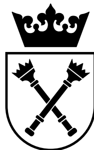
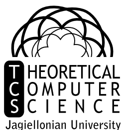
AMPPZ 2021: Prezentacja rozwiązań

Uniwersytet Jagielloński

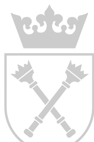
7 listopada 2021

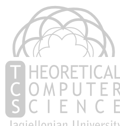




Statystyki

Zgłoszeń: 664
Zaakceptowanych: 218

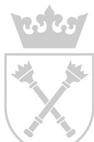


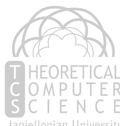


Statystyki

Zgłoszeń: 664
Zaakceptowanych: 218

Pierwsze wbite zadanie: (H) Hasła
UJ4 [Zięba, Michno, Mazur]



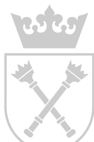


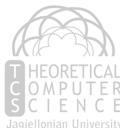
Zadanie H
Hasła

Zgłoszeń: 59
Zaakceptowanych: 44 (wszystkie drużyny)

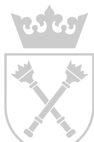
Pierwsze rozwiązanie:
UJ4 [Zięba, Michno, Mazur]
0:03

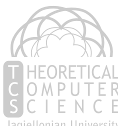
Autor: Krzysztof Maziarz





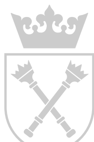
Mamy dwa ciągi znaków H_1 i H_2. Wiemy, że po zakodowaniu szyfrem Cezara o przesunięciu d, słowo H_1 przechodzi na H_2, a H_2 na H_1. Mając dane H_1, trzeba odtworzyć H_2.





Mamy dwa ciągi znaków H_1 i H_2. Wiemy, że po zakodowaniu szyfrem Cezara o przesunięciu d, słowo H_1 przechodzi na H_2, a H_2 na H_1. Mając dane H_1, trzeba odtworzyć H_2.

Po dwukrotnym zaszyfrowaniu, H_1 przechodzi w to samo słowo. Ponieważ H_1 i H_2 są różne, jest to możliwe tylko jeśli $d = 13\ldots$ co już wystarczy do rozwiązania.

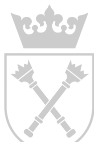


Zadanie D
Dwuczęściowy mechanizm

Zgłoszeń: 69
Zaakceptowanych: 34

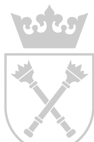
Pierwsze rozwiązanie:
UW1 [Jamka, Paluszek, Kondraciuk]
0:26

Autor: Daniel Goc

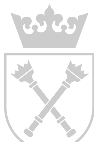


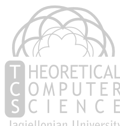
Zadanie

Dane są dwa izotetyczne kształty na płaszczyźnie oraz ciąg instrukcji mówiących, by jeden z nich przesuwać w górę, w dół, w lewo lub w prawo aż uderzy on w drugi kształt. Należy stwierdzić czy podczas przesuwania jeden z kształtów mogliśmy przesuwać dowolnie daleko.



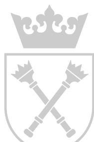
Obie figury mają na tyle mało komórek, że możemy brutalnie symulować cały proces.





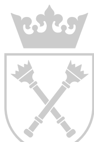
Obie figury mają na tyle mało komórek, że możemy brutalnie symulować cały proces.

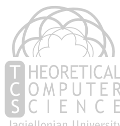
Przesuwamy kształt po jednym polu tak długo, jak możemy. Gdy dostatecznie się oddali stwierdzamy, że figury zostały rozłączone.



Obie figury mają na tyle mało komórek, że możemy brutalnie symulować cały proces.

Przesuwamy kształt po jednym polu tak długo, jak możemy. Gdy dostatecznie się oddali stwierdzamy, że figury zostały rozłączone. Komórki nieruchomej figury warto zostawić zaznaczone w tablicy, drugiej zaś trzymać np. na liście.



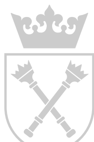


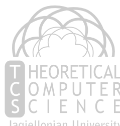
Zadanie J
Jadowite węże

Zgłoszeń: 77
Zaakceptowanych: 27

Pierwsze rozwiązanie:
UJ4 [Zięba, Michno, Mazur]
0:44

Autor: Krzysztof Maziarz



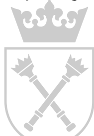


Zadanie

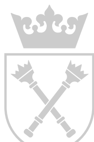
Dana jest plansza o wymiarach $n \times m$. Każde pole może być puste, zablokowane lub zawierać siedlisko jadowitych bądź niejadowitych węży.

Odwrócenie wiersza/kolumny sprawia, że każde siedlisko jadowitych węży zamienia się w siedlisko węży niejadowitych, i vice versa. Jeżeli jakieś siedlisko zostanie odwrócone dwukrotnie, powraca do swojego oryginalnego stanu.

Musimy odwrócić pewne wiersze i kolumny tak, aby być w stanie przejść z lewego górnego do prawego dolnego narożnika planszy, używając jedynie pól pustych oraz pól z niejadowitymi wężami. W każdym ruchu możemy przejść o jedno pole w prawo albo o jedno pole w dół.

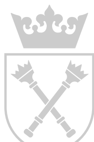


Jeżeli nie istnieje żadna ścieżka łącząca narożniki planszy, która nie używa pól zablokowanych, odpowiedzią jest oczywiście “NIE“. W przeciwnym przypadku, znajdźmy dowolną taką ścieżkę.



Jeżeli nie istnieje żadna ścieżka łącząca narożniki planszy, która nie używa pól zablokowanych, odpowiedzią jest oczywiście “NIE“. W przeciwnym przypadku, znajdźmy dowolną taką ścieżkę.

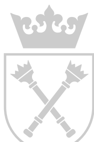
Kluczowa obserwacja: w każdym kroku, nasza ścieżka wchodzi bądź na wiersz bądź na kolumnę (w “kroku 0”, ma miejsce i jedno i drugie) nigdy wcześniej przez nas nie odwiedzone.



Jeżeli nie istnieje żadna ścieżka łącząca narożniki planszy, która nie używa pól zablokowanych, odpowiedzią jest oczywiście "NIE". W przeciwnym przypadku, znajdźmy dowolną taką ścieżkę.

Kluczowa obserwacja: w każdym kroku, nasza ścieżka wchodzi bądź na wiersz bądź na kolumnę (w "kroku 0", ma miejsce i jedno i drugie) nigdy wcześniej przez nas nie odwiedzone.

Jeżeli więc wchodzimy właśnie na pole z siedliskiem jadowitych węży, możemy bezpiecznie odwrócić albo obecny wiersz albo obecną kolumnę. Bieżące pole zamienia się wtedy w siedlisko węży niejadowitych, zaś wykonane odwrócenie nie może zepsuć żadnych wcześniejszych pól na naszej ścieżce.

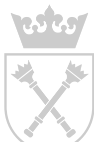


Zadanie K
Kot i Roomba

Zgłoszeń: 92
Zaakceptowanych: 26

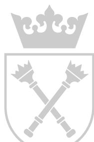
Pierwsze rozwiązanie:
UWr2 [Górkiewicz, Agrawal, Martowicz]
0:18

Autor: Krzysztof Maziarz



Zadanie

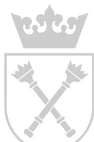
Mamy dane drzewo o n wierzchołkach, po którym jeździ Roomba odwiedzająca kolejno wierzchołki $a_1, \ldots, a_m$. W wierzchołku c początkowo śpi kot; gdy Roomba wjeżdża do wierzchołka z kotem, ten budzi się, i ucieka do losowego sąsiedniego wierzchołka. Wyznaczyć oczekiwaną liczbę obudzeń kota.



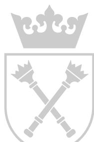
Zadanie

Mamy dane drzewo o n wierzchołkach, po którym jeździ Roomba odwiedzająca kolejno wierzchołki $a_1, \ldots, a_m$. W wierzchołku c początkowo śpi kot; gdy Roomba wjeżdża do wierzchołka z kotem, ten budzi się, i ucieka do losowego sąsiedniego wierzchołka. Wyznaczyć oczekiwaną liczbę obudzeń kota.

