Omówienie zadań z turnieju indywidualnego nr 31

Franciszek Malinka

22.03.2025

Omówienia przedstawione są w kolejności od najprostszego do najtrudniejszego, wg opinii autorów.

1 Kość

Przejście od 1 pola do N-tego odbędzie się najszybciej, jeżeli będziemy wyrzucać maksymalną możliwą liczbę oczek K, a na końcu, odległość pionka od ostatniego pola wyniesie co najwyżej K, wyrzucimy dokładnie tę liczbę oczek. Wykonamy zatem $\lceil \frac{N-1}{K} \rceil$ rzutów kością, ponieważ musimy przemieścić się o $N-1$ pól w przód. Zapis $\lceil \frac{N-1}{K} \rceil$ czyta się jako "sufit" i oznacza wartość liczby $\frac{N-1}{K}$ zaokrągloną w górę. Prosta i szybka implementacja sufitu w C++ wygląda tak:

```
long long ceil(long long n, long long k) {
  /* Jeśli k dzieli n, to (n + k - 1) / k == n / k.
     W przeciwnym wypadku (n + k - 1) / k == n / k + 1; */
  return (n + k - 1) / k;
}
```

Złożoność czasowa rozwiązania to $\mathcal{O}(1)$ dla każdego zestawu testowego.

2 Zoom

Po pierwsze, rogi prostokątów podanych na wejściu mogą przyjść w różnej konfiguracji (np. najpierw prawy-dolny, a później lewy-górny). Możemy zatem każdy prostokąt sprowadzić do jednej, wybranej przez nas reprezentacji, np. lewy-dolny i prawy-górny. Wystarczy zmienić ze sobą wartości zmiennych x_1, x_2 o ile $x_1 > x_2$ i analogicznie zmienić wartości zmiennych y_1, y_2 o ile $y_1 > y_2$.

Zauważmy, że jeżeli prostokąty dałoby się ułożyć tak, żeby każdy kolejny zawierał się w poprzednim, to posortowanie ciągu tych prostokątów najpierw po rosnącym lewym-dolnym rogu, a następnie po malejącym prawym-dolnym rogu da nam oczekiwane posortowanie. W C++, jeżeli utrzymujemy ciąg prostokątów jako `vector<pair<pair<int, int>, pair<int, int>>>`, możemy wywołać na tym vectorze funkcję `sort` z dodatkowym argumentem, który "powie" sortowi w jaki sposób porównać ze sobą dwa prostokąty (innym, trickowym rozwiązaniem, byłoby zanegować wartości prawych-górnych rogów, wtedy domyślny sort zadziała tak jakbyśmy chcieli).

```
bool compare(const pair<pair<int, int>, pair<int, int>>& p1,
             const pair<pair<int, int>, pair<int, int>>& p2) {
  if (p1.first == p2.first) {
    /* lewe-dolne rogi są te same, więc porównujemy
       prawe-górne rogi w malejącej kolejności */
    return p1.second > p2.second;
  }
  return p1.first < p2.first;
}

int main() {
  vector<pair<pair<int,int>, pair<int,int>>> prostokaty;
  /* wczytywanie ... */
  sort(prostokaty.begin(), prostokaty.end(), compare);
}
```

Jedyne, co pozostało do zrobienia, to przejść się po posortowanej tablicy prostokątów i sprawdzić czy prostokąty faktycznie spełniają warunki zadania.

Złożoność czasowa rozwiązania to $\mathcal{O}(n \log n)$.

3 Ulubione liczby

Zauważmy, że liczba każdego z dostępnych klocków jest niewielka, zatem nie jesteśmy w stanie ułożyć więcej, niż 100 000 liczb A. Możemy zatem przeiterować się po każdej możliwej liczbie ułożonych liczb A i zobaczyć, ile liczb B możemy ułożyć z pozostałych klocków. Żeby sprawdzić, ile liczb B możemy ułożyć, należy sprawdzić, dla której z cyfr występujących w B mamy relatywnie najmniej klocków do dyspozycji, tj. szukamy wartości

$$\min_{c=0,1,\ldots,9} \left\lfloor \frac{\text{liczba dostępnych klocków } c}{\text{liczba wystąpień } c \text{ w } B} \right\rfloor$$

Dla wygody implementacji, polecam napisać sobie funkcję `vector<int> ile_cyfr(int k)`, która dla danej liczby k zwraca 10-elementowy vector przedstawiający liczbę wystąpień każdej z cyfr w zapisie dziesiętnym k.

