Revision

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents 2

- **Problem statement** There are $n+1$ towns, joined by n trains $(0 \to 1, 1 \to 2, \ldots, (n-1) \to n)$. Each train has duration a hours and leaves every b hours from time 0. Maximise the minimum time spent waiting at any town while arriving at town n by time T.
- **Input** First line, n T, $2 \leq n \leq 100,000$, $1 \leq T \leq 1,000,000,000$. Following this, N lines, a_i b_i, $1 \leq a_i, b_i \leq 100,000$.
- **Output** One number, the most hours that can be spent at each town.

- Not obvious how to tackle this directly.
- Note the phrase "maximise the minimum" - can we reduce this optimisation problem to a decision problem using binary search?
- Check monotonicity: let $f(x)$ be 1 if a waiting time of x is feasible, and 0 otherwise. Clearly $f(x) = 1$ implies $f(x-1) = 1$ also, so f is indeed monotonic.
- Is the decision problem significantly easier?

- How to find $f(x)$, that is, whether a waiting time of x is feasible?
- Greedy: after waiting x hours at each town, there's no reason not to get on the next train departing the town.
- Can test $f(x)$ in one pass: keep track of time, and at each town add x before rounding up to the nearest multiple of the appropriate departure time (details to follow)
- **Complexity** Answer is no larger than T, so binary search overhead is $O(\log T)$, for overall complexity $O(n \log T)$.

- You must use long longs (or be very proactive in checking for overflow)
- How to round p up to the nearest multiple of q? Mathematically, we are calculating $\lceil p/q \rceil \times q$.
- In C/C++, `p/q` calculates $\lfloor p/q \rfloor$, and we have

$$\left\lceil \frac{p}{q} \right\rceil = \begin{cases} \lfloor p/q \rfloor & \text{if } q \mid p \\ \lfloor p/q \rfloor + 1 & \text{if } q \nmid p \end{cases}.$$

- Equivalently, we can directly round up as follows:

```
||    if (p%q) p += (q - p%q);
```

- Alternatively, use the fact that $\lceil p/q \rceil = \lfloor (p-1)/q \rfloor + 1$.
- Some students got the verdict COMPILE-ERROR because they declared arrays too big to fit in memory.

```
const int N = 100100;
int d[N], f[N], n, T;
bool go (int x) {
  long long t = d[0];
  for (int i = 1; i < n; i++) {
    t += x;
    t = ((t-1)/f[i]+1)*f[i]; // round t up to the nearest multiple of f[i]
    t += d[i];
  }
  return t <= T;
}
int main() {
  cin >> n >> T;
  for (int i = 0; i < n; i++) cin >> d[i] >> f[i];
  int bestSoFar = -1;
  int lo = 0;
  int hi = T;
  while (lo <= hi) {
    int mid = (lo+hi)/2;
    if (go(mid)) {
      bestSoFar = mid
      lo = mid + 1;
    } else {
      hi = mid-1;
    }
  }
  cout << bestSoFar << '\n';
}
```

- **Problem statement** There are n sizes of boxes (size 1 to size n), each with a value. A box of size k can carry other boxes, as long as its direct contents have sizes adding up to less than k. What is the maximum value that can be stolen, if you can only carry a single box?
- **Input** First line, n, $1 \leq n \leq 5,000$. Following this, N integers, v_i, $1 \leq v_i \leq 200,000$, the value of a box of size i.
- **Output** One number, the maximum value that can be achieved.

- Even though this is an optimisation problem, and the corresponding decision problem is monotonic, we find that the decision problem isn't any easier, so binary search isn't helpful.
- **Observation** the box you carry at the end should be of maximum size; otherwise, you could put it in a box of maximum size and increase the total value.
- So the answer is v_n plus the maximum value you can fit into $n-1$ units of space.
- Can be solved using dynamic programming.

- Natural order: increasing size.
- Try simplest state: let $dp[k]$ be the maximum value you can fit into k units of space.
- Can we make a recurrence? Consider the moves available to fill this amount of space.
- Either take a box of size k, and fill its interior for maximum value, or use multiple boxes, which requires some break point j between boxes.
- Therefore the recurrence is

$$dp[k] = \max\left(v_k + dp[k-1], \max_{0<j<k}\left(dp[j] + dp[k-j]\right)\right).$$

- **Complexity** $O(n)$ states, each requiring $O(n)$ work to compute, for a total of $O(n^2)$.

