A1. Bumble Sort (subtask)

Algorithm

The brute force approach is to copy the code snippet from the problem statement and add a counter to keep track of the swaps performed. This answers each query in linear time, so the running time is $O(nq)$ which is too slow for $n, q \leq 100,000$.

Instead, we need to make use of the fact that the array is already sorted.

- Any array elements smaller than or equal to $x$ do not result in a swap.
- Every array element strictly greater than $x$ results in a swap.

Therefore, we need to identify the first index $i$ where $a[i] > x$ (using `upper_bound`) and do some pointer arithmetic to recover the answer. This runs in $O(n + q \log n)$, which is sufficient for $n, q \leq 100,000$.

The problem can also be solved using an order statistics tree in $O((n + q) \log n)$.

There is an alternative two-pointer method which involves sorting the query values $x$, and therefore runs in $O(n + q \log q)$. This involves a bit more code, as you need to recover the original order of queries for output.

Implementation Notes

Writing your own binary search can lead to bugs. Use `upper_bound` instead.

Reference Solution

```cpp
// Solution by Aaveen, using upper_bound
#include <algorithm>
#include <iostream>
using namespace std;

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

int main (void) {
    int n, q;
    cin >> n >> q;
    for (int i = 0; i < n; i++)
        cin >> a[i];

    for (int i = 0; i < q; i++) {
        int x;
        cin >> x;
        cout << (a+n) - upper_bound(a, a+n, x) << '\n';
    }
}
```
A2. Bumble Sort (full)

Algorithm

The only change from the subtask to the full problem is that duplicate array elements are allowed. As before, any array elements smaller than or equal to \( x \) do not result in a swap. However, not every array element strictly greater than \( x \) results in a swap. In fact, we only get one swap for each distinct value greater than \( x \). Thus, we should preprocess the array by ignoring duplicates, and then we proceed as in the subtask.

Implementation Notes

You can use `std::unique` to remove duplicates, but it is probably simpler to write it yourself.

After removing duplicates, the array might no longer be of length \( n \). Make sure your `upper_bound` is taken over the correct range.

Reference Solutions

```cpp
// Solution by Raveen, using upper_bound
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main ( void ) {
  int n, q;
  cin >> n >> q;
  vector<int> a;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    if ((a.empty()) || x > a.back())
      a.push_back(x);
  }
  for (int i = 0; i < q; i++) {
    int x;
    cin >> x;
    cout << a.end() - upper_bound(a.begin(), a.end(), x) << 'n';
  }
}

// Solution by a student, using order statistics tree
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, greater<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

int main ( void ) {
  int n, q;
  cin >> n >> q;
  ordered_set a;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    a.insert(x);
  }
  for (int i = 0; i < q; i++) {
    int x;
    cin >> x;
    cout << a.order_of_key(x) << 'n';
  }
}
```
B1. Arcade (subtask)

Algorithm

You might briefly consider greedy approaches such as:

- get to the next special level of either game, choosing whichever takes less time, and spending any ‘excess’ time on the other game.

However, with a bit of thought you can construct counterexamples, where in fact you had to look quite far ahead in order to decide which game to start with.

One such example is

\[
\begin{array}{ccc}
3 & 3 & 8 \\
1 & 1 & 1 \\
7 & 1 & 1 \\
7 & 1 & 1 \\
2 & 1 & 1 \\
5 & 1 & 0 \\
5 & 1 & 0 \\
\end{array}
\]

where the only way to win all six tickets is to start on Pac-Man rather than Frogger.

This should then lead us to consider dynamic programming. Indeed, the problem is somewhat reminiscent of 0–1 Knapsack.

Subproblems: For \(0 \leq i \leq n\), \(0 \leq j \leq m\) and \(0 \leq t \leq k\), let \(f(i, j, t)\) represent whether it is possible to have \(t\) minutes remaining after playing \(i\) levels of Frogger and \(j\) levels of Pac-Man.

Recurrence: Here we must take cases.

