<u>ALGORITHMS ASSIGNMENT</u>

SUBJECT: PRACTICAL ALOGORITHMS FOR PROGRAMMERS

SUBJECT CODE: M.TECH.2018.R.CSN.1.18SN601

Archana Sreekumar
AM.EN.P2CSN18005
M.Tech Cyber Security Systems & Networks
Semester 1 (July-Nov 2018)
Amrita School of Engineering

Table of Contents

Time complexity of sorting algorithms

```
//Instructions to run the c program:
//gcc filename.c -o filename
//./filename
//program runs for lenth n=100,1000,10000
//output file 'time of sorts.txt' will be generated in the same folder
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

//insertion sort function
void insertionSort(int arr[], int size)
{
   int i, key, j;
   for (i = 1; i < size; i++)
   {
       key = arr[i];
       j = i-1;

       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}

//selection sort function

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int size)
{
    int i, j, min;
    for (i = 0; i < size-1; i++){
        min = i;
```

```
        for (j = i+1; j < size; j++)//finding min element
          if (arr[j] < arr[min])
            min = j;

        swap(&arr[min], &arr[i]);
    }
}

//bubble sort function

void swap2(int *xp2, int *yp2)
{
    int temp2 = *xp2;
    *xp2 = *yp2;
    *yp2 = temp2;
}
void bubbleSort(int arr[], int size)
{
   int i, j;
   for (i = 0; i < size-1; i++)
       for (j = 0; j < size-i-1; j++)
           if (arr[j] > arr[j+1])
              swap2(&arr[j], &arr[j+1]);
}

//merge sort functions

void merge(int arr[], int l, int m, int r) //function to merge the sorted arrays
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2]; //initializing 2 temporary arrays

    for (i = 0; i < n1; i++) //copying data to L from main array
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++) //copying data to R from main array
        R[j] = arr[m + 1+ j];

    //merging  L and R to get final sorted array
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
```

```
    while (i < n1 && j < n2) //comparing elements of L&R and storing it in main array
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) //copying remaining elements of L if any
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) //copying remaining elements of R if any
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r) //function for merging
{
    if (l < r)
    {
        int m = l+(r-l)/2; //to find the median;avoids overflow for large l &h

        mergeSort(arr, l, m); //sorting first half of array
        mergeSort(arr, m+1, r); //sorting second half of array

        merge(arr, l, m, r); //to merge the sorted sub-arrays
    }
}
```

