Name : Abhishek Srivastava.

Reg. No. : 19BCE10071

Subject : Design And Analysis of Algorithm.

Date : 21st January, 2021.

Class No. : 1047

Answers.

(1)

(a)

ar = [93, 81, 50, 60, 46, 5, 55, 87, 27, 90, 70, 64, 26, 67, 23]

Numbe to be searched : 81.

In Linear Search.

We move from index 0 to (size-1) and compare ~~them~~ it with 81.,

Time Complexity = O(N)

In binary search

It is an efficient algorithm. It requires array to be sorted. If the array is sorted, time complexity is O(log N).

~~Lin~~

Tabular Comparism of Linear and Binary Search.

Linear Search	Binary Search
It is iterative in nature and uses sequential approach.	It implements divide and conquer algorithm.
It can be implemented in an array as well as linked list.	It can not be implemented on a linked list.
It does not require array to be sorted.	It requires array to be sorted.
It is easy to use and quite straight forward.	It is a bit tricky, but more efficient.

Searching 81 in given Array.

ar = [93, 81, 50, 60, 46, 5, 55, 87, 27, 90, 70, 64, 26, 67, 23]

sorted ar = [5, 23, 26, 27, 46, 50, 55, 60, 64, 67, 70, 81, 87, 90, 93].

In linear Search

ar

i = 0 ar[i] = 93 ×

i = 1 ar[i] = 81 FOUND.

Found in O(2)

In binary Search.

sorted ar : Middle = 55

7 : 1st iteration = 55 ×

$\frac{8+15}{2}$ = 11 : 2nd iteration = 70 ×

$\frac{9+15}{2}$ = 12 : 3rd iteration = 81 FOUND.

Complexity = SORTING + O(3).

2) (a) NAIVE STRING MATCHING ALGORITHM.

Given a text [0...n-1] and a pattern pat [0...m-1], we need to find occurance of pat in text.

Algorithm.

Slide the pattern over text one by one and check for a match. If a match is found, then slide by 1 again to check subsequent matches.

Example

String = "INTRODUCTION. TO. ALGORITHMS".
KEY = "ALGO".

Output = Pattern found at index 16.

Code

```
#include <bits/stdc++.h >
using namespace std;
 void search (char* pat, char* txt)
   {
     int M = strlen(pat);
     int N= strlen( txt);

     for(int i=0; i<=N-M; i++)
       {
         for(int j=0; j<M; j++)
          {
             if (txt[i+j] != pat[j])
                 break;
          }
       }  if (j==M)
           cout << "Pattern found at index " << i << "\n";
   }
 int main ()
   {
     char txt[]= "INTRODUCTION TO ALGORITHM";
     char pat[]= "ALGO";
     search (pat, txt);
     return 0;
   }
```

Complexity : $O(m*(n-m+1))$.

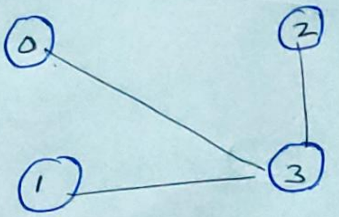
3) Finding Minimal Vertex Cover:

The given problem can be tackled by using Vertex Cover Algorithm.

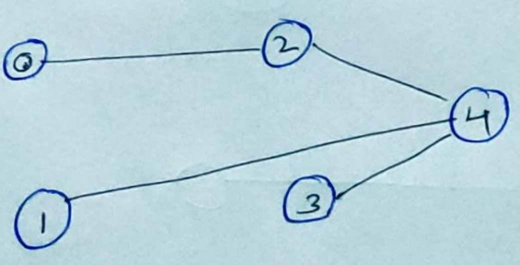
Examples:-

Minimum Vertex Cover is Empty

Minimum Vertex Cover is (3).



