Contest 2 Editorial

COMP4128 23T3

18th October 2023

A1. Project Allocation (subtask)

Algorithm

Since $k = 1$, we must give the first two projects to different people: `AB` and `BA` are both valid, whereas `AA` and `BB` are both invalid. The choice we make for these two tasks doesn't impose any restrictions on the future allocations, so we can choose the better of the two valid arrangements for each pair. The last project should be given to whoever will produce greater quality.

Implementation Notes

General

- Make sure to correctly handle both the odd and even cases for n.
  - You can assign the last task first.
  - Alternatively, explicitly or implicitly add a dummy task with $a_i = b_i = 0$ at the end of the odd case.

Reference Solution

```cpp
// Solution by Raveen

#include <iostream>
using namespace std;

const int N = 1010;
int a[N], b[N];

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

    for (int i = 0; i < n; i++)
        cin >> a[i] >> b[i];

    int ans = 0;
    if (n % 2 == 1) { // last task
        ans += max(a[n-1],b[n-1]);
        n--;
    }

    for (int i = 0; i < n; i += 2) // each pair is AB or BA
        ans += max(a[i]+b[i+1], b[i]+a[i+1]);

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

A2. Project Allocation (subtask)

Algorithm

It is natural to solve in increasing order of the number of projects. Each project will then be given to either Arda or Bimala, so long as the difference doesn't exceed k. To know which of these allocations is valid, we'll need to also keep around the number of projects given to Arda (or equivalent information).

Subproblem: let $f(i, j)$ be the maximum total quality from the first i tasks, with j of them done by Arda.

Recurrence: for $i \geq 1$ and $|j - (i - j)| \leq k$, we have

$$f(i,j) = \begin{cases} f(i-1,j) + b_i & \text{if } j = 0, \\ \max\left(f(i-1,j-1) + a_i, f(i-1,j) + b_i\right) & \text{if } 0 < j < i, \\ f(i-1,j-1) + a_i & \text{if } j = i. \end{cases}$$

Base cases: $f(0, 0) = 0$, and if $|j - (i - j)| > k$ then $f(i, j) = -\infty$.

Overall answer: the best total quality after all tasks are allocated, i.e. $\max_{0 \leq j \leq n} f(n, j)$.

Order of computation: increasing i is sufficient, since $f(i, \cdot)$ depends only on $f(i - 1, \cdot)$.

Implementation Notes

General

- It might be tempting to choose 0 as the default value for invalid states. Remember that this value has to be something that can't be achieved by a valid state; zero is unfortunately the value of the base case (only).
- It's easier to check validity on the fly or implicitly[1] than to explicitly restrict the loop counter j.
- Both 0 projects allocated and n projects allocated are valid states, so declare the DP table slightly larger than the input number n.
- The DP can be implemented either bottom up or top down.

Reference Solutions

```cpp
// Solution by Raveen, bottom up

#include <algorithm> // max
#include <cstdlib> // abs
#include <iostream>
using namespace std;

const int INF = 1000*1000*1000+7;

const int N = 1010;
int dp[N][N];
// dp[i][j] is best score where i projects have been allocated
// j of them to Arda and the remaining i-j to Bimala
int a[N], b[N];

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

    for (int i = 1; i <= n; i++)
        cin >> a[i] >> b[i];

    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
            dp[i][j] = -INF;
    dp[0][0] = 0;
```

[1] a past subproblem with value $-\infty$ must be invalid

```
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= i; j++)
            if (abs(j - (i-j)) <= k) { // check valid
                if (j > 0 && dp[i-1][j-1] != -INF) // Arda does project i
                    dp[i][j] = dp[i-1][j-1] + a[i];
                if (j < i && dp[i-1][j] != -INF) // Bimala does project i
                    dp[i][j] = max(dp[i][j], dp[i-1][j] + b[i]);
            }

    int ans = 0;
    for (int j = 0; j <= n; j++)
        ans = max(ans, dp[n][j]);
    cout << ans << '\n';
}
```

