Rozwiązania zadań
z I etapu XXIX Olimpiady Informatycznej

Tomasz Idziaszek

Domino

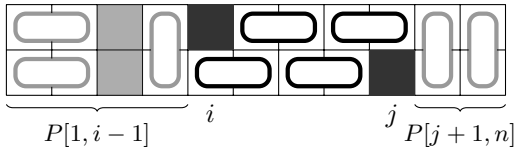
Autor zadania: Karol Pokorski

Na początek pokażemy, że jeśli istnieje prostokąt i takie zamalowanie jego pól, żeby istniało dokładnie m pokryć dominem, to zamalowanie prostokąta o minimalnej szerokości musi spełniać warunek, że każda kolumna nie jest w ogóle zamalowana albo jest zamalowana w całości.

Ponumerujmy kolumny prostokąta rozmiaru $2 \times n$ od lewej do prawej liczbami od 1 do n. Niech $P[i, j]$ oznacza podprostokąt złożony z kolumn o numerach od i do j.

Załóżmy nie wprost, że istnieje jakaś kolumna, w której mamy zamalowane dokładnie jedno pole i niech i będzie najmniejszym numerem takiej kolumny. Wtedy w $P[1, i-1]$ mamy niezamalowanych parzystą liczbę pól $2k$, więc te kolumny muszą być pokryte przez dokładnie k domin. Zatem wolne pole w kolumnie i musi być pokryte przez poziome domino, którego drugi koniec musi pokrywać pole w kolumnie $i + 1$. Tak więc albo w kolumnie $i + 1$ też mamy jedno zamalowane pole, albo jest ona pusta, ale wtedy to drugie pole musi być pokryte przez poziome domino na kolumnach $i + 1$ oraz $i + 2$.

Niech więc j będzie drugim numerem kolumny o zamalowanym dokładnie jednym polu (taka kolumna istnieje, bo pokrycie istnieje tylko dla parzystej liczby zamalowanych pól). Wtedy wszystkie kolumny od $i + 1$ do $j - 1$ są niezamalowane a w $P[i, j]$ jest dokładnie $2(j - i)$ niezamalowanych pól, które mogą być pokryte $j - i$ klockami domina na dokładnie jeden sposób.



Zatem te jednoznacznie pokryte kolumny rozbijają nam nasz prostokąt na dwa kawałki: $P[1, i-1]$ oraz $P[j+1, n]$, które możemy pokryć niezależnie. Nietrudno zauważyć, że jeśli zastąpimy kawałek $P[i, j]$ przez jedną kolumnę zamalowaną w pełni (a więc też jednoznacznie pokrytą i oddzielającą te dwa kawałki), to dostaniemy prostokąt, który też ma dokładnie m pokryć, ale jest krótszy o $j - i$ kolumn, co przeczy minimalności pierwotnego prostokąta. To kończy dowód warunku.

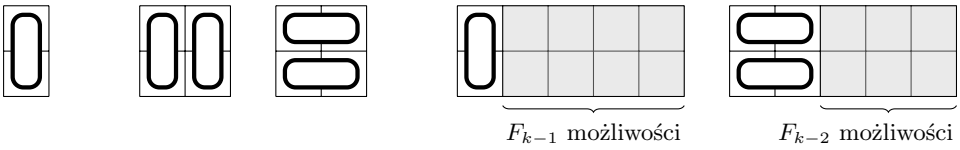
Możemy ten warunek wzmocnić, łatwo pokazując, że ani pierwsza, ani ostatnia kolumna nie mogą być zamalowane (bo można je po prostu odrzucić) oraz żadne dwie kolejne (bo można je zastąpić jedną).

Tak więc, jeśli rozwiązanie istnieje, to składa się z bloków niezamalowanych kolumn, rozdzielonych pojedynczymi kolumnami zamalowanymi w całości.

Liczby Fibonacciego. Rozważmy jeden blok niezamalowanych kolumn, czyli prostokąt o wymiarach $2 \times k$. Jest dosyć znanym faktem, że można go pokryć na F_k sposobów, gdzie F jest ciągiem liczb Fibonacciego zdefiniowanym następująco:

$$F_1 = 1, \quad F_2 = 2, \quad F_k = F_{k-2} + F_{k-1} \text{ dla } k \geqslant 3.$$

Ten fakt można łatwo udowodnić przez indukcję. Dla $k = 1$ mamy jedno pokrycie (domino pionowe), co odpowiada $F_1 = 1$, zaś dla $k = 2$ mamy dwa pokrycia (dwa domina pionowe lub dwa poziome), co odpowiada $F_2 = 2$.



Dla $k \geqslant 3$ patrzymy na pokrycie pierwszej kolumny: jeśli pokrywamy ją dominem pionowym, to pozostałe $k-1$ kolumn pokryjemy (z założenia indukcyjnego) na F_{k-1} sposobów. Natomiast jeśli pokrywamy ją dwoma dominami poziomymi (pokrywając również całą drugą kolumnę), to pozostałe $k-2$ kolumn pokryjemy na F_{k-2} sposobów. Ostatecznie będziemy mieli $F_{k-1} + F_{k-2} = F_k$ pokryć, co kończy dowód.

Ponieważ każdy blok pokrywamy niezależnie, liczba pokryć całego prostokąta jest iloczynem liczby pokryć bloków. Zatem rozwiązanie istnieje wtedy, gdy m jest iloczynem pewnych liczb Fibonacciego.

Jeśli więc mamy rozkład $m = F_{k_1} \cdot F_{k_2} \cdot \ldots \cdot F_{k_s}$, to możliwym rozwiązaniem jest prostokąt o szerokości $n = k_1 + k_2 + \ldots + k_s + s - 1$ i zamalowanych odpowiednich $s - 1$ kolumnach. Zatem szukamy takiego ciągu $k_1, k_2, \ldots, k_s$, który minimalizuje tę sumę.