For \(t = k\), we need to have just completed a special level. Thus \(f(i, j, t)\) is true if:

- Frogger level \(i\) is special and \(f(i - 1, j, z)\) is true for some \(z \geq t_i\), or
- Pac-Man level \(j\) is special and \(f(i, j - 1, z)\) is true for some \(z \geq t_{n+j}\).

On the other hand, for \(t < k\), we need to have not just completed a special level. Thus \(f(i, j, t)\) is true if:

- Frogger level \(i\) is not special, \(t + t_i \leq k\) and \(f(i - 1, j, t + t_i)\) is true, or
- Pac-Man level \(j\) is not special, \(t + t_{n+j} \leq k\) and \(f(i, j - 1, t + t_{n+j})\) is true.

Base case: \(f(0, 0, t)\) is true if \(t = k\) and false otherwise.

Then, the final answer is recovered by considering all triples \(i, j, t\) such that \(f(i, j, t)\) is true, and accumulating the tickets earned on \(i\) levels of Frogger and \(j\) levels of Pac-Man in constant time using prefix sums.

There are \(O(nmk)\) subproblems. One in every \(k + 1\) of these is solved in \(O(k)\) time, and the rest are solved in constant time, so the average time per subproblem is \(O(1)\). The final answer calculation takes \(O(nmk)\). This is sufficient for \(n, m, k \leq 300\).

However, the \(O(k)\) lookup for all times \(\geq t_i\) or \(\geq t_{n+j}\) might motivate us to consider an alternative solution. What if we instead computed this directly?

Subproblems: For \(0 \leq i \leq n\), \(0 \leq j \leq m\) and \(0 \leq t \leq k\), let \(g(i, j, t)\) represent whether it is possible to have at least \(t\) minutes remaining after playing \(i\) levels of Frogger and \(j\) levels of Pac-Man.

Recurrence: Again we take cases.

For \(t = k\), we need to have just completed a special level. Thus \(g(i, j, t)\) is true if:

- Frogger level \(i\) is special and \(g(i - 1, j, t_i)\) is true, or
- Pac-Man level \(j\) is special and \(g(i, j - 1, t_{n+j})\) is true.
On the other hand, for \( t < k \), we could have just completed a special level or not. Thus \( g(i, j, t) \) is true if:

- \( g(i, j, k) \) is true,
- Frogger level \( i \) is not special, \( t + t_i \leq k \) and \( g(i - 1, j, t + t_i) \) is true, or
- Pac-Man level \( j \) is not special, \( t + t_{n+j} \leq k \) and \( g(i, j - 1, t + t_{n+j}) \) is true.

**Base case:** \( g(0,0,t) \) is true for all \( t \).

Then, the final answer is recovered by considering all pairs \( i, j \) such that \( g(i, j, 0) \) is true, and accumulating the tickets earned on \( i \) levels of Frogger and \( j \) levels of Pac-Man in constant time using prefix sums.

There are \( O(nmk) \) subproblems, each solved in constant time, and the final answer calculation takes \( O(nm) \). This is fast enough for \( n, m, k \leq 300 \).

**Implementation Notes**

In the second method, note that \( g(i, j, t) \) depends on \( g(i, j, k) \) for all \( t < k \). Therefore we must solve for \( t = k \) before solving for \( t < k \).

**Reference Solution**

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

const int N=330;
int n, m, k;
// dp[i][j][t] is whether it's possible to have exactly t minutes remaining
// after i Frogger levels and j Pac-Man levels
bool dp[N][N][N];
int at[N], bt[N]; // time per level
int ap[N], bp[N]; // prefix sum of tickets
bool as[N], bs[N]; // special

int main() {
    cin >> n >> m >> k;
    for (int i = 1; i <= n; i++) {
        cin >> at[i] >> ap[i] >> as[i];
        ap[i] += ap[i-1];
    }
    for (int j = 1; j <= m; j++) {
        cin >> bt[j] >> bp[j] >> bs[j];
        bp[j] += bp[j-1];
    }
    dp[0][0][k]=true;

