Lecture planning

Algorithms -1

By-

Srimahn V	- 19HS20050
Tushar Mohta	- 19HS20055
K.Yogeshwara Krishna	- 19EC10032

Problem Statement

In *“Charles Xavier's School for Gifted Youngsters”* the professors have to plan for the autumn semester 2021. In the particular subject of history taught by Professor Logan, he has to decide the correct schedule of topics to teach students and as we know history has to be taught in the right order of events. Let us simplify this schedule of topics by variables . He plans to give **n lectures** on **n different topics**. Each topic should completed properly in a single lecture (so that it’s easier for the students to understand the flow). We would like to choose which topic will be explained during the 1st, 2nd, …, nth lecture.

For each topic (except of course the first topic), **there exists a prerequisite**(an event occurred in history before another event) :-for the topic **i**, the prerequisite is **p_i**). Obviously he cannot give a lecture on an event(topic) before giving a lecture on its prerequisite topic **BUT**, there are some topics which **<u>MUST</u>** be taught right after each other. Let's say we have **k pairs** of topics **(x_i , y_i)** such that its beneficial if **y_i is conducted immediately after x_i.**

For example :
Prerequisite: They have to learn about world war 1 before world war 2 (not necessarily immediately)
Immediate topics: *"The United States detonated two nuclear weapons over the Japanese cities of Hiroshima and Nagasaki on August 6 and 9, 1945, respectively."*

Professor Logan is good at history (among other things), But not at algorithms, he needs help in finding a good schedule of topics to teach the students.

Essentially what we have to do is...

Given a set of pairs (a,b) of numbers where b must come immediately after a AND we have another set of pairs of numbers (p,q which are prerequisites) p must come somewhere after q in the final sequence, we must design a sequence of numbers that follow both these conditions.

We want to choose some permutation of integers from 1 to n (let's call this permutation q). q_i is the index of the topic Prof.Logan will explain during the ith lecture.

Constraints

- $2 \le n \le 3*10^5$
- $1 \le k \le n-1$
- For each prerequisite p_i : $0 \le p_i \le n$
- For each pair of immediate topics: $1 \le x_i , y_i \le n$ AND $x_i \ne y_i$
- All values of x_i are pairwise distinct; similarly, all values of y_i are pairwise distinct.

Output / What we need.

If there does not exist any schedule which follows all the given conditions, just print 0 or tell ”No”, else give **any correct schedule**.

Some examples: Example 1

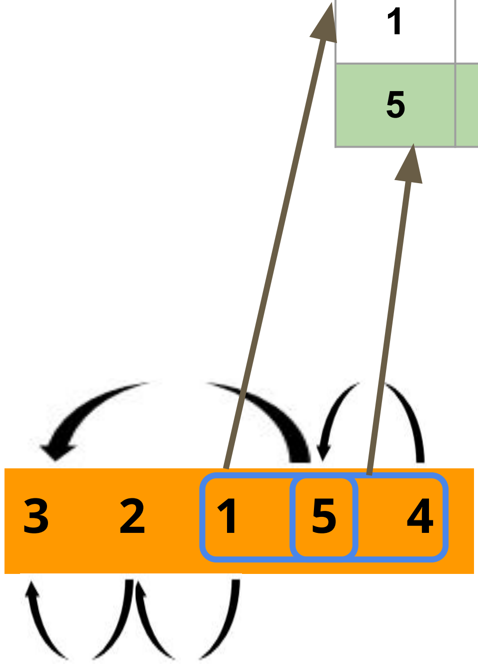
N = 5 and K = 2

Topic number	It’s prerequisite
1	2
2	3
3	0 (no prerequisite)
4	5
5	3

K (here 2) pairs of immediately following topics

x_i	y_i
1	5
5	4

Optimal solution:



Example 2

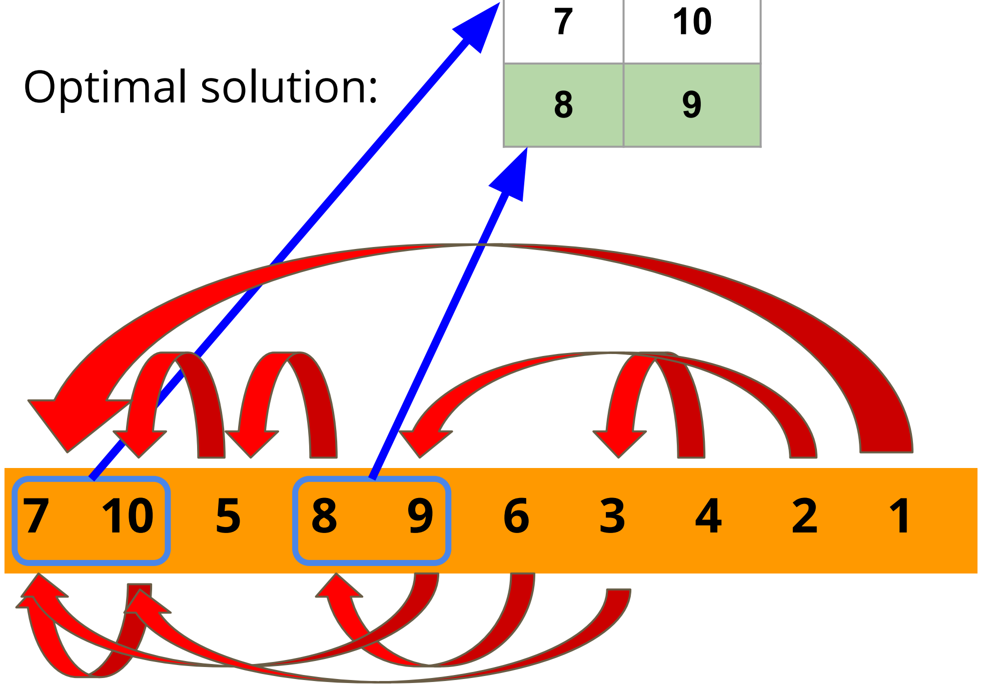
N = 10 and K = 2

Topic number	It's prerequisite
1	7
2	9
3	10
4	3
5	10
6	8
7	0(no prerequisite)
8	5
9	7
10	7

K (here 2) pairs of immediately following topics

Optimal solution:

x_i	y_i
7	10
8	9



Example 3

N = 5 and K = 2

Topic number	It's prerequisite
1	2
2	3
3	0(no prerequisite)
4	5
5	3

K (here 2) pairs of immediately following topics

x_i	y_i
1	5
5	1

Incompatible! Here the immediately followed topics themselves conflict with each other.

As we will come to know later, this case is like a bidirectional edge between two nodes, which is not plausible.

No Solution! ✗

Example 4

N = 5 and K = 1

Topic number	It's prerequisite
1	2
2	3
3	0(no prerequisite)
4	5
5	3

K (here 1) pairs of immediately following topics

x_i	y_i
4	5

Incompatible! Here the immediately followed topic conflict with the prerequisite.

As we will come to know later, this case is like a cycle in different type of graph, which is not plausible.

No Solution!

Example 5

N = 3 and K = 1

K (here 1) pairs of immediately following topics

Topic number	It's prerequisite
1	0(no prerequisite)
2	3
3	1

x_i	y_i
1	2

✖

Incompatible! It's not easy to detect by observation, a set of prerequisites violate the immediate topics condition.

This case is also like a cycle in different type of graph, which is not plausible.

No Solution! ✖

Some minor observations.

Whichever lecture has no prerequisite - always comes at the beginning.

Whenever we have an Immediate pair of topics , we can immediately put them together in our schedule, no matter where they *collectively* are in the whole schedule.

Many pairs of these immediate topics may not be disjoint and hence we can club them all together and consider them as one entity, we will use this idea further.

So.....

What do we immediately think of when we have to consider a set of topics as a single entity and sometimes even single topics as a single entity??

WE USE GRAPHS !!

