Contest 2 Editorial

COMP4128 21T3

9th October 2021

A1. Rivalry (subtask)

Algorithm

We use brute force.

First, read the string. Then, for every query, walk forward from the given start year s, counting the number of times that team t won, and output the first year when this count reaches k (or 0 if it does not reach k by the end of the string).

This algorithm runs in $O(nq)$ time, which is sufficient for $n, q \leq 5,000$.

Implementation Notes

Take care to translate correctly between 0-based indexing (the default for the string) and 1-based indexing (used in the problem).

Reference Solution

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

const int N = 100100;

int main (void) {
  int n, q;
  cin >> n >> q;

  string w;
  cin >> w;

  for (int j = 0; j < q; j++) {
    char t;
    int k, s;
    cin >> t >> k >> s;
    int cnt = 0;
    for (int i = s-1; i < n; i++)
      if (w[i] == t)
        if (++cnt == k) {
          cout << i+1 << '\n';
          break;
        }
    if (cnt != k)
      cout << "0\n";
  }
}
```

A2. Rivalry (full)

Algorithm

Let $f_t(i)$ be the number of wins by team t in the first i years. Then the answer to each query is the smallest i such that $f_t(i) = k + f_t(s-1)$.

Observation: each value $f_t(i)$ can be computed as the ith prefix sum of an array with 1s corresponding to years when team t won and 0s when they lost.

We can compute each of $O(n)$ values $f_t(i)$ in constant time. It remains to handle each query in one of two ways.

1. Since f_t is a nondecreasing function (i.e. $f_t(i+1) \geq f_t(i)$), we can use binary search to handle each query in $O(\log n)$, for a total of $O(n + q \log n)$.
2. Alternatively, we can precompute a reverse lookup table which stores for each team and each value of f_t the first year in which that value of f_t was attained. Then each query can be answered in constant time, for a total of $O(n+q)$.

Either solution is sufficient for $n, q \leq 100,000$.

Some students constructed two sets to store the winning years for each team, and used `set::lower_bound` to answer each query. This solution is also acceptable.

Implementation Notes

Be very careful when writing your own binary search - common mistakes include off-by-one errors and infinite loops. We recommend using the lecture code or `lower_bound` where possible.

Reference Solutions

```cpp
// Solution by Angus, using reverse lookup table
#include <cstdio>
#define MAXN 100010

// pre[i][0] is the number of X wins at or before day i, pre[i][1] is the number of Y
    wins at or before day o
// lookup[i][0] is the day that the ith win happened for X, lookup[i][1] is the same
    thing for Y
int n, q, pre[MAXN][2], lookup[MAXN][2];

int main() {
  scanf("%d%d", &n, &q);

  for (int i = 1; i <= n; i++) {
    char a;
    scanf(" %c", &a);
    pre[i][0] = pre[i-1][0] + (a == 'X');
    pre[i][1] = pre[i-1][1] + (a == 'Y');
    if (a == 'X')
      lookup[pre[i][0]][0] = i;
    else
      lookup[pre[i][1]][1] = i;
  }

  for (int i = 0; i < q; i++) {
    char a;
    int k, s;
    scanf(" %c%d%d", &a, &k, &s);
    k += pre[s-1][a == 'Y'];
    printf("%d\n", lookup[k][a == 'Y']);
  }
}
```

```cpp
// Solution by Raveen, using binary search
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;
```

```cpp
const int N = 100100;
int a[2][N];

int main (void) {
  int n, q;
  cin >> n >> q;

  string w;
  cin >> w;

  a[0][1] = (w[0] == 'X');
  a[1][1] = (w[0] == 'Y');
  for (int i = 1; i < n; i++) {
    a[0][i+1] = a[0][i] + (w[i] == 'X');
    a[1][i+1] = a[1][i] + (w[i] == 'Y');
  }

