# Revision

## COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

# Table of Contents

# Example problem: Crowd Surfing



- **Problem statement** You want to crowd surf to the southeast corner (+) of an $m \times n$ rectangle. Every other cell will either move you south (v), move you east (>) or drop you (*). No cell will move you outside the grid.
  A cell is said to be *good* if the crowd will move you from this cell to the southeast corner, and *bad* otherwise. How many bad cells are there in a given grid?

# Example problem: Crowd Surfing



- **Input** First line, $m$ $n$, $1 \leq m, n \leq 1,000$. Following this, $m$ lines, the $i$th of which is a string $s_i$ of length $n$ representing the cells in row $i$.
- **Output** A single integer, the number of bad squares.
- **Source** AIO 2008

# Example problem: Crowd Surfing



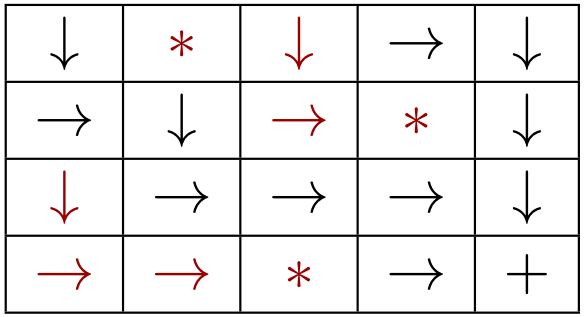
- **Example Input**

```
4 5
v*v>v
>v>*v
v>>>v
>>*>+
```

- **Example Output**

```
8
```

- **Explanation** The input represents this rectangle:



in which the bad cells are marked in red.

- Naïve approach: for every cell, follow the path starting here
- If it fails to take us to the southeast corner, increment the answer
- Clearly correct, but for each of $O(mn)$ cells the required path could have up to $m + n$ steps. Since $mn(m + n)$ could be up to $2 \times 10^9$, this is too slow

**Observation**

Suppose one step from cell $(i_1, j_1)$ takes you to $(i_2, j_2)$. Then $(i_1, j_1)$ is bad if and only if $(i_2, j_2)$ is bad

- Overlapping subproblems! DP
- There are other ways to use this observation

# Example problem: Crowd Surfing



- Natural choice of state is a cell $(i, j)$
- Recurrence is

$$\text{bad}(i,j) = \begin{cases} T & \text{if } s_{i,j} \text{ is } * \\ \text{bad}(i+1,j) & \text{if } s_{i,j} \text{ is } \texttt{v} \\ \text{bad}(i,j+1) & \text{if } s_{i,j} \text{ is } \texttt{>} \end{cases}.$$

- Base case is $\text{bad}(m-1, n-1) = F$

# Example problem: Crowd Surfing



- To ensure that the answers for $(i+1, j)$ and $(i, j+1)$ have already been computed by the time we get to $(i, j)$, we can solve bottom up in order of decreasing $i$ and decreasing $j$
- When a cell is marked as bad, increment the answer
- Each of $O(mn)$ cells is now answered in constant time
- Question guarantees that we never refer to a cell off the end of the array, so no bounds checking required

# Example problem: Crowd Surfing



```
#include <iostream>
using namespace std;
const int N = 1010;
char s[N][N];
bool bad[N][N];

int main() {
  int m, n;
  cin >> m >> n;
  for (int i = 0; i < m; i++)
    cin >> s[i];

  int ans = 0;
  for (int i = m-1; i >= 0; i--)
    for (int j = n-1; j >= 0; j--) {
      if (s[i][j] == '+')
        bad[i][j] = false;
      else if (s[i][j] == '*')
        bad[i][j] = true;
      else if (s[i][j] == 'v')
        bad[i][j] = bad[i+1][j];
      else if (s[i][j] == '>')
        bad[i][j] = bad[i][j+1];
      ans += bad[i][j];
    }
  cout << ans << '\n';
}
```

# Example problem: LIS



- **Problem statement** Given $n$ ($1 \leq n \leq 3,000$) integers in a sequence $S$, what is the longest subsequence of $S$ that is strictly increasing? If there are multiple solutions, print any one of them.
- **Example** For the sequence $[1, 9, 5, 8, 7]$, the longest increasing subsequence has length 3. There are multiple solutions, such as $[1, 5, 7]$.

# Example problem: LIS



- On this sort of sequence, the only way to define our state is to use a position in the sequence.
- We can ask the question, "What's the longest increasing subsequence that we can obtain, finishing at the $i$th element of the sequence?"
- The recurrence is also straightforward: we can try every previous element in our sequence as the last element of our subsequence to be included, and then take the best one.
- This leads to an $O(n^2)$ time, $O(n)$ space algorithm.

# Example problem: LIS



- We aren't done yet!
- We need to find not only the length of a longest increasing subsequence, but we need to find the subsequence itself.
- How do we recover an actual answer from our dynamic programming algorithm?
- Each state represents an increasing subsequence, which we constructed by extending another one, also represented by a state. Then all we need to do is store the index of the state we came from.
- Then to build an optimal solution, we can just backtrack from some terminal state through the optimal move we've stored until we reach our initial state.

