5B – Kodowanie macierzami

Autor zadania: Adam Gąsienica-Samek

Autorzy omówienia: Adam Gąsienica-Samek, Paweł Parys

Zadanie

Dane są dwa procesy, Algosia i Bajtek. Algosia 25-krotnie otrzymuje liczbę nie większą niż $3 \cdot 10^{16}$. Algosia chce przesłać otrzymaną liczbę kodując ją w macierzy binarnej 10×10. Przeciwnik permutuje w utworzonej macierzy kolumny i wiersze, a następnie przesyła tę macierz do Bajtka, którego zadaniem jest odczytanie oryginalnej liczby.

Rozwiązanie

Naszym celem będzie iterować się po części klas równoważności, znacząco przybliżając się do ich łącznej liczby wynoszącej około 10^{17} (https://oeis.org/A002724). Warto zastanowić się, jak dla danej klasy wyznaczyć jej reprezentanta. Będzie to przydatne jako pierwszy krok dekodera.

Przypuśćmy na razie, że wszystkie wiersze mają różne liczby jedynek. Posortujmy po tym wiersze. Spróbujmy teraz znaleźć permutację kolumn, która minimalizuje leksykograficznie ciąg złożony z wierszy ułożonych obok siebie od góry do dołu. W pierwszym wierszu należy ułożyć wszystkie zera po lewej stronie wszystkich jedynek. Nie obchodzi nas za to w jakiej kolejności względem siebie są ułożone zera. Zostawimy sobie zatem możliwość ułożenia ich przy rozważaniu kolejnych wierszy. Mamy zatem podział na dwie niezależne grupy kolumn. W drugim wierszu robimy to samo, ale patrząc już na te dwie części niezależnie. Podzieli nam to kolumny na jeszcze więcej grup. Warto zauważyć, że jeśli w danym wierszu grupa w całości składa się z tych samych cyfr, to nie dzieli się ona na więcej części.

Teraz możemy się zastanowić ile jest takich macierzy reprezentantów. Skorzystamy przy tym z programowania dynamicznego. Idąc do kolejnego wiersza nasz aktualny stan możemy opisać przez numer wiersza, liczbę jedynek w danym wierszu, podział kolumn, który możemy sobie trzymać jako maskę 9 bitów, gdzie i-ty bit mówi o tym, czy kolumny i i $i+1$ są w różnych grupach. Mamy zatem `dp[10][10][2^9]`. Po policzeniu go wyjdzie nam około 10^{10}.

Dzięki temu, że wynik liczymy za pomocą programowania dynamicznego jesteśmy w stanie też wygenerować k-tą macierz. Robimy to „schodząc" po dp'ku. Będąc w stanie v patrzymy na jego wszystkie przejścia $u_1, u_2, \ldots, u_n$. Jeśli $\mathrm{dp}[u_i] \geq k$, to idziemy do u_i, w przeciwnym przypadku odejmujemy od k $\mathrm{dp}[u_i]$ i rozważamy u_{i+1} z nowym k. To pozwala nam już na zdobycie dodatniej liczby punktów za to zadanie.

Największym problemem w tym rozwiązaniu było założenie, że wiersze muszą mieć różne liczby jedynek. Jeśli pozwolimy, aby wiele wierszy miało tyle samo jedynek, to dalej musimy umieć je jakoś porządkować. Posortujmy kubełkowo, dla danego kubełka sortujemy wiersze po tym jak wyglądają względem podziału, z którym weszliśmy do kubełka. To dalej jest ograniczenie, które sprawia, że nie wszystkie klasy odwiedzimy, ale jest już dużo słabsze niż poprzednie wymaganie.

Nasz podstawowy dp dalej zostaje taki sam, ale teraz przejścia między stanami wymagają liczenia *ile jest zbiorów wierszy, które mają parami różne sortowania względem podziału.* Użyjemy do tego kolejny raz programowania dynamicznego. Tym razem zaczynamy z pewnym podziałem początkowym oraz liczbą jedynek w wierszu i liczymy dla nich `dp2`[*ile wierszy zostało użytych*][*jaki jest aktualny nowy podział*][*jak wygląda sortowanie ostatniego wiersza*]. Liczymy `dp2` dla wszystkich możliwych początków, sumarycznie nie będą one bardzo duże.