//quick sort function

```
void Swap1(int *xp2, int *yp2)
{
   int temp2 = *xp2;
   *xp2 = *yp2;
   *yp2 = temp2;
}

  int Median3( int A[ ], int Left, int Right )
       {
          int Center = ( Left + Right ) / 2;

          if( A[ Left ] > A[ Center ] )
              Swap1( &A[ Left ], &A[ Center ] );
          if( A[ Left ] > A[ Right ] )
              Swap1( &A[ Left ], &A[ Right ] );
          if( A[ Center ] > A[ Right ] )
              Swap1( &A[ Center ], &A[ Right ] );

          // A[ Left ] <= A[ Center ] <= A[ Right ]

          Swap1( &A[ Center ], &A[ Right - 1 ] );  // Hide pivot
          return A[ Right - 1 ];                //Return pivot
       }
 void quicksort( int A[ ], int Left, int Right )
       {
          int i, j;
          int Pivot;

     if( Left + 30 <= Right )
          {
        Pivot = Median3( A, Left, Right );
        i = Left; j = Right - 1;
         for( ; ; )
              {
           while( A[ ++i ] < Pivot ){ }
            while( A[ --j ] > Pivot ){ }
            if( i < j )
                Swap1( &A[ i ], &A[ j ] );
                  else
                break;
              }
```

```
         Swap1( &A[ i ], &A[ Right - 1 ] );  // Restore pivot

        quicksort( A, Left, i - 1 );
        quicksort( A, i + 1, Right );
           }
           else
         insertionSort( A + Left, Right - Left + 1 );// Do an insertion sort on the subarray
        }

//heap sort function

void Swap(int *xp2, int *yp2)
{
    int temp2 = *xp2;
    *xp2 = *yp2;
    *yp2 = temp2;
}

  #define LeftChild( i )  ( 2 * ( i ) + 1 )

 void PercDown( int A[ ], int i, int N )
 {
    int Child;
    int Tmp;

    for( Tmp = A[ i ]; LeftChild( i ) < N; i = Child )
      {
       Child = LeftChild( i );
        if( Child != N - 1 && A[ Child + 1 ] > A[ Child ] )
             Child++;
          if( Tmp < A[ Child ] )
              A[ i ] = A[ Child ];
                else
              break;
      }
    A[ i ] =Tmp;
 }

 void heapsort( int A[ ], int N )
 {
   int i;
   for( i = N / 2; i >= 0; i-- )  // BuildHeap
     PercDown( A, i, N );
```

```
   for( i = N - 1; i > 0; i-- )
      {
       Swap( &A[ 0 ], &A[ i ] );  // DeleteMax
       PercDown( A, 0, i );
       }
     }
//main function

int main()
{ int r=0;
double time_i,time_i1,time_i2,time_s,time_s1,time_s2;
double time_b,time_b1,time_b2,time_q,time_q1,time_q2;
double time_h,time_h1,time_h2,time_m,time_m1,time_m2;
   do {
   printf("Enter lenth of array: ");
   int size;
   scanf("%d",&size);
   int arr[size];
   {for (int p=0;p < size;++p)
             {arr[p] = rand() % 10000000 + 1;}

   }

//time complexity of insertion sort
clock_t t;
t = clock();
insertionSort(arr, size);//calling insertion sort
t = clock() - t;
double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
 if(r==0)
  {time_i = time_taken;}
  else if (r==1)
  {time_i1 = time_taken;}
 else
  {time_i2 = time_taken;}


//time complexity of selection sort
clock_t t1;
t1 = clock();
selectionSort(arr, size);//calling selection sort
t1 = clock() - t1;
double time_taken1 = ((double)t1)/CLOCKS_PER_SEC; // in seconds
```

```
 if(r==0)
  {time_s = time_taken1;}
 else if (r==1)
  {time_s1 = time_taken1;}
 else
  {time_s2 = time_taken1;}


//time complexity of quick sort
 clock_t t4;
 t4 = clock();
 quicksort(arr,0,size-1);//calling quick sort
 t4 = clock() - t4;
 double time_taken4 = ((double)t4)/CLOCKS_PER_SEC; // in seconds

 if(r==0)
   { time_q = time_taken4;}
 else if (r==1)
   { time_q1 = time_taken4;}
 else
    {time_q2 = time_taken4;}

//time complexity of heap sort

 clock_t t5;
 t5 = clock();
 heapsort(arr,size);//calling heap sort
 t5 = clock() - t5;
 double time_taken5 = ((double)t5)/CLOCKS_PER_SEC; // in seconds

 if(r==0)
  {time_h = time_taken5;}
 else if (r==1)
  { time_h1 = time_taken5;}
 else
  {time_h2 = time_taken5;}

//time complexity of merge sort
clock_t t6;
    t6 = clock();
    mergeSort(arr,0,size - 1);//calling merge sort
   t6 = clock() - t6;
   double time_taken6 = ((double)t6)/CLOCKS_PER_SEC; // in seconds
```

```
 if(r==0)
  {time_m = time_taken6;}
 else if (r==1)
  { time_m1 = time_taken6;}
 else
  {time_m2 = time_taken6;}

//time complexity of bubble sort
 clock_t t3;
 t3 = clock();
 bubbleSort(arr, size);//calling bubble sort
 t3 = clock() - t3;
 double time_taken3 = ((double)t3)/CLOCKS_PER_SEC; // in seconds
 if(r==0)
   {time_b = time_taken3;}
 else if (r==1)
   {time_b1 = time_taken3;}
 else
    {time_b2 = time_taken3;

// writing the output to file
  FILE *fp;
  fp = fopen("time of sorts.txt","w");
  if (fp == NULL)
    {
       fprintf(stderr, "\nError opening file\n");
       exit (1);
    }
fprintf(fp,"\t \t \t \t n=100 \t \t \t \t \t   n=1000 \t \t \t \t n=10000 \n");

fprintf(fp,"insrtn \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_i, time_i1,time_i2);
fprintf(fp,"bubble \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_b, time_b1,time_b2);
fprintf(fp,"slctn \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_s, time_s1,time_s2);
fprintf(fp,"merge \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_m, time_m1,time_m2);
fprintf(fp,"quick \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_q, time_q1,time_q2);
fprintf(fp,"heap \t \t \t \t %f \t \t \t \t %f \t \t \t \t %f \n",time_h, time_h1,time_h2);
fclose(fp);

}
r++;
}while (r<= 2);
return 0;
}
```

Sample Output:

	n=100	n=1000	n=10000
insertion	0.000078	0.004938	0.082814
bubble	0.000083	0.008064	0.120585
selection	0.000102	0.008176	0.126339
merge	0.000053	0.000737	0.001013
quick	0.000010	0.000122	0.000235
heap	0.000055	0.000767	0.001261

