Week 1

`lineup` - wines, find the correct combination

> **bruteforce, all permutations**

- `next_permutation(vec.begin(), vec.end())`

`outoforder` - sort and bruteforce

> **bruteforce, sorting**

`theanswer` - find 3 numbers in a list that sum up to 42

> **binary search on a sorted array**

- `sort(vec.begin(), vec.end())`
- `binary_search(vec.begin(), vec.end(), query_value)`

Week 2

`bordeaux`

`microbes` - Query-Update problem,

> **divide-and-Conquer**

- the helpful thing here was to compupte sums over chunks
- if update was in some chunk add/subtract the value of the chunk
- if query was over multiple chunks, the whole chunks neednt be checked, only their sums

`theanswercount` - subset checking, generating power-set

> **subset, lower-bound/upper-bound on a sorted array**

- split set into 2 subsets
- generate its power set using mask (compute sums)
- check for the remaining number in the other sorted set using lb/ub
- this could be done with binary_search, but there may be duplicit numbers that are correct, say 5 times the number you are looking for in a series

```
// generate all subsets
uint pow_set_size = pow(2, A1.size());
for (uint cnt = 0; cnt < pow_set_size; cnt++){

    ll temp_sum = 0;
    for (uint i=0; i < A1.size(); i++){
```

```
            if (cnt & (1 << i)){
                temp_sum += A1[i];
            }
        }
        S1.push_back(temp_sum);
    }
```

```
for(uint i=0; i<S1.size(); i++){
            ll candidate = S1[i];
            // S2 is sorted
            auto lb = lower_bound(S2.begin(), S2.end(), (x - candidate));
            auto ub = upper_bound(S2.begin(), S2.end(), (x - candidate));
            total_count += ub-lb;
        }
```

Week 3

begging - check for optimal subproblems

> dynamic programming

```
class dpClass {        // The class
  public:              // Access specifier
    int val{0};         // Attribute (int variable)
    int prev;
};
//...
vector<dpClass> dp(n, dpClass{});
```

- check for sufficient sub-solutions
- optimal choice of waiting time to beg for wine

cashier - Coin-change problem

> dynamic programming

cake - Tiling problem

> 2-D prefix sums, Dynamic Programming

```
vector<vector<int>> p_sums(vector<vector<int>> tray){
    vector<vector<int>> sums(tray.size(), vector<int>(tray[0].size(), 0));
    for (uint r = 0; r < tray.size(); r++){
        for (uint c = 0; c < tray[0].size(); c++){
            if (r == 0 && c == 0){
```

```
                sums[r][c] = tray[r][c];
            }
            else if (r == 0){
                sums[0][c] = sums[0][c-1] + tray[0][c];
            }
            else if (c == 0){
                sums[r][0] = sums[r-1][0] + tray[r][0];
            }
            else{
                sums[r][c] = sums[r-1][c] + sums[r][c-1] - sums[r-1][c-1] +
tray[r][c];
            }
        }
    }
    return sums;
}
//...
for (int r=0; r <= h-h_cake; r++){
        for (int c=0; c <= w-w_cake; c++){
            temp_sum = psv[r+h_cake-1][c+w_cake-1];
            if (r == 0 && c == 0){

            }
            else if (r == 0){
                // sums[0][c] = sums[0][c-1] + tray[0][c];
                temp_sum -= psv[r+h_cake-1][c-1];
            }
            else if (c == 0){
                // sums[r][0] = sums[r-1][0] + tray[r][0];
                temp_sum -= psv[r-1][c+w_cake-1];
            }
            else{
                // sums[r][c] = sums[r-1][c] + sums[r][c-1] - sums[r-1][c-
1] + tray[r][c];
                temp_sum += -psv[r+h_cake-1][c-1] - psv[r-1][c+w_cake-1] +
psv[r-1][c-1];
            }
            if (temp_sum == 0){
                cout << r+1 <<" "<< c+1 <<" "<< r+h_cake <<" "<< c+w_cake
<< "\n";
                return 0;
            }
        }
    }
```

