character Comparison:
  - ❖ Naive Algorithm: This straightforward algorithm slides the pattern over the text one character at a time and checks for a match. If a match is found, it slides by one character and continues checking for subsequent matches.
  - ❖ KMP (Knuth Morris Pratt) Algorithm: The KMP algorithm avoids redundant comparisons by utilising information about previous matches. It pre-processes the pattern to create a partial match table (also known as the “failure function”) which helps in bypassing unnecessary comparisons when a mismatch occurs.
  - ❖ Boyer Moore Algorithm: This algorithm improves efficiency by starting the comparison from the last character of the pattern and using two heuristics (Bad Character and Good Suffix) to skip sections of the text, thus reducing the number of comparisons.

Notes

- ❖ Trie Data Structure: A trie is an efficient data structure for storing a set of strings. It allows quick retrieval of any string in the set by character-wise comparisons from the root to the leaf nodes, representing individual characters.

- Deterministic Finite Automaton (DFA) Method:
  - ❖ Automaton Matcher Algorithm: This algorithm constructs a finite automaton for the pattern. Starting from the initial state, it processes each character of the text sequentially, transitioning between states based on the current character and the automaton's predefined transitions. When the automaton reaches an accepting state, a match is found.
- Algorithms based on Bit (Parallelism Method):
  - ❖ Aho-Corasick Algorithm: This algorithm is used for multiple pattern matching. It constructs a trie for the given set of patterns and then transforms it into a finite automaton with failure links. This allows it to find all occurrences of any pattern in the text in linear time.
- Hashing-String Matching Algorithms:
  - ❖ Rabin Karp Algorithm: The Rabin Karp algorithm uses a hash function to compute the hash value of the pattern and compares it with the hash values of substrings of the text. If a hash value matches, a character-by-character comparison is performed to confirm the match.

2. Approximate String Matching Algorithms

   Approximate string matching algorithms allow for inexact matches, which is useful in scenarios where minor differences between the pattern and text are acceptable. These algorithms are often used in applications like DNA sequence analysis, spell checking, and data retrieval where minor errors or variations are expected.

- Applications in Real World
  - ❖ Database Searching: String matching algorithms are essential for querying databases efficiently. They help in quickly locating records that match a given pattern, which is crucial for database management systems (DBMS).
  - ❖ Network Security: In network systems, string matching algorithms are used for intrusion detection by matching patterns against a database of known attack signatures. This helps in identifying and mitigating security threats in real time.
  - ❖ Text Processing: Applications such as text editors, search engines, and word processors rely on string matching algorithms to find and replace text, search for specific patterns, and perform text analysis.
  - ❖ Bioinformatics: In the field of bioinformatics, approximate string matching algorithms are used to compare DNA, RNA, and protein sequences, allowing researchers to identify similarities and differences between biological sequences.
  - ❖ Spam Filtering: Email systems use string matching algorithms to detect and filter out spam messages by matching the content against known spam patterns.

4.1.1 Naïve String Matching Algorithm

Naive Bayes, a frequently used probabilistic algorithm for order issues, is known

Notes

for its simplicity and surprisingly effective performance as a classification rule. This is evident in its application for spam filtering in email programs, which often rely on Naive Bayes algorithms. This explanation will delve into the underlying concepts of Naive Bayes and demonstrate its implementation for spam classification in Python, focusing on binary classification problems for simplicity.

Theory

$X = (X_1, X_2, \dots, X_k)$ *represent k features.Y is the label with K possible values (class)*

From a probabilistic perspective, $X_i \in X$ *and Y are random variables*

The value of X_i *is* x*,Y is* y*.*

Basic Idea

To make orders, we need to utilise X to anticipate Y. At the end of the day, given an information point X=(x1,x2,…,xn), what is the odd of Y being y. This can be revamped as the accompanying condition:

$$P(Y = y|X = (x_1, x_2, \dots x_k))$$

$$Find\ the\ most\ likely\ y \rightarrow prediction = argmax_y P(Y = y|X = (x_1, x_2, \dots x_k))$$

This is the fundamental thought of Naive Bayes; the remainder of the calculation is truly more zeroing in on the best way to compute the contingent likelihood above.

(a) Bayes Theorem

So far Mr. Bayes does not commit the algorithm. Presently is his chance to hit one out of the ballpark. As indicated by the Bayes Theorem:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \rightarrow Posterior = \frac{likelihood \times prior}{evidence}$$

This is a somewhat straightforward change, yet it overcomes any issues between what we need to do and what we can do. We can't get P(Y|X) straightforwardly, yet we can get P(X|Y) and P(Y) from the preparation information. Here's an example:

Outlook	Temperature	Humidity	Windy	Play
sunny	hot	high	false	NO
sunny	hot	high	true	NO
overcast	hot	high	false	YES
rainy	mild	high	false	YES
rainy	cool	normal	false	YES
rainy	cool	normal	true	NO
overcast	cool	normal	true	YES
sunny	mild	high	false	NO
sunny	cool	normal	false	YES
rainy	mild	normal	false	YES
sunny	mild	normal	true	YES
overcast	mild	high	true	YES
overcast	hot	normal	false	YES
rainy	mild	high	true	NO

Notes

For this situation, X =(Outlook, Temperature, Humidity, Windy), and Y=Play. P(X|Y) and P(Y) can be determined:

$$P(Y = Play) = \frac{\#Play}{\#Play + \#\ not\ Play} = \frac{9}{14}$$

$$P(X = (sunny)|Y = Play) = \frac{\#Play\ \&\ Sunny}{\#Play} = \frac{2}{9}$$

Naive Bayes Assumption and Why

Theoretically, it isn't elusive P(X|Y). Notwithstanding, it is a lot harder truly as the quantity of highlights develops.

$X_1 = 1$	$X_2 = 1$	$Y = 1$	$P(X_1 = 1, X_2 = 1 \mid Y = 1) = ?$
		$Y = 0$	$P(X_1 = 1, X_2 = 1 \mid Y = 0) = ?$
	$X_2 = 0$	$Y = 1$	$P(X_1 = 1, X_2 = 0 \mid Y = 1) = ?$
		$Y = 0$	$P(X_1 = 1, X_2 = 0 \mid Y = 0) = ?$
$X_1 = 0$	$X_2 = 1$	$Y = 1$	$P(X_1 = 0, X_2 = 1 \mid Y = 1) = ?$
		$Y = 0$	$P(X_1 = 0, X_2 = 1 \mid Y = 0) = ?$
	$X_2 = 0$	$Y = 1$	$P(X_1 = 0, X_2 = 0 \mid Y = 1) = ?$
		$Y = 0$	$P(X_1 = 0, X_2 = 0 \mid Y = 0) = 1 - others$

7 parameters are needed for a 2-feature binary dataset

$$P(X = (x_1, x_2, \ldots, x_k)|Y = y)\ is\ called\ a\ parameter\ in\ Naive\ Bayes.$$

$$If\ x_j\ \forall j\ is\ binary, there\ are\ 2^{k+1} - 1\ parameters$$

Having this measure of boundaries in the model is unreasonable. To tackle this issue, a guileless supposition is made. We imagine all highlights are free. What's the significance here?

$$if\ X_1\ and\ X_2\ are\ indep. P(X_1, X_2|Y) = P(X_1|Y)P(X_2|Y)$$

Presently with the assistance of this innocent supposition (credulous because highlights are seldom free), we can make grouping with many fewer boundaries:

$$P(Y|X) = \frac{P(Y)\prod_i P(X_i|Y)}{P(X)}$$

$X_1 = 1$	$Y = 1$	$P(X_1 = 1 \mid Y = 1)$
	$Y = 0$	$P(X_1 = 1 \mid Y = 0)$
$X_1 = 0$	$Y = 1$	$P(X_1 = 0 \mid Y = 1) = 1 - P(X_1 = 0 \mid Y = 1)$
	$Y = 0$	$P(X_1 = 0 \mid Y = 0) = 1 - P(X_1 = 1 \mid Y = 0)$

Notes

$X_2 = 1$	$Y = 1$	$P(X_2 = 1\|Y = 1)$
	$Y = 0$	$P(X_2 = 1\|Y = 0)$
$X_2 = 0$	$Y = 1$	$P(X_2 = 0\|Y = 1)$ $= 1 - P(X_2 = 0\|Y = 1)$
	$Y = 0$	$P(X_2 = 0\|Y = 0)$ $= 1 - P(X_2 = 1\|Y = 0)$

Naive Bayes need fewer parameters (4 in this case)

This is serious. We changed the number of boundaries from outstanding to direct. This implies that Naive Bayes handles high-dimensional information well.

Categorical and Continuous Features

Categorical Data

For categorical structures, the approximation of P(Xi|Y) is easy.

$$P(X_i = x|Y = y) = \frac{\#(X_i = x, Y = y)}{\#Y = y}$$

Calculate the likelihood of categorical structures

In any case, one issue is that if some component esteems never show (perhaps the absence of information), their probability will be zero, which makes the entire back likelihood zero. One basic approach to fix this issue is called Laplace Estimator: add nonexistent examples (generally one) to every class

$$P(X_i = x|Y = y) = \frac{\#(X_i = x, Y = y) + 1}{\#Y = y + m}, m = \#\ of\ data\ points\ such\ that\ X_i = X, Y = y$$

Laplace Estimator

Continuous Data

For continuous structures, there are fundamentally two selections: discretization and continuous Naive Bayes.

Discretization works by breaking the information into all-out qualities. The least difficult discretization is uniform binning, which makes receptacles with fixed reach. There are, obviously, more brilliant and more convoluted ways, for example, Recursive negligible entropy dividing or SOM-based parcelling.

	GLOBAL	LOCAL
SUPERVISED	Recursive Minimal Entropy Partitioning	Hierarchical Maximum Entropy
UNSUPERVISED	Equal Width Interval, Equal Frequency Interval	K-means Clustering

Discretizing Continuous Feature for Naive Bayes

Notes

The second choice is utilising known distributions. If the structures are continuous, the Naive Bayes algorithm can be written as:

$$P(Y|X) = \frac{P(Y)\prod_i f(X_i|Y)}{f(X)}$$

f is the chance density function

For example, on the off chance that we imagine the information and see a ringer bend like conveyance, it is reasonable to make a presumption that the element is typically appropriated

The initial step is ascertaining the mean and fluctuation of the component for a given mark y:

$$S: data\ points\ with\ Y = y \rightarrow \mu' = \frac{\sum^S X_i}{len(S)}, \sigma'^2 = \frac{1}{m-1}\sum^S (X_i - \mu')^2$$

variance accustomed by the degree of freedom

Now we can analyse the chance density f(x):

$$f(X_i = x|Y = y) = \frac{1}{\sqrt{2\pi\sigma'^2}} e^{\frac{(x-\mu')^2}{2\sigma'^2}}$$

There are, of course, other distributions:

Although these strategies fluctuate in structure, the centre thought behind them is the equivalent: expecting the component to fulfil a specific dispersion, assessing the boundaries of the circulation, and afterward get the likelihood thickness work.

4.1.2 String Searching Algorithm: Knuth Morris and Pratt

Overview

We next depict a more effective algorithm, distributed by Donald E. Knuth, James H. Morris, and Vaughan R. Pratt, 1977 in: “Quick Pattern Matching in Strings.” In SIAM Diary on Computing, 6(2): 323–350. To outline the thoughts of the calculation, we consider the accompanying model:

T = xyxxyxyxyyxyxyxyyxyxyxxy

And

P = xyxyyxyxyxx

- At an undeniable level, the KMP algorithm is like the naive algorithm: it thinks about movements all together from 1 to n−m, and decides whether the example matches that move. The distinction is that the KMP calculation utilises data gathered from halfway matches of the example also, text to skirt moves that are ensured not to bring about a match.
- Assume that, beginning with the example adjusted under the content at the furthest left end, we over and over “slide” the example to one side and endeavour to coordinate it with the content.
- How about we take a gander at certain instances of how sliding should be possible. The content and example are incorporated in Figure 1, with numbering, to make it simpler to follow.

Notes

```
algorithm Simple-Pattern-Finding(P[1, . . . , m], T[1, . . . , n])
  input:          pattern P of length m and text T of length n
  preconditions:  1 ≤ m ≤ n
  output:         list of all numbers s, such that P occurs with shift s in T

  for s ← 0 to n − m
  {
    if (P[1, . . . , m] == T[s + 1, . . . , s + m]) { output s }
  }
```

1. Think about the circumstance when P[1, . . . , 3] is effectively coordinated with T[1, . . . , 3]. Weat that point discovers a confusion: P[4] 6= T[4]. In light of our insight that P[1, . . . , 3] = T[1, . . . , 3], and disregarding images of the example and text after position 3, what can we find about where a potential match may be? For this situation, the calculation slides the example 2 situations to one side so P[1] is agreed with T[3]. The following correlation is somewhere in the range of P[2] and T[4].
2. Since P[2] 6= T[4], the example slides to the privilege once more, so the following examination is somewhere in the range of P[1] and T[4].
3. At a later point, P[1, . . . , 10] is coordinated with T[6, . . . , 15]. At that point a jumble is found: P[11] 6= T[16]. In view of the way that we know T[6, . . . , 15] = P[1, . . . , 10] (also, disregarding images of the example after position 10 and images of the content after position 15), we can tell that the main conceivable move that may bring about a match is 12. Along these lines, we will slide the example right, and next find out if P[1, . . . , 11] = T[13, . . . , 23]. Along these lines, the following examinations done are P[4] == T[16], P[5] == T[17], P[6] == T[18], etc, insofar as matches are found.