Graph algorithms

Input file: 'graph.dat'

```
3    2  1     3   1     4  1
2    4  1     5   1
1    6  1
3    3  1    6   1    7  1
2    4  1    7   1
0
1    6   1
2    2   2    4   1
2    4   3    5   10
2    1   4    6   5
4    3   2    5   2     6  8     7  4
1    7   6
0
1    6   1
3    2   2    3   4     4   1
3    1   2    4   3     5   10
3    1   4    4   2     6   5
6    1   1    2   3     3   2       5   7     6  8     7  4
3    2   10   4   7     7   6
3    3   5    4   8     7   1
3    4   4    5   6     6   1
2    2 1     4   1
2    1 1     3   1
3    2 1     4   1     7   1
4    1 1     3   1      5  1    6  1
2    4 1     6   1
2    4 1     5   1
1    3  1
3
```

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

Instructions for reading input file:

Each group of 7 lines (from the beginning of the file) represents one graph (for a total of 6 graphs). Each one of these seven lines provides information to construct the adjacency list for vertex v, v= 1, 2, 3, 4, 5, 6, 7. The first int on a line indicates the number of vertices adjacent to v. The remaining pairs of ints indicate an adjacent vertex (w) and the weight of the edge (v, w).
The exceptions are graph 4 and 5 (used to test for articulation points): each of these two graphs is described by 8 lines. The first 7 lines are as above, the 8^{th} line contains an int indicating which vertex should be the root of the dfs tree. Thus line 29 indicates the root of graph 4's dfs tree should be vertex3 (C) and line 37 indicates the root of graph 5's dfs tree should be vertex 1 (A).