In particular we need **Directed Acyclic Graphs**! (DAG's).
Reason being:

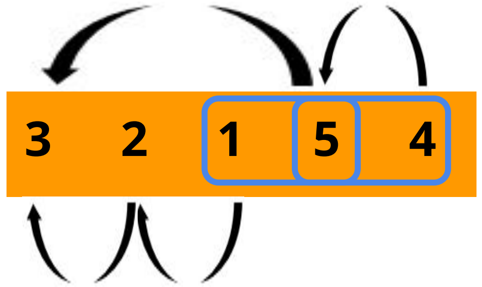
1. **Directed -** The conditions ensure this fact.
2. **Acyclic -** We can't have cycles in them (implausibility).

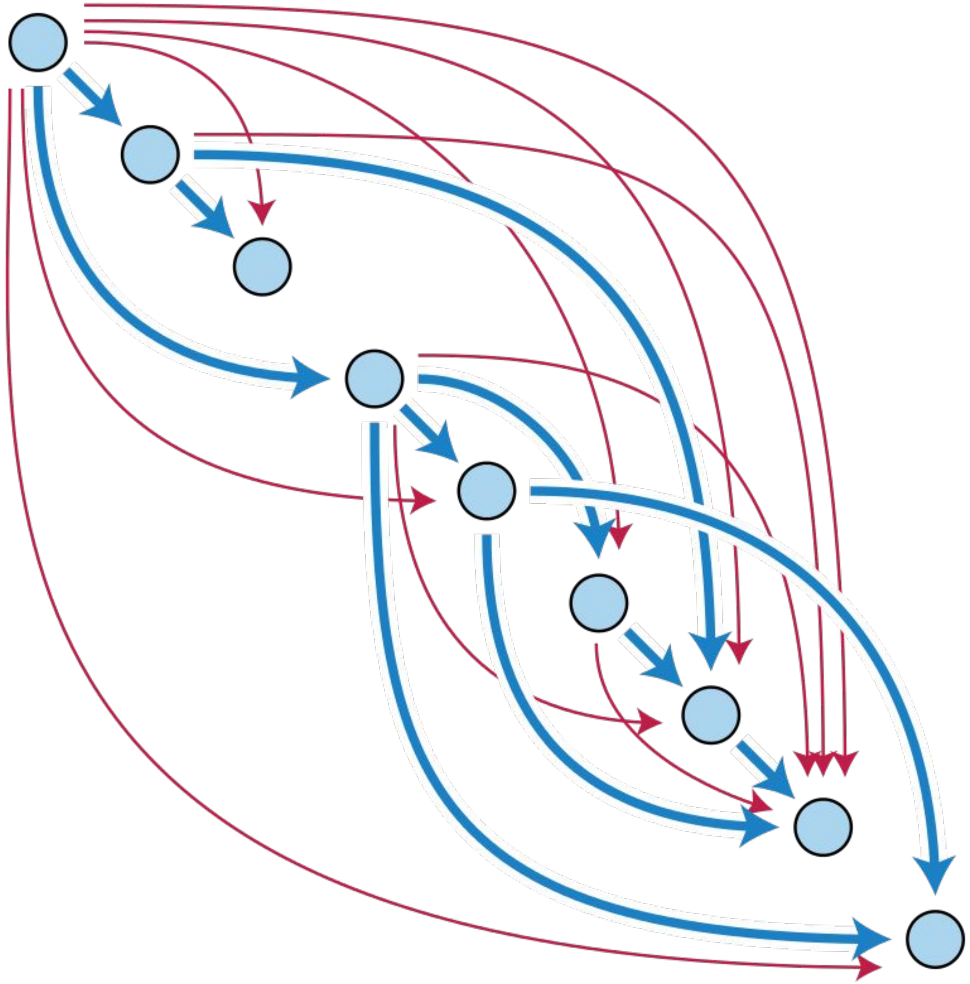
But how do we traverse this graph? (we can't use DFS, BFS etc). The direction of the edges indicate how the elements should be traversed / appear in the output. Hence we use an algorithm called - **topological sorting.**

DAG...

Looks very similar to the example arrows.

Root would be the topic with no prerequisite.





Idea

1. If there are no special pairs → each topic is a single entity (single node) → simple topological sorting would work.

But what is Topological sorting ??

Topological sorting

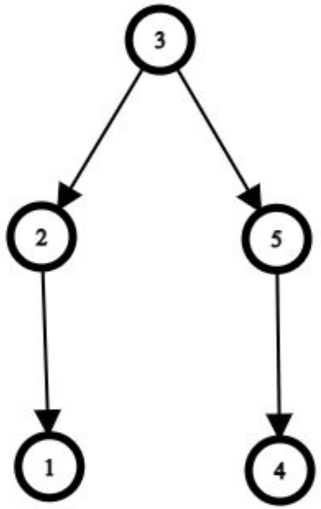
N = 5 and K = 0

Topic number	Pre-req
1	2
2	3
3	0
4	5
5	3

Topological sorting

N = 5 and K = 0

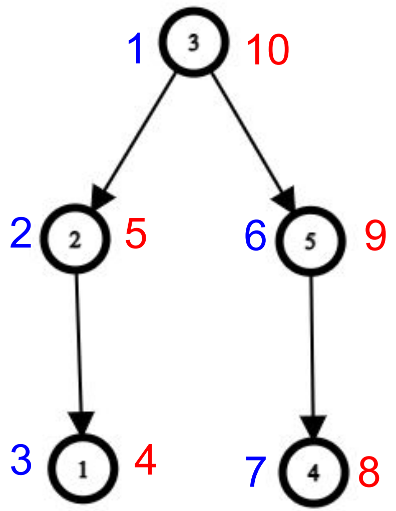
Topic number	Pre-req
1	2
2	3
3	0
4	5
5	3



Topological sorting

N = 5 and K = 0

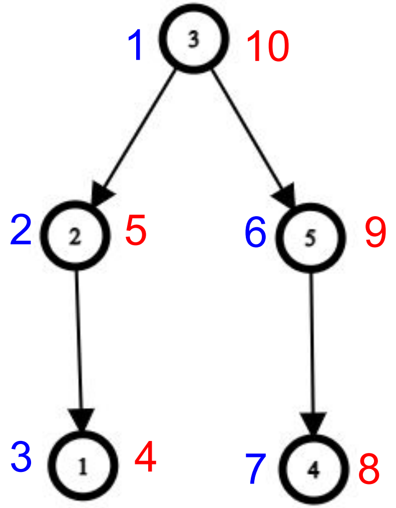
Topic number	Pre-req
1	2
2	3
3	0
4	5
5	3



Topological sorting

N = 5 and K = 0

Topic number	Pre-req
1	2
2	3
3	0
4	5
5	3



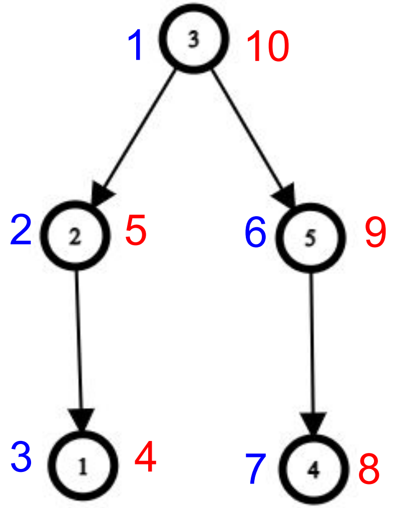
Sorted Order:

3 5 4 2 1

Topological sorting

N = 5 and K = 0

Topic number	Pre-req
1	2
2	3
3	0
4	5
5	3



Sorted Order:

3 5 4 2 1

Idea

1. If there are no special pairs, simple topological sorting would work
2. If there are special pairs, the special pairs are to be arranged together.
    a. Compress the special pairs into a one vertex each and run topological sorting
    b. Expand the compressed version so that ordering is preserved