- Long longs are not required here; no more than n boxes can ever be used (proof by induction). But if you're not sure, there's no harm in using long longs.
- You can alternatively use a state with two parameters (e.g. fill k units of space using boxes of size up to j), so there are $O(n^2)$ states each requiring $O(1)$ work to compute (the recurrence is left as an exercise).
  - This requires a two-dimensional array.
  - In C/C++, it is significantly faster to traverse a 2D array by rows than by columns!
  - This is because each row is stored in consecutive locations in memory, allowing better caching.

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

const int N = 100100;
int a[N];
int dp[N];

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; i++)
    cin >> a[i];
  for (int i = 1; i < n; i++) {
    dp[i] = a[i] + dp[i-1];
    for (int j = 1; j <= i-j; j++)
      dp[i] = max(dp[i],dp[j]+dp[i-j]);
  }
  cout << a[n] + dp[n-1] << '\n';
}
```

- **Problem statement** There are n beakers, numbered from 1 to n, all initially containing different chemicals. Each chemical is an acid (A), base (B), or neutral (N). There are m reactions. Each reaction involves pouring some beaker x into another beaker y, causing their contents to combine. However, if the combined contents include at least one acid and at least one base, the entire mixture inside the beaker is turned into gas.
  After all the reactions have occurred, you wish to know, for each initial chemical, which beaker it ends up in, or if it was turned into gas.

- **Input** First line contains n and m, $2 \leq n \leq 100,000$, $1 \leq m \leq 100,000$. Second line n characters, 'A', 'B' or 'N', the initial state of the beakers. The next m lines each contain two integers, x_i and y_i, the two beakers used in the i-th reaction.
- **Output** Output n lines, the ith of which is either the number of the beaker where chemical i finishes, or 0 if chemical i is turned to gas.

- Note that we are merging objects - perhaps we can use union-find?
- However, we cannot just use standard union-find to merge beakers - this is because a chemical can leave a beaker!
- To handle this, we need to keep track of both the set which chemical i is in, and the set which is currently occupying beaker i (for all i).

- In our union-find, we have an array to store which set each chemical is in (as is standard in union-finds). We have another array to store, for each set, if it has an acid or base in it.
- We have another array which stores, for each beaker, which set currently occupies it, or if it's empty.
- When we merge two beakers, we merge the corresponding sets. However, if one of the sets is empty, we should not merge them (otherwise, we would be turning chemicals into gas when we shouldn't be).
- We will turn a chemical into gas by merging it with 0.
- The time complexity is the same as a standard union-find.

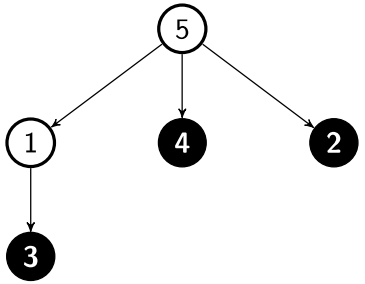
```
#define MAXN 100010
struct UF {
    int rep[MAXN]; // rep[i] is the set that chemical i is in
    int type[MAXN]; // the type of a chemical. 0 = neutral, 1 = acid, 2 =
        base
    int inBeaker[MAXN]; // inBeaker[i] is the set currently occupying beaker
        i
    int whichBeaker[MAXN]; // whichBeaker[i] is the beaker that chemical i
        is currently in
    UF() {
        for (int i = 0; i < MAXN; i++) rep[i] = inBeaker[i] = whichBeaker[i]
            = i;
    }
    int findrep(int a) {
        if (rep[a] == a) return a;
        return rep[a] = findrep(rep[a]); // path compression
    }
    void merge(int x, int y) { // We are merging beaker x into beaker y
        int inx = findrep(inBeaker[x]), iny = findrep(inBeaker[y]);
        if (inx && iny) {
            rep[inx] = iny;
            if (type[inx] + type[iny] == 3) rep[iny] = inBeaker[y] = 0; //
                If a chemical reaction occured, turn the chemical into a
                gas
            else type[iny] |= type[inx];
        }
        else if (inx && !iny) {
            inBeaker[y] = inx;
            whichBeaker[inx] = y;
        }
        inBeaker[x] = 0; // Beaker x is now empty
    }
```

