Zweite Runde Praxis

Lösungsbüchlein

Swiss Olympiad in Informatics

8.–12. März 2018

Zen Garden

Gesucht ist eine Permutation von $\{0, 1, \ldots, n-1\}$, bei der keine zwei aufeinanderfolgenden Elemente eine absolute Differenz von 1 haben.

Eine der möglichen Lösungen ist es, zuerst die ungeraden Zahlen zu nehmen und dann die geraden:

$$1, 3, 5, 7, 9, 11, \ldots, 0, 2, 4, 6, 8, 10, \ldots$$

Die Differenz ist überall 2, bis in der Mitte. Dort ist sie aber mindestens $n-2$ (die grösste ungerade Zahl vor $n-1$ wird gefolgt von 0). Deshalb ist funktioniert diese Lösung für alle $n \geq 4$.

Schauen wir uns also die Spezialfälle $n < 4$ genauer an:

- $n = 1$: Geht, die einzige Möglichkeit ist: 0.
- $n = 2$: Geht nicht. Die 0 darf nicht neben der 1 stehen.
- $n = 3$: Geht nicht. Die 1 darf weder neben der 0, noch der 2 stehen.

Hätten wir zuerst die geraden und dann die ungeraden Zahlen genommen, müsste man $n = 4$ als weiteren Spezialfall betrachten (da in $0, 2, 1, 3$ die Zahlen 1 und 2 nebeneinander stehen).

Eine direkte Implementierung dieser Idee in C++ sieht so aus:

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

int main() {
  int n;
  cin >> n;
  if (n == 1) {
    cout << "0\n";
  } else if (n == 2 || n == 3) {
    cout << "impossible\n";
  } else {
    for (int i = 1; i < n; i += 2)
      cout << i << ' ';
    for (int i = 0; i < n; i += 2)
      cout << i << ' ';
    cout << '\n';
  }
}
```

Die erste `if`-Bedingung ist gar nicht notwendig, da die Schleifen unten es sowieso richtig ausgeben würden. In Python kann man ausserdem ranges verwenden, dann ist die Lösung ganz kurz:

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

Kanji-Schwindel

20 Punkte sind erreichbar, in dem man alle Permutationen der Ziffern durchgeht (das sind die einzigen Zuordnungen, die erlaubt sind). In C++ gibt es dafür beispielsweise die Funktion `std::next_permutation`.

Für mehr Punkte, müssen wir die Berechnung des Totalpreises vorschnelleren (über die Preise zu iterieren ist zu langsam). Für jedes Symbol σ merken wir uns v_σ, die Summe über alle Preise der Stellenwerte, wo dieses Symbol vorkommt. Z.B. für die Preise `ab`, `ba` und `aa` haben wir $v_\text{a} = 10 + 01 + 11$ und $v_\text{b} = 01 + 10 + 00$. Die Stellenwerte sind abhängig von k, denn die i-t letzte Stelle hat den Wert k^{i-1}.

Mit diesen Werten können wir die Summe der Preise für eine Zuordnung viel schneller berechnen: Wir können einfach den zugeordneten Wert eines Symbols σ mit v_σ multiplizieren und erhalten den Beitrag dieses Symbols zur Totalsumme. Wenn wir diese Produkte für alle Symbole aufsummieren, erhalten wir genau den Totalpreis. Die Vorverarbeitung dauert $O(n \cdot k)$ oder $O(n)$. Im gleichen Schritt finden wir auch heraus, welche Symbole den Wert 0 annehmen dürfen.

Jetzt haben wir es mit einem viel simpleren Problem zu tun, wo wir nicht über alle möglichen Wertzuordnungen gehen müssen: wir wollen eine Permutation P finden, so dass $\sum_{i=0}^{k-1} P(i)v_i$ minimal ist. Das ist relativ einfach: wir ordnen $P(i) = 0$ zu für das i mit grösstem v_i, $P(i) = 1$ für das zweitgrösste v_i und so weiter.

Das ganze wird noch etwas komplizierten, da nicht alle Symbole den Wert 0 annehmen dürfen. Wir ordnen dann $P(i) = 0$ zu, für das i das unter den Symbolen, die den Wert Null annehmen dürfen, den maximalen Wert v_i hat. Für die restlichen i gehen wir normal vor. Das Ganze dauert $O(n)$ für das Vorberechnen und $O(k \log k)$ für das Sortieren der v_i, also total $O(n + k \log k)$. Es wird $O(n + k)$ Speicher gebraucht.

