Rozwiązania zadań z Mistrzostw Polski Szkół Średnich w Programowaniu Zespołowym 2022

Zadanie A

W zadaniu należy sprawdzić, czy daną liczbę N możemy zapisać jako sumę cyfr 2 i 3. Skoro mamy użyć jak najmniej cyfr, to będziemy chcieli użyć jak najwięcej cyfr 3. Zauważmy, że jeżeli N przy dzieleniu przez 3 daje resztę:

- 0 – to optymalnie jest zapisać liczbę przy użyciu samych cyfr 3.
- 2 – skoro liczba nie jest podzielna przez 3, to musimy użyć co najmniej jednej 2. W tym przypadku wystarczy użyć jednej, a resztę liczby zapisać jako sumę tylko cyfr 3.
- 1 – liczba znowu nie jest podzielna przez 3, ale tym razem musimy użyć dwóch cyfr 2. Trzeba uważać na przypadek brzegowy, gdy $N = 1$, wtedy nie istnieje rozwiązanie.

Zadanie B

W tym zadaniu kluczową obserwacją jest fakt, że liczba dająca się ułożyć z N zapałek może mieć co najwyżej $\frac{N}{2}$ cyfr. Oczywiście takich liczb jest $\mathcal{O}\left(10^{\frac{N}{2}}\right)$. Ponadto skoro interesują nas tylko liczby podzielne przez 111 to możemy przeiterować się tylko przez nie, to znaczy zaczynamy od 0, potem 111, następnie 222 itd.

Te dwie obserwacje pozwalają na rozwiązanie całego zadania – liczb które powinniśmy sprawdzić jest co najwyżej $\mathcal{O}\left(\frac{10^{\frac{N}{2}}}{111}\right)$, czyli dla $N = 20$ jest ich rzędu 10^8. Każdą z nich możemy sprawdzić liniowo wydobywając z niej kolejne cyfry za pomocą operacji reszty z dzielenia przez 10 oraz dzielenia przez 10.

Zadanie C

W tym zadaniu mieliśmy odszyfrować trzy listy.

Gdybyśmy znali ziarno, którego użyto do szyfrowania wiadomości, to w prosty sposób możemy ją odszyfrować. Zauważmy, że wynik funkcji `los` nie zależy od tego, jaka jest aktualnie przetwarzana literka. Zatem, żeby odszyfrować pojedynczą literkę, możemy stworzyć funkcję `deszyfruj_znak`, która działa tak samo jako `szyfruj_znak`, ale zamiast dodawać przesunięcie, to je odejmuje. Należy jednak pamiętać, że operator modulo w C++ zachowuje znak, więc trzeba jeszcze zadbać o to, żeby wynik był dodatni. Jeżeli mamy funkcję deszyfrującą znak to możemy napisać funkcję

`deszyfruj_tekst`, która działa prawie tak samo, jak `szyfruj_tekst` z tą różnicą, że wywołuje funkcję `deszyfruj_znak`. Nasz kod możemy sprawdzić na teście przykładowym podanym w treści.

W treści zadania były ukryte trzy wskazówki:

1. Pierwszy list zawiera palindrom składający się co najmniej z 6 liter.
2. Drugi list zawiera słowo "kwadratowe" o długości co najmniej 5.
3. Podpis autora jest za każdym razem taki sam, więc odszyfrowanie dwóch poprzednich listów pomoże odszyfrować ostatni.

Skoro wiemy jakie warunki muszą spełniać oryginalne treści listów, to możemy sprawdzić po kolei każde dostępne ziarno i spróbować odszyfrować list z jego pomocą. Taka pętla powinna działać kilka sekund i wygenerować kilkanaście kandydatów na oryginalny tekst, z których łatwo ręcznie wybrać poprawną odpowiedź.

Po odszyfrowaniu dwóch pierwszych wiadomości wystarczy zauważyć, że wspólna część podpisu to *dalibor gosciwuj*. Odszyfrować list można analogicznie jak dwa poprzednie.

Jednym z alternatywnych rozwiązań było napisanie funkcji, która heurystycznie szacuje prawdopodobieństwo, że dany tekst jest poprawnym tekstem w języku polskim, a nie ciągiem losowych znaków. Mając wystarczająco dobrą funkcję, wystarczy posortować wyniki `deszyfruj_tekst` uruchomionej na wszystkich możliwych ziarnach, w kolejności malejących prawdopodobieństw. Przykładami prostych funkcji, które zagwarantują, że poprawny tekst będzie jednym z pierwszych, są:

- Zliczanie kilku częstych słów, na przykład spójników.
- Sprawdzanie częstotliwości samogłosek.

Zadanie D

Podstawową obserwacją w tym zadaniu jest fakt, że wykonanie tej samej operacji dwa razy pod rząd w ogóle nie zmienia układu zer i jedynek. Stąd dla każdego stanu początkowego najkrótszy ciąg operacji, który zamienia wszystkie jedynki na zera, nie może zawierać dwóch identycznych operacji pod rząd. Wobec tego, musi się on składać z obydwu operacji stosowanych naprzemiennie. Ponadto, ostatnią operacją musi być zamiana na pierwszej pozycji od prawej – operacja drugiego typu nie może doprowadzić do otrzymania samych zer.

Właśnie pokazaliśmy, że dla każdego początkowego układu zer i jedynek istnieje dokładnie jedno najszybsze rozwiązanie. Zauważmy, że jeśli zaczniemy od ciągu samych zer i wykonamy ten ciąg operacji od końca, to dostaniemy dokładnie ten sam początkowy układ. Wobec tego, aby otrzymać układ, dla którego najkrótsze rozwiązanie ma długość N, należy zacząć od samych zer i N razy wykonywać raz pierwszą, raz drugą operację.

Teraz wystarczy tylko wydajnie symulować ciąg naprzemiennych operacji (N może być rzędu 10^{18}). Pomoże nam w tym następujące obserwacje:

- Jeśli zaczynamy od stanu, w którym pierwsza cyfra od prawej to zero, to wykonanie pierwszej, a potem drugiej operacji jest zamianą na pozycji drugiej od prawej, np.: `10` $\rightarrow$ `11` $\rightarrow$ `01`. Wobec tego te dwie operacje (pierwszego, a potem drugiego typu) efektywnie wykonają operację pierwszą na ciągu cyfr, w którym pomijamy pozycję pierwszą od prawej.

- Natomiast jeśli pierwsza cyfra od prawej to jeden, to pierwsza operacja sprawi, że na pierwszej pozycji będzie zero, natomiast druga operacja zmieni pozycję na lewo od pierwszej jedynki od prawej. Na przykład $\texttt{11} \to \texttt{10} \to \texttt{110}$. W tym wypadku efektywnie wykonaliśmy operację typu drugiego na ciągu znaków o jeden krótszym (znowu, zapominamy o pierwszej cyfrze z prawej).

Oczywiście powyższy argument można iterować, aby przewidzieć, co się dzieje nie tylko z drugą, ale i kolejnymi cyframi od prawej. Stąd widać, że k-ta cyfra od prawej zmienia się co 2^k naprzemiennych operacji.

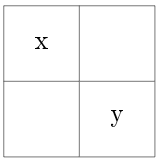
Korzystając z powyższego faktu, można zasymulować N naprzemiennych operacji w czasie $\mathcal{O}(\log N)$.