# Example problem: LIS



- **Implementation (DP, $O(n^2)$)**

```
int best = 0;
for (int i = 1; i <= n; i++) {
  dp[i] = 1;
  for (int j = 1; j < i; j++) {
    // try j as the penultimate index
    if (s[j] < s[i] && dp[j] + 1 > dp[i]) {
      dp[i] = dp[j] + 1;
      from[i] = j;
    }
    // update answer
    if (dp[best] < dp[i])
      best = i;
  }
}
```

- **Implementation (solution building)**

```
vector<int> ans;
int u = best;
while (from[u]) {
  ans.push_back(s[u].second);
  u = from[u];
}
ans.push_back(s[u].second);
reverse(ans.begin(), ans.end());
```

# Example problem: LIS





This can be improved to $O(n \log n)$.

- As $i$ iterates through the sequence, maintain a sequence $M[j]$, the index of the minimum terminal value of an increasing subsequence of length $j$ among the first $i$ terms of $S$.
- It can be seen that the elements $S[M[j]]$ always form an increasing subsequence.
- To process a new element $S[i+1]$, we augment the longest of these subsequences whose terminal value is less than $S[i+1]$.
  That is, we find the largest $j$ where $S[M[j]] < S[i+1]$.
  This gives us the longest subsequence so far, so we extend that by updating $M[j+1]$.
- Rather than trying all possibilities in $O(n)$, we can perform a binary search on the $S[M[j]]$.

# Example problem: Counting Subsequences



- **Problem statement** Given a permutation of the first $n$ positive integers, find the number of increasing subsequences with $\ell + 1$ elements.
- **Input** First line, $n$ $\ell$, $1 \leq n \leq 100,000$, $0 \leq \ell \leq 10$. Second line, $n$ distinct integers $a_0, \ldots, a_{n-1}$, where $1 \leq a_i \leq n$.
- **Output** A single integer, the number of increasing subsequences of length $\ell + 1$. It is guaranteed that the answer is not greater than $8 \times 10^{18}$.
- **Source** Codeforces Testing Round #12.

# Example problem: Counting Subsequences



- **Example Input**

```
5 2
1 2 3 5 4
```

- **Example Output**

```
7
```

- **Explanation** There are seven increasing subsequences of length three: $\langle 1,2,3\rangle$, $\langle 1,2,5\rangle$, $\langle 1,2,4\rangle$, $\langle 1,3,5\rangle$, $\langle 1,3,4\rangle$, $\langle 2,3,5\rangle$ and $\langle 2,3,4\rangle$.

# Example problem: Counting Subsequences



- Obvious similarities between this and the previous problem
- How many increasing subsequences of each length end at index $i$ of the sequence?
- Look to extend subsequences with one fewer term again!
- But you can't extend from all earlier finishing points $k < i$
- Only those which also have $a_k < a_i$

# Example problem: Counting Subsequences



- Subproblem is $dp(i, j)$ the number of increasing subsequences of length $j + 1$ ending at index $i$
- Recurrence is

$$dp(i, j) = \sum_{k<i : a_k < a_i} dp(k, j - 1).$$

- Base case is $dp(i, 0) = 1$ for all $i$.
- Answer is recovered by $\sum_{i=0}^{n-1} dp(i, \ell)$.

- There are $O(n\ell)$ subproblems, each solved in $O(n)$ time, so the time complexity is $O(n^2\ell)$ which is too slow.
- Unclear how we could reduce the state space, but how about the recurrence?
- We wanted the sum over $k < i$ (handled by the order of processing) with $a_k$ also less than $a_i$ - this is a range query!

- Create range trees for $j = 0..\ell$
- In range tree $j$, a leaf corresponding to the value $a_i$ will store $dp[i][j]$, the number of increasing subsequences of length $j+1$ ending at index $i$.
- We can now facilitate

$$\sum_{k<i:a_k<a_i} dp(k, j-1)$$

  by querying the sum over the range $[1, a_i)$ in range tree $j-1$.
- The recurrence now takes $O(\log n)$, so the overall time complexity is $O(n\ell \log n)$.

# Example problem: Counting Subsequences





```cpp
#include <iostream>
using namespace std;

const int N = 100100;
const int L = 15;
long long dp[N][L];
long long tree[3*N][L];

// usual 'point update, range query' range tree
// there are k+1 many of these range trees, indexed by j
void update(int j, int a, long long v, int i = 1, int start = 0, int end = N) {
  if (end - start == 1) {
    tree[i][j] = v;
    return;
  }
  int mid = (start + end) / 2;
  if (a < mid) update(j, a, v, i * 2, start, mid);
  else update(j, a, v, i * 2 + 1, mid, end);
  tree[i][j] = tree[i*2][j] + tree[i*2+1][j];
}

long long query(int j, int a, int b, int i = 1, int start = 0, int end = N) {
  if (start == a && end == b) return tree[i][j];
  int mid = (start + end) / 2;
  long long answer = 0;
  if (a < mid) answer += query(j, a, min(b, mid), i * 2, start, mid);
  if (b > mid) answer += query(j, max(a, mid), b, i * 2 + 1, mid, end);
  return answer;
}
```

# Example problem: Counting Subsequences



```
int main (void) {
  int n, l;
  cin >> n >> l;
  long long ans = 0;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    dp[i][0] = 1;
    for (int j = 0; j < l; j++) {
      update(j, x, dp[i][j]);
      dp[i][j+1] = query(j, 1, x);
    }
    ans += dp[i][l];
  }
  cout << ans << '\n';
}
```