Pozwala nam to na iterowanie się po około $2.8 \cdot 10^{16}$ macierzach.

Aby dobić do $3 \cdot 10^{16}$ można wybrać inną kolejność sortowania po liczbie jedynek, najlepszą kolejnością w tym konkretnym rozwiązaniu będzie `0, 1, 9, 2, 8, 3, 7, 4, 6, 5, 10`. Pozwoli ona na zakodowanie $3.8 \cdot 10^{16}$ macierzy. Możemy kodować dużo więcej macierzy sortując kubełki w pierwszej kolejności po ich rozmiarach, ale wtedy potrzebujemy w podstawowym dp dodatkowego 2^{10}.

Rozwiązanie alternatywne

Będę mówił, że macierze są *równoważne*, jeśli jedną można uzyskać z drugiej przez permutację wierszy i kolumn.

Typem wiersza nazwiemy liczbę jedynek w tym wierszu. Mamy więc wiersze typu od 0 do 10. Oczywiście typ wiersza nie zmienia się przy permutowaniu kolumn, dlatego w dekoderze możemy łatwo posortować wiersze po typach.

Do kodowania użyjemy dwóch *rodzin* macierzy. W pierwszej rodzinie są macierze, gdzie wierszy typu 5 jest od 0 do 5, a wierszy każdego z pozostałych typów od 0 do 2. W drugiej rodzinie są macierze, gdzie wierszy typu 4 jest od 3 do 5, a wierszy każdego z pozostałych typów od 0 do 2. Zwróćmy uwagę, że rodziny te są rozłączne, a określenie do której rodziny należy macierz jest proste. Macierzy z pierwszej rodziny (po utożsamieniu macierzy równoważnych) jest ponad $24 \cdot 10^{15}$, a z drugiej rodziny ponad $12 \cdot 10^{15}$. Starczy nam więc ich na zakodowanie $30 \cdot 10^{15}$ liczb, co jest wymagane w zadaniu (na przykład mniejsze liczby kodujemy z użyciem pierwszej rodziny, a kolejne z użyciem drugiej).

Nawiasem mówiąc, śmiało można byłoby też użyć trzecią rodzinę (tak samo liczną jak druga), gdzie to wierszy typu 6 (a nie 4) jest od 3 do 5. W ten sposób moglibyśmy bez zwiększania wysiłku zakodować do $49 \cdot 10^{15}$ liczb – czy nawet jeszcze więcej, dokładając kolejne analogiczne (choć już coraz mniejsze) rodziny. Pozwalając natomiast w tych rodzinach również na 3 wiersze w każdym z pozostałych typów, a nie tylko do 2 (a wtedy tego głównego, czyli 4 lub 6, tylko od 4 do 5), można przekroczyć $90 \cdot 10^{15}$ kodowanych liczb; taki program byłby jednak wolniejszy i nie jest jasne, czy zdążyłby przekazać 25 liczb w 10 sekund. Wszystkich macierzy jest nieco poniżej $106 \cdot 10^{15}$; jesteśmy tu już więc blisko granicy.

Wróćmy jednak do oryginalnego problemu, gdzie rodziny są dwie. W opisie skupimy się na pierwszej rodzinie; kodowanie z użyciem drugiej działa identycznie. W pierwszej rodzinie typ 5 jest szczególny; wiersze tego typu będziemy mieli na samej górze, a potem będą po kolei wiersze kolejnych typów. Macierze będziemy wypełniać od góry, a więc górne części macierzy będę nazywał *prefiksami*, a dolne *sufiksami*.