Input

N = 7 and K = 2

Topic number	Pre-req
1	5
2	7
3	2
4	7
5	0
6	5
7	5

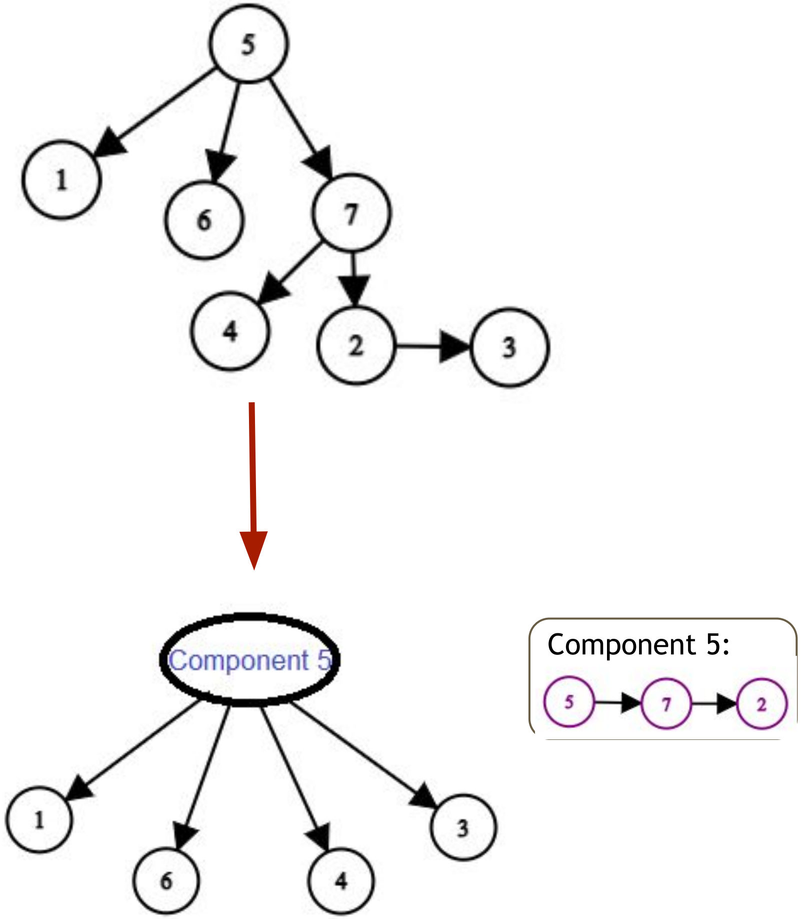
Special pairs:
(5,7) and (7,2)

Compression

N = 7 and K = 2

Topic number	Pre-req
1	5
2	7
3	2
4	7
5	0
6	5
7	5

Special pairs:
(5,7) and (7,2)

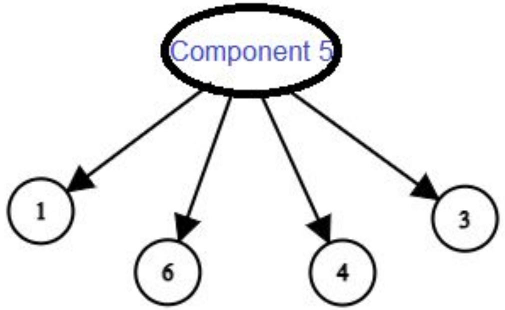


Topological sorting

N = 7 and K = 2

Topic number	Pre-req
1	5
2	7
3	2
4	7
5	0
6	5
7	5

Special pairs:
(5,7) and (7,2)



Top Sort order:

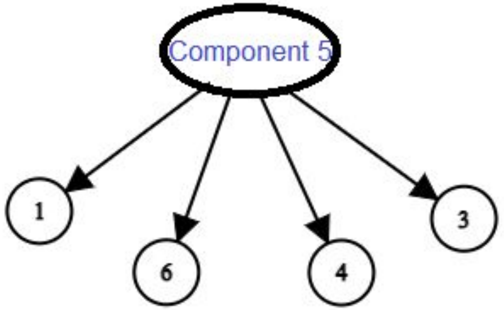
Comp-5 1 3 4 6

Expanding

N = 7 and K = 2

Topic number	Pre-req
1	5
2	7
3	2
4	7
5	0
6	5
7	5

Special pairs:
(5,7) and (7,2)



Top Sort order:

Comp-5 1 3 4 6

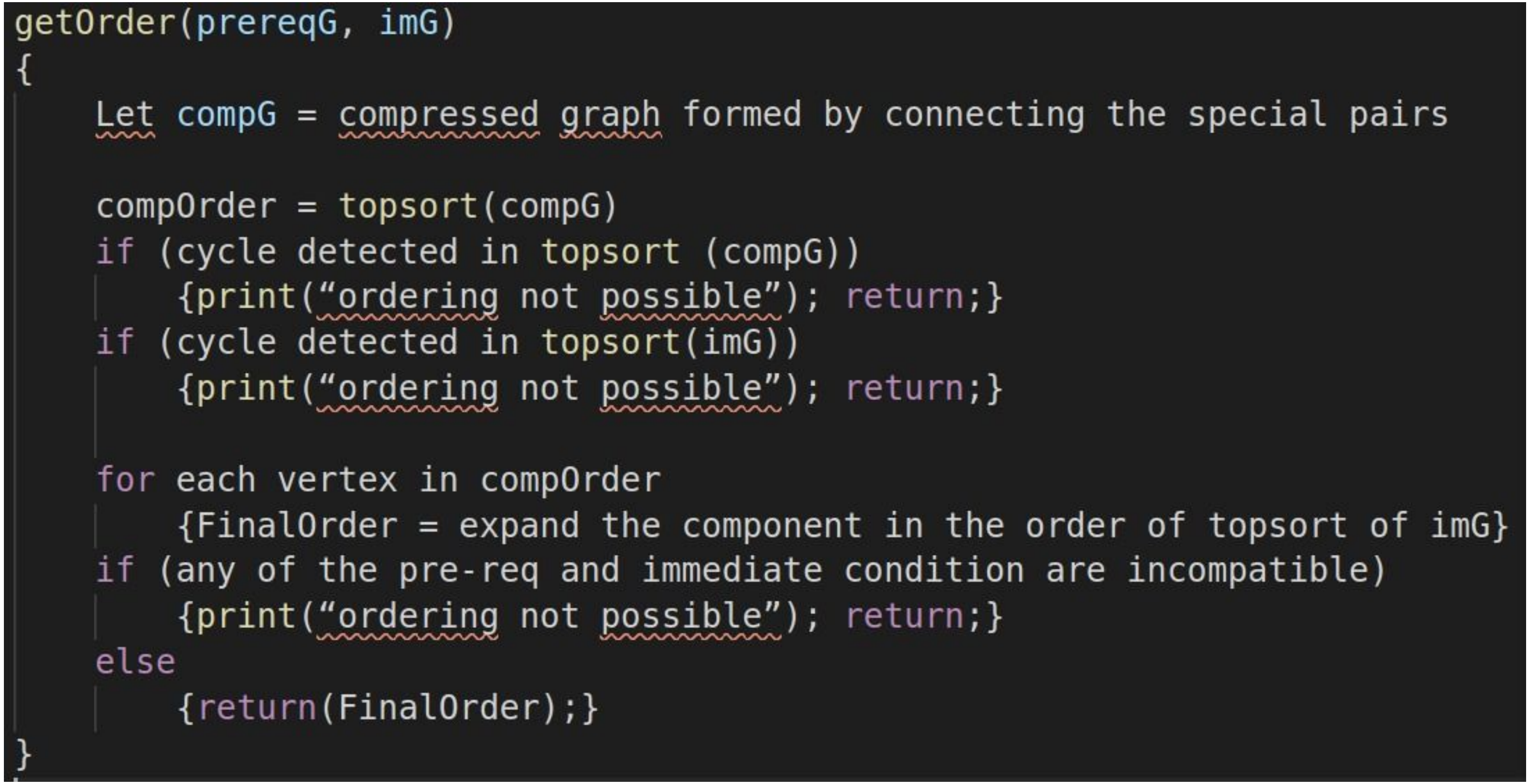
Now, simply expand the component

5 7 2 1 3 4 6

Idea

1. If there are no special pairs, simple topological sorting would work
2. If there are special pairs, the special pairs are to be arranged together.
    a. Compress the special pairs into a one vertex each and run topological sorting
    b. Expand the compressed version so that ordering is preserved
3. There will we no solution
    a. If there are loops either in the components, or in the compressed graph
    b. If any of the pre-requisite and special pair conditions are incompatible