Rekurencyjne generowanie rozkładów. Możemy rekurencyjnie wygenerować wszystkie ciągi, których iloczyny nie przekraczają m. W poniższej procedurze argument `iloczyn` oznacza iloczyn konstruowanego ciągu, `pos` jest pozycją liczby Fibonacciego, którą aktualnie rozważamy do domnożenia, a `s` jest szerokością konstruowanego prostokąta. Zakładamy, że mamy wygenerowaną tablicę liczb Fibonacciego `f`.

```
ans = defaultdict(lambda: 10**9)

def rek(iloczyn, pos, s):
    ans[iloczyn] = min(ans[iloczyn], s)
    if f[pos] <= m // iloczyn:
        rek(iloczyn * f[pos], pos, s + 1 + pos)
    if pos+1 < len(f):
        rek(iloczyn, pos+1, s)

rek(1, 2, -1)
return ans[m]
```

Co prawda jest to rozwiązanie wykładnicze, ale można sprawdzić eksperymentalnie, że dla $m = 10^9$ w krótkim czasie znajdzie odpowiedź (więc również będzie działać dobrze dla mniejszych liczb, gdyż tak naprawdę generuje wszystkie odpowiedzi dla liczb nie większych niż m). Zatem przejdzie ono drugie podzadanie. Niestety, dla $m = 10^{18}$ jest zbyt wolne.

Rozkłady dla danego m. Możemy jednak wykorzystać fakt, że potrzebujemy znać odpowiedź dla tylko jednej wartości m i generować tylko takie ciągi, których iloczyny dają się rozszerzyć do m, zatem tylko te, które dzielą m:

```
ans = 10**9

def rek(zostalo, pos, s):
    if zostalo == 1:
        ans = min(ans, s)
    else:
        if zostalo % f[pos] == 0:
            rek(zostalo // f[pos], pos, s + 1 + pos)
        if pos+1 < len(f):
            rek(zostalo, pos+1, s)

rek(m, 2, -1)
return ans
```

Ponieważ funkcja „liczba dzielników" nie rośnie zbyt szybko (największa możliwa liczba dzielników dla liczb nie większych niż 10^{18} wynosi 103 680 i występuje dla 897 612 484 786 617 600), a liczb Fibonacciego nie większych niż 10^{18} jest 86, więc liczba przejrzanych par (`zostalo`, `pos`) przez ten algorytm to około 9 milionów. Jeśli zatem przerobić go tak, aby dla danej pary znajdował minimalną odpowiedź `s` oraz dodać spamiętywanie (np. przy pomocy tablicy haszującej), to możemy mieć pewność, że zmieści się w czasie dla każdego $m \leqslant 10^{18}$.

Okazuje się jednak, że te zmiany nie są konieczne i nawet taki niezoptymalizowany program zaliczy wszystkie podzadania. Wynika to z faktu, że liczby Fibonacciego w rozkładzie na czynniki pierwsze mają w większości przypadków co najmniej po jednym unikalnym czynniku pierwszym (czyli takim, którego nie mają inne liczby), a w takich przypadkach obecność (lub nieobecność) takiego czynnika w m jednoznacznie determinuje, czy tę liczbę Fibonacciego dostaniemy w odpowiedzi.

Druk

Autorzy zadania: Jakub Radoszewski, Wojciech Rytter

Ponieważ w zadaniu mamy wypisać wszystkie możliwe długości szablonu, którymi można wydrukować tabliczkę, więc możemy spróbować dwuczęściowego rozwiązania:

- Najpierw wygenerujemy pewien zbiór słów, będących kandydatami na szablon. Nie każde z tych słów musi być szablonem, ale chcemy, aby każdy prawidłowy szablon był w tym zbiorze oraz żeby zbiór ten był możliwie mały.
- Następnie dla każdego słowa ze zbioru niezależnie sprawdzimy, czy rzeczywiście jest ono prawidłowym szablonem.

Na koniec wypiszemy długości znalezionych szablonów. Oczywiście złożoność czasowa rozwiązania będzie zależeć od liczby słów w zbiorze kandydatów oraz szybkości działania procedury sprawdzającej.

Generowanie zbioru kandydatów. Na początek zastanówmy się, jak wygląda pasek pokrywający górne lewe pole tabliczki. Musi być on zaczynać się w tym polu i być albo poziomy, albo pionowy. Mamy zatem dokładnie m kandydatów poziomych na szablon (będących prefiksami górnego wiersza tabliczki) oraz n kandydatów pionowych (będących prefiksami lewej kolumny). Daje nam to razem $n+m-1$ kandydatów.

Możemy jednak zrobić jeszcze jedną, dość oczywistą obserwację. Na tabliczce mamy $n \cdot m$ liter, a każda litera musi być wydrukowana dokładnie raz przez jedno przyłożenie szablonu. Zatem, jeśli szablon ma długość s, to musimy go przyłożyć do tabliczki dokładnie nm/s razy, zatem s musi być dzielnikiem nm. Zatem możemy ze zbioru kandydatów wyrzucić wszystkie słowa, których długości nie dzielą nm.

Można jednak pójść dalej z pomysłem na dzielniki i zrobić ciut mniej oczywistą obserwację, która również będzie wynikać z tego, kiedy paskami o wielkości $s \times 1$ (lub $1 \times s$) jesteśmy w stanie pokryć prostokąt rozmiaru $n \times m$.