```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)
```

Gefängnisausbruch

Wir sortieren die Möglichkeiten dem Rang nach aufsteigend und die Gänge dem Niveau nach aufsteigend. Dann initialisieren wir eine Union-Find Datenstruktur auf N (zunächst isolierten) Knoten.

Jeder Zusammenhangskomponente ist das minimale Bestechungsgeld eines Wächters an einem Knoten der Komponente zugeordnet. Am Anfang ist es das Bestechungsgeld des Wächters am jeweiligen Knoten oder $+\infty$, wenn es am jeweiligen Knoten keinen Wächter gibt.

Nun bearbeiten wir die einzelnen Möglichkeiten. Sei c_i und r_i der aktuelle Raum und Rang. Wir fügen zuerst alle Gänge mit Niveau höchstens r_i in die Union-Find. Wenn dabei zwei Zusammenhangskomponenten verbunden werden, wird das zugehörige Bestechungsgeld der neuen Komponente durch das Minimum der beiden Zusammenhangskomponenten ersetzt.

Schliesslich wird die Zusammenhangskomponente von r_i ermittelt. Falls das zugehörige minimale Bestechungsgeld gleich $+\infty$ ist, geben wir `IMPOSSIBLE` aus. Sonst geben wir das abgespeicherte minimale Bestechungsgeld aus.

Die Sortierung am Anfang erfordert eine Zeit in $O(N \log N + Q \log Q)$. Die Initialisierung einer Union-Find Datenstruktur auf N Knoten lässt sich in $O(N)$ ausführen.

Insgesamt werden in die Union-Find höchstens alle M Gänge eingefügt. Pro Gang brauchen wir eine Zeit in $O(\alpha(n))$ (amortisiert), um den Gang in die Union-Find einzufügen.

Pro Möglichkeit machen wir zusätzlich eine Find-Operation, die sich auch in $O(\alpha(n))$ (amortisiert) ausführen lässt.

Die Gesamtlaufzeit beträgt somit $O(N \log N + M\alpha(n) + Q \log Q)$. Der benötigte Speicherplatz ist in $O(N + M + Q)$.

Bemerkung. $\alpha(n)$ ist die Umkehrfunktion der Ackermannfunktion $A(n, n)$. Für alle praktischen Anwendungen kann ihr Wert als konstant angenommen werden.

```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";
        }
```

```
88
89          return 0;
90  }
```

Museum

In diese Aufgabe muss etwas gezählt werden, also kann man erwarten, dass die Lösung dynamische Programmierung und mathematische Ausdrücke enthalten wird.

Teilaufgabe 1

Es gibt 2^n Möglichkeiten, an bestimmten Positionen Glasschränke und an den anderen Vitrinen zu platzieren. (An jeder der n Positionen kann entweder einen Schrank oder eine Vitrine stehen.)

In dieser Teilaufgabe waren die Limits sehr klein, also konnte man alle diese Möglichkeiten durchprobieren, überprüfen ob die SOI-Flagge platziert werden kann und dann mit der Anzahl Möglichkeiten, für jeden Schrank- und jede Vitrinenposition ein Exemplar auszuwählen, multiplizieren. Die resultierende Laufzeit ist $O(n\,2^n)$.

```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

Die Aufgabe wird ein wenig einfacher, wenn wir anstelle der Konfigurationen, die die SOI-Flagge enthalten, die Konfigurationen, die die SOI-Flagge nicht enthalten, zählen.