Limity: $n \leq 1\,000\,000$, $m \leq 5\,000\,000$.

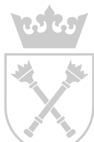


Dla każdego wierzchołka utrzymujemy prawdopodobieństwo, że obecnie jest w nim kot. Początkowo $p[c] = 1.0$.

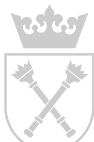


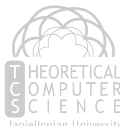
Dla każdego wierzchołka utrzymujemy prawdopodobieństwo, że obecnie jest w nim kot. Początkowo $p[c] = 1.0$.

Gdy Roombda wjeżdża do v do wyniku dodajemy $p[v]$, po czym do wszystkich d_v sąsiadów v dodajemy $\frac{p[v]}{d_v}$, na koniec ustawiając $p[v] = 0$.



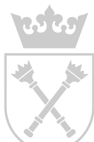
Naiwnie operacja dodania na wszystkich sąsiadach v zajmuje czas $\mathcal{O}(d_v)$, co jest zbyt wolne (sumaryczny czas może wynieść $\mathcal{O}(n \cdot m)$).

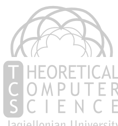




Naiwnie operacja dodania na wszystkich sąsiadach v zajmuje czas $\mathcal{O}(d_v)$, co jest zbyt wolne (sumaryczny czas może wynieść $\mathcal{O}(n \cdot m)$).

Optymalizacja: dla każdego wierzchołka oprócz $p[v]$ utrzymujemy $lazy[v]$, oznaczające wartość którą leniwie chcielibyśmy dodać do p dla wszystkich dzieci v. Nigdy jednak nie będziemy "spychać" tej wartości; żeby odczytać wartość w v, zawsze będziemy obliczać $p[v] + lazy[parent[v]]$.

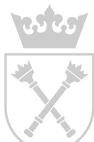




Naiwnie operacja dodania na wszystkich sąsiadach v zajmuje czas $\mathcal{O}(d_v)$, co jest zbyt wolne (sumaryczny czas może wynieść $\mathcal{O}(n \cdot m)$).

Optymalizacja: dla każdego wierzchołka oprócz $p[v]$ utrzymujemy $lazy[v]$, oznaczające wartość którą leniwie chcielibyśmy dodać do p dla wszystkich dzieci v. Nigdy jednak nie będziemy "spychać" tej wartości; żeby odczytać wartość w v, zawsze będziemy obliczać $p[v] + lazy[parent[v]]$.

Teraz operacja dodania na wszystkich sąsiadach v sprowadza się do zwiększenia $lazy[v]$ oraz $p[parent[v]]$.

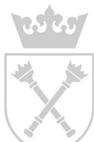


Naiwnie operacja dodania na wszystkich sąsiadach v zajmuje czas $\mathcal{O}(d_v)$, co jest zbyt wolne (sumaryczny czas może wynieść $\mathcal{O}(n \cdot m)$).

Optymalizacja: dla każdego wierzchołka oprócz $p[v]$ utrzymujemy $lazy[v]$, oznaczające wartość którą leniwie chcielibyśmy dodać do p dla wszystkich dzieci v. Nigdy jednak nie będziemy "spychać" tej wartości; żeby odczytać wartość w v, zawsze będziemy obliczać $p[v] + lazy[parent[v]]$.

Teraz operacja dodania na wszystkich sąsiadach v sprowadza się do zwiększenia $lazy[v]$ oraz $p[parent[v]]$.

Złożoność: $\mathcal{O}(n + m)$.

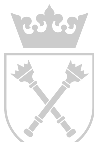


Zadanie C
Ciasto

Zgłoszeń: 73
Zaakceptowanych: 25

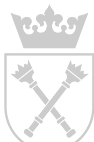
Pierwsze rozwiązanie:
PUT1 [Pawłowski, Piechowiak, Woźniak]
0:38

Autor: Krzysztof Maziarz



Zadanie

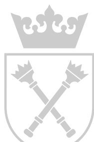
Mamy dane dwie tabelki $2 \times n$ wypełnione liczbami całkowitymi. Chcemy przekształcić jedną w drugą za pomocą operacji obracania podkwadratów 2×2 o 180 stopni. Wypisz minimalną liczbę ruchów (albo odpowiedz, że jest to niemożliwe).

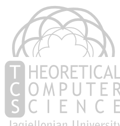


Zadanie

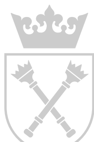
Mamy dane dwie tabelki $2 \times n$ wypełnione liczbami całkowitymi. Chcemy przekształcić jedną w drugą za pomocą operacji obracania podkwadratów 2×2 o 180 stopni. Wypisz minimalną liczbę ruchów (albo odpowiedz, że jest to niemożliwe).

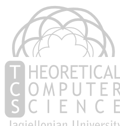
Limity: $n \leq 500\,000$.





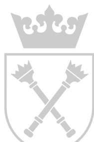
Obrót o 180 stopni odpowiada zamianie miejscami pól $t[0][i]$ z $t[1][i+1]$ oraz $t[1][i]$ z $t[0][i+1]$ dla pewnego i.

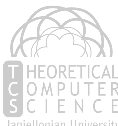




Obrót o 180 stopni odpowiada zamianie miejscami pól $t[0][i]$ z $t[1][i+1]$ oraz $t[1][i]$ z $t[0][i+1]$ dla pewnego i.

Trik: zamieńmy górną liczbę z dolną w *co drugiej kolumnie* (zarówno w początkowej jak i końcowej tabelce). Teraz naszym celem wciąż jest przekształcenie jednej tabelki w drugą, ale zamiast obrotów o 180 nasza operacja to *zamiana miejscami dwóch sąsiednich kolumn*.

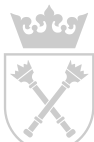




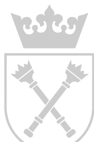
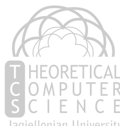
Obrót o 180 stopni odpowiada zamianie miejscami pól $t[0][i]$ z $t[1][i+1]$ oraz $t[1][i]$ z $t[0][i+1]$ dla pewnego i.

Trik: zamieńmy górną liczbę z dolną w *co drugiej kolumnie* (zarówno w początkowej jak i końcowej tabelce). Teraz naszym celem wciąż jest przekształcenie jednej tabelki w drugą, ale zamiast obrotów o 180 nasza operacja to *zamiana miejscami dwóch sąsiednich kolumn*.

Teraz mamy ciąg par $(t[0][i], t[1][i])$ który chcemy przekształcić na inny ciąg par.

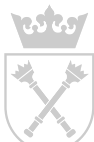


Pary w docelowym ciągu numerujemy w kolejności jako $1, 2, ..., n$ (używamy wszystkich identyfikatorów nawet jeśli niektóre z par są takie same).

Pary w docelowym ciągu numerujemy w kolejności jako $1, 2, \ldots, n$ (używamy wszystkich identyfikatorów nawet jeśli niektóre z par są takie same).

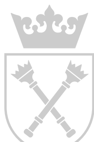
Jeśli pary w początkowym ciągu ponumerujemy odpowiednio, to odpowiedzią jest *liczba inwersji* otrzymanego ciągu. (W celu formalnego udowodnienia tej zależności, należy wyjść od spostrzeżenia, że wykonanie dowolnej zamiany sąsiednich elementów zmienia sumaryczną liczbę inwersji w ciągu o dokładnie 1.)



Pary w docelowym ciągu numerujemy w kolejności jako $1, 2, \ldots, n$ (używamy wszystkich identyfikatorów nawet jeśli niektóre z par są takie same).

Jeśli pary w początkowym ciągu ponumerujemy odpowiednio, to odpowiedzią jest *liczba inwersji* otrzymanego ciągu. (W celu formalnego udowodnienia tej zależności, należy wyjść od spostrzeżenia, że wykonanie dowolnej zamiany sąsiednich elementów zmienia sumaryczną liczbę inwersji w ciągu o dokładnie 1.)