```
UF uf;
int n, m;
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        char a;
        scanf(" %c", &a);
        if (a == 'A') uf.type[i] = 1;
        else if (a == 'B') uf.type[i] = 2;
    }
    for (int i = 0; i < m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        uf.merge(x, y);
    }
    for (int i = 1; i <= n; i++) {
        printf("%d\n", uf.whichBeaker[uf.findrep(i)]);
    }
}
```

- We can also solve this problem using small-merge.
- For each beaker, we maintain a list of all the chemicals currently in that beaker.
- When we merge two beakers, we insert all the elements from the smaller list into the bigger list. This is very similar to the size-heuristic for union-find.
- If a beaker is turned into gas, we iterate over all of the elements in the list and mark them as gas, before clearing the list.
- Small merge has an overall complexity of $O(n \log n)$, as each element can change lists at most $O(\log n)$ times.
- Further reading: https://cp-algorithms.com/data_structures/disjoint_set_union.html#toc-tgt-15

- 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.

- **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**



- 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 steps we can make, 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:
  - Let $f(u)$ be the size of the maximum weighted independent set in the subtree rooted at u that contains u.
  - Let $g(u)$ be 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:

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

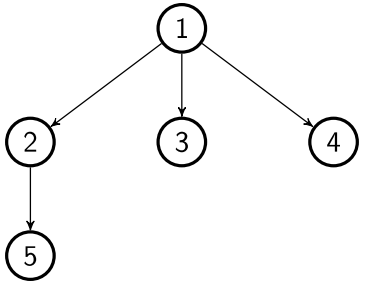
$$g(u) = \sum_{v \in N(u)} \max(f(v), g(v)).$$

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

- **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]);
  }
}
```

- **Problem statement** You are given a description of a tree T ($1 \leq |T| \leq 1,000$), and an integer K ($1 \leq K \leq 100$). How many subtrees have size at most K, modulo $10^9 + 7$?
- **Example**
- $K = 3$.
- 5 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.

- 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...

- 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.

- 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)

- 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 $O(NK^2)$.

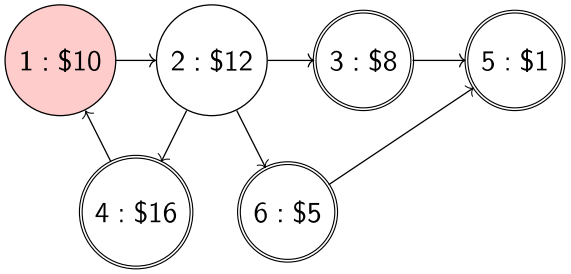
```
#define MOD 1000000007
typedef long long ll;
vector<int> adj[MAX_N]; // We will assume this only has edges down the tree
vector<ll> f[MAX_N][MAX_K]; // Each vector has a length equal to the number
     of children of u. Initially, all -1.
ll dp(int u, int k, int child) {
  if (adj[u].empty()) { // 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];
  }
}
```

```
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);
}
```

- **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**

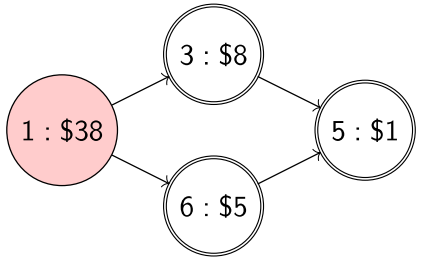


- The path $1 \to 2 \to 4 \to 1 \to 2 \to 3 \to 5$ collects \$47.

- 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**



- 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: $O(N + M)$

```
// 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];
```

```
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])));
}
```

- **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, ℓ_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 ith 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 Input**

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

- **Example Output**

```
3
2
1
-1
```

- 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.

- 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 $O((N + Q) \log N)$.

```
#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]);
}
```

```
// 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;
}
```

```
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]]);
    }
  }
}
```

- **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.

