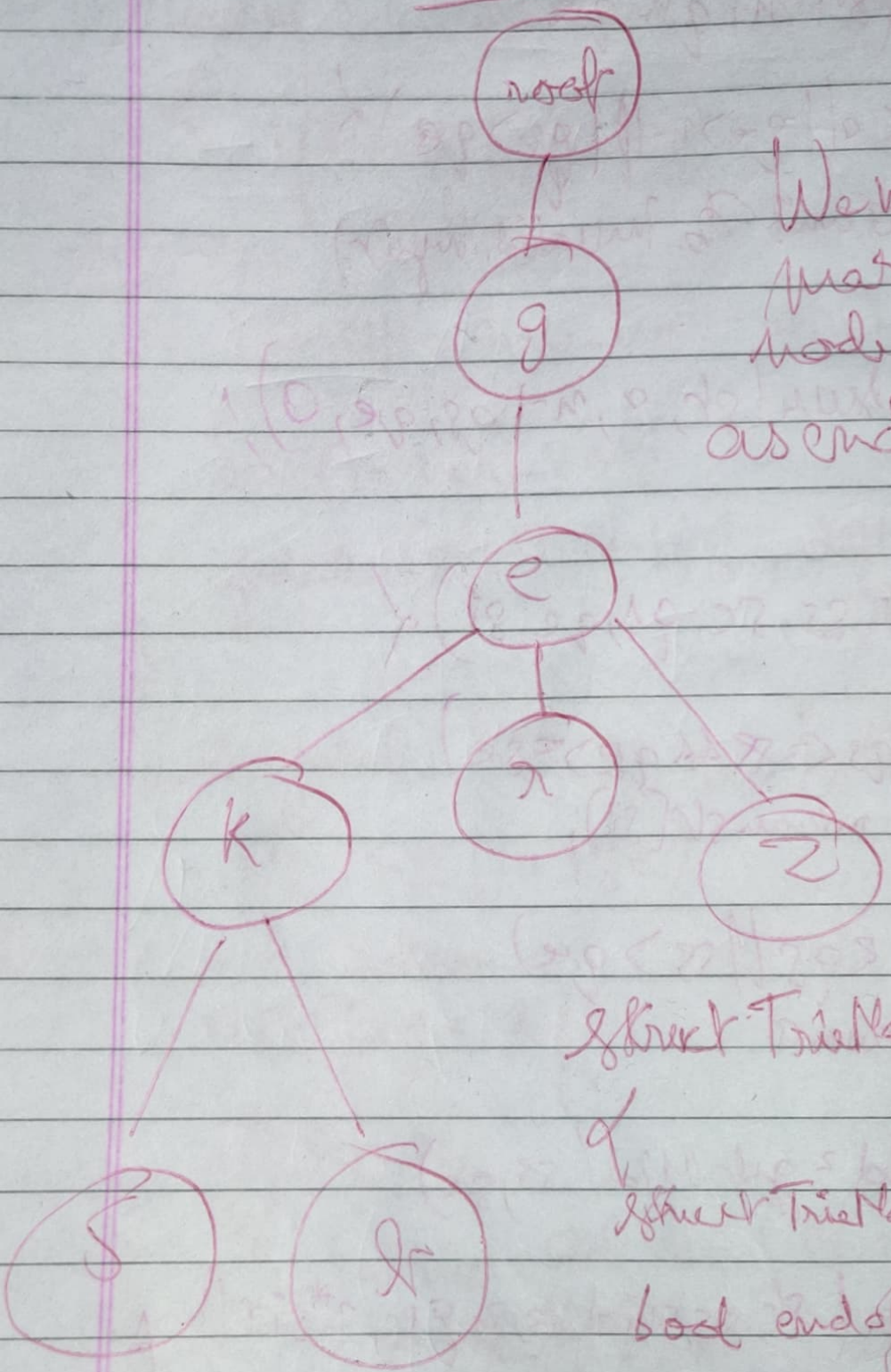
Trie



We need to mark the last node of every key as end of word node.

```
struct TrieNode
{
    struct TrieNode* children[ ];
    bool endOfWord;
};
```

1st Choice

Page No.
Date

Insert:

Every character of the input key is inserted as an individual Trie node.