Program:

```
#Directions to run:
#Command: python2 filename.py outputfilename
#Keep the input file 'graph.dat' in the same folder along with filename.py
#output file is also genetated in the same folder after program execution

#Program contains the following:
#parsing the data from input file to the required graph format,eg:{
    # 'B': {'D': 3, 'E': 10},
    # 'A': {'B': 2, 'D': 1},
    # 'D': {'C': 2, 'G': 4, 'E': 2, 'F': 8},
    # 'G': {'F': 1},
    # 'C': {'A ': 4,'F': 5},
    # 'E': {'G': 6},
    # 'F': {}}
#finding topological sort,dijkstra,kruskal,articulation point,strongly connected components

import sys
import re
from collections import defaultdict
from collections import OrderedDict
#parsing the input from input.dat & creating graph in required form
output_file = sys.argv[1]

input_file_name = 'graph.dat'

input_file_object = open(input_file_name, "r")
graph_array = []
input_lines = input_file_object.readlines()#reading the input lines
nodes = ('A', 'B', 'C', 'D', 'E', 'F', 'G')

graph_4_root = 0#initializing variables to store start nodes of articulation point
graph_5_root = 0

# 6 Graphs
currentLineNo = 0
for graph_no in range(6):
  # 7 Vertices
  graph = {}
  for vertex_no in range(7):
    weightDict = {}
    currentLine = re.split('\s+',input_lines[currentLineNo])
    #print currentLine
```

```
    if (currentLine[0] == ''):
      continue
  number_of_weights = int(currentLine[0])
  for join_vertex_no in range(number_of_weights):
      #print join_vertex_no
     join_vertex = int(currentLine[(join_vertex_no * 2) + 1]) - 1#parsing out vertex values
     join_weight = int(currentLine[(join_vertex_no * 2) + 2])#parsing out weight values
     weightDict[nodes[join_vertex]] = join_weight
     graph[nodes[vertex_no]] = weightDict#creating dict inside dict
     currentLineNo += 1
if graph_no == 3:
    # 4th Graph Root of DFS
    graph_4_root = int(input_lines[currentLineNo])
    currentLineNo += 1
if graph_no == 4:
    # 5th Graph Root of DFS
    graph_5_root = int(input_lines[currentLineNo])
    currentLineNo += 1
graph_array.append(graph)#appending all graphs in an array
#storing each graphs
g1=graph_array[0]
g2=graph_array[1]
g3=graph_array[2]
g4=graph_array[3]
g5=graph_array[4]
g6=graph_array[5]
root11=[graph_4_root,graph_5_root]

##############################################################################
#topological sort

top=[]
map1={'1':'A','2':'B','3':'C','4':'D','5':'E','6':'F','7':'G'}
map2={'A':'1','B':'2','C':'3','D':'4','E':'5','F':'6','G':'7'}
ind={'1':0,'2':0,'3':0,'4':0,'5':0,'6':0,'7':0}#initialising the indegrees to zero
l=len(ind)

for i in g1:
        for j in g1[i]:
                ind[map2[j]]+=1#finding the indegrees of each node
queue=[]#initialising an empty queue
for i in range(1,8):
        if ind[str(i)]==0:
                queue.append(map1[str(i)])#adding nodes with indegree 0 to queue
```

```
while len(queue)!=0:#while q not empty
	v=queue.pop(0)#empty q
	top.append(v)#assign topological number

	for i in g1[v]:#for neighbrng nodes
		ind[map2[str(i)]]-=1#decrement the indegree
		if ind[map2[str(i)]]==0:#if indgree becomes 0 add it to q and repeat the steps
			queue.append(i)

f = open(output_file, "w")
f.write('\n')
f.write('The topological sort of the first graph is:\n')
f.write('\n')
f.write('VERTEX'+"  "+'NUMBER\n')
f.write("  "+top[0]+"      "+'1\n')
f.write("  "+top[1]+"      "+'2\n')
f.write("  "+top[3]+"      "+'3\n')
f.write("  "+top[2]+"      "+'4\n')
f.write("  "+top[4]+"      "+'5\n')
f.write("  "+top[5]+"      "+'6\n')
f.write("  "+top[6]+"      "+'7\n')
f.write('************************************************\n')
################################################################################

#dijikstras algorithm

visited = {node: 0 for node in nodes}  #initially all nodes unvisited

vertex='A'
visited[vertex]=1
t_distance={node: None for node in nodes}  #none==inf
t_distance[vertex]=0#making dist of A as 0
final={}

dist1=defaultdict()
dist2=defaultdict()

def dij(vertex,dist1,dist2):
	visited[vertex]=1#making the called node as known/visited

	for n,w in g2[vertex].items():  #for neighbors+weight of the called node

		if visited[n]==0:#if neighbr not known
			if t_distance[n]==None:  #if distance is inf
```

```
                                t_distance[n]=t_distance[vertex]+w  #update distance
                                dist2[n]=t_distance[n]   #to keep track of shortest path node
                        elif t_distance[n]>(t_distance[vertex]+w):
                                t_distance[n]=t_distance[vertex]+w
                                dist2[n]=t_distance[n]

        dist1=sorted(dist2.items(), key=lambda x: x[1])#sorting the pairs(node,weight) based
on weight value

        vertex=dist1[0]
        vertex=vertex[0]#updating 'vertex' with next node with shortest distance(weight)
        del dist2[vertex]#delete the vertex which became known from the tracking list
        if len(dist2)!=0:
                dij(vertex,dist1,dist2)
        else:
                flag=0

dij(vertex,dist1,dist2)
f.write('Shortest path for the second graph is:\n')
f.write('\n')
f.write('Shortest path from A to A:    A (distance = '+str(t_distance['A'])+')\n')
f.write('Shortest path from A to B:    A (distance = '+str(t_distance['B'])+')\n')
f.write('Shortest path from A to C:    A (distance = '+str(t_distance['C'])+')\n')
f.write('Shortest path from A to D:    A (distance = '+str(t_distance['D'])+')\n')
f.write('Shortest path from A to E:    A (distance = '+str(t_distance['E'])+')\n')
f.write('Shortest path from A to F:    A (distance = '+str(t_distance['F'])+')\n')