Sliding Rule

We need to make exactly precisely how to actualize the sliding guideline. The accompanying documentation is valuable. Let S = s1s2 . . . sk be a string. Each line of the structure s1 . . . si, 1 ≤ I ≤ k is called a prefix of s. Additionally, we characterise the unfilled string (containing no images) to be a prefix of s. A prefix s 0 of s is an appropriate prefix if s 0 6= s. Likewise, each line of the structure si . . . sk, 1 ≤ I ≤ k is known as a postfix of s. Additionally, the unfilled string (containing no images) is an addition of s. A postfix s 0 of s is an appropriate postfix if s 0 6= s. Assume that P[1, . . . , q] is coordinated with the content T[i−q+1, . . . , i] and a jumble at that point happens: P[q+1] 6= T[i+1]. At that point, slide the example right so the longest conceivable legitimate prefix of P[1, . . . , q] that is additionally a postfix of P[1, . . . , q] is presently lined up with the content, with the last image of this prefix adjusted at T[i]. If π(q) is the number with the end goal that P[1, . . . , π(q)] is the longest legitimate prefix that is additionally a postfix of P[1, . . . , q], at that point the example slides so that

P[1, . . . , π(q)] is lined up with T[i − π(q) + 1, . . . , i].

The KMP algorithm precomputes the qualities π(q) and stores them in a table π[1, . . . , m].

We will clarify later how this is finished. A portion of the qualities π(q) for our model are given in Table 1. Would you be able to sort out the excess qualities?

Notes

In outline, a "progression" of the KMP algorithm gains ground in one of two different ways. Previously the progression, assume that P[1, . . . , q] is now coordinated with T[i − q + 1, . . . , i].

- If P[q + 1] = T[i+ 1], the length of the match is broadened, except if q + 1 = m, in which case we have discovered a total match of the example in the content.
- If P[q + 1] 6= T[i + 1], the example slides to one side.

Regardless, progress is made. The calculation rehashes such strides of progress until the end of the content is reached. Pseudocode for the KMP calculation is given in Algorithm 1.

P:	x	y	x	y	y	x	y	x	y	x	x
q:	1	2	3	4	5	6	7	8	9	10	11
$\pi(q)$:	0	0	1	2	0					3	

Table 1: Table of π values for pattern P.

Running Time

Each time through the circle, possibly we increment I or we slide the example right. Both of these occasions can happen all things considered n times; thus, the recurrent circle has executed all things considered 2n times. The expense of every emphasis of the recurrent circle is O(1). Along these lines, the running time is O(n), expecting that the qualities π(q) are now figured.

Computing the values π(q)

We currently depict how to register the table π[1, . . . , m]. Note that π(1) is consistently equivalent to 0. Assume that we have processed π[1, . . . , i] and we need to process π(i + 1). At first we know P[1, . . . , π(i)] is the longest appropriate prefix of P[1, . . . , i] that is likewise an addition of P[1, . . . , i]. Let q = π(i). On the off chance that P[i + 1] = P[q + 1], it should be that π(i + 1) = q + 1. Else, we set q = π(q) and rehash the test. We proceed until q = 0, at which point we just set π(i + 1) = 0. Some pseudocode is given in Algorithm 2; you should fill in the rest as an activity. Calculation 2 runs in straight an ideal opportunity for a much similar explanation as does the KMP algorithm. Subsequently, the entire KMP calculation runs in time O(n + m), which is superior to the straightforward quadratic time algorithm.

algorithm KMP($P[1,\ldots,m], T[1,\ldots,n]$)
input: pattern P of length m and text T of length n
preconditions: $1 \leq m \leq n$
output: list of all numbers s, such that P occurs with shift s in T

$q \leftarrow 0$;
$i \leftarrow 0$;
while $(i < n)$ /* $P[1,\ldots,q] == T[i-q+1,\ldots,i]$
{
  if $(P[q+1] == T[i+1])$
  {

Notes

```
        q ← q + 1;
        i ← i + 1;
        if (q == m)
        {
          output i − q;
          q ← π(q); /*slide the pattern to the right
        }
      }
      else /* a mismatch occurred
      {
        if (q == 0) { i ← i + 1 }
        else { q ← π(q) }
      }
    }
```

Algorithm 1: KMP algorithm

1. Construct a pattern of length 10, over the alphabet {x, y}, such that the number of iterations of the while loop of Algorithm 2, when i = 10, is as large as possible.
2. Suppose that the pattern P and the text T are strings over an alphabet of size 2. In this case, if you know that P[q + 1] 6= T[i + 1], it is possible to tell by looking only at the pattern (specifically at P[q + 1]), what is the symbol of the text at position i + 1. Such knowledge could be used (in some cases) to increase the amount by which the pattern slides, thus speeding up the algorithm. How might you change the algorithm to take advantage of this? How does this affect the amount of memory needed by the algorithm?

```
algorithm Compute-π-values(P[1, . . . , m])
  input:          pattern P of length m
  preconditions:  1 ≤ m
  output:         table π[1, . . . , m]

  π(1) ← 0;
  for (i ← 1 to m − 1)
  /* π[1, . . . , i] is already calculated; calculate π[i + 1]
  {
```

Algorithm 2: Algorithm to compute the π values. Can you fill in the details?

Appendix: a review of Big-Oh notation Table 2 summarises big-oh notation, which we use to describe the asymptotic running time of algorithms.

We say that $f(n)$ is:	Mean that $f(n)$ grows:	Write:	If:
little-oh of $g(n)$	more slowly than $g(n)$	$f(n) = o(g(n))$	$\lim_{n\to\infty} f(n)/g(n) = 0$
big-oh of $g(n)$	no faster than $g(n)$	$f(n) = O(g(n))$	there exist some c, $n_0 > 0$: for all $n > n_0$, $f(n) <= cg(n)$

Notes

theta of $g(n)$	about as fast as $g(n)$	$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$
approximately equal to $g(n)$	as fast as $g(n)$	$f(n) \approx g(n)$	$\lim_{n\to\infty} f(n)/g(n) = 1$
omega of $g(n)$	no slower than $g(n)$	$f(n) = \Omega(g(n))$	$g(n) = O(f(n))$

Table 2: Summary of big-oh notation.

T:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x	y

P:	1	2	3	4	5	6	7	8	9	10	11
	x	y	x	y	y	x	y	x	y	x	x

Figure 1: Text and pattern used in our examples, with characters numbered.

Notes

4.1.3 String Matching Algorithm: Rabin Karp

The Rabin-Karp algorithm is a string-looking through a calculation that utilizations hashing to discover designs in strings. A string is a theoretical information type that comprises a grouping of characters. Letters, words, sentences, and more can be addressed as strings.

String coordination is a vital use of software engineering. If you've at any point looked through an archive for a specific word, you have profited by string-coordinating innovation. String coordination can likewise be utilised to recognize copyright infringement by contrasting strings in record AA and strings in report BB.

Here are instances of strings in Python:

string = 'abcdefghijklmnopqrstuvwxyz'

sport = 'basketball'

food = 'pizza'

The Rabin-Karp algorithm utilizes hash capacities and the moving hash strategy. Hash work is a capacity that maps one thing to the worth. Specifically, hashing can plan information of subjective size to an estimation of fixed size.

Hashing - Hashing is an approach to relate values utilizing a hash capacity to plan a contribution to a yield. The reason for a hash is to take an enormous piece of information and have the option to be addressed by a more modest structure. Hashing is inconceivably helpful and however has a few destructions, specifically, crashes. The categorized standard reveals to us that we can't place xx balls into x - 1x−1 cans without having at any rate, one basin with two balls in it. With hashing, we can consider the contributions to the capacity of the bails and the pails as the yields of the capacity—a few data sources should unavoidably share a yield worth, and this is known as a crash.

A rolling hash permits a calculation to ascertain an incentive without repeating the whole string. For instance, while looking for a word in content, as the calculation shifts one letter to one side (as in the movement for animal power), rather than computing the hash of the part of text [1:3] as it shifts from text [0:2], the calculation can utilize a moving hash to do a procedure on the hash to get the new hash from the old hash.

Example

Let's assume you have a hash work that maps each letter in the letter sent to a novel indivisible number, [\text{prime}_a \ldots \text{prime}_z][prime a … prime z]. You are looking through a content TT of length nn for a word WW of length mm. Suppose that the content is the accompanying string "abcdefghijklmnopqrstuvwxyz'' and the word is "fgh". How might we utilize a moving hash to look for "fgh"?

At each stage, we compare three letters of the text to the three letters of the word. At the first step, we can multiply primeaXprimeb X primecto get the hash of the string "abc ''.

Similarly, the hash value of "fgh" is primeaXprimeb X primec,Comparison the hash value of "abc" with "fgh" and see that the numbers do not match.

To move onto the next iteration, we need to calculate the hash value of "bcd". How can we do this? We could compute primeb X primecX primed but we would be repetition work we've already done!

Notes

Rabin-Karp uses only simple developments and additions in its continuing hash implementation.

All the tasks are done modulo an indivisible number pp to try not to manage huge numbers. For intelligibility and straightforwardness, the modulo pp has been barred, yet the counts hold when modulo p is available.

The Rabin-Karp Rolling Hash

H=c1ak−1+c2ak−2+c3ak−3+....+cka0

aa is a constant, c_1 \ldots c_kc1…ck is the input characters, and kk is the number of characters there are in the string we are comparing (this is the length of the word).

At each step, there are a constant number of operations (one subtraction, one multiplication, and one addition) and these are done in constant time, O(1)O(1). When checking if a substring of the text is equal to the word, the algorithm just needs to see if the hash values are equal. This can also be done in constant time. Because the algorithm performs O(1)O(1) operations on each character of the text and there are nn characters in the text, this step takes O(n)O(n) time.

4.2 Approximation Algorithms

An approximation algorithm is a technique used to address NP-complete optimization problems. These algorithms aim to find solutions that are close to the optimal solution within a feasible amount of time, typically polynomial time. While they do not guarantee the best solution, they are often practical for solving complex problems where finding the exact optimal solution is computationally infeasible.

Features of Approximation Algorithms

1. Polynomial Time: Approximation algorithms are designed to run in polynomial time, making them practical for large instances of NP-complete problems.
2. Near-Optimal Solutions: They aim to produce solutions that are close to the optimal, often within a specified accuracy.
3. High Accuracy: Many approximation algorithms can guarantee solutions within a certain percentage of the optimal solution, ensuring high quality results.
4. Practicality: These algorithms are particularly useful for real-world applications where an exact solution is not necessary or too costly to compute.

Performance Ratios for Approximation Algorithms

The performance of an approximation algorithm is often measured using a performance ratio, which indicates how close the algorithm's solution is to the optimal solution.

Scenario 1: Optimization Problem with Cost

- Maximisation Problem: For maximisation problems, where the goal is to maximise the cost, let C be the cost of the solution produced by the algorithm and C∗ be the cost of the optimal solution. The performance ratio P(n) is defined such that: max(CC∗,C∗C)≤P(n)
- Minimization Problem: For minimization problems, where the goal is to minimise the cost, the performance ratio P(n) is defined such that: max(CC∗,C∗C)≤P(n)

Notes

Scenario 2: Approximation Ratio

An algorithm is said to achieve an approximation ratio of P(n) if, for any input size n, the cost C of the solution produced by the algorithm is within a factor of P(n) of the cost $C*$ of the optimal solution.

- For a maximisation problem,$0<C\leq C*$, and the approximation ratio is $C*C$, which represents how much smaller the algorithm's solution is compared to the optimal solution.
- For a minimization problem,$0<C*\leq C$, and the approximation ratio is, which represents how much larger the algorithm's solution is compared to the optimal solution.

Examples of Approximation Algorithms

1. Vertex Cover Problem:
   - ❖ Optimization Problem: Find the vertex cover with the fewest vertices.
   - ❖ Approximation Problem: Find a vertex cover with relatively few vertices.
2. Travelling Salesman Problem (TSP):
   - ❖ Optimization Problem: Find the shortest possible cycle that visits each city exactly once.
   - ❖ Approximation Problem: Find a cycle that is relatively short compared to the shortest possible cycle.
3. Set Covering Problem:
   - ❖ Optimization Problem: Model problems requiring resource allocation, such as minimising the number of sets needed to cover all elements.
   - ❖ Approximation Problem: Use a logarithmic approximation ratio to find a near-optimal cover.
4. Subset Sum Problem:
   - ❖ Optimization Problem: Find a subset of {x1,x2,x3,…,xn} whose sum is as large as possible but not larger than a given target value t.
   - ❖ Approximation Problem: Find a subset sum that is close to the optimal sum without exceeding the target.

4.2.1 Vertex Cover Algorithm

There are a few advancement issues like Minimum Spanning Tree (MST), Min-Cut, Maximum Matching, in which you can tackle this precisely and productively in polynomial time. However, numerous down-to-earth huge advancement issues are NP-Hard, in which we are probably not going to discover a calculation that tackles the issue precisely in polynomial time. Instances of the standard NP-Hard issues with a portion of their short portrayal are as following:

- Travelling Salesman Problem (TSP) - finding a base expense visit through all urban communities
- Vertex Cover - discover the least arrangement of the vertex that covers all the edges in the chart (we will portray this in more detail)
- Max Clique
- Set Cover - locate a littlest size cover set that covers each vertex

Notes

- Shortest Superstring - given a bunch of string, locate the littlest subset of strings that contain determined words

These are NP-Hard issues, i.e., If we could tackle any of these issues in polynomial time, at that point P = NP. An illustration of an issue that isn't known to be either NP-Hard: Given 2 diagrams of n vertices, would they say they are the equivalent up to the stage of vertices? This is called Graph Isomorphism. As of now, there is no known polynomial careful calculation for NP-Hard issues. Notwithstanding, it could be conceivable to locate a close ideal arrangement in polynomial time. A calculation that runs in polynomial time and yields an answer near the ideal arrangement is called a guess calculation. We will investigate polynomial-time guess calculations for a few NP-Hard issues.

Definition: Let P be a minimization issue, and I be a case of P. Leave An alone a calculation that finds doable answer for occurrences of P. Let A(I) is the expense of the arrangement returned by A for the occasion I and OP T(I) is the expense of the ideal arrangement (minimum) for me. At that point, An is said to be a α-estimate calculation for P if

$\forall I, A(I)$ OP $T(I) \leq \alpha$

where $\alpha \geq 1$

Notice that since this is a base streamlining issue $A(I) \geq$ OP T(I). Accordingly, 1-estimation calculation creates an ideal arrangement, a guess calculation witH an enormous α may restore an answer that is a lot more terrible than ideal. So the more modest α is, the better nature of the estimation the calculation produces

For instance size n, the most common approximation classes are: $\alpha = O(n^c)$ for $c < 1$, e.g. Clique.