Zadanie E

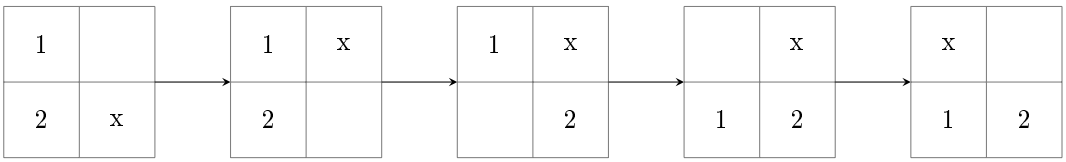
Układanie kafelków podzielimy na trzy części. W pierwszej będziemy przesuwać, po kolei, na swoje miejsce kafelki o numerach od 1 do $(n-2) \cdot m$ (czyli kafelki, których miejsce docelowe nie jest w dwóch ostatnich rzędach). Następnie będziemy układali dwa ostatnie wiersze na raz, od lewej do prawej, aż zostaną dwa ostatnie kafelki. W ostatniej fazie poprawimy ostatnie dwa kafelki.

Pierwszą fazę można zaimplementować następująco. Załóżmy, że chcemy przesunąć kafelek o numerze x:

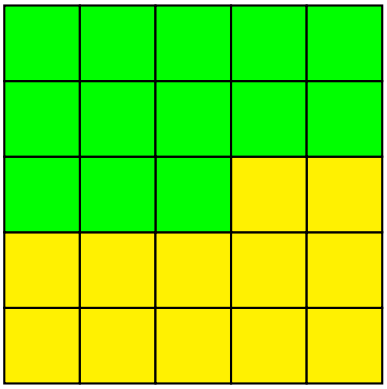
1. Przesuń kafelki tak, żeby pola na prawo i na dole od x były puste:

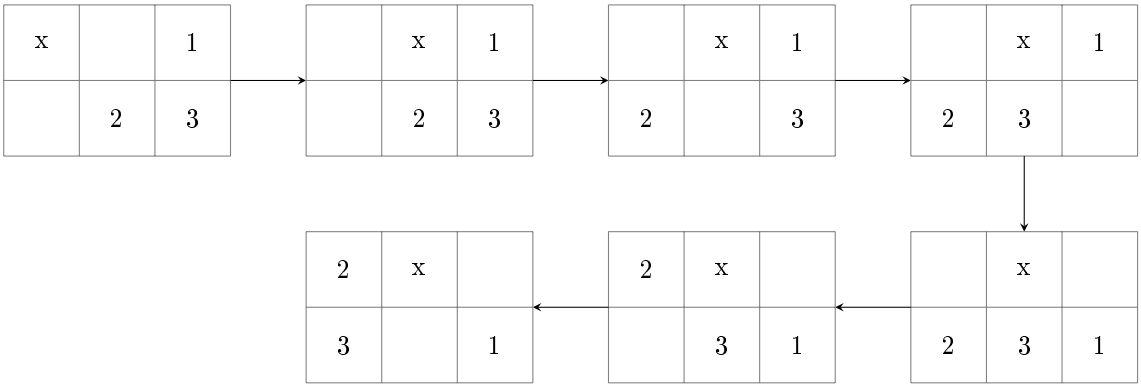
   Możemy to uzyskać przy pomocy dowolnego algorytmu przeszukania, np. BFS'em, szukając ścieżki od pustego pola do sąsiada x. Najpierw przesuniemy pierwsze puste pole, a następnie drugie. Należy tylko uważać, żeby przy przeszukaniu nie przesuwać już poprawnie ułożonych kafelków o numerach mniejszych niż x ani kafelka x. Jeżeli kafelek przylega do krawędzi, to najpierw musimy go odsunąć od niej.

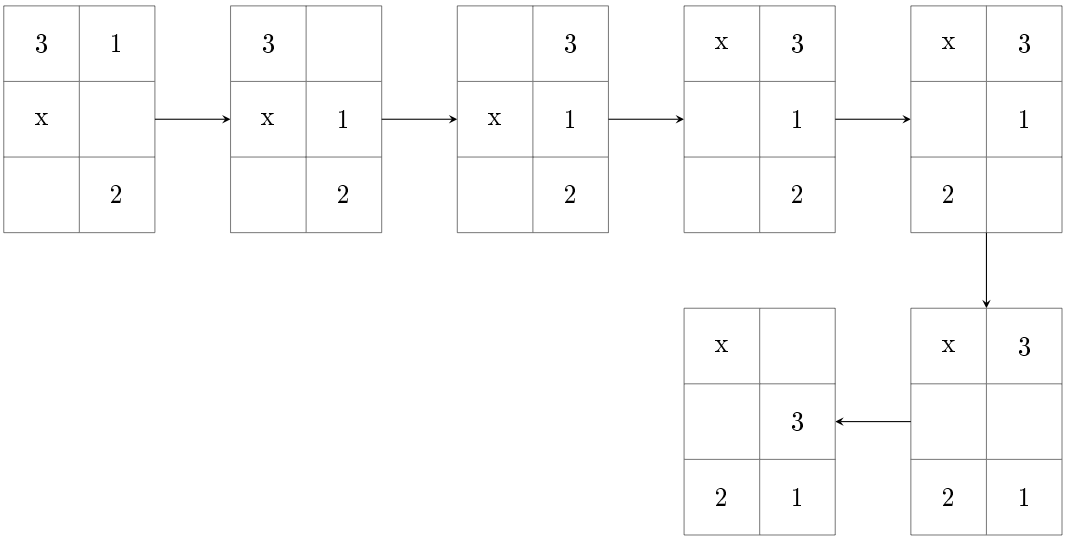
2. Teraz musimy przesunąć x na jego docelowe pole. Osiągniemy to, wykonując najpierw ruchy w poziomie, a następnie w pionie. Z uwagi na to, że naprawiamy planszę po kolei, to kafelki przesunięte do tej pory tworzą w pełni poprawnie ułożone wiersze, a tylko najniższy, aktualnie układany wiersz, może nie być w pełni poprawny (ale cały jego początek jest ułożony):



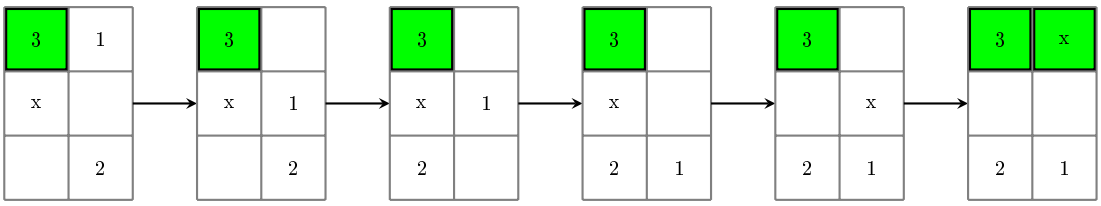
Przesuwając więc x w takiej kolejności nigdy nie najedziemy na kafelek o mniejszym numerze.
Przesunięcie w lewo możemy wykonać następująco:



Przesunięcie w górę możemy wykonać następująco:



Jedyny problem pojawi się, gdy kafelek x jest ostatni w swoim wierszu. Możemy jednak doprowadzić go najpierw powyższymi ruchami, tak żeby był na lewo i na dół od docelowego miejsca, a następnie naprawić ułożenie w następujących ruchach:



Gdy zostaną do ułożenia dwa ostatnie wiersze, to przechodzimy do fazy drugiej i będziemy układać kafeki kolumnami od lewej do prawej. Najpierw układamy kafelek w wierszu przedostatnim, a następnie w ostatnim. Algorytm jest podobny do tego w fazie pierwszej.

Na sam koniec zostaną dwa ostatnie kafelki do ułożenia, które możemy poprzestawiać w kilku krokach (w najgorszym przypadku dwa ostatnie kafelki będą zamienione miejscami)

Do ułożenia całości potrzebujemy przestawić każdy kafelek $(n \cdot m - 2)$. Przestawienie jednego wymaga przesunięcia pustych pól $(2 \cdot (n+m))$, być może odsunięcia od krawędzi (4) i przesunięcie na poprawne miejsce $(6 \cdot (n+m))$ i ewentualne poprawienie, jeżeli docelowe miejsce jest ostatniej kolumnie (5).

