Theoretical Assignment 3

Amit Yadav

March 2018

Question 1

Given:

- A min heap,number of elements in heap, a number x, a number k ($k < n$)

Algorithm:

```
function(heap_array, n, x, k, index){
  static int i=0;
  if(i>=k) return 1; // 1 represent x is smaller than k_th
smallest element
  if(index>n) return 0; //reached the end
  if(heap_array[index]<x){
     i++;
     return function(heap_array, n, x, k, 2*index) | function(
heap_array, n, x, k, 2*index+1);
  }
  else return 0;
}

void main(){
  /* heap_array = [0,a1,a2,a3,a4,...,an] */
  function(heap_array, n, x, k, 1);
}
```

Explanation:
Static int i counts number of elements less than k. The moment $i \geq k$, our function returns 1.

Time complexity:
It take k+t max number of comparisions, where t is a number dependent upon heap and k. Therefore,

$$\text{Time} = O(k)$$

$$\text{Space} = O(1)$$

Question 2

Part 1

Aim: To find if an undirected graph is acyclic or not.
Graph is given as a Adjacency list. As known, DFS visit takes O(V+E) time. I will do the same DFS visit but it will halt execution as soon as a cycle is found. Since, maximum length of a cycle can be $\mid V \mid$, so it will take max O(V) time. Worst case will be when it visits all the vertices and edges, which is only possible in case of no cycles present. In that case $\mid E \mid \leq \mid V - 1 \mid$, equality holds when graph is connected. So, time will be O(2V) = O(V).
Algorithm:

```
1       cycleFound=False
2       DFS-visit(v, Adj){
3         for u in Adj[v]:
4           if(cycleFound==True):
5             return;
6           if(u!=parent[v]):
7             if(mark[u]==1):
8               cycleFound=True; """If it visits a vertex which is
    already in process, means cycle is found"""
9               return;
10            else:
11              parent[u]=v;
12              mark[u]=1;
13              DFS-visit(u, Adj);
14              mark[u]=0;
15      }
16
17      DFS(Adj){
18        for v in V:
19          if v not in parent:
20            if(cycleFound==True):
21              return;
22            parent[v]=None;
23            mark[v]=1;
24            DFS-visit(v, Adj);
25            mark[v]=0;
26      }
27
28      DFS(Adj); """calling DFS function"""
29
```

Part 2

Aim: To find if a given undirected graph is tree or not.
An undirected acyclic graph is a tree if it is connected. So, we essentially need to check if it is acyclic or not, and additionally, if it is connected or not. So our Algorithm will be almost same as **Part 1**, except there will be one additional condition.
Algorithm:

```
    cycleFound=False
    Tree=True

    DFS-visit(v, Adj){
      for u in Adj[v]:
        if(cycleFound==True):
          return;
        if(u!=parent[v]):
          if(mark[u]==1):
            cycleFound=True; """If it visits a vertex which is already in process, means cycle is found"""
            return;
          else:
            parent[u]=v;
            mark[u]=1;
            DFS-visit(u, Adj);
            mark[u]=0;
    }

    DFS(Adj){
      parent[v]=None;
      mark[v]=1;
      DFS-visit(v, Adj);
      mark[v]=0;
      for v in V: """We run DFS-visit on any of the vertex and all the vertices should have been visited if it is a tree"""
        if v not in parent:
          Tree=False;
    }

    DFS(Adj); """calling DFS function"""

```

Question 3

First we build a undirected graph with given words as vertices and two vertiices are connected if one is **allowed edit** of other. And then we run **BFS on word A**.
So first we need to build the graph from a given D.

```
allowed_edit(word1,word2){
  diff=0;
  size_diff=abs(len(word1)-len(word2));
  if size_diff>1:
    return 0;
  for i in range(0,min( len(word1), len(word1) )):
    if[word1[i]!=word2[i]]:
        diff+=1;
  if(size_diff==0 and diff==1):
    return 1;
  if(size_diff==1 and diff==0):
    return 1;
  return 0;
}

```

```
create_graph{
  for i in range(0, len(D)):
    for j in range(i+1, len(D)):
      if allowed_edit(D[i],D[j]):
        Adj[D[i]].append(D[j]);
        Adj[D[j]].append(D[i]);
}
```