Pseudocode



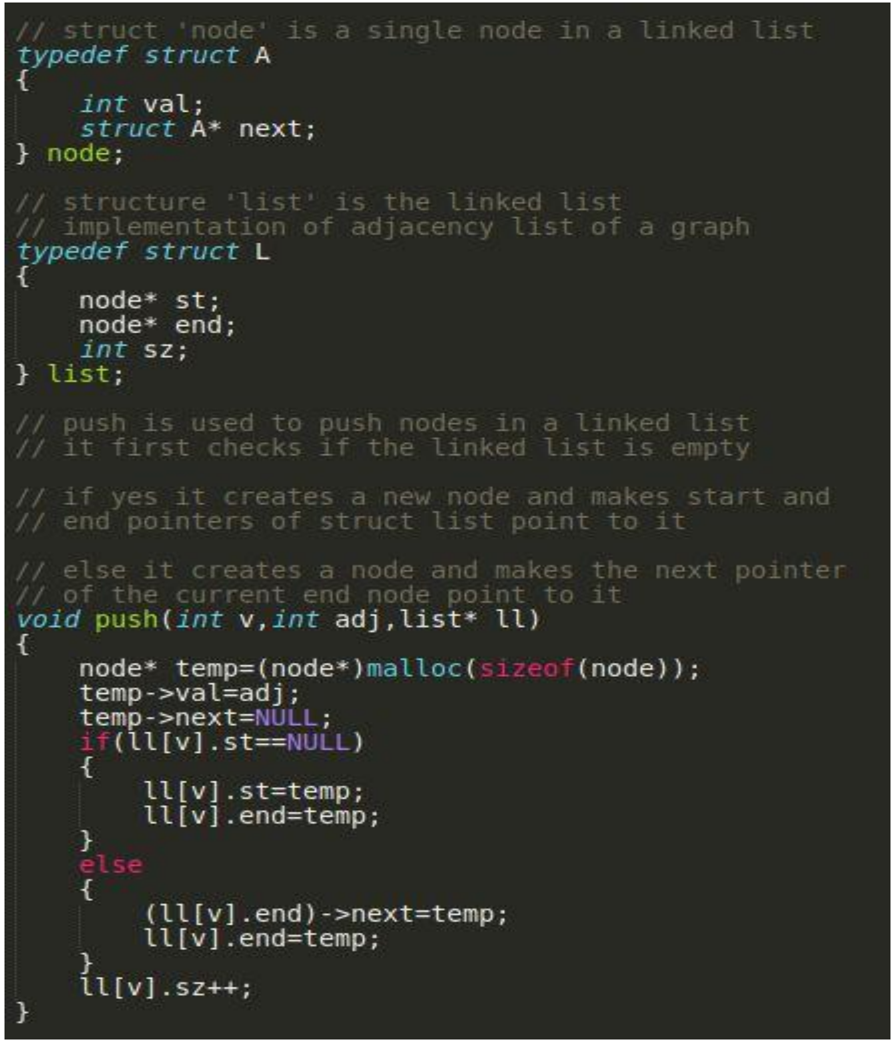
```
getOrder(prereqG, imG)
{
    Let compG = compressed graph formed by connecting the special pairs

    compOrder = topsort(compG)
    if (cycle detected in topsort (compG))
        {print("ordering not possible"); return;}
    if (cycle detected in topsort(imG))
        {print("ordering not possible"); return;}

    for each vertex in compOrder
        {FinalOrder = expand the component in the order of topsort of imG}
    if (any of the pre-req and immediate condition are incompatible)
        {print("ordering not possible"); return;}
    else
        {return(FinalOrder);}
}
```

Implementation in C

Adjacency list

- Struct ‘node’ is a single node in a linked list
- Struct ‘list’ is the linked list implementation of adjacency list of a graph
- Function push() is used in later code to push nodes in a linked list.

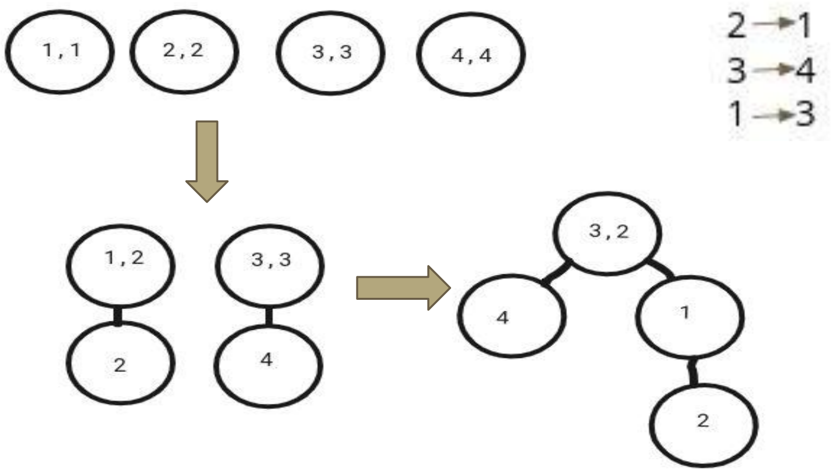
In our implementation push can be done only through the end. Also, since there is no need to pop elements from the linked list in our algorithm we don’t have a function for pop()

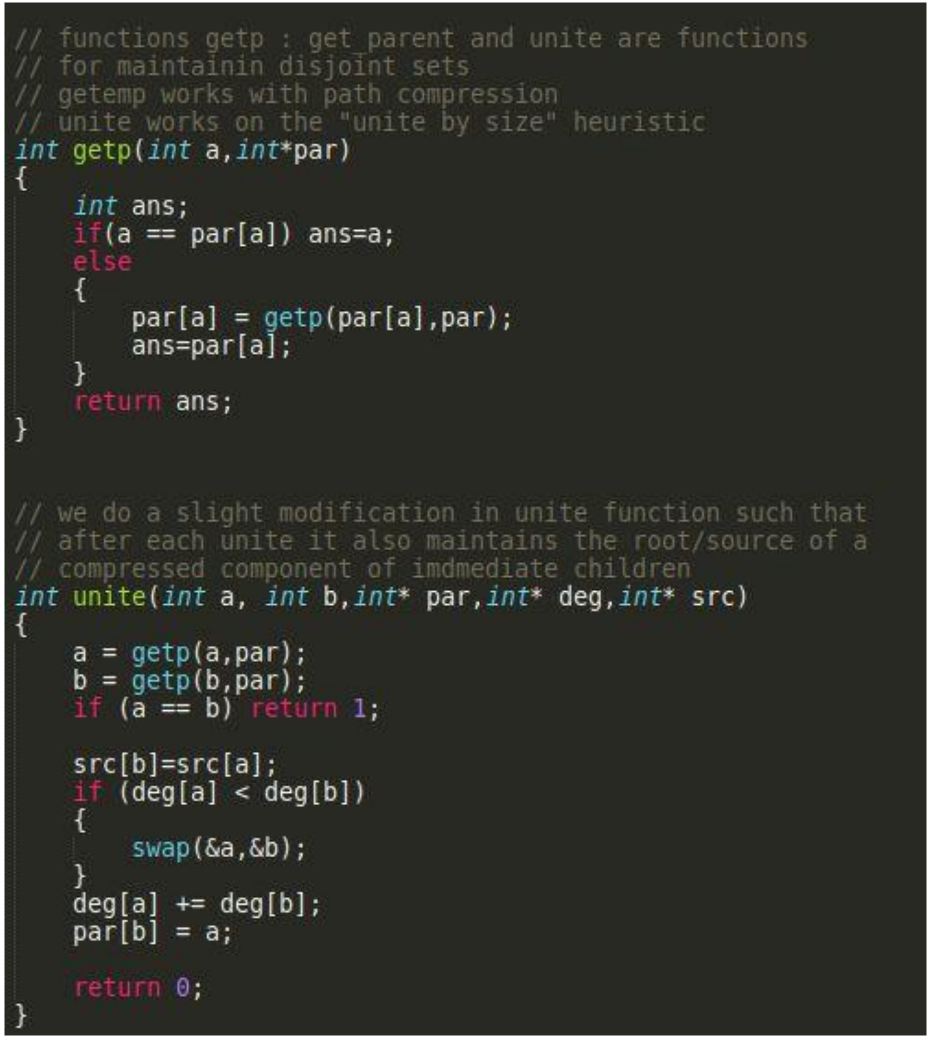


```
// struct 'node' is a single node in a linked list
typedef struct A
{
    int val;
    struct A* next;
} node;

// structure 'list' is the linked list
// implementation of adjacency list of a graph
typedef struct L
{
    node* st;
    node* end;
    int sz;
} list;

// push is used to push nodes in a linked list
// it first checks if the linked list is empty

// if yes it creates a new node and makes start and
// end pointers of struct list point to it

// else it creates a node and makes the next pointer
// of the current end node point to it
void push(int v,int adj,list* ll)
{
    node* temp=(node*)malloc(sizeof(node));
    temp->val=adj;
    temp->next=NULL;
    if(ll[v].st==NULL)
    {
        ll[v].st=temp;
        ll[v].end=temp;
    }
    else
    {
        (ll[v].end)->next=temp;
        ll[v].end=temp;
    }
    ll[v].sz++;
}
```