Najpierw, dla każdej liczby wierszy w oraz dla każdego typu t policzymy sobie, na ile sposobów można wypełnić najniższe w wierszy zawartością typów począwszy od t tak, aby wierszy każdego typu było nie więcej niż 2 (wykluczamy tu typ 5, który jest na samej górze; wierszy tego typu nie używamy w sufiksie macierzy). Utożsamimy przy tym sufiksy macierzy różniące się tylko permutacjami wierszy; jednak permutacjami kolumn na razie się nie przejmujemy (tzn. nawet jeśli dwa sufiksy stają się równe po permutacji kolumn, to i tak liczymy oba). Jest to proste programowanie dynamiczne: jeśli liczba możliwych zawartości wiersza typu t (czyli liczb 10-bitowych mających t jedynek) wynosi x, to albo dokładamy 0 takich wierszy na 1 sposób, albo 1 taki wiersz na x sposobów, albo dwa takie wiersze na $x \cdot (x+1)/2$ sposobów. Zwróćmy uwagę, że uzyskiwane w ten sposób liczby przekraczają 2^{63}, ale nie przekraczają 2^{100}, używamy więc typu `__int128`, a nie `long long`. Nazwijmy uzyskaną tablicę `ile_opcji`.

Załóżmy teraz, że mamy już ustalony jakiś prefiks macierzy. *Twardym stabilizatorem* tego prefiksu nazwiemy zbiór tych permutacji kolumn, które nie zmieniają tego prefiksu. Można zauważyć (choć nie jest to chyba w tym zadaniu istotne), że ten zbiór permutacji jest grupą, czyli jest zamknięty na składanie oraz na odwracanie permutacji, a także zawiera permutację identycznościową. Twardy stabilizator łatwo wyznaczyć: patrzymy po prostu na to, które kolumny są identyczne – mogą być one dowolnie permutowane między sobą.

Jednak tak naprawdę interesuje nas *miękki stabilizator* danego prefiksu macierzy. Nazwę tak zbiór (znowu będący grupą) takich permutacji kolumn, które zastosowane do naszego prefiksu dają nam taką zawartość, z której możemy odzyskać oryginalny prefiks permutując wiersze. Generując macierz, będziemy chcieli w każdym momencie (tzn. po dodaniu wierszy każdego kolejnego typu) znać miękki stabilizator aktualnego prefiksu; za chwilę wyjaśnimy jak dokładnie go liczymy.

Kluczowe jest, aby mając pewien ustalony prefiks macierzy oszacować, ile nierównoważnych macierzy możemy uzyskać dopisując do tego prefiksu jakiś sufiks. Dokładniej: mamy na myśli jakieś początkowe typy wierszy już obsłużone w prefiksie, a w sufiksie będziemy chcieli mieć tylko typy począwszy od jakiegoś typu t (nie występujące w prefiksie); oczywiście ograniczamy się do

sufiksów, gdzie wierszy każdego typu jest co najwyżej 2. Otóż, w tablicy `ile_opcji` mamy już liczbę m mówiącą, ile jest takich macierzy zanim utożsamimy macierze przechodzące na siebie przy permutacjach kolumn (ale po utożsamieniu macierzy przechodzących na siebie przy permutacjach wierszy). Z drugiej strony, znamy też miękki stabilizator S naszego prefiksu. Zastosowanie jakiejkolwiek permutacji kolumn spoza S (nawet złożonej z jakąś permutacją wierszy) wyprowadza nas poza zbiór zliczanych macierzy, gdyż zmienia prefiks, który miał mieć konkretną, ustaloną zawartość. Zatem każda macierz mająca ten ustalony prefiks jest równoważna co najwyżej $|S|$ innym takim macierzom (ściślej: klasom macierzy względem relacji pozwalającej na permutacje wierszy). Oznacza to, że każdą klasę równoważnych macierzy policzyliśmy w wartości m (z tablicy `ile_opcji`) co najwyżej $|S|$ razy. Nierównoważnych macierzy jest więc co najmniej $m/|S|$, a nawet $\lceil m/|S| \rceil$.