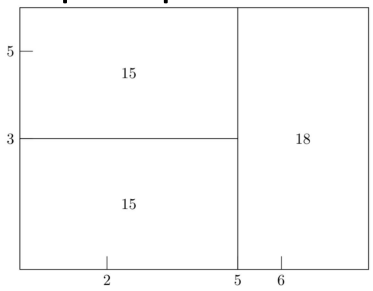
  You will take the largest slice, but do not want to appear greedy. You will choose which cuts are performed. What is the smallest possible area of the largest slice?

- **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.

- **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.

- **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.

- 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!

- **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.

- **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.

```
#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...
```

```
  // 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);
}
```

- Sometimes when you're doing DP your solution will be too slow but your recurrence is very structured.
- You've seen this before with ranges and range trees.
- This lecture will show a few more cases of structured recurrences where there are well known techniques for speeding them up.
- Again, this does not help if your state space is too large. This should not really change your overall approach to DP but you should keep an eye out for recurrences that can be sped up.

- One of the easiest to apply and most useful optimizations.
- Handy when your recurrence is formed by linear functions.
- Though once you get good at spotting these, more things look like a linear function than you might expect!

- General setting is:
  - You are doing a 1D DP. Let's say you are calculating $dp[N]$ in order of increasing i.
  - You have an array $m[N]$. Think of these as gradients. The array $m[N]$ is in *decreasing order*.
  - You have an array $p[N]$. Think of these as the positions of the points for which you are calculating $dp[N]$.
- You have some base case value $dp[0]$ and the recurrence is:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Take a look at the recurrence:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Hopefully you see an $O(N^2)$ solution.
- What do those terms in the min look like?
- Equations for lines with $dp[j]$ as the y-intercept and $m[j]$ as the slope.

- Recurrence:

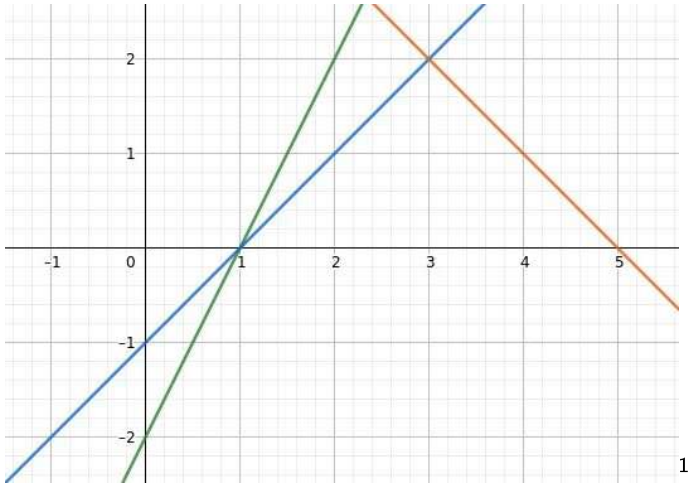
$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

- Suppose so far we have:
  - $dp[0] = -2, m[0] = 2$
  - $dp[1] = -1, m[1] = 1$
  - $dp[2] = 5, m[2] = -1$
- And we want to calculate $dp[3]$ (leave $p[3]$ unfixed for now).
- Then $dp[i]$ is the minimum of a set of lines at the x coordinate $p[3]$ where our lines are:
  - $y = m[0]x + dp[0] = 2x - 2$
  - $y = m[1]x + dp[1] = x - 1$
  - $y = m[2]x + dp[2] = -x + 5$
- So far we have done this in $O(N)$ for each $dp[i]$ calculation. We will see how to speed this up.

- To speed up this recurrence, we will build a data structure that exactly supports the operations we need.
- It will export 2 methods:
  - `void add(int m, int b)`: Add a line $l = mx + b$. **Requirement:** m is strictly less than the gradient of all lines in the data structure already.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return $\min_i(m_i q + b_i)$.
- We will have `add` run in $O(1)$ *amortized* and `query` run in $O(\log n)$ where n is the number of lines added so far.
- Afterwards we will apply this data struture as a black box to some DP problems.

