WOBURN CHALLENGE

2015-16 On-Site Finals

Solutions

Automated grading is available for these problems at:
wcipeg.com

For problems to this contest and past contests, visit:
woburnchallenge.com

Problem J1: Vertical Posting

We are asked to display the input string in the format specified. Note that the string is a single word with no spaces, so most programming languages will have no trouble with input. To display the horizontal section of the posting, we should print the input string with a single space between adjacent letters. This can be done with a loop through the indices of the string, and printing a space before printing each character. We should neglect to print a space only for the first iteration of the loop. Alternatively, we can print a space *after* printing each letter (except for the last). Then, we can print a newline character to start the vertical part of the posting. For the vertical part, we loop through the string once again, except this time starting at the *second* letter. For each letter printed, we also must print a newline character afterwards.

Official Solution (C++)

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

string s;

int main() {
  cin >> s;
  for (int i = 0; i < (int)s.length(); i++) {
    if (i > 0) cout << " ";
    cout << s[i];
  }
  cout << endl;
  for (int i = 1; i < (int)s.length(); i++)
    cout << s[i] << endl;
  return 0;
}
```

Problem J2: The Oxford Comma

The most important observation is that each sentence can contain exactly one occurrence of the word “and”, one occurrence of the word “or”, or no occurrence of either word (but it *cannot* contain both “and” and “or”). If that were not the case, then the specifications would be much more confusing. We also notice from reading the question carefully that we are only to add an Oxford comma if a list has 3 or more entries, with an “and”/“or” strictly preceding the last entry. In light of this, notice that we’re simply looking for the pattern “a, b and c” and “a, b or c” where a, b, and c are any words. This is sufficient, because any list with more than 3 entries will contain this pattern as well. So, we loop through the words and for every and/or, check if the previous-previous word is followed by a comma, and whether the previous word is (if not, then add one). To make sure that we’re not adding a comma to the pattern “a, b. and c”, we should also check to make sure the previous word is not followed by a period. We do not have to check whether “and”/“or” is followed by another entry, because if “and” is ever at the end of a sentence, then it would be followed by a period (and so testing equality with the strings “and”/ “or” would return false).

Official Solution (C++)

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

int N;
string s[1005];

bool comma(string s)
{ return s[s.size() - 1] == ','; }

bool period(string s)
{ return s[s.size() - 1] == '.'; }

int main() {
  cin >> N;
  for (int i = 0; i < N; i++)
    cin >> s[i];
  for (int i = 2; i < N; i++) {
    if (s[i] == "and" || s[i] == "or") {
      if (comma(s[i - 2]) && !comma(s[i - 1])
          && !period(s[i - 1]))
        s[i - 1] += ',';
    }
  }
  for (int i = 0; i < N; i++) {
    if (i > 0) cout << " ";
    cout << s[i];
  }
  return 0;
}
```

Problem J3/S1: Fuzzbiz

Let $V[i]$ be true if i is a valid number at which the given sequence of M words could've started at (in other words, if i could've been the M-th last number in the game), and false otherwise. For a given i, we can determine the value of $V[i]$ in $O(M)$ time with simulation by testing if, for each j from 1 to M, the j-th words are the correct words for number $i + j - 1$.

The simulation is very similar to naively searching through a string, where all N rounds of the game form a "haystack" to be searched through and the M words is the "needle" of which we must count occurrences.

To test all $N - M + 1$ possible starting indices, the overall running time is $O((N - M + 1)\times M) = O(NM)$ since N is significantly larger than M. However, this will not pass since N is as big as 10^9. Even using faster string searching algorithms such as the KMP or Aho-Corasick algorithms will not pass, as their time complexities all depend on N.