    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            for (int t = 0; t <= k; t++) {
                if (t == k) {
                    if (as[i])
                        // accept any time >= at[i]
                        for (int z = at[i]; z <= k; z++)
                            if (dp[i-1][j][z]) {
                                dp[i][j][t] = true;
                                break;
                            }  
                }
                if (bs[j])
                    // accept any time >= bt[j]
                    for (int z = bt[j]; z <= k; z++)
                        if (dp[i][j-1][z]) {
                            dp[i][j][t] = true;
                            break;
                        }  
            } else {
                if ((i>0 && at[i] + t <= k && dp[i-1][j][at[i] + t]) ||
                    (j>0 && bt[j] + t <= k && dp[i][j-1][bt[j] + t]))
                    dp[i][j][t] = true;
            }
    return 0;
}
```

4
```cpp
// Solution by Raveen
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 330;
int n, m, k;
// dp[i][j][t] is whether it's possible to have *at least* t minutes remaining
// after i Frogger levels and j Pac-Man levels
bool dp[N][N][N];
int at[N], bt[N]; // time per level
int ap[N], bp[N]; // prefix sum of tickets
bool as[N], bs[N]; // special

int main() {
    cin >> n >> m >> k;
    for (int i = 0; i <= n; i++) {
        cin >> at[i] >> ap[i] >> as[i];
        ap[i] += ap[i-1];
    }
    for (int j = 0; j <= m; j++) {
        cin >> bt[j] >> bp[j] >> bs[j];
        bp[j] += bp[j-1];
    }
    fill(dp[0][0][0], dp[0][0][0]+k+1, true); // base case
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (i || j) { // skip base case
                if ((as[i] && dp[i-1][j][at[i]]) || (bs[j] && dp[i][j-1][bt[j]]))
                    // can get here from a special level
                    fill(dp[i][j], dp[i][j]+k+1, true);
                else
                    for (int t = 0; t <= k; t++) {
                        dp[i][j][t] = (i > 0 && !as[i] &&
                                       t + at[i] <= k &&
                                       dp[i-1][j][t+at[i]] ||
                                       (j > 0 && !bs[j] &&
                                        t + bt[j] <= k &&
                                        dp[i][j-1][t+bt[j]]);
                    }
            }
    int ans = 0;
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (dp[i][j][0])
                ans = max(ans, ap[i] + bp[j]);
    cout << ans << 'n';
}
```

5
B2. Arcade (full)

Algorithm

For $k \leq 10^9$, no optimisation of the recurrence will save us; we have to reduce the state space. Thus far, for every pair $i, j$ we are calculating $g(i, j, t)$ for each $t$. But $g(i, j, t + 1) \implies g(i, j, t)$, so these values are true for $t \leq t^*$ and false thereafter. Let's just find $t^*$ directly!

Subproblems: For $0 \leq i \leq n$ and $0 \leq j \leq m$, let $h(i, j)$ be the maximum possible time remaining after playing $i$ levels of Frogger and $j$ levels of Pac-Man.

Recurrence: We take cases on whether we were able to complete the previous level of each game.

- If $h(i - 1, j) \geq t_i$, we can reach this state immediately after playing level $i$ of Frogger. The time remaining will be $k$ if $s_i$ is true, or $h(i - 1, j) - t_i$ otherwise.

- If $h(i, j - 1) \geq t_{n+j}$, we can reach this state immediately after playing level $j$ of Frogger. The time remaining will be $k$ if $s_{n+j}$ is true, or $h(i, j - 1) - t_{n+j}$ otherwise.

If both apply, we take the maximum of the two candidate answers. If neither applies, then we set $h(i, j) = -1$ to denote that it is impossible to reach this state.

Base case: $h(0, 0) = k$.

Then, the final answer is recovered by considering all pairs $i, j$ such that $h(i, j) \geq 0$, and accumulating the tickets earned on $i$ levels of Frogger and $j$ levels of Pac-Man in constant time using prefix sums.

There are $O(nm)$ subproblems, each solved in constant time, and the final answer calculation takes $O(nm)$. This is fast enough for $n, m \leq 1,000$.

Implementation Notes

Each level can give $10^9$ tickets, so the answer may not fit in a 32-bit integer. Therefore, you should use `long long` for the prefix sums and the final answer.

Reference Solution