Powyższa wartość jest tylko ograniczeniem dolnym. Dla niektórych sufiksów mogło się bowiem zdarzyć, że zastosowanie jakiejś permutacji z S i potem jeszcze jakiejś permutacji wierszy daje ten sam sufiks, czyli daną klasę równoważności sufiksów policzyliśmy w m mniej niż $|S|$ razy. Dokładniej, jeśli dla jakiejś macierzy istnieje k permutacji kolumn takich, że po zastosowaniu tej permutacji, a następnie pewnej permutacji wierszy, uzyskujemy oryginalną macierz, to macierz tę policzymy tylko $1/k$ razy, zamiast raz. Intuicja jednak podpowiada, a obliczenia potwierdzają, że takich „symetrycznych" macierzy, dla których to k jest większe niż 1, jest względnie mało. Innymi słowy, powyższa wartość całkiem nieźle przybliża (od dołu) faktyczną liczbę nierównoważnych macierzy. Zauważmy przy okazji, że ta liczba k (czyli to, ile dana macierz wnosi do naszego oszacowania) nie zależy od tego, gdzie jest granica między ustalonym prefiksem a zmiennym sufiksem.

To daje nam już zarys algorytmu. Mianowicie, będziemy iść od góry i ustalać fragmenty kolejnych typów. Rozważając typ t, będziemy mieli już ustalony prefiks składający się z wierszy poprzednich typów i będziemy chcieli zakodować liczbę nie większą niż nasze oszacowanie b na liczbę sposobów uzupełnienia tego prefiksu do całej macierzy. Przeiterujemy się po wszystkich możliwych zawartościach wierszy typu t. Dla każdej (i-tej) z tych zawartości policzymy miękki stabilizator, co da nam pewne ograniczenie dolne b_i na liczbę sposobów uzupełnienia uzyskanego prefiksu do całej macierzy. Idąc dalej, pierwsze b_1 liczb będziemy kodować za pomocą pierwszej zawartości, kolejne b_2 liczb za pomocą drugiej zawartości, itd. Wyznacza nam to zawartość wierszy typu t, powiedzmy i-tą. Wówczas od liczby do zakodowania odejmujemy $b_1 + \cdots + b_{i-1}$ i kontynuujemy kodowanie liczby nie większej niż b_i.

Jak już powiedzieliśmy, każda macierz wnosi do oryginalnego oszacowania b tyle samo, co do pewnego oszacowania b_i. Konkretnie, nasze oszacowania b oraz $b_1, \ldots, b_n$ to sufity z pewnych potencjalnie niecałkowitych oszacowań b' oraz $b'_1, \ldots, b'_n$, dla których zachodzi $b'_1 + \cdots + b'_n = b'$. Wynika z tej równości, że $\lceil b'_1 \rceil + \cdots + \lceil b'_n \rceil \geq \lceil b' \rceil$, czyli możliwych zawartości nam nie zabraknie, co najwyżej będzie ich trochę za dużo (jeśli jednak bralibyśmy podłogi, to byłoby źle, bo mogłoby się przytrafić, że $\lfloor b'_1 \rfloor + \cdots + \lfloor b'_n \rfloor < \lfloor b' \rfloor$).

Pozostaje nam pokazać dwie rzeczy: jak sprawnie iterować po nierównoważnych zawartościach wierszy typu t oraz jak liczyć miękki stabilizator. Przede wszystkim zobaczmy, że tych nierównoważnych zawartości nie jest zbyt dużo. Dla najwyższego typu, czyli 5, wychodzi tego około 10000; pozwalamy tu co prawda na aż 5 wierszy, więc wszystkich możliwych zawartości jest znacznie więcej, jednak ratuje nas fakt, że równoważności także jest bardzo dużo, bo jeszcze nic wyżej nie jest ustalone, czyli można dowolnie permutować kolumny i wiersze. Natomiast dla kolejnych wierszy pozwalamy tylko na maksymalnie 2 wiersze, czyli w ogóle liczba możliwych zawartości wychodzi takiego rzędu.

Iterowanie po możliwych nierównoważnych zawartościach wierszy typu t przeprowadzimy na zasadzie backtrackingu: po ustaleniu pewnej ilości górnych wierszy iterujemy po możliwych zawartościach kolejnego wiersza, ale kontynuujemy w dół tylko wtedy, gdy uzyskiwana zawartość nie jest równoważna żadnej wcześniejszej. Będziemy też przy okazji liczyć miękki stabilizator aktualnej zawartości. Przyjrzyjmy się pojedynczemu krokowi: powiedzmy, że pewną liczbę wierszy typu t mamy ustalonych i chcemy przeiterować po możliwych zawartościach kolejnego wiersza typu t. Niech S będzie miękkim stabilizatorem tego, co mamy do tej pory. Próbując po kolei każdą kolejną zawartość, zaaplikujemy do niej od razu wszystkie permutacje z S i zobaczymy jakie inne zawartości można uzyskać; oznaczymy uzyskane zawartości jako już przebadane (a przy okazji zapiszemy