Building graph takes $O(n^2c^2)$ time.
Then running time of BFS on given word is $O(V + E)$, where $\mid V \mid = len(D)$.

```
BFS(B, Adj){
  level={B:0};
  parent={B:None}
  i=1;
  x=[B]
  while(x):
    neighbours=[];
    for u in x:
      for v in Adj[u]:
        if v not in level:
          level[v]=i;
          parent[v]=u;
          neighbours.append(v);
          if(v==A): """stop when we reach word A"""
            x=[];
            break;
      x=neighbours;
      i+=1;

  while(A!=None):
    print(A);
    A=parent[A];
}
```

Time Complexity:

1. Creating Graph:
   Since, it iterates over D for every word i.e $O(n^2)$, and every check of allowed edit take $O(c^2)$ time, so total time complexity of creating Adjacency list is $O(n^2c^2)$.

2. BFS on word B:
   Since BFS takes O(V+E) time, in worst case, $\mid E \mid = O(\mid V \mid^2)$. So, BFS take $O(n^2)$ time.

Therefore total time can be expressed as $O(n^2c^2)$.
This is also the worst case time complexity of the algorithm.

Question 4

Given: $h(x) = xmod10$, a hash table with some entries.
To find: Number of ways to create this given hash table.
Sol. 42, 23, 34 and **46** are at their excpected places but **52** and **33** are not. So it is clear that 42, 23 and 34 must be inserted before 52, else 52 would have occupied one of their places. Also, 33 must be inserted after 23, 34, 52 and 46 because of the same reason. Since 52 is inserted before 33 implies 42 is also insertes before 33, making 33 to be inserted in the last. So possible orders of insertion are:

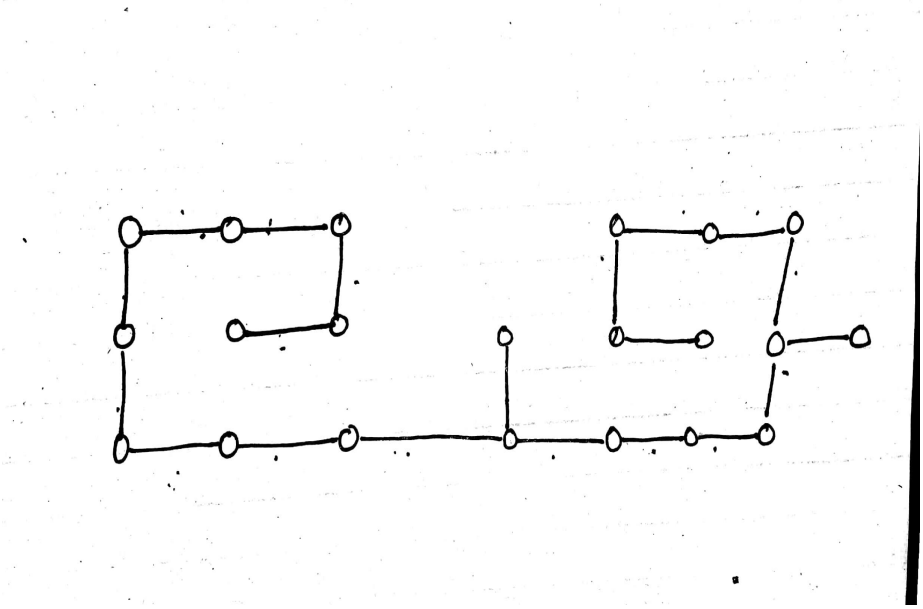
$\underbrace{42, 23, 34}_{3!}, 52, 46, 33$

$\underbrace{42, 23, 34, 46}_{4!}, 52, 33$

$Therefore, total number of ways$ =3! + 4! = 30

Question 5:

Spanning tree with maximum height will look like this:



When we run DFS, we keep only the tree edges and no backward edges which lead to cycles. So, the result will be this spanning tree.
Height of the tree is **18**.