Inwersje możemy policzyć w czasie $\mathcal{O}(n \log n)$ na wiele sposobów, na przykład używając drzewa przedziałowego albo algorytmu *merge sort*.

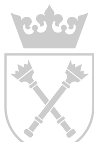


Zadanie L
Leniwce

Zgłoszeń: 69
Zaakceptowanych: 24

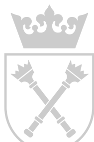
Pierwsze rozwiązanie:
UJ4 [Zięba, Michno, Mazur]
0:39

Autor: Daniel Goc

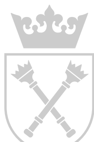


Zadanie

Na planszy $n \times m$ znajdują się legowiska leniwców. Każde legowisko ma obszar żerowania – wszystkie pola w odległości k w metryce miejskiej od niego. Mając dane pola, na których leniwce żerują, rozstrzygnąć czy istnieje dla nich prawidłowy układ legowisk.

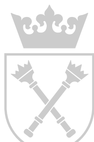


Zauważmy, że jeśli leniwce gdzieś **nie** żerują, to żadne legowisko nie może znajdować się w odległości mniejszej lub równej niż k od tego pola. Wykreślmy zatem wszystkie takie pola z planszy – możemy to zrobić na przykład za pomocą algorytmu BFS w czasie liniowym.



Czy da się na pozostałych (niewykreślonych) polach tak rozmieścić legowiska, żeby dosięgły do wszystkich pól żerowania? Niezależnie jak byśmy je umieścili, zasięg żerowania nie będzie „za duży”, może jedynie nie dosięgać wszystkich potrzebnych pól.

Możemy zatem bez straty rozmieścić legowiska na **wszystkich** niewykreślonych polach – takie legowisko nigdy nie będzie sprzeczne z naszymi danymi. Umieszczamy zatem legowiska wszędzie gdzie to możliwe i sprawdzamy ich łączny zasięg – znów możemy to zrobić algorytmem BFS w czasie liniowym.

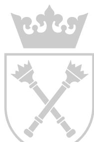


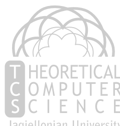
Zadanie F
Farba

Zgłoszeń: 66
Zaakceptowanych: 18

Pierwsze rozwiązanie:
UJ2 [Podkanowicz, Pióro, Potępa]
0:54

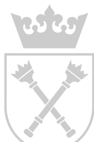
Autor: Krzysztof Maziarz



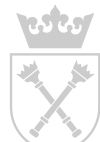


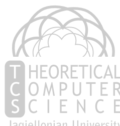
Zadanie

Dany jest ciąg $a_1, a_2, \ldots, a_n$. Dla pewnego b możemy wykonać następujący proces: rozłożyć każde a_i na $\lfloor \frac{a_i - 1}{b} \rfloor$ powtórzeń liczby b oraz liczbę $((a_i - 1) \mod b) + 1$. Np. dla $a = [4, 5, 6]$ i $b = 2$ otrzymamy ciąg $[2, 2, 2, 2, 1, 2, 2, 2]$.
Dla każdego (sensownego) b należy obliczyć sumę elementów znajdujących się na nieparzystych pozycjach w otrzymanym ciągu.



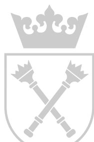
Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

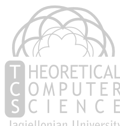




Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

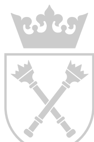
$[5, 4, 3, 2, 1]$

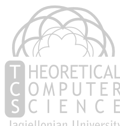




Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

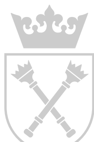
$[5, 4, 3, 2, 1]$
Liczba elementów $\rightarrow$ 5





Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

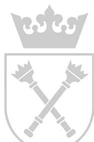
$[5, 4, 3, 2, 1]$
Liczba elementów $\rightarrow$ 5
Suma na nieparzystych pozycjach $\rightarrow$ 9



THEORETICAL COMPUTER SCIENCE Jagiellonian University

Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

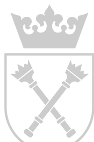
$[5, 4, 3, 2, 1]$
Liczba elementów $\rightarrow$ 5
Suma na nieparzystych pozycjach $\rightarrow$ 9
Suma na parzystych pozycjach $\rightarrow$ 6



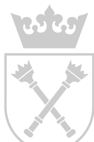
Spróbujmy z długiego ciągu wyciągnąć najważniejsze informacje.

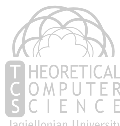
$[5, 4, 3, 2, 1]$
Liczba elementów $\rightarrow$ 5
Suma na nieparzystych pozycjach $\rightarrow$ 9
Suma na parzystych pozycjach $\rightarrow$ 6

Potrzebne nam tylko tyle!

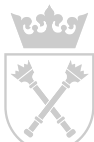


Znając b jesteśmy w stanie uzyskać informacje na temat rozłożonego z a_i ciągu w $O(1)$.

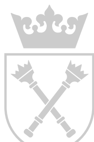


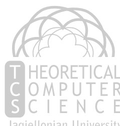


Znając b jesteśmy w stanie uzyskać informacje na temat rozłożonego z a_i ciągu w $O(1)$.
Zbudujmy nad ciągiem drzewo przedziałowe.

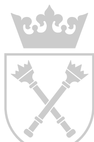


Znając b jesteśmy w stanie uzyskać informacje na temat rozłożonego z a_i ciągu w $O(1)$.
Zbudujmy nad ciągiem drzewo przedziałowe.
Możemy dla każdego przedziału bazowego trzymać informacje na jego temat.

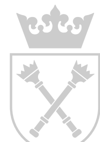
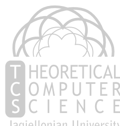




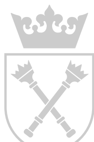
Znając b jesteśmy w stanie uzyskać informacje na temat rozłożonego z a_i ciągu w $O(1)$.
Zbudujmy nad ciągiem drzewo przedziałowe.
Możemy dla każdego przedziału bazowego trzymać informacje na jego temat.
Iterując się po b od 1 w górę możemy aktualizować drzewo dla wartości $a_i \geq b$. Drzewo zaktualizujemy $\sum_{i=1}^{n} a_i$ razy, co daje nam złożoność $O((\sum_{i=1}^{n} a_i) \cdot \log(n))$.

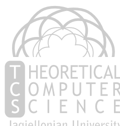


Da się szybciej!

Da się szybciej!
Trzymajmy ciąg skompresowanych informacji o przedziałach w które urosły liczby.

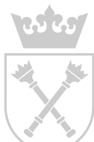




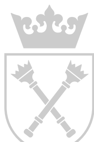
Da się szybciej!

Trzymajmy ciąg skompresowanych informacji o przedziałach w które urosły liczby.

Jeśli dwie sąsiednie liczby już nie będą się zmieniać, to możemy połączyć ich przedziały.



Da się szybciej!
Trzymajmy ciąg skompresowanych informacji o przedziałach w które urosły liczby.
Jeśli dwie sąsiednie liczby już nie będą się zmieniać, to możemy połączyć ich przedziały.
Elementów w ciągu gdy rozważamy b jest co najwyżej dwa razy więcej niż liczb a_i większych lub równych b.



THEORETICAL COMPUTER SCIENCE
Jagiellonian University

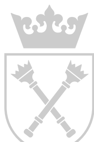
Da się szybciej!

Trzymajmy ciąg skompresowanych informacji o przedziałach w które urosły liczby.

Jeśli dwie sąsiednie liczby już nie będą się zmieniać, to możemy połączyć ich przedziały.

Elementów w ciągu gdy rozważamy b jest co najwyżej dwa razy więcej niż liczb a_i większych lub równych b.

Kończymy ze złożonością $O(\sum_{i=1}^{n} a_i)$.