```cpp
// Solution by Raveen
#include <algorithm>
#include <iostream>
using namespace std;
typedef long long ll;
const int N = 1010;
int n, m, k;

// dp[i][j] is max time remaining after i Frogger levels and j Pac-Man levels
int dp[N][N];
int at[N], bt[N]; // time per level
ll ap[N], bp[N]; // prefix sum of tickets
bool as[N], bs[N]; // special

int main() { 
    cin >> n >> m >> k;
    for (int i = 1; i <= n; i++) { 
        cin >> at[i] >> ap[i] >> as[i];
        ap[i] += ap[i-1];
    }
    for (int j = 1; j <= m; j++) { 
        cin >> bt[j] >> bp[j] >> bs[j];
        bp[j] += bp[j-1];
    }
    dp[0][0] = k; // base case
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
            if (i || j) { // skip base case
                dp[i][j] = -1;
                if (i > 0 && dp[i-1][j] >= at[i])
                    dp[i][j] = max(dp[i][j], dp[i-1][j]-at[i]);
            }
   ...
```
if (j > 0 && dp[i][j-1] >= bt[j])
    dp[i][j] = bs[j] ? k : max(dp[i][j], dp[i][j-1] - bt[j]);
}

ll ans = 0;
for (int i = 0; i <= n; i++)
    for (int j = 0; j <= m; j++)
        if (dp[i][j] >= 0)
            ans = max(ans, ap[i] + bp[j]);
    cout << ans << endl;
C1. Skyscrapers (subtask)

Algorithm

Finding the number of skyscrapers in the original grid naïvely takes $O(mn(m+n))$ (for every cell, check its row and column). However, we can bring this down to $O(mn)$ as follows:

- find the maximum of every row (and its column index)
- find the maximum of every column (and its row index)
- for each row maximum, if it is also the maximum of its column, increment the answer.

Then, it remains to handle updates. An update to the building at $(i,j)$ can only affect the number of skyscrapers in row $i$ and column $j$. We will handle the update by:

- uncounting any skyscrapers in row $i$ or column $j$ before the update, and
- counting any skyscrapers in row $i$ or column $j$ after the update.

Both steps can be handled by a single function. We check whether the maximum of row $i$ is also the maximum of its column, and whether the maximum of row $j$ is also the maximum of its row, taking care not to double-count when these maxima coincide at $(i,j)$. Because updates only ever increase building heights, we can maintain two arrays for the row and column maxima and their corresponding indices.

The time complexity of this solution is $O(mn)$ for the original number of skyscrapers and $O(1)$ for each update, for a total of $O(mn+p)$ which is sufficient for $m, n \leq 1,000$ and $p \leq 1,000,000$. The time limit was very high (for reasons relating to the full problem only) so a well-optimised solution taking $O(\min(m,n))$ time per query may have passed.

Implementation Notes

The uniqueness of the heights ensures that we do not have to handle ties, i.e. equal tallest buildings in a row or column.

Calculating the column maxima by sweeping through each column in turn is slow; instead, sweep row by row and update the column maxima as you go. For further details on the performance of nested loops in C/C++, refer to the tips page of the course website.

The less code you write, the less bugs you will make, so try to reuse methods where possible. Be particularly wary of copying and pasting entire code snippets. When you later find a bug in one copy, will you remember to edit the other copy accordingly?

Reference Solution

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

typedef pair<int, int> pii;

const int N = 1010;
int a[N][N];
pair<int, pii> rows[N], cols[N];
// rows[i] = (max val in row i, (i, j))
// where (i, j) is the index of that max val
const int P = 1001001;
char deltas[P];

// is $a[i][j]$ the max of its row and its column
bool isBest(int i, int j) {
    return rows[i].first == a[i][j] && cols[j].first == a[i][j];
}

int update(int i, int j) {
```
pii b1 = rows[i].second;
pii b2 = cols[j].second;