W każde pole tabliczki wpiszmy jedną z s liczb ze zbioru $S = \{0, 1, 2, \ldots, s-1\}$. Liczbą w i-tym wierszu i j-tej kolumnie (przy czym wiersze i kolumny numerujemy od 0) tabliczki będzie $(i + j)$ mod s:

$s = 4$

0	1	2	3	0	1	2	3	0	1
1	2	3	0	1	2	3	0	1	2
2	3	0	1	2	3	0	1	2	3
3	0	1	2	3	0	1	2	3	0
0	1	2	3	0	1	2	3	0	1
1	2	3	0	1	2	3	0	1	2

n', m'

Na takiej tabliczce każde przyłożenie paska długości s (pionowe lub poziome) będzie pokrywało wszystkie s liczb ze zbioru S. Tak więc, jeśli tabliczkę da się pokryć szablonem, to każda z liczb musi wystąpić w niej tyle samo razy – dokładnie nm/s.

Załóżmy więc nie wprost, że s dzieli nm, ale nie dzieli ani n ani m. Dopóki $n > s$, to usuwamy z prostokąta s górnych wierszy. Usunięte fragmenty da się pokryć paskami rozmiaru $s \times 1$ i zawierają one wszystkie liczby z S – każdą po m razy. Analogicznie, dopóki $m > s$, to usuwamy z prostokąta s lewych kolumn (takie fragmenty da się pokryć paskami rozmiaru $1 \times s$).

Na końcu zostanie nam prostokącik spełniający $n' < s$, $m' < s$ oraz $n'm'$ dzieli s. Tak więc $n'm' = sx$ dla pewnego $x < r = \min(n', m')$. Warunkiem koniecznym (choć niekoniecznie wystarczającym) istnienia pokrycia jest to, żeby każda liczba ze zbioru S występowała w tym prostokąciku x razy. Ale nietrudno się przekonać, że liczba $r-1$ będzie występowała co najmniej $r > x$ razy. Mamy więc sprzeczność, co kończy dowód, że s musi dzielić n lub m.

Niech d będzie liczbą dzielników liczb n i m (z powtórzeniami). Mamy wtedy do rozważenia d kandydatów. Zważywszy na to, że $n, m \leqslant 1000$, to największa liczba dzielników dla pojedynczej liczby wyniesie tylko 32 (dla liczby 840), tak więc będziemy musieli rozważyć co najwyżej $d \leqslant 64$ kandydatów.

Przypadek tabliczki unarnej. Zanim przejdziemy do ogólnej procedury sprawdzania kandydata, zatrzymajmy się na moment nad przypadkiem szczególnym, gdy cała tabliczka zawiera tylko jedną literę. Wtedy wszyscy kandydaci też będą unarni i wystarczy po prostu sprawdzić, czy paski długości s są w stanie pokryć prostokąt. Ale łatwo pokazać, że jest to możliwe dla każdego kandydata: jeśli s dzieli n, to pokrywamy prostokąt samymi pionowymi paskami, a jeśli s dzieli m to samymi poziomymi.

Zatem jako odpowiedź wypisujemy wszystkie dzielniki liczb n i m.

Sprawdzanie jednego kandydata. Teraz zakładamy, że tabliczka zawiera przynajmniej dwie różne litery. Mając ustalone słowo o długości s mamy sprawdzić, czy jest ono szablonem pokrywającym tabliczkę. Jeśli to słowo jest unarne, to możemy od razu zwrócić odpowiedź negatywną – na pewno nie pokryje tabliczki. Zakładamy zatem, że również w słowie s występują co najmniej dwie różne litery. (Tak naprawdę muszą w nim występować wszystkie litery, które występują na tabliczce, i dokładnie w tych samych proporcjach. Zaimplementowanie tego sprawdzenia pozwala szybko odsiewać dużą część kandydatów.)

Spróbujemy skonstruować pokrycie, przeglądając pola tabliczki wierszami od góry do dołu, a w każdym wierszu od lewej do prawej. Będziemy utrzymywać niezmiennik, że wszystkie przejrzane pola są już pokryte. Gdy aktualnie przeglądane pole jest już pokryte, pomijamy je. Jeśli nie jest pokryte, to musimy je pokryć paskiem, który zaczyna się w tym polu, przy czym należy podjąć decyzję, czy będzie to poziomy czy pionowy pasek. W czasie $O(s)$ możemy sprawdzić oba przyłożenia i jeśli żadne nie jest prawidłowe, to kończymy sprawdzenie z wynikiem negatywnym, a jeśli tylko jedno z nich jest prawidłowe, to zaznaczamy pokryte pola i kontynuujemy.

W końcu jeśli oba przyłożenia są prawidłowe, to musimy albo sprawdzić obie możliwości, albo na podstawie jakiegoś kryterium wybrać którąś z nich. W pierwszym przypadku uzyskamy rozwiązanie potencjalnie wykładnicze, co nie będzie nas satysfakcjonowało. Okazuje się jednak, że istnieje bardzo proste zachłanne kryterium, które tu możemy zastosować: wybieramy zawsze przyłożenie poziome.

Dlaczego to jest poprawne? Oznaczmy przez a pierwszą literę słowa, przez k liczbę liter a na jego początku, a przez b pierwszą literę różną od a (czyli literę na pozycji $k+1$). Jeśli przyłożenie poziome słowa jest poprawne, to w wierszu mamy k niepokrytych liter a, za którymi występuje niepokryta litera b. Jeśli zdecydowalibyśmy się pokryć aktualną literę paskiem pionowym, to dla kolejnych k liter pokrycia poziome byłyby nieprawidłowe, zatem musielibyśmy wykonywać pokrycia pionowe. Ale to nie będzie możliwe dla k-tej litery, co daje sprzeczność.

Tak więc zachłanna procedura jest poprawna i jeśli rozwiązanie istnieje, to znajdzie je w czasie $O(nms)$.