The key character acts as an index into the array children.

If the input key is new or an extension of the existing, we need to construct non existing nodes of the key, and mark end of the word for the last node.

If the input key is a prefix of the existing key, we simply mark the last node of the key as end of word.

⇒ The key length determines Trie depth.

For search we only compare the characters and move down. if end of word is true for last node then the key exists in the trie.

Insert and search costs O(key_length)

Space requirements of Trie is O(ALPHABET_SIZE * key_length * N) where N is the number of keys in Trie.

Code: // functions are iterative instead of recursive for simplicity of course.

```
struct TrieNode {
    bool terminating = 0;
    map<char, TrieNode*> myp;
    TrieNode() { terminating = 0; }
};

// Initialize root pointer in main

void insert(TrieNode* root, string word) {
    TrieNode* curr = root;
    for(char c : word) {
        if(curr->myp.find(c) == curr->myp.end())
            curr->myp[c] = new TrieNode();
        curr = curr->myp[c];
    }
    curr->terminatingAt;
}
```

```
int search (Trie, Node*, string word) {
    TrieNode* curr = root;

    for (char c: word) {
        if (curr -> mp.find(c) == curr -> mp.end())
            return -1; // is not present.
        curr = curr -> mp[c];
    }

    return curr -> terminating; // freq of word.
}
```

★ Sieve of Eratosthenes ★

Code :-

```
Sieve of Eratosthenes (int n){
vector<bool> prime(n+1, true)
for (int p=2; p*p <= n; p++)
{
    if (prime[p]) {
    for(int i=p*p; i<=n; i+=p)
        prime[i] = false;

}
```

$O(n * \log(\log(n))$

1st Choice

Page No.
Date

Candy - Leetcode Hard Greedy

Initially assign 1 candy to everyone.
Do a two pass left to right and right to left
in left assign according to left neighbour,
in right assign according to right neighbor.

Since it is necessary to satisfy both.

In the third pass we can make
candies[i] to be max of left[i], right[i]
as any greater value will satisfy the
condition and max(left[i], right[i]) will
be minimum such value.

Syntax → lower_bound()
upper_bound()
binary_search()

lowerB_found (vec.begin(), vec.end(), key)

map. lower_bound (key)

returns iterator to element not less than key.

upper_bound (vec.begin(), vec.end, key)

map. upper_bound (key)

returns iterator to element greater than key.

binary_search (vec.begin(), vec.end()), key)

returns true or false.

Merge intervals

Given an array of intervals [st, end] merge all ~~~~ overlapping intervals.

Solution

Sort w.r.t start time.

```
currend = -1  currst = -1

for(i←0; i<n; i++)
    if( interval[i].st > currend){
        if(currend != -1)  currst, currend
            ans.pushback(~~~~)
        currend = interval[i].end
        currst = interval[i].st
    }
    else
        currend = max(currend, interval[i].end)
}
ans.pushback({currst, currend});
```

Video Stitching - LC

Given a set of video clips of sporting event that lasted time seconds.

Read problem statement from LC.

Approach 1 (try to push the max forward by connecting with connectable)

```
int min=0, max=0, total=0
while (max<time){
    for (i=0; i<n; i++){
        if(clips[i][0]<=min &&
                clips[i][1]>
                max)
            max=clips[i][1]
    }
    if(min==max)
        return -1
    min=max
    total++;
}
return total;
```

Approach 2 - Sort + DP

Sort w.r.t. start time.

for each clip:

relax dp[start+1] to dp[end]

Approach-3 : DP → (0, time)

dp[0] = 0;

For each(i)

for each clip

relax dp[i] if (start <= i <= end)

Min No. of taps to open to Water a Garden. - LC

Same as video switching.

Make interval from ranges

And use approach 1 and approach 2 acc. to constraints.