int ret = 0;
ret += isBest(b1.first, b1.second);
if (b1 != b2)
    ret += isBest(b2.first, b2.second);
return ret;
}

int main() {
    int m, n, p;
    cin >> m >> n >> p;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++) {
            cin >> a[i][j];
            rows[i] = max(rows[i], {a[i][j], {i, j}});
            cols[j] = max(cols[j], {a[i][j], {i, j}});
        }

    int ans = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            ans += isBest(i, j);
    cout << ans << \n;
    for (int q = 0; q < p; q++) {
        int del = 0;
        int i, j, v;
        cin >> i >> j >> v;
        a[i][j] = v;
        rows[i] = max(rows[i], {a[i][j], {i, j}});
        cols[j] = max(cols[j], {a[i][j], {i, j}});
        del -= update(i, j);
        if (del > 0)
            deltas[q] = '+';
        else if (del == 0)
            deltas[q] = '=';
        else
            deltas[q] = '-';
    }
    deltas[p] = '\0';
    cout << deltas << \n;
}
C2. Skyscrapers (full)

Algorithm

As above, we’d like to first count the original skyscrapers, then handle updates by uncounting and recounting. However, buildings might now decrease in height, and this poses some problems for us.

We might have to process a decrease in the height of the tallest building in a row, which might make it no longer the tallest. Since we are not tracking the second tallest building in a row, we would have to identify the new row maximum in linear time. Furthermore, keeping the tallest and second tallest building in each row only kicks the can down the road, as we will sometimes promote the third tallest building in a row to the second tallest as the result of an update.

So, we need to keep more than just the tallest building (or tallest $k$ buildings) in each row and column; we actually need to keep everything in the row and column, supporting the following operations:

- arbitrary update
- query maximum

There are two approaches, each using a different data structure.

Method I: Set

Sets keep the values in order and allow arbitrary deletion and insertion, making them a common choice for this kind of task.

For each row and each column, maintain a set of its elements paired with their indices (or equivalently a map from values to indices). Then, we can handle updates by deleting the old item and inserting the new one in logarithmic time, and query the maximum using the $\text{rbegin()}$ iterator in constant time.

This approach takes $O(mn(\log m + \log n))$ to build all the sets, and $O(\log m + \log n)$ to handle each query, for a total of $O((mn + p) \log mn)$. In principle, this should run within a second for $m,n \leq 1,000$ and $p \leq 1,000,000$. However, the constant factor in STL set operations is substantial, and as a result our judge solutions ran in about five seconds per test case.

We considered lowering the bounds, but this would have made it impossible to distinguish the intended solution from an $O(\min(m,n)q)$ solution (and indeed the latter still passed with some optimisation).

Method II: Priority Queue

The need to query the maximum naturally suggests that we should use a max heap. Heaps support arbitrary insertion, but not arbitrary deletions; we can only delete efficiently from the top of the heap.

How do we process updates? Need to keep ‘inactive’ values in the heap until they come to the top, and only delete them then. Two ways:

Store additional metadata

- In the max heap of values in a row or column, store not only the building height but also the coordinates of the building.
- Now, whenever the value at the top of the heap doesn’t match the current building height at its coordinates, delete it.

Cache deletions

- Alongside the max heap of values in a row or column, maintain another max heap of values to be deleted from it.
- Now, whenever the same value appears at the top of both heaps, delete it from both heaps.