warming - Longest Increasing Subsequence

> LIS - Longest Increasing Subsequence, DP - Dynamic Programming

```
int lis = 0;
for (int i=0; i<n; i++){
```

```
        dp[i] = 1;
        for (int j=0; j<i; j++){
            if (temperatures[j] < temperatures[i]){
                dp[i] = max(dp[i], dp[j]+1);
            }
        }
        lis = max(lis, dp[i]);
    }
    cout << lis << "\n";
```

warming2 - LIS with binary search

> LIS, Binary Seacrh

```
int minus_int_inf = numeric_limits<int>::min();
g_vals.push_back(minus_int_inf);

for (int i=0; i < n; i++){
    int g_idx = lower_bound(g_vals.begin(), g_vals.end(), A[i]) -
g_vals.begin();
    f_vals.push_back(1 + g_idx-1);  // f_i = 1 + max{l|g_i[l] < A[i]}

    // now the g_vals(i+1) = min{g[f[i]], A[i]}
    if (f_vals[i] > (int)(g_vals.size()-1)){
        // if g_vals too small, append
        g_vals.push_back(A[i]);
    }
    else{
        // else just update the number already there
        g_vals[f_vals[i]] = min(g_vals[f_vals[i]], A[i]);
    }
}
```

Week 4

sumup and equations

> Tree parsing, Custom Nodes

```
class Node {
    public:
        ll val{-1};
        bool X{false};
        vector<Node *> children{};
        Node * parent{nullptr};
};
```

```
int deeper(Node *node, ll *i, const string &inp){

    while((*i) < (ll) inp.size()){

        if (inp[*i] == ')'){
            (*i)++;
            break;
        }
        if (inp[*i] == '(' || inp[*i] == ','){
            (*i)++;
        }

        if (isdigit(inp[*i])){
            Node* new_node = new Node{whole_num(i, inp), false, {}, node};
            node->children.push_back(new_node);
        }
        else if (inp[*i] == 'X')
        {
            Node* new_node = new Node{'X', true, {}, node};
            node->children.push_back(new_node);
            (*i)++;
        }
        else if (inp[*i] == '+')
        {
            // Node new_node;
            Node* new_node = new Node{'+', false, {}, node};
            node->children.push_back(new_node);
            (*i)++;
            deeper(new_node, i, inp);

        }
        else if (inp[*i] == '*')
        {
            Node* new_node = new Node{'*', false, {}, node};
            node->children.push_back(new_node);
            (*i)++;
            deeper(new_node, i, inp);
        }
    }
    return 0;
}
```

- **Note that, creation with `new` needs to be deleted afterwards:**

```
void delete_tree(Node * root){

    if (root->children.empty()){
            return;
        }
    else{

        for (uint i = 0; i < root->children.size(); i++){
```

```
            delete_tree(root->children[i]);
            delete root->children[i];
        }
    }
}
```

- evaluation of the tree:

```
ll eval_tree(Node * root, ll x){
    ll total;

    if (root->children.empty()){
        if (root->X){
            return x;
        }
        else{
            return root->val;
        }

    }
    else
    {
        if(root->val == '*'){
            total = 1;
            for (uint i = 0; i < root->children.size(); i++){
                total *= eval_tree(root->children[i], x);
            }
        }
        else{
            total = 0;
            for (uint i = 0; i < root->children.size(); i++){
                total += eval_tree(root->children[i], x);
            }
        }
    }
    return total;
}
```