```
// Solution by Isaiah, top down

#include <algorithm> // max
#include <cstdio>
using namespace std;

const int INF = 1000*1000*1000+7;

const int N = 1010;
int n, k, a[N], b[N];
int cache[N][2*N];
// cache[i][j] is best score from project i (1-based) to project n
// where so far Arda has done j projects more than Bimala has
bool seen[N][2*N];

int dp(int i, int j) {
    if (abs(j) > k) // invalid
        return -INF;

    if (i > n) // done, no more score from here
        return 0;

    if (seen[i][j]) // already solved
        return cache[i][j];

    seen[i][j] = true;
    return cache[i][j] = max(a[i] + dp(i+1, j+1),  // Arda does project i
                             b[i] + dp(i+1, j-1)); // Bimala does project i
}

int main() {
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; i++)
        scanf("%d%d", &a[i], &b[i]);
    printf("%d\n", dp(1, 0));
}
```

B1. Tower Power (subtask)

Algorithm

The grid is only 1000×1000, so we can store it in its entirety.

We then proceed by brute force. Each tower can fire to fewer than $m + n$ land plots, so we can walk in all four directions and count the visible plots one-by-one.

Implementation Notes

General

- Any kind of bounds checking should be fine here. Alternatively you could explicitly place dummy towers throughout rows 0 and $m + 1$, and throughout columns 0 and $n + 1$.
- Make sure not to count the tower's position towards its own power.

Reference Solution

```
// Solution by Ryan

#include <iostream>
using namespace std;

const int N = 1010;
bool grid[N][N];
int m, n, k;

int getPower (int r, int c) {
    int rl = r;
    while (rl >= 1 && !grid[rl][c]) // walk north
        rl--;
    int rh = r;
    while (rh <= m && !grid[rh][c]) // walk south
        rh++;
    int cl = c;
    while (cl >= 1 && !grid[r][cl]) // walk west
        cl--;
    int ch = c;
    while (ch >= c && !grid[r][c2]) // walk east
        ch++;

    // exclude (r,c) from both NS and EW
    return (rh - rl - 2) + (ch - cl - 2);
}

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

    for (int i = 0; i < k; ++i) {
        int r, c;
        cin >> r >> c;
        cout << getPower(r, c) << '\n';
        grid[r][c] = true;
    }
}
```

B2. Tower Power (full)

Algorithm

It is helpful to first think about the more restrictive bounds $1 \leq m, n, k \leq 100,000$. We now have a grid too large to store in memory. Remembering that most plots will remain empty throughout, we should hope to represent the grid much more concisely. The important information is:

- in each row, which cells have a tower, and
- in each column, which cells have a tower.

We need to represent each row and column's cells in a data structure that allows:

- arbitrary insertion and
- query of the towers immediately before and after a new position,

each in faster than linear time. The correct data structure is `std::set`, with the above operations handled by the member functions `insert()` and `upper_bound()`. [2]

This leaves one last issue: the original bounds allowed m and n up to 10^9. This means that we can't make a `set` for every row and every column. No matter; most of the rows and columns remain empty throughout, so we can just create them for the rows and columns that appear in the input. Similar to coordinate compression, the appropriate container is a `std::map`.

Implementation Notes

General

- If you choose to use dummy towers again, make sure to place them only in the rows and columns that will eventually have at least one tower.
- Some students got the verdict `COMPILE-ERROR` because the linker identified their program as exceeding the memory limit. Refer to the last dot point on the tips page of the course website for more details.
- The power of each tower is less than $m + n$, so 32-bit integers are sufficient.
- Make sure you don't query a non-existent entry of the `map`. Recall that square bracket indexing creates an empty entry if no entry exists already.

Reference Solution

```
// Solution by Evan

#include <cstdio>
#include <set>
#include <map>
using namespace std;

map<int, set<int>> row, col;

int main() {
    int m, n, k;
    scanf("%d %d %d", &m, &n, &k);
    for (int i = 0; i < k; i++) {
        int r, c;
        scanf("%d %d", &r, &c);

        auto next_col = row[r].lower_bound(c);
        int east = (next_col == row[r].end()) ? n : (*next_col - 1);
        int west = (next_col == row[r].begin()) ? 1 : (*prev(next_col) + 1);

        auto next_row = col[c].lower_bound(r);
        int south = (next_row == col[c].end()) ? m : (*next_row - 1);
        int north = (next_row == col[c].begin()) ? 1 : (*prev(next_row) + 1);
```

[2] `lower_bound()` is equivalent, since the value being queried cannot be in the set, because no two towers coincide.