Końcowa złożoność czasowa rozwiązania jest liniowa względem liczby dostępnych klocków.

4 Wyrównanie terenu

Ustalmy konkretny poziom k, do którego chcemy wyrównać nasz teren. Zauważmy, że jeżeli w optymalnym rozwiązaniu jakiś fragment terenu zmniejszamy, to już nigdy nie będziemy chcieli go również zwiększyć (dowód obserwacji jest techniczny, ale nietrudny, pozostawiamy go czytelnikowi do samodzielnego rozważenia). To samo dotyczy fragmentów zwiększanych. Możemy zatem rozbić szukanie odpowiedzi na dwie części: szukanie kosztu zmniejszenia wszystkich fragmentów do wysokości co najwyżej k oraz zwiększenia wszystkich fragmentów do jwysokości co najmniej k. Od tej pory zajmiemy się

tylko częścią liczenia kosztu zwiększenia fragmentów do poziomu k, ponieważ druga część jest analogiczna (może być zaimplementowana tą samą funkcją, jeżeli dobrze się to przemyśli).

Na początku, dla każdej wartości policzmy wszystkie spójne przedziały składające się z tej wartości. Ponadto zapiszmy wszystkie wartości, które pojawiły się w zadaniu (czy to jako początkowe wysokości, czy wysokości z zapytań) i posortujmy rosnąco. Będziemy iterować się teraz od najmniejszej wysokości do największej, żeby dla każdej z nich obliczyć odpowiedź (czyli ile kosztuje wyrównanie wszystkich fragmentów z dołu do tej wartości). Kiedy rozważamy konkretną wartość v_{i+1}, znamy już odpowiedź dla wartości v_i, co więcej, możemy również znać liczbę spójnych przedziałów, które mają wartość dokładnie v_i (o tym jak to utrzymywać, w następnym akapicie). Dokładnie te przedziały musimy podnieść $|v_{i+1} - v_i|$ razy, aby wyrównać je z poziomem v_{i+1}. Zatem liczba tych przedziałów pomnożona przez różnicę między v_{i+1} a v_i to dodatkowa praca, jaką musimy wykonać po wyrównaniu wszystkich terenów do v_i, żeby teraz były wyrównane z v_{i+1}.

Jedyne co pozostało, to otrzymać zbiór przedziałów, które mają teraz wartość v_{i+1}. Można to zrobić np. przy użyciu struktury union-find. Iterujemy się po wszystkich przedziałach o wysokości v_{i+1}, jeżeli ich prawy lub lewy koniec sąsiaduje z jakimkolwiek elementem o mniejszej wysokości, wykonujemy operację union.

Końcowa złożoność czasowa całego rozwiązania to $\mathcal{O}(n \log n)$.

5 Redukcja

Po pierwsze, rozwiążmy powiązane, narzucające się zadanie. Dla danej liczby N, jaka jest moc zbioru R_N (moc, czyli jego rozmiar)? Spójrzmy na zapis binarny liczby N. Zauważmy, że jeżeli jakieś dwie jedynki bądź dwa zera występują obok siebie, to nigdy nie będziemy mogli zredukować żadnych cyfr na prawo i lewo od tego miejsca w jednej redukcji. Podzielmy zatem zapis binarny N na najdłuższe ciągi naprzemiennych zer i jedynek, przykładowo dla $N = 10101101100_{(2)}$ otrzymamy ciągi $10101, 101, 10, 0$.

Każdy z części zapisu binarnego N możemy redukować niezależnie, a wybór redukcji na każdym z nich daje unikalną liczbę. Zastanówmy się zatem, na ile sposobów można zredukować ciąg naprzemiennych jedynek i zer? Nietrudno zauważyć, że nie ma znaczenia, czy w danym ciągu zredukujemy 010 czy 101, nie ma też znaczenia którą z takich trójek zredukujemy, otrzymamy tę samą liczbę. Znaczenie ma zatem jedynie długość takiego ciągu. Jeżeli taki ciąg ma długość D, to liczba możliwych do otrzymania liczb wynosi $\lceil \frac{D}{2} \rceil$.