1st Choice

Page No.
Date

Union find

find with path compression recursive :-

```
int find (int i) {
    if (parent[i] != i)
        parent[i] = find(parent[i])
    return parent[i]
}

void union (int p, int q) {
    int x = find(p)
    int y = find(q)
    if (x != y) {
        if (size[x] <= size[y]) {
            parent[x] = y
            size[y] += size[x]
        } else {
            parent[y] = x; size[x] += size[y]
        }
    }
}
```

Kruskal's Algorithm MST

$O(E \log E)$

```
int find(int i){
    if(parent[i] != i)
        parent[i] = find(parent[i])
    return parent[i]
}
void Union(int i, int j){
    int x = find(i)
    int y = find(j)
    if(x != y){
        if(size[x] < size[y]){
            parent[x] = y
            size[y] += size[x]
        }
        else{
            parent[y] = x
            size[x] += size[y]
        }
    }
```

```
Void Kruskal (adj[], Edges) {
Edge result[V]; → Vector<pair<int,int>> (user defined datatype)
    sort the edges;
    initialise parent[i] = i and size[i] = 1.
    for(int i=0, e=0; i<edges.size() && e<V-1; i++){
        int src = edge[i].first
        int dest = edge[i].second
        int x = find(src)
        int y = find(dest)
        if(x!=y){
            result.pb(edge[i])
            e++
            Union(x,y)
        }
    }
```

Topological Sorting ($O(V+E)$)

Possible for only directed acyclic graph.

Algo: Visit unvisited vertices using dfs.

Add current vertex to vector

reverse the vector.

<u>Code</u>

```
vector(int) ~~topsort~~ ans;

void tps(){
    for(each vertex)
        if(vertex is unvisited) dfs(v);
    reverse(ans)
}
dfs(int s){
    visited[s] = true.

    for(v: adj[s])
        if(!visited[v])
            dfs v
    ans.pb(s);
}
```

Graph Coloring

Used to detect cycle in directed graphs.

Use normal dfs use color[u] == 0 → unvisited
color[u] == 1 → currently being visited (cycle)
color[v] == 2 → visited full not in cycle

Code

```
dfs(int u) {
    color[u] = 1
    for(v adj(u)) {
        if(color[v] == 1)
            cycle detected
        else if(color[v] == 0)
            dfs(v)
    }
    color[u] = 2
}
```

Kosaraju's algorithm O(V+E)

- Detect strongly connected components
  - path from a to b and b to a both exists

for directed acyclic graphs

Algorithm

Find topological order.

Reverse edges

Visits vertices in topological order

The vertices reachable from a vertex are strongly connected set

---

Code: main

```
while (n--)
    adj[u].pb(v)
    radj[v].pb[u]
}
```

```
vector<int> toporder;
void tps(){
    for(each v)
        if(!visited[v])
            dfs
    reverse(toporder)
}
void dfs(int u)
    visited[u] = true
    for(adj[u])
        if(!visited[v])
            dfs(v)
    toporder.pb(u)
}
```

```
vvi  Kosaraju(){
vvi  Strongly connected gfks → SCS
     dfs();
     rgdfs(visited)
     for(v: toforder){
        if(!visited[v]){
           vector<int> a;
           rdfs(v, a)
        }SCS.pb(a)
     }
     return SCS
}

void rdfs(int u, vector<int>& a){
     visited[u]=true
     a.pb(u)
     for(v: radj[u])
        if(!visited[v]){
           rdfs(v,a)
        }
}
```

☆ Time complexity of BFS and DFS

$O(V+E)$ for adj list

$O(V^2)$ for adj (matrix)

☆ Time complexity of dijkstra

$O(V+E \log V)$

Bipartite Graph

↳ Divide into two sets

BFS and DFS both can be used just maintain alternating order.

Floyd Warshall

3 loops

```
if(distr[i][k] != inf && distr[k][j] != inf)
    distr[i][j] = distr[i][k] + distr[k][j]
```

☆ K should be in outermost loop.