- To get the right complexity, we need some observations first regarding the set of lines we are querying.
- For concreteness, let us return to our earlier example. Suppose we have:
  - $b[0] = -2, m[0] = 2$
  - $b[1] = -1, m[1] = 1$
  - $b[2] = 5, m[2] = -1$

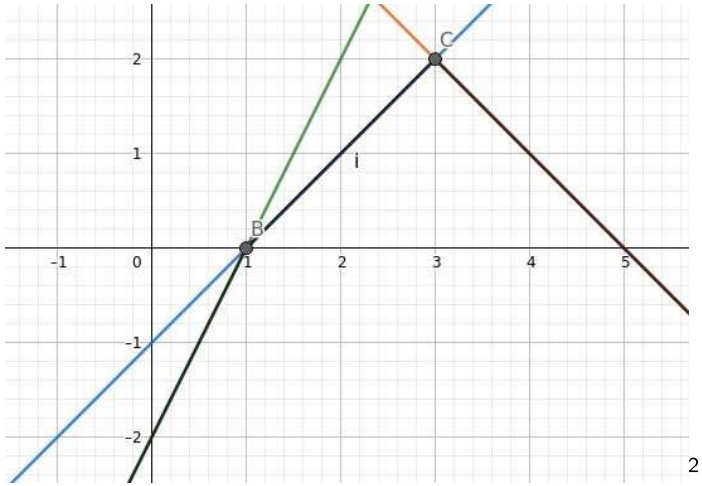
  (where $m[i]$ are the gradients and $b[i]$ are the y-intercepts)



[1]

[1] https://www.geogebra.org/graphing

- Remember our aim is to query the minimum over these 3 lines at the x coordinate q.
- What could this possibly be?
- Let's emphasize what the minimum value is at each x coordinate.



2

[2] https://www.geogebra.org/graphing

- **Key Observation:** This is the upper convex hull of the areas below the lines.
- This means each line is optimal for a contiguous range of x (possibly empty) and as we move left to right, the gradient of the optimal line never increases.
- Our goal is to maintain the convex hull by maintaining the set of line segments that the convex hull is made out of.
- We will say a line l is dominant at x-coordinate x if it is the line that forms the convex hull at x. This is the line that gives us our minimum in the equation

$$query(q) = \min_i(m_i \cdot q + b_i)$$

- For this, it suffices to store the lines that make up the convex hull in left to right order. (same as in decreasing gradient order).
- **Note:** Importantly, this does not contain all the lines. It omits any line that is never in the upper convex hull.
- Given this data, we can calculate the range of x at which each line l is dominant. The segment is the range between the intersection of l with the line before it, and after it, in the convex hull.
- We keep the lines in a vector.

```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;
/*
 * The intersection points are
 *  intersect(cht[0], cht[1]), intersect(cht[1], cht[2]), ...
 *  Line i is optimal in the range
 *      [intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]
 *  where for i = 0, the first point is -infinity,
 *  and for i = N-1, the last point is infinity.
 */
```

- We keep the lines in a vector.
- Recall the 2 methods our data structure is meant to support are:
  - What's the line on the upper convex hull at $x = q$? (Equivalently, what is $\min_i(m_i \cdot q + b_i)$ over all the lines)
  - Add the line $y = m[i] * x + dp[i]$.
- We handle the former with binary search.

- Our aim is to find the line that is dominant at $x = q$.
- Suppose our lines are ordered in decreasing gradient order. Then recall the range at which line l_i is dominant is

$$[intersect(l_{i-1}, l_i), intersect(l_i, l_{i+1})]$$

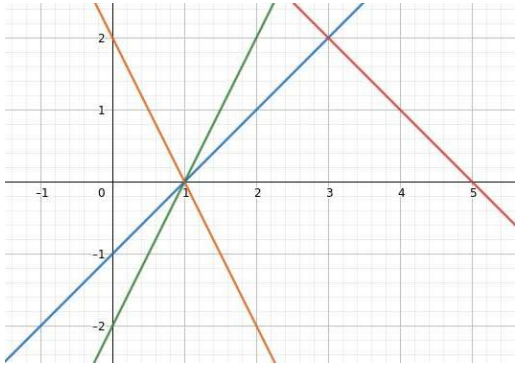
  where for $i = 0$, the left term is $-\infty$ and for the last line, the right term is $+\infty$.
- So to find the i for which the segment contains q, it suffices to find the minimum i such that