```
            printf("%d\n", south - north + east - west);

            row[r].insert(c);
            col[c].insert(r);
        }
    }
```

C1. One Millionth Visitor (subtask)

Algorithm

The day of the first visit is the smallest a_i value (say a_1, without loss of generality). Then the second visit could be person 1's second visit, or anyone else's second visit. Therefore it occurs on day

$$\min(a_1 + b_1, a_2, a_3, \ldots, a_n).$$

This should inspire a solution based on simulating the visits one-by-one.

- Maintain the next visit day of each of the n visitors.
- For each visit:
  - pick the smallest of these days, and
  - update it by adding the appropriate b_i value.

We need a data structure which supports:

- query minimum and
- update minimum (or equivalently, both delete minimum and arbitrary insert)

in faster than linear time. The appropriate data structure is a min heap.

Implementation Notes

General

- The largest day that could be relevant is day $1,000,000,000$ (if there is only one person, who visits every one thousand days, and we want the one millionth visit). Therefore 32-bit integers are sufficient.
- We'll need to associate to every day in the heap the index of the person visiting that day, in order to look up the appropriate b_i value. In a happy coincidence, this also orders visits on the same day for us.

Reference Solution

```
// Solution by Raveen

#include <iostream>
#include <queue>
#include <utility>
using namespace std;

typedef pair<int, int> visit; // (day, index)

const int N = 1001001;
int a[N], b[N];

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

    priority_queue<visit,vector<visit>,greater<visit>> pq;
    for (int i = 1; i <= n; i++) {
        cin >> a[i] >> b[i];
        pq.emplace(a[i],i);
    }

    for (int j = 1; j < k; j++) {
        visit next = pq.top();
        pq.pop();
        pq.emplace(next.first+b[next.second],next.second);
    }
    cout << pq.top().second << '\n';
}
```

C2. One Millionth Visitor (full)

Algorithm

Now that k is up to 10^{12}, we cannot hope to simulate the visits one-by-one.

However, the number of visits by a person *up to* a particular day is easily answered, using division. Adding these numbers over all people gives us the total number of visits up to a particular day. This quantity is of course monotonic,[3] so we can use binary search to determine the day of the kth visit.

If we know the day that the prize is awarded, and the total number of visits on all prior days, it just remains to step one-by-one through the people and count the visits on this last day, until the prize is awarded.

Implementation Notes

General

- With the larger bounds, we will easily overflow 32-bit integers. However, the complications do not end here!
  - The largest day that could be relevant is day 10^{18} (if there is only one person, who visits every one million days, and we want the $10^{12}th$ visit).
  - However, if we set this as the upper bound of our binary search, the total number of visits might overflow `long long` (if there are one million people, who each visit every day, the number of visits could be up to 10^{24}).
  - The correct choice is $10^{18}/n$.
    - Each of the n people must visit at least once every one million days, so the total visits are at least 10^{12}, i.e. not smaller than k.
    - Also, each of the n people could visit once per day, so the total visits will not exceed 10^{18} and hence fit in a `long long`.
- Make sure you only count visits from day a_i!
- The testdata for C1 was unfortunately stronger than for C2, as some of the random cases were regenerated with a different seed for C2 rather than copied. Some students solved C2 but not C1, despite having bugs in their programs. Fortunately, no student whose submission for C1 was rejected would have solved C2 with the same program, so the error at least affected everyone equally. We apologise for the error.

Reference Solution

```
// Solution by Raveen

#include <iostream>
using namespace std;

typedef long long ll;

const int N = 1001001;
ll a[N], b[N];

ll n, k;

ll eval (ll day) {
    ll ret = 0;
    for (int i = 1; i <= n; i++)
        if (day >= a[i]) // only count people who have started visiting
            ret += (day - a[i])/b[i] + 1;
    return ret;
}

ll binarysearch (void) {
```

[3] since the number of visits up to day x is less than or equal to the number of visits up to day $x + 1$

```
    ll lo = 0;
    ll hi = 1LL*N*N*N/n;
    ll bestSoFar = -1;
    // Range [lo, hi];
    while (lo <= hi) {
        ll mid = (lo + hi) / 2;
        if (eval(mid) >= k) {
            bestSoFar = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return bestSoFar;
}

int main (void) {
    cin >> n >> k;
    for (int i = 1; i <= n; i++)
        cin >> a[i] >> b[i];

    ll day = binarysearch(); // first day to have at least k visits
    ll cnt = eval(day-1); // number of visits strictly before that day

    for (int i = 1; i <= n; i++) // test who visited that day
        if (day >= a[i] && (day - a[i]) % b[i] == 0)
            if (++cnt == k) { // kth visit found
                cout << i << '\n';
                return 0;
            }
}
```