Disjoint set forests

- Function getp() returns the parent/representative element of a disjoint set simultaneously performing path compression.
- Function unite() unites two disjoint sets using ‘union by size’ heuristic. But, here, we also have another array src inside unite()

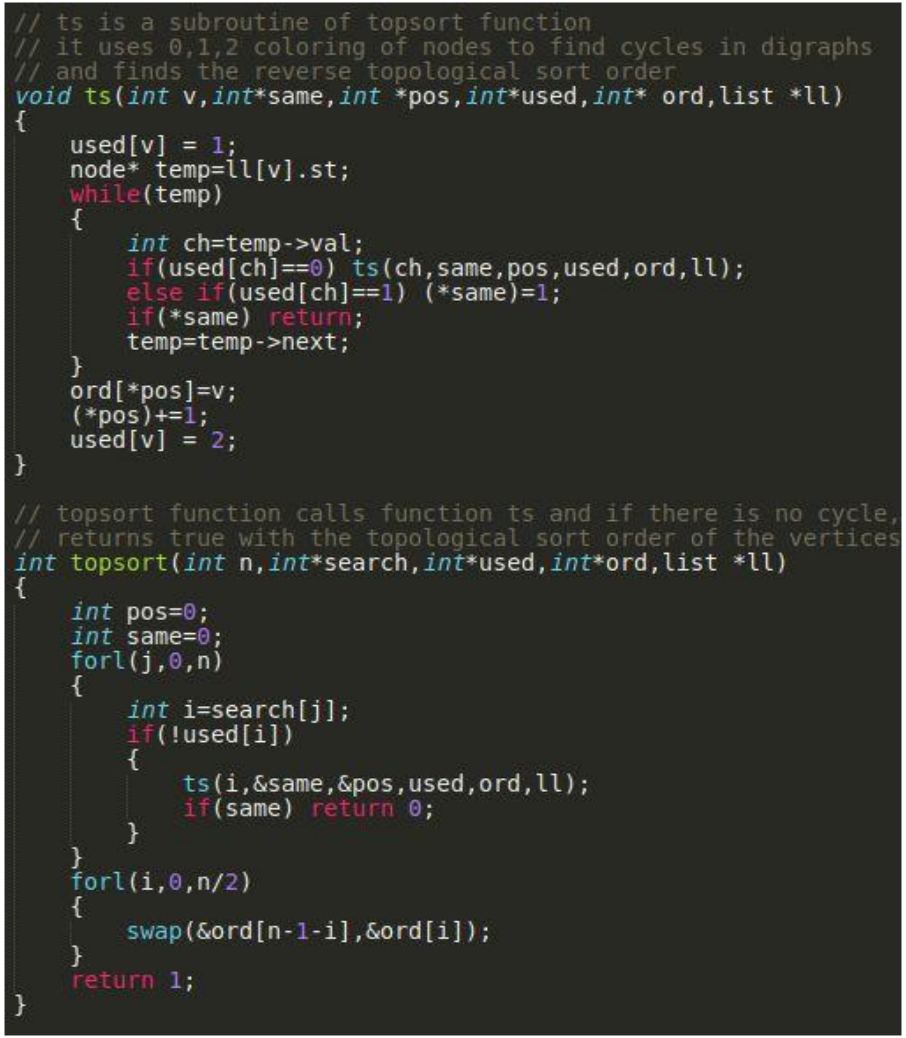




```
// functions getp : get parent and unite are functions
// for maintainin disjoint sets
// getemp works with path compression
// unite works on the "unite by size" heuristic
int getp(int a,int*par)
{
    int ans;
    if(a == par[a]) ans=a;
    else
    {
        par[a] = getp(par[a],par);
        ans=par[a];
    }
    return ans;
}


// we do a slight modification in unite function such that
// after each unite it also maintains the root/source of a
// compressed component of imdmediate children
int unite(int a, int b,int* par,int* deg,int* src)
{
    a = getp(a,par);
    b = getp(b,par);
    if (a == b) return 1;

    src[b]=src[a];
    if (deg[a] < deg[b])
    {
        swap(&a,&b);
    }
    deg[a] += deg[b];
    par[b] = a;

    return 0;
}
```

Topological Sort

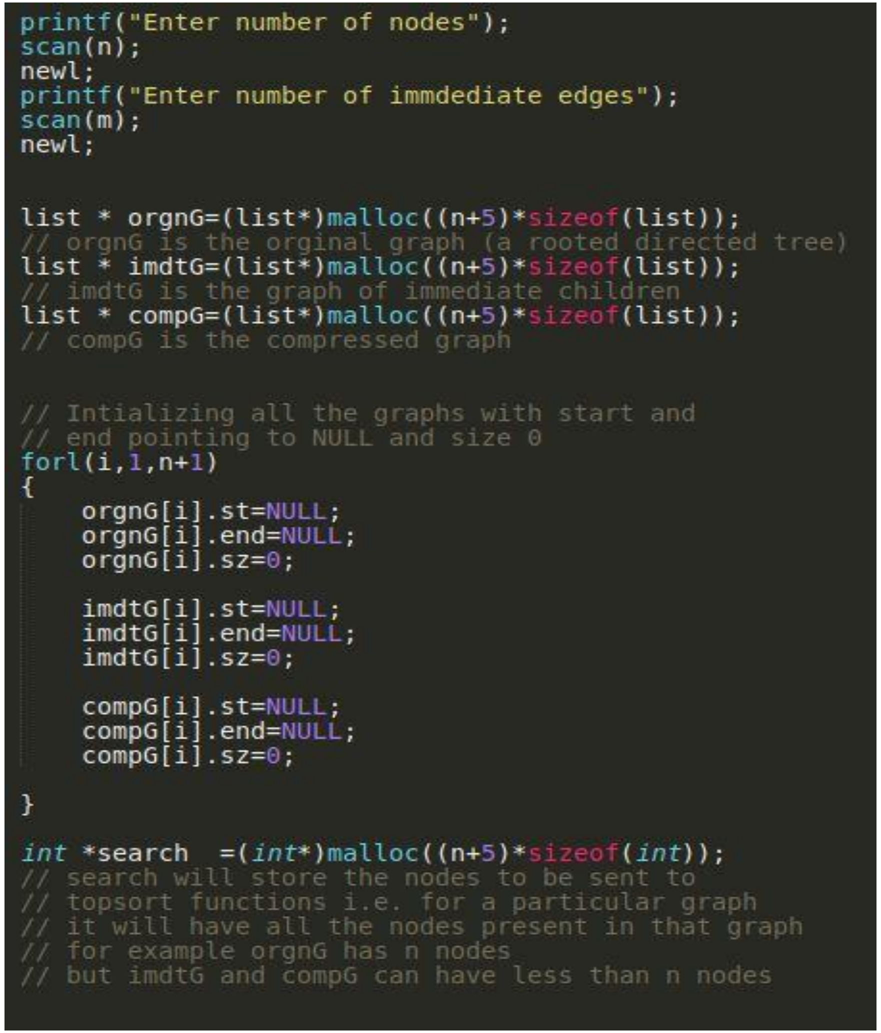
- Function ts() is a subroutine of function topsort(). ts() adds vertices to ord[ ] according to exit times of depth-first-searches
- topsort() returns the topological sort ordering of vertices if cycles are not present in the graph



```c
// ts is a subroutine of topsort function
// it uses 0,1,2 coloring of nodes to find cycles in digraphs
// and finds the reverse topological sort order
void ts(int v,int*same,int *pos,int*used,int* ord,list *ll)
{
    used[v] = 1;
    node* temp=ll[v].st;
    while(temp)
    {
        int ch=temp->val;
        if(used[ch]==0) ts(ch,same,pos,used,ord,ll);
        else if(used[ch]==1) (*same)=1;
        if(*same) return;
        temp=temp->next;
    }
    ord[*pos]=v;
    (*pos)+=1;
    used[v] = 2;
}

// topsort function calls function ts and if there is no cycle,
// returns true with the topological sort order of the vertices
int topsort(int n,int*search,int*used,int*ord,list *ll)
{
    int pos=0;
    int same=0;
    forl(j,0,n)
    {
        int i=search[j];
        if(!used[i])
        {
            ts(i,&same,&pos,used,ord,ll);
            if(same) return 0;
        }
    }
    forl(i,0,n/2)
    {
        swap(&ord[n-1-i],&ord[i]);
    }
    return 1;
}
```