dla nich w tablicy `cofaj`, jaka permutacja cofa je do tej pierwszej). To eliminuje już zawartości, które są równoważne bez zamiany ostatniego wiersza z wcześniejszymi. Musimy jednak zbadać jeszcze równoważności wymagające zmiany kolejności wierszy. W tym celu stosujemy po prostu wszystkie możliwe permutacje wierszy (jest ich zaledwie $5! = 120$, a dalej już tylko $2! = 2$). Po zaaplikowaniu takiej permutacji, przechodzimy po kolejnych wierszach i zaglądamy do naszej tablicy odwiedzonych zawartości, gdzie możemy zobaczyć jedną z trzech rzeczy. Opcja 1: być może ta zawartość, potencjalnie z przepermutowanymi kolumnami, była odwiedzona już wcześniej (czyli nie należy jej ponownie rozważać). Opcja 2: być może naszej zawartości jeszcze nie widzieliśmy (czyli jest nowa, należy ją rozważyć). Opcja 3: dostaliśmy z powrotem naszą własną zawartość, potencjalnie z przepermutowanymi kolumnami (tu także nasza zawartość jest nowa i należy ją rozważyć).

Przy okazji wykrycie opcji 3 pozwala nam znaleźć miękki stabilizator. Dla każdej permutacji wierszy dostajemy bowiem (o ile istnieje, czyli o ile pojawiła się opcja 3) pewną permutację kolumn π, która prowadzi z powrotem do oryginalnej zawartości (odczytujemy tę permutację ze wspomnianej powyżej tablicy `cofaj`). Jednocześnie każda permutacja z miękkiego stabilizatora powstaje poprzez złożenie jednej z tak uzyskanych permutacji π z pewną permutacją z twardego stabilizatora (czyli w ogóle nie zmieniającą zawartości żadnego z wierszy).

Dopowiedzmy jeszcze jak reprezentować wspomniane stabilizatory. Otóż, o ile są one niewielkie, możemy trzymać po prostu listę permutacji. Natomiast niewątpliwie konieczny jest szczególny przypadek na grupę wszystkich permutacji, gdyż jest ich jednak za dużo (innymi słowy, zamiast pamiętać listę wszystkich permutacji, mamy specjalną stałą „wszystko"). W naszym programie mamy też stałe na dwie inne specjalne grupy: grupę pozwalającą na dowolne permutacje na pewnym zbiorze pozycji i dowolne permutacje na zbiorze pozostałych pozycji (taka grupa powstaje po pierwszym wierszu: można dowolnie permutować zera i dowolnie jedynki), a także na podobną grupę, gdzie jednak dodatkowo pozwalamy na zamianę jednego zbioru z drugim (oba muszą więc mieć rozmiar 5). Post factum wydaje się jednak, że w tych przypadkach także wystarczyło trzymać listę permutacji (a szczególny przypadek mieć tylko na grupę wszystkich permutacji).

Kodując 25 liczb, część obliczeń (w szczególności te dotyczące prefiksu zawierającego wiersze typu 5) można uwspólnić i coś stablicować. Jednak w naszym programie nic takiego nie zrobiliśmy, po prostu każdą liczbę kodujemy niezależnie.

Nie wspomnieliśmy też jeszcze nic o dekoderze, jednak działa on prawie tak samo jak enkoder. Różnica jest jedynie taka, że zamiast wybierać zawartość kolejnego fragmentu na podstawie liczby, mamy ją daną i odczytujemy przesunięcie, czyli sumę oszacowań $b_1 + \cdots + b_{i-1}$ opisanych powyżej). Musimy przy tym wykryć, czy zawartość fragmentu otrzymanej macierzy jest równoważna aktualnie rozważanej. Jest to jednak bardzo podobne do opisanego powyżej sprawdzania, czy zawartość otrzymana po permutacji wierszy była już rozważana.