# Table of Contents

# DP on a Tree

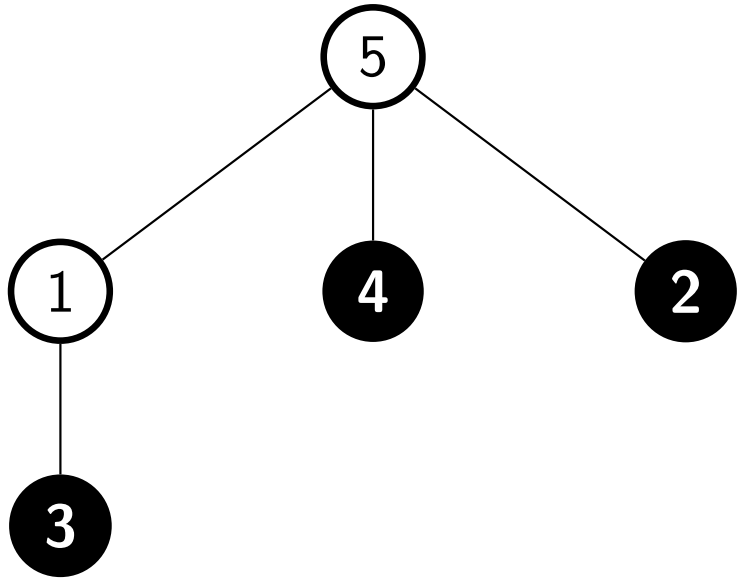


- Trees are very nice for doing DP on because subtrees do not share vertices.
- Compare to DAGs. DAGs are nice for certain tasks (like finding longest path) but it is hard to do anything where overlaps matter (e.g: for each vertex, count the number of vertices it can reach).
- However, this is trivial on trees, just recurse into our children and collect the results.
- Usually it is assumed or does not hurt to assume the tree is rooted.
- Then the order is either from the root down, or leaves up.
- Our state will be `subtree`, represented by the root of the subtree, along with additional metadata.
- Generally we code these top down, recursing with our `children` array.

# Example problem: Weighted MIS of a tree



- **Problem statement** Given a description of a tree $T$ $(1 \leq |T| \leq 1,000,000)$, with vertex weights, what is the maximum sum of vertex weights of an independent subset of the tree? An independent set is a subset of the nodes of the tree, with the property that no two nodes in the subset share an edge.
- **Example**



- As mentioned, root the tree arbitarily.
- It seems like the problem can be broken down into whether the root is included in the MIS or not. As such, a dp approach seems reasonable. We will try the dp on tree idea mentioned before.
- Let's just stick with the usual order for trees, down from the root.

- An obvious first guess at state: For each subtree $R$ rooted at some vertex $u$ of our tree $T$, what is the maximum weighted independent set contained in $R$?
- Then how do we calculate the maximum weighted independent set for the subtree rooted at $u$? We have two choices: either include $u$ in the MIS or do not.
  - If we do not include $u$ then the best MIS is just the sum of our children's best MIS.
  - If we do include $u$ then we need the best MIS of each of our children still. But these MIS can not include the children themselves.
  - Uh oh...we don't have access to this information. So let's amend our state.

- **Subproblems** For each subtree $R$ rooted at some vertex $u$ of our tree $T$, what is the maximum weighted independent set contained in $R$?
- Also we need the best MIS in $R$ that does **not** contain $u$.
- Define subproblems:
  - $f(u)$, the size of the maximum weighted independent set in the subtree rooted at $u$ that contains $u$.
  - $g(u)$, the size of the maximum weighted independent set in the subtree rooted at $u$ that does not contain $u$.

- **Recurrence** If we are considering an independent set that contains the root, then we must not take any of its children. However, if we don't have the root, then it doesn't matter whether we take the children or not. Hence, denoting the set of children of $u$ by $N(u)$, we have:

$$f(u) = w_u + \sum_{v \in N(u)} g(v)$$

$$g(u) = \sum_{v \in N(u)} \max(f(v), g(v)).$$

- **Base case** If $u$ is a leaf, then $f(u) = w_u$ and $g(u) = 0$.

# Example problem: Weighted MIS of a tree



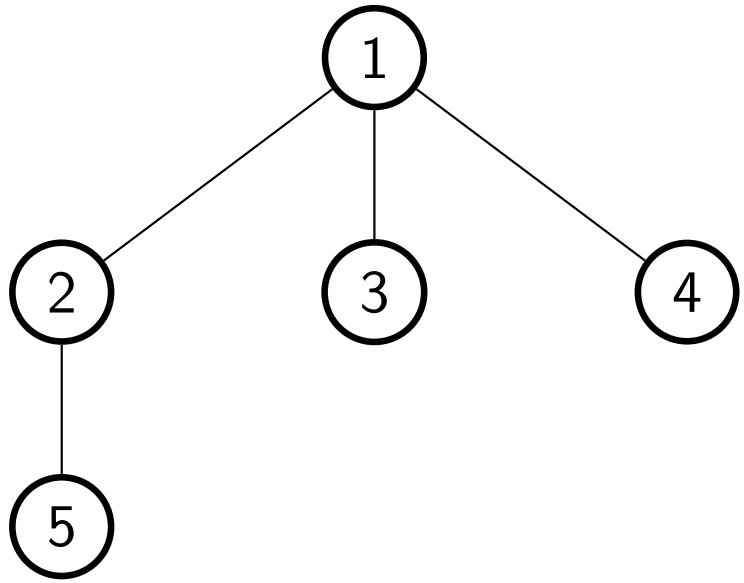
- **Complexity** Since we have $O(n)$ values of $f$ and $g$ to calculate, each taking $O(n)$ time to calculate, it seems that the overall complexity is $O(n^2)$.
- However, if we observe that each vertex $v$ only appears on the right hand side once (when $u$ is its parent), it can be seen that the overall complexity is in fact only $O(n)$.
- **Implementation**

```
void calculate_wmis (int u) {
  f[u] = w[u];
  g[u] = 0;
  for (int i = 0; i < children[u].size(); i++) {
    int v = children[u][i];
    calculate_wmis(v);
    f[u] += g[v];
    g[u] += max(f[v],g[v]);
  }
}
```

# Example problem: Count the subtrees



- **Problem statement** You are given a description of a tree $T$ ($1 \leq |T| \leq 1,000$), which is rooted at vertex 1, as well as an integer $K$ ($1 \leq K \leq 100$). How many subtrees have size at most $K$, modulo $10^9 + 7$?
- **Example**




- Suppose $K = 3$. There are 5 subtrees of size 1, 4 of size 2, 4 of size 3 = 13 total.