$\alpha = O(\log n)$, e.g. Set Cover.

$\alpha = O(1)$, e.g. Vertex Cover.

$\alpha = 1 + \varepsilon$, $\forall \varepsilon > 0$, this is called Polynomial-time Approximation Scheme (PTAS), e.g. certain scheduling problems. $\alpha = 1 + \varepsilon$ in time that is polynomial in (n, 1ε), this is called Fully Polynomial-time approximation. Scheme (FPTAS), e.g. Knapsack, Subset Sum.

Approximation Algorithm for Vertex Cover

Given a G = (V, E),

find a minimum subset $C \subseteq V$, such that C "covers" all edges in E, i.e., every edge $\in$ E is incident to at least one vertex in C.

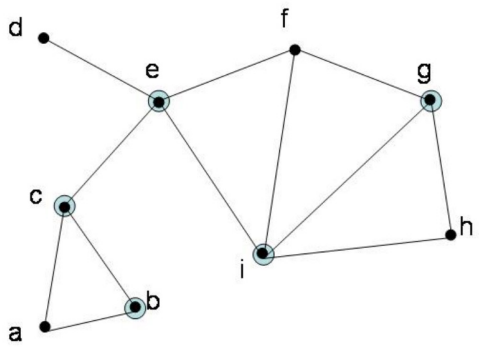


Figure 1: An instance of Vertex Cover problem. An optimal vertex cover is {b, c, e, i, g}.

Notes

Algorithm 1: Approx-Vertex-Cover(G)

1 C←∅

2 while E 6= ∅

pick any {u, v} ∈ E C ← C ∪ {u, v}

delete all edges incident to either u or v

return C

As it turns out, this is the best approximation algorithm known for vertex cover. It is an open problem to either do better or prove that this is a lower bound. Observation: The set of edges picked by this algorithm is matching, no 2 edges touch each other (edges disjoint). It is a maximal matching. We can then have the following alternative description of the algorithm as follows. Find a maximal matching M Return the set of end-points of all edges ∈ M.

Analysis of Approximation Algorithm for VC

Claim 1: This algorithm gives a vertex cover Proof: Every edge ∈ M is covered. If an edge, e /∈ M is not covered, then M ∪ {e} is a matching, which contradicts to maximality of M.

Claim 2: This vertex cover has size ≤ 2×minimum size (optimal solution) Proof:

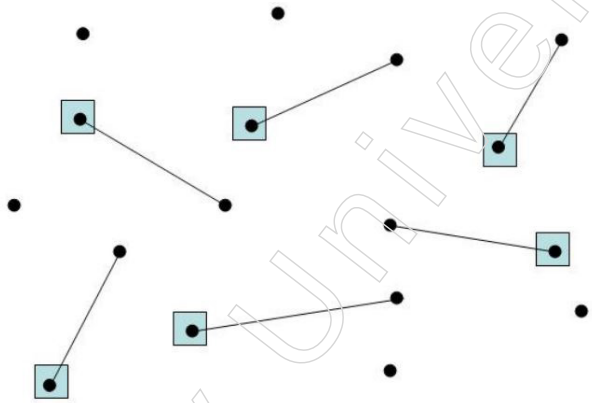


Figure 2: Another instance of Vertex Cover and its optimal cover shown in blue squares

The optimum vertex cover must cover every edge in M. So, it must include at least one of the endpoints of each edge ∈ M, where no 2 edges in M share an endpoint. Hence, optimum vertex cover must have a size

OP T(I) ≥ |M|

But the algorithm A return a vertex cover of size 2|M|, so ∀I we have

A(I) = 2|M| ≤ 2 × OP T(I)

implying that A is a 2-approximation algorithm.

We realise that the ideal arrangement is unmanageable (else we can presumably concoct a calculation to discover it). Along these lines, we can't make an immediate correlation between calculation An's answer and the ideal arrangement. Be that as it may, we can demonstrate Claim 2 by making aberrant examinations of An'sthe answer and the ideal arrangement with the size of the maximal coordinating, |M|. We regularly utilise this strategy for guess verifications for NP-Hard issues, as you will see later on

Notes

But is α = 2 a tight bound for this algorithm? Is it possible that this algorithm can do better than 2-approximation? We can show that 2-approximation is a tight bound by a tight example: Tight Example: Consider a complete bipartite graph of n black nodes on one side and n red nodes on the other side, denoted Kn,n. Notice that the size of any maximal matching of this graph equals n,

|M| = n

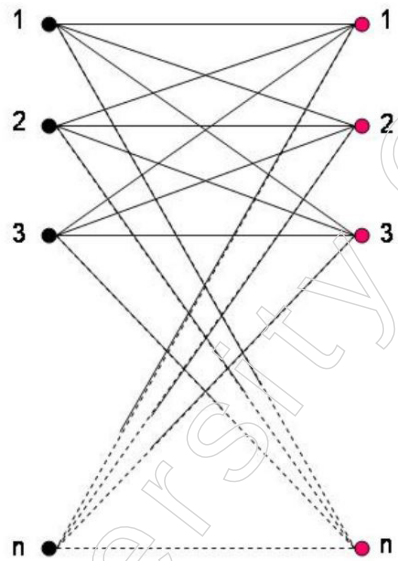


Figure 3: $K_{n,n}$, complete bipartite graph

so the Approx-Vertex-Cover(G) algorithm returns a cover of size 2n.

A(Kn,n) = 2n

But, clearly the optimal solution = n.

OP T(Kn,n) = n

Note that a tight model is required to have the self-assertively enormous size to demonstrate snugness of analysis, otherwise we can simply utilise savage power for little charts and A for huge ones to get a calculation that dodge that tightly bound. Here, it shows that this calculation gives 2-guess regardless of what size n is

Last time: α-approximation algorithms

Definition: For a minimization (or maximisation) problem P, A is an α-approximation algorithm if for every instance I of P, A(I) OPT(I) ≤ α (or OPT(I) A(I) ≤ α).

Last time we saw a 2-approximation for Vertex Cover [CLRS 35.1]. Today we will see a 2- approximation for the Travelling Salesman Problem (TSP) [CLRS 35.2].

Example: A salesman wants to visit each of n cities exactly once each, minimising total distance travelled, and returning to the starting point.

Travelling Salesman Problem (TSP). Input: a complete, undirected graph G = (V, E), with edge weights (costs) w : E −→ R +, and where |V | = n. Output: a tour (a cycle that visits all n vertices exactly once each, and returning to starting vertex) of minimum cost

Inapproximability Result for General TSP

Theorem: For any constant k, it is NP-hard to approximate TSP to a factor of k.

Notes

Proof: Recall that Hamiltonian Cycle (HC) is NP-complete (Sipser). The definition of HC is as follows. Input: an undirected (not necessarily complete) graph G = (V, E). Output: YES if G has a Hamiltonian cycle (or tour, as defined above), NO otherwise. Suppose A is a k-approximation algorithm for TSP. We will use A to solve HC in polynomial time, thus implying P = NP

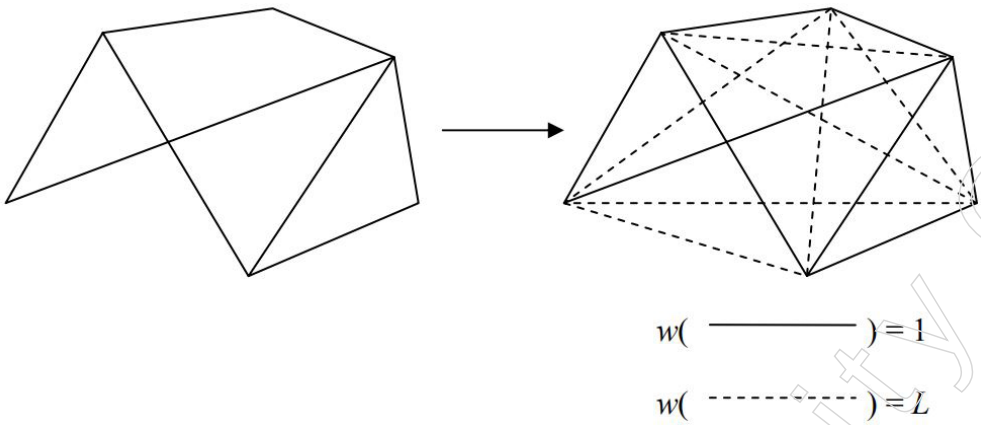


Figure 4: Example of construction of G0 from G for HC-to-TSP-approximation reduction.

Given the input G = (V, E) to HC, we modify it to construct the graph G0 = (V 0, E0) and weight function w as input to A as follows (Figure 4). Let all edges of G have weight 1. Complete the resulting graph, letting all new edges have weight L for some large constant L. The algorithm for HC is then:

Algorithm 2: HC-Reduction(G) Construct G0 1 as described above.

if A(G0 2) returns a 'small' cost tour (≤ kn) then 3 return YES if A(G0 4) returns a 'large' cost tour (≥ L) then 5 return NO It then remains to choose our constant L ≥ kn, to ensure that the 2 cases are differentiated

Approximation Algorithm for Metric TSP

Definition. A metric space is a pair (S, d), where S is a set and d : S 2 −→ R + is a distance function that satisfies, for all u, v, w ∈ S, the following conditions.

1. d(u, v) = 0
2. d(u, v) = d(v, u)
3. d(u, v) + d(v, w) ≥ d(u, w) (triangle inequality)

For a complete graph G = (V, E) with cost c : E −→ R +, we say "the costs form a metric space" if (V, cˆ) is a metric space, where ˆc(u, v) := c({u, v}). Given this restriction (in particular, the addition of the triangle inequality condition), we have the following simple approximation algorithm for TSP.

Algorithm 3: MetricTSPApprox(G)

1 Compute a weighted MST of G.
2 Root MST arbitrarily and traverse in pre-order: $v_1, v_2, \ldots, v_n$.
3 Output tour: $v_1 \to v_2 \to \cdots \to v_n \to v_1$.

Notes

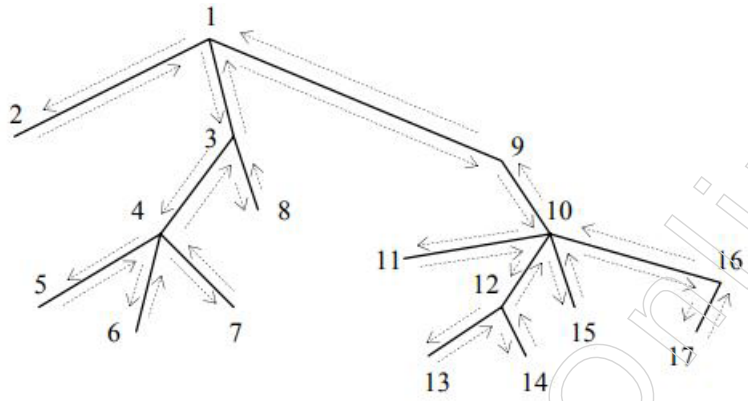


Figure 5: Example MST, where the output tour would be 1 → 2 → · · · → 17 → 1.

Analysis of Approximation Algorithm for Metric TSP On an instance I of TSP, let us compare A(I) to OPT(I), via the intermediate value MST(I) (the weight of the MST). Claim: Comparing A(I) to MST(I): A(I) ≤ 2 × MST(I). Proof: Let σ be a full walk along the MST in pre-order (that is, we revisit vertices as we backtrack through them). In Figure 5, σ would be the path along with all the arrows, wrapping around the entire MST, namely, 1 → 2 → 1 → 3 → 4 → 5 → 4 → 6 → 4 → · · · →

It is clear that cost(σ) = 2 × MST(I). Now, the tout output by A is a subsequence of the full walk σ, so by the triangle inequality: A(I) ≤ cost(σ) = 2 × MST(I) proving our claim. Claim: Comparing OPT(I) to MST(I): OPT(I) ≥ MST(I). Proof: Let σ ∗ be an optimum tour, that is, cost(σ ∗) = OPT(I). Deleting an edge from σ ∗ results in a spanning tree T, whose cost by definition is the cost(T) ≥ MST(I). Hence, OPT(I) = cost(σ ∗) ≥ cost(T) ≥ MST(I)as required.

Combining these 2 claims, we get: A(I) ≤ 2 × MST(I) ≤ 2 × OPT(I) Hence, A is a 2-approximation algorithm for (Metric) TSP.

Concluding Remarks

It is possible (and relatively easy) to improve the approximation factor to 3/2 for Metric TSP. Note that in the original wording of the problem, with the salesman touring cities, the cost (distance) function is even more structured than just a metric. Here, we have Euclidean distance, and as it turns out, this further restriction allows us to get a PTAS, although this is a more difficult algorithm.

4.2.2 Set Covering Problem

What is the set cover problem?

The set cover problem can be thought of as a resource allocation challenge. Imagine you have a collection of items, and each item needs a specific set of resources to function. You're given a bunch of different "kits" that contain various combinations of these resources. Your goal is to find the smallest number of kits you can pick that will provide all the resources needed for every single item. In an ideal scenario, you'd also want to keep the total cost of these kits as low as possible (if the kits have different costs).

Input:

Ground elements, or Universe = { 21 ,...,, uuuU n } Subsets 21 ,...,,

Notes

- k ⊆ USSS Costs k ,...,, ccc 21 Goal:
- Find a set I ⊆ { ,...,2,1 m} that minimises ∑∈Ii i c ,
- such that USIi = ∈U .

(note: in the un-weighted Set Cover Problem, = 1 for all j)

Why is it useful?