Minimum Vertex Cover is {4,2} or {4,0}



Statement

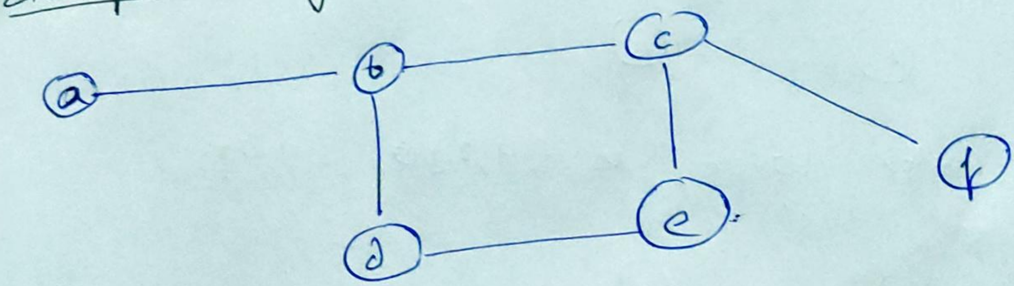
Vertex Cover Problem is known as NP-complete problem, i.e, there is no polynomial time solution for this unless P=NP. There are approximate polynomial time algorithm to solve the problem though.

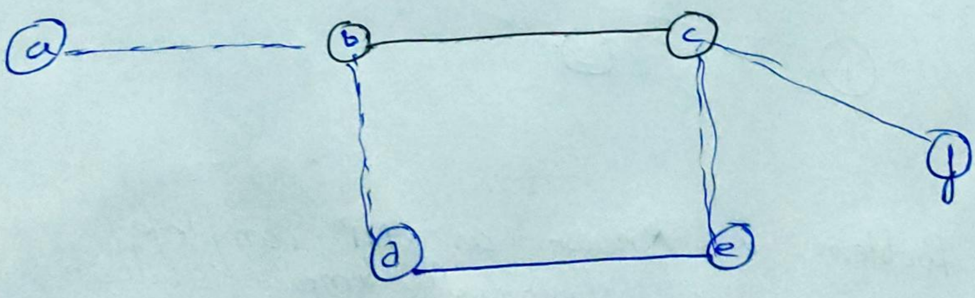
Approximate Algorithm for Vortex Cover :

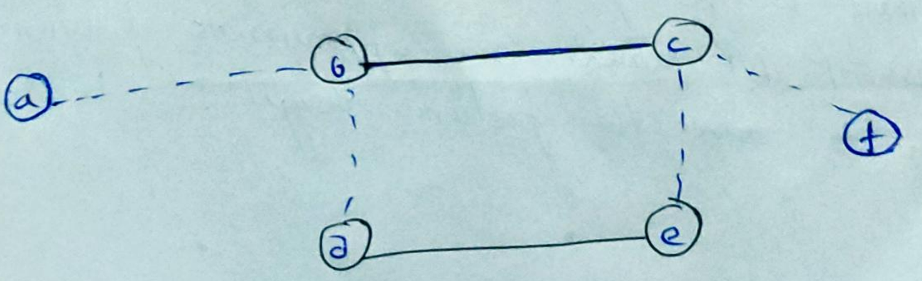
- Initialise the result as { }.
- Consider a set of all edges in given graph. Let the set be E.
- Do following while E is not empty:
  - → Pick an arbitary edge (u,v) from set E and add 'u' and 'v' to result.
  - → Remove all edges from E which are either incident ~~of~~ on 'u' or 'v'.
- Return result.

TIME COMPEXITY : $O(V+E)$

Example through figure.







Minimum Vertex Cover is {b, c, d} or {b, c, e}.

```
# include ~~std~~ < bits/stdc++.h >
using namespace std;

class Graph {
        int V;
        list <int> *adj;
        public:
            Graph (int V);
            ~~int~~ void addEdge (int V, int W);
            void printVertexCover ( );
  };
 Graph: Graph (int V)
        {
            this->V = V;
            adj = new list <int> [V];
        }
void
 Graph : addEdge (int V, int W)
    {
        adj [V].push_back (W);
        adj [W].push_back(V);
    }
void Graph : printVertexCover ( )
        {
            bool visited[V];
            for (int i = 0; i<V; i++)
                    visited [i] = false;

            list <int>::iterator i;
    for (int u=0; u<V; u++)
      {
        if (visited[u]== false)
          {
            for (i= adj[u]. begin(); i!=adj[u].end ();
                                                    i++)
              {
               int v= *i;
               if (visited[v]== false)
                 {
                   visited[v]= true;
                   visited[u]=true;
                   break;
                 }
              }
          }
      }
    for (int i=0; i<V; i++)
       if (visited[i])
          cout << i << ",";

}
int main ()
 {
   Graph g(7);
   g.addEdge (0,1);
   g.addEdge (0,2);
   g.addEdge (1,3);
   g.addEdge (3,4);
   g.addEdge (4,5);
   g.addEdge (5,6);
   g.addEdge (5,6);
                           g.print VertexCover ();
                           return 0;
                   }
```