Możemy przyspieszyć tę procedurę. Zauważmy, że tak naprawdę to litera b wymuszała zastosowanie pokrycia poziomego. Możemy więc przeglądnąć każdy wiersz dwa razy. Pierwszy raz będziemy poszukiwać niepokrytych liter b. Dla każdej takiej litery stawiamy poziomy pasek, zaczynając k liter wcześniej (jeśli się nie da, to kończymy). Następnie jeszcze raz przechodzimy wiersz i teraz muszą pozostać niepokryte jedynie litery a, które pokrywamy paskami pionowymi (znowu, jeśli się nie da, to kończymy).

Taka procedura sprawdzająca zadziała w czasie $O(nm)$, tak więc cały algorytm będzie miał złożoność czasową $O(nmd)$.

Impreza krasnali

Autor zadania: Tomasz Idziaszek

Zacznijmy od kilku prostych obserwacji. Załóżmy, że i-ty krasnal miał czapkę o wysokości x. Jego czapka mogła być widziana jedynie przez jego sąsiadów, czyli krasnali o numerach $i-1$ oraz $i+1$ (o ile ci istnieją). W związku z tym liczba x może się pojawić w ciągu wejściowym co najwyżej dwa razy, jako $h_{i-1} = x$ i/lub $h_{i+1} = x$.

Z tego wynika, że dowolna liczba x z przedziału od 1 do n może się pojawić w ciągu 0, 1 lub 2 razy. Przy czym w tym ostatnim przypadku odległość między jej dwoma wystąpieniami musi być równa dokładnie 2. Jeśli któraś z liczb x pojawia się więcej niż dwa razy lub jej dwa wystąpienia mają inną odległość – możemy od razu zwrócić 0 jako odpowiedź, bo zeznania krasnali są sprzeczne.

Rozważmy teraz liczby x, które pojawiają się dwa razy. Jeśli mamy $h_{i-1} = x$ oraz $h_{i+1} = x$, to wiemy, że i-ty krasnal musiał mieć czapkę wysokości x. Takiego krasnala nazwiemy *pewniakiem*.

Brak pewniaków. Załóżmy na początek, że nie ma pewniaków, czyli w ciągu każda liczba pojawia się 0 lub 1 razy. (Tak naprawdę skoro ciąg ma długość n, to każda liczba musi pojawić się w nim dokładnie raz, czyli ciąg wejściowy jest permutacją.) Przez a oznaczymy ciąg możliwych wysokości czapek.

Pierwszy krasnal widzi jedynie czapkę drugiego krasnala, więc ta jest wyznaczona jednoznacznie, zatem $a_2 = h_1$. Z kolei trzeci krasnal widział czapkę drugiego lub czwartego krasnala, ale ponieważ $h_1 \neq h_3$ (bo ciąg wejściowy nie ma powtórzeń), zatem musiał on opisać czapkę czwartego krasnala, więc $a_4 = h_3$. W ten sposób przez indukcję możemy pokazać, że dla dowolnego $i \geqslant 1$ mamy $a_{2i} = h_{2i-1}$, czyli czapki krasnali na pozycjach parzystych są wyznaczone jednoznacznie.

Podobnie możemy iść od tyłu: ostatni krasnal widzi jedynie czapkę przedostatniego, więc $a_{n-1} = h_n$. Znowu przez indukcję pokazujemy, że dla dowolnego $i \geqslant 1$ mamy $a_{n-2i+1} = h_{n-2i+2}$.

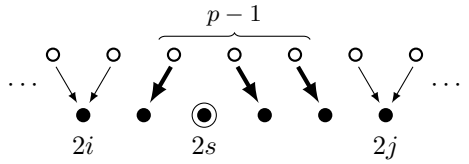
I teraz mamy dwa przypadki w zależności od parzystości n. Dla n parzystego to przejście „od tyłu" wyznacza nam jednoznacznie jak wyglądają czapki krasnali na pozycjach nieparzystych. To pokazuje, że mamy dokładnie jedną odpowiedź: $a = h_2 h_1 h_4 h_3 \ldots h_n h_{n-1}$.

Z kolei dla n nieparzystego z przejścia „od przodu" mamy $a_{n-1} = h_{n-2}$. Z kolei z przejścia „od tyłu" mamy $a_{n-1} = h_n$, co prowadzi do sprzeczności, bo $h_{n-2} \neq h_n$. Tak więc w tym przypadku odpowiedzią jest 0.

Istnieją pewniaki. Niech w ciągu jest więc co najmniej jeden pewniak. Ponieważ krasnale siedzący na pozycjach parzystych widzą jedynie czapki krasnali na pozycjach nieparzystych i na odwrót – krasnale z pozycji nieparzystych widzą jedynie czapki z pozycji parzystych, więc możemy rozdzielić nasz ciąg na dwa, w zależności od parzystości pozycji.

Zajmijmy się więc wyznaczaniem czapek dla pozycji parzystych – dla pozycji nieparzystych algorytm będzie analogiczny. Załóżmy, że na pozycjach $2i$ oraz $2j$ stoją dwaj pewniacy oraz nie ma żadnego innego pewniaka na pozycji parzystej

między nimi. Niech $p = j - i - 1$ będzie liczbą tych parzystych pozycji pomiędzy nimi. Zatem $h_{2i-1} = h_{2i+1}$ oraz $h_{2j-1} = h_{2j+1}$, a poza tym $p - 1$ liczb h_{2k+1} dla $i < k < j - 1$ jest różnych. Na pozycjach $2i + 2, \ldots, 2j - 2$ mamy p krasnali. Dokładnie $p-1$ z nich musi dostać tak czapki, żeby było widoczne tych $p-1$ liczb, a jeden z nich krasnali dostanie *jakąś inną* czapkę.