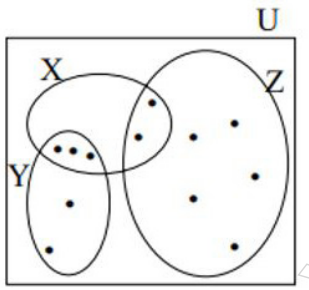
It was one of Karp's NP-complete problems, shown to be so in 1972. Other applications: edge covering, vertex cover Interesting example: IBM finds computer viruses (Wikipedia) elements- 5000 known viruses sets- 9000 substrings of 20 or more consecutive bytes from viruses, not found in 'good' code A set cover of 180 was found. It suffices to search for these 180 substrings to verify the existence of known computer viruses. Another example: Consider General Motors needs to buy a certain amount of varied supplies and some suppliers offer various deals for different combinations of materials (Supplier A: 2 tons of steel + 500 tiles for $x; Supplier B: 1 ton of steel + 2000 tiles for $y; etc.). You could use set covering to find the best way to get all the materials while minimising cost.

How can we solve it?

- Greedy Method – or "brute force" method
- Let C represent the set of elements covered so far
- Let cost-effectiveness, or α, be the average cost per newly covered node.

Algorithm

Example

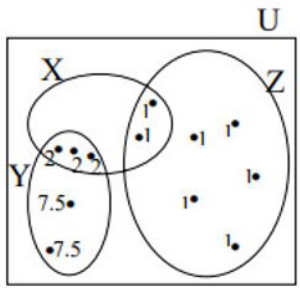


c(X) = 6
c(Y) = 15
c(Z) = 7

Choose Z: $\alpha_Z = \frac{c(Z)}{|S-C|} = \frac{7}{7} = 1$

Choose X: $\alpha_X = \frac{c(X)}{|S-C|} = \frac{6}{3} = 2$

Choose Y: $\alpha_Y = \frac{c(Y)}{|S-C|} = \frac{15}{2} = 7.5$



Total cost = 6 + 15 + 7 = 28

1. C Å 0
2. While C ≠ U do Find the set whose cost-effectiveness is smallest, say S

Let CS Sc − =)(α For each e∈S-C, set price(e) = α C Å C ∪ S 3.

Output picked sets.

Notes

(note: The greedy algorithm is not optimal. An optimal solution would have chosen Y and Z for a cost of 22)

Theorem: The greedy algorithm is a factor approximation algorithm for the minimum set Hncover problem, where

Hn= 1+1/2+…+1/n = log n

Proof:

- We know ∑ = cost of the greedy algorithm = ∈Ue e price)( )(...)()(1 2 ++ + ScScSc m because of the nature in which we distribute costs of elements.
- We will show 1)(+− ≤ n k OPT price k , where the kth element is covered. k e Say the optimal sets are . OOO p ,..., 21 So, OPT)(...)()(. 1 2 p a +++= OcOcOc Now, assume the greedy algorithm has covered the elements in C so far. Then we know the uncovered elements, or −CU, are at most the intersection of all of the optimal sets intersected with the uncovered elements:)(...)()(1 2 p CUOCUOCUOCU b −∩++−∩+−∩≤− In the greedy algorithm, we select a set with cost-effectiveness α, where pi CUO Ocii c ...1,)( )(= −∩ α ≤ . We know this because the greedy algorithm will always choose the set with the smallest cost-effectiveness, which will either be smaller than or equal to a set that the optimal algorithm chooses.

4.2.3 Randomization and Linear Programming

In this segment, we study two strategies that are helpful in planning guess calculations: randomization and direct programming. We will give a straightforward randomised calculation for an improvement variant of 3-CNF satisfiability, and afterward we will utilise direct programming to help plan an estimation calculation for a weighted rendition of the vertex-cover issue. This part just starts to expose these two amazing methods. The part notes give references for additional investigation of these spaces.

Theorem

Given an instance of MAX-3-CNF satisfiability with n variables x_1, x_2,..., xn and m clauses, the randomised algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomised 8/7-approximation algorithm.

Proof Suppose that we have independently set each variable to 1 with probability 1/2 and to 0 with probability 1/2. For i = 1, 2, ..., n, we define the indicator random variable

Yi = I{clause i is satisfied},

so that Yi = 1 as long as at least one of the literals in the ith clause has been set to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so Pr {clause i is not satisfied} = (1/2)3 = 1/8. Thus, Pr{clause i is satisfied} = 1 - 1/8 = 7/8. therefore, E [Yi] = 7/8. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Ym$. Then we have

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} E[Y_i] \quad \text{(by linearity of expectation)} \\ &= \sum_{i=1}^{m} 7/8 \\ &= 7m/8 . \end{aligned}$$

Clearly, m is an upper bound on the number of satisfied clauses, and hence the approximation ratio is at most m/(7m/8) = 8/7.

Approximating weighted vertex cover using linear programming

In the minimum-weight vertex-cover problem, we are given an undirected graph G = (V, E) in which each vertex $v \in V$ has an associated positive weight w(v). For any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \Sigma v \in V' w(v)$. The goal is to find a vertex cover of minimum weight.

We cannot apply the algorithm used for unweighted vertex cover, nor can we use a random solution; both methods may give solutions that are far from optimal. We shall, however, compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. We will then "round" this solution and use it to obtain a vertex cover.

Suppose that we associate a variable x(v) with each vertex $v \in V$, and let us require that $x(v) \in \{0, 1\}$ for each $v \in V$. We interpret x(v) = 1 as v being in the vertex cover, and we interpret x(v) = 0 as v not being in the vertex cover. Then we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as x(u) + x(v) = 1. This view gives rise to the following 0-1 integer program for finding a minimum-weight vertex cover:

minimize

(35.12) $$\sum_{v \in V} w(v)x(v)$$

subject to

(35.13) $x(u) + x(v) \geq 1$ for each $v \in V$

(35.14) $x(v) \in \{0, 1\}$ for each $v \in V$.

By , we know that just finding values of x(v) that satisfy is NP-hard, and so this formulation is not immediately useful. Suppose, however, that we remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$. We then obtain the following linear program, which is known as the linear-programming relaxation:

minimize

(35.15) $$\sum_{v \in V} w(v)x(v)$$

subject to

(35.16) $x(u) + x(v) \geq 1$ for each $v \in V$

(35.17) $x(v) \leq 1$ for each $v \in V$

(35.18) $x(v) \geq 0$ for each $v \in V$.

Notes

Any feasible solution to the 0-1 integer program in lines is also a feasible solution to the linear program in lines (35.15)-(35.18). Therefore, an optimal solution to the linear program is a lower bound on the optimal solution to the 0-1 integer program, and hence a lower bound on an optimal solution to the minimum-weight vertex-cover problem.

The following procedure uses the solution to the above linear program to construct an approximate solution to the minimum-weight vertex-cover problem:

APPROX-MIN-WEIGHT-VC(G, w)

1 C ← Ø

2 compute , an optimal solution to the linear program in lines (35.15)-(35.18)

3 for each v ∈ V

4 do if

5 then C ← C ∪ {v}

6 return C

The APPROX-MIN-WEIGHT-VC procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear program in lines and then solves this linear program. An optimal solution gives each vertex v an associated value $\bar{x}(v)$, where $0 \le \bar{x}(v) \le 1$. We use this value to guide the choice of which vertices to add to the vertex cover C in lines 3-5. If $\bar{x}(v) \ge 1/2$, we add v to C; otherwise we do not. In effect, we are "rounding" each fractional variable in the solution to the linear program to 0 or 1 in order to obtain a solution to the 0-1 integer program in lines. Finally, line 6 returns the vertex cover C.

Theorem

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the for loop of lines 3-5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

Now we show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. Let C* be an optimal solution to the minimum-weight vertex-cover problem, and let z* be the value of an optimal solution to the linear program in lines. Since an optimal vertex cover is a feasible solution to the linear program, z* must be a lower bound on w(C*), that is,

(35.19) $z^* \le w(C^*)$.

Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies w(C) ≤ 2z*. To see that C is a vertex cover, consider any edge (u, v) ∈ E. By constraint ,we know that x(u) + x(v) ≥ 1, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least 1/2. Therefore, at least one of u and v will be included in the vertex cover, and so every edge will be covered.

Now we consider the weight of the cover. We have

Notes

$$
\begin{aligned}
z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) \, .
\end{aligned}
\tag{35.20}
$$

Combining inequalities

$w(C) \leq 2z^* \leq 2w(C^*)$,

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm.

4.2.4 The Subset Sum Problem

Subset total issue is to discover a subset of components that are chosen from a given set whose summation adds to a given number K. We are thinking about whether the set contains non-negative qualities. It is expected to be that the info set is novel (no copies are introduced).

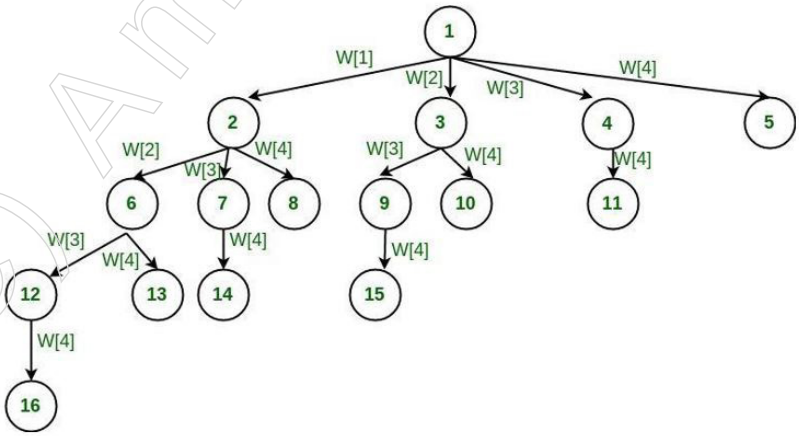
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is 2N.

Backtracking Algorithm for Subset Sum

Using exhaustive search, we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume a given set of 4 elements, say w[1] … w[4]. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts the approach of generating variable sized tuples.



Notes

In the above tree, a hub addresses a capacity call and a branch addresses an applicant component. The root hub contains 4 kids. All in all, the root thinks about each component of the set as an alternate branch. The powerful sub-trees relate to the subsets that incorporate the parent hub. The branches at each level address tuple components to be thought of. For instance, on the off chance that we are at level 1, tuple_vector[1] can take any worth of four branches created. On the off chance that we are at level 2 of the leftmost hub, tuple_vector[2] can take any worth of three branches created, etc…

For instance the left most offspring of root produces each one of those subsets that incorporate w[1]. Also the second offspring of the root produces each one of those subsets that incorporates w[2] and prohibits w[1].

As we go down along the profundity of the tree we add components up until now, and if the additional total is fulfilling express limitations, we will keep on creating youngster hubs further. At whatever point the imperatives are not met, we stop further age of subtrees of that hub, and backtrack to past hubs to investigate the hubs not yet investigated. In numerous situations, it saves extensive measures of handling time.

The tree should trigger a sign to execute the backtracking calculation (attempt yourself). It prints every one of those subsets whose summation adds to a given number. We need to investigate the hubs along the expansiveness and profundity of the tree. Producing hubs along expansiveness is constrained by a circle and hubs along the profundity are created utilising recursion (post request crossing). Pseudo code given underneath,

```
if(subset is satisfying the constraint)
print the subset
exclude the current element and consider next element
else
generate the nodes of present level along breadth of tree and
recur for next levels
```

Following is the execution of subset-total utilising variable size tuple vectors. Note that the accompanying system investigates all prospects like thorough inquiry. It is to show how backtracking can be utilised. See next code to check how we can improve the backtracking arrangement.

```
#include <bits/stdc++.h>
using namespace std;
#define ARRAYSIZE(a) (sizeof(a)) / (sizeof(a[0]))
#define mx 200

static int total_nodes;

// Prints subset found
void printSubset(int A[], int size)
{
for(int i = 0; i < size; i++)
```

Notes

```
{
cout << A[i] << " ";
}
cout << "\n ";
}

// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                    int s_size, int t_size,
                    int sum, int ite,
                    int const target_sum)
{
total_nodes++;

if (target_sum == sum )
{

// We found subset
printSubset(t, t_size);

// Exclude previously added item
// and consider next candidate
subset_sum(s, t, s_size, t_size - 1,
sum - s[ite], ite + 1,
target_sum);
return;
  }
  else
  {

  // Generate nodes along the breadth
```

Notes

```
    for(int i = ite; i < s_size; i++)
    {
    t[t_size] = s[i];

    // Consider next level node (along depth)
    subset_sum(s, t, s_size, t_size + 1,
          sum + s[i], i + 1, target_sum);
          }
    }
}

// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size,
int target_sum)
{
    int *tuplet_vector = new int[mx];

    subset_sum(s, tuplet_vector, size,
    0, 0, 0, target_sum);

    free(tuplet_vector);
}

// Driver Code
int main()
{
    int weights[] = { 10, 7, 5, 18, 12, 20, 15 };
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, 35);
    cout << "Nodes generated " << total_nodes << "\n";
    return 0;
}
```

The force of backtracking seems when we consolidate express and understood requirements, and we quit producing hubs when these checks fizzle. We can improve the above calculation by fortifying the requirement checks and presorting the information. By arranging the underlying cluster, we need not to think about the rest of the exhibit, when the aggregate so far is more noteworthy than the target number. We can backtrack and check different potential outcomes.