$$intersect(l_i, l_{i+1}) \geq q$$

```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;

/* Recall that the range the ith line is dominant in is:
 * [intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]
 * We want to find the line that is dominant at x.
 * To do this, we note that the sequence (intersect(cht[i], cht[i+1]))
 * is monotonically increasing in i.
 * Hence we can binary search for the minimum i such that
 * intersect(cht[i], cht[i+1]) >= x
 */
long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    // Find largest idx such that x <= intersect(cht[idx], cht[idx+1])
    // If this doesn't exist then idx should be cht.size()-1.
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

- To add a line we crucially use the fact that $m[N]$ is in decreasing order.
- So the new line has to go on the end of our convex hull (recall the convex hull is sorted in non-increasing order of gradients).
- However, this may cause some lines to disappear from the convex hull.

- For example, consider adding the line $y = -2x + 2$ to the earlier example:
- 

- What is the new convex hull?

- So some of the lines may become useless and we need to remove them.
- When does a line l become useless? When the line we just added covers the entire range l is dominant in.
- **Observation:** The useless lines are always at the end of the convex hull.
- So we just need to keep popping the last line of the convex hull as long as it is useless.
- How do we check if the last line is useless?
- For this, it helps to draw pictures and move your new line l around.

- You will hopefully observe some variant of the following: Letting $cht[N-1]$ be the last line, it is useless if:

  $$intersect(cht[N-1], l) \leq intersect(cht[N-2], cht[N-1])$$

- Recall the range that $cht[N-1]$ is dominant in is $(intersect(cht[N-2], cht[N-1]), \infty)$.
- The above essentially says l is better than $cht[N-1]$ for this entire range.
- Another way of phrasing this is that the intersections $(intersect(cht[i], cht[i+1]))$ need to be kept in increasing order.

```cpp
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;

void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
           intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
}

long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    // Find largest idx such that x <= intersect(cht[idx], cht[idx+1])
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

- **Complexity?** $O(1)$ *amortized* per add. $O(\log n)$ per query.
- But do *remember*, we did assume the gradients are in decreasing order. It is non-trivial to remove this assumption.
- In certain special cases, we can actually get a better complexity!

- One common case, each query we make is at least the previous query we make. (formally, say our code calls query with $query(q_1), query(q_2), \ldots, query(q_Q)$. Then $q_1 \leq q_2 \leq \ldots \leq q_Q$)
- In this case, the index of the dominant line for each q_i only increases.
- So we keep track of the index of the dominant line. Whenever we query, we check if the current dominant line is still the dominant line for the new query point.

- If cp is the current dominant line, this amounts to checking if

$$p[i] \leq intersect(cht[cp], cht[cp+1])$$

- While this does not hold, increase cp.
- The above had an omission. There is a special case. Since the number of lines in the convex hull can decrease, cp may be out of bounds. To fix this, after each update we just need to update cp to point to the last line if it is out of bounds.
- If you want to rigorously check this, the invariant you are maintaining is

$$intersect(cht[cp-1], cht[cp]) \leq p[i]$$

```
struct line { long long m, b; };
double intersect(line a, line b) {
    return (double)(b.b - a.b) / (a.m - b.m);
}
// Invariant: cht[i].m is in decreasing order.
vector<line> cht;
int cp;

void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
           intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
    cp = min(cp, (int)cht.size()-1);
}

long long query(long long x) {
    while (cp+1 != cht.size() &&
           intersect(cht[cp], cht[cp+1]) < x) cp++;
    return cht[cp].m*x + cht[cp].b;
}
```

- **Complexity?** Now updates and queries are both $O(1)$ amortized.

- Currently query returns the minimum over all lines. We can also instead return the maximum. If so, our invariant is that our gradients should be increasing. We are now calculating the lower convex hull of the area above all lines.
- But we can actually use the exact same code! (literally no modifications needed)

- Not hard to adjust for gradients non-increasing (instead of decreasing). Just one extra special case.
- Can also have no conditions on the gradients. We still keep lines in sorted gradient order. However, we now need to keep lines in a set since insertions are arbitrary. Not that common mostly because it is tedious to get right.
- Alternatively we can also use a "Li Chao tree".