Question 6:

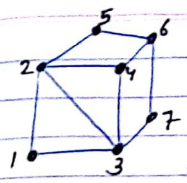
Given: A graph with vertices and edges representing islands and bridges respectively.
Aim: To find a Euler's Path in the graph.
Sol: Euler's path exists only if the graph has two vertices with odd degree or no vertex with odd degree.
Now we taking a vertex as starting point as follows:

- If the graph has all nodes with even degree, choose any point as starting point.
- If it has exactly two vertices with odd degree, choose any one of them as starting point.
- Else, Euler's path do not exist.

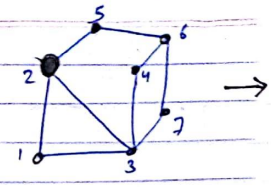
Procedure
We maintain two stacks: **STACK**, **PATH**. PATH is the final stack which contains our resulting Euler's path. **Step 2**

- If the present vertex has no neighbour, then:
  - Push the present vertex to **PATH**.
  - Pop the last pushed vertex in the stack, and make it present vertex.
- Else:
  - Push the present vertex to **STACK** and make nay of it's neighbour present vertex.
  - Remove the edge between last vertex and present vertex.

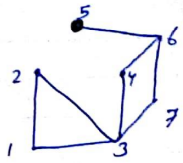
Repeat Step 2 till there is no neighbour of present vertex is left **and STACK** get empty.
Then push the last present element to **PATH**. Now **PATH** contains a Euler's path.
An example is shown below:



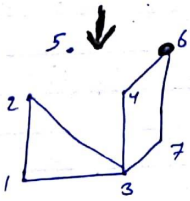
STACK = empty
PATH = empty
Present V. = 4