Dynamic Allocation and Initialization

- orgnG is the original graph given as input in the form of a rooted tree
- imdtG is the forest of linear trees of immediate courses
- compG is the compressed graph formed out of orgnG by compressing the nodes of imdtG into representative elements for each tree in imdtG



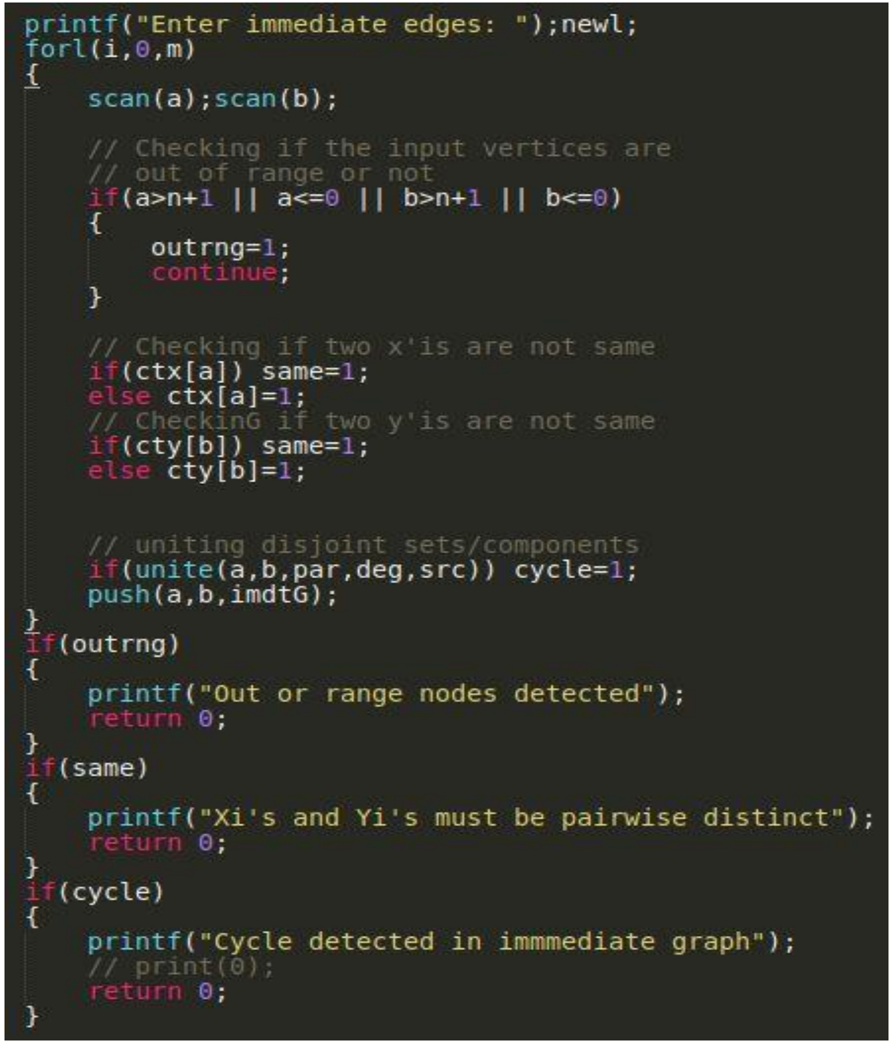
```
printf("Enter number of nodes");
scan(n);
newl;
printf("Enter number of immdediate edges");
scan(m);
newl;


list * orgnG=(list*)malloc((n+5)*sizeof(list));
// orgnG is the orginal graph (a rooted directed tree)
list * imdtG=(list*)malloc((n+5)*sizeof(list));
// imdtG is the graph of immediate children
list * compG=(list*)malloc((n+5)*sizeof(list));
// compG is the compressed graph


// Intializing all the graphs with start and
// end pointing to NULL and size 0
forl(i,1,n+1)
{
    orgnG[i].st=NULL;
    orgnG[i].end=NULL;
    orgnG[i].sz=0;

    imdtG[i].st=NULL;
    imdtG[i].end=NULL;
    imdtG[i].sz=0;

    compG[i].st=NULL;
    compG[i].end=NULL;
    compG[i].sz=0;

}

int *search  =(int*)malloc((n+5)*sizeof(int));
// search will store the nodes to be sent to
// topsort functions i.e. for a particular graph
// it will have all the nodes present in that graph
// for example orgnG has n nodes
// but imdtG and compG can have less than n nodes
```

Input sanity and Checks for cycles

- 'cycle' is a binary variable and checks for cycles in the immediate graph
- 'same' is a binary variable and checks if two x'is or y'is in the immediate graph are same.

  Here, (x’i , y’i) is a directed edge from vertex x’i to vertex y’i

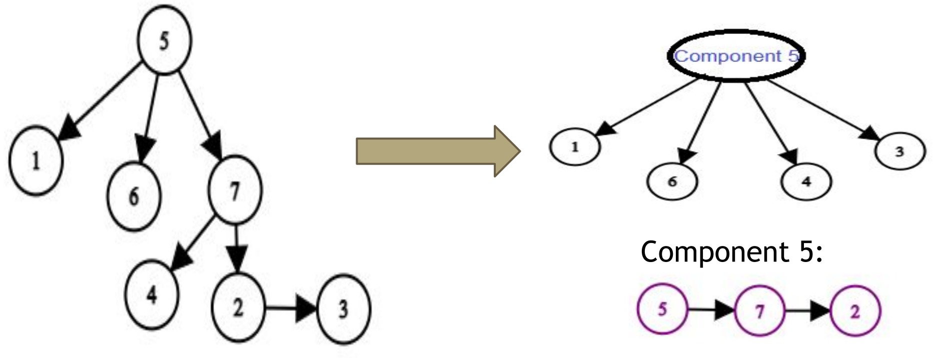
- 'outrng' is a binary variable and checks if the input vertices are out of range