f.write('Shortest path from A to G:    A (distance = '+str(t_distance['G'])+')\n')
f.write('*************************************************\n')
################################################################################

#Kruskal algorithm
def find(v):#function to find set of an element
        if parent[v]==v:
                return v
        return find(parent[v])

def union(x,y):#function for doing union of two sets x and y
        root1=find(x)#find roots of x & y
        root2=find(y)
        #attach smaller rank node below higher rank node
        if root1<root2:
                parent[root1]=root2
        elif root1>root2:
                parent[root2]=root1
```

```
	#if ranks are same then take one as root and increment its rank
	else:
		parent[root1]=root2
		rank[root2]+=1

def kruskal(G):
	global edges_accptd
	minimum_spanning_tree = set()
	while edges_accptd<(len(vertices)-1):
		for edge in G:
			w,v1,v2 = edge#for each (weight,node1,node2) in adj list copy that value to (w,v1,v2)

			x=find(v1)
			y=find(v2)
			if x!=y:
				edges_accptd+=1
				minimum_spanning_tree.add(edge)
				union(x,y)
	return sorted(minimum_spanning_tree)


vertices=['A','B','C','D','E','F','G']
parent={key:key for key in vertices}
rank={key:0 for key in vertices}
msp=set()
adj1=set()
adj=[]

for key in g3:
	for key1 in g3[key]:
		adj1=((g3[key][key1]),key,key1)#(weight,node1,node2)
		adj.append(adj1)#list of (weight,node1,node2)

adj.sort()#created adjacency list from the given graph
			#with (weight,node1,node2)&sorted on ascending order of weight
edges_accptd=0
sum1=0
msp=kruskal(adj)#calling function
#print msp
l=len(msp)
f.write('The edges in the minimum spanning tree for the third graph are:\n')
f.write('\n')
```

```
for i in range(0,l):
	w,v1,v2=msp[i]
	sum1+=w#finiding the total cost
	f.write("("+str(v1)+","+str(v2)+")\t")
f.write("\n"+"Its cost is "+str(sum1)+'\n')
f.write('**************************************************\n')
##############################################################################

#articulation point

def min(a,b):
	if a>b:
		return b
	else:
		return a
def dfs(v):
	global counter
	dfsnum[v] = counter#assigning dfs number
	visited1[v]=1#making the node as visited
	low[v]=dfsnum[v]#assigning low of the node
	counter=counter+1
	for w in adj[v]:#for each neighbour adjacent to v
		if visited1[w] !=1:#checking if visited

			parent[w]=v#assigning parent
			dfs(w)#calling dfs
			if (low[w]>=dfsnum[v]): #checking condition for articulation point

				art.append(v)#adding it to a list

			low[v]=min(low[v],low[w])#updating low for tree edge
		elif parent[v]!=w:

			low[v]=min(low[v],dfsnum[w])#updating low for back edge

art=[]
adj=dict()
keys=[]
counter=1
visited1=dict()
dfsnum=dict()
low=dict()
parent=dict()
nodes1 = {'A':1, 'B':2, 'C':3, 'D':4, 'E':5, 'F':6, 'G':7}
```

```
map11={'1':'A','2':'B','3':'C','4':'D','5':'E','6':'F','7':'G'}
x={str(root11[0]):'fourth',str(root11[1]):'fifth'}
#initializing all dictionaries with vertex as key and value of each vertex as 0
adj={key:[] for key in nodes1}
visited1={key:0 for key in nodes1}
dfsnum={key:0 for key in nodes1}
parent={key:0 for key in nodes1}
low={key:0 for key in nodes}
for key in g4:
	for key2 in g4[key]:
		adj[key].append(map11[str(nodes1[key2])])#creating adjacency list from the graph

for key in root11:
	dfs(map11[str(key)])#calling dfs ,one with start node 3 and one with start node 1
	f.write('For the '+str(x[str(key)])+' graph, the articulation points are:\n')
	#f.write('\n')
	f.write(str(art[0])+'\n')
	if map11[str(key)]=='C':
		f.write(str(art[1])+" (root of the dfs tree)\n")
	else:
		f.write(str(art[1])+'\n')
f.write('*************************************************\n')
################################################################################

#strongly connected

def large(num):
	s1=0
	for key in keys:
		if num[key]>s1:
			s=key
			s1=num[key]#finding the larger value to find the start node of second dfs
	return s
def dfsrev(v):
	global list1
	visited3[v]=1
	keys.remove(v)
	list1.append(v)
	for w in g7[v]:
		if visited3[w]!=1:
			dfsrev(w)
```

```
def dfs(v):
        global count
        visited3[v]=1
        keys.remove(v)
      for w in g6[v]:
                if visited3[w]!=1:
                        dfs(w)
                        count=count+1
                        num[w]=count#post order dfs numbering
        num[v]=count+1#list of nodes and there post order numbers

keys=[]
list1=[]
visited3=dict()
num=dict()
count=0
keys=['A','B','C','D','E','F','G']
g7={key:[] for key in keys}####initializing
visited3={key:0 for key in keys}
num={key:0 for key in keys}#post order number list
for key in g6:
        for key1 in g6[key]:
                g7[key1].append(key)####reversed graph

dfs('A')#calling dfs with a start node
if len(keys)!=0:
      count=count+1
      dfs(keys[0])#calling dfs form another node which is not yet visited
keys=['A','B','C','D','E','F','G']
visited3={key:0 for key in keys}
s=large(num)#finding a starting node with hisghest post order number
f.write('The strongly connected components of the sixth graph are:\n')
f.write('\n')
out={'1':[],'2':[]}
while len(keys)!=0:
        s=large(num)
      dfsrev(s)#calling dfs with the reversed graph
      #print list1
      if len(list1)==3:
        f.write('{'+str(list1[2])+' '+str(list1[0])+' '+str(list1[1])+'}\n')
      else:
        f.write('{'+str(list1[2])+' '+str(list1[0])+' '+str(list1[1])+' '+str(list1[3])+'}\n')
      list1=[]
       f.write('*****************************************************\n')
```