- The "modulo $10^9 + 7$" is there because the number of possible trees could be very large (well exceeding a long long...)
- We can compute the answer as we otherwise would, but along the way we need to make sure to modulo by $10^9 + 7$ to avoid integer overflow.
- Since this is a counting problem, the dp will associate a state with how many subtrees can be created from this state.

# Example problem: Count the subtrees



- As is typical in tree dp problems, our state can include the node which will be the root of the subtree.
- However, the size of the subtree we want to create is important too.
- So, we can try the state (Root of subtree, number of nodes in subtree)
- If we can compute this dp, the answer to the problem is just the sum of the dp values across all nodes, for all subtree sizes $\leq K$.

- We will now consider the recurrence for this state.
- If we want to create a subtree of size $K$ rooted at some node, we want to distribute the $K-1$ non-root nodes between its children.
- We could try every way to distribute nodes among the children (e.g. 5 to the first child, 3 to the second child, etc...)
- If it has one child, this is easy as there is only one way to do this.
- If it has two children, then there are $K$ ways to do this.
- In fact, if it has $C$ children, there are $K+C-1$ choose $C-1$ ways to distribute the $K-1$ nodes among them.
- If a node has a lot of children, this will be very large...

# Example problem: Count the subtrees



- Let's quickly analyse the complexity of our solution on a binary tree.
- There are $O(NK)$ states, and the recurrence at each state is $O(K)$, so we have an $O(NK^2)$ dp on binary trees.
- This solution is efficient because we pick how many nodes to give to our first child, and give the rest to our second child.
- For an arbitrary tree, we will try something similar.
- Pick some number of nodes to give to our first child, and then give the rest to the remaining children.

# Example problem: Count the subtrees



- We will modify our state - our new state will be (root of subtree, number of nodes in subtree, child we are up to)
- In other words, $dp(u, k, c)$ means we are building a subtree rooted at $u$, the subtree will have $k$ nodes in it (including $u$), and the subtree will not include the first $c$ children of $u$.
- Let $v$ be the $c$-th child of $u$. Then, the recurrence for $dp(u, k, c)$ would be
  - If $v$ is the final child of $u$, give $k-1$ nodes to $v$ (i.e. $dp(v, k-1, 0)$)
  - Otherwise, for all $0 \leq i < k$, give $i$ nodes to $v$, and the rest to the remaining children (i.e. $\sum_{i=0}^{k-1} dp(v, i, 0) \times dp(u, k-i, c+1)$).
  - Base cases ($k = 0$, no children)

# Example problem: Count the subtrees



- How many states are there?
- Naively, since each node can have $N$ children, there are $O(N^2K)$ states.
- However, since the sum of children across all nodes is $O(N)$, we actually have $O(NK)$ states!
- The recurrence at each state is $O(K)$.
- Overall, the dp is $O(NK^2)$.

# Example problem: Count the subtrees



```
#define MOD 1000000007
typedef long long ll;
vector<int> adj[MAX_N]; // We will assume this only has edges down the tree
vector<ll> f[MAX_N][MAX_K]; // Each vector has a length equal to the number of
      children of u. Initially, all -1.
ll dp(int u, int k, int child) {
  if (adj[u].empty()) { // No children
        return k <= 1;
  }
  if (f[u][k][child] != -1) return f[u][k][child];
  if (!k) return f[u][k][child] = 1;
  if (child+1 == adj[u].size()) { // Last child
    return f[u][k][child] = dp(adj[u][child], k-1, 0);
  } else { // Not last child
    f[u][k][child] = 0;
    for (int i = 0; i < k; i++) {
      f[u][k][child] += dp(adj[u][child], i, 0) * dp(u, k-i, child+1);
      f[u][k][child] %= MOD;
    }
    return f[u][k][child];
  }
}
```

# Example problem: Count the subtrees





```
int n, k;
ll ans;
int main() {
  scanf("%d%d", &n, &k);
  for (int i = 1; i < n; i++) {
    int a, b;
    scanf("%d%d", &a, &b);
    adj[a].push_back(b); // for simplicity, we will assume edges are given
          directed down the tree
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= k; j++) {
      f[i][j].assign(adj[i].size(), -1);
    }
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      ans += dp(i, j, 0);
      ans %= MOD;
    }
  }
  printf("%lld\n", ans);
}
```