Notes

Also, the exhibit is presorted and we discovered one subset. We can create the next hub bearing the current hub just when incorporation of the next hub fulfils the requirements. Given beneath is enhanced execution (it prunes the subtree on the off chance that it isn't fulfilling constraints).

```
#include <bits/stdc++.h>
using namespace std;

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))
static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
   for(int i = 0; i < size; i++)
   {
   cout<<" "<< A[i];
   }
   cout<<"\n";
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
   int *lhs = (int *)pLhs;
   int *rhs = (int *)pRhs;
   return *lhs > *rhs;
}

// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                  int s_size, int t_size,
                  int sum, int ite,
                  int const target_sum)
{
   total_nodes++;
   if( target_sum == sum )
```

Notes

```
    {
    // We found sum
    printSubset(t, t_size);

    // constraint check
    if( ite + 1 < s_size && sum - s[ite] + s[ite + 1] <= target_
sum )
    {
    // Exclude previous added item and consider next candidate
    subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1,
target_sum);
    }
    return;
    }
    else
    {

       // constraint check
       if( ite < s_size && sum + s[ite] <= target_sum )
       {

       // generate nodes along the breadth
       for( int i = ite; i < s_size; i++ )
       {
       t[t_size] = s[i];
       if( sum + s[i] <= target_sum )
       {

    // consider next level node (along depth)
    subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1,
target_sum);
       }
       }
       }
       }
    }

    // Wrapper that prints subsets that sum to target_sum
    void generateSubsets(int s[], int size, int target_sum)
    {
       int *tuplet_vector = (int *)malloc(size * sizeof(int));
       int total = 0;

       // sort the set
```

Notes

```
qsort(s, size, sizeof(int), &comparator);
for( int i = 0; i < size; i++ )
{
total += s[i];
}
if( s[0] <= target_sum && total >= target_sum )
{
subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
}
free(tuplet_vector);
}

// Driver code
int main()
{
int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
int target = 53;
int size = ARRAYSIZE(weights);
generateSubsets(weights, size, target);
cout << "Nodes generated " << total_nodes;
return 0;
}
Output- 8 9 14 22n 8 14 15 16n 15 16 22nNodes generated 68
```

Summary

- String matching algorithms play a crucial role in computer science, enabling efficient searching of patterns within larger texts. These algorithms have a wide range of applications, including database searching, network security, and text processing.
- Naive Bayes, a frequently used probabilistic algorithm for order issues, is known for its simplicity and surprisingly effective performance as a classification rule.
- KMP (Knuth Morris Pratt) Algorithm: The KMP algorithm avoids redundant comparisons by utilising information about previous matches. It pre-processes the pattern to create a partial match table (also known as the "failure function") which helps in bypassing unnecessary comparisons when a mismatch occurs.
- The Rabin-Karp algorithm utilises hash capacities and the moving hash strategy. A hash work is a capacity that maps one thing to a worth. Specifically, hashing can plan information of subjective size to an estimation of fixed size.
- An approximation algorithm is a technique used to address NP-complete optimization problems. These algorithms aim to find solutions that are close to the optimal solution within a feasible amount of time, typically polynomial time.
- Subset total issue is to discover a subset of components that are chosen from a given set whose summation adds to a given number K. We are thinking about whether the set contains non-negative qualities.
- Hashing - Hashing is an approach to relate values utilising a hash capacity to plan a contribution to a yield. The reason for a hash is to take an enormous piece of information and have the option to be addressed by a more modest structure.

Notes

Glossary

- Naive Algorithm: This straightforward algorithm slides the pattern over the text one character at a time and checks for a match. If a match is found, it slides by one character and continues checking for subsequent matches.
- Boyer Moore Algorithm: This algorithm improves efficiency by starting the comparison from the last character of the pattern and using two heuristics (Bad Character and Good Suffix) to skip sections of the text, thus reducing the number of comparisons.
- Trie Data Structure: A trie is an efficient data structure for storing a set of strings. It allows quick retrieval of any string in the set by character-wise comparisons from the root to the leaf nodes, representing individual characters.
- Aho-Corasick Algorithm: This algorithm is used for multiple pattern matching. It constructs a trie for the given set of patterns and then transforms it into a finite automaton with failure links.
- Rabin Karp Algorithm: The Rabin Karp algorithm uses a hash function to compute the hash value of the pattern and compares it with the hash values of substrings of the text.
- Polynomial Time: Approximation algorithms are designed to run in polynomial time, making them practical for large instances of NP-complete problems.
- Near-Optimal Solutions: They aim to produce solutions that are close to the optimal, often within a specified accuracy.
- High Accuracy: Many approximation algorithms can guarantee solutions within a certain percentage of the optimal solution, ensuring high quality results.

Check Your Understanding

1. Which algorithm slides the pattern over the text one character at a time and checks for a match?

   a) KMP Algorithm
   b) Boyer Moore Algorithm
   c) Naive Algorithm
   d) Trie Data Structure

2. What is the purpose of the Rabin Karp algorithm?

   a) Constructing a trie for pattern matching
   b) Finding multiple occurrences of patterns in linear time
   c) Allowing for inexact matches using hash values
   d) Avoiding redundant comparisons by utilising previous matches

3. What is the fundamental idea behind Naive Bayes algorithm?

   a) To find exact matches in a string
   b) To efficiently classify data based on probabilities
   c) To perform string searching using pattern matching
   d) To compute the hash values of substrings for matching

4. What is the randomised approximation algorithm's approximation ratio for MAX-3-CNF satisfiability?

   a) 1
   b) 8/7
   c) 3/2
   d) 2

Notes

5. In the context of the minimum-weight vertex-cover problem, what does linear programming relaxation provide?
   a) An upper bound on the optimal solution
   b) A lower bound on the optimal solution
   c) An exact solution to the problem
   d) A heuristic approach to solve the problem

Exercise

1. What is the main difference between exact string matching and approximate string matching algorithms?
2. Explain the concept of performance ratios in approximation algorithms. How are they measured?
3. How does the Knuth-Morris-Pratt (KMP) algorithm improve upon the Naive string matching algorithm?
4. Discuss the role of approximation algorithms in dealing with NP-complete problems. Provide an example.
5. How can string matching algorithms be applied to DNA sequencing and bioinformatics? How do approximation algorithms improve upon traditional optimization techniques in real-world applications?

Learning Activities

1. Students will implement various string matching algorithms such as Naive, KMP, and Rabin-Karp algorithms. They will analyse the runtime performance of each algorithm on different input sizes and discuss their advantages and limitations.
2. In this activity, students will explore real-world case studies where approximation algorithms have been successfully applied. They will analyse the problem, identify the optimization objectives, and discuss how approximation algorithms provide near-optimal solutions within polynomial time.

Check Your Understanding Answer

1. c) 2. c) 3. b) 4. b)
5. b)

Further Readings and Bibliography

1. Gusfield, D. (1997). Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press.
2. Crochemore, M., & Rytter, W. (1994). Text Algorithms. Oxford University Press.
3. Vazirani, V. V. (2001). Approximation Algorithms. Springer.
4. Williamson, D. P., & Shmoys, D. B. (2011). The Design of Approximation Algorithms. Cambridge University Press.
5. Hochbaum, D. S. (1996). Approximation Algorithms for NP-hard Problems. PWS Publishing Company.
6. Motwani, R., & Raghavan, P. (1995). Randomised Algorithms. Cambridge University Press.

Notes

Module - V: NP-Completeness

Learning Objectives

At the end of this module, you will be able to:

- Understand the concept of polynomial time complexity in the context of algorithm analysis.
- Learn to identify algorithms with polynomial time complexity and differentiate them from algorithms with exponential or factorial time complexity.
- Apply polynomial time complexity analysis to evaluate the performance of algorithms in various problem-solving scenarios.
- Understand the concept of NP-completeness and its significance in computational complexity theory.
- Apply techniques such as reduction proofs and Cook-Levin transformations to identify and solve NP-complete problems.

Introduction

In the realm of computer science, computational problems are central to various tasks and applications. These problems often involve processing input data to achieve a specific goal, which can range from simple calculations to complex optimization tasks.

5.1 Polynomial Time

Defining Polynomial-Time Algorithms:

A polynomial-time algorithm is one where the time taken to solve the problem is upper-bounded by a polynomial function of the size of the input. This means that for every instance of the problem, the algorithm's execution time is limited by a polynomial evaluated at the size of the input.

Measuring Execution Time:

The execution time of an algorithm is typically determined by counting the number of steps it takes to solve the problem, assuming each step takes one unit of time. The size of the input consists of two components: the quantity of data to be processed and the size of each data value.

Encoding Input Data:

The size of an input value depends on how it is encoded. For example, if a value is represented as a binary number, its size is logarithmic in its magnitude. This distinction leads to two common encoding methods: unary representation and binary representation.

Understanding Computational Problem Size:

The size of a computational problem is often denoted as |X| and is determined by the quantity of data and the size of each data value. For instance, in scheduling problems, the size could be the number of jobs (n) and the maximum processing time (pmax).

Notes

Classification of Polynomial-Time Algorithms:

1. Pseudo-Polynomial Time Algorithms: These algorithms run in polynomial time when the data is represented in unary format. However, they are not truly polynomial-time algorithms.
2. Strongly Polynomial-Time Algorithms: Algorithms that run in polynomial time of the number of data points, regardless of the size of the actual data, are termed strongly polynomial-time algorithms.

Example:The Shortest Processing Time (SPT) algorithm, which sorts jobs based on their processing times, runs in O(n log n) time and is considered a strongly polynomial-time algorithm. This assumes that the time taken to compare two numbers is constant, regardless of their size.

Polynomial-Time Reductions

In computational theory, the concept of polynomial-time reductions allows us to relate the complexity of different computational problems. This concept can be applied to a wide range of problems, enabling us to understand their relative difficulty and classify them accordingly.

Definition of Polynomial-Time Reductions:

A computational problem X is said to be polynomial-time reducible to another problem Y if, given an algorithm for problem Y with time complexity T(Y), we can solve an instance of problem X in time (p(|X|) + q(|X|)T(|X|)). This reduction is denoted as X Đ Y or X ≤P Y.

Example:Consider the Hamiltonian Circuit Problem (HCP) and the Travelling Salesman Problem (TSP). It can be shown that HCP is polynomial-time reducible to TSP. By defining distances between vertices in a specific way, we can transform an instance of HCP into an instance of TSP. This reduction allows us to solve HCP instances using algorithms designed for TSP.

Utility of Polynomial-Time Reductions:Polynomial-time reductions are useful for classifying problems into “easy” and “hard” categories. If X is polynomial-time reducible to Y and we have a polynomial-time algorithm for Y, then we can construct a polynomial-time algorithm for X as well. This helps in understanding the computational complexity landscape and identifying problems that are likely to be difficult to solve efficiently.

Complexity Classes:

1. P (Polynomial Time): Problems for which there exists a polynomial-time algorithm to solve them belong to the complexity class P.
2. NP (Nondeterministic Polynomial Time): Decision problems for which “yes” instances can be verified in polynomial time belong to the complexity class NP. Each NP problem has a corresponding optimization version.

Decision vs Optimization Problems:Decision problems involve determining a yes/no answer, while optimization problems seek to find the best solution among possible options. Optimization problems often have corresponding decision versions, where a solution’s optimality can be verified.

Understanding polynomial-time reductions, complexity classes like P and NP, and the distinction between decision and optimization problems is crucial in computational

Notes

theory. These concepts help in analysing the computational complexity of problems and designing efficient algorithms to solve them.

Polynomial Time and NP Classes

The distinction between the classes P and NP is essential to grasp in computational theory. While "P" signifies "polynomial time," "NP" stands for "nondeterministic polynomial time." It's crucial to clarify that NP doesn't mean "not polynomial"; rather, it refers to problems that can be solved in polynomial time if one could "non-deterministically guess" the polynomial-sized certificate for each yes-instance.

Defining the Classes:

1. P (Polynomial Time): A decision problem X is in P if there exists a polynomial-time algorithm A that correctly determines "yes" for every yes-instance and "no" for every no-instance.
2. NP (Nondeterministic Polynomial Time): A decision problem X is in NP if there exists a polynomial-time verifier V such that:
   - For every yes-instance $x \in X$, there exists a polynomial-sized proof y such that V(x, y) = yes.
   - For every no-instance $x \in X$, and for every proof y, we have V(x, y) = no.

Relationship between P and NP:

It's established that P is a subset of NP, meaning every problem solvable in polynomial time is also verifiable in polynomial time. For any problem X in P, a verifier can simply execute the polynomial-time algorithm designed to solve X on the given instance x to verify if it's a yes-instance.

Exploring NP-Complete Problems:

While many problems we encounter fall within the NP class, not all problems are known to be in NP. For instance, consider the PRIMES problem, where the objective is to determine whether a given number is prime or not. While it's possible to construct a certificate for yes-instances, proving primality isn't straightforward and typically requires mathematical techniques beyond simple verification.

Understanding the relationship and characteristics of the P and NP classes is fundamental in analysing the complexity of computational problems and exploring their solvability within polynomial time constraints.

What is the connection between the classes P and NP? Well $P \subseteq NP$.

$P \subseteq NP$.

Proof. Let X alone any choice issue in P (the advancement issues can be taken care of comparably). Leave An alone a calculation to take care of the issue. Leave x alone a yes-case of X. As a verifier, one can just run An on x to check if it is a yes-example. Subsequently, the testament is unfilled and the verifier is only the calculation A.

Are all choice issues in NP? All the issues we have taken a gander at so far are in NP. Yet, that doesn't imply that all issues lie in this class, or possibly are not known to lie in this class.

Notes

Let us turn the COMPOSITE issue on its head to get the PRIMES issue: Given a characteristic number N, choose whether the number N is an indivisible number or not. Could you presently locate a decent testament for yes-occurrences? That is, given an indivisible number N, would you be able to give some additional verification that N is great which can be checked in polynomial time? It turns out that one can, anyway the appropriate response isn't basic and one requirement to utilise variable-based maths to give this endorsement.

5.1.1 Green Technology

The term "green technology" refers to a broad spectrum of inventions and methods intended to enhance environmental sustainability and lessen adverse effects on the environment. When examining green technology within the framework of computational complexity theory, the main goal is to create effective computational solutions and algorithms that support sustainability and environmental preservation. Let's examine the connections between green technology and ideas like NP-completeness and polynomial time:

Algorithms with Polynomial Time for Green Solutions:

- Polynomial time algorithms are quite popular in the field of green technology because they provide effective solutions that can be completed in a fair length of time, which lowers the need for computational resources and energy.
- Polynomial time algorithms, for instance, can be used to schedule renewable energy resources, optimise energy use in smart grids, and design energy-efficient infrastructure or buildings.
- In the end, these algorithms contribute to environmental sustainability by optimising resource allocation, reducing waste, and maximising the use of renewable energy sources.