Całość wymaga zatem $(n \cdot m - 2) \cdot \big(2 \cdot (n+m) + 4 + 6 \cdot (n+m) + 5\big) = (8 \cdot 8 - 2) \cdot \big(2 \cdot (8+8) + 4 + 6 \cdot (8+8) + 5\big) = 8494 \ll 100\,000$

Zadanie F

Aby rozwiązać to zadanie zbudujemy strukturę danych, która będzie przechowywać multizbiory liczb i udostępniać następujące operacje na nich:

1. Dla liczby b i multizbioru A znajdź takie $a \in A$, że $a \oplus b = \max_{c \in A} c \oplus b$.
2. Dla liczby b i multizbioru A znajdź wszystkie takie $a \in A$, że $a \oplus b > \max_{c \in A} c \oplus b$.
3. Dodaj element b do multizbioru A.

Przy pomocy powyższych operacji można rozwiązać zadanie w następujący sposób. Podczas przeglądania krawędzi w kolejności ich odśnieżania będziemy utrzymywać multizbiory etykiet odpowiadające graczom znajdującym się w poszczególnych spójnych składowych. Ponadto dla każdego gracza utrzymujemy największego xora wewnątrz spójnej składowej. W momencie, gdy odśnieżona zostaje krawędź, która łączy dwie różne spójne składowe o multizbiorach etykiet A i B (bez straty ogólności $|B| < |A|$), wykonujemy następujące operacje:

1. Dla każdego elementu $b \in B$, za pomocą pierwszej operacji sprawdzamy, czy odśnieżenie nowej drogi poprawia wynik gracza o etykiecie b.
2. Dla każdego elementu $b \in B$, za pomocą drugiej operacji tworzymy listę etykiet graczy z A, dla których b poprawia wynik. Następnie łączymy te listy dla wszystkich $b \in B$ oraz uaktualniamy odpowiednie wyniki.
3. Kolejno dodajemy elementy ze zbioru B do zbioru A za pomocą trzeciej operacji.

Zauważmy, że przy takim sposobie użycia struktury danych, element b jest zmienną losową niezależną od aktualnego zbioru A podczas każdej operacji. Teraz, korzystając z powyższej obserwacji, pokażemy, że odpowiednią strukturę danych można wydajnie zaimplementować.

Nasza struktura danych będzie przechowywała każdy multizbiór osobno. Multizbiór A będzie przechowywany w 2^k kubełkach indeksowanych liczbami od 0 do $2^k - 1$. Kubełek x będzie zawierał te elementy z A, których k najbardziej znaczących bitów zapisuje liczbę x. Parametr k będzie dobrany w taki sposób, żeby żaden kubełek nie jest pusty. Przy czym gdy tylko (w wyniku dodania elementu) pojawi się możliwość zwiększenia k przy zachowaniu tego warunku, struktura danych przetwarza cały zbiór A i buduje kubełki na nowo w czasie $\mathcal{O}(|A|)$ dla $k' = k + 1$. Ponieważ multizbiory (a więc i k) mogą tylko rosnąć, każdy element dodany do struktury będzie brał udział w takiej przebudowie co najwyżej $\log N$ razy (zawsze mamy $k \leq \log N$). To sprawia, że sumaryczny koszt operacji trzeciego typu to $\mathcal{O}(N \log N)$

W celu analizy pierwszych dwóch operacji, pochylmy się nad reprezentacją A podzieloną na 2^k niepustych kubełków. Po pierwsze, każdy z elementów A ma najlepszy xor równy co najmniej $2^{40} - 2^{40-k}$ – parując elementy z kubełków o przeciwnych etykietach możemy otrzymać k najstarszych bitów równych jeden w xorze. Co więcej, parowanie elementów, które nie należą do takiej pary kubełków da wynik gorszy niż $2^{40} - 2^{40-k}$, ponieważ jeden z k najbardziej znaczących bitów będzie taki sam. Dlatego, aby przetworzyć zapytania pierwszego i drugiego typu wystarczy przejrzeć dokładnie jeden kubełek w reprezentacji A – ten o etykiecie przeciwnej (zanegowanej) niż najstarsze k bitów b. Przy czym, ponieważ b jest wybrane jednostajnie oraz niezależnie od A, prawdopodobieństwo przejrzenia wszystkich kubełków przechowujących A jest równe. Dlatego oczekiwany koszt tej operacji to $\mathcal{O}\left(\frac{\mathbb{E}[|A|]}{2^k}\right)$. Teraz wystarczy tylko oszacować wartość oczekiwaną $|A|$, który wypełnia 2^k kubełków, ale nie wypełnia 2^{k+1}.

Ograniczymy ją z góry przez wartość oczekiwaną rozmiaru rozmiaru zbioru A w momencie, gdy zaczyna wypełniać wszystkie 2^{k+1} kubełków (moment, gdy k jest zwiększane o jeden, a reprezentacja zbioru A jest budowana od nowa). Ponieważ elementy b dodawane do A zawsze są niezależne od dotychczasowych elementów A oraz mają rozkład jednostajny spośród wszystkich dodanych wartości, b trafia do jednego z 2^{k+1} kubełków wybranego jednostajnie i niezależnie od zbioru dotychczasowo zapełnionych kubełków. W przypadku takiego problemu oczekiwana liczba elementów, które muszą się znaleźć w zbiorze A, aby w każdym z 2^{k+1} znalazł się przynajmniej jeden element, to $\mathcal{O}\left((k+1)2^{k+1}\right)$ (problem ten jest znany jako Problem Kolekcjonera Kuponów). Wobec tego oczekiwany koszt pojedynczej pierwszej i drugiej operacji to $\mathcal{O}(k)$ (czyli $\mathcal{O}(\log N)$).

Przypomnijmy sobie, każdy element b dla którego wykonywaliśmy pierwsze dwie operacje, był następnie dodawany do zbioru A. Ponadto, pochodził on z mniejszego multizbioru B. Stąd na każdym elemencie wykonujemy co najwyżej $\log N$ operacji pierwszego i $\log N$ drugiego typu. Wobec tego, sumaryczny oczekiwany koszt tych operacji to $\mathcal{O}\left(N \log^2 N\right)$. W ten sposób otrzymujemy rozwiązanie o złożoności $\mathcal{O}\left(N \log^2 N\right)$.

Zadanie G

W tym zadaniu mieliśmy dany graf, w którym każdy wierzchołek miał przydzielony (inny niż inne wierzchołki) kolor, a każda krawędź miała dwa różnokolorowe końce. Operacje, które mogliśmy wykonywać, polegały na wybraniu wierzchołka, a następnie "obróceniu go, razem z przyczepionymi do niego krawędziami" – oznacza to, że krawędzie przyczepione do niego zamieniały się kolejno miejscami, tak że pierwsza wskakiwała na miejsce drugiej, ta na miejsce trzeciej i tak dalej, aż

ostatnia wskoczyła na miejsce pierwszej. Krawędzie nie zmieniały kierunku z punktu widzenia wierzchołka, który obracaliśmy.

Okazuje się, że jeśli w ułożonej zagadce weźmiemy sobie pewną krawędź i obrócimy ją, to nie będziemy w stanie powrócić do pierwotnego ustawienia za pomocą legalnych obrotów. Podobnie, jeśli zamienimy ze sobą miejscami dwie krawędzie, to nie zawsze będziemy w stanie za pomocą legalnych obrotów powrócić do pierwotnego ustawienia. Sprawdźmy od czego to wszystko zależy.