# Table of Contents

# Example problem: The Great ATM Robbery 43

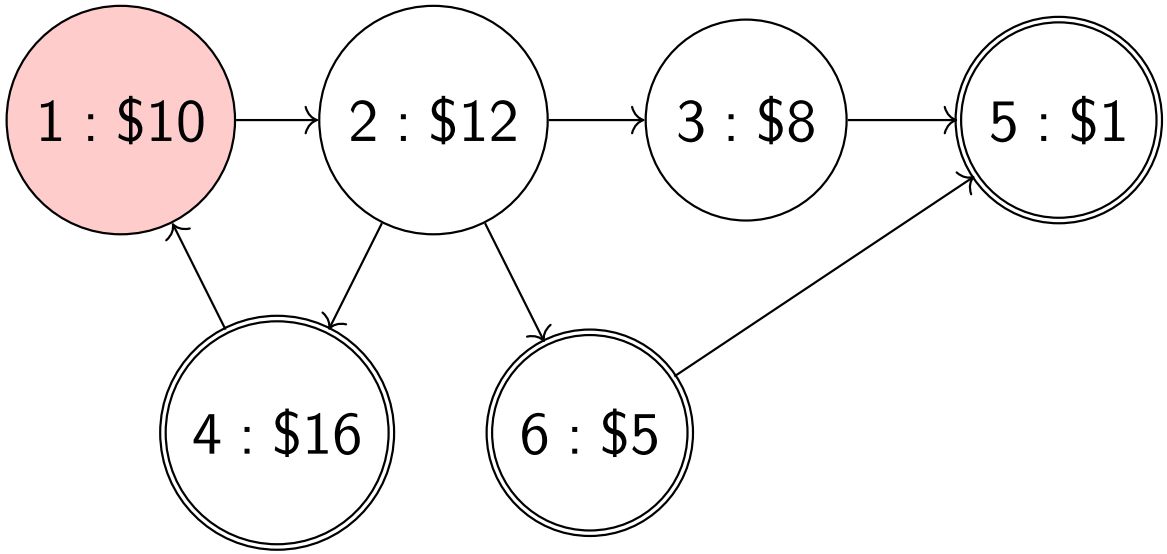
- **Problem statement** A city has $N$ intersections, joined by $M$ one-way roads. At each intersection $i$ is an ATM, with $c_i$ dollars of cash. Some specified intersections also have a pub.
  A bandit wishes to start at intersection $S$ (the city centre) and drive around, robbing all the ATMs he passes, before ending the day at one of the city's pubs. He may pass an ATM more than once, but the cash is not replenished after it is stolen. What is the maximum amount of money that he can steal?

- **Input** First line, $N$ $M$, $1 \leq N, M \leq 500,000$. Following this, $M$ lines, each with a pair $u_i$ $v_i$, $1 \leq u_i, v_i, \leq N$, $u_i \neq v_i$, representing a road from intersection $u_i$ to intersection $v_i$. Following this, $N$ lines, each with an integer $c_i$, $0 \leq c_i \leq 4,000$, representing the cash at the ATM at intersection $i$.
  Next line, $S$ $P$, $1 \leq S, P \leq N$, the starting intersection and the number of intersections with pubs. Following this, $P$ distinct integers, $p_i$, $1 \leq p_i \leq N$, the intersections containing pubs.
- **Output** A single number, the maximum amount of money that can be stolen on the way from the city centre to any one of the pubs.
- **Source** APIO 2009

# Example problem: The Great ATM Robbery



- **Example**




- The path $1 \to 2 \to 4 \to 1 \to 2 \to 3 \to 5$ collects \$47.

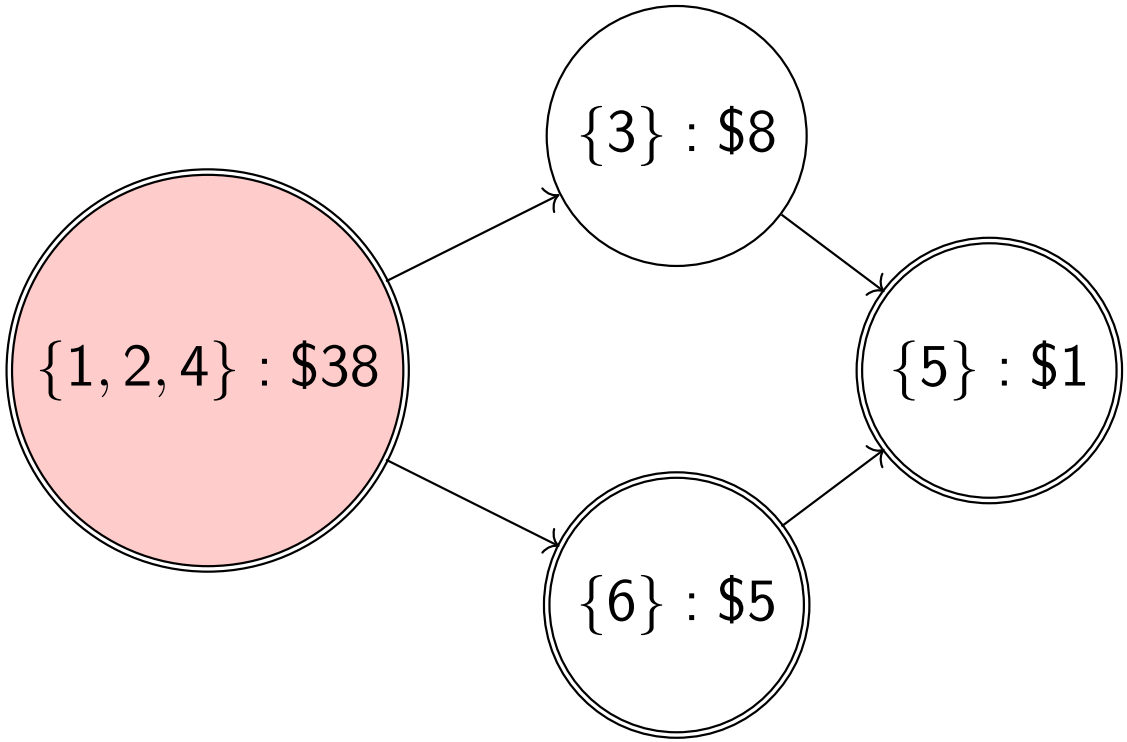
# Example problem: The Great ATM Robbery