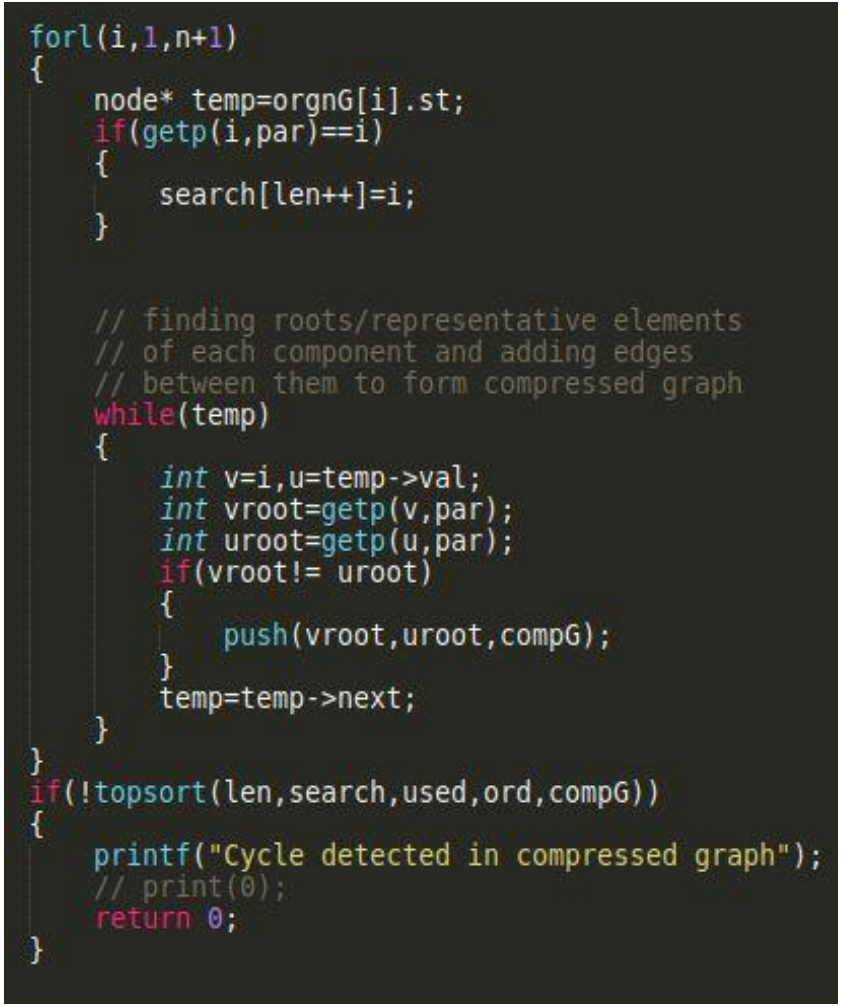


```
printf("Enter immediate edges: ");newl;
forl(i,0,m)
{
    scan(a);scan(b);

    // Checking if the input vertices are
    // out of range or not
    if(a>n+1 || a<=0 || b>n+1 || b<=0)
    {
        outrng=1;
        continue;
    }

    // Checking if two x'is are not same
    if(ctx[a]) same=1;
    else ctx[a]=1;
    // CheckinG if two y'is are not same
    if(cty[b]) same=1;
    else cty[b]=1;


    // uniting disjoint sets/components
    if(unite(a,b,par,deg,src)) cycle=1;
    push(a,b,imdtG);
}
if(outrng)
{
    printf("Out or range nodes detected");
    return 0;
}
if(same)
{
    printf("Xi's and Yi's must be pairwise distinct");
    return 0;
}
if(cycle)
{
    printf("Cycle detected in immmediate graph");
    // print(0);
    return 0;
}
```

Compressed Graph

- Here, whenever we find that parent of a node in disjoint set data structure is itself we append that node to search array.
- Then we form the compressed graph by traversing the graph and for each directed edge between two nodes in the orgnG, we add a directed edge between their representative elements in the compG

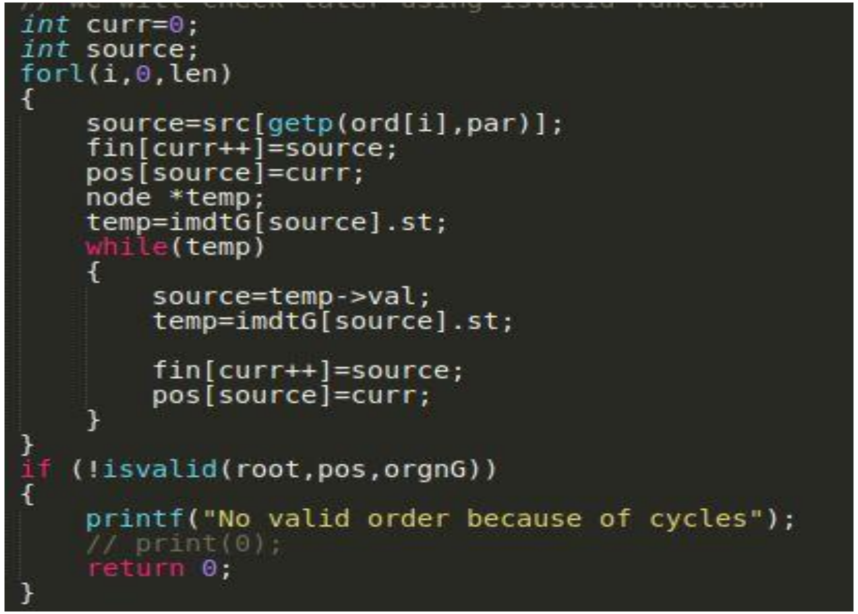




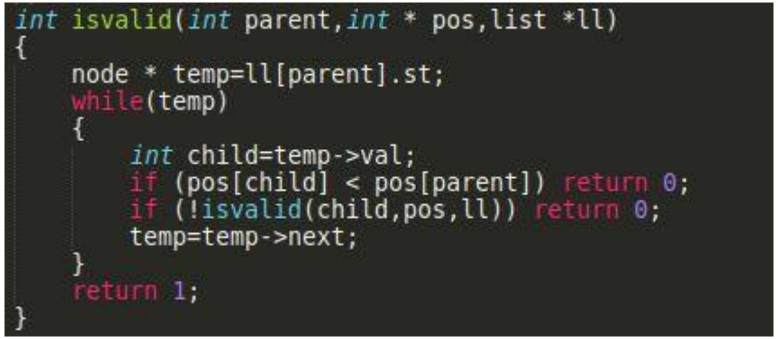
```
forl(i,1,n+1)
{
    node* temp=orgnG[i].st;
    if(getp(i,par)==i)
    {
        search[len++]=i;
    }


    // finding roots/representative elements
    // of each component and adding edges
    // between them to form compressed graph
    while(temp)
    {
        int v=i,u=temp->val;
        int vroot=getp(v,par);
        int uroot=getp(u,par);
        if(vroot!= uroot)
        {
            push(vroot,uroot,compG);
        }
        temp=temp->next;
    }
}
if(!topsort(len,search,used,ord,compG))
{
    printf("Cycle detected in compressed graph");
    // print(0);
    return 0;
}
```

Final schedule and Its validity

- We find a final schedule by decompressing components in the order they are present in the topsort of the compressed graph
- Such a schedule may still be invalid if there is a cyclic dependency such that there is directed edge is in one direction in the immediate graph and in the other direction in the original graph



```
// we will check later using isvalid function
int curr=0;
int source;
forl(i,0,len)
{
    source=src[getp(ord[i],par)];
    fin[curr++]=source;
    pos[source]=curr;
    node *temp;
    temp=imdtG[source].st;
    while(temp)
    {
        source=temp->val;
        temp=imdtG[source].st;

        fin[curr++]=source;
        pos[source]=curr;
    }
}
if (!isvalid(root,pos,orgnG))
{
    printf("No valid order because of cycles");
    // print(0);
    return 0;
}
```



```
int isvalid(int parent,int * pos,list *ll)
{
    node * temp=ll[parent].st;
    while(temp)
    {
        int child=temp->val;
        if (pos[child] < pos[parent]) return 0;
        if (!isvalid(child,pos,ll)) return 0;
        temp=temp->next;
    }
    return 1;
}
```

Time Complexity Analysis

Time Complexity Analysis

- As we have already noted that given the input we can deduce that we will get a rooted directed tree
- Also, since two immediate edges can’t have the same start and end nodes, so at max for a graph of N nodes we can have only N immediate edges
- Considering the above points we may have an O(n) input at worst

So, we moved forward with the adjacency list representation of graphs and not adjacency matrix since complexity for initializing the matrix for n nodes will itself be O(n^2)

Time Complexity Analysis

- In our algorithm, we compress immediate edges into a single node and then run topsort on the compressed graph
- We may do it naively i.e. traversing back to the root of each immediate tree for every vertex in the immediate graph and form the compressed graph using the roots as the vertices.
- Or, we may use union find disjoint set forests data structures to the compression.

In the naive way, for a ‘k’ length immediate tree of parent-child, we will have to do k + k-1 + k-2 ... 0 considering we want to find the root starting from the last node in the tree, second last in the tree … starting at the root itself.