- Another useful modification is when everything (gradients and query points) is in integers.
- Currently we use doubles to compute the intersection points. This gives us the precise ranges at which we dominate (up to precision errors) $[intersect(cht[i-1], cht[i]), intersect(cht[i], cht[i+1])]$.
- Doubles are not precise though even for the range of a long long. Usually not problematic for competitions since numbers generally only go up to 10^9 but you have to keep it in mind if you need to query higher.
- If we only care about integer coordinates of x then we can round these down and work entirely in integers:

$$([\lfloor intersect(cht[i-1], cht[i]) \rfloor, \lfloor intersect(cht[i], cht[i+1]) \rfloor]$$

- But if we only care about integer coordinates of x then we can round these down:

  $$(\lfloor intersect(cht[i-1], cht[i]) \rfloor, \lfloor intersect(cht[i], cht[i+1]) \rfloor]$$

- Just **be careful**. If you want to do this, **make sure** it is clear to you why the above is exclusive-inclusive.
- In integers, whether your inequalities are strict or not actually matters. When you compare with an intersection point, mentally check if it agrees with the above ranges.
- If we only care about *positive* integers, we can omit the floor and just use regular integer division. But if you care about negatives, **note that** integer division isn't floor, it rounds towards 0.

CHT without floats


```
struct line { long long m, b; };
long long floordiv(long long a, long long b) {
    return a / b - (a%b && ((a<0) ^ (b<0)));
}
long long intersect(line a, line b) {
    floordiv(b.b - a.b, a.m - b.m);
    // for POSITIVE ints: can do:
    // return (b.b - a.b) / (a.m - b.m);
}
vector<line> cht;
void add(line l) {
    auto n = cht.size();
    while (n >= 2 &&
           intersect(cht[n-1], cht[n-2]) >= intersect(cht[n-1], l)) {
        cht.pop_back();
        n = cht.size();
    }
    cht.push_back(l);
}
long long query(long long x) {
    int lo = 0; int hi = cht.size()-2;
    int idx = cht.size()-1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        // NOTE: It is critical here that this is >= not >.
        if (intersect(cht[mid], cht[mid+1]) >= x) {
            idx = mid; hi = mid-1;
        } else { lo = mid+1; }
    }
    return cht[idx].m*x + cht[idx].b;
}
```

- The above construction gives us exactly the data structure we needed.
- As a reminder, it supports 2 methods:
  - `void add(int m, int b)`: Add a line $l = mx + b$.
    Requirement: m is strictly less than the gradient of all lines in the data structure already.
    Complexity: $O(1)$.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return

$$\min_i m_i q + b_i$$

    Complexity: $O(\log n)$ in general, we can make it $O(1)$ if queries are given in non-decreasing order.

- There are 2 knobs we can tweak for our data structure.
  - If queries are in non-decreasing order, we can replace our query code with an $O(1)$ routine.
  - If everything is in integers, we can use integer division.
- However, aside from these, we can essentially use it as a black box!
- So the construction is good to know, but application wise, you can just copy paste it in if need be.

- Let us return to our earlier DP problem. We had the recurrence:

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

  where we assumed $m[j]$ is in decreasing order.
- Since $m[j]$ is decreasing, we fulfil the one requirement of our CHT data structure. Hence we can directly apply convex hull trick as a black box.

```
int main() {
    // Base case:
    dp[0] = baseCaseCost;
    line l;
    l.m = m[0];
    l.b = dp[0];
    add(l);

    for (int i = 1; i < N; i++) {
        dp[i] = query(p[i]);
        line l;
        l.m = m[i];
        l.b = dp[i];
        add(l);
    }
    return 0;
}
```

- **Complexity?** $O(N \log N)$.
- This is a significant improvement from $O(N^2)$!
- Note, we could even get $O(N)$ if our $p[i]$ are non-decreasing.

- DPs with recurrences in the form

$$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

  can be done in $O(N \log N)$.
- This is done by creating a data structure for CHT which supports:
  - `void add(int m, int b)`: Add a line $l = mx + b$. **Requirement:** m is strictly less than the gradient of all lines in the data structure already.
  - `int query(q)`: Over all lines $\{l_i = m_i x + b_i\}$ that we have added so far, return $\min_i(m_i q + b_i)$.
- You can treat this data structure as a black box. However it is good to at least know how to tweak it using the 2 knobs mentioned above.