Sei DP$[len][cnt]$ die Anzahl Möglichkeiten, len Schränke und/oder Vitrinen aufzustellen, sodass die SOI-Flagge keinen Platz findet, wobei wir an den letzten cnt Positionen Vitrinen platziert haben. Die Antwort auf das ursprüngliche Problem ist dann $(A+B)^n - \sum_{i=0}^{k-1} \text{DP}[n][i]$.

Wenn $cnt > 0$, so müssen wir an der letzten Position eine Vitrine platziert haben, also gilt

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

ansonsten haben wir einen Schrank platziert und können davor höchstens $k-1$ aufeinanderfolgende Vitrinen platziert haben. (Wenn mindestens k aufeinanderfolgende Vitrinen platziert wurden, so gäbe es einen Platz für die SOI-Flagge.) Es gilt also

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

Anfangs haben wir keine Vitrinen platziert, also gilt $\text{DP}[0][0] = 1$ und $\text{DP}[0][j] = 0 \; \forall j > 0$.

Diese Lösung läuft in $O(n \cdot k)$, was schnell genug ist, um die ersten zwei Teilaufgaben zu lösen.

```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

Diese Lösung aus Teilaufgabe 2 lässt sich auf $O(n)$ optimieren, wenn wir $\text{DP}[i]$ in einer Datenstruktur abspeichern, die folgende Operationen unterstützt

- Eine Zahl vorne einfügen.
- Eine Zahl hinten entfernen.
- Alle Zahlen mit A multiplizieren.
- Die Summe aller Zahlen berechnen.

Mit einer *std::queue* lassen sich diese Operationen in $O(1)$ implementieren, doch dies ist nicht ganz einfach.

Für eine einfachere Lösung betrachten wir $\text{DP}[len]$ als die Anzahl Möglichkeiten, len Möbel aufzustellen, sodass es keinen Platz für die SOI-Flagge gibt.

Wenn wir schon $len-1$ Möbel aufgestellt haben und ein weiteres Möbel aufstellen, kann genau dann ein Platz für die SOI-Flagge entstehen, wenn wir auf den letzten k Positionen Vitrinen

aufgestellt haben. In diesem Fall wurde auf der $(k+1)$-letzten Position ein Schrank aufgestellt (ansonsten gab es vorhin schon einen Platz für die SOI-Flagge), also gibt es

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

Möglichkeiten, die wir subtrahieren müssen. Unsere DP-Tabelle lässt sich also wie folgt berechnen

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

wobei $\mathrm{DP}[i] = (A+B)^k$ für $0 \leq i < k$ und $\mathrm{DP}[k] = (A+B)^k - A^k$.

Die Berechnung dauert $O(N)$, was 75 Punkte erreicht.

```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

In dieser Teilaufgabe ist n sehr gross, dafür ist k klein. Wenn k noch etwas kleiner wäre, könnte man den DP-Übergang als Matrixmultiplikation schreiben, doch diese Lösung läuft in $O(k^3 \log n)$, was zu langsam ist.

Stattdessen versuchen wir einen divide-and-conquer artigen Ansatz. Wenn wir eine Folge von n Möbeln in der Mitte teilen, endet der linke Teil mit einer bestimmten Anzahl X von Vitrinen und der rechte Teil beginnt mit einer bestimmten Anzahl Y von Vitrinen. Da nicht genügend Platz für die SOI-Flagge vorhanden ist, gilt $X + Y < k$.

Für den DP aus Teilaufgabe 3 ergibt sich also folgende Formel für $len > 2k$. (Für $len \leq 2k$ könnte es passieren, dass auf der linken/rechten Seite nur Vitrinen stehen. Solche Spezialfälle

will man möglichst vermeiden, also wird für $len \leq 2k$ die Lösung von Teilaufgaben 2 oder 3 verwendet.)