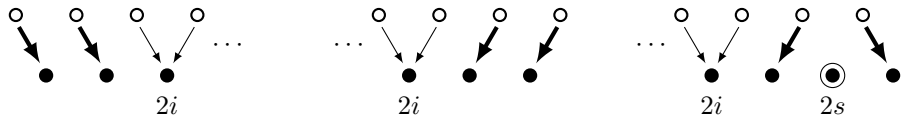


Okazuje się, że dla każdego z p wyborów, który krasnal będzie miał tę inną czapkę, mamy dokładnie jeden sposób, na który możemy przyporządkować czapki pozostałym krasnalom. A konkretnie, jeśli wybierzemy krasnala $2s$, to mamy

$$a_{2k} = \begin{cases} h_{2k+1} & \text{dla } k < s, \\ h_{2k-1} & \text{dla } k > s. \end{cases}$$

Tak więc dla każdego przedziału o długości p pomiędzy dwoma pewniakami o tej samej parzystości mamy p możliwości wyboru oraz pozostaje tam jeden krasnal, któremu trzeba będzie przydzielić jedną z czapek, których wysokości nie było w ciągu wejściowym.

Podobne rozumowanie możemy zastosować do przedziału przed pierwszym pewniakiem oraz za ostatnim pewniakiem. Dla pierwszego pewniaka parzystego mamy dokładnie jedno przyporządkowanie czapek na lewo od niego. Natomiast dla ostatniego pewniaka parzystego mamy dwie opcje dla czapek na prawo od niego w zależności od parzystości liczby krasnali – dla nieparzystego n mamy jedno przyporządkowanie, a dla parzystego n mamy możliwość wyboru krasnala, który dostanie inną czapkę:



Możemy uniknąć pisania osobnego kodu dla tych przypadków, jeśli rozważymy tylko przedziały pomiędzy pewniakami oraz dla n parzystego dodamy „wirtualnego" pewniaka na pozycji $n + 2$.

Jest jeszcze jedna możliwość do rozważenia, gdy nie ma pewniaków parzystych (ale są jacyś nieparzyści). Wtedy dla n parzystego mamy dokładnie jedno rozwiązanie, a dla n nieparzystego mamy sprzeczność.

Podsumowanie. Algorytm na początku sprawdza oczywiste warunki, w których nie istnieje rozwiązanie i wyznacza listę pewniaków parzystych i nieparzystych (dodając „wirtualnych"). Następnie przegląda wszystkie pary kolejnych pewniaków o tej samej parzystości i dla każdej z nich mnoży wynik przez liczbę czapek pomiędzy pewniakami. Jeśli w sumie rozważyliśmy k par, to na końcu mnożymy

odpowiedź przez $k!$, czyli liczbę przyporządkowań k nieprzyporządkowanych jeszcze czapek dla k krasnali wybranych w przedziałach.

Złożoność czasowa tego rozwiązania jest $O(n)$.

Montażysta

Autor zadania: Tomasz Idziaszek

Na początek zróbmy dwie proste obserwacje. Przede wszystkim, nie opłaca się robić przerw między montowanymi filmami, tzn. po zmontowaniu filmu można od razu montować kolejny. Tak jest, bo w dowolnym rozwiązaniu, w którym istnieje przerwa długości t dni, możemy ją usunąć i zacząć montować wszystkie filmy po niej o t dni wcześniej. Oczywiście nie popsuje to poprawności rozwiązania.

Po drugie, wybrane filmy opłaca się montować w kolejności ich terminów publikacji. Załóżmy dla uproszczenia naszych rozważań, że terminy te są nierosnące:

$$d_1 \leqslant d_2 \leqslant \ldots \leqslant d_n.$$

(W programie to założenie realizujemy przez wstępne posortowanie filmów po ich terminach publikacji, jednak z zapamiętaniem ich oryginalnych numerów, żeby można było je odtworzyć w odpowiedzi.) Zatem jeśli wybierzemy pewien podzbiór filmów, to można je montować w kolejności rosnących numerów. Tak jest, bo jeśli w optymalnym rozwiązaniu mamy jakąś inną kolejność, to znaczy, że istnieją w nim dwa filmy a i b, które montujemy po sobie, dla których $a > b$, więc $d_a > d_b$.

Zamieńmy teraz te filmy miejscami. Ta zamiana nie ma wpływu na inne filmy, bo nadal sumaryczny czas montażu tych dwóch wynosi $t_a + t_b$. Film b będzie zmontowany wcześniej (więc nic się tu nie popsuje), a film a zostanie zmontowany później, ale nadal zdąży na termin publikacji d_b (bo wcześniej film b zdążył) więc również na termin d_a.

Zamiana zmniejsza liczbę inwersji w ciągu numerów filmów i nie psuje poprawności rozwiązania. Wykonując zamiany tak długo, jak liczba inwersji jest dodatnia, w końcu dojdziemy do poprawnego rozwiązania, w którym zbiór filmów montujemy w kolejności rosnących numerów.

Tak więc rozwiązanie jest jednoznacznie wyznaczone przez wybór podzbioru filmów do montowania. Dla danego podzbioru łatwo jest w czasie $O(n)$ sprawdzić, czy prowadzi on do poprawnego rozwiązania, przeglądając filmy po kolei i sprawdzając, czy nie przekraczamy żadnego terminu.

To prowadzi do rozwiązania, w którym sprawdzamy wszystkie możliwe podzbiory. Ma ono złożoność czasową $O(2^n n)$ i zalicza pierwsze podzadanie.

Programowanie dynamiczne. Spróbujmy wybrać największy podzbiór filmów, stosując programowanie dynamiczne. Będziemy dodawać filmy po kolei; numer aktualnie rozważanego filmu będzie pierwszym wymiarem naszej tabelki. Co musimy wiedzieć o prawidłowym rozwiązaniu ograniczonym do filmów ze zbioru $\{1, \ldots, i\}$, żeby móc je rozszerzyć o kolejny film? Potrzebujemy wiedzieć, ile filmów zostało wybranych w tym rozwiązaniu oraz jaki jest sumaryczny czas ich montażu.