- In practice, the difficulty of CHT comes from:
  - Recognizing it is useful. Often you just have to write out the recurrences. Be suspect whenever the recurrence is given by a formula.
  - Figuring out how to calculate the gradients and y intercepts. You will often have to juggle around terms in the recurrence to make it work. This comes mostly with practice.
  - Essentially, the recurrence

    $$dp[i] = \min_{j<i}(dp[j] + m[j] * p[i])$$

    tells us that we need the gradient to be a function of only j and the query point to be a function of only i.

- **Problem Statement:** I have N points on a walkway I need to cover. To cover the walkway from point x to point y inclusive costs $C + (x - y)^2$ where C is a fixed, given constant. Note that you can cover a single point with cost C. What is the minimum cost needed to cover all the points?
- **Input Format:** First line 2 integers, $N, C, 1 \leq N \leq 10^6, 1 \leq C \leq 10^9$. The next N lines contain the points in increasing order. All points are in the range $[1, 10^9]$.
- **Source:** 2012 University of Chicago Invitational Programming Contest.

- Hope you can see a $O(N^2)$ DP.
- Calculate $dp[N]$, where $dp[i]$ is the min cost to cover exactly the points up to the ith.
- But the recurrence here is a very nice formula. Let us unpack it.

$$\begin{aligned} dp[j] &= \min_{i<j} dp[i-1] + C + (x[j] - x[i])^2 \\ &= \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j] \end{aligned}$$

- Focus on the last term. What does this look like?
- A linear function (if you squint hard enough)! So we should try to fit this into our CHT framework.

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- If you recall, we were earlier looking at recurrences of the form $\min_{i<j} dp[i] + m[i] * p[j]$.
- What should $m[i]$ and $p[j]$ be?
- We should have $m[i] = -2x[i]$, $p[j] = x[j]$ (since we need the gradient to be determined by i and the query point by j).
- Note that our gradients are decreasing which we need for our CHT!

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- We also need to modify the y-intercept of the line, it isn't just $dp[i]$ anymore. What should it be?
- **Key:** It needs to include all terms dependent on i. So it needs to include $dp[i-1]$ and $x[i]^2$. **Why?**
- But it can't include $x[j]^2$ for obvious reasons. So we should add the $x[j]^2$ part when we calculate $dp[j]$.
- You can choose whether to add C when you calculate $dp[j]$ or whether to add C to the y-intercept.

- Recurrence:

$$dp[j] = \min_{i<j} dp[i-1] + C + x[i]^2 + x[j]^2 - 2 * x[i] * x[j]$$

- So for i we will add a line where:
  - y-intercept is: $dp[i-1] + x[i]^2$.
  - Gradient is: $-2x[i]$.
- And to calculate $dp[j]$, we query our CHT with point $x[j]$. Define $r := query(x[j])$. Then $dp[j] = r + C + x[j]^2$.

```
/* Insert CHT code here: You can use the version with only
 * positive integer queries and non-decreasing queries */
void add(line l);
long long query(long long x);

const int MAXN = 1000000;
int N;
long long x[MAXN+1];
long long C;

long long dp[MAXN+1];

int main() {
    scanf("%d %lld", &N, &C);
    for (int i = 0; i < N; i++) {
        scanf("%lld", &x[i]);
    }
    for (int i = 0; i < N; i++) {
        // Compare to formulas written in slides
        dp[i] = query(x[i]) + C + x[i]*x[i];
        line l;
        l.m = -2*x[i];
        // Base case is i == 0, dp[i-1] = 0.
        l.b = (i == 0 ? 0 : dp[i-1]) + x[i]*x[i];
        add(l);
    }
    // Again, calculating dp[N-1] using the same formula as above.
    printf("%lld\n", dp[N-1]);
}
```

- **Complexity?** We note our query points, $p[j] = x[j]$ are increasing in j. So we can use the $O(1)$ amortized CHT. Then the complexity is just $O(N)$.
- Since our query points are integers, we can also use the version of CHT with no doubles. Using just integer division doesn't change the complexity but in practice is a large speed up (I get a 3 times speedup locally).

- This is how many CHT problems go.
  - First you come up with a normal DP that is too slow. This step is the same as in the DP lecture.
  - Then you note the recurrence is a nice formula. Write it out.
  - Now, split up the recurrence as in this example. Figure out which part corresponds to the slope and query point and how to split up the constant into the y intercept and the part added when you calculate $dp[j]$.
  - Usually this is just breaking up the terms depending on if they depend on j or i.