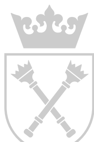


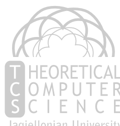
Zadanie I
Interesujące liczby

Zgłoszeń: 53
Zaakceptowanych: 6

Pierwsze rozwiązanie:
UW1 [Jamka, Paluszek, Kondraciuk]
1:05

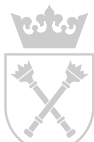
Autor: Praca zbiorowa





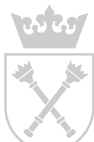
Zadanie

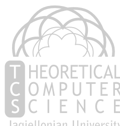
Dany jest ciąg liczb całkowitych a_i oraz liczba k. Tworzymy graf w którym wierzchołki i oraz j są połączone krawędzią wtedy i tylko wtedy gdy $a_i \oplus a_j \leq k$. W zadanym grafie należy policzyć rozmiar największej kliki.



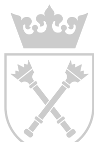
THEORETICAL COMPUTER SCIENCE
Jagiellonian University

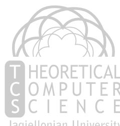
Podejdźmy do problemu rekurencyjnie.



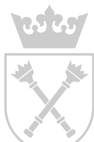


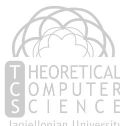
Podejdźmy do problemu rekurencyjnie.
Spróbujmy rozwiązać problem dla przedziału liczb $[0, 2^g)$, ustalonej wartości k oraz multizbioru wartości a_i w tym przedziale.





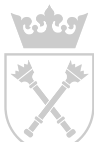
Jeśli $g = 0$ to problem jest trywialny.

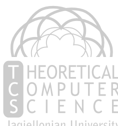




Jeśli $g = 0$ to problem jest trywialny.

Jeśli $k < 2^{g-1}$, to jeśli dla dwóch wartości a_i oraz a_j dokładnie jedna z nich ma zapalony $(g-1)$-szy bit, to na pewno nie są połączone krawędzią i możemy niezależnie rozwiązać przedziały $[0, 2^{g-1})$ i $[2^{g-1}, 2^g)$.

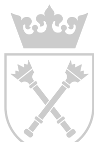




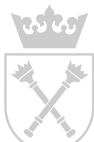
Jeśli $g = 0$ to problem jest trywialny.

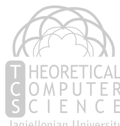
Jeśli $k < 2^{g-1}$, to jeśli dla dwóch wartości a_i oraz a_j dokładnie jedna z nich ma zapalony $(g-1)$-szy bit, to na pewno nie są połączone krawędzią i możemy niezależnie rozwiązać przedziały $[0, 2^{g-1})$ i $[2^{g-1}, 2^g)$.

Jeśli $k \geq 2^{g-1}$ i dla dwóch liczb a_i oraz a_j albo obie mają zapalony $(g-1)$-szy bit, albo obie nie mają, to na pewno obie są połączone krawędzią, krawędzi może zatem nie być jedynie między liczbami a_i mniejszymi bądź równymi 2^{g-1} oraz większymi od 2^{g-1}.



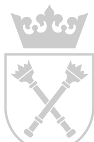
Mamy zatem dwie kliki między którymi występują niektóre krawędzie.
Możemy do takiego zadania podejść na kilka sposobów.

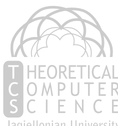




Mamy zatem dwie kliki między którymi występują niektóre krawędzie. Możemy do takiego zadania podejść na kilka sposobów.

Możemy dalej rozwiązywać problem rekurencyjnie, co pozwala nam uzyskać OK z sumarycznym czasem działania $\mathcal{O}(n \log(max_i a_i))$.

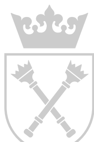


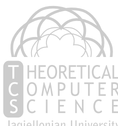


Mamy zatem dwie kliki między którymi występują niektóre krawędzie. Możemy do takiego zadania podejść na kilka sposobów.

Możemy dalej rozwiązywać problem rekurencyjnie, co pozwala nam uzyskać OK z sumarycznym czasem działania $\mathcal{O}(n \log(max_i a_i))$.

Alternatywnie, możemy zauważyć, że nasz problem sprowadził się do szukania największego zbioru niezależnego w grafie dwudzielnym!



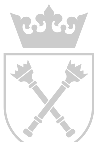


Mamy zatem dwie kliki między którymi występują niektóre krawędzie. Możemy do takiego zadania podejść na kilka sposobów.

Możemy dalej rozwiązywać problem rekurencyjnie, co pozwala nam uzyskać OK z sumarycznym czasem działania $\mathcal{O}(n \log(max_i a_i))$.

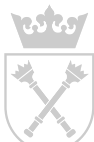
Alternatywnie, możemy zauważyć, że nasz problem sprowadził się do szukania największego zbioru niezależnego w grafie dwudzielnym!

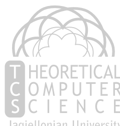
Tutaj z pomocą biegnie twierdzenie Kőniga, które mówi, że rozmiar maksymalnego zbioru niezależnego jest równy liczbie wierzchołków pomniejszonej o rozmiar maksymalnego skojarzenia.



THEORETICAL COMPUTER SCIENCE Jagiellonian University

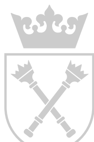
Szukając skojarzenia możemy dostosować Turbo Matching aby działał w zadanym, specyficznym grafie...

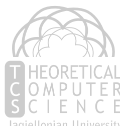




Szukając skojarzenia możemy dostosować Turbo Matching aby działał w zadanym, specyficznym grafie...

...lub zbudować nad ciągiem drzewo przedziałowe którego użyjemy do kompresji grafu. Następnie przyda się dowolny sensowny algorytm przepływowy, np. algorytm Dinica.



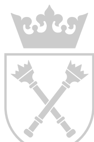


Zadanie E
Epidemia

Zgłoszeń: 34
Zaakceptowanych: 5

Pierwsze rozwiązanie:
UW3 [Kądziołka, Kluk, Łyżwa]
1:41

Autor: Krzysztof Kleiner



Zadanie

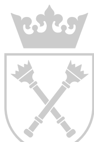
Danych jest n osób, z których każda może być początkowo zdrowa lub zarażona bajtobakterią. Następuje ciąg k zdarzeń następującej postaci:

1. Grupa osób spotyka się – jeżeli którakolwiek z nich była zarażona, to wszystkie stają się zarażone (i pozostają zarażone do końca życia).
2. Pewna osoba otrzymuje negatywny wynik testu na obecność bajtobakterii.
3. Pewna osoba otrzymuje pozytywny wynik testu na obecność bajtobakterii i zostaje skierowana na bezterminową kwarantannę.
4. Otrzymujesz zapytanie, czy da się udowodnić, że nikt oprócz osób przebywających na kwarantannie nie może być zarażony[1].

Odpowiedzi na wszystkie zapytania muszą być udzielane *online*.
Limity: $n \leq 500\,000$, $k \leq 1\,000\,000$.

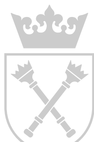
[1] Jeśli nie, należy podać przykład takiej osoby.

Aktualny stan naszej wiedzy będziemy reprezentować jako graf acykliczny skierowany (DAG). Początkowo, składa się on z n izolowanych wierzchołków, w i-tym z nich umieszczamy "pionek" i-tej osoby.



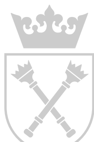
Aktualny stan naszej wiedzy będziemy reprezentować jako graf acykliczny skierowany (DAG). Początkowo, składa się on z n izolowanych wierzchołków, w i-tym z nich umieszczamy "pionek" i-tej osoby.

- Trzymamy również zbiór *possibly_infected* wszystkich osób, które mogą potencjalnie być zarażone, a nie przebywają na kwarantannie. Początkowo są to wszystkie osoby.



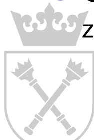
Aktualny stan naszej wiedzy będziemy reprezentować jako graf acykliczny skierowany (DAG). Początkowo, składa się on z n izolowanych wierzchołków, w i-tym z nich umieszczamy "pionek" i-tej osoby.

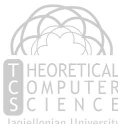
- Trzymamy również zbiór *possibly_infected* wszystkich osób, które mogą potencjalnie być zarażone, a nie przebywają na kwarantannie. Początkowo są to wszystkie osoby.
- Gdy pewna grupa spotyka się, tworzymy nowy "wierzchołek spotkania", do którego prowadzimy krawędzie z wierzchołków w których obecnie znajdują się pionki uczestniczących w spotkaniu osób. Następnie przesuwamy wszystkie te pionki do nowo utworzonego wierzchołka. Jeżeli którakolwiek osoba znajduje się w zbiorze *possibly_infected*, to dokładamy do tego zbioru wszystkie spotykające się osoby.