To get full marks, we will have to eliminate the bottleneck of generating the words of all N rounds of the game. The trick is to notice that in a perfect game of FizzBuzzOok, the sequence of words repeats after every 15 terms. This also means that V repeats every 15 terms – that is, if the sequence could have started at number i, it could have also started at number $i + 15\times j$ (for any non-negative integer j). Therefore, we only need to compute $V[1..15]$ using the above approach (in a total of $O(M)$ time), and can then observe that $V[16] = V[1]$, $V[17] = V[2]$, and so on.

Official Solution (C++)

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

int N, M, ans = 0;
string s[100005];

int main() {
  cin >> N >> M;
  for (int i = 0; i < M; i++) {
    cin >> s[i];
  }
  for (int i = 1; i <= 15; i++) {
    bool good = true;
    for (int j = 0; good && j < M; j++) {
      if ((i + j) % 15 == 0) {
        good &= (s[j] == "fizzbuzz");
      } else if ((i + j) % 3 == 0) {
        good &= (s[j] == "fizz");
      } else if ((i + j) % 5 == 0) {
        good &= (s[j] == "buzz");
      } else {
        good &= (s[j] == "ook");
      }
    }
    if (good) {
      ans += (N - M - i + 16)/15;
    }
  }
  cout << ans << endl;
  return 0;
}
```

To compute the answer, then, we should consider each value of i between 1 and 15 for which $V[i]$ = true, and count the number of non-negative integers j such that $i + 15\times j \le N - M + 1$ (which is the largest number that could've been the first word in the given sequence). Using some algebra to rearrange this inequality, we get $15\times j \le N - M - i + 1$, which ultimately implies that $j \le (N - M - i + 1)/15$.

Assuming the left-hand side formula is non-negative, then the number of possible non-negative integers j is equal to the expression rounded down, plus 1 to account for 0 being a valid value of j. If it happens that the left-hand side expression is negative, then we should report 0 as the answer. These two cases can be merged into the expressions $\text{ceil}((N - M - i + 2)/15) = \text{floor}((N - M - i + 16)/15)$. The official solution which implements this idea is given above, and has a running time of $O(M)$.

Problem J4/S2: Hydration

If it's possible for the cows to all get a drink, then it must be doable in at most N minutes. If it's possible for them to finish drinking in m minutes, then it must be possible for them to finish in $m + 1$ minutes. This means that we can binary search for the answer on the [1, $N + 1$], and if it's impossible in even N minutes, we'll get a result of $N + 1$ and can output –1 instead.

What remains is being able to determine whether or not the cows can all have a drink within m minutes. In particular, we would like to know if there exists a way to assign the cows to the troughs such that each trough is assigned at most m cows, and that each cow i's assigned trough j satisfies $C_i - K \le T_j \le C_i$.

If cow a is shorter than cow b, then intuitively, we should never assign cow a to a taller trough than cow b because otherwise they could be swapped for a better configuration. This idea suggests a greedy algorithm.

Assuming that the heights of the cows and troughs have both been sorted in non-decreasing order (respectively doable in time complexities $O(N \log N)$ and $O(M \log M)$), we can iterate over each cow i in order, assigning it to the first trough j that it can validly drink from (if any). In other words, we're looking for the smallest j such that $C_i - K \le T_j \le C_i$, and fewer than m cows have been assigned to drink from trough j so far.

As we iterate i from 1 to N, note that j will never decrease. Therefore, we only have to keep track of the current value of j and the number of cows c that have been assigned to trough j, for each cow i, we'll first increase j as long as it's invalid (resetting c to be 0 whenever j is increased), and then assign cow i to it (increasing c by 1). If j exceeds M, then we'll know that there was no valid trough, meaning that not all of the cows can drink within m minutes. The total time complexity of this algorithm is therefore $O((N + M) \log N + M \log M)$.

Official Solution (C++)

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

int N, M, K, C[1000005], T[1000005];

bool check(int x) {
  int j = 0, c = 0;
  for (int i = 0; i < N; i++) {
    while (j < M &&
           (c == x || T[j] < C[i] - K || T[j] > C[i])) {
      j++;
      c = 0;
    }
    if (j == M) return true;
    c++;
  }
  return j == M;
}

int main() {
  cin >> N >> M >> K;
  for (int i = 0; i < N; i++)
    cin >> C[i];
  for (int i = 0; i < M; i++)
    cin >> T[i];
  sort(C, C + N);
  sort(T, T + M);
  int lo = 1, hi = N + 1;
  while (lo < hi) {
    int mid = (lo + hi)/2;
    if (check(mid))
      lo = mid + 1;
    else
      hi = mid;
  }
  if (hi > N) hi = -1;
  cout << hi << endl;
  return 0;
}
```