covering

```
int get_whole_num(ll *i, const string& inp){
    char num_chars[4];
    int cnt = 0;
    while (true){
        if (inp[*i] == ',' || inp[*i] == ')') break;
        num_chars[cnt] = inp[*i];
        cnt++;
        (*i)++;  // moves pointer by one
    }
```

```
    num_chars[cnt] = '\0';
    // cout << "`whole_num`: \n" << stoi(num_chars) << "\n";
    return stoi(num_chars);
}

int main(){
    // bool vb = false;
    vector<string> trees;
    ios::sync_with_stdio(false);
    string line;
    while(getline(cin, line)){
        trees.push_back(line);
    }
    vector<set<int>> s((int) trees.size(), set<int>{});

    int max_num = 0;
    int num;

    /* -- PARSE THE INPUT INTO SETS -- */

    for (uint i = 0; i < trees.size(); i++){ // for tree in trees

        for (ll j = 0; j < (ll) trees[i].size(); j++){ // for char in tree

            if (isdigit(trees[i][j])){
                num = get_whole_num(&j, trees[i]);
                s[i].insert(num);
                if (num > max_num){
                    max_num = num;
                }
            }
        }
    }

    vector<int> markers(max_num+1, -1);
    vector<ll> masks(s.size(), 0);
    uint max_arity = 0;
    uint arity_cnt = 0;
    ll mask_len = 8 * sizeof(ll);

    // generate masks
    for (uint i = 0; i < s.size(); i++){
        for (auto const &k : s[i])
        {
            masks[i] = masks[i] | 1 << (k % mask_len);
        }
    }

    // choose examined candidate superset
    for (uint i = 0; i < s.size(); i++)
    {
        arity_cnt = 0;
        for (auto const &k : s[i])
        {
```

```
                markers[k] = i; // mark itself into the marks
            }

            // cycle through candidate subsets
            for (uint j = 0; j < s.size(); j++)
            {
                if (j == i) continue;  // dont examine self

                if ((~masks[i] & masks[j]) != 0)
                {
                    continue;
                }

                else
                {
                    bool is_subset = true;
                    for (auto const &k : s[j]){
                        if (markers[k] != (int) i)
                        {
                            is_subset = false;
                            break;
                        }
                    }
                    if(is_subset) arity_cnt++;

                }
            }
            if (arity_cnt > max_arity) max_arity = arity_cnt;
        }
        cout << max_arity << "\n";

        return 0;
    }
```

Week 5 - Graphs, DFS, BFS

utility stuff

loading stuff

```
// read the edges and save them into Adjacency List
vector<vector<pair<ll, ll>>> adj(n_nodes);  // initialized to zeros in the
first dim
for (int i = 0; i < b_lines; i++){
    ll first, second, weight;
    cin >> first >> second >> weight;
    first--; second--;
    adj[first].push_back({second, weight});
    adj[second].push_back({first, weight});
}
```

printing stuff

```
void show_adj_table(const vector<vector<int>> &adj){
    cout << "--------------\n| ADJ. TABLE |\n--------------\n";
    for (uint i = 0; i < adj.size(); i++){
        cout << i+1 << " | ";
        for (uint j = 0; j < adj[i].size(); j++){
            cout << adj[i][j]+1 << " ";

        }
        cout << "\n";
    }
}
```

bars

> DFS - Depth First Search

recurrent DFS

```
void dfs(int v, vector<bool> *visited, vector<vector<int>> *adj, int
max_depth, int curr_depth){
        (*visited)[v] = true;
        vector<bool> temp_visited = *visited;

        for (int u: (*adj)[v])
        {
            if (curr_depth < max_depth)
            {
                dfs(u, visited, adj, max_depth, curr_depth+1);
            }
        }
    }
```

stacked BFS

```
void bfs(int start, vector<bool> &visited, const vector<vector<int>> &adj,
int max_depth){
    queue<pair<int,int>> Q;

    Q.push({start, 0});
    visited[start] = true;
    // int curr_depth = 0;

    while (!Q.empty())
    {
```

```
        auto v = Q.front();
        if (v.second >= max_depth) break;
        Q.pop();

        for (int u: (adj)[v.first]){
            if (!visited[u]){
                Q.push({u, v.second+1});
                visited[u] = true;
            }
        }
    }
}
```