Ponieważ liczba wybranych filmów jest ograniczona przez i, to zrobimy z niej drugi wymiar tabelki. Niech zatem $dp[i, k]$ oznacza minimalny czas na prawidłowe zmontowanie dokładnie k filmów ze zbioru $\{1, \ldots, i\}$. Rekurencja jest następująca:

$$dp[i,k] = \begin{cases} \min(dp[i-1,k],\ dp[i-1,k-1]+t_i) & \text{jeśli } dp[i-1,k-1]+t_i \leqslant d_i, \\ dp[i-1,k] & \text{w przeciwnym przypadku.} \end{cases}$$

Jej poprawność wynika z faktu, że możemy albo nie montować i-tego filmu (i wtedy k filmów musi być wziętych ze zbioru $\{1, \ldots, i-1\}$, więc minimalny czas ich montażu to $dp[i-1, k]$), albo zmontować i-ty film, ale tylko wtedy, gdy możemy odpowiednio wcześnie skończyć montowanie $k-1$ filmów ze zbioru $\{1, \ldots, i-1\}$.

Warunkami początkowymi rekurencji są $dp[0, 0] = 0$ oraz $dp[0, k] = \infty$ dla $k > 0$. Natomiast odpowiedzią będzie maksymalne k, dla którego $dp[n, k] \neq \infty$. Czas działania tego algorytmu to $O(n^2)$, co daje zaliczenie drugiego podzadania.

Algorytm zachłanny. Czasem analiza rozwiązania opartego na programowaniu dynamicznym może nas prowadzić do odkrycia algorytmu zachłannego. Zauważmy, że jeśli dla zbioru $\{1, \ldots, i-1\}$ maksymalna liczba zmontowanych filmów to k, to dla zbioru $\{1, \ldots, i\}$ ta liczba może być równa albo k, albo $k+1$, bo możemy co najwyżej rozszerzyć rozwiązanie o film i-ty. Rozszerzenie możemy zrobić wtedy, gdy $dp[i-1, k] + t_i \leqslant d_i$ i sumaryczny czas będzie wynosić $dp[i-1, k] + t_i$.

Jeśli nie możemy rozszerzyć rozwiązania, to wtedy odpowiedź pozostaje równa k, ale nadal musimy wykonać trochę dodatkowej pracy, by obliczyć minimalny czas montażu. Nie musi być on bowiem równy $dp[i-1, k]$. W ogólności musimy popatrzeć na optymalny czas dla $k-1$ filmów $dp[i-1, k-1]$ i jeśli $dp[i-1, k-1]+t_i \leqslant d_i$, to możemy wziąć $\min(dp[i-1, k],\ dp[i-1, k-1]+t_i)$. Jak jednak obliczyć $dp[i-1, k-1]$ bez wykonywania całego programowania dynamicznego?

Niech $S \subseteq \{1, \ldots, i-1\}$ będzie optymalnym zbiorem k filmów o czasie montażu $dp[i-1, k]$. Może się okazać, że istnieje pewien film $s \in S$, który ma dłuższy czas montażu niż film i-ty, czyli: $t_s > t_i$. Wtedy, zamieniając film s-ty na i-ty, dostajemy poprawne rozwiązanie $S \setminus \{s\} \cup \{i\}$ z k filmami o krótszym czasie montażu $dp[i-1, k] - t_s + t_i$. Obiecującym pomysłem może być więc wymiana filmu s o maksymalnym czasie montażu spośród wszystkich filmów ze zbioru S.

Pytanie, czy taka ewentualna wymiana będzie optymalna? Innymi słowy, czy jest spełnione $dp[i-1, k-1] = dp[i-1, k] - t_s$. Załóżmy nie wprost, że nie jest, czyli $dp[i-1, k-1] + t_s < dp[i-1, k]$. Niech r będzie ostatnim filmem wziętym w rozwiązaniu S. Ponieważ s miał najdłuższy czas montażu, więc $t_r \leqslant t_s$. Jeśli dodamy r jako k-ty film do optymalnego zbioru $S' \subseteq \{1, \ldots, i-1\}$ zawierającego $k-1$ filmów, a możemy to zrobić, bo

$$d_r \geqslant dp[i-1, k] > dp[i-1, k-1] + t_s \geqslant dp[i-1, k-1] + t_r,$$

to dostaniemy rozwiązanie dla k filmów o czasie

$$dp[i-1, k-1] + t_r \leqslant dp[i-1, k-1] + t_s < dp[i-1, k],$$

czyli lepszym niż optymalny, co da nam sprzeczność (i tym samym kończy dowód).

Zatem możemy zaproponować następujący algorytm zachłanny: będziemy dodawać filmy po kolei, cały czas utrzymując częściowe rozwiązanie (sumę czasów oraz zbiór użytych filmów) dla maksymalnego k. Jeśli możemy je rozszerzyć do $k+1$, to to robimy, w przeciwnym wypadku dodajemy film i-ty do zbioru i wyrzucamy z niego film o najdłuższym czasie montażu.

Zbiór możemy reprezentować na kolejce priorytetowej, wtedy operacje dorzucenia i usunięcia będą działały w czasie $O(\log n)$. Uwzględniając początkowe sortowanie filmów, dostaniemy algorytm o złożoności $O(n \log n)$, zaliczający wszystkie podzadania.

Układanie kart

Autor zadania: Wojciech Rytter

W zadaniu tym wejściem jest jedna liczba n (pomijamy w rozważaniach liczbę m, która służy jedynie do ustalenia jak wypisujemy odpowiedź), więc nic nie stoi na przeszkodzie, żeby ręcznie rozpisać sobie pełne rozwiązania dla małych wartości n. Niestety, liczba permutacji, którą musimy przeanalizować (wyrażająca się wzorem $n!$) rośnie wykładniczo, więc zadanie to już dla $n = 4$ staje się dość nużące. Możemy jednak zaprząc do niego komputer...