  for (int j = 0; j < q; j++) {
    char t;
    int k, s;
    cin >> t >> k >> s;
    int tid = (t == 'X' ? 0 : 1);
    int ans = lower_bound(&a[tid][0],&a[tid][n+1],a[tid][s-1]+k) - &a[tid][0];
    cout << (ans <= n ? ans : 0) << '\n';
  }
}
```

B1. Skyline (subtask)

Algorithm

The most naïve brute force solution runs in $O(n^3)$: for every start point ℓ, for every end point r, calculate the width $w = r - \ell + 1$ and check whether $h_i \geq w$ for each $\ell \leq i \leq r$. This is clearly too slow for $n \leq 5000$.

An improvement on this idea can be achieved by finding the smallest h_i in the range $\ell \leq i \leq r$ by querying a range tree, and comparing it to w; this reduces the complexity to $O(n^2 \log n)$ but this is still too slow (since $5000^2 \log 5000 > 300,000,000$).

Note however that there are only queries, and no updates. We can therefore use a sparse table to query the minimum over a range in constant time. The time complexity of this solution is $O(n^2)$, which is sufficient for $n \leq 5000$. Note that building the sparse table takes $O(n \log n)$ time.

An alternative solution is as follows. From every start point ℓ, we walk forwards, maintaining the height and width of a candidate rectangle. Initially, the height is h_ℓ and the width is 1. At each step, we increase the width by one and update the minimum height seen since building ℓ. The size of the square that can be formed is the minimum of the two dimensions of this candidate rectangle. This solution also runs in $O(n^2)$ time.

Implementation Notes

In the first reference solution, the lowest level of the sparse table is initialised to 100100 by default rather than 0, to ensure that `min` queries are handled correctly.

In the second reference solution, the constant factor is improved by noting that the height of the candidate rectangle never increases, so we can early exit if the height no longer exceeds the width. This optimisation was not required to obtain a passing solution.

Reference Solutions

```
// Solution by Raveen, using sparse table
#include <iostream>
using namespace std;

const int MAXN = 100100, LOGN = 18;
// sparseTable[l][i] = min a[i...i+2^l)
int n, sparseTable[LOGN][MAXN], log2s[MAXN];

void precomp() {
  for (int l = 1; l < LOGN; l++) {
    int prevp2 = 1 << (l-1);
    for (int i = 0; i < MAXN; i++) {
      int intEnd = i + prevp2;
      if (intEnd < MAXN)
        sparseTable[l][i] = min(sparseTable[l-1][i],
            sparseTable[l-1][intEnd]);
      else
        sparseTable[l][i] = sparseTable[l-1][i];
    }
  }
  for (int i = 2; i < MAXN; i++)
    log2s[i] = log2s[i/2] + 1;
}

int query (int l, int r) {
  int l2 = log2s[r-l];
  return min(sparseTable[l2][l], sparseTable[l2][r-(1<<l2)]);
}

int main() {
  for (int i = 0; i < MAXN; i++)
    sparseTable[0][i] = MAXN;
  cin >> n;
  for (int i = 0; i < n; i++)
    cin >> sparseTable[0][i];
```

```
  precomp();

  int ans = 0;
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      if (query(i,j+1) >= j+1-i)
        ans = max(ans,j+1-i);
  cout << ans << '\n';
}
```

```
// Solution by Raveen, using loop with early exit
#include <iostream>
using namespace std;

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

int main (void) {
  int n;
  scanf("%d",&n);
  for (int i = 0; i < n; i++)
    scanf("%d",&h[i]);