Podsumowując, przez $D_1, D_2, \ldots, D_m$ oznaczmy długości kolejnych części zapisu binarnego N po podzialen opisanym powyżej. Wtedy rozmiar zbioru R_N wynosi

$$\prod_{i=1}^{m} \lceil \frac{D_i}{2} \rceil.$$

Zastanówmy się zatem, jak powinna wyglądać najmniejsza liczba N, która ma dokładnie K możliwych redukcji. Skoro ma być ona najmniejsza, to nie warto, żeby $D_i \leq 2$ dla jakiegokolwiek i, ponieważ nie zwiększa to w żaden sposób liczby możliwych redukcji, a "wydłuża" liczbę, przez co zwiększa jej

wartość. Ponadto, nie warto, żeby $\lceil \frac{D_i}{2} \rceil$ było złożoną. Gdyby $\lceil \frac{D_i}{2} \rceil = a \cdot b$ dla $a, b > 1$, to zmiana tego fragmentu na dwa ciągi długości $2a+1$ i $2b+1$ spowodowałyby, że liczba redukcji zostałaby ta sama, a liczba na pewno byłaby mniejsza, ponieważ jej zapis w systemie binarnym byłby krótszy (gdyż $2a+2b+2 \leq 2ab \leq D_i$).

Wiemy zatem, że szukana przez nas liczba musi dzielić się na fragmenty naprzemiennych jedynek i zer, których długości są równe $2p_i+1$, gdzie p_i jest kolejną liczbą w rozkładzie na czynniki pierwsze liczby K. Jeżeli jakakolwiek liczba pierwsza występuje w potędze większej, niż 1, to musimy zapisać ją wielokrotnie. Zatem, przykładowo, dla liczby $K = 12$, nasze szukane N dzieli się na fragmenty 101, 101, 10101. Skoro chcemy, żeby nasza liczba była jak najmniejsza, to fragmenty odpowiadające największym liczbom pierwszym powinny znajdować się "na lewo" względem krótszych fragmentów, ponieważ później pojawi się powtórzona cyfra 1.

Ostatecznie, jeżeli $p_1 p_2 \ldots p_k = K$ oraz $p_1 \geq p_2 \geq \ldots \geq p_k$, to szukane przez nas N w zapisie binarnym jest następującej postaci:

$$\underbrace{101\ldots101}_{2p_1+1}\underbrace{101\ldots101}_{2p_2+1}\ldots\underbrace{101\ldots101}_{2p_k+1}.$$

Pozostało jedynie policzyć wartość $N \mod M$, gdzie $M = 998\,244\,353$. Dla czytelności, niech $a_p = \underbrace{101\ldots101}_{p}{}_{(2)}$. Zastanówmy się najpierw, jak dla danego p policzyć wartość $a_p \mod M$. Możemy zastosować podobną sztuczkę jak w przypadku potęgowania modularnego, rozpisując wzór na wartość a_p.

- Gdy $p = 0$, to $a_p = 1$.
- Gdy $p = 1$, to $a_p = 101_{(2)} = 5$.
- Gdy $p = 2q$, to $a_p = 2^{2q+1} a_q + a_q$.
- Gdy $p = 2q+1$, to $a_p = 2^{2q+3} a_q + 4a_q + 1$.

Innym sposobem jest zauważyć, że $\underbrace{101\ldots10}_{p}{}_{(2)} = 1+4+16+\ldots+4^p$, możemy zatem skorzystać z sumy ciągu geometrycznego, $a_p = \frac{4^{p+1}-1}{4-1}$, a zatem

$$a_p \equiv (4^{p+1}-1) \cdot 3^{-1} \mod M$$

. Wartość $3^{-1} \mod M$ można łatwo policzyć już napisanym potęgowaniem modularnym, gdyż $3^{-1} \equiv 3^{M-2} \mod M$, z małego twierdzenia Fermata.

Ostateczną wartość $N \mod M$ liczymy iteracyjnie stosując wzór

$$N = ((a_{p_1} 2^{2p_2+1}) + a_{p_2}) 2^{2p_3+1} + a_{p_3}) \ldots + a_{p_k}).$$

Złożoność czasowa całego rozwiązania to $\mathcal{O}(n \log^2 n)$ w przypadku pierwszego podejścia z liczeniem wartości $a_p \mod M$ lub $\mathcal{O}(n \log n)$ w drugim przypadku.