Decision-Making that is Sustainable and NP-Complete:

- NP-completeness theory addresses difficult-to-solve choice issues effectively. Although the complexity of some environmental and sustainability-related problems may place them in this category, knowledge of their NP-completeness can direct attempts to identify heuristic algorithms or approximation solutions.
- NP-completeness can be seen, for example, in issues like sustainable transportation route optimisation, biodiversity conservation planning, and optimal waste management. Although exact solutions might be unfeasible in terms of processing, heuristics and approximation methods might yield workable solutions in a fair amount of time.
- Researchers and practitioners can effectively handle complex sustainability concerns by developing optimisation models and decision-support tools by utilising computational techniques based in NP-completeness theory.

Utilising Computational Complexity in Green Technology Applications:

- Concepts of computational complexity play a key role in the design and analysis of green technology in a number of fields, such as climate modelling, renewable energy, environmental monitoring, and sustainable resource management.

Notes

- These ideas aid in assessing the scalability and efficiency of algorithms used in green technology applications, guaranteeing that computational resources are employed efficiently while reducing their negative effects on the environment.
- Furthermore, by comprehending the computational complexity of sustainability-related issues, reliable and scalable solutions that can adjust to changing technological environments and environmental changes can be developed.

5.1.2 Hamiltonian Cycles

Definition of Directed Graph:

- A directed graphG is represented as a pairG=(V,E), where$E \subseteq V \times V$.
- The elements ofV are referred to as vertices or nodes.
- An edge$(u,v) \in E$is a connection between vertices u and v.

Simplicity in Directed Graphs:

- We focus on simple graphs, meaning that there are no self-loops, i.e., edges of the form(u,u).
- In a simple graph G=(V,E), whenever$(u,v) \in E$, $u \neq v$

Limited Directed Graphs:

- We consider directed graphs with a finite set of vertices, denoted as V.
- Our attention is on finite graphs, ensuring a finite number of vertices and edges.

Hamiltonian Cycle Problem in Directed Graphs

Definition of Hamiltonian Cycle:

- A Hamiltonian cycle in a directed graph G is a cycle that traverses all vertices exactly once, with the possibility of some edges not being crossed.

Problem Statement:

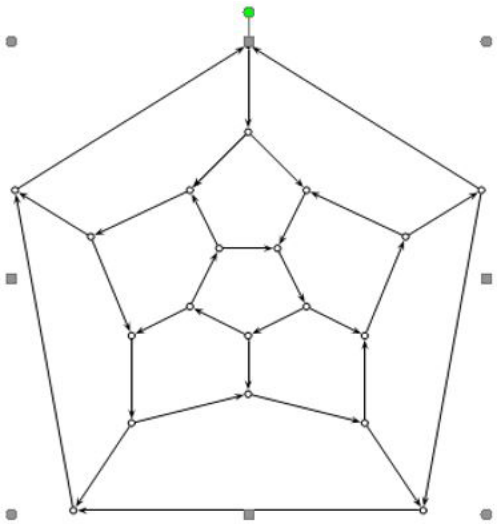
- The Hamiltonian Cycle Problem for Directed Graphs seeks to determine if a given directed graph G contains a Hamiltonian cycle.

Key Concepts

- Vertex: A node or point in the graph.
- Edge: A connection between two vertices.
- Simple Graph: A graph without self-loops.
- Finite Graph: A graph with a finite set of vertices.
- Hamiltonian Cycle: A cycle in the graph that visits every vertex exactly once.
- Hamiltonian Cycle Problem: The problem of determining whether a graph contains a Hamiltonian cycle.

Is there is a Hamiltonian cycle in the coordinated chart D appeared in the Figure

Notes



Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure.

The Complexity of Hamiltonian Cycle in Directed Graphs

Membership in NP:

- The Hamiltonian Cycle problem for directed graphs is in NP because for each "yes" instance (a graph with a Hamiltonian cycle), there exists a polynomial-sized proof (the Hamiltonian cycle itself) that can be verified in polynomial time.

NP-Completeness:

- To prove NP-completeness, we reduce the Exact Cover problem to the Hamiltonian Cycle problem for directed graphs.
- This reduction demonstrates that for every instance of Exact Cover, there exists a polynomial-time transformation to an instance of Hamiltonian Cycle such that the original problem has a solution if and only if the transformed problem has a solution.

Complexity of Reduction:

- The reduction from Exact Cover to Hamiltonian Cycle is considered one of the most challenging reductions in computational complexity theory.
- It requires devising a method that efficiently converts instances of Exact Cover to instances of Hamiltonian Cycle, preserving the existence of solutions between the two problems.

Key Points

- NP Membership: Hamiltonian Cycle in directed graphs is in NP due to the existence of polynomial-sized proofs for "yes" instances.
- NP-Completeness: To establish NP-completeness, we need to reduce the Exact Cover problem to the Hamiltonian Cycle, demonstrating a polynomial-time transformation that preserves the solution's existence.
- Challenging Reduction: The reduction from Exact Cover to Hamiltonian Cycle is considered one of the most difficult reductions in computational complexity theory, requiring careful design to ensure correctness and efficiency.

Notes

Understanding Hamiltonian Cycles in Undirected Graphs

Graph Representation:

- An undirected graph G is represented as a pair G = (V, E), where V is the set of vertices (or nodes) and E is the set of edges, consisting of pairs {u, v} of distinct vertices.

Paths and Cycles:

- Given any two vertices u, v ∈ V, a path from u to v is a sequence of vertices u = u_1, u_2, ..., un = v such that {ui, ui+1} ∈ E for i = 1, 2, ..., n-1. If n = 2, a path from u to v is simply a single edge {u, v}.
- A closed path, or cycle, is a path from some vertex back to itself.

Hamiltonian Cycles:

- In an undirected graph G, a Hamiltonian cycle is a cycle that passes through all the vertices exactly once, where some edges may not be traversed at all.

Proof of NP-Completeness

To prove that the Hamiltonian Cycle problem for undirected graphs is NP-complete, we reduce the Hamiltonian Cycle problem for directed graphs to it. This means providing a polynomial-time method that converts each instance of Hamiltonian Cycle for directed graphs to an instance of Hamiltonian Cycle for undirected graphs, such that the original problem has a solution if and only if the converted problem has a solution.

Reduction Process:

- The reduction involves devising a method that efficiently transforms instances of Hamiltonian Cycle for directed graphs to instances of Hamiltonian Cycle for undirected graphs.
- This reduction demonstrates that the existence of a Hamiltonian cycle in an undirected graph can be determined by examining the corresponding directed graph, preserving the solution between the two problems.

Key Points

- Graph Representation: Undirected graphs are represented by pairs of vertices and edges.
- Paths and Cycles: Paths and cycles are sequences of vertices and edges in a graph.
- Hamiltonian Cycles: A Hamiltonian cycle in an undirected graph traverses all vertices exactly once.
- NP-Completeness Proof: The Hamiltonian Cycle problem for undirected graphs is NP-complete, as demonstrated by a polynomial-time reduction from the Hamiltonian Cycle problem for directed graphs.

5.2 NP-completeness

NP-Complete problems are a special class of problems in computational complexity theory. These problems are notable because they are both in NP (nondeterministic polynomial time) and as hard as any problem in NP. This means:

Notes

1. Verifiability in Polynomial Time: Solutions to NP-Complete problems can be verified in polynomial time.
2. Reduction from Any NP Problem: Any problem in NP can be reduced to an NP-Complete problem in polynomial time.

If a polynomial-time algorithm is found for any NP-Complete problem, it would imply that all NP problems can be solved in polynomial time, effectively proving that P = NP. This is a major unsolved question in computer science.

Formal Definition of NP-Completeness

A language M is NP-Complete if it satisfies the following two conditions:

- M is in NP: This means that given a solution to a problem in M, we can verify the correctness of this solution in polynomial time.
- NP-Hardness: Every language A in NP is polynomial time reducible to M. This implies that M is at least as hard as any other problem in NP.

NP-Hard Problems

An NP-Hard problem is one to which every problem in NP can be reduced in polynomial time. However, an NP-Hard problem itself does not have to be in NP, meaning it might not have a solution verifiable in polynomial time.

Practical and Theoretical Importance of NP-Completeness

NP-Completeness is significant because:

- Theoretical Insight: Understanding NP-Complete problems helps us comprehend the limits of efficient computation.
- Practical Implications: In practice, knowing a problem is NP-Complete guides us to seek approximate or heuristic solutions instead of exact polynomial-time solutions, which are unlikely to exist.

Examples of NP-Complete Problems

- Travelling Salesman Problem (TSP): Given a list of cities and the distances between each pair of cities, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city.
- Knapsack Problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- Vertex Cover Problem: Given a graph, find the smallest set of vertices such that each edge of the graph is incident to at least one vertex in the set.
- Hamiltonian Cycle Problem: Determine whether a given graph contains a Hamiltonian cycle, a cycle that visits each vertex exactly once.
- 3-SAT Problem: Given a Boolean formula in conjunctive normal form with three literals per clause, determine if there is a truth assignment to the variables that makes the formula true.

Notes

5.2.1 NP-Completeness Proofs

Exact Cover

To prove that the Exact Cover is -complete, we reduce the Satisfiability Problem to it:

Satisfiability Problem ≤P Exact Cover

Given a set F = {C1,. .., Cℓ} of ℓ clauses constructed from n propositional variables x1, .., xn, we must construct in polynomial time an instance τ(F) = (U,) of Exact Cover such that F is satisfiable iff τ(F) has a solution.

Example If

F = {C1 = (x1 ∨ x2), C2 = (x1 ∨ x2 ∨ x3), C3 = (x2), C4 = (x2 ∨ x3)},

then the universe U is given by

U = {x1, x2, x3, C1, C2, C3, C4, p11, p12, p21, p22, p23, p31, p41, p42},

and the family F consists of the subsets

{p11}, {p12}, {p21}, {p22}, {p23}, {p31}, {p41}, {p42}

T1,F = {x1, p11}

T1,T = {x1, p21} T2,F = {x2, p22, p31} T2,T = {x2, p12, p41} T3,F = {x3, p23}

T3,T = {x3, p42}

{C1, p11}, {C1, p12}, {C2, p21}, {C2, p22}, {C2, p23},

{C3, p31}, {C4, p41}, {C4, p42}.

It is easy to check that the set consisting of the following subsets is an exact cover:

T1,T = {x1, p21}, T2,T = {x2, p12, p41}, T3,F = {x3, p23},

{C1, p11}, {C2, p22}, {C3, p31}, {C4, p42}.

The general method to construct (U, F) from

F = {C1,. .., Cℓ} proceeds as follows. Say

Cj = (Lj1 ∨ · · · ∨ Ljmj)

is the jth clause in F, where Ljk denotes the kth literal in Cj and mj 1. The universe of τ(F) is the set

U = {xi | 1 ≤ i ≤ n} ∪ {Cj | 1 ≤ j ≤ ℓ}

∪ {pjk | 1 ≤ j ≤ ℓ, 1 ≤ k ≤ mj}

wherein the third set pjk corresponds to the kth literal in Cj.

The following subsets are included in F:

(a) There is a set {pjk} for every pjk.

(b) For every boolean variable xi, the following two sets are in F:

Ti,T = {xi} ∪ {pjk | Ljk = xi}

which contains xi and all negative occurrences of

xi, and

Ti,F = {xi} ∪ {pjk | Ljk = xi}

Notes

which contains xi and all its positive occurrences. Note carefully that Ti,T involves negative occurrences of xi whereas Ti,F involves positive occurrences of xi.

(c) For every clause Cj, the mj sets {Cj, pjk} are in F.

It remains to prove that F is satisfiable iff τ(F) has a solution.

We claim that if v is a truth assignment that satisfies

F, then we can make an exact cover C as follows:

For each xi, we put the subset Ti,T in C iff v(xi) = T, else we we put the subset Ti,F in C iff v(xi) = F.

Also, for every clause Cj, we put some subset {Cj, pjk} in C for a literal Ljk which is made true by v.

By construction of Ti, T and Ti, F, this pjk is not in any set selected so far. Since hypothesis F is satisfiable, such a literal exists for every clause.

Having covered all xi and Cj, we put a set {pjk} in C for every remaining pjk which has not yet been covered by the sets already in C.

Going back to Example 13.2, the truth assignment

v(x1) = T, v(x2) = T, v(x3) = F satisfies

F = {C1 = (x1 ∨ x2), C2 = (x1 ∨ x2 ∨ x3), C3 = (x2), C4 = (x2 ∨ x3)},

so we put

T1,T = {x1, p21}, T2,T = {x2, p12, p41}, T3,F = {x3, p23},

{C1, p11}, {C2, p22}, {C3, p31}, {C4, p42}

in C.

Conversely, if is an exact cover of τ(F), we define a truth assignment as follows:

For every xi, if Ti,T is in C, then we set v(xi) = T, else if Ti,F is in C, then we set v(xi) = F.

Example Given the exact cover

T1,T = {x1, p21}, T2,T = {x2, p12, p41}, T3,F = {x3, p23},

{C1, p11}, {C2, p22}, {C3, p31}, {C4, p42},

we get the satisfying assignment v(x1) = T, v(x2) =

T, v(x3) = F .

If we now consider the proposition is CNF given by

F2 = {C1 = (x1 ∨ x2), C2 = (x1 ∨ x2 ∨ x3), C3 = (x2),

C4 = (x2 ∨ x3 ∨ x4)}

where we have added the boolean variable x4 to clause C4, then U also contains x4 and p43 so we need to add the following subsets to F:

T4,F = {x4, p43}, T4,T = {x4}, {C4, p43}, {p43}.