This will take O(k^2) time

If we use union-find disjoint set, then both uniting sets and finding the representative element will take O(1) time. Hence we can compress a ‘k’ length immediate tree in O(k) time

Time Complexity Analysis

- Both topsort() and isvalid() functions perform depth first searches on graphs
- As we have already discussed during our theory classes depth first search on a rooted graph and topological sorting runs in O(n) time
- Finding a schedule involves decompressing components found from topological sort and iterating through the root to leaf of all immediate edges in a components. Since we already store the roots in src{ } array, this can also be done in O(n) time

Therefore, overall complexity: **O(n)**

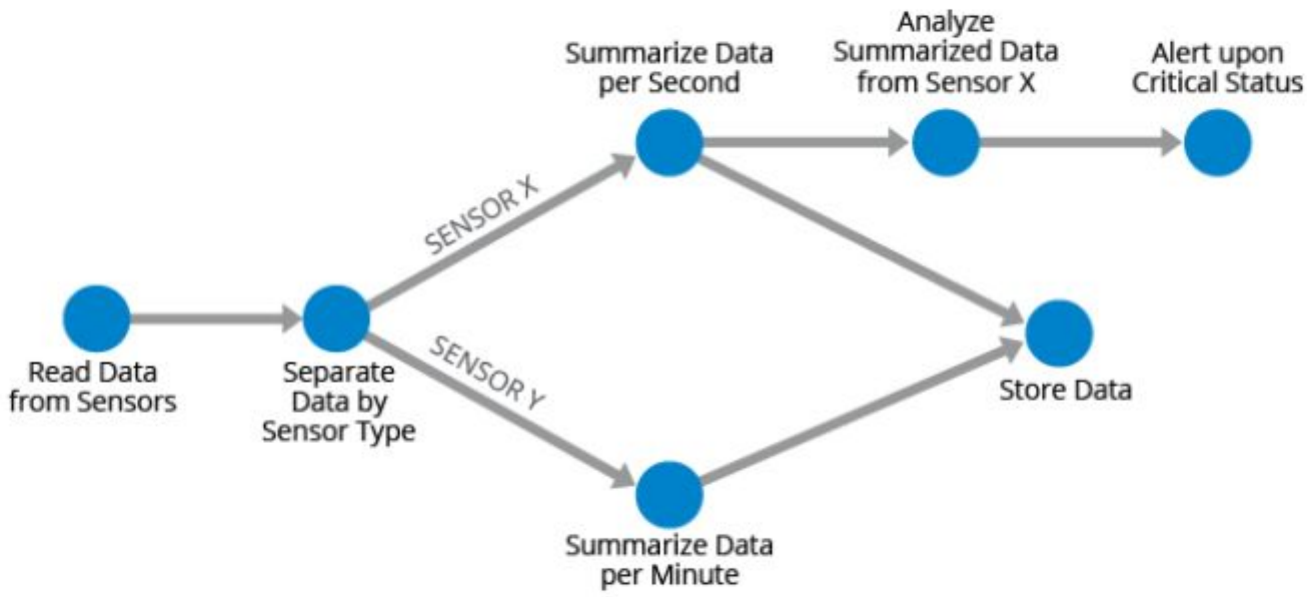
But, since there are n nodes in the original graph and we must read all of them, so we also get a lower bound of: **Ω(n)**

Combining the above, we get; **Θ(n)**

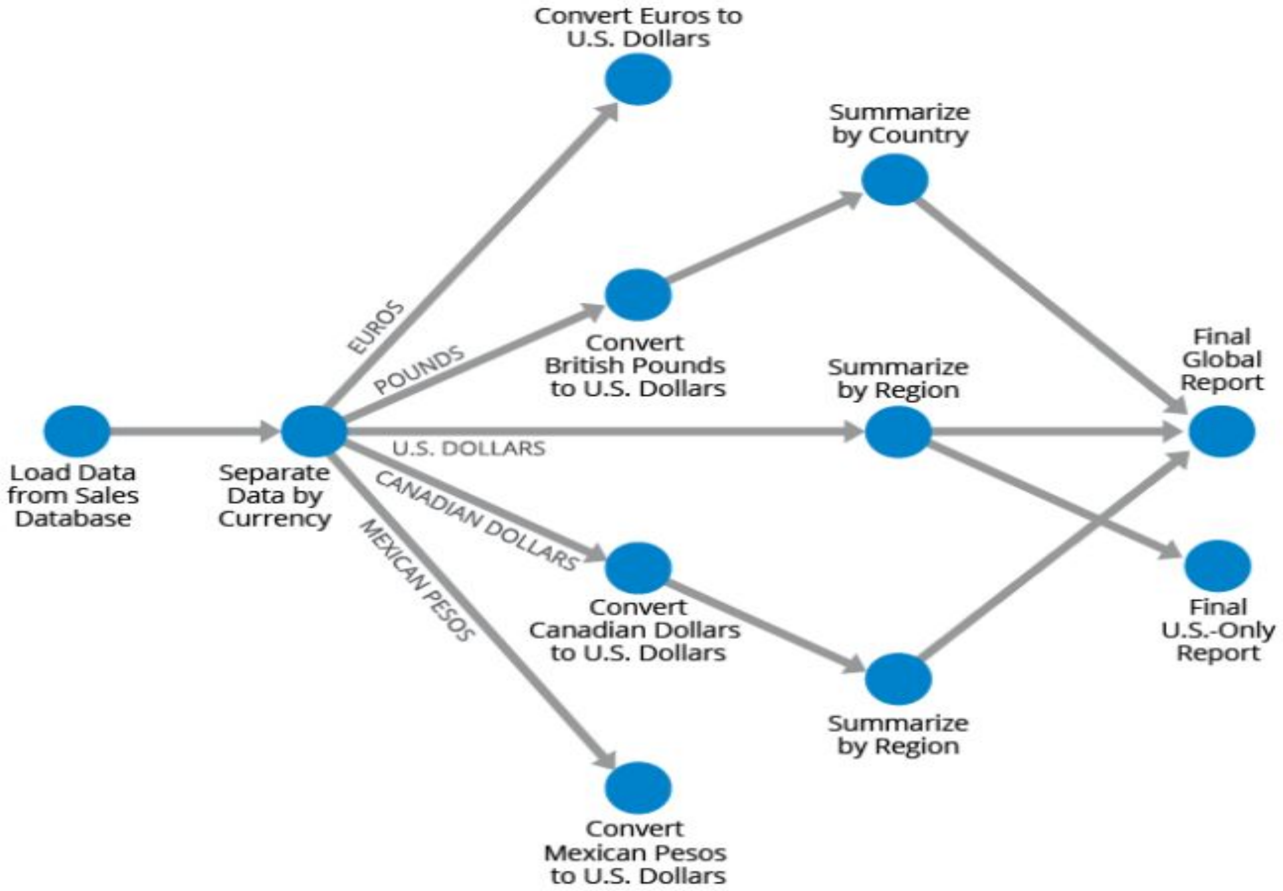
Hence, we have found an optimal algorithm for our given problem

Motivation

- Useful for representing many different types of flows, including **data processing flows** and one can clearly organize the various steps and the associated order for these jobs.
- **Sales transaction data** might be processed immediately to prepare it for making real-time recommendations to consumers.
- As part of the processing lifecycle, the data can go through many steps including **cleansing** (correcting data), **aggregation** (calculating summaries), **enrichment** (identifying relationships with other relevant data), and transformation (writing the data into a new format).
- **Industrial processes** which need precision timing and best efficiency, for example: Glass industry, Metallurgical extraction, Silicon Wafer industry etc.


Summarize Data per Second
Analyze Summarized Data from Sensor X
Alert upon Critical Status
SENSOR X
SENSOR Y
Read Data from Sensors
Separate Data by Sensor Type
Store Data
Summarize Data per Minute

A stream of sensor data represented as a directed acyclic graph.


Convert Euros to U.S. Dollars
Summarize by Country
EUROS
POUNDS
Convert British Pounds to U.S. Dollars
Summarize by Region
Final Global Report
Load Data from Sales Database
Separate Data by Currency
U.S. DOLLARS
CANADIAN DOLLARS
MEXICAN PESOS
Convert Canadian Dollars to U.S. Dollars
Final U.S.-Only Report
Summarize by Region
Convert Mexican Pesos to U.S. Dollars

Global sales data represented by the directed acyclic graph.

Thank You!