Rozwiązanie zadania wymaga podstawowej wiedzy o parzystości permutacji. Przede wszystkim należy wiedzieć, że złożenie dwóch permutacji parzystych daje permutację parzystą, złożenie dwóch nieparzystych też daje parzystą, ale złożenie parzystej z nieparzystą daje permutację nieparzystą (wszystko tak, jak dodawanie). W naszym grafie będziemy interpretować ułożenie naszych krawędzi jako permutację.

Na początku zweryfikujmy wszystkie przypadki z których nie da się dojść do poprawnego ustawienia. Zacznijmy od sprawdzenia czy skierowanie krawędzi jest dobre, a potem sprawdzimy poprawność ich ustawienia.

1. Sprawdzanie skierowania krawędzi

   (a) Graf dwudzielny

   Jeżeli graf jest dwudzielny, to możemy podzielić wszystkie jego wierzchołki na dwa zbiory A i B. Jeżeli któraś z krawędzi będzie obrócona odwrotnie, to zauważmy, że nie jesteśmy w stanie obrócić jej za pomocą legalnych ruchów. Wtedy odpowiedzią do zadania jest `NIE`.

   (b) Graf niedwudzielny

   Przekolorujmy każdą krawędź na dwa kolory, powiedzmy czarny i biały, a niech czarny koniec będzie ten, który ma wyższy numer koloru. Możemy teraz policzyć ile krawędzi jest ułożonych poprawnie (ma czarny kolor przy wyższym numerze wierzchołka), a ile z nich niepoprawnie. Zauważmy teraz, że jeśli liczba niepoprawnie ułożonych krawędzi jest nieparzysta, to po każdym obrocie zostanie ona nieparzysta. Fakt ten można zweryfikować na przykład tak, że jeśli wykonamy jeden obrót, to zawsze parzysta liczba krawędzi zmieni swój stan poprawności. Zatem jeśli liczba niepoprawnie ułożonych krawędzi jest nieparzysta, odpowiedzią do zadania jest `NIE`.

2. Sprawdzanie położenia krawędzi

   (a) Graf jest gwiazdą (drzewem o $n-1$ liściach)

   Tutaj jedyny obrót jaki jesteśmy w stanie wykonać to obrót wokół środka gwiazdy. Łatwo można zweryfikować czy kolejność krawędzi pasuje do kolejności wierzchołków.

   (b) Graf zawierający tylko wierzchołki o nieparzystych stopniach

   Zauważmy, że w tym przypadku każda permutacja wynikająca z obrotu wierzchołka jest cyklem o nieparzystej długości, a zatem jest parzystą permutacją. W takim razie, jeśli ułożenie krawędzi jest nieparzystą permutacją, to nigdy nie dojdziemy za pomocą obrotów do poprawnego ułożenia, które jest parzystą permutacją. Wtedy należy wypisać `NIE`.

   (c) Graf zawierający wierzchołek o parzystym stopniu

   W tym przypadku możemy przestawić krawędzie do każdego możliwego ustawienia.

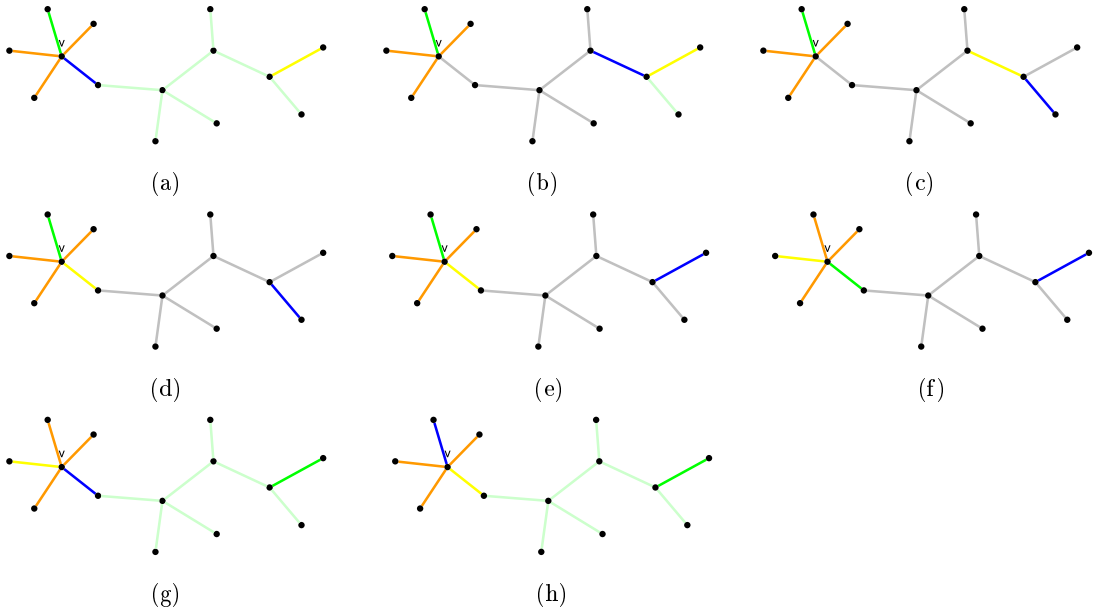


Figure 1: Symulacja ruchów z Algorytmu 1

W powyższych przypadkach uzasadniliśmy czemu z niektórych ustawień krawędzi nie jesteśmy w stanie osiągnąć stanu początkowego. Aby dokończyć dowód poprawności wskażemy algorytmy, które pozwolą nam osiągnąć stan początkowy w pozostałych sytuacjach.

Algorytm 1. Zamiana miejscami trzech krawędzi Zakładamy, że krawędzie niebieska i zielona leżą przy jednym wierzchołku, a żółta nie (taki zestaw krawędzi jesteśmy w stanie znaleźć tylko jeśli nasz graf nie jest gwiazdą). Tutaj naszym celem jest zamienienie za sobą miejscami trzech krawędzi, oznaczonych na rysunku kolorami zielonym, niebieskim i żółtym. Chcielibyśmy przy tym nie zmienić ustawienia żadnej z pozostałych krawędzi.

Na początku zamieńmy ze sobą krawędzie niebieską i żółtą, dbając tylko o to, aby nie ruszyć żadnej z pomarańczowych krawędzi (jasno zielone krawędzie zmieniły kolor na szary, gdyż tymczasowo zmieniły one swoje położenie). Zauważmy, że możemy to zrobić w prosty sposób, obracając odpowiednią liczbę razy odpowiednie wierzchołki na ścieżce pomiędzy tymi krawędziami. Zamiana jest zobrazowana na obrazkach od (a) do (e).

Kluczowym spostrzeżeniem teraz jest to, że gdybyśmy wykonali teraz dokładnie taką samą sekwencję ruchów, w odwrotnej kolejności, wszystkie krawędzie wróciłyby na swoje miejsca.

Idea naszego algorytmu jest taka – podmienimy teraz krawędź żółtą z zieloną, po czym wykonamy całą sekwencję ruchów od tyłu. W ten sposób wszystkie szare krawędzie wrócą na swoje miejsca (zamienią się z powrotem na jasno zielone), a my zamieniliśmy wybrane przez nas krawędzie miejscami!