Sample Output:

The topological sort of the first graph is:
VERTEX NUMBER
A 1
B 2
D 3
E 4
C 5
G 6
F 7
****
Shortest path for the second graph is:

Shortest path from A to A: A (distance = 0)
Shortest path from A to B: A (distance = 2)
Shortest path from A to C: A (distance = 3)
Shortest path from A to D: A (distance = 1)
Shortest path from A to E: A (distance = 3)
Shortest path from A to F: A (distance = 6)
Shortest path from A to G: A (distance = 5)
****
The edges in the minimum spanning tree for the third graph are:

(A,D) (F,G)(A,B) (C,D) (D,G) (E,G)
Its cost is 16
****
For the fourth graph, the articulation points are:
D
C (root of the dfs tree)
For the fifth graph, the articulation points are:
D
C
****
The strongly connected components of the sixth graph are:
{G D E}
{B A C F}
****

Divide and conquer algorithm

Long Integer Multiplication

```
#command to run
#python2 filename.py
import math
print 'enter the first number'
a=input()
print 'enter the second number'
b=input()
a1=a#copying the value of a
b1=b#coying the value of b
n=0
count=0
n1=0
while(a1>0):#loop to find length of a
   x=a1%10
   count+=1
   a1=a1/10
n1=count#length of a
count=0
n2=0
while(b1>0):#loop to find length of b
   x=b1%10
   count+=1
   b1=b1/10
n2=count#length of b
if(n1>n2):#taking the largest length as n
   n=n1
else:
   n=n2
num=0
aR=0
aL=0
```

```
bR=0
bL=0
x1=0
x2=0
x3=0
#finding with which number we need to divide a to get aR
num=pow(10,((n1+1)/2))
aR=a%num
aL=a-aR
aL=a/num
num=0
#finding with which number we need to divide a to get bR
num=pow(10,((n2+1)/2))
bR=b%num
bL=b-bR
bL=b/num
num=0
x1=aL*bL
x2=aR*bR
x3=(aL+aR)*(bL+bR)
#finding the product using the formula
num=x1*pow(10,n)+(x3-x1-x2)*pow(10,(n/2))+x2;
print"Product of "+str(a)+" and "+str(b)+" is "+str(num)
```

****

Sample input:

enter first number:1234

enter second number:5678

Sample output:

Product of 1234 and 5678 is 7006652

Maximum Subarray Sum

```
//command to run
//gcc filename.c -o filename
// ./filename

#include<stdio.h>
#include <limits.h>

//function to find the maximum of two elements
int max1(int p,int q)
{
	return (p>q)? p:q;
}
//function to find the maximum of three elements
int max(int x,int y,int z)
{
	return max1(max1(x,y),z);
}

//function to find the sum of overlapping suarrays
int max_cross_sum(int a[],int l,int m,int h)
{

	int l_max=INT_MIN;//min value is set
	int sum=0;
	//finding the sum of elements in the range mid to low
	//and finding left max value
	for (int i=m;i>=l;i--)
	{
		sum+=a[i];
		if(sum>l_max)
			l_max=sum;
	}

	int r_max=INT_MIN;//min value is set
	sum=0;
	//finding the sum of elements in the range mid+1 to high
	//and finding right max value
	for(int j=m+1;j<=h;j++)
	{
		sum+=a[j];
		if(sum>r_max)
			r_max=sum;
	}

	return l_max+r_max;//returning the sum of both
}
```

```
int max_sub_sum(int a[],int l,int h)
{
	if(h==l)
		return a[l];//if high=low then return that element
	int m=(l+h)/2;//else find the mid

	//finding the maximum among the three
	return max(max_sub_sum(a,l,m),max_sub_sum(a,m+1,h),max_cross_sum(a,l,m,h));
}



int main(int argc, char const *argv[])
{
	int arr[100];
	int n;
	printf("enter size of array\n");
	scanf("%d",&n);
	printf("enter the array elements\n");
	for (int i=0;i<n;i++)
	{
		scanf("%d",&arr[i]);

	}

	int max=max_sub_sum(arr,0,n-1);
	printf("Maximum subarray sum is %d\n",max );
	return 0;
}
```
****

Sample input:

enter size of array: 5

enter the array elements: -1 2 3 -4 6

Sample output:

Maximum subarray sum is 7

Greedy algorithm

Huffman coding:

```
#command to run the program
#python2 filename.py

from heapq import heappush, heappop, heapify
from collections import defaultdict

def encode(symb2freq):
    """Huffman encode the given dict mapping symbols to weights/frequency"""
    heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
    #print heap
    heapify(heap)  #maintains pripriity qeue.heap[0] always the smallest element(according to
                                                                              frequency)
    #print heap
    while len(heap) > 1:
        lo = heappop(heap)# smallest of heap element is popped(min heap)
        hi = heappop(heap)#2nd smallest of heap element is popped(min heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]#assigning 0 to the left element
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]#assigning 1 to the left element
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])#building tree structure
        #print heap
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))#sorting the heap based on
                                                                   bits represented
```

```
#driver program
print 'enter the text'
txt = raw_input()
symb2freq = defaultdict(int)
for ch in txt:
    symb2freq[ch] += 1#calculating the frequency of characters
#print symb2freq
huff = encode(symb2freq)
#print huff
print "Symbol\tWeight\tHuffman Code"
for p in huff:
    print "%s\t%s\t%s" % (p[0], symb2freq[p[0]], p[1])
```

****

Sample input:

enter the text: abbcdac

Sample output:

Symbol	Weight	Huffman Code
a	2	01
b	2	10
c	2	11
d	1	00

Coin change problem

```
#python2 filename.py

def find(amount,n):
	result=[]
	i=n-1
	while (i>=0):
		#comparing the amount and given changes
		while(amount>=coins[i]):
		#finding the remaining amount to fill using the change
			amount=amount- coins[i]
			result.append(coins[i]) #adding the change value to a list
		i-=1


	print result #printing the list of coins
	print len(result) #printing the number of coins

#driver function
print 'enter the no:of coins'
coin_number=input()
coins=[]#list of coins
print 'enter the changes'
for i in range(coin_number):
	i=input()
	coins.append(i)
print 'enter the amount'
amount=input()
find(amount,coin_number)
```

****

Sample input:

enter the no:of coins: 3

enter the changes: 1 4 6

enter the amount: 8

Sample output:

3

Dynamic Programming

Chain Matrix Multiplication

```
#python2 filename.py

import sys

# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
	# For simplicity of the program, one extra row and one
	# extra column are allocated in m[][]. 0th row and 0th
	# column of m[][] are not used
#creating a list of 'n' 0's , 'n' times
#which is equivalent to matrix with 5 rows and 5 colums initialized to 0
  #0th row and 0th used for simplicity
	m = [[0 for x in range(n)] for x in range(n)]

	# m[i,j] = Minimum number of scalar multiplications needed
	# cost is zero when multiplying one matrix.
	for i in range(1, n):
		m[i][i] = 0# cost zero for diagonal elements

	for L in range(2, n): # L is chain length.
		for i in range(1, n-L+1):
			j = i+L-1
			m[i][j] = sys.maxint#assigning infinity
			for k in range(i, j):
				q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j] #calculating the cost
				if q < m[i][j]:
					m[i][j] = q #assigning the minimum cost

	return m[1][n-1]
```

```
#driver function
arr=[]
arr1=0
print 'enter the no:of matrices'
mtrx=input()

for i in range(mtrx+1):
	print 'enter the order list'
	arr1=input()
	arr.append(arr1)

#arr = [50, 10, 40 ,30 ,5]
size = len(arr)

print("Minimum number of multiplications is " +
	str(MatrixChainOrder(arr, size)))
```

****

Sample input:

enter the no:of matrices:4 (ie ABCD)

enter the order list : 50 10 40 30 5

Sample output:

Minimum number of multiplications is 10500

0/1 Knapsack

```
#python2 filename.py

def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)] #creating matrix with extra row and column
    #print K
    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0#assigning 0's to extra row column
            #considering the max value(using recursion formula)
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]

#driver function
val=[]
val1=0
wt=[]
wt1=0
W=0
print 'enter the no:of items'
items=input()
print 'enter the values'
for i in range(items):
    val1=input()
    val.append(val1)#list of values
```

```
#val = [10, 40, 30, 50]
print 'enter the weights'
for i in range(items):
    wt1=input()
    wt.append(wt1)#list of weights
#wt = [5, 4, 6, 3]
print 'enter the max weight'
W =input()
# W=10
n = len(val)
print'maximum value that can be put in a knapsack of capacity '+str(W)+' is '+str((knapSack(W, wt, val, n)))
```
****