- Intersections and roads correspond naturally to vertices and directed edges.
- Always steal full amount upon first visit to an ATM.
- This directed graph can have cycles (e.g. $1 \to 2 \to 4 \to 1$ in the sample case).
- If we enter a cycle, we may as well steal from all its ATMs!
    - We can always do a second pass of the cycle (for no money) if we need to reach a particular edge out of the cycle, or to finish at a pub.
- Is this approach (steal from every vertex, then if necessary traverse it again) limited to cycles?

- We want to take this approach for any (maximal) subset of vertices in which any pair can reach each other.
- Recall such subsets are called strongly connected components (SCCs), and can be found using Tarjan's algorithm or Kosaraju's algorithm.
- So we want to treat each SCC as a unit - condensation!
- In the new graph:
  - each vertex represents an SCC of the original graph,
  - each vertex weight represents the sum of vertex weights in the corresponding SCC, and
  - each edge represents an edge of the original graph (reattached accordingly, and with duplicates ignored).

- The condensation graph is directed, but no longer has cycles - it is a DAG.
- Apply topological sort (at least in principle) and then think about the problem.
- **Example**



# Example problem: The Great ATM Robbery



- We want the maximum weight path in a DAG, starting from the specified vertex $S$ and ending at one of the specified vertices $p_i$.
- Standard DP for longest path in a DAG + implementation details.
- **Complexity**
  - Find SCCs: $O(N+M)$ with either algorithm
  - Build condensation graph: $O(N+M)$
  - Topological sort: $O(N+M)$
  - DP: $O(N+M)$

# Example problem: The Great ATM Robbery





```
// Kosaraju's SCC finding algorithm, from the graph lecture
#define MAXN 500010
int n, m, seen[MAXN], postorder[MAXN], p, seen_r[MAXN], scc[MAXN];
vector<int> edges[MAXN], revEdges[MAXN], sccEdges[MAXN];
void dfs(int u) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) dfs(v);
  postorder[p++] = u;
}
void dfs_r(int u, int mark) {
  if (seen_r[u]) return;
  seen_r[u] = true;
  scc[u] = mark;
  for (int v : revEdges[u]) dfs_r(v, mark);
}
int compute_sccs() {
  int sccs = 0;
  for (int i = 1; i <= n; i++)
  if (!seen[i]) dfs(i);
  for (int i = p - 1; i >= 0; i--) {
    int u = postorder[i];
    if (!seen_r[u]) dfs_r(u, sccs++); // ignore visited vertices
  }
  return sccs;
}
int s, numPubs, inp, val[MAXN], hasPub[MAXN]; // both arrays are for SCCs, not
    individual nodes
int dpDone[MAXN], dpMemo[MAXN];
```

# Example problem: The Great ATM Robbery



```
int dp(int a) { // if we start at SCC a, what is the most money we can take?
  if (dpDone[a]) return dpMemo[a];
  dpDone[a] = 1;
  int ans = -2e9;
  if (hasPub[a]) ans = val[a];
  for (auto v : sccEdges[a]) ans = max(ans, dp(v)+val[a]);
  return dpMemo[a] = ans;
}
int main() {
  scanf("%d%d", &n, &m);
  for (int i = 0; i < m; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    edges[u].push_back(v);
    revEdges[v].push_back(u);
  }
  // compute SCC graph
  compute_sccs();
  for (int i = 1; i <= n; i++) {
    for (auto v : edges[i]) if (scc[i] != scc[v]) sccEdges[scc[i]].push_back(scc
        [v]);
  }
  for (int i = 1; i <= n; i++) scanf("%d", &inp), val[scc[i]] += inp;
  scanf("%d%d", &s, &numPubs);
  for (int i = 0; i < numPubs; i++) scanf("%d", &inp), hasPub[scc[inp]] = 1;
  printf("%d\n", max(0, dp(scc[s])));
}
```

# Example problem: Night Out



- **Problem statement** A street has $N$ restaurants, each with a distinct rating. For each of $Q$ nights, you want to visit the highest rated restaurant in a specified range, excluding any restaurants visited in the last $K$ nights.
- **Input** First line, $N$ $K$ $Q$, $1 \leq N, K, Q \leq 100,000$, $K \leq N$. Following this, $N$ distinct integers, $a_i$, $(1 \leq a_i \leq 100,000)$, the rating of each restaurant. Following this, $Q$ lines, $\ell_i$ $r_i$, $1 \leq \ell_i \leq r_i \leq N$, the left and right endpoints of the range considered on night $i$.
- **Output** $Q$ numbers on separate lines, the $i$th representing the number of the restaurant visited on night $i$. If there are no valid restaurants to choose on a given night, print $-1$ instead.
- **Source** ORAC

# Example problem: Night Out



- **Example Input**

```
5 1 4
10 30 80 20 70
2 4
2 3
1 2
1 1
```

- **Example Output**

```
3
2
1
-1
```

# Example problem: Night Out



- We have to query the maximum of each range in faster than linear time.
  - Sparse table: $O(N \log N)$ preprocessing, $O(1)$ query, but inflexible
  - Segment tree: $O(N \log N)$ preprocessing (can get this down to $O(N)$ if required), $O(\log N)$ query and supports updates
- This will fetch the maximum rating, and we can quickly identify the corresponding restaurant with a reverse lookup array, as all ratings are distinct. If the numbers used for ratings were larger, we would use a map instead.