Używając powyższego algorytmu możemy w prosty sposób zamienić miejscami trzy wspomniane wyżej krawędzie. Jeśli za to chcielibyśmy zamienić miejscami trzy wybrane przez siebie krawędzie, możemy po prostu ustawić je jak wyżej za pomocą dowolnej sekwencji ruchów, wykonać algorytm,

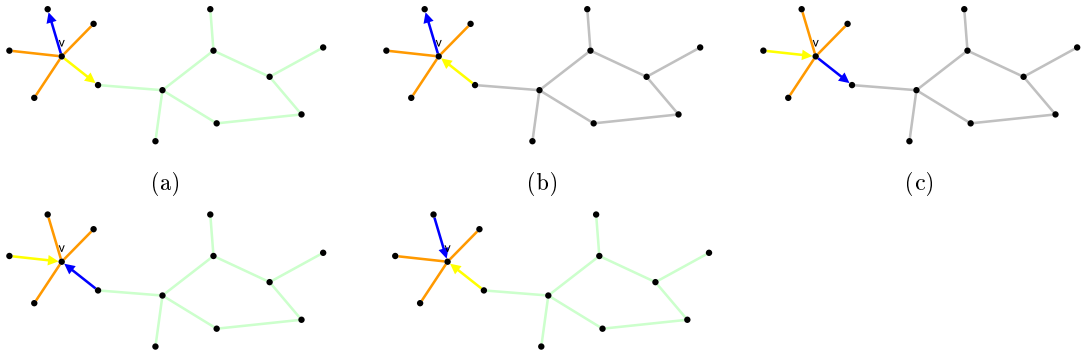


Figure 2: Symulacja ruchów z Algorytmu 2

po czym cofnąć wspomnianą sekwencję ruchów. W ten sposób jesteśmy w stanie nałożyć na nasz graf dowolną parzystą permutację, a przez to, jeśli permutacja krawędzi była pierwotnie parzysta, to możemy doprowadzić krawędzie do stanu początkowego.

W przypadku gdy nasz graf ma jakiś wierzchołek stopnia parzystego, a graf jest permutacją nieparzystą, sytuacja jest bardzo prosta – wystarczy wykonać jeden obrót tego wierzchołka, co sprawi, że permutacja stanie się parzysta, a wtedy jesteśmy już w stanie ułożyć graf.

Teraz zajmijmy się obracaniem krawędzi. Użyjemy do tego algorytmu, który obraca dwie sąsiadujące ze sobą krawędzie w grafie niedwudzielnym.

Algorytm 2. Obrót dwóch sąsiadujących krawędzi w grafie niedwudzielnym Skoro graf nie jest dwudzielny, to możemy znaleźć w nim cykl o nieparzystej długości.

Wybierzmy sobie dwie krawędzie przylegające do jednego wierzchołka, które będziemy chcieli obrócić. Znajdźmy też cykl nieparzystej długości, a za jego pomocą ścieżkę o nieparzystej długości, która kończy się i zaczyna na jednej z wybranych krawędzi (na rysunku jest to krawędź żółta). Kierunki strzałek na rysunku oznaczają ich skierowanie, gdyż teraz właśnie zajmujemy się jego zmianą.

Analogicznie do poprzedniego algorytmu, wykonamy teraz ciąg zmian, który wprowadza oczekiwaną przez nas modyfikację, a następnie po wykonaniu podmiany, cofniemy wszystkie ruchy.

Jak widać na rysunku, w pierwszej kolejności będziemy przesuwać żółtą krawędź po ścieżce nieparzystej długości tak długo aż wróci ona obrócona na swoje miejsce (rysunek (b)). Ścieżka ma nieparzystą długość, więc krawędź wróci obrócona. Następnie możemy wykonać podmianę żółtej krawędzi na niebieską, po czym cofnąć wszystkie wykonane uprzednio ruchy. W ten sposób obróci się też i niebieska krawędź. Po ostatecznym przesunięciu żółtej krawędzi na swoje miejsce, otrzymaliśmy oczekiwany obrót dwóch sąsiadujących ze sobą krawędzi.

Zauważmy teraz, że jeżeli chcielibyśmy wykonać obrót dwóch krawędzi, które nie sąsiadują ze sobą, możemy na przykład ustawić je obok siebie używając dowolnej sekwencji ruchów, obrócić za pomocą algorytmu, a następnie cofnąć pierwotne ruchy, co sprawi, że tylko te dwie krawędzie zmieniły swoje ustawienie.

Powyższe algorytmy mają jeszcze kilka przypadków do doprecyzowania, ale analizuje się je w podobny sposób.

Podsumowując: rozwiązanie zadania (poza dowodami poprawności) wymagało:

- sprawdzenia czy graf jest gwiazdą (oraz ewentualnego sprawdzenia czy dwie gwiazdy mają krawędzie w tej samej kolejności),
- sprawdzenia dwudzielności grafu (z ewentualnym sprawdzeniem czy konkretne wierzchołki leżą po tej samej stronie),
- sprawdzenia czy liczba krawędzi o niepokrywającym się skierowaniu jest nieparzysta,
- sprawdzenia czy graf zawiera wierzchołek o stopniu parzystym,
- sprawdzenia czy permutacja krawędzi grafu jest parzysta.

Implementacja każdego z powyższych warunków nie wymaga znajomości zaawansowanej algorytmiki.

Zadanie H

Zauważmy, że $x \oplus y \leq x + y$ dla dowolnych nieujemnych liczb x oraz y. Dlatego zamieniając dwie karty w jedną Jaś nigdy nie pogarsza swojego wyniku. Wobec tego istnieje optymalne rozwiązanie, w którym Jaś zostaje z tylko jedną kartą. Ponieważ xor jest łączny, wartość tej ostatniej karty musi być równa xorowi wszystkich kart, które Jaś początkowo miał na ręce.

Zadanie I

W tym zadaniu chcemy wybrać kolejność specjalnych punktów położonych na prostej tak, aby suma odległości pomiędzy dwoma kolejnymi punktami w ciągu była jak największa.

Niech $p_1, p_2, \ldots, p_N$ oznaczają posortowane rosnąco pozycje na prostej kolejnych specjalnych punktów. Zamiast patrzeć bezpośrednio na odległości, zauważmy, że przebyć odcinek pomiędzy punktami p_1 i p_2 możemy maksymalnie dwa razy, najpierw docierając do p_1 z jakiegoś wierzchołka na prawo od niego, a następnie wychodząc z niego. Pomiędzy p_2 a p_3 możemy już przejść 4 razy, pomiędzy p_3 i p_4 już 6, i tak dalej, aż dotrzemy do środkowego punktu(ów). Dalej schemat się odwraca i liczba możliwości przejść zaczyna maleć. Bardziej formalnie, do każdego punktu po jednej ze stron krawędzi możemy raz wejść i raz wyjść, więc maksymalna liczba odwiedzin danego odcinka to dwukrotność minimum z liczby wierzchołków po prawej i po lewej stronie. Mamy dwa przypadki:

- N jest parzyste:

  Popatrzmy na środkową krawędź – odcinek między punktami $p_{\frac{N}{2}}$ i $p_{\frac{N}{2}+1}$. Niezależnie od wyboru punktu początkowego i końcowego, nie jesteśmy w stanie przejść tą krawędzią więcej niż $N - 1$ razy (tyle jest wszystkich skoków, które wykonamy). Jest to jedyny odcinek, dla którego maskymalna możliwa liczba przejść jest inna niż dwukrotność mniejszej z liczb wierzchołków po jednej ze stron. Zacznijmy od wierzchołka $p_{\frac{N}{2}}$, przejdźmy z niego do punktu p_N, potem wróćmy do $p_{\frac{N}{2}-1}$, dalej p_{N-1}, i tak dalej, aż dotrzemy do $p_{\frac{N}{2}+1}$. Łatwo sprawdzić, że taka ścieżka przechodzi przez każdy z odcinków maksymalną możliwą liczbę razy, więc jest optymalna.