Aktualny stan naszej wiedzy będziemy reprezentować jako graf acykliczny skierowany (DAG). Początkowo, składa się on z n izolowanych wierzchołków, w i-tym z nich umieszczamy “pionek” i-tej osoby.

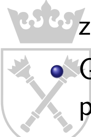
- Trzymamy również zbiór *possibly_infected* wszystkich osób, które mogą potencjalnie być zarażone, a nie przebywają na kwarantannie. Początkowo są to wszystkie osoby.
- Gdy pewna grupa spotyka się, tworzymy nowy “wierzchołek spotkania”, do którego prowadzimy krawędzie z wierzchołków w których obecnie znajdują się pionki uczestniczących w spotkaniu osób. Następnie przesuwamy wszystkie te pionki do nowo utworzonego wierzchołka. Jeżeli którakolwiek osoba znajduje się w zbiorze *possibly_infected*, to dokładamy do tego zbioru wszystkie spotykające się osoby.
- Gdy pewna osoba otrzymuje pozytywny wynik testu, usuwamy ją ze zbioru *possibly_infected* oraz zabieramy z grafu jej pionek.



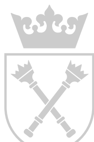


Aktualny stan naszej wiedzy będziemy reprezentować jako graf acykliczny skierowany (DAG). Początkowo, składa się on z n izolowanych wierzchołków, w i-tym z nich umieszczamy "pionek" i-tej osoby.

- Trzymamy również zbiór *possibly_infected* wszystkich osób, które mogą potencjalnie być zarażone, a nie przebywają na kwarantannie. Początkowo są to wszystkie osoby.
- Gdy pewna grupa spotyka się, tworzymy nowy "wierzchołek spotkania", do którego prowadzimy krawędzie z wierzchołków w których obecnie znajdują się pionki uczestniczących w spotkaniu osób. Następnie przesuwamy wszystkie te pionki do nowo utworzonego wierzchołka. Jeżeli którakolwiek osoba znajduje się w zbiorze *possibly_infected*, to dokładamy do tego zbioru wszystkie spotykające się osoby.
- Gdy pewna osoba otrzymuje pozytywny wynik testu, usuwamy ją ze zbioru *possibly_infected* oraz zabieramy z grafu jej pionek.
- Gdy pewna osoba otrzymuje negatywny wynik testu, będziemy musieli przeliczyć stan naszej wiedzy.



Kiedy osoba p otrzymuje negatywny wynik testu, przeglądamy graf algorytmem DFS, rozpoczynając od tego wierzchołka (v) w którym obecnie znajduje się pionek osoby p.



Krok algorytmu **DFS** (dla wierzchołka v) wygląda następująco:

- Wiemy, że osoba uczestnicząca w spotkaniu v nie zaraziła się. Możemy mieć więc pewność, że wszystkie osoby uczestniczące w tym spotkaniu były (w tamtym momencie) zdrowe. Te spośród nich, których pionki nadal znajdują się w wierzchołku v, możemy więc **oznaczyć jako zdrowe** [2], tj. usunąć je ze zbioru *possibly_infected*.

[2] Natomiast osoby, które brały udział w spotkaniu v, lecz od tego czasu ich pionki zostały już przesunięte do dalszych wierzchołków, będą mogły zostać przeanalizowane dopiero kiedy algorytm dotrze do obecnych lokalizacji ich pionków.

Krok algorytmu **DFS** (dla wierzchołka v) wygląda następująco:

- Wiemy, że osoba uczestnicząca w spotkaniu v nie zaraziła się. Możemy mieć więc pewność, że wszystkie osoby uczestniczące w tym spotkaniu były (w tamtym momencie) zdrowe. Te spośród nich, których pionki nadal znajdują się w wierzchołku v, możemy więc **oznaczyć jako zdrowe** [2], tj. usunąć je ze zbioru *possibly_infected*.
- **Usuwamy** z grafu wierzchołek v (musiał on być “zdrowy”).

[2] Natomiast osoby, które brały udział w spotkaniu v, lecz od tego czasu ich pionki zostały już przesunięte do dalszych wierzchołków, będą mogły zostać przeanalizowane dopiero kiedy algorytm dotrze do obecnych lokalizacji ich pionków.

Krok algorytmu **DFS** (dla wierzchołka v) wygląda następująco:

- Wiemy, że osoba uczestnicząca w spotkaniu v nie zaraziła się. Możemy mieć więc pewność, że wszystkie osoby uczestniczące w tym spotkaniu były (w tamtym momencie) zdrowe. Te spośród nich, których pionki nadal znajdują się w wierzchołku v, możemy więc **oznaczyć jako zdrowe** [2], tj. usunąć je ze zbioru *possibly_infected*.
- **Usuwamy** z grafu wierzchołek v (musiał on być "zdrowy").
- Dla każdej krawędzi $u \to v$, wywołujemy rekurencyjnie **DFS(**u**)**. Skoro bowiem wszystkie osoby obecne na spotkaniu v były (w tamtym momencie) zdrowe, to musiały być zdrowe również podczas wszystkich swoich wcześniejszych spotkań.

[2] Natomiast osoby, które brały udział w spotkaniu v, lecz od tego czasu ich pionki zostały już przesunięte do dalszych wierzchołków, będą mogły zostać przeanalizowane dopiero kiedy algorytm dotrze do obecnych lokalizacji ich pionków.

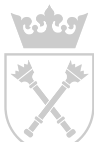
Krok algorytmu **DFS** (dla wierzchołka v) wygląda następująco:

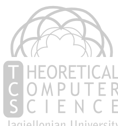
- Wiemy, że osoba uczestnicząca w spotkaniu v nie zaraziła się. Możemy mieć więc pewność, że wszystkie osoby uczestniczące w tym spotkaniu były (w tamtym momencie) zdrowe. Te spośród nich, których pionki nadal znajdują się w wierzchołku v, możemy więc **oznaczyć jako zdrowe** [2], tj. usunąć je ze zbioru *possibly_infected*.
- **Usuwamy** z grafu wierzchołek v (musiał on być "zdrowy").
- Dla każdej krawędzi $u \to v$, wywołujemy rekurencyjnie **DFS(u)**. Skoro bowiem wszystkie osoby obecne na spotkaniu v były (w tamtym momencie) zdrowe, to musiały być zdrowe również podczas wszystkich swoich wcześniejszych spotkań.
- Dla każdej krawędzi $v \to w$, **usuwamy** tę krawędź z grafu oraz sprawdzamy czy była to **ostatnia** krawędź wchodząca do w, która mogła potencjalnie powodować zarażenie tego wierzchołka. Jeżeli tak, wywołujemy **DFS(w)**.

[2] Natomiast osoby, które brały udział w spotkaniu v, lecz od tego czasu ich pionki zostały już przesunięte do dalszych wierzchołków, będą mogły zostać przeanalizowane dopiero kiedy algorytm dotrze do obecnych lokalizacji ich pionków.

W momencie otrzymania zapytania z ministerstwa, wyszukujemy identyfikator poszukiwanej osoby w zbiorze *possibly_infected*.

Jeżeli zbiór ten jest pusty, to nikt (oprócz osób przebywających na kwarantannie) nie może być zarażony, a epidemia została opanowana.





Analiza złożoności

Niech n oznacza liczbę osób, s – liczbę spotkań, zaś c – sumę wielkości wszystkich spotkań.

Analiza złożoności

Niech n oznacza liczbę osób, s – liczbę spotkań, zaś c – sumę wielkości wszystkich spotkań.

- Utworzony graf ma $n + s$ wierzchołków (po jednym wierzchołku dla każdej osoby, i po jednym dla każdego spotkania) oraz c krawędzi (krawędź tworzona jest dla każdej osoby uczestniczącej w każdym spotkaniu).

Analiza złożoności

Niech n oznacza liczbę osób, s – liczbę spotkań, zaś c – sumę wielkości wszystkich spotkań.