Sample input:

enter the no:of items

4

enter the values

10 40 30 50

enter the weights

5 4 6 3

enter the max weight

10

Sample output:

maximum value that can be put in a knapsack of capacity 10 is 90

Longest Common Subsequence

```
#python2 filename.py

def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0#assigning 0's for extra row and column
            elif X[i-1] == Y[j-1]: #if match found
                L[i][j] = L[i-1][j-1]+1
            else:                   #if match not found
                L[i][j] = max(L[i-1][j] , L[i][j-1])



    return L[m][n] #returning the lcs length
#end of function lcs



# driver function
print 'enter the first string'
X = raw_input()
print 'enter the second string'
Y = raw_input()
print "Length of LCS is ", lcs(X, Y)
```

****

Sample input:

enter the first string

ABCBB

enter the second string

ABCB

Sample output:

Length of LCS is 4

Coin change problem

```
#python2 filename.py
def recDC(coinValueList,change,knownResults)
  minCoins = change
  #if the amount itself if present among values then return 1
  if change in coinValueList:
    knownResults[change] = 1
    return 1
  elif knownResults[change] > 0:
    return knownResults[change]
    #recursive calls for each different coin value
    #less than the amount of change we are trying to make
  else:
     for i in [c for c in coinValueList if c <= change]:
      numCoins = 1 + recDC(coinValueList, change-i,
                     knownResults)
      if numCoins < minCoins:
        minCoins = numCoins
        #adding te result to the list ' knownResults'
        knownResults[change] = minCoins
  return minCoins
#driver function
print "enter the no:of coins"
n=input()
coinValueList=[]
#creating a list of coins
print "enter values"
for i in range(n):
  i=input()
  coinValueList.append(i)
print "enter the amount"
change=input()
print(recDC(coinValueList,change,[0]*(change+1)))
```

Sample input:

enter the no:of coins

3

enter values

1

4

6

enter the amount

8

Sample output:

2

Floyd-Warshall's algorithm

```
#python2 filename.py

def floydwarshall(graph):

    # Initialize dist and pred:
    # copy graph into dist, add infinite where there is
    # no edge, and 0 in the diagonal
    dist = {}

    for u in graph:
        dist[u] = {}

        for v in graph:
            dist[u][v] = 1000

        dist[u][u] = 0
        for neighbor in graph[u]:
            dist[u][neighbor] = graph[u][neighbor]



    for t in graph:
        # given dist u to v, check if path u - t - v is shorter
        for u in graph:
            for v in graph:
                newdist = dist[u][t] + dist[t][v]
                if newdist < dist[u][v]:
                    dist[u][v] = newdist

    return dist
```

```
#Driver function

graph = {1 : {5:-4, 2:3, 3:8},
         2 : {5: 7, 4:1},
         3 : {2:4},
         4 : {1:2, 3:-5},
         5 : {4:6}}

dist = floydwarshall(graph)

print 'Shortest distance from each vertex:';
for v in dist:
   print '%s: %s' % (v, dist[v])
```

****

Sample output:

```
Shortest distance from each vertex:
1 : {1: 0, 2: 1, 3: -3, 4: 2, 5: -4}
2 : {1: 3, 2: 0, 3: -4, 4: 1, 5: -1}
3 : {1: 7, 2: 4, 3: 0, 4: 5, 5: 3}
4 : {1: 2, 2: -1, 3: -5, 4: 0, 5: -2}
5 : {1: 8, 2: 5, 3: 1, 4: 6, 5: 0}
```

Cycle detection

```
#python2 filename.py

from collections import defaultdict
def cycle(adj):
	visit={u:False for u in adj}#initialize the nodes with False
	found=[False]#initialize found(cycle) with False

	for u in adj:
		if not visit[u]:#if u is not visited
			dfs(adj,u,found,u,visit)#calling dfs
		if found[0]:#if cycle found then return True else return False
			break
	return found[0]



def dfs(adj,u,found,prenode,visit):
	if found[0]:
		return
	visit[u]=True#make u visited
	for i in adj[u]:
		if visit[i] and i!=prenode:#if node i is visited and is not a prenode then a cycle
						#exist
			found[0]=True
			return
		if not visit[i]:#if i not visited then call dfs
			dfs(adj,i,found,u,visit)
```

```
#driver function
adj={}
print "enter th no:of vertex"
v=input()
print "enter the no:of edges"
e=input()
adj=defaultdict(list)
for i in range(1,e+1):
	print "enter the edges"
	a=input()
	b=input()
	adj[a].append(b)#creating adjacency list for the undirected graph
	adj[b].append(a)
adj=dict(adj)
print adj
value=cycle(adj)#calling the function to check for cycles
print value#printing the result
```

****