Problem J5/S3: Driving Range

Let's consider a single instance of Tiny completing the course for now. Let's say that there are NR targets at the ends of rows $R[1..NR]$, and NC targets at the ends of columns $C[1..NC]$, with both arrays R and C sorted in increasing order. It's easy to imagine that it's never optimal for Tiny to drive North or West – instead, he should only move South and East, hitting targets as he goes, in some order. This implies that he'll hit the R row targets in increasing order and the C column targets also in increasing order, with the only question being how they should be interleaved.

This observation suggests a naive recursive solution which tries all possible interleaved orders of targets. We should keep track of how many row and column targets Tiny has hit, and his current row and column. At each step, Tiny can either hit the next row target, if any (by driving South to its row, driving 1 unit East, and firing a missile), or hit the next column target, if any. This recursive process takes $O(2^{NR + NC})$ time, yielding an $O(N \times 2^N)$ solution in total.

The above algorithm can be improved with the use of dynamic programming. The recursion can be directly memoized to improve its running time to $O(NR \times NC \times R[NR] \times C[NC])$, which is the number of possible states (with at most two transitions from each state). This yields an $O(N^3 \times K^2)$ solution in total, where K is the largest P_i value of any target.

To achieve a dramatically more efficient solution, a different approach to the problem will be required. Before we proceed any further, we should observe that a target in column 1 is special in that it's the only target that can be hit without needing to rotate to face it first – as such, if there's a target in column 1, let's simply remove it from the C array (also decrementing NC by 1), and add 1 to the answer (as the first move in this case will clearly consist of firing a missile downwards at it).

Now, let's consider counting how many of each type of move Tiny will need to perform. Clearly, he'll fire exactly $NR + NC$ missiles in total. Additionally, before firing at each target, he'll need to rotate to face it at some point after entering its row/column – this implies needing to drive East a minimum of NR times, and South a minimum of NC times. Besides these $2(NR + NC)$ required moves, it's a fact that Tiny will need to drive South at least $R[NR] – 1$ times in total to reach the last row target's row (or 0 times if $R = 0$), and similarly East at least $C[NC] – 1$ times in total. However, the answer can be smaller than $2(NR + NC) + R[NR] + C[NC] – 2$, because some of the $NR + NC$ moves required for rotation can also count towards the $R[NR] + C[NC] – 2$ moves required for reaching the last row/column!

Let's assume that the final target that Tiny will hit will be in a row (we'll also want to consider the case in which he hits a column target last, and take the minimum of the two resulting answers). In this case, each of the NC times that Tiny moves South to rotate himself to hit a column target will also move him towards row $R[NR]$. Therefore, he'll have to drive South exactly $NC + \max(R[NR] – NC – 1, 0)$ times. On the other hand, each of the NR times that Tiny moves East to rotate himself to hit a row target will also move him towards column $C[NC]$ except for the last time (because he will already have hit the final column target by then). Therefore, he'll have to drive East exactly $NR + \max(C[NC] – NR – 2, 0)$ times.

