Second Round Practical

Solution Booklet

Swiss Olympiad in Informatics

March 10–13, 2017

Zen Garden

EZ

```python
n = int(input())
if n in (2, 3):
    print("impossible")
else:
    print(*range(1, n, 2), *range(0, n, 2))
```

Kanji Scam

20 points can be achieved by iterating over all permutations of digits (which are the only mappings allowed). For example, in C++ there exists the function `std::next_permutation`.

For more points, we have to speed up the evaluation of the total value (iterating over all prices is too slow). For each symbol σ, we remember v_σ, the sum over all prices of place values of the positions where it occurs. For example, for the prices `ab`, `ba` and `aa` we would get $v_\text{a} = 10 + 01 + 11$ and $v_\text{b} = 01 + 10 + 00$. The place values of course depend on k, i.e. the i-th last place has the value k^{i-1}.

Now, notice how the sum of prices for an assignment can be computed much more quickly: We can just multiply the assigned value for a symbol by the v_σ, giving us the contribution of this symbol. When summing this up for all symbols, we get the total value of the prices. We go from $O(n)$ to $O(k)$ time for computing the cost with a permutation. The preprocessing step takes $O(n \cdot k)$ or $O(n)$. In the same step we can also find out for each symbol if it is allowed to become zero.

Now we have a much simplified version of the problem, for which we don't need to go over all possile digit assignments: we want to find a permutation P such that $\sum_{i=0}^{k-1} P(i)v_i$ is minimal. This is easy: we just assign $P(i)$ to zero for the i with largest v_i, to one for the second largest v_i, and so forth.

This becomes a bit more complicated because not all symbols may assume the value 0. We just assign zero to $P(i)$ for the maximizer among the symbols which are allowed to become 0. For the remaining i, we proceed normally. This takes $O(n)$ time for preprocessing and $O(k \log k)$ for the sorting of the v_i, so $O(n + k \log k)$ in total. The memory consumption is in $O(n + k)$.

```python
k, n = map(int, input().split())
vals = [input().strip() for _ in range(n)]

occs = [[0,True] for _ in range(k)]
for val in vals:
    occs[ord(val[0])-ord('a')][1] = False
    for i,v in enumerate(reversed(val)):
        occs[ord(v)-ord('a')][0]+=k**i

occs.sort()

ans = 0
did_zero = False
val = 1
for v_i, can_zero in reversed(occs):
    if not did_zero and can_zero:
        did_zero = True
    else:
        ans+=v_i*val
        val+=1
if not did_zero: print("impossible")
else: print(ans)
```

Escape the Prison

We sort the possibilities by the rank in ascending order and the corridors by the level in ascending order. Then we initialize a Union-Find data structure on N (originally isolated) nodes.

Each connected component is assigned the minimum bribe of a guard at a node of the component. Initially, it is just the bribe of the guard at the respective node or $+\infty$ if there is no guard at the respective node.

Now, we are going to process the possibilities. Let c_i and r_i be the current chamber and rank. We first insert all corridors with level up to r_i into the Union-Find. Whenever two components are merged, the new component is assigned the minimum bribe from the two merged components.

Finally, the component of r_i is found. If the respective minimum bribe equals $+\infty$, the string `IMPOSSIBLE` is outputted. Otherwise, the respective minimum bribe is outputted.

The initial sorting requires time in $O(N \log N + Q \log Q)$. The initialization of a Union-Find on N nodes can be performed in $O(N)$.

Overall, at most M corridors are inserted into the Union-Find. Per corridor, an amortised time in $O(\alpha(n))$ is required to insert the corridor into the Union-Find.

Per possibility, a find-operation is performed which needs an amortised time in $O(\alpha(n))$.

The total runtime is thus in $O(N \log N + M\alpha(n) + Q \log Q)$. The memory complexity is in $O(N + M + Q)$.

Note. $\alpha(n)$ is the inverse function of the Ackermann function $A(n, n)$. This value can be considered constant for all practical purposes.

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INFTY = 1000000007;

int N, M, E, Q;

struct corridor {
        int a, b; // chambers
        int l; // level
        bool operator<(const corridor &c) const {
                return l < c.l;
        }
} cor[1000005];

struct possibility {
        int c; // chamber
        int r; // rank
        int id;
        bool operator<(const possibility &p) const {
                return r < p.r;
        }
} pos[100005];

int ans[100005];

// Union-Find
int par[100005]; // parent of a node
int ran[100005]; // this is rank in Union-Find, not rank of Mouse Stofl
int bribe[100005]; // minimum bribe of a component

int find(int v) {
```

```
        if(par[v] == v) return v;
        else return par[v] = find(par[v]);
}

void uni(int v1, int v2) {
        int r1 = find(v1), r2 = find(v2);
        if(r1 == r2) return;
        if(ran[r1] > ran[r2]) swap(r1, r2);
        par[r1] = r2;
        if(ran[r1] == ran[r2]) ran[r2]++;
        if(bribe[r1] < bribe[r2]) bribe[r2] = bribe[r1];
}

int main() {
        ios_base::sync_with_stdio(false);
        cin >> N >> M >> E >> Q;