fares - Shortest Path, positive weights

> Dijkstra's algorithm

```
void dijkstra(ll start, vector<ll> & dist, const vector<vector<pair<ll,
ll>>> & adj){
    dist[start] = 0;
    priority_queue<pair<ll,ll>, vector<pair<ll,ll>>, greater<pair<ll,ll>>>
pq;
    pq.push({0, start});  // <distance, vertex>
    while(!pq.empty())
    {
        auto front = pq.top();
        pq.pop();
        ll d = front.first;  // distance of the vertex
        ll v = front.second;  // ID of the vertex
        if (d > dist[v]) continue;
        for (auto p : adj[v])  // for child with <target, weight>
        {
            ll u = p.first;
            ll w = p.second;
            if (dist[v] + w < dist[u])
            {
                dist[u] = dist[v] + w;
                pq.push({dist[u], u});
            }
        }
    }
} // compro lecture_5, slide 38
```

inequalities

> Toposort, topsort, topological sort

- more advanced topological sort given further below

Week 6

bridges

> DFS for finding bridges

```
int dfs(int u, int depth, int dfsRoot, int rootChildren, const
vector<vector<int>> &adj, vector<int> &dfs_min, vector<int> &dfs_num,
vector<int> &dfs_parent){
    dfs_min[u] = dfs_num[u] = depth;
    depth++;
    int n_bridges = 0;
    for (auto v: adj[u]){
        if (dfs_num[v] == -1)  // tree edge
        {
            dfs_parent[v] = u;

            if (u==dfsRoot) {
                rootChildren++;
            }

            n_bridges += dfs(v, depth, dfsRoot, rootChildren, adj, dfs_min,
dfs_num, dfs_parent);

            if (dfs_num[u] <= dfs_min[v] && u != dfsRoot){
                // is AP
            }
            if (dfs_num[u] < dfs_min[v]){
                // is a bridge
                // cout << "ISABRDIGE\n";
                n_bridges++;
            }
            dfs_min[u] = min(dfs_min[u], dfs_min[v]);
        }
        else if (v != dfs_parent[u]){  // Back Edge
            dfs_min[u] = min(dfs_min[u], dfs_num[v]);
        }
    }
    return n_bridges;
} // comprog22 lecture 6, slide 18 and 19a
```

caves1 - Longest path - only linear for DAG

> DAG - Directed Acyclic Graph, toposort, topsort, topological sort

- compute toposort

```
bool compute_toposort(int start, vector<bool> &visited, vector<bool>
&local_visited,
```

```
          const vector<vector<int>> &adj, vector<int> &ts){

    /* toposort might need to be reversed */

    local_visited[start] = true;

    for(int u: adj[start])
    {
        if(local_visited[u]){
        return false;
        }
        if (!visited[u]){
            if(!compute_toposort(u, visited, local_visited, adj, ts))
return false;
        }
    }

    local_visited[start] = false;
    if(!visited[start]){
        ts.push_back(start);
    }
    visited[start] = true;

    return true;
}
```

- and then finding the longest path (aka the max of gold):

```
vector<ll> max_gold_vec = gold;
for (auto u: ts){
    for (auto node: adj[u]){
        ll maybe_gold = gold[node];
        // print_vector_ll(max_gold_vec, "max_gold_vec");
        max_gold_vec[node] = max(max_gold_vec[node], max_gold_vec[u] +
maybe_gold);
    }
}
```

'caves2' - turn DCG to DAG

> SCC - Strongly Connected Component, Turn DCG to DAG, graph condensation

- find SCCs then condense the SCCs to single nodes, ergo creating a DAG

```
int dfs_counter = 0;
stack<int> S;
vector<int> ts;
deque<deque<int>> SCCs;
```

```
void scc(int u, vector<int> &dfs_num, vector<int> &dfs_min, vector<bool>
&on_stack, vector<vector<int>> &adj){
    dfs_min[u] = dfs_num[u] = dfs_counter;
    dfs_counter++;
    S.push(u);
    on_stack[u] = true;

    for (auto v: adj[u])
    {
        if (dfs_num[v] == UNVISITED)
        {
            scc(v, dfs_num, dfs_min, on_stack, adj);
            dfs_min[u] = min(dfs_min[u], dfs_min[v]);
        }
        else if (on_stack[v])  // only on_stack can use back edge
        {
            dfs_min[u] = min(dfs_min[u], dfs_num[v]);
        }
    }

    if (dfs_min[u] == dfs_num[u])  // output result
    {
        // cout << "    SCC: ";
        int v = -1;
        deque<int> component;
        do  // output SCC starting in u
        {
            v = S.top();
            S.pop();
            on_stack[v] = false;
            // cout << v + 1<< " ";
            ts.push_back(v);
            component.push_front(v);
        } while (v != u);

        SCCs.push_front(component);
        // cout << "\n";
    }
}
```