# Example problem: Night Out



- The complication is that once a restaurant is picked, it is ineligible for the next $K$ nights.
- This can be accounted for with updates (so we must use a segment tree).
- When we select a restaurant, we immediately update its rating to 0 (less than all actual updates), then after $K$ nights, perform another update to restore its original rating.
- **Complexity** $O(N \log N)$ preprocessing, then $Q$ times we perform up to two updates and one range query in $O(\log N)$ each, so the total complexity is $O((N+Q)\log N)$.

# Example problem: Night Out





```
#define MAX_N 100001
// range tree code is taken (and slightly modified) from the DS lecture
// the number of additional nodes created can be as high as the next power of
      two up from MAX_N (131,072)
int tree[266666];

// a is the index in the array. 0- or 1-based doesn't matter here, as long as it
       is nonnegative and less than MAX_N.
// v is the value the a-th element will be updated to.
// i is the index in the tree, rooted at 1 so children are 2i and 2i+1.
// instead of storing each node's range of responsibility, we calculate it on
      the way down.
// the root node is responsible for [0, MAX_N)
void update(int a, int v, int i = 1, int start = 0, int end = MAX_N) {
  // this node's range of responsibility is 1, so it is a leaf
  if (end - start == 1) {
    tree[i] = v;
    return;
  }
  // figure out which child is responsible for the index (a) being updated
  int mid = (start + end) / 2;
  if (a < mid) update(a, v, i * 2, start, mid);
  else update(a, v, i * 2 + 1, mid, end);
  // once we have updated the correct child, recalculate our stored value.
  tree[i] = max(tree[i*2], tree[i*2+1]);
}
```

# Example problem: Night Out





```
// range tree code is taken (and slightly modified) from the DS lecture
// query the sum in [a, b)
int query(int a, int b, int i = 1, int start = 0, int end = MAX_N) {
  // the query range exactly matches this node's range of responsibility
  if (start == a && end == b) return tree[i];
  // we might need to query one or both of the children
  int mid = (start + end) / 2;
  int answer = 0;
  // the left child can query [a, mid)
  if (a < mid) answer = max(answer, query(a, min(b, mid), i * 2, start, mid));
  // the right child can query [mid, b)
  if (b > mid) answer = max(answer, query(max(a, mid), b, i * 2 + 1, mid, end));
  return answer;
}
```

# Example problem: Night Out





```
int n, k, q, val[MAX_N], reverseLookup[MAX_N], removed[MAX_N];
int main() {
  scanf("%d%d%d", &n, &k, &q);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &val[i]);
    reverseLookup[val[i]] = i;
    update(i, val[i]);
  }
  for (int i = 0; i < q; i++) {
    int a, b;
    scanf("%d%d", &a, &b);
    int r = reverseLookup[query(a, b+1)];
    update(r, 0);
    if (r) printf("%d\n", r);
    else printf("-1\n");
    removed[i] = r;
    if (i >= k) {
      // Re-add restaurants after k days
      update(removed[i-k], val[removed[i-k]]);
    }
  }
}
```

# Example: Greed



- **Problem Statement:** You have a rectangular cake with $R$ rows and $C$ columns. You will make $K$ cuts, cutting the cake into $K+1$ pieces. You must obey the following rules when cutting
  - Every cut must be made parallel to the sides of the rectangle.
  - Every cut must begin on either the left or bottom side of the cake, and must continue until it hits either the opposite side of the cake or an existing cut.
  - There are $H$ allowed locations for horizontal cuts and $V$ locations allowed for vertical cuts.

  You will take the largest slice, but do not want to appear greedy. You will choose which cuts are performed. What is the smallest possible area of the largest slice?

- **Input Format:** The first line contains 6 integers, $R, C, K, H, V$, the size of the cake, number of cuts, and number of allowed cut positions respectively.
  The next line contains $H$ integers, the $y$-coordinates of the allowed positions of the horizontal cuts. These will be distinct and increasing.
  The next line contains $W$ integers, the $x$-coordinates of the allowed positions of the vertical cuts. These will be distinct and increasing.
- **Bounds:** You are guaranteed that $2 \leq R, C$, $4 \leq RC \leq 10^9$, $K \leq H + V$ and $0 \leq H, V \leq 1,500$.
- **Source:** FARIO 2018.

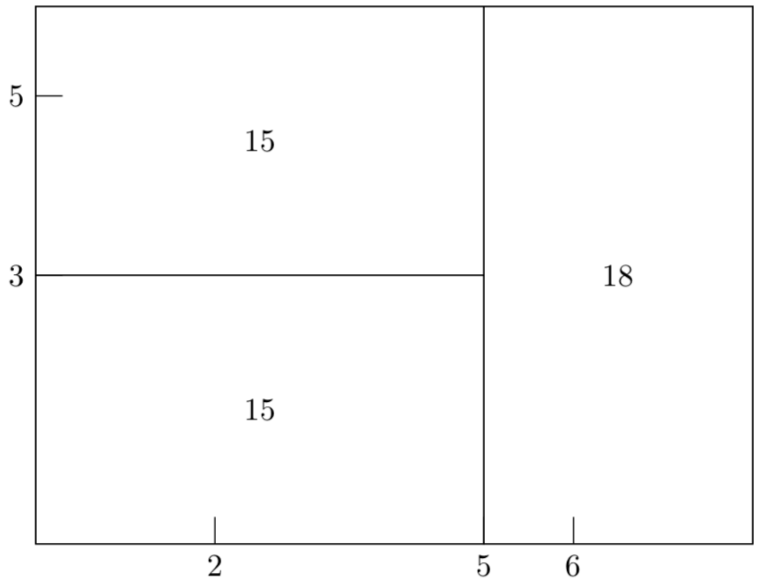
# Example: Greed



- **Sample Input:**

  6 8 2 2 3
  3 5
  2 5 6

- **Sample Output:** 18




- **Explanation:** First, do a vertical cut at position 5. Then, do a horizontal cut at position 3.