        // initialize Union-Find
        for(int i = 0; i < N; i++) {
                par[i] = i;
                ran[i] = 0;
                bribe[i] = INFTY;
        }

        for(int i = 0; i < M; i++) {
                cin >> cor[i].a >> cor[i].b >> cor[i].l;
        }
        sort(cor, cor + M);

        for(int i = 0; i < E; i++) {
                int e, p;
                cin >> e >> p;
                if(p < bribe[e]) bribe[e] = p;
        }

        for(int i = 0; i < Q; i++) {
                cin >> pos[i].c >> pos[i].r;
                pos[i].id = i;
        }
        sort(pos, pos + Q);

        int cor_idx = 0; // next corridor to insert
        for(int i = 0; i < Q; i++) {
                while(cor_idx < M && cor[cor_idx].l <= pos[i].r) {
                        uni(cor[cor_idx].a, cor[cor_idx].b);
                        cor_idx++;
                }
                int comp = find(pos[i].c);
                ans[pos[i].id] = bribe[comp];
        }

        for(int i = 0; i < Q; i++) {
                if(ans[i] == INFTY) cout << "IMPOSSIBLE\n";
                else cout << ans[i] << "\n";
        }

        return 0;
}
```

Museum

Whenever you need to count something, you can guess that the solution will be using dynamic programming and combinatorics.

Teilaufgabe 1

There are 2^n possibilities to place glass cases at some positions and the glass cupboards at the other positions. (At each position, we have two possibilities: Either there is a glass case or a glass cupboard.)

In the first subtask, the limits were quite small, so it was feasible to try all possibilities, check whether you can place the SOI flag there and then multiply with the number of possibilities to select a case or cupboard at a given position.

The resulting runtime is $O(n\,2^n)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int mod = 1e9+7;

signed main(){
    int n, k, A, B;
    cin >> n >> k >> A >> B;
    long long ans = 0;
    // 0: case, 1: cupboard
    for(unsigned mask = 0;mask < (1u<<n);++mask){
        bool can_flag = false;
        for(int i=0;i<=n-k;++i){
            // check if flag can be placed at position i
            if(!((mask>>i) & ((1<<k)-1))) can_flag = true;
        }
        if(can_flag){
            long long possibilties = 1;
            for(int i=0;i<n;++i){
                if((mask>>i)&1){
                    possibilties = possibilties * B % mod;
                } else {
                    possibilties = possibilties * A % mod;
                }
            }
            ans+=possibilties;
            ans%=mod;
        }
    }
    cout << ans << "\n";
}
```

Teilaufgabe 2

Instead of counting the number of configurations that contain a SOI flag, it is easier to count the opposite: The number of configurations that don't contain a SOI flag.

Let DP$[len][cnt]$ be the number of possibilities, to place len cases and/or cupboards such that there is no way to put a SOI flag, and we have used cases for the last len positions. The answer to the original problem will then be $(A+B)^n - \sum_{i=0}^{k-1} \text{DP}[n][i]$.

If $cnt > 0$, we need to put a case at the last position, so we have

$$\text{DP}[len][cnt] = \text{DP}[len-1][cnt-1] \cdot A$$

otherwise we have put a cupboard. Before that, we can have used at most $k-1$ consecutive cases. (If there *would* be k consecutive cases, there would be enough place for the SOI flag.) Therefore,

$$\mathrm{DP}[len][0] = \left(\sum_{i=0}^{k-1} \mathrm{DP}[len-1][i]\right) \cdot B.$$

Initially, we did not use any cases, thus $\mathrm{DP}[0][0] = 1$ and $\mathrm{DP}[0][j] = 0 \; \forall j > 0$.

This solution runs in $O(n \cdot k)$, which is fast enough to solve the first two subtasks.