The truth assignment v(x1) = T, v(x2) = T, v(x3) =

F, v(x4) = T satisfies F2, so an exact cover C is

T1,T = {x1, p21}, T2,T = {x2, p12, p41},

Notes

T3,F = {x3, p23}, T4,T = {x4},

{C1, p11}, {C2, p22}, {C3, p31}, {C4, p42}, {p43}.

Observe that this time, because the truth assignment v makes both literals corresponding to p42 and p43 true and since we picked p42 to form the subset {C4, p42}, we need to add the singleton p43 to cover all elements of U.

5.2.2 NP-Completeness Problems

Lemma for NPC Reductions:

- If a problem Y is in NP and another problem X is polynomial-time reducible to Y, where X is NP-complete, then Y is also NP-complete.
- Proving a problem to be NP-complete involves showing that it is in NP and reducing an existing NP-complete problem to it.

SAT and 3SAT:

- The first problem proven to be in NPC was SAT (Boolean Satisfiability): Given a boolean formula, determine if there is an assignment of true and false values to variables that makes the formula true.
- A simpler variation, 3SAT, deals with formulas in 3-CNF (conjunctive normal form) where each clause consists of the OR of 3 literals.
- 3SAT is NP-complete, demonstrated by reducing SAT to 3SAT, showing that any instance of SAT can be transformed into an instance of 3SAT in polynomial time.

Clique Problem:

- Clique asks whether a graph G has a subset of vertices of a specific size such that every pair of vertices in the subset is connected by an edge.
- The decision version involves determining the existence of a maximal clique.
- While brute force testing each possible subset of vertices to find a maximal clique is feasible for small graphs, it becomes computationally infeasible for larger graphs due to exponential time complexity.

Clique and NPC:

- Clique is in NP because given a subset of vertices, it can be verified in polynomial time whether it forms a clique.
- It's proven that Clique is NP-complete, which means it's as hard as any problem in NP, including SAT.
- This hardness is established by showing that SAT can be polynomial-time reduced to Clique, indicating that solving Clique is at least as difficult as solving SAT.

NPC Reduction:

- The surprising result that 3SAT is polynomial-time reducible to Clique highlights the versatility of NPC reductions, as it shows a connection between logical formulas and graph structures.

The following lemma helps us to prove a problem NP-complete using another NP-complete problem.

Lemma: If $Y \in NP$ and $X \leq_P Y$ for some $X \in NPC$ then $Y \in NPC$

Notes

– To prove Y NPC we just need to prove Y NP (often easy) and reduce the problem in NPC to Y (no lower bound proof needed!).

Finding the first problem in NPC is somewhat difficult and require quite a lot of formalism

- The first problem proven to be in NPC was SAT:
- Give a boolean formula, is there an assignment of true and false to the variables that make the formula true?
- For example:
- Can $((x1 \Rightarrow x2) \vee \neg((\neg x1 \Leftrightarrow x3) \vee x4)) \wedge \neg x2$ be satisfied?

Last time we discussed what seems to be an easier problem 3SAT: Given a formula in 3-CNF, is it satisfactory?

- A formula is in 3-CNF (conjunctive normal form) if it consists of an AND of 'clauses' each of which is the OR of 3 'literals' (a variable or the negation of a variable)
- Example: $(x1 \vee \neg x2 \vee \neg x3) \wedge (\neg x1 \vee x2 \vee x3) \wedge (x1 \vee x2 \vee x3)$

We prove that 3SAT is an NPC, that is, that it is as hard as the general SAT.

- $3SAT \in NP$
- $SAT \leq_P 3SAT$

(we showed how to transform the general formula into 3-CNF in polynomial time.)

5.2.3 NP-Hard and SAT problems.

NP-Hard Problem

NP vs. NP-Hard:

- NP (Nondeterministic Polynomial-time) is not about problems that cannot be solved in polynomial time, but rather about problems solvable in polynomial time through non-deterministic computation.
- NP includes problems where the solution can be checked in polynomial time, but also those where a solution can be guessed and verified in polynomial time.
- NP-hard problems are at least as hard as the hardest problems in NP, but they don't necessarily need to be in NP themselves.

Nondeterministic Computation and Guessing:

- In non-deterministic computation, guess steps are allowed. A nondeterministic algorithm is considered correct if there exists at least one guessed result leading to a correct answer.
- Guessing steps need to be implemented in polynomial time, and legal guesses are drawn from a pool with a size independent of input size.

Model of Computation and Legal Guesses:

- The Turing machine (TM) is the computation model used to study computational complexity.
- In the nondeterministic TM, the transition function allows for multiple possible transitions, each representing a legal guess.

Notes

- Legal guesses must be bounded by a constant size independent of input size, ensuring that nondeterministic polynomial-time computation is feasible.

In non-deterministic computation, there may be some guess steps.

Let us look at an example. Consider the Hamiltonian Cycle problem:

Given a graph G = (V, E), does G contain a Hamiltonian cycle? Here, a Hamiltonian cycle is a cycle passing through each vertex exactly once. The following is a non terminated algorithm for the Hamiltonian Cycle problem

- input a graph G = (V, E).
- step 1 guess a permutation of all vertices.
- step 2 check if guessed permutation gives a Hamiltonian cycle.
- if yes, then accept input

Note that in step 2, if the outcome of checking is no, then we cannot give any conclusion, and hence non deterministic computation stucks. However, a nondeterministic algorithm is considered to solve a decision problem correctly if there exists a guessed result leading to a correct yes-answer. For example, in the above algorithm, if the input graph contains a Hamiltonian cycle, then there exists a guessed permutation which gives a Hamiltonian cycle and hence gives a yes-answer. Therefore, it is a nondeterministic algorithm that solves the

Hamiltonian Cycle problem.

Why (2) is wrong? This is because not only checking step is required to be polynomial-time computable, but also guessing step is required to be polynomial-time computable. How do we estimate time? Let us explain this starting from what is a legal guess. A legal guess is a guess from a pool with a size-independent from input size. For example, in the above algorithm, guessing in step 1 is not legal because the number of permutations of n vertices is n! which depends on input size.

What is the running time of step 1? It is the number of legal guesses spent in the implementation of the guess in step 1. To implement the guess in step 1, we may encode each vertex into a binary code of length log2 n.

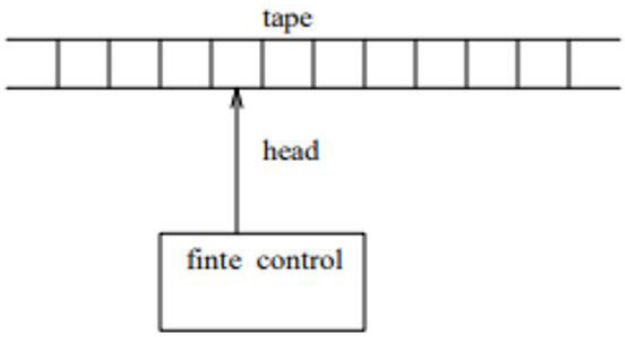
Then each permutation of n vertices is encoded into a binary code of length O(n log n). Now, guessing a permutation can be implemented by O (n log n) legal guesses each of which chooses either 0 or 1. Therefore, the running time of step 1 is O(n log n).

Why is the pool size for a legal guess restricted to be bounded by a constant independent from input size? To answer, we need first to explain the model of computation.

The computation model for the study of computational complexity is the Turing machine (TM) as shown in Fig. 10.1, which consists of three parts, a tape, ahead, and a finite control. Each TM can be described by the following parameters: an alphabet Σ of input symbols, an alphabet Γ of tape symbols, a finite set Q of states, infinite control, a transition function δ, and an initial state. There exist two types of Turing machines, deterministic TM and nondeterministic TM. They differ for the transition function of the finite control. In the deterministic TM, the transition function δ is a mapping from Q × Γ to Q × Γ × {R, L}. A transition δ(q, a) = (p, b, R) (δ(q, a) = (p, b, L)) means that when the machine in state q reads symbol a, it would change

Notes

state to p, the symbol to b and move the head to the right (left) as shown in Fig. 10.2. However, in the nondeterministic TM,



The transition function δ is a mapping from Q × Γ to 2Q×Γ×{R,L}. That is, for any (q, a) ∈ Q×Γ, δ(q, a) is a subset of Q×Γ× {R, L}. In this subset, every member can be used for the rule to guide the transition. When this subset contains at least two members, the transition corresponds to a legal guess. The size of the subset is at most |Q| · |Γ| · 2 which is a constant independent of the input size. In many cases, the guessing step is easily implemented by a polynomial number of legal guesses. However, there are some exceptions; one of them is the following.

Example of NP-Hard Problem: Integer Programming:

- Guessing an n-dimensional integer vector in Integer Programming and checking if it satisfies the constraints is NP, but ensuring guessing can be done in nondeterministic polynomial time is challenging.
- To prove Integer Programming in NP, it's crucial to show that if a solution exists, there's a solution of polynomial size.
- Lemmas are used to demonstrate this, such as bounding the absolute value of elements and showing that solutions can be guessed within a polynomial-sized space.

Integer Programming: Given an m×n integer matrix A and an n-dimensional integer vector b, determine whether there exists a m-dimensional integer vector x such that Ax ≥ b.

To prove Integer Programming in NP, we may guess an

n-dimensional integer vector x and check whether x satisfies Ax ≥ b. However, we need to make sure that guessing can be done in nondeterministic

polynomial-time. That is, we need to show that if the problem has a solution, then there is a solution of polynomial size. Otherwise, our guess cannot find it. This is not an easy job. We include the proof into the following three lemmas.

Let α denote the maximum absolute value of elements in A and b. Denote q = max(m, n).

Lemma 10.1.1 If B is a square submatrix of A, then | det B| ≤ (αq) q

Proof: Let k be the order of B. Then | det B| ≤ k!α k ≤ k kα k ≤ q qα q = (qα) q.

∗Lemma 10.1.2 If rank(A) = r < n, then there exists a nonzero vector z such that Az = 0 and every component of z is at most (αq) q. Proof. Without loss of generality, assume that the left-upper r×r submatrix B is nonsingular. Set xr+1 = · · · = xn−1 = 0 and xn = −1. Apply Cramer's rule to the system of equations

Notes

B(x1, · · · , xr) T = (a1n, · · · , arn) T

where aij is the element of A on the ith row and the jth column. Then we can obtain xi = det Bi/ det B where Bi is a submatrix of A. By Lemma 3.1, | det Bi | ≤ (αq) q. Now, set z1 = det B1, · · · , zr = det Br, zr+1 = · · · = zn−1 = 0, and zn = det B. Then Az = 0.

Lemma 10.1.3 If Ax ≥ b has an integer solution, then it must have an integer solution whose components of absolute value do not exceed 2(αq) 2q+1.

Proof. Let ai denote the ith row of A and bi the ith component of b. Suppose that Ax ≥ b has an integer solution. Then we choose a solution x such that the following set gets the maximum number of elements.

Ax = {ai | bi ≤ aix ≤ bi + (αq) q+1} ∪ {ei | |xi | ≤ (αq) q },

where ei = (0, · · · , 0, 1 | {z } i , 0, · · · , 0). We first prove that the rank of Ax is n

For otherwise, suppose that the rank of Ax is less than n. Then we can find nonzero integer vector z such that for any d ∈ Ax, dz = 0, and each component of z does not exceed (αq) q. Note that ek ∈ Ax implies that kth component zk of z is zero since 0 = ekz = zk. If zk 6= 0, then ek 6∈ Ax, so, |xk| > (αq) q . Set y = x + z or x − z such that |yk| < |xk|. Then for every ei ∈ Ax, yi = xi, so, ei ∈ Ay, and for ai ∈ Ax, aiy = aix, so, ai ∈ Ay. Thus, Ay contains Ax. Moreover, for ai 6∈ Ax, aiy ≥ aix − |aiz| ≥ bi +(αq) q+1 −nα(αq) q ≥ bi. Thus, y is an integer solution of Ax ≥ b. By the maximality of Ax, Ay = Ax. This means that we can decrease the value of the kth component again. However, it cannot be decreased forever. Finally, a contradiction would appear. Thus, Ax must have rank n. Now, choose n linearly independent vectors d1, · · ·, dn from Ax. Denote ci = dix. Then |ci | ≤ α + (αq) q+1. Applying Cramer's rule to the system of equations dix = ci, i = 1, 2, · · ·, n, we obtain a representation of x through ci's: xi = det Di/ det D where D is a square submatrix of (AT, I) T and Di is a square matrix obtained from D by replacing the ith column by the vector (c1, · · ·, cn) T. Note that the determinant of any submatrix of (AT, I) T equals to the determinant of a submatrix of A. By Laplace expansion, it is easy to see that |xi | ≤ | det Di | ≤ (αq) q (|c1| + · · · + |cn|) ≤ (αq) qn(α + (αq) q+1) ≤ 2(αq) 2q+1

By Lemma 10.1.3, it is enough to guess a solution x whose total size is at most n log2 (2(αq) 2q+1)) = O(q 2 (log2 q + log2 α). Note that the input A and b have total length at least β = Pm i=1 Pn j=1 log2 |aij | + Pn j=1 log2 |bj | ≥ mn + log2 α ≥ q+ IP is in NP.

Theorem 10.1.4 Integer Programming is in NP

Proof. It follows immediately from Lemma 10.1.3

The definition of the class NP involves three concepts, non-deterministic computation, polynomial-time, and decision problems. The first two concepts have been explained as above. Next, we explain the decision problem.

A problem is called a decision problem if its answer is "Yes" or "No". For example, the Hamiltonian Cycle problem is a decision problem and all combinatorial optimization problems are not decision problems. However, every combinatorial optimization problem can be transformed into a decision version. For example, consider the Travelling Salesman problem as follows: Given n cities and a distance table between n cities, find the shortest Hamiltonian tour where a Hamiltonian tour is a Hamiltonian cycle in the complete graph on the n cities.

Notes

Its decision version is as follows: Given n cities, a distance table between n cities, and an integer K > 0, is there a Hamiltonian tour with total distance at most K?