- N jest nieparzyste:

  Tutaj sytuacja wygląda bardzo podobnie. Tym razem jednak możemy przejść każdy odcinek maksymalnie tyle razy, ile wskazuje nam znaleziony wcześniej wzór. Popatrzmy na dwa środkowe odcinki, których jednym z końców jest $p_{\frac{N+1}{2}}$. Każdego z osobna można przejść maksymalnie $N-1$ razy jeśli zaczniemy w punkcie po stronie z większą liczbą wierzchołków i za każdym razem będziemy przechodzić przez tą krawędź, zmieniając stronę. Zauważmy, że tylko środkowy punkt znajduje się po „większej stronie" dla obu tych odcinków. Co więcej, jeśli zdecydujemy się odwiedzić następnie punkt po jednej ze stron, ominiemy drugą ze środkowych krawędzi – jedną z nich możemy przejść co najwyżej $N-2$ razy! Zacznijmy od środkowego punktu, dalej odwiedźmy punkt p_N, potem $p_{\frac{N+1}{2}-1}$, p_{N-1}, itd. Wszystkie odcinki oprócz tego pomiędzy punktami $p_{\frac{N+1}{2}}$ i $p_{\frac{N+1}{2}-1}$ odwiedzimy maksymalną możliwą liczbę razy. Może się jednak okazać, że odcinek pomiędzy $p_{\frac{N+1}{2}}$ i $p_{\frac{N+1}{2}+1}$ jest od niego krótszy – wtedy wystarczy odbić lustrzanie ścieżkę.

Zadanie J

W tym zadaniu mieliśmy dany jeden wiersz obrazka logicznego, w którym mamy określone już zamalowanie niektórych pól (czy będą one pełne czy puste), a naszym zadaniem jest sprawdzenie dla pozostałych pól, czy też możemy już deterministycznie określić ich zamalowanie.

Pomysł na rozwiązanie jest prosty – dla każdego nieokreślonego jeszcze pola możemy spróbować określić czy będzie ono pełne czy puste, a następnie sprawdzić czy istnieje jakiekolwiek zamalowanie, które spełnia to oraz poprzednie kryteria.

Teraz skupmy się na sprawdzeniu czy dla danego układu istnieje jakieś poprawne zamalowanie. Można to zweryfikować za pomocą programu dynamicznego. Niech $d_1, \ldots, d_K$ oznaczają długości kolejnych odcinków, które chcemy zamalować, a pola do zamalowania będą oznaczone numerami $1, \ldots, N$. Niech stan $DP[i][j]$ oznacza czy jesteśmy w stanie zamalować pola od 1 do i tak, aby na obrazku zamalowane były dokładnie odcinki $d_1, \ldots, d_j$, w tejże kolejności. Stany programu dynamicznego $DP[i][j]$ możemy łatwo aktualizować, poniższym sposobem.

- Jeśli chcemy zostawić pole i niezamalowane, to musimy sprawdzić czy może ono być niezamalowane, oraz czy dało się zamalować pola od 1 do $i-1$ za pomocą j odcinków (czyli czy prawdziwe jest $DP[i-1][j]$). Wtedy $DP[i][j]$ jest prawdziwe.

- Jeśli chcemy aby pole i zostało zamalowane, to znaczy, że końcowy fragment o długości d_j musi zostać zamalowany. Zatem musimy sprawdzić czy pola od $i-d_j+1$ do i mogą być zamalowane, czy pole $i-d_j$ może zostać puste (bo musimy mieć pole przerwy pomiędzy zamalowanymi paskami) oraz czy dało się zamalować pola od 1 do $i-d_j-1$ za pomocą $j-1$ odcinków (czyli czy prawdziwe jest $DP[i-d_j-1][j-1]$). Wtedy $DP[i][j]$ też jest prawdziwe.

Sprawdzenie dla pewnego przedziału, czy wszystkie kratki znajdujące się w nim mogą zostać zamalowane, można zrobić na przykład za pomocą sum prefiksowych, zliczających pola które mogą zostać zamalowane. Program dynamiczny napisany z takim usprawnieniem działa w czasie $O(N \cdot K)$, a zatem cały program (sprawdzający możliwości dla wszystkich pól po kolei) działa w czasie $O(N^2 \cdot K)$, co mieściło się w wyznaczonych limitach.

Warto dodać, że powyższe rozwiązanie można jeszcze usprawnić do takiego, które działa w sumarycznym czasie $O(N \cdot K)$. Aby to zrobić możemy policzyć sobie program dynamiczny dla

każdego prefiksu planszy (czyli taki jak wyżej), a dodatkowo podobny program dla każdego sufiksu planszy (analogiczny do powyższego). Oba te programy da się wyliczyć w czasie $O(N \cdot K)$. Mając dane takie programy nie jest konieczne już początkowe sprawdzenie zamalowania kolejnych pól. Teraz sprawdzenie dla każdej komórki czy może ona zostać niezamalowana jest dość proste. Za to dla każdego odcinka d_j jesteśmy w stanie sprawdzić czy może on być położony na każdym możliwym miejscu, a jeśli tak, to możemy zaznaczyć w tych miejscach, że mogą one być zamalowane. Daje nam to sumaryczną złożoność $O(N \cdot K)$.

Zadanie K

Ponieważ dróg jest co najwyżej tyle, co osad, TO graf, który będziemy dzielić na części, jest drzewem lub zawiera dokładnie jeden cykl. Rozpatrzmy najpierw przypadek drzewa.

Rozwiązanie dla drzewa opiera się na programowaniu dynamicznym (tutaj przedstawimy jedynie definicje rozwiązywanych podproblemów, wzory na obliczanie częściowych rozwiązań są dość proste do wyprowadzenia). Zaczniemy od ukorzenienia drzewa w pewnym wierzchołku r. Przez $Dp[v][mask]$ oznaczymy liczbę takich poprawnych podziałów poddrzewa zaczynającego się w v z tym wyjątkiem, że część zawierająca v niekoniecznie ma dostęp do wszystkich surowców, ale do tych zaznaczonych w masce bitowej $mask$. Znając wartości Dp dla synów v, tablicę $Dp[v]$ można obliczyć w czasie liniowym od liczby synów v (wystarczy zacząć od sytuacji jak gdyby v był liściem, a uaktualniać wynik uwzględniając kolejne dzieci v – uwzględnienie kolejnego syna w wyniku wymaga liczby operacji rzędu $(2^3)^2$ – wystarczy się przeiterować po wszystkich parach masek bitowych oznaczających typy wierzchołków).

Teraz rozważmy przypadek grafu zawierającego cykl. Ustalmy dowolną krawędź leżącą na cyklu i oznaczmy jej końce przez a i b. Przez T oznaczmy drzewo powstałe w wyniku usunięcia krawędzi (a, b) z grafu. Poprawne podziały wierzchołków będziemy liczyć w dwóch rozłącznych kategoriach: takich, że wszystkie wierzchołki na ścieżce w drzewie T z a do b należą do jednego lub większej liczby zbiorów.

Wszystkie wierzchołki na ścieżce z a do b należą do jednego zbioru: Aby obliczyć liczbę podziałów w tej kategorii wystarczy zastosować programowanie dynamiczne w poprzedniego punktu dla drzewa T z tą modyfikacją, że nie wolno rozdzielać wierzchołków na ścieżce z a do b.