To summarize the above thoughts, the minimum number of moves required to complete the course is $2(NR + NC) + \min\{\max(R[NR] – NC – 1, 0) + \max(C[NC] – NR – 2, 0), \max(R[NR] – NC – 2, 0) + \max(C[NC] – NR – 1, 0)\}$, plus 1 if there's a target in the 1st column (which we removed from the C array). As such, as targets are added one at a time, all we need to do is maintain the values NR, NC, $R[NR]$ (simply the max target row so far, initialized at 0), $C[NC]$ (the max target column so far), and a flag for whether or not a target exists in the 1st column. The time complexity of this is $O(N)$.

Official Solution (C++)

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

int N, p, NR, NC, MR, MC, c = 0;
char d;

int main() {
  NR = NC = MR = MC = 0;
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> d >> p;
    if (d == 'R') {
      NR++;
      MR = max(MR, p);
    } else if (p > 1) {
      NC++;
      MC = max(MC, p);
    }
    c++;
    cout << (c + NR + NC +
              min(max(MR - NC, 0) + max(MC - NR - 1, 0),
                  max(MR - NC - 1, 0) + max(MC - NR, 0))) << endl;
  }
  return 0;
}
```

Problem S4: Server Hacking

For starters, we need to be able to compare a pair of public keys and determine which one is weaker. This can be done by prime factorizing each one to compute its private key, and then comparing the private keys by the earliest index at which they differ. The process can be simplified and sped up in the average case by prime factorizing both values simultaneously, stopping as soon one of their private keys has been found to be lexicographically smaller than the other, though this is not necessary.

Prime factorizing a single integer x is a standard procedure which can be accomplished in $O(\sqrt{x})$ time by looping a variable p up from 2 to the square root of x. At each iteration, if x is divisible by p, then p must be a prime factor of x, at which point we can repeatedly divide x by p as long as it's still divisible by it in order to determine p's exponent in the factorization. Once p has exceeded the square root of x, if the final value of x is still larger than 1, it must be a prime factor itself (with an exponent of 1).

A naive solution consists of testing if each computer is a valid weakpoint, until one is found. In the worst case, this requires comparing every pair of adjacent computers, which requires prime factorizing all N public keys at least once. As such, the time complexity is $O(N\sqrt{K})$, where K is the largest A_i value of any computer. Since $N \le 100{,}000$ and $K \le 10^9$ (so $\sqrt{K}$ is roughly less than 32000), the worst case number of steps taken will be $100{,}000 \times 32{,}000 = 3.2$ billion, which is too slow. The $\sqrt{K}$ factor can be optimized through advanced prime factorization techniques such as the Sieve of Eratosthenes, Fermat's method, or Pollard's rho algorithm. However, these techniques can be dubious to depend on during a live contest. To achieve a better time complexity, we can instead try to optimize the N factor – in other words, find a way to somehow not have to prime factorize every public key.

We can formulate this problem in another way. If we let $F(i)$ be the private key of the i-th computer, then we're looking for any local minimum of the function F. Despite the fact that F is only defined at integer values, we can use a technique common for finding local minima of continuous curves – namely, ternary search. At all points in the ternary search, we'll maintain the invariant that at least one local minimum must exist in the range of indices $a..b$. Initially, $a = 1$ and $b = N$, and we'll stop once $a = b$. Let's now compare the values of the function F one-third and two-thirds between a and b – specifically, if we let $x = \text{floor}((a + a + b) / 3)$ and $y = \text{ceil}((a + b + b) / 3) = \text{floor}((a + b + b + 2) / 3)$, we want to compare $F(x)$ to $F(y)$ (x is rounded down while y is rounded up so that, when $b = a + 1$, we'll have $x = a$ and $y = b$). Now, if $F(x) < F(y)$, then there must be at least one local minimum in the range of indices $a…(y – 1)$, so we can set $b = y – 1$. Otherwise, if $F(x) > F(y)$, we can set $a = x + 1$. At each iteration of this ternary search, the size of the range $a…b$ reduces by approximately 1/3, meaning that it will be reduced to size one after $O(\log_3 N)$ iterations. Therefore, the time complexity of this algorithm is $O(\log(N) \times \sqrt{K})$, which is sufficient to pass.

The following solution implements this idea, applying the slightly more optimized method to compare weaknesses of the keys.

Solution – Ternary Search (C++)

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

int N, A[100001];

bool weaker(int v1, int v2) {
  for (int k = 2; k*k <= max(v1, v2) && v1 > 1 && v2 > 1; k++) {
    int c1, c2;
    for (c1 = 0; v1 % k == 0; c1++)
      v1 /= k;
    for (c2 = 0; v2 % k == 0; c2++)
      v2 /= k;
    if (c1 == 0 && c2 == 0) continue;
    if (c2 == 0) return true;
    if (c1 == 0) return false;
    if (c1 < c2) return true;
    if (c1 > c2) return false;
  }
  return v1 < v2;
}

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++)
    cin >> A[i];
  int lo = 1, hi = N;
  while (lo < hi) {
    int m1 = (lo + lo + hi) / 3;
    int m2 = (lo + hi + hi + 2) / 3;
    if (weaker(A[m1], A[m2]))
      hi = m2 - 1;
    else
      lo = m1 + 1;
  }
  cout << lo << endl;
  return 0;
}
```