If the Hamiltonian Cycle problem can be solved in polynomial-time, so is its decision version. Conversely, if its decision version can be solved in polynomial-time, then we may solve the Hamiltonian Cycle problem in polynomial-time in the following way. Let dmin and dmax be the smallest distance and the maximum distance between two cities. Let a = ndmin and b = ndmax. Set K = [(a + b)/2].

Determine whether there is a tour with total distance at most K by solving the decision version of the Hamiltonian Cycle problem. If the answer is Yes, then set b ← K; else set a ← K. Repeat this process until |b − a| ≤ 1. Then, compute the exact optimal objective function value of the Hamiltonian Cycle problem by solving its decision version twice with K = a and K = b, respectively. In this way, suppose the decision version of the Hamiltonian Cycle problem can be solved in polynomial-time p(n). Then the Hamiltonian Cycle problem can be solved in polynomial-time O(log(ndmax))p(n).

SAT Problem

Remember that a Boolean formula is defined to be satisfiable if there is at least one set of its input variables that makes it true. The language SAT is the set of satisfiable Boolean formulas.

Recall that a boolean formula is in conjunctive normal form (CNF) if it is the AND of zero or more clauses, each of which is the OR of zero or more literals. A formula is in 3-CNF if it is in CNF and has at most three literals in any clause. The language CNF-SAT is the set of CNF formulas that are satisfiable, and the language 3-SAT is the set of 3-CNF formulas that are satisfiable.

Note that 3-SAT is a subset of CNF-SAT, which is a subset of SAT. In general, if A $\subseteq$ B, we can't be certain that A $\leq_p$ B. Although the identity function maps elements of A to elements of B, we can't be sure that it doesn't map a non-element of A to an element of B. But here we know more – we can easily test the formula to see whether it is in CNF or 3-CNF. To reduce 3-SAT to SAT, for example, we map a formula ϕ to itself if it is in 3-CNF, and to 0 if it is not. A similar reduction works to show A $\leq_p$ B whenever A is such an identifiable special case of B – that is, when A = B $\cap$ C and C $\in$ P.

The Cook-Levin Theorem tells us that SAT is NP-complete, essentially by mapping an instance of the generic NP problem to a formula that says that a particular string is a witness for a particular NP-procedure on a particular input. (Recall that if A is an NP language defined so that x $\in$ A iff $\exists$y : (x, y) $\in$ B, we call y variously a witness, proof, or certificate of x's membership in A.)

We'd like to show that CNF-SAT and 3-SAT are also NP-complete. It's clear that they are in NP, but the easy special case reduction does not suffice to show them NP-complete. We can reduce 3-SAT to SAT, but what we need is to reduce the known NP-complete language, SAT, to the language we want to show to be NP-complete, 3- SAT.

On HW#4 I'll have you work through the general reduction from SAT to 3-SAT. Here, I'll present the easier reduction from CNF-SAT to 3-SAT. (The proof of the Cook-Levin Theorem given in CMPSCI 601 shows directly that CNF-SAT is NP-complete.)

Let's now see how to reduce CNF-SAT to 3-SAT. We need a function f that takes a CNF formula ϕ, in CNF, and produces a new formula f(ϕ) such that f(ϕ) is in 3- CNF and the two formulas are either both satisfiable or both unsatisfiable. If we could make ϕ and f(ϕ) equivalent, this would do, but there is no reason to think that an arbitrary CNF

Notes

formula will even have a 3-CNF equivalent form. (Every formula can be translated into CNF, but not necessarily into 3-CNF.)

Instead, we will make f(ϕ) have a different meaning from ϕ, and even a different set of variables. We will add variables to f(ϕ) in such a way that a satisfying set of both old and new variables of f(ϕ) will exist if and only if there is a satisfying set of the old variables alone in ϕ. The old variables will be set the same way in each formula.

Because ϕ is in CNF, we know that it is the AND of clauses, which we may name (`11 ∨ . . . ∨ `1k1), (`21 ∨ . . . ∨ `2k2),. . .(`1m ∨ . . . ∨ `mkm), where the `'s are each literal. For each of these clauses in ϕ, we will make one or more 3-CNF clauses in f(ϕ), possibly including new variables, so that the one clause in ϕ will be satisfied iff all the corresponding clauses in f(ϕ) are satisfied.

So let's consider a single clause `1∨. . .∨`k in ϕ. If k ≤ 3, we can simply copy the clause over to f(ϕ), because it is already suitable for a CNF formula. What if k = 4? We can add one extra variable and make two clauses: (`1 ∨ `2 ∨ x1) and (¬x1 ∨ `3 ∨ `4). It's not too hard to see that both of these clauses are satisfied iff at least one of the `'s is true. If `1 or `2 is true, we can afford to make x1 false, and if `3 or `4 is true, we can make x1 true.

The general construction for k > 4 is similar. We have k − 2 clauses and k − 3 new variables: The clauses are (`1 ∨ `2 ∨ x1), (¬x1 ∨ `3 ∨ x2), (¬x2 ∨ `4 ∨ x3), and so on until we reach (¬xk−4 ∨ `k−2 ∨ xk−3) and finally (¬xk−3 ∨ `k−1 ∨ `k).

If we satisfy the original clause with some `i , this satisfies one of the new clauses, and we can satisfy the others by making all the xi's before it true and all those after it false. Conversely, if we satisfy all the new clauses, we cannot have done it only with xi's because there are more clauses than xi's and each xi only appears at most once as true and at most once as false, and so can satisfy at most one clause.

Since this reduction is easily computable in polynomial time, it shows that CNF-SAT ≤p 3-SAT, and thus (with the quoted result that CNF-SAT is NP-complete) that 3-SAT is NP-complete.

3-SAT is often the most convenient problem to reduce to something else, but other variants of SAT are also sometimes useful. One we'll use later is not-all-equal-SAT or NAE-SAT. Here the input is a formula in 3-CNF, but the formula is "satisfied" only if there is both a true literal and a false literal in each clause.

Let's prove that NAE-SAT is NP-complete. Is it in NP? Yes, if we guess a satisfying assignment it is easy (in linear time) to check the input formula and verify that there is a true literal and a false literal in each clause. So we need to reduce a known NP-complete problem to NAESAT – we'll choose 3-SAT itself. Again we'll transform each old clause into the AND of some new clauses, in this case, three of them.

Given the clause `1 ∨ `2 ∨ `3, we introduce two new variables x and y that appear only in the new clauses for this clause, and a single new variable α that appears several times. The three new clauses are (`1 ∨ `2 ∨ x) ∨ (¬x ∨ `3 ∨ y) ∨ (x ∨ y ∨ α).

We must show that the three new clauses are jointly NAE satisfiable if the original clause is satisfiable in an ordinary way. First, we assume that the old clause is satisfied and show that we can choose values for x, y, and α to NAE-satisfy the new clauses. We make α true (for all the clauses in the formula) and consider the seven possibilities for the values of `1, `2, and `3. In each of the seven cases, we can set x and y, not both true, to NAE-satisfy the three new clauses – we'll check this on the board.

Notes

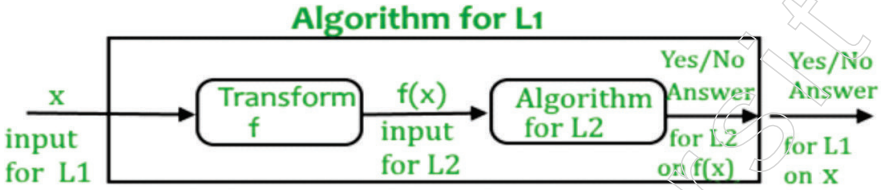
Now assume that the three new clauses are NAE-satisfied – we will show that at least one of the `i's is true. First assume that α is true, because if the new formula is NAE satisfied with α false we can just negate every variable in the formula and get a setting that NAE-satisfies all the clauses but has α true.

If α is true, then either x or y must be false. If x is false, then either `1 or `2 must be true. If y is false and x is true, then `3 must be true. So one of the three ``s must be true, and the original clause is satisfied ordinarily.

5.2.4 NP-completeness and Reducibility

Let L1 and L2 alone be two choice issues. Assume calculation A2 settles L2. That is, if y is a contribution for L2, calculation A2 will answer Yes or No depending on if y has a place with L2.

The thought is to discover a change from L1 to L2 so the calculation A2 can be essential for a calculation A1 to address L1



Learning decreases when all is said and done is vital. For instance, on the off chance that we have library capacities to take care of a certain issue and if we can decrease another issue to one of the tackled issues, we save a great deal of time. Consider the case of an issue where we need to discover the least item way in a given coordinated diagram where the result is the duplication of loads of edges along the way. If we have code for Dijkstra's calculation to discover the most limited way, we can take the log, all things considered, and utilise Dijkstra's calculation to locate the base item way instead of composing a new code for this new issue.

Summary

- A polynomial-time algorithm is one where the time taken to solve the problem is upper-bounded by a polynomial function of the size of the input. This means that for every instance of the problem, the algorithm's execution time is limited by a polynomial evaluated at the size of the input.
- The size of a computational problem is often denoted as |X| and is determined by the quantity of data and the size of each data value.
- Polynomial-time reductions are useful for classifying problems into "easy" and "hard" categories. If X is polynomial-time reducible to Y and we have a polynomial-time algorithm for Y, then we can construct a polynomial-time algorithm for X as well.
- A Hamiltonian cycle in a directed graph G is a cycle that traverses all vertices exactly once, with the possibility of some edges not being crossed.
- NP-Complete problems are a special class of problems in computational complexity theory. These problems are notable because they are both in NP (nondeterministic polynomial time) and as hard as any problem in NP.
- An NP-Hard problem is one to which every problem in NP can be reduced in polynomial time. However, an NP-Hard problem itself does not have to be in NP, meaning it might not have a solution verifiable in polynomial time.

Notes

- Remember that a Boolean formula is defined to be satisfiable if there is at least one set of its input variables that makes it true. The language SAT is the set of satisfiable Boolean formulas.
- If we have code for Dijkstra's calculation to discover the most limited way, we can take the log, all things considered, and utilise Dijkstra's calculation to locate the base item way instead of composing a new code for this new issue.

Glossary

- Pseudo-Polynomial Time Algorithms: These algorithms run in polynomial time when the data is represented in unary format. However, they are not truly polynomial-time algorithms.
- Strongly Polynomial-Time Algorithms: Algorithms that run in polynomial time of the number of data points, regardless of the size of the actual data, are termed strongly polynomial-time algorithms.
- Travelling Salesman Problem (TSP): Given a list of cities and the distances between each pair of cities, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city.
- Knapsack Problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- Vertex Cover Problem: Given a graph, find the smallest set of vertices such that each edge of the graph is incident to at least one vertex in the set.
- Hamiltonian Cycle Problem: Determine whether a given graph contains a Hamiltonian cycle, a cycle that visits each vertex exactly once.
- 3-SAT Problem: Given a Boolean formula in conjunctive normal form with three literals per clause, determine if there is a truth assignment to the variables that makes the formula true.

Check Your Understanding

1. Which of the following statements is true regarding the relationship between the complexity classes P and NP?
   a) NP is a subset of P.
   b) P is a subset of NP.
   c) P and NP are disjoint sets.
   d) There is no relationship between P and NP.
2. Consider the PRIMES problem, where the objective is to determine whether a given number is prime or not. Which of the following statements accurately describes PRIMES in relation to the NP class?
   a) PRIMES is in NP because a polynomial-time verifier can efficiently verify the primality of any given number.
   b) PRIMES is not in NP because it cannot be efficiently verified using a polynomial-time verifier.
   c) PRIMES is in P because it can be solved in polynomial time using efficient algorithms.

Notes

d) PRIMES is in NP because it can be efficiently verified using a polynomial-time algorithm.

3. Which of the following statements accurately describes the Hamiltonian Cycle problem for directed graphs?
   a) The problem seeks to determine if a directed graph contains a cycle that passes through all vertices exactly once.
   b) A Hamiltonian cycle in a directed graph is a cycle that traverses all edges exactly once.
   c) The problem involves finding a cycle that visits each vertex in a directed graph at least once.
   d) The Hamiltonian Cycle problem for directed graphs is not in NP.
4. Which of the following is an example of an NP-Complete problem?
   a) Sorting a list of integers in ascending order.
   b) Calculating the shortest path between two nodes in a graph.
   c) Determining the maximum flow in a network.
   d) Finding the shortest possible route that visits each city exactly once in the Travelling Salesman Problem.
5. What is the primary purpose of reducing one computational problem to another?
   a) To make the original problem more difficult.
   b) To demonstrate the equivalence of the two problems.
   c) To leverage existing algorithms and solutions to solve new problems efficiently.
   d) To simplify the computational complexity of both problems.

Exercise

1. Explain what is meant by polynomial time complexity and why it is desirable in algorithm design.
2. Define the complexity class NP and explain the difference between NP and P.
3. Give examples of algorithms with polynomial time complexity and discuss their practical applications.
4. What is the significance of the Cook-Levin theorem in proving NP-completeness? Provide an overview of its proof technique.
5. How does the concept of NP-completeness impact the development of approximation algorithms? Discuss with an example.

Learning Activities

1. Students will participate in a workshop where they analyse the time complexity of various algorithms. They will categorise algorithms based on their complexity classes, including polynomial time, exponential time, and factorial time. They will also discuss the implications of different time complexities on algorithm selection and optimization.
2. In this activity, students will work in groups to solve NP-complete problems using reduction techniques. They will choose a known NP-complete problem and attempt to reduce it to another NP-complete problem. They will analyse the complexity of the

Notes

reduction and discuss the implications of NP-completeness on the solvability of the problems.

Check Your Understanding Answer

1. b) 2. b) 3. c) 4. d)
5. c)

Further Readings and Bibliography

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
2. Papadimitriou, C. H., & Steiglitz, K. (1998). Combinatorial Optimization: Algorithms and Complexity. Dover Publications.
3. Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.
4. Sipser, M. (2006). Introduction to the Theory of Computation. Cengage Learning.
5. Arora, S., & Barak, B. (2009). Computational Complexity: A Modern Approach. Cambridge University Press.