Wierzchołki na ścieżce z a do b należą do wielu różnych zbiorów: W tym przypadku również posłużymy się takimi podziałami drzewa T, że wszystkie zbiory wierzchołków rozpinają spójne podgrafy oraz zawierają przynajmniej po jednym z każdego surowców. Jednak ograniczając się do drzewa T zapominamy o krawędzi (a, b), czyli o tym, że zbiory zawierające a i b można połączyć. Dlatego drugie ograniczenie w odniesieniu do tych grup należy zrelaksować, tzn. dla każdej pary masek bitowych $mask_a$ oraz $mask_b$ chcemy policzyć liczbę takich podziałów wierzchołków na spójne podzbiory, że:

- Wierzchołki a i b należą do różnych zbiorów.
- Zbiór, w którym jest a (b), zawiera typy wierzchołków zaznaczone w masce $mask_a$ ($mask_b$).
- Zbiory, do których nie należy a ani b, zawierają wierzchołki wszystkich trzech typów.

Takie wartości można uzyskać ukorzeniając T w wierzchołku a oraz, podobnie jak wcześniej, obliczając wartości $Dp'[v][mask][mask_b][cut]$ – liczbę podziałów poddrzewa o korzeniu w v na spójne podzbiory wierzchołków takich, że zbiór wierzchołków z v zawiera typy wierzchołków zaznaczone w $mask$, a zbiór z b te zaznaczone w $mask_b$. Ponadto gdy $cut = 0$, wymagamy by v oraz b były w jednym zbiorze (wtedy $mask = mask_b$), a gdy $cut = 1$, w różnych. Oczywiście ostatnie dwa warunki mają znaczenie tylko wtedy, gdy b jest potomkiem v. W przeciwnym wypadku wystarczy obliczyć zwykłe $Dp[v][mask]$. $Dp'[v]$ można obliczyć analogicznie do $Dp[v]$, przy czym wymaga to liczby operacji rzędu $deg(v) \cdot (2^3)^3 = deg(v) \cdot 512$. Aby otrzymać ostateczny wynik, wystarczy się przeiterować po wszystkich parach masek $mask_a$ oraz $mask_b$, które wspólnie zawierają wszystkie typy wierzchołków i posumować $Dp[a][mask_a][mask_b][1]$.

Ostatecznie otrzymujemy algorytm, którego liczba operacji jest rzędu $512N$.

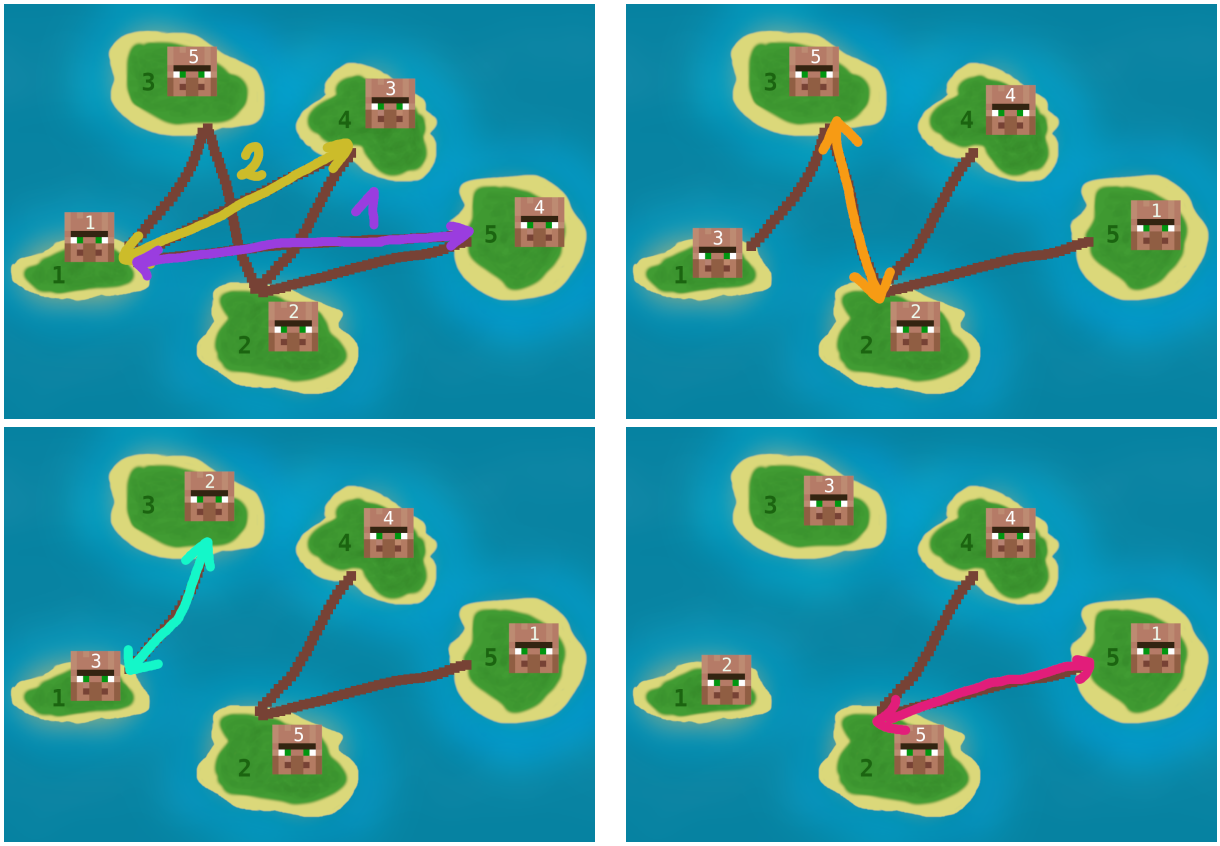
Zadanie L

W tym zadaniu dla podanej daty należało określić jaka pora roku wtedy obowiązuje. Najłatwiej to zrobić pisząc jedną pomocniczą funkcję, która porównuje dwie daty i stwierdza, która z nich jest wcześniej. Taka funkcja może operować na napisach, ale jeszcze prościej jest, gdy operuje ona na liczbach. Należy tylko pamiętać o zamianie wczytanych dat na liczby. Następnie za pomocą tak napisanej funkcji i instrukcji warunkowych wystarczy wypisać nazwę odpowiedniej pory roku.

Zadanie M

Wieśniaków na wyspach możemy reprezentować jako permutację - jeśli wieśniak, który powinien być na wyspie i, znajduje się obecnie na wyspie j to liczba i w permutacji znajdzie się na j-tym miejscu. Wiemy z treści zadania, że dwa elementy z permutacji są na poprawnych miejscach oraz nigdy nie były ruszane.

Załóżmy na razie, że permutacja składa się tylko z tych dwóch nieruszonych elementów o numerach 1 i 2 oraz jednego cyklu, którego kolejnymi elementami są $c_1, c_2, c_3, \ldots, c_k$. Pokażemy, że da się skonstruować ciąg zamian tak, żeby powstała nam permutacja identycznościowa niezależnie od tego, które z zamian elementów są jeszcze dostępne. Zaczniemy od zamiany miejscami elementów na pozycjach 1 i tej, na której znajduje się c_1. Następnie wykonamy zamianę elementu c_1 będącego teraz na pozycji 1 na jego prawidłowe miejsce, na którym jest c_2 (ze względu na cykl). Powtarzamy to, aż na pozycji 1 będzie element c_{k-1}. W ten sposób uda nam się naprawić prawie cały cykl. Na pozycji, na której był element c_1 znajdzie się teraz element 1, a na pozycji 1 element c_{k-1}. Za pomocą kroków przedstawionych na poniższych rysunkach możemy dokończyć naprawę cyklu:



Na koniec elementy 1 i 2 są zamienione miejscami, ale na szczęście nigdy nie używaliśmy zamian tych dwóch pozycji, więc zwyczajnie zamieniamy je miejscami.