The ternary search can also be replaced with a very similar binary search. Consider a range of the array $A[a..b]$. First we can check to make sure that neither a nor b (the endpoints) are weakpoints. If even one of them is a weakpoint, then we are done. Otherwise, we can find the midpoint $m = (a + b)/2$ to break the interval in half. There are two cases:

$F(m) < F(m + 1)$: `\.../.../` – in which we should recursively examine the left subinterval $[a, m]$
$F(m) > F(m + 1)$: `\...\.../` – in which we should recursively examine the right subinterval $[m + 1, b]$

Eventually, we must have found a local minimum when the length of the interval is reduced to 1. In fact, a careful implementation like the following will render it unnecessary to check the endpoints. The official solution below uses this idea, generating the private keys explicitly into an std::vector and comparing them with the built-in relational operator (which compares using lexicographical_compare as intended). The time complexity of this algorithm is also $O(\log(N) \times \sqrt{K})$, which is faster than the previous ternary search solution by only a constant factor.

Official Solution (C++)

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

int N, A[100001];

vector<int> factorize(int n) {
  vector<int> res;
  for (int i = 2; i*i <= n; i++) {
    if (n % i == 0) {
      int cnt;
      for (cnt = 0; n % i == 0; cnt++)
        n /= i;
      res.push_back(i);
      res.push_back(cnt);
    }
  }
  if (n != 1) {
    res.push_back(n);
    res.push_back(1);
  }
  return res;
}

int rec(int lo, int hi) {
  if (lo == hi) return lo;
  int m = (lo + hi)/2;
  if (factorize(A[m]) < factorize(A[m + 1]))
    return rec(lo, m);
  return rec(m + 1, hi);
}