- Utworzony graf ma $n + s$ wierzchołków (po jednym wierzchołku dla każdej osoby, i po jednym dla każdego spotkania) oraz c krawędzi (krawędź tworzona jest dla każdej osoby uczestniczącej w każdym spotkaniu).
- Wszystkie wywołania algorytmu DFS we wszystkich iteracjach zajmują łącznie czas $\mathcal{O}(n + s + c)$ ponieważ każdy odwiedzony (w którejkolwiek iteracji) wierzchołek oraz każda analizowana krawędź zostają permanentnie usunięte z grafu.

Analiza złożoności

Niech n oznacza liczbę osób, s – liczbę spotkań, zaś c – sumę wielkości wszystkich spotkań.

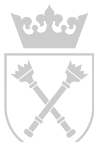
- Utworzony graf ma $n + s$ wierzchołków (po jednym wierzchołku dla każdej osoby, i po jednym dla każdego spotkania) oraz c krawędzi (krawędź tworzona jest dla każdej osoby uczestniczącej w każdym spotkaniu).
- Wszystkie wywołania algorytmu DFS we wszystkich iteracjach zajmują łącznie czas $\mathcal{O}(n + s + c)$ ponieważ każdy odwiedzony (w którejkolwiek iteracji) wierzchołek oraz każda analizowana krawędź zostają permanentnie usunięte z grafu.
- Liczba operacji wykonanych na zbiorze *possibly_infected* jest liniowa od długości wejścia. Każda osoba może bowiem zostać włożona do zbioru co najwyżej tyle razy, w ilu spotkaniach uczestniczyła (i jeden raz na początku algorytmu). Co najwyżej tyle samo razy może też zostać z niego usunięta. Każda z operacji na zbiorze zajmuje czas $\mathcal{O}(\log n)$.

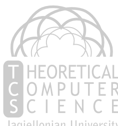
Zadanie A
AMPPZ w czasach zarazy

Zgłoszeń: 46
Zaakceptowanych: 3

Pierwsze rozwiązanie:
UJ1 [Gawryał, Salata, Ziobro]
4:05

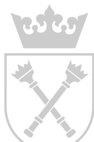
Autor: Krzysztof Maziarz





Zadanie

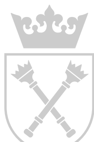
Podziel n punktów na płaszczyźnie na k (niepustych) grup, tak, żeby największa odległość między punktami w tej samej grupie była mniejsza niż najmniejsza odległość między punktami z różnych grup.



Zadanie

Podziel n punktów na płaszczyźnie na k (niepustych) grup, tak, żeby największa odległość między punktami w tej samej grupie była mniejsza niż najmniejsza odległość między punktami z różnych grup.

Uwaga: możesz założyć, że rozwiązanie zawsze istnieje!



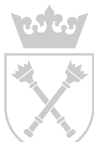
THEORETICAL COMPUTER SCIENCE
Jagiellonian University

Zadanie

Podziel n punktów na płaszczyźnie na k (niepustych) grup, tak, żeby największa odległość między punktami w tej samej grupie była mniejsza niż najmniejsza odległość między punktami z różnych grup.

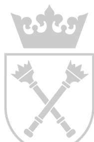
Uwaga: możesz założyć, że rozwiązanie zawsze istnieje!

Limity: $n \leq 2\,000\,000$, $k \leq 20$.



Podejście 1:

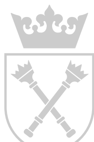
Weźmy dowolny punkt x_1, bez straty ogólności dodajmy go do grupy 1.



Podejście 1:

Weźmy dowolny punkt x_1, bez straty ogólności dodajmy go do grupy 1.

Niech x_2 to punkt najdalszy od x_1 (w przypadku remisów dowolny). Można pokazać, że x_2 musi być w innej grupie niż x_1, niech będzie to grupa 2.

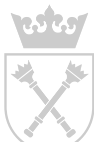


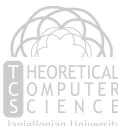
Podejście 1:

Weźmy dowolny punkt x_1, bez straty ogólności dodajmy go do grupy 1.

Niech x_2 to punkt najdalszy od x_1 (w przypadku remisów dowolny). Można pokazać, że x_2 musi być w innej grupie niż x_1, niech będzie to grupa 2.

W ogólności: jeśli przydzieliliśmy już punkty $x_1, ..., x_i$ do grup $1, ..., i$, to x_{i+1} który maksymalizuje minimalną odległość do $x_1, ..., x_i$ musi być z nowej grupy.





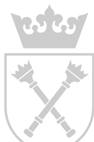
Podejście 1:

Weźmy dowolny punkt x_1, bez straty ogólności dodajmy go do grupy 1.

Niech x_2 to punkt najdalszy od x_1 (w przypadku remisów dowolny). Można pokazać, że x_2 musi być w innej grupie niż x_1, niech będzie to grupa 2.

W ogólności: jeśli przydzieliliśmy już punkty $x_1, ..., x_i$ do grup $1, ..., i$, to x_{i+1} który maksymalizuje minimalną odległość do $x_1, ..., x_i$ musi być z nowej grupy.

Na koniec każdy z pozostałych $n - k$ punktów przypisujemy do najbliższego z $x_1, ..., x_k$.



Podejście 1:

Weźmy dowolny punkt x_1, bez straty ogólności dodajmy go do grupy 1.

Niech x_2 to punkt najdalszy od x_1 (w przypadku remisów dowolny). Można pokazać, że x_2 musi być w innej grupie niż x_1, niech będzie to grupa 2.

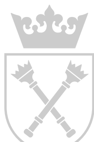
W ogólności: jeśli przydzieliliśmy już punkty $x_1, \ldots, x_i$ do grup $1, \ldots, i$, to x_{i+1} który maksymalizuje minimalną odległość do $x_1, \ldots, x_i$ musi być z nowej grupy.

Na koniec każdy z pozostałych $n - k$ punktów przypisujemy do najbliższego z $x_1, \ldots, x_k$.

Złożoność naiwnej implementacji to $\mathcal{O}(nk^2)$, ale prosto można ją przyspieszyć do $\mathcal{O}(nk)$.

Podejście 2:

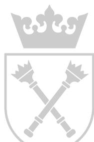
Weźmy dowolne $k+1$ punktów. Któreś dwa muszą być w tej samej grupie, a więc w szczególności dwa najbliższe muszą być w tej samej grupie, oznaczmy je x oraz y.



Podejście 2:

Weźmy dowolne $k+1$ punktów. Któreś dwa muszą być w tej samej grupie, a więc w szczególności dwa najbliższe muszą być w tej samej grupie, oznaczmy je x oraz y.

Dodajemy krawędź (x, y), wyrzucamy jeden z nich ze zbioru, i dorzucamy nowy punkt. Powtarzamy ten proces dla każdego pozostałego punktu; spójne składowe otrzymanego grafu wyznaczają podział na grupy.

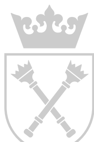


Podejście 2:

Weźmy dowolne $k+1$ punktów. Któreś dwa muszą być w tej samej grupie, a więc w szczególności dwa najbliższe muszą być w tej samej grupie, oznaczmy je x oraz y.

Dodajemy krawędź (x, y), wyrzucamy jeden z nich ze zbioru, i dorzucamy nowy punkt. Powtarzamy ten proces dla każdego pozostałego punktu; spójne składowe otrzymanego grafu wyznaczają podział na grupy.

Złożoność naiwnej implementacji to $\mathcal{O}(nk^2)$, używając set'ów można poprawić teoretyczną złożoność do $\mathcal{O}(nk \log k)$ (w praktyce bardzo wolne).