PART-2

(4)

Polynomial Time Verification.

Many problems are hard to solve, but they have the property that is easy to authenticate the solution if one is provided.

Here we will Examine a problem for which we know of no polynomial time decision algorithm and yet, given a ~~ce~~ ~~cerfi~~ ~~cerf~~ certificate, ~~verf~~ verification is easy.

Hamiltonian Cycle Problem.

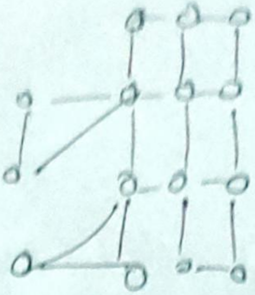
Consider hamiltonian cycle problem.

Given a Graph G, does G have a cycle that visits each vertex exactly once?

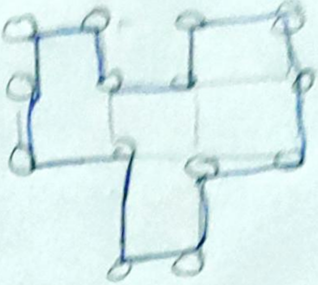
There is no polynomial time algorithm for this dispute.

NOTE :-

It means you can't built a Hamiltonian cycle in a graph with a polynomial time even if there is no specific path is given for the hamiltonian cycle with a particular vertex, yet you can't verify a hamiltonian cycle within a particular time.



Non-Hamiltonean.



Hamiltonian

Explanation.

Let us understand that a Graph did have a hamiltoniar cyclye. It would be easy for someone to convince of this. ~~They~~ They would similarly say; "The period is h v3, v7, v1... v13 1.

Certificate:- It is a piece of information which contains in a given path of a vertex is known as certificate.

We could inspect the Graph and check that this is indeed a legal cycle. ~~and~~ Thus, even though we know of no efficient way to solve the hamiltonian cycle problem, there is a beneficial way to verify that a cycle is indeed hamiltonian.

5

Grahams Scan Algorithm.

~~Usig~~ Using this algorithm, we can find convex hull in O(nlogn) time.

Algorithm.

Let points [0...n-1] be the input array.

1) Find the bottom most point by comparing y cordindtes of all points. It there are two points with the same y-value, then the point with smaller x- co-ordinate will be considered. Let bottom most point be PO. Put PO at first position of output hull.

2) Consider the remaining (n-1) points and sort them by polar angle in counterclockwise order around points [0]. It the polar angle of two points is same, then put the nearest first.

3) After sorting, check if two or more points have the same angle. It two or more points have same angle, then remore all same angle except the farest from PO. Let the size of new array be m.

4) If n is less than 3, return NOT POSSIBLE

5) Create an empty stack 's' and push points[0], points[1] and points [2].

6) Process the remaining n-3 one by one. Do the following for every point 'points[i]'.

    6.1] Keep removing points from stack while orienting 3 points in not counterclockwise.

        a) Point next to top of stack.

        b) Point at top of stack.

        c) points [i].

    6.2) Push points[i] to S.

7) Return contents of S.

Correctness of Graham Scan Algorithm.

It lies in two phase:-

Phase 1: Sort points.
We first find bottom-most points. The idea is to pre-process points ~~that~~ be sorting them with respect to the bottom-most point. Once points are sorted, they form a simple close path.

Phase 2: Accept or Reject Points.
Once we have the close path, the next step is to traverse the path and remove concave points in the path. Again, orientation helps here.