driving - Euler paths

> Euler paths

```
bool first_necessary_condition(int &n, const vector<vector<int>> &adj,
const vector<int> &indegree){
    /*
    i) the first vertex has out_degree == 1 + indegree
    and
    ii) the last vertex has in_degree == 1 + outdegree
```

```
    iii) all other vertices: in_degree == out_degree
    */
    int temp_n = n;
    for (int i = 0; i < n; i++){


        if(((int) adj[i].size() == 0) && (indegree[i] == 0)){
            // cout << "i: " << i << " | (int) adj[i].size() " << (int)
adj[i].size() << " | indegree[i] : " << indegree[i] << "\n";
            // cout << "temp_n--\n";
            temp_n--;
            continue;
        }

        // i)
        if (i == 0){
            if ((int) adj[i].size() != indegree[i] + 1)
            {
                if(vb){
                    cout << "i: " << i << "\n";
                    cout << "1 false\n";
                }
                return false;
            }
        }
        // ii)
        else if (i == n-1){
            if (indegree[i] != 1 + (int) adj[i].size())
            {
                if(vb){
                    cout << "i: " << i << "\n";
                    cout << "2 false\n";
                }

                return false;
            }
        }
        else{
        // iii)
            if (indegree[i] != (int) adj[i].size())
            {
                if(vb){
                    cout << "i: " << i << "\n";
                    cout << "3 false\n";
                }

                return false;
            }
        }
    }
    n = temp_n;
    return true;
}
```

```
void flood_dfs(int v, int &visited_cnt, const vector<vector<int>> &adj,
vector<bool> &visited){
    visited[v] = true;
    visited_cnt++;
    for (int u : adj[v])
    {
        if (!visited[u]){
            flood_dfs(u, visited_cnt, adj, visited);
        }
    }
}
```

- main:

```
int main(){
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n);
    vector<bool> visited(n, false);
    vector<int> indegree(n, 0);
    int visited_cnt = 0;
    // vector<int> has_path(n, 0);

    // read the edges and save them into Adjacency List
    for (int i = 0; i < m; i++){
        int first, second;
        cin >> first >> second;
        first--; second--;
        adj[first].push_back(second);
        indegree[second]++;

        // one directional graph
        // adj[second].push_back(first);
    }
    if (vb){
        show_adj_table(adj);
        print_vector(indegree, "Indegree");
    }
    // int temp_n = n;
    if (first_necessary_condition(n, adj, indegree)){
        flood_dfs(0, visited_cnt, adj, visited);
        // cout << "visited_cnt: " << visited_cnt << " n: " << n << "\n";


        if (visited_cnt == n){
            cout << "possible\n";
            return 0;
        }
        else{
            if (vb){
```

```
                cout << "Because visited_cnt != n:\n";
                cout << "n: " << n << "\n";
            }
            cout << "impossible\n";
            return 0;
        }

    }
    else cout << "impossible\n";

    return 0;
}
```

General printing stuff

```
void print(const vector<bool> &vec, const string &txt){
    cout << "vec: " << txt << "\n";
    for (uint i = 0; i < vec.size(); i++){
        cout << i+1 << " | " << vec[i] << "\n";
    }
    cout << "\n";
}

void print(const vector<int> &vec, const string &txt){
    cout << "vec: " << txt << "\n";
    for (uint i = 0; i < vec.size(); i++){
        cout << i+1 << " | " << vec[i] << "\n";
    }
    cout << "\n";
}

void print(const vector<ll> &vec, const string &txt){
    cout << "vec: " << txt << "\n";
    for (uint i = 0; i < vec.size(); i++){
        cout << i+1 << " | " << vec[i] << "\n";
    }
    cout << "\n";
}
```