  int ans = 1;
  for (int i = 0; i < n; i++) {
    int sqh = h[i], sqw = 1;
    for (int j = i+1; j < n; j++) {
      sqh = min(sqh,h[j]);
      sqw++;
      ans = max(ans,min(sqh,sqw));
      if (sqh <= sqw)
        break;
    }
  }
  cout << ans << '\n';
}
```

B2. Skyline (full)

Algorithm

There are several ways to solve the full problem.

Binary Search

If a square of size k exists, then a square of size $k-1$ also exists. Therefore the answer to the corresponding decision problem is monotonic, so we can use binary search!

To test whether a square of size k exists, we will test for each start point ℓ whether the next k buildings all have height at least k. Iterating through all k buildings would give an $O(n^2 \log n)$ solution, which is too slow. However, we can look up the minimum height of buildings $[\ell, \ell + k)$ in $O(\log n)$ time using a range tree, or $O(1)$ time using a sparse table. These solutions run in $O(n \log^2 n)$ and $O(n \log n)$ respectively, and both are acceptable for $n \leq 100,000$.

However, there is an even simpler way to test for a square of size k. All we need to do is to traverse the height array from left to right, maintaining a count of how many consecutive buildings of height $\geq k$ we have seen and resetting to zero when we find a building of height $< k$. If the count ever reaches k, we have found a square of size k. This solution requires one pass of the array for each of $O(\log n)$ stages of the binary search, so it also runs in $O(n \log n)$, but with a better constant factor.

Merging Ranges

We process the buildings in decreasing height, and keep a running store of the buildings higher than our current building in a set. In order to speed up computation, we ensure that consecutive blocks of buildings are combined together to form *ranges*. For example, instead of storing the set $\{1, 2, 3, 5, 7, 8, 9, 10\}$, we instead store the set $\{(1, 3), (5, 5), (7, 9)\}$

For each building we process, we search through our set for the range immediately to the left and to the right, then merge the current building into the ranges if it is adjacent to any of them. Afterwards, the largest artwork that can be placed into this range is limited by the length of the range, as well as the minimum height of the range, which is given by the height of the current building.

The total complexity of this algorithm is $O(n \log n)$, which is sufficient when $n \leq 100,000$.

Largest Square Under Histogram

Observe that this problem is a variation on Largest Rectangle Under Histogram from Lecture 2. As such, we can apply the same solution. For each building, we find the widest rectangle whose height is contrained by this building - first scan left to right to find the left endpoints of all such rectangles, then right to left for the right endpoints. Then rather than considering the area of each candidate rectangle, we take the smaller of its two dimensions, and report the largest square formed in this way.

A small modification allows us to solve the problem in one pass. We process the bars of the histogram as in the lecture, keeping a map of "important" columns indexed by height. However, we consider each bar's candidate rectangle when it is removed from the map. A bar is removed when a shorter bar to its right is found, at which point the candidate rectangle has width constrained on the left by the second tallest bar in the map `impCols` and on the right by the new bar. Again, the corresponding square has side length equal to the smaller of the two dimensions of this rectangle.

Implementation Notes

For a solution viewing the problem as the largest square under a histogram, we must ensure that all bars are eventually removed from the map. This is enabled by adding a dummy bar at the end, shorter than all buildings.

Reference Solutions

```
// Solution by Raveen, using binary search and range tree
#include <iostream>
using namespace std;
```

```
const int MAX_N = 100100;
int n;

int tree[300000];

void update(int a, int v, int i = 1, int start = 0, int end = MAX_N) {
  if (end - start == 1) {
    tree[i] = v;
    return;
  }
  int mid = (start + end) / 2;
  if (a < mid)
    update(a, v, i * 2, start, mid);
  else
    update(a, v, i * 2 + 1, mid, end);
  tree[i] = min(tree[i*2],tree[i*2+1]);
}

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

int binarysearch(void) {
  int lo = 1;
  int hi = n;
  int bestSoFar = -1;
  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    bool ok = false;
    for (int i = 0; i + mid <= n; i++)
      if (query(i,i+mid) >= mid) {
        ok = true;
        break;
      }
    if (ok) {
      bestSoFar = mid;
      lo = mid + 1;
    } else
      hi = mid - 1;
  }
  return bestSoFar;
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    update(i,x);
  }

  cout << binarysearch() << '\n';
}
```

```
// Solution by Raveen, using binary search and sparse table
#include <iostream>
using namespace std;

const int MAXN = 100100, LOGN = 18;
int n, sparseTable[LOGN][MAXN], log2s[MAXN];

void precomp() {
  for (int l = 1; l < LOGN; l++) {
    int prevp2 = 1 << (l-1);
```