int main() {
  cin >> N;
  for (int i = 1; i <= N; i++)
    cin >> A[i];
  cout << rec(1, N) << endl;
  return 0;
}
```

Problem S5: Supply Chain

One major step in solving this problem is the reduction to solving it for a line of pastures (starting with pasture 1) rather than a cycle, which is much more manageable. Let's consider having two lines of N pastures each. The first line will consist of pastures 1, 2, ..., $N - 1$, N (made by removing bridge N), while the second will consist of pastures 1, N, ..., 3, 2 (made by removing bridge 1). If we compute the number of bananas delivered in each line and add these two values together, it may exceed the number of bananas delivered in the original cycle – but by how much? The only bananas which will be counted twice will be all of those delivered by trucks which are light enough to drive around the entire cycle in either direction - namely, the trucks whose weights are no larger than the minimum weight S_i supported by any bridge i. Therefore, if we can maintain this minimum value (which is easy) and be able to query the sum of the B values of trucks whose W values are no larger than it (which will be doable), we can go ahead and solve the problem for both of these lines independently, and then combine their answers!

Now that we're dealing with a line (let's say the first line, consisting of pastures 1 to N in order), let $P[i]$ be the maximum truck weight that would be capable of reaching each pasture i. We have $P[1]$ = infinity, while $\mathrm{P}[i] = \min(P[i - 1], S[i - 1])$ for $2 \le i \le N$. Note that $P[1..N]$ is a non-increasing sequence. The number of bananas delivered to pasture i is then the sum of the B values of trucks whose W values are no larger than $P[i]$. Handling each day independently, we can compute the P array in $O(N)$ time, and then compute the total number of bananas delivered on it efficiently in various ways. For example, we can consider each truck i in turn, use binary search to find the last pasture j that truck i can reach (i.e., the largest j such that $P[j] \ge W[i]$), and add $B[i]\times(j - 1)$ to our result. After computing the results for both lines, we must remember to subtract the double-counted bananas, which can be done by summing the bananas delivered by light-enough trucks in $O(M)$ time. This yields a solution with a running time of $O(D\times(N + M \log N))$. Similar $O(D\times(M + N \log M))$ and $O(D\times(N + M))$ solutions are possible.

In order to optimize this approach, we'll need to be able to update the answer from day to day without re-computing it from scratch, which will require a couple of data structures to maintain and update useful information about the trucks and pastures.

Firstly, for the trucks, it seems useful to be able to query the sum of the B values of trucks whose W values are no larger than some given value (for example, this is exactly what's required to compute the number of double-counted bananas each day). Fortunately, there exists a simple data structure which does exactly that – a binary indexed tree (also known as a Fenwick tree). We'll initialize the tree with each truck i by adding $B[i]$ to index $W[i]$, and each time a truck i's weight changes from w to w', we'll subtract $B[i]$ from index w and add $B[i]$ to index w'. We'll be querying this tree for the sum of the B values of trucks whose W values are no larger than some weight x – let's call this sum $F(x)$. Each update and query on this tree will take $O(\log L)$ time, where L is the maximum possible truck weight (10^6).

Secondly, for the pastures in a line, we'll want a data structure to represent their corresponding P array. Let's consider only the set of K "interesting" indices P' at which P changes (decreases), plus a sentinel value of $P'[K] = N + 1$. We then have an increasing list $P'[1..K]$ such that each interval of indices $P'[i]\ldots(P'[i + 1] - 1)$ has a uniform P value, which means that each pasture in that interval will receive the same number of bananas (namely, $F(P[P'[i]])$ of them). To allow this list to be updated as necessary, we'll want to store it as a set (a balanced binary search tree) of pairs (i, $P[i]$), rather than maintaining the actual arrays P and P'. This set is initialized in $O(N \log N)$ by iterating over the pastures in order and inserting each one which is preceded by a bridge with a new minimum weight limit. Each time a bridge i's weight limit is reduced to w, this set can be updated in amortized $O(\log N)$ by potentially adding a point (i, w), and deleting existing points (j, w') where $j \ge i$ and $w' \ge w$.