Bellman Ford

- Time complexity O(VE)

Works well with -ve edges

Shortest path algo

Algo detects -ve cycles in the graph

Algo:-

```
Initialise:
    memset(dist, inf, sizeof(dist))
    dist[src] = 0;

for(i=0; i<V-1; i++):
    for all edges = for(j=0; j<edges.size(); j++)
        if(dist[v] > dist[u] + weight(u→v))
            dist[v] = dist[u] + weight(u→v);

// checking for -ve cycle
for all edges
    if(dist[v] > dist[u] + weight(u→v))
        contains -ve cycle.
```

To detect position of -ve cycle In Bellman Ford

Keep a parent array

parent(n), parent[src] = -1

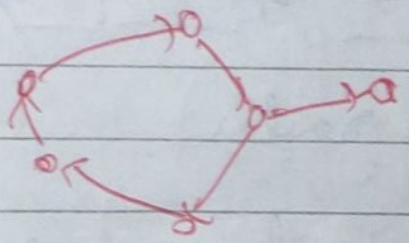
when updating dist

dist[v] > dist[u] + weight u→v

dis[v] = dist[u] + weight u→v

parent[v] = u.

Now when we detect cycle, there's a chance that the cycle has propagated



As there can be at max n-1 parents, backtrack parents n times, this will bring us inside the cycle.

Now print the cycle using similar parent backtracking.

Prim's MST $O((E+V)*\log(V))$

Similar to dijkstra, in dijkstra we are concerned with min cost path to each vertex individually. But here we want min cost to span whole graph together i.e. sum of all edges in MST

In dijkstra we take dist, here we take min edge cost to reach that node.

Key[v] = graph[u][v], Edge cost $u \to v$

Select min key vertex then work

☆ DP on a Graph ☆

Que: Game routes

Find no. of ways to reach dest from source

2 Approaches, dfs, organised bfs

Vimp sort order

★ DFS approach ★

```
void dfs(int u, int dest){
    dp[u] = (u==dest) ? 1:0;
    for(v: adj[u]){
        if(dp[v]==-1){
            dfs(v, dest);
        }
        dp[u] = (dp[u]+dp[v]);
    }
}
```

Organised BFS, can only be used for DAG

Find topological sort order,
relax edges in topological sort order.

```
count[0]=1
for(int i=0; i<topsort.size(); i++){
    s=topsort[i]
    for(v: adj[s])
        count[v] += count[s];
}
```

Dijkstra ~~DFS~~ in Undirected/DP
or directed
doesn't matter.

Ques Investigation

find ~~sub-no~~ number of minimum price routes. from source to dest

Use dijkstra with dp:

Code:

dist[i] = inf, numflights[i] = 0;
numflights[0] = 1, dist[0] = 0

pq.push(~~0,0~~) (src, dist);
while(!pq.empty)
auto u = pq.pop() → checking visited
if(u[0] > dist[u[1]]) continue
for(v: adj[u[1]])
if(dist[u[1]] + wt == dist[v])
numflights += numflights[u]
else if(dist[u[1]] + wt < dist[v])
update dist, numflight[v] = numflight[u]
pq.push!

Fast I/O

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

WILDCARD MATCHING

Recursion

```
int solve(string s, string p, int n, int m, int i, int j){
  if(i==n && j==m)
    return 1; // means strings have totally matched

  if(i==n){ // text has ended so pat should also end
            or have all '*'s
    for(int k=j; k<m; k++){
      if(p[k]!='*')
        return 0;
    }
    return 1;
  }
  if(j==m) // pat ended but text hasn't
    return 0;
```

```
if(p[j]==='?'||s[i]==p[j])
  // match this character and check next
  return solve(s,p,n,m,i+1,j+1);
if(p[j]=='*'){ // '*' means I can skip any number of characters till end of str
  int res=0;
  for(int k=i;k<=n;k++){
    res=solve(s,p,n,m,k,j+1);
    if(res==1)
      return 1
  }
  return 0
}
return 0; // characters don't match.
}
```