It takes $O(mn(\log m + \log n))$ to build all the heaps (in principle this could be lowered to $O(mn)$, but it’s not the bottleneck of the algorithm). Any single update might result in several immediate deletions, but in either method we can guarantee that each building height is only inserted to and deleted from each heap once. The total time complexity is therefore $O((mn + p) \log mn)$ using amortisation.
The constant factor in STL priority queue operations is much smaller, so this solution runs considerably faster (one to two seconds per test case).

Implementation Notes
Nil.

Reference Solution

```cpp
// Solution by Angus, using set
#include <iostream>
#include <set>
#include <utility>
using namespace std;

typedef pair<int, int> pii;

const int N = 1010;
int a[N][N];
set<pair<int, pii>> rows[N], cols[N];

const int P = 1001001;
char deltas[P];

// is a[i][j] the max of its row and its column
bool isBest(int i, int j) {
    return rows[i].rbegin()->first == a[i][j] && cols[j].rbegin()->first == a[i][j];
}

int update(int i, int j, int mult) {
    pii b1 = rows[i].rbegin()->second;
    pii b2 = cols[j].rbegin()->second;

    int ret = 0;
    ret += isBest(b1.first, b1.second);
    if (b1 != b2)
        ret += isBest(b2.first, b2.second);
    return ret;
}

int main() {
    int m, n, p;
    cin >> m >> n >> p;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++) {
            cin >> a[i][j];
            rows[i].insert({a[i][j], {i, j}});
            cols[j].insert({a[i][j], {i, j}});
        }

    int ans = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            ans += isBest(i, j);
    cout << ans << 'n';

    for (int q = 0; q < p; q++) {
        int del = 0;
        int i, j, v;
        cin >> i >> j >> v;
        del += update(i, j);
        rows[i].erase({a[i][j], {i, j}});
        cols[j].erase({a[i][j], {i, j}});
        a[i][j] = v;
        rows[i].insert({a[i][j], {i, j}});
        cols[j].insert({a[i][j], {i, j}});
        del += update(i, j);
        if (del > 0)
            deltas[q] = '+';
        else if (del == 0)
            deltas[q] = '=';
    }
```
```cpp
else
    deltas[q] = '+';
}
deltas[p] = '\0';
cout << deltas << '\n';
}

// Solution by Angus, using priority queue
#include <iostream>
#include <queue>
#include <utility>
using namespace std;

typedef pair<int, int> pii;

const int N = 1010;
int a[N][N];
priority_queue<pair<int, pii>> rows[N], cols[N];

const int P = 1001001;
char deltas[P];
pair<int, pii> getTop(priority_queue<pair<int, pii>> *arr, int i) {
    while (a[arr[i].top().second.first][arr[i].top().second.second] != arr[i].top().first)
        arr[i].pop();
    return arr[i].top();
}

// is a[i][j] the max of its row and its column
bool isBest(int i, int j) {
    return getTop(rows, i).first == a[i][j] && getTop(cols, j).first == a[i][j];
}

int update(int i, int j) {
    pii b1 = getTop(rows, i).second;
    pii b2 = getTop(cols, j).second;

    int ret = 0;
    ret += isBest(b1.first, b1.second);
    if (b1 != b2)
        ret += isBest(b2.first, b2.second);
    return ret;
}

int main() {
    int m, n, p;
    cin >> m >> n >> p;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++) {
            cin >> a[i][j];
            rows[i].push({a[i][j], {i, j}});
            cols[j].push({a[i][j], {i, j}});
        }

    int ans = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            ans += isBest(i, j);
    cout << ans << '\n';

    for (int q = 0; q < p; q++) {
        int del = 0;
        int i, j, v;
        cin >> i >> j >> v;
        del += update(i, j);
        a[i][j] = v;
        rows[i].push({a[i][j], {i, j}});
        cols[j].push({a[i][j], {i, j}});
        del += update(i, j);
        if (del > 0)
            deltas[q] = '+';
    }
}
```
else if (del == 0)
    deltas[q] = '=';
else
    deltas[q] = '-';
}
deltas[p] = '\0';
cout << deltas << '\n';