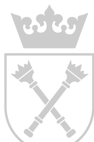


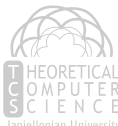
Zadanie G
Glątwa Gebajta

Zgłoszeń: 14
Zaakceptowanych: 2

Pierwsze rozwiązanie:
UW2 [Czarkowski, Staniewski, Nowak]
5:08

Autor: Krzysztof Maziarz





Zadanie

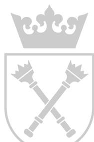
Rozważamy wiedźmina o pewnej liczbie punktów żywotności H, oraz trzy rodzaje obiektów:

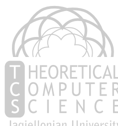
- *bestię*, która zmniejsza żywotność o b_i (i zabija w przypadku $H \leq b_i$);
- *karczmę*, która zabija w przypadku $H < k_i$ i zmniejsza H do poziomu k_i w przeciwnym wypadku; i
- *czarownicę*, która ustawia $H = \max(H, c_i)$.

Szlak wiedźmina to ciąg n obiektów. Musimy obsługiwać q operacji:

- *zmiany*: jeden obiekt zmienia się na inny
- *zapytania*: zaczynamy na pozycji l_i z żywotnością H_0, i zastanawiamy się, do jakiego $r_i \geq l_i$ jesteśmy w stanie dojechać nie umierając

Limity: $n \leq 2\,000\,000$, $q \leq 4\,000\,000$.





Zadanie

Rozważamy wiedźmina o pewnej liczbie punktów żywotności H, oraz trzy rodzaje obiektów:

- *bestię*, która zmniejsza żywotność o b_i (i zabija w przypadku $H \leq b_i$);
- *karczmę*, która zabija w przypadku $H < k_i$ i zmniejsza H do poziomu k_i w przeciwnym wypadku; i
- *czarownicę*, która ustawia $H = \max(H, c_i)$.

Szlak wiedźmina to ciąg n obiektów. Musimy obsługiwać q operacji:

- *zmiany*: jeden obiekt zmienia się na inny
- *zapytania*: zaczynamy na pozycji l_i z żywotnością H_0, i zastanawiamy się, do jakiego $r_i \geq l_i$ jesteśmy w stanie dojechać nie umierając

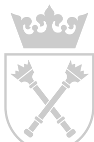
Limity: $n \leq 2\,000\,000$, $q \leq 4\,000\,000$.

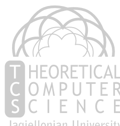
glątwa

Termin wprowadzony przez Witkacego, określający stan który częściej opisujemy dziś słowem "kac".

Jagiellonian University

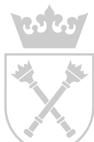
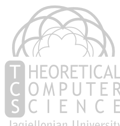
O każdym z obiektów możemy myśleć jako o funkcji przekształcającej życie początkowe wiedźmina w końcowe (lub w 0 jeśli umrze).





O każdym z obiektów możemy myśleć jako o funkcji przekształcającej życie początkowe wiedźmina w końcowe (lub w 0 jeśli umrze).

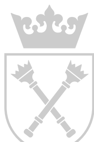
Funkcje dla pojedynczych obiektów są proste; przedział wielu obiektów też jest funkcją, choć bardziej skomplikowaną...

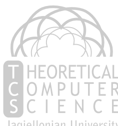



O każdym z obiektów możemy myśleć jako o funkcji przekształcającej życie początkowe wiedźmina w końcowe (lub w 0 jeśli umrze).

Funkcje dla pojedynczych obiektów są proste; przedział wielu obiektów też jest funkcją, choć bardziej skomplikowaną...

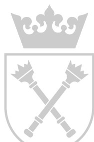
...ale czy bardzo skomplikowaną?

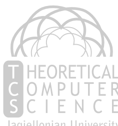




Okazuje się, że *dowolny* ciąg obiektów po złożeniu daje funkcję następującej postaci:

$$f(x) = \begin{cases} 0 & \textit{for } x \in [0, a] \\ y & \textit{for } x \in [a+1, b] \\ x - b + y & \textit{for } x \in [b+1, +\infty] \end{cases}$$

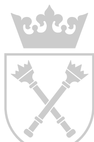


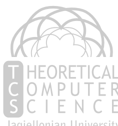


Okazuje się, że *dowolny* ciąg obiektów po złożeniu daje funkcję następującej postaci:

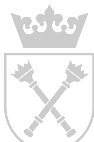
$$f(x) = \begin{cases} 0 & \textit{for } x \in [0, a] \\ y & \textit{for } x \in [a+1, b] \\ x - b + y & \textit{for } x \in [b+1, +\infty] \end{cases}$$

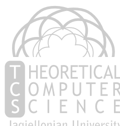
Dowód: każdy z trzech bazowych obiektów jest szczególnym przypadkiem powyższej funkcji, zaś złożenie dwóch funkcji tej postaci daje również funkcję tej postaci.





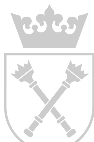
Utrzymujemy drzewo przedziałowe, w każdym węźle trzymamy złożenie obiektów z odpowiedniego przedziału bazowego jako opis wynikowej funkcji (trzy liczby a, b i c).





Utrzymujemy drzewo przedziałowe, w każdym węźle trzymamy złożenie obiektów z odpowiedniego przedziału bazowego jako opis wynikowej funkcji (trzy liczby a, b i c).

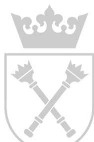
Złożenie dwóch węzłów implementujemy w czasie stałym.



Utrzymujemy drzewo przedziałowe, w każdym węźle trzymamy złożenie obiektów z odpowiedniego przedziału bazowego jako opis wynikowej funkcji (trzy liczby a, b i c).

Złożenie dwóch węzłów implementujemy w czasie stałym.

Operacje zmiany implementujemy zmieniając w liściu, i poprawiając $\mathcal{O}(\log n)$ węzłów. By odpowiedzieć na operację, zaczynamy w liściu odpowiadającym l_i, najpierw idziemy w górę, a potem odpowiednio schodzimy w dół (również czas $\mathcal{O}(\log n)$).

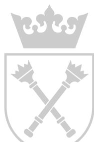


Utrzymujemy drzewo przedziałowe, w każdym węźle trzymamy złożenie obiektów z odpowiedniego przedziału bazowego jako opis wynikowej funkcji (trzy liczby a, b i c).

Złożenie dwóch węzłów implementujemy w czasie stałym.

Operacje zmiany implementujemy zmieniając w liściu, i poprawiając $\mathcal{O}(\log n)$ węzłów. By odpowiedzieć na operację, zaczynamy w liściu odpowiadającym l_i, najpierw idziemy w górę, a potem odpowiednio schodzimy w dół (również czas $\mathcal{O}(\log n)$).

Złożoność: $\mathcal{O}((n+q)\log n)$.

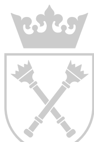


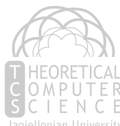
Zadanie B
Babcia i pierogi

Zgłoszeń: 12
Zaakceptowanych: 0

Pierwsze rozwiązanie:
brak

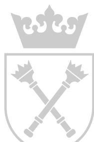
Autor: Daniel Goc



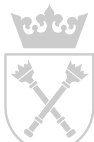


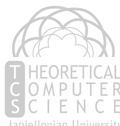
Zadanie

Dany jest ciąg n liczb całkowitych a_i oraz ciąg n liczb całkowitych p_i. W obu ciągach elementy są parami różne, zaś po posortowaniu oba ciągi są identyczne. Dana jest również liczba C. W jednym ruchu możemy wybrać dwa indeksy i oraz j zamienić miejscami liczby a_i oraz a_j płacąc koszt $|a_i - b_j| + C$.
Naszym zadaniem jest minimalnym kosztem zmienić ciąg a w p.



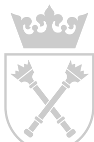
Możemy oszacować minimalny koszt od dołu przez
$\frac{1}{2}\sum_{i=1}^{n}|a_i - p_i| + (n - liczba_cykli_w_permutacji) \cdot C$



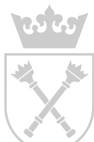


Możemy oszacować minimalny koszt od dołu przez
$\frac{1}{2}\sum_{i=1}^{n}|a_i - p_i| + (n - liczba_cykli_w_permutacji) \cdot C$

Okazuje się, że możemy osiągnąć dokładnie taki koszt!

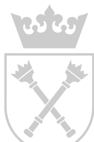


Wszystkie cykle musimy rozwiązać niezależnie, wybierzmy więc jeden z nich i trzymajmy jego elementy w strukturze danych (np. w secie z C++).