Wizualizacja małych przykładów. Zaczniemy od rozwiązania inżynierskiego – napiszemy program, który dla danego n przeiteruje się po wszystkich $n! - 1$ permutacjach niekońcowych (czyli z pominięciem „posortowanej" permutacji identycznościowej $123 \ldots n$). Dla każdej permutacji p wykona operację Bajtazara: znajdzie kartę do przełożenia i wyznaczy permutację q, która wychodzi po przełożeniu karty. Zapamięta również koszt tej operacji.

Łatwiej będzie nam analizować taki przykład, jeśli przedstawimy go w postaci grafu. Będzie on miał $n!$ wierzchołków odpowiadających permutacjom, a z każdego wierzchołka (oprócz permutacji identycznościowej) będzie wychodzić dokładnie jedna ważona krawędź (z wierzchołka p do wierzchołka q o wadze będącej kosztem operacji). Ten graf będzie więc tak naprawdę drzewem ukorzenionym w wierzchołku permutacji identycznościowej, w którym każda krawędź jest skierowana od dziecka do ojca.

Odpowiedzią do zadania będzie suma wag dla ścieżek z każdego wierzchołka do korzenia.

Do zwizualizowania grafu wykorzystamy program `dot` dostępny w dystrybucjach Linuksa. Będzie on potrzebował opisu grafu w następującym formacie (przykład dla $n = 3$):

```
digraph G {
  132 -> 312 [label="1"];
  213 -> 123 [label="1"];
  231 -> 123 [label="2"];
  312 -> 231 [label="2"];
  321 -> 231 [label="1"];
}
```

Żeby stworzyć taki opis, napiszemy krótki program `graf.py` w języku Python, który wczytuje z wejścia liczbę n i wypisuje na wyjście opis grafu:

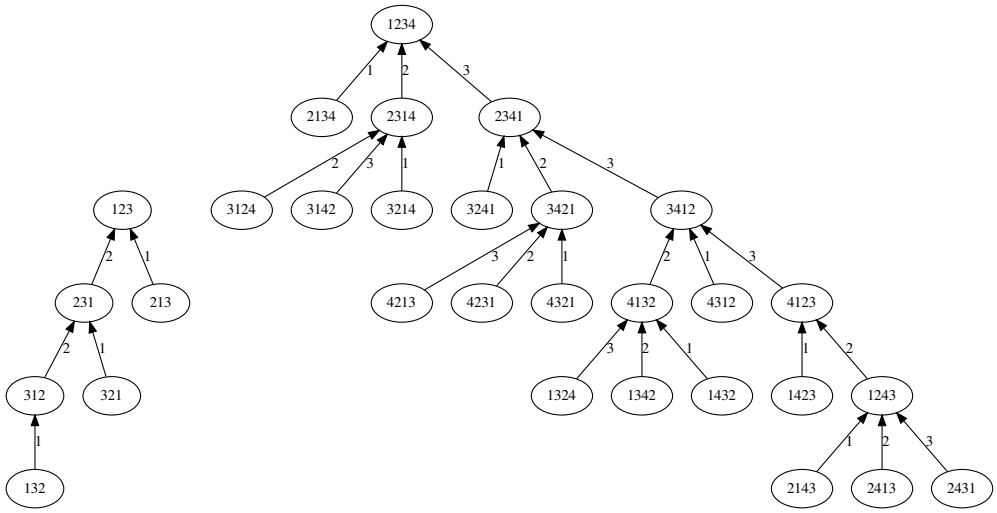
```
from itertools import permutations

n = int(input())

print('digraph G {')
for i, p_ in enumerate(permutations(range(n))):
    if i == 0: continue
```

```
        p = list(p_)
        card = (p[0] + n-1) % n
        pos = p.index(card)
        q = [card] + p[:pos] + p[pos+1:]
        ps = ''.join([str(x+1) for x in p])
        qs = ''.join([str(x+1) for x in q])
        print('  %s -> %s [label="%d"];' % (ps, qs, pos))
    print('}')
```

Całość uruchamiamy następującą komendą, która generuje plik `graf.pdf` z obrazkiem grafu:

```
echo 3 | python graf.py | dot -Tpdf -Grankdir=BT > graf.pdf
```

Liczba po komendzie `echo` to oczywiście interesująca nas wartość n. Na poniższym rysunku przedstawiono efekty tej komendy dla $n = 3$ oraz $n = 4$:



Analiza drzewa. Dzięki takiej wizualizacji możemy dostrzec pewne regularności w drzewie, tym bardziej widoczne dla $n = 5$ lub $n = 6$. Składa się ono z „głównej" ścieżki długości n, która $n-1$ krawędziami o wadze $n-1$ łączy kolejne rotacje cykliczne permutacji identycznościowej (w każdej takiej permutacji operacja Bajtazara przekłada ostatnią kartę). Do każdego z wierzchołków ścieżki dochodzą też krawędzie o wagach $1, 2, \ldots, n-2$, które dołączają do niej pewne poddrzewa. Krawędzią o wadze 1 jest dołączane zawsze poddrzewo jednowierzchołkowe. Krawędzią o wadze 2 jest dołączane drzewo o dwóch poziomach, jego korzeń ma krawędzie o wszystkich wagach $1, 2, \ldots, n-1$.

Dalsza analiza drzew pozwala nam na wysnucie przypuszczenia, że wszystkie drzewa dołączane krawędziami o wadze i (dla $1 \leqslant i \leqslant n-2$) mają co prawda różne permutacje w wierzchołkach, ale kształtem i wagami są izomorficzne; oznaczmy je

przez T_i (w szczególności T_1 jest drzewem jednowierzchołkowym). Drzewo T_i ma i poziomów, a każdy wierzchołek jest albo liściem, albo ma $n-1$ krawędzi do synów.