Now that both of these data structures are in place, let's consider how to use them to update each line's answer after each event. When a truck i's weight changes from w to w', we should determine how many pastures c are reachable by trucks with weight w and how many pastures c' are reachable by trucks with weight w', and increase the answer by $B[i]\times(c' - c)$. Each of these values can be looked up in our set in $O(\log^2 N)$ time using binary search, or even in $O(\log N)$ time if we maintain an accompanying set of points searchable by P value (in other words a set of pairs $(P[i], i)$ in addition to the set of pairs $(i, P[i])$). When a bridge's supported weight decreases and we insert and/or delete points from our set, some intervals of pastures with equal P values (that is, the intervals between adjacent indices in the set) will change, and we can update the answer accordingly. For example, if an interval of k pastures all have their P values reduced from p to p', we should decrease the answer by $F(p) - F(p')$. An amortized constant number of pasture intervals will change as a result of each such event, meaning that it can be handled in amortized $O(\log L)$ time. The total time complexity of this algorithm is $O(M \log L + N \log N + D\times(\log N + \log L))$, where L is again the maximum possible truck weight.

Official Solution (C++)

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

const int MAXM = 300005, SIZE = 1000005, INF = 0x3f3f3f3f;

int N, M, E, t, x, y, R[2][MAXM], W[MAXM], B[MAXM];
long long BIT[SIZE], ans = 0;
set< pair<int, int> > S[2], S2[2];

void update(int i, int v) {
  for (; i < SIZE; i += (i & -i))
    BIT[i] += v;
}

long long query(int i) {
  long long res = 0;
  for (; i > 0; i -= (i & -i))
    res += BIT[i];
  return res;
}

long long overlap() {
  int m = min(S[0].rbegin()->second, S[1].rbegin()->second);
  return query(m) * (N - 1);
}

void insert(int i, pair<int, int> p) {
  int j;
  set< pair<int, int> >::iterator it = S[i].lower_bound(p);
  if (it == S[i].end()) {
    j = N - 1;
  } else {
    j = it->first;
  }
  ans += query(p.second) * (j - p.first);
  S[i].insert(p);
  S2[i].insert(make_pair(-p.second, p.first));
}

void erase(int i, set< pair<int, int> >::iterator it) {
  int j;
  set< pair<int, int> >::iterator it2;
  it2 = it;
  if (++it2 == S[i].end()) {
    j = N - 1;
  } else {
    j = it2->first;
  }
  ans -= query(it->second) * (j - it->first);
  S[i].erase(it);
  S2[i].erase(make_pair(-it->second, it->first));
}
```

```
void updateR(int i, int j, int v) {
  pair<int, int> p;
  set< pair<int, int> >::iterator it, it2;
  R[i][j] -= v;
  v = R[i][j];
  it = S[i].lower_bound(make_pair(j + 1, 0));
  --it;
  if (v >= it->second) return;
  p = *it;
  it2 = it;
  ++it;
  erase(i, it2);
  while (it != S[i].end() && v <= it->second) {
    it2 = it;
    ++it;
    erase(i, it2);
  }
  insert(i, make_pair(j, v));
  if (j > p.first) insert(i, p);
}

int count(int i, int w) {
  set< pair<int, int> >::iterator it;
  it = S2[i].lower_bound(make_pair(-w, INF));
  if (it != S2[i].end()) return it->second;
  return N - 1;
}

void updateT(int i, int w) {
  for (int j = 0; j < 2; j++)
    ans -= (long long)B[i] * count(j, W[i]);
  update(W[i], -B[i]);
  W[i] = w;
  update(W[i], B[i]);
  for (int j = 0; j < 2; j++)
    ans += (long long)B[i] * count(j, W[i]);
}

int main() {
  cin >> N >> M >> E;
  for (int i = 0; i < N; i++) {
    cin >> R[0][i];
    R[1][N - i - 1] = R[0][i];
  }
  for (int i = 0; i < M; i++) {
    cin >> W[i] >> B[i];
    update(W[i], B[i]);
  }
  for (int i = 0; i < 2; i++) {
    long long s;
    int m = INF;
    for (int j = 0; j < N - 1; j++) {
      if (R[i][j] < m) {
        m = R[i][j];
        S[i].insert(make_pair(j, m));
        S2[i].insert(make_pair(-m, j));
        s = query(m);
      }
      ans += s;
    }
  }
  for (int i = 0; i < E; i++) {
    cin >> t >> x >> y;  x--;
    if (t == 1) {
      updateR(0, x, y);
      updateR(1, N - x - 1, y);
    } else {
      updateT(x, y);
    }
    cout << (ans - overlap()) << endl;
  }
  return 0;
}
```