A co jeśli cyklów permutacji jest więcej? Rozwiązanie nie różni się zbyt wiele. Zanim zamienimy ze sobą 1 i 2, naprawmy analogicznie następny cykl. Dopiero po naprawie wszystkich pozostałych cykli ewentualnie należy zamienić miejscami pozycje 1 i 2.

Zadanie N

Głównym spostrzerzeniem potrzebnym do rozwiązania zadania jest fakt, że jeśli planujemy kupić jakiegoś robotnika, to opłaca nam się to zrobić jak najwcześniej, gdyż zawsze kosztuje on tyle samo, a im wcześniej go kupimy, tym więcej zysku może on przynieść.

Drugim spostrzeżeniem jest to, że skoro wszystkie wartości na wejściu są z przedziału $[1, 1000]$, to czas zbierania pożywienia nie będzie większy niż 1000 (bo tyle zajmie to jednemu robotnikowi), oraz nigdy nie będziemy potrzebować wiecej niż 1000 robotników (bo są oni w stanie wyprodukować całe potrzebne pożywienie w sekundę).

Rozwiązanie zadania polega na sprawdzeniu po jakim czasie najwcześniej jesteśmy w stanie zebrać odpowiednią ilość pożywienia, zakładając, że kupimy dokładnie x robotników. Możemy zasymulować taką sytuację za pomocą pętli od 1 do 1000, która na początku dokupuje robotnika (jeśli nie mamy wystarczająco pożywienia, możemy stopniowo przesuwać czas, za każdym razem

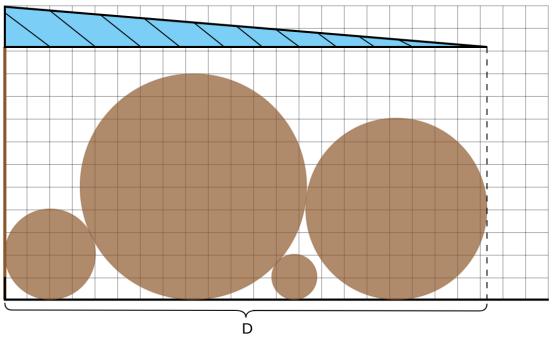
zwiększając liczbę pożywienia o $V \cdot (x-1)$, bo tyle produkują aktualnie nasi robotnicy), a następnie wylicza po jakim czasie byliby oni w stanie wyprodukować pożywienie potrzebne do awansu (są oni w stanie wyprodukować y pożywienia w czasie $\lceil y/(V \cdot x) \rceil$).

Rozwiązanie to działa w czasie liniowym od wielkości danych, ale ze względu na małe limity, akceptowane były również wolniejsze rozwiązania.

Zadanie O

Treść

Chcemy ułożyć na prostej N kół (o różnych promieniach) w danej kolejności, tak by każde kolejne koło przylegało ściśle do jednego z poprzednich kół. Podaj szerokość figury, która jest na końcu sumą wszystkich kół.



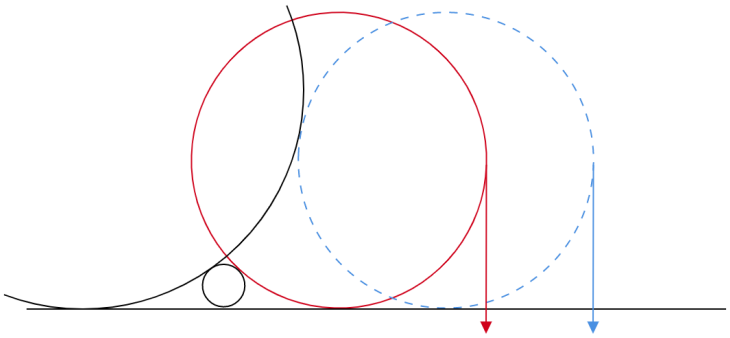
Rozwiązanie

Ponieważ limity w zadaniu nie są za duże ($N \leq 3000$), będziemy szukali rozwiązania działającego w złożoności $\mathcal{O}(N^2)$.

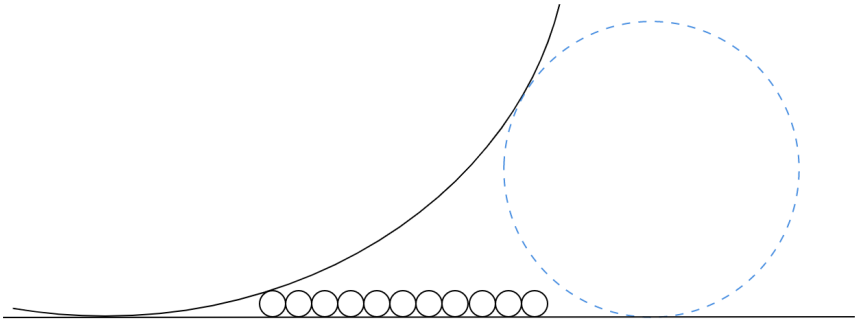
Będziemy kolejno dodawać każde kolejne koło i sprawdzać, w którym miejscu się zatrzyma jeżeli dosuniemy je od prawej strony.

Rozważymy, w którym miejscu zatrzymałoby się koło, jeżeli miałoby stykać się z jakimś z kół, które były wcześniej i mają już jednoznacznie ustaloną pozycję. Pozycją nowego koła, jest maksimum z rozważonych możliwości.

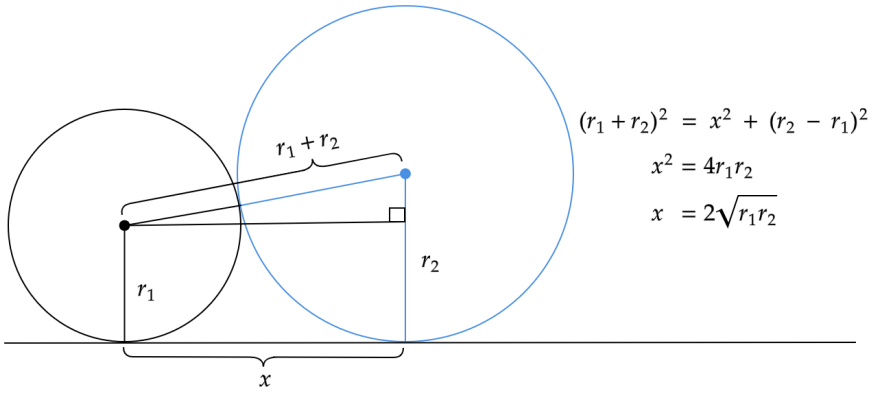
Symulując proces przesuwania koła w lewo w pewnym momencie natrafi ono na przeszkodę (inne koło, albo ścianę magazynu) i się tam zatrzyma. Rozważając wszystkie poprzednie koła, na pewno rozważymy też to, na którym rzeczywiście się zatrzyma. Na pewno też nie postawimy nowego koła za bardzo na prawo.



Zwróćmy uwagę na to, że w skrajnym przypadku musimy rozważyć okrąg, który pojawił się dużo wcześniej.



Za pomocą prostych rachunków i Twierdzenia Pitagorasa wyznaczamy na poziomą odległość między środkami kół.



$$(r_1 + r_2)^2 = x^2 + (r_2 - r_1)^2$$
$$x^2 = 4r_1r_2$$
$$x = 2\sqrt{r_1r_2}$$

Rozwiązania szybsze

Okazuje się, że istnieją rozwiązania działające w czasie $\mathcal{O}(N \log N)$ oraz $\mathcal{O}(N)$, korzystające z obserwacji, że okręgi mogące w przyszłości być jeszcze kandydatami na te kolidujące, mają malejące rozmiary. Jeżeli więc zwiększamy rozmiar dodawanego okręgu, to będzie się on zatrzymywał, na coraz wcześniejszych okręgach.