Jeszcze dalsza analiza utwierdza nas w przekonaniu, że drzewa T_i powstają rekurencyjnie. Jak mówiliśmy, drzewo T_1 jest pojedynczym wierzchołkiem. Natomiast drzewo T_k dla $k \geqslant 2$ jest wierzchołkiem do którego dochodzą krawędzie o wagach $w = 1, 2, \ldots, n-1$, każda podczepiająca drzewo $T_{\min(w,k-1)}$.

Można więc łatwo napisać rekurencję, która wyznaczy liczbę wierzchołków $W[k]$ w drzewie T_k (czyli korzeń plus liczba wierzchołków wszystkich poddrzew):

$$W[k] = 1 + \sum_{1 \leqslant w \leqslant n-1} W[\min(w, k-1)]$$

oraz sumę długości ścieżek $S[k]$ od wszystkich wierzchołków do korzenia w drzewie T_k (czyli koszty krawędzi prowadzących do korzenia przemnożone przez liczbę przechodzących przez nie ścieżek plus długości ścieżek w poddrzewach):

$$S[k] = \sum_{1 \leqslant w \leqslant n-1} w \cdot W[\min(w, k-1)] + S[\min(w, k-1)]$$

Dla drzewa jednowierzchołkowego mamy, oczywiście, $W[1] = 1$ oraz $S[1] = 0$.

Na koniec sumujemy długości ścieżek biegnących we wszystkich n kopiach drzew $T_1, T_2, \ldots, T_{n-2}$:

$$S' = n \cdot \sum_{1 \leqslant w \leqslant n-2} (w \cdot W[w] + S[w])$$

oraz biegnących po głównej ścieżce (waga krawędzi razy liczba przejść krawędzią z uwagi na głębokość na ścieżce głównej razy liczba przejść z uwagi na liczbę wierzchołków w podczepionych poddrzewach):

$$S'' = (n-1) \cdot (1 + 2 + \ldots + n - 1) \cdot \big(1 + \sum_{1 \leqslant w \leqslant n-2} W[w]\big).$$

Odpowiedzią do zadania jest $S' + S''$. Bezpośrednie zaimplementowanie powyższych wzorów da program o złożoności czasowej $O(n^2)$, który przejdzie drugie podzadanie.

Nie jest trudno tak przerobić te wzory z wykorzystaniem sum prefiksowych, by ich obliczanie miało złożoność czasową $O(n)$. Przykładowo, jeśli na bieżąco będziemy obliczali sumy prefiksowe $P[k] = W[1] + W[2] + \ldots + W[k]$ (pisząc po prostu, że $P[k] = P[k-1] + W[k]$), to wtedy

$$W[k] = 1 + P[k-1] + (n-k) \cdot W[k-1].$$

Zapisanie analogicznego wzoru dla tablicy S pozostawiamy jako ćwiczenie dla Czytelnika.

Dowód poprawności. Dla pełności rozwiązania należałoby formalnie udowodnić, że każde drzewo permutacji ma rzeczywiście taki kształt, jak opisaliśmy powyżej. Niech p będzie pewną permutacją kart. Kluczowa tutaj będzie długość k „arytmetycznego prefiksu" tej permutacji, w którym każda kolejna liczba jest większa

o 1 od poprzedniej (przy czym po n może następować 1). Długość tego prefiksu nazwiemy *rangą* permutacji. W szczególności rotacje cykliczne permutacji identycznościowej (czyli n permutacji z głównej ścieżki drzewa) to jedyne permutacje o randze n.

Zauważmy, że każda operacja na permutacji o randze mniejszej niż n zwiększa jej rangę. Zwiększenie jest o 1 lub więcej, bo „prefiks arytmetyczny" jest rozszerzany z lewej o przekładaną kartę i może być rozszerzony z prawej o liczby występujące w oryginalnej permutacji za przekładaną kartą:

Co łączy rangi permutacji z drzewem? Otóż w korzeniu drzewa T_k znajduje się zawsze permutacja o randze k.

Dowiedziemy tego, pokazując jak wyglądają dzieci danej permutacji p, czyli z jakich permutacji q przy pomocy jednej operacji możemy uzyskać permutację p. Jeśli p ma rangę 1, to nie ma żadnych dzieci, bo muszą one mieć rangę niższą.

Jeśli zaś p ma rangę $k > 1$ i zaczyna się liczbą x, to permutacja q wygląda tak, że przesuwamy liczbę x o pewną liczbę pozycji w prawo (od 1 do $n-1$). Jeśli przesunęliśmy ją o co najmniej $k-1$ pozycji, to permutacja q ma rangę $k-1$, jeśli zaś o $w < k-1$ pozycji, to liczba x wypadnie w środku prefiksu, zmniejszając rangę permutacji do w. Przesunięcie jest oczywiście kosztem operacji, więc widać, że permutacja o randze k jest połączona krawędziami $w = 1, \ldots, n-1$ z permutacjami o rangach $\min(w, k-1)$.

Przypadkiem szczególnym są permutacje cykliczne (czyli o randze n), w których przesunięcie o $n-1$ prowadzi z powrotem do permutacji cyklicznej.

Na koniec warto zauważyć, że do przeprowadzenia powyższego dowodu (i w konsekwencji rozwiązania zadania) nie było konieczne wykonanie wizualizacji małych przykładów. Zamiast tego można było popatrzeć na zachowanie operacji Bajtazara dla różnych permutacji i odkryć, że kluczową rolę w klasyfikacji permutacji odgrywa tu długość ich „arytmetycznych prefiksów". Jednak im więcej mamy intuicji związanych z zadaniem (a wizualizacje takich intuicji dostarczają), tym łatwiej nam takich odkryć dokonywać.