```
    for (int i = 0; i < MAXN; i++) {
      int intEnd = i + prevp2;
      if (intEnd < MAXN)
        sparseTable[l][i] = min(sparseTable[l-1][i],
            sparseTable[l-1][intEnd]);
      else
        sparseTable[l][i] = sparseTable[l-1][i];
    }
  }
  for (int i = 2; i < MAXN; i++) log2s[i] = log2s[i/2] + 1;
}

int query (int l, int r) {
  int l2 = log2s[r-l];
  return min(sparseTable[l2][l], sparseTable[l2][r-(1<<l2)]);
}

int binarysearch(void) {
  int lo = 1;
  int hi = n;
  int bestSoFar = -1;
  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    bool ok = false;
    for (int i = 0; i + mid <= n; i++)
      if (query(i,i+mid) >= mid) {
        ok = true;
        break;
      }
    if (ok) {
      bestSoFar = mid;
      lo = mid + 1;
    } else
      hi = mid - 1;
  }
  return bestSoFar;
}

int main() {
  for (int i = 0; i < MAXN; i++)
    sparseTable[0][i] = MAXN;
  cin >> n;
  for (int i = 0; i < n; i++)
    cin >> sparseTable[0][i];
  precomp();

  cout << binarysearch() << '\n';
}
```

```
// Solution by Angus, using binary search and counter
#include <cstdio>
#include <algorithm>
#define MAXN 100010
using namespace std;
int n, h[MAXN];

bool canDo(int k) {
  int longest = 0, curr = 0;
  for (int i = 0; i < n; i++) {
    if (h[i] >= k) {
      curr++;
      if (curr == k)
        return true;
    } else
      curr = 0;
  }
  return false;
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++)
    scanf("%d", &h[i]);
```

```
  int s = 0;
  int e = n;
  while (s != e) {
    int mid = (s+e+1)/2;
    if (canDo(mid))
      s = mid;
    else
      e = mid-1;
  }
  printf("%d\n", s);
}
```

```
// Solution by Kevin
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <set>
#include <utility>
using namespace std;
typedef pair<int, int> pii;

int main () {
  int n;
  cin >> n;
  pii h[n]; // {height, pos}
  for (int i = 0; i < n; i++) {
    cin >> h[i].first;
    h[i].second = i;
  }
  // sort in terms of decreasing height, and compute max squares
  sort(h, h+n, greater<pii>());

  // set stores ranges of currently considered skylines
  int ans = 0;
  set<pii> s;
  for (int i = 0; i < n; i++) {
    int l = h[i].second;
    int r = h[i].second;
    auto it = s.upper_bound({l, r});

    if (!s.empty()) {
      if (it != s.begin() && prev(it)->second == l-1) {
        // merge with range to the left
        l = min(l, prev(it)->first);
        s.erase(prev(it));
      }

      if (it != s.end() && it->first == r+1) {
        // merge with range to the right
        r = max(r, it->second);
        s.erase(it);
      }
    }

    // insert and find maximum square under this range
    s.insert({l, r});
    ans = max(ans, min(r-l+1, h[i].first));
  }

  printf("%d\n", ans);
}
```

```
// Solution by Raveen, using largest rectangle under histogram
#include <iostream>
#include <map>
using namespace std;

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

int main (void) {
  int n;
  cin >> n;
```

```
  for (int i = 0; i < n; i++)
    cin >> h[i];
  h[n] = -1;

  map<int,int> impCols;
  impCols[-2] = -1;
  int ans = 0;

  for (int i = 0; i <= n; i++) {
    auto it = prev(impCols.lower_bound(h[i]));
    while (impCols.rbegin()->first >= h[i]) {
      auto last = impCols.rbegin();
      ans = max(ans,min(last->first,i-1-next(last)->second));
      impCols.erase(last->first);
    }
    impCols[h[i]] = i;
  }