6 Klany

Gdyby wszystkie klany miały ten sam herb, zadanie sprowadziłoby się do wyznaczenia najmłodszego wspólnego przodka, czyli skorzystanie ze standardowego algorytmu LCA (least common ancestor). Pełne rozwiązanie będzie opierać się na tym algorytmie. Na początku wczytajmy wszystkie zapytania, i dla każdego z nich policzmy faktyczne LCA w drzewie. W każdym wierzchołku, który jest LCA dla pewnych wierzchołków z zapytania, zatrzymajmy informację o tym, które to było zapytanie oraz o jaki kolor herba było pytanie.

Następnie, po obsłużeniu wszystkich zapytań, przejdźmy za pomocą DFS po całym drzewie. Dla każdego herbu trzymajmy stos odwiedzonych wierzchołków z danym herbem. Wtedy, odwiedzając dany wierzchołek v, czubek stosu dla danego herbu h będzie określał najbliższy wierzchołek na ścieżce do korzenia u, który ma herb h. Wierzchołek u jest odpowiedzią na wszystkie zapytania o herb h zapamiętane w wierzchołku v.

Alternatywnym, krótszym do napisania rozwiązaniem jest wariant offline LCA. Na początku w każdym wierzchołku zapiszmy listę numerów zapytań, które dotyczą tego wierzchołka, tj. dla zapytania nr i postaci u_i, v_i, h_i, zapiszmy w u_i oraz v_i, że i-te pytanie ich dotyczy. Ponadto będziemy utrzymywać strukturę union-find, a dla każdego zbioru będziemy trzymać dodatkowo wartość "ojca" tego zbioru. Wychodząc z danego wierzchołka v, wykonujemy union z jego ojcem u, a ponadto ustawiamy wartość zbioru v na u.

Wtedy, odwiedzając wierzchołek v, którego dotyczy zapytanie i, jeżeli v jest drugim odwiedzanym wierzchołkiem dla tego zapytania (pierwszym był jakiś u), LCA tego zapytania jest w ojcu zbioru u.

Złożoność czasowa rozwiązania to $\mathcal{O}(n \log n)$ w pierwszej wersji oraz $\mathcal{O}(n\alpha(n))$ w drugiej z union-find.

7 Podział ciągu

Na wstępie zastanówmy się, jak wygląda optymalny podział dla danego ciągu wartości. Kluczową obserwacją jest to, że istnieje optymalny podział, w którym każdy kawałek jest monotoniczny, tj. elementy tego kawałka nie maleją lub nie rosną.

To narzuca pewne rozwiązanie, mianowicie, moglibyśmy utrzymywać drzewo przedziałowe, a w każdym węźle utrzymywać kilka informacji: odpowiedź dla tego węzła, wartości skrajnych elementów oraz informację o tym czy są elementami kawałków rosnących czy malejących. To rozwiązanie jest poprawne, aczkolwiek istnieje inne, znacznie prostsze implementacyjnie rozwiązanie.

Zmieńmy nasz początkowy ciąg $a_1, \ldots, a_N$ w ciąg różnic między kolejnymi wartościami $d_i = a_i - a_{i+1}$. Wtedy wartość podziału na **kawałki monotoniczne** to suma $\sum |d_i|$ z wyłączeniem tych d_i, w których dokonujemy cięcia. Zatem nasz problem sprowadza się do wybrania takiego podzbioru d_i, że suma ich wartości bezwzględnych jest maksymalna, ale nie możemy wybrać sąsiednich takich sąsiednich d_i, d_{i+1}, że ich znaki są przeciwne (to by znaczyło, że mamy jakiś kawałek, który nie jest monotoniczny).

Teraz budujemy drzewo przedziałowe na naszych wartościach, ponownie w każdym węźle chcemy trzymać odpowiedź dla przedziału obejmującego ten

węzeł, również w przypadkach, w których dokonujemy nie bierzemy skrajnie lewego lub prawego elementu. Wtedy scalenie dwóch węzłów musi jedynie sprawdzać w jaki sposób scalić skrajnie prawą wartość lewego węzła ze skrajnie lewą wartością prawego węzła. Jeśli mają te same znaki, można połączyć je w jeden monotoniczny ciąg, inaczej musimy dokonać odpowiedniego cięcia. Update również stają się prostsze, ponieważ teraz dokonujemy zmian w dwóch punktach, na końcach przedziałów (różnice wewnątrz aktualizowanego przedziału pozostają takie same).

Całość rozwiązania kosztuje nas $\mathcal{O}(n + q \log n)$.