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

Wenn wir diesen DP rekursiv mit Memoisierung berechnen, werden nur $O(k \log n)$ Zustände besucht. (Auf Rekursionstiefe d werden nur Zustände mit $\left\lfloor \frac{n}{2^d} \right\rfloor - 2k \leq len \leq \left\lfloor \frac{n}{2^d} \right\rfloor$ erreicht.)

Ein Zustand lässt sich in $O(k)$ berechnen, wenn wir für die innere Summe Präfixsummen berechnen.

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

Die Zustände mit $len \leq 2k$ lassen sich in $O(k^2)$ (Teilaufgabe 2) oder auch $O(k)$ (Teilaufgabe 3) berechnen. Die Gesamtlaufzeit ist demnach $O(k^2 \log n)$, was 75 Punkte erreicht. Wenn man die Lösungen von Teilaufgaben 3 und 4 kombiniert, erhält man 100 Punkte.

Ein paar Bemerkungen zur Implementation.

- Die Potenzen A^X für $X \leq k$ kann man im Voraus berechnen.
- Um $(A + B)^n$ in $O(\log n)$ zu berechnen, muss man *Binäre Exponentiation* verwenden.
- Wenn man eine *std::map* oder *std::unordered_map* zur Memoisation verwendet, gibt es höchst wahrscheinlich TLE. Stattdessen sollte man nur Vektoren verwenden und die Zustände anhand der Rekursionstiefe d und $\left\lfloor \frac{n}{2^d} \right\rfloor - len$ indexieren. Dies reduziert die Laufzeit auf rund einen Viertel.

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

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

long long n, A, B;
int k;

vector<ll> power_ab, power_a, small_dp;

vector<vector<ll> > cache;

ll dp(ll len, int depth){
    if(len < small_dp.size()){
        return small_dp[len];
    }
    ll index = ((n>>depth) - len);
    ll& ret = cache[depth][index];
    if(ret == -1){
        ret = 0;
        // prefix sum of Y-terms
        vector<ll> y_vals(k);
        for(int Y=0;Y<k;++Y){
            y_vals[Y] = dp((len+1)/2 - Y - 1, depth+1) * power_a[Y] % mod * B % mod;
        }
```

```
        // compute prefix sums
        for(int Y=1;Y<k;++Y){
            y_vals[Y] = (y_vals[Y-1] + y_vals[Y]) %mod;
        }
        // sum over X
        for(int X=0;X<k;++X){
            ret = (ret + dp(len/2 - X - 1, depth+1) * power_a[X] % mod * B % mod *
            ↪  y_vals[k-X-1])%mod;
        }
    }
    return ret;
}
// compute pow(a, exp) % mod
ll fastpow(ll a, ll exp){
    ll ret = 1;
    for(;exp;exp>>=1){
        if(exp&1) ret = ret * a % mod;
        a = a * a % mod;
    }
    return ret;
}

signed main(){
    cin >> n >> k >> A >> B;
    // precompute powers of a and of (a+b)
    power_ab.resize(k+1, 1);
    power_a.resize(k+1, 1);
    for(int i=0;i<k;++i){
        power_ab[i+1] = power_ab[i] * (A+B) % mod;
        power_a[i+1] = power_a[i] * A % mod;
    }
    // precompute dp for len <= 2*k
    small_dp.assign(2*k+1, 0);
    small_dp[0] = 1;
    for(int i=1;i<k;++i){
        small_dp[i] = small_dp[i-1]*(A+B)%mod;
    }
    small_dp[k] = (power_ab[k] + mod - power_a[k]) % mod;
    for(int i=k+1;i<=2*k;++i){
        small_dp[i] = (small_dp[i-1]*(A+B) - small_dp[i-k-1]*B%mod*power_a[k]) % mod;
    }
    // set up memoization
    int max_depth = 1;
    while(1ll << max_depth <= n) ++max_depth;
    cache.assign(max_depth+1, vector<ll>(2*k+1, -1));
    long long ans = fastpow(A+B, n) - dp(n, 0);
    ans%=mod;
    if(ans<0) ans+=mod;
    cout << ans << '\n';
}
```