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int mod = 1e9+7;
using ll = long long;

signed main(){
    ll A, B;
    int n, k;
    cin >> n >> k >> A >> B;
    vector<vector<ll> > dp(n+1, vector<ll>(k, 0));
    dp[0][0] = 1;
    for(int i=0;i<n;++i){
        for(int j=1;j<k;++j){
            dp[i+1][j] = dp[i][j-1] * A % mod;
        }
        ll sum = accumulate(dp[i].begin(), dp[i].end(), 0ll)%mod;
        dp[i+1][0] = sum * B % mod;
    }
    ll ab_pow = 1;
    for(int i=0;i<n;++i){
        ab_pow = ab_pow * (A+B) % mod;
    }
    ll ans = ab_pow - accumulate(dp[n].begin(), dp[n].end(), 0ll);
    ans%=mod;
    if(ans<0) ans+=mod; // take care of negative numbers
    cout << ans << '\n';
}
```

Teilaufgabe 3

The solution from subtask 2 can be optimized to $O(n)$, if we store $\mathrm{DP}[i]$ in a data structure that supports the following operations:

- Insert a number to the front
- Remove the number at the back
- Multiply all numbers with A
- Compute the sum of all numbers

It is possible to implement those operations with a *std::queue* in $O(1)$ time, but we won't do it here.

An easier solution is to use a different DP approach: Let $\mathrm{DP}[len]$ be the number of possibilities to place len cases/cupboards such that there is no way to place the SOI flag.

If we have already filled $len-1$ positions and we want to add another case/cupboard, we can only create space for the SOI flag under the following condition: The last k positions have been

used up by cases and at the $(k+1)$-last position we put a cupboard (otherwise there already was enough space for the SOI flag). Thus we have

$$\mathrm{DP}[len-k-1] \cdot B \cdot A^k$$

possibilities that we need to substract. We can fill our DP-table like this:

$$\mathrm{DP}[len] = \mathrm{DP}[len-1] \cdot (A+B) - \mathrm{DP}[len-k-1] \cdot B \cdot A^k$$

where $\mathrm{DP}[i] = (A+B)^k$ for $0 \le i < k$ and $\mathrm{DP}[k] = (A+B)^k - A^k$.

The computation is possible in $O(N)$, which can earn you 75 points.

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int mod = 1e9+7;
using ll = long long;

signed main(){
    ll A, B;
    int n, k;
    cin >> n >> k >> A >> B;
    vector<ll> dp(n+1, 0ll);
    dp[0] = 1;
    for(int i=1;i<k;++i){
        dp[i] = dp[i-1] * (A+B) % mod;
    }
    ll Ak = 1, ABk = 1;
    for(int i=0;i<k;++i){
        Ak = Ak * A % mod;
        ABk = ABk * (A+B) % mod;
    }
    dp[k] = (ABk + mod - Ak)%mod;

    for(int i=k+1;i<=n;++i){
        dp[i] = (dp[i-1]*(A+B) - dp[i-k-1]*B%mod*Ak)%mod;
    }
    ll ab_pow = 1;
    for(int i=0;i<n;++i){
        ab_pow = ab_pow * (A+B) % mod;
    }
    ll ans = ab_pow - dp[n];
    ans%=mod;
    if(ans<0) ans+=mod; // take care of negative numbers
    cout << ans << '\n';
}
```

Teilaufgabe 4

This time we have n large, but k small. If k would be slighly smaller, we could have computed the DP transition using matrix multiplication. However, that solution runs in $O(k^3 \log n)$, which is too slow.

Instead, we can use a divide and conquer style approach. If we split a sequence of n cases/cupboards in the middle, the left part ends at a certain number of cases X and the right part starts with a certain number of cases Y. Because we don't have enough space for the SOI flag, we must have $X + Y < k$.

For the DP from subtask 3, we obtain the following formula for $len > 2k$. (If $len \le 2k$ it would be possible to have only cases on the left/right side. We want to avoid such special cases, so for

$len \leq 2k$ we use the solution of subtasks 2 or 3.)

$$\mathrm{DP}[len] = \sum_{\substack{0 \leq X,Y \\ X+Y<k}} \mathrm{DP}\left[\left\lfloor \frac{len}{2} \right\rfloor - X - 1\right] \cdot A^X B \cdot \mathrm{DP}\left[\left\lceil \frac{len}{2} \right\rceil - Y - 1\right] \cdot A^Y B$$

Computing this DP recursively with memoization only needs $O(k \log n)$ states. (In depth d of the recursion we can only reach states $\left\lfloor \frac{n}{2^d} \right\rfloor - 2k \leq len \leq \left\lfloor \frac{n}{2^d} \right\rfloor$.)

A state can be computed in $O(k)$ if we use prefix sums.

$$\mathrm{DP}[len] = \sum_{X=0}^{k-1} \mathrm{DP}\left[\left\lfloor \frac{len}{2} \right\rfloor - X - 1\right] \cdot A^x B \cdot \underbrace{\sum_{Y=0}^{k-1-X} \mathrm{DP}\left[\left\lceil \frac{len}{2} \right\rceil - Y - 1\right] \cdot A^Y B}_{\text{Here, use prefix sums}}$$

The states with $len \leq 2k$ can be computed in $O(k^2)$ (subtask 2) or $O(k)$ (subtask 3). The total running time is therefore $O(k^2 \log n)$, which scores 75 points.

To get the full 100 points, you need to combine the solutions of subtasks 3 and 4.