# Example: Greed



- **Observation:** given some sequence of cuts, there always exists a way to reorder the cuts so that the horizontal cuts occur in descending order, as do the vertical cuts, and the same slices are produced.
- Therefore, we can always assume horizontal and vertical cuts occur in descending order.
- Additionally, if we know the smallest horizontal and vertical cuts that have occurred, we have enough information to identify the remaining uncut cake.
- This inspires a dp, where the state is the smallest horizontal and vertical cuts we have performed so far.

# Example: Greed



- This inspires a dp, where the state is the smallest horizontal and vertical cuts we have performed so far.
- Each state will find the minimum area of the largest slice, only considering future cuts.
- However, this alone is not enough information! The number of cuts remaining will impact the answer.
- Modified state: (smallest horizontal cut so far, smallest vertical cut so far, number of cuts remaining)
- This dp will work, however there are too many states for it to be fast enough!

# Example: Greed



- **New idea:** We could try to binary search the answer.
- We must now solve a decision problem. Let $b(M)$ be true if we can perform $k$ cuts, with the largest slice having an area of at most $M$.
- This is the same as ensuring all slices have area at most $M$.
- **New problem:** What is the smallest number of cuts required so that every slice produced has an area at most $M$.

# Example: Greed



- **New problem:** What is the smallest number of cuts required so that every slice produced has an area at most $M$.
- As before, we can try to use dp. Because we are trying to minimise the cuts, we do not need to store this in our state. It is sufficient to have a state consisting of just the smallest horizontal cut and the smallest vertical cut.
- **Recurrence.** We could try all possible next cuts which do not produce a slice with area $> M$. However, if we have the choice between two horizontal cuts, we should take the one that produces the largest slice (with area $\leq M$). The same applies for vertical cuts.
- So, our recurrence is to try two options: the horizontal cut which produces the largest slice (with area $\leq M$), and the vertical slice which does the same.

- **Complexity:** There are $O(HV)$ states.
- If we do the recurrence naively, it's $O(H+V)$ (too slow).
- We can improve the recurrence to $O(\log)$ by using binary search to find the cuts we will perform.
- With two-pointers, we can solve the recurrence in amortised $O(1)$! (see code for details)
- Hence, we can solve the decision problem $b(M)$ in $O(HV)$ time.
- Hence, we can solve the optimisation problem (i.e., the actual problem) in $O(HW\log(RC))$ using binary search.

# Example: Greed





```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 1510
int r, c, k, h, v, horCuts[MAXN], verCuts[MAXN], dp[MAXN][MAXN], bestHorCut[MAXN
    ][MAXN], bestVerCut[MAXN][MAXN];
bool canDo(int area) {
  // Precomp best cuts using two-pointers
  for (int i = 0; i <= h; i++) { // fix the horizontal cut, iterate over
      vertical cuts
    int cut = 0;
    for (int j = 0; j <= v; j++) {
      while (horCuts[i]*(verCuts[j]-verCuts[cut]) > area) cut++; // keep
          increasing cut until we have a cut which produces a small-enough
          slice
      bestVerCut[i][j] = cut;
    }
  }
  for (int j = 0; j <= v; j++) { // fix the vertical cut, iterate over
      horizontal cuts
    int cut = 0;
    for (int i = 0; i <= h; i++) {
      while (verCuts[j]*(horCuts[i]-horCuts[cut]) > area) cut++; // keep
          increasing cut until we have a cut which produces a small-enough
          slice
      bestHorCut[i][j] = cut;
    }
  }
  // continued on next slide...
```

# Example: Greed





```
  // do the dp
  for (int i = 0; i <= h; i++) {
    for (int j = 0; j <= v; j++) {
      dp[i][j] = 1e9;
      if (horCuts[i]*verCuts[j] <= area) dp[i][j] = 0; // Case 1: no cuts
          required
      if (bestHorCut[i][j] != i) dp[i][j] = min(dp[i][j], dp[bestHorCut[i][j]][j
          ]+1); // Case 2: do horizontal cut
      if (bestVerCut[i][j] != j) dp[i][j] = min(dp[i][j], dp[i][bestVerCut[i][j
          ]]+1); // Case 3: do vertical cut
    }
  }
  return dp[h][v] <= k;
}
int main() {
  scanf("%d%d%d%d%d", &r, &c, &k, &h, &v);
  for (int i = 0; i < h; i++) scanf("%d", &horCuts[i]);
  horCuts[h] = r; // We include r as a horizontal cut so that the dp can
      consider the case where we have done no horizontal cuts
  for (int i = 0; i < v; i++) scanf("%d", &verCuts[i]);
  verCuts[v] = c; // We include c for a similar reason
  // binary search
  int s = 0, e = r*c;
  while (s != e) {
    int m = (s+e)/2;
    if (canDo(m)) e = m;
    else s = m+1;
  }
  printf("%d\n", s);
}
```