  cout << ans << '\n';
}
```

C1. Superstore (subtask)

Algorithm

If there is only one checkout lane, then n customers can be arranged in $n!$ many ways.

If there are two checkout lanes, we again line up the n customers, but we split the line at a position between two customers. Therefore there are $n-1$ choices of where to split the line. However, this double-counts the arrangements (e.g. `1 4 | 3 2 5` and `3 2 5 | 1 4`), so we have to divide by 2 for a final answer of

$$\frac{(n-1) \times n!}{2}.$$

Implementation Notes

The modulus used throughout this problem is $p = 10^9 + 7$, which is less than 2^{31}. Therefore we can multiply any two numbers up to this size without overflowing `long long`, then mod the result. It is best to write helper functions to handle operations like addition and multiplication (and if necessary subtraction) modulo p.

In the case $k = 2$, we have to divide the answer by two. However, dividing modulo p is particularly complicated. It may be the case that $(n-1) \times n! \bmod p$ is odd, so dividing by 2 and rounding down would give a `WRONG-ANSWER`. There are two correct ways to handle this:

1. Instead of dividing by 2, multiply by the inverse of 2 modulo p. The multiplicative inverse is a number $0 \leq q < p$ such that $2q \equiv 1 \mod p$; it is easy to confirm that $q = \frac{p+1}{2} = 5 \times 10^8 + 4$.
2. Alternatively, handle the division in the computation of $n!/2$. This is the approach favoured in the reference solution, as requires less mathematical knowledge.

Reference Solution

```
// Solution by Raveen
#include <iostream>
using namespace std;
typedef long long ll;

const int N = 1010;
const ll MOD = 1000*1000*1000+7;
ll fac[N], halffac[N];

inline ll modmult (ll a, ll b) { return (a * b) % MOD; }

int main (void) {
  int n, k;
  cin >> n >> k;

  fac[1] = 1;
  for (int i = 2; i <= n; i++)
    fac[i] = modmult(i,fac[i-1]);

  halffac[2] = 1;
  for (int i = 3; i <= n; i++)
    halffac[i] = modmult(i,halffac[i-1]);

  if (k == 1)
    cout << fac[n] << '\n';
  else if (k == 2)
    cout << modmult(n-1,halffac[n]) << '\n';
}
```

C2. Superstore (full)

We use dynamic programming. The natural choice of subproblem is $f(i, j)$, the number of ways (modulo p) to arrange i shoppers in j non-empty lines. How do we arrive at the recurrence?

Let's consider the placement of the ith shopper.

- They may be at the front of a lane with other shoppers. The other shoppers can be arranged in $f(i-1, j)$ ways, and then shopper i is placed at the front of any of the j lanes.
- They may be behind another shopper. The other shoppers can be arranged in $f(i-1, j)$ ways, and then shopper i is placed immediately behind any of the $i-1$ other shoppers.
- They may be in a line of their own. The other shoppers can be arranged in $f(i-1, j-1)$ ways.

Therefore the recurrence is

$$f(i, j) = (j + i - 1)f(i - 1, j) + f(i - 1, j - 1).$$

The base case is $f(0, 0) = 1$, as there is only way to assign zero shoppers to zero lanes.[1]

Using a bottom-up implementation, each of $O(nk)$ subproblems is solved in constant time, for a total of $O(nk)$ which is sufficient for $n, k \leq 1,000$.

The answers to this problem are exactly the Lah numbers $L(n, k)$.

Implementation Notes

As mentioned for the subtask, take particular care when working modulo $p = 10^9 + 7$, as we need to avoid overflow. It is prudent to write helper functions for each arithmetic operation you use.

Reference Solution

```
// Solution by Kevin
#include <iostream>
using namespace std;
typedef long long ll;

#define MAXN 1005
#define MOD ll(1e9+7)

ll dp[MAXN][MAXN]; // dp[i][j] = number of ways to put the first i people into j lanes

inline ll modadd(ll a, ll b) { return (a + b) % MOD; }
inline ll modmult(ll a, ll b) { return (a * b) % MOD; }

int main () {
    int n, k;
    cin >> n >> k;

    dp[0][0] = 1;

    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= k; j++) {
            // number of ways of placing person i at the front of an existing line
            dp[i][j] = modadd(dp[i][j], modmult(dp[i-1][j],j));

            // number of ways of placing person i behind someone
            dp[i][j] = modadd(dp[i][j], modmult(dp[i-1][j] ,i-1));

            // number of ways to put person i in a new line
            dp[i][j] = modadd(dp[i][j],dp[i-1][j-1]);
        }

    cout << dp[n][k] << endl;
}
```

[1] You could instead use $f(1, 1) = 1$.