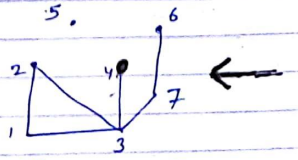
STACK = 4
PATH = empty
P.V = 2



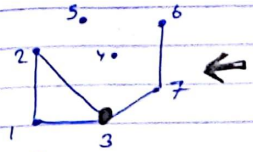
STACK = 4,2
PATH = empty
P.V = 5



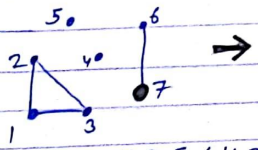
STACK = 4,2,5
PATH = ϕ
P.V = 6



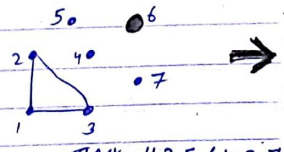
STACK = 4,2,5,6
PATH = ϕ
P.V = 4



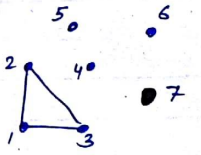
STACK = 4,2,5,6,4
PATH = ϕ
P.V = 3



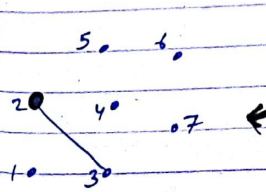
STACK = 4,2,5,6,4,3
PATH = ϕ
P.V = 7



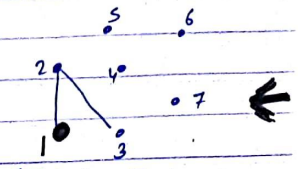
STACK = 4,2,5,6,4,3,7
PATH = ϕ
P.V = 6



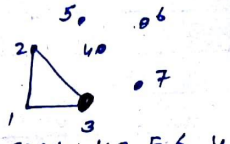
STACK = 4,2,5,6,4,3
PATH = 6
P.V = 7



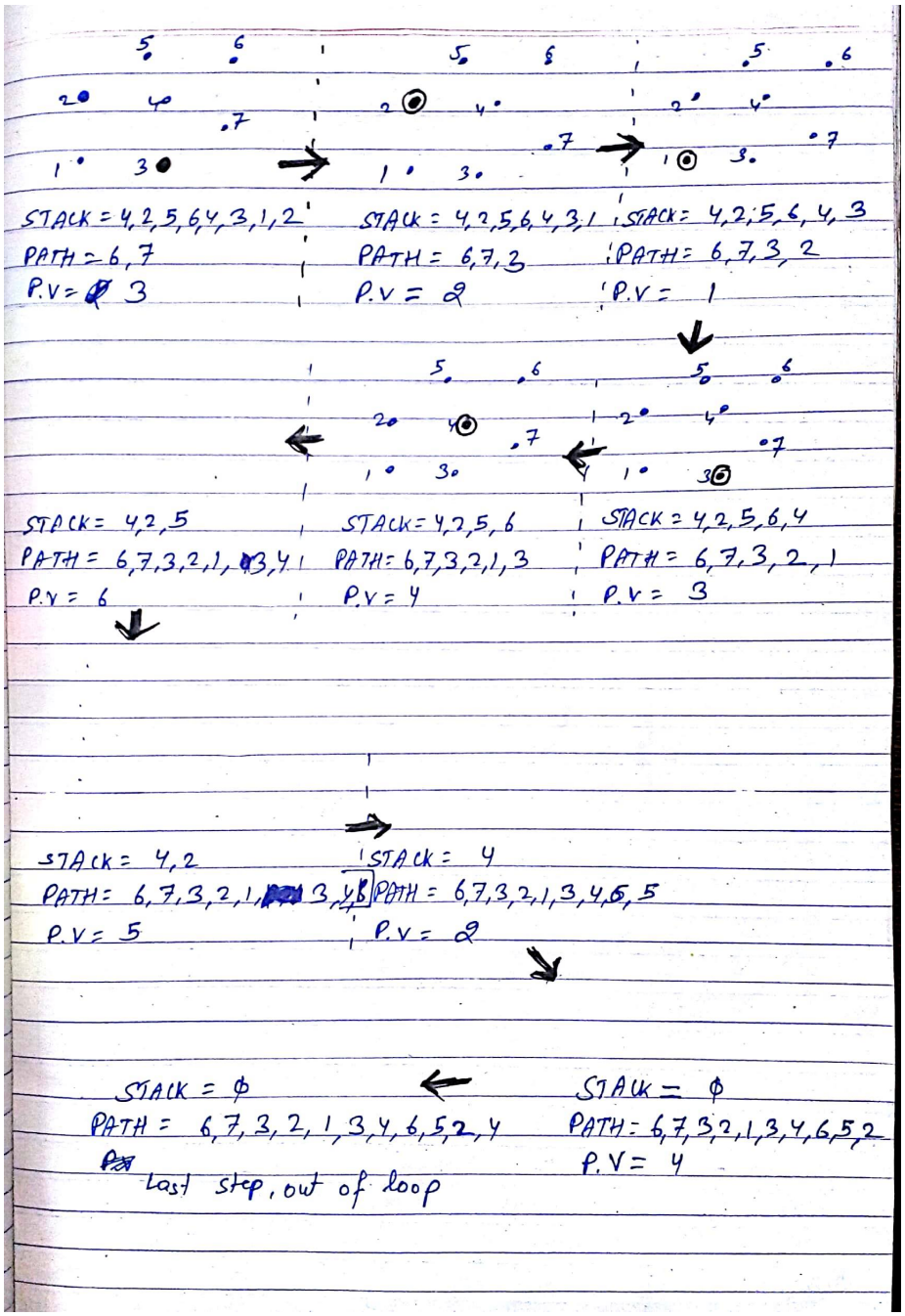
STACK = 4,2,5,6,4,3,1
PATH = 6,7
P.V = 9



STACK = 4,2,5,6,4,3
PATH = 6,7
P.V = 1



STACK = 4,2,5,6,4
PATH = 6,7
P.V = 3


STACK = 4,2,5,6,4,3,1,2
PATH = 6,7
P.V = 3
STACK = 4,2,5,6,4,3,1
PATH = 6,7,3
P.V = 2
STACK = 4,2,5,6,4,3
PATH = 6,7,3,2
P.V = 1
STACK = 4,2,5
PATH = 6,7,3,2,1,3,4
P.V = 6
STACK = 4,2,5,6
PATH = 6,7,3,2,1,3
P.V = 4
STACK = 4,2,5,6,4
PATH = 6,7,3,2,1
P.V = 3
STACK = 4,2
PATH = 6,7,3,2,1,3,4,6
P.V = 5
STACK = 4
PATH = 6,7,3,2,1,3,4,6,5
P.V = 2
STACK = ϕ
PATH = 6,7,3,2,1,3,4,6,5,2,4
Last step, out of loop
STACK = ϕ
PATH = 6,7,3,2,1,3,4,6,5,2
P.V = 4