Wszystkie cykle musimy rozwiązać niezależnie, wybierzmy więc jeden z nich i trzymajmy jego elementy w strukturze danych (np. w secie z C++).

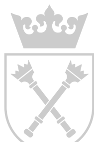
Wybierzmy element cyklu o największej wartości (stół na którym ma się znaleźć największa liczbą pierogów). Okazuje się, że zawsze możemy zamienić go z innym elementem zachowując optymalność i rozbić cykl na dwa.



Wszystkie cykle musimy rozwiązać niezależnie, wybierzmy więc jeden z nich i trzymajmy jego elementy w strukturze danych (np. w secie z C++).

Wybierzmy element cyklu o największej wartości (stół na którym ma się znaleźć największa liczbą pierogów). Okazuje się, że zawsze możemy zamienić go z innym elementem zachowując optymalność i rozbić cykl na dwa.

Aby znaleźć taki element możemy równolegle przeszukiwać cykl w obie strony od największej wartości aż na taki element natrafimy, co pozwoli przerzucać elementy między strukturami co najwyżej $O(n \log n)$ razy.

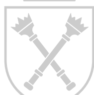


Wszystkie cykle musimy rozwiązać niezależnie, wybierzmy więc jeden z nich i trzymajmy jego elementy w strukturze danych (np. w secie z C++).

Wybierzmy element cyklu o największej wartości (stół na którym ma się znaleźć największa liczbą pierogów). Okazuje się, że zawsze możemy zamienić go z innym elementem zachowując optymalność i rozbić cykl na dwa.

Aby znaleźć taki element możemy równolegle przeszukiwać cykl w obie strony od największej wartości aż na taki element natrafimy, co pozwoli przerzucać elementy między strukturami co najwyżej $O(n \log n)$ razy.

Korzystając ze wspomnianej struktury możemy rozwiązać zadanie w złożoności $O(n \cdot \log^2(n))$. Złożoność tę da się też zmniejszyć, co nie było jednak wymagane.

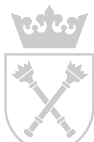


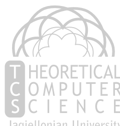
Zadanie M
Magiczne trójki

Zgłoszeń: 0
Zaakceptowanych: 0

Pierwsze rozwiązanie:
brak

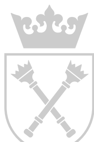
Autor: Krzysztof Maziarz





Zadanie

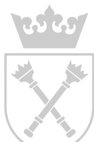
Mamy nietypowy algorytm obliczający medianę. Działa on następująco: jeśli ciąg ma co najwyżej dwa elementy, to algorytm zwraca poprawny wynik; w przeciwnym razie dzieli ciąg na trzy (mniej więcej) równe części, wywołuje się rekurencyjnie, i zwraca medianę trzech otrzymanych liczb.



Zadanie

Mamy nietypowy algorytm obliczający medianę. Działa on następująco: jeśli ciąg ma co najwyżej dwa elementy, to algorytm zwraca poprawny wynik; w przeciwnym razie dzieli ciąg na trzy (mniej więcej) równe części, wywołuje się rekurencyjnie, i zwraca medianę trzech otrzymanych liczb.

Mając dany ciąg n liczb z przedziału $[0, m-1]$, gdzie q wartości jest nieznanych, wyznaczyć (modulo 10^9+7) na ile sposobów można wpisać nieznane wartości, tak, aby opisany algorytm zwracał poprawną medianę.

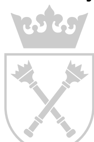


Zadanie

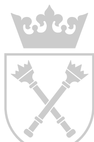
Mamy nietypowy algorytm obliczający medianę. Działa on następująco: jeśli ciąg ma co najwyżej dwa elementy, to algorytm zwraca poprawny wynik; w przeciwnym razie dzieli ciąg na trzy (mniej więcej) równe części, wywołuje się rekurencyjnie, i zwraca medianę trzech otrzymanych liczb.

Mając dany ciąg n liczb z przedziału $[0, m-1]$, gdzie q wartości jest nieznanych, wyznaczyć (modulo $10^9 + 7$) na ile sposobów można wpisać nieznane wartości, tak, aby opisany algorytm zwracał poprawną medianę.

Limity: $n \leq 3^8 = 6561$, $q \leq 30$, $m \leq 10^9$.

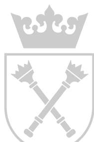


Ustalmy wartość $t \in [0, m-1]$. Na ile sposobów możemy wpisać nieznane wartości, tak, żeby zarówno prawdziwa mediana jak i wynik algorytmu "Magiczne Trójki" wyniosły t?



Ustalmy wartość $t \in [0, m-1]$. Na ile sposobów możemy wpisać nieznane wartości, tak, żeby zarówno prawdziwa mediana jak i wynik algorytmu "Magiczne Trójki" wyniosły t?

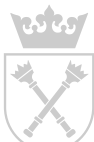
Zastępujemy liczby mniejsze przez t przez -1, równe przez 0, i większe przez 1. Ustalmy ponadto ile razy wpiszemy odpowiednio -1 i 1 pod nieznane wartości jako x i y ($x, y \in [0, q]$). To czy prawdziwa mediana wyjdzie 0 tłumaczy się na proste ograniczenia na x oraz y.

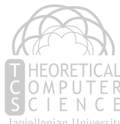


Ustalmy wartość $t \in [0, m-1]$. Na ile sposobów możemy wpisać nieznane wartości, tak, żeby zarówno prawdziwa mediana jak i wynik algorytmu "Magiczne Trójki" wyniosły t?

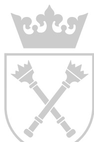
Zastępujemy liczby mniejsze przez t przez -1, równe przez 0, i większe przez 1. Ustalmy ponadto ile razy wpiszemy odpowiednio -1 i 1 pod nieznane wartości jako x i y ($x, y \in [0, q]$). To czy prawdziwa mediana wyjdzie 0 tłumaczy się na proste ograniczenia na x oraz y.

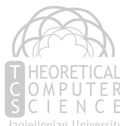
Jeśli przy danych x i y prawdziwa mediana wychodzi 0, to wyliczamy odpowiednią liczbę sposobów które sprawiają że "Magiczne trójki" również zwraca 0, mnożymy wynik przez $t^x(m-1-t)^y$, i dodajemy do wyniku.





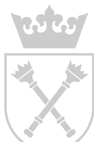
Szukaną liczbę sposobów dla wszystkich x i y możemy wyliczyć programowaniem dynamicznym po drzewie ternarnym opisującym przebieg "Magicznych trójek". Stan w każdym węźle to: x i y, oraz wynik jaki chcemy uzyskać ($\{-1, 0, 1\}$).





Szukaną liczbę sposobów dla wszystkich x i y możemy wyliczyć programowaniem dynamicznym po drzewie ternarnym opisującym przebieg "Magicznych trójek". Stan w każdym węźle to: x i y, oraz wynik jaki chcemy uzyskać ($\{-1, 0, 1\}$).

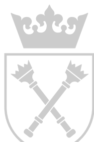
Powyższe programowanie dynamiczne działa w czasie $\mathcal{O}(n + q^4)$. Niestety, uruchamianie go dla każdego t byłoby zbyt wolne.



Szukaną liczbę sposobów dla wszystkich x i y możemy wyliczyć programowaniem dynamicznym po drzewie ternarnym opisującym przebieg "Magicznych trójek". Stan w każdym węźle to: x i y, oraz wynik jaki chcemy uzyskać ($\{-1, 0, 1\}$).

Powyższe programowanie dynamiczne działa w czasie $\mathcal{O}(n + q^4)$. Niestety, uruchamianie go dla każdego t byłoby zbyt wolne.

Niech $x_1 < x_2 < \dots < x_l$ to *zbiór* znanych nam wartości ciągu.



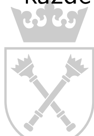
THEORETICAL COMPUTER SCIENCE
Jagiellonian University

Szukaną liczbę sposobów dla wszystkich x i y możemy wyliczyć programowaniem dynamicznym po drzewie ternarnym opisującym przebieg "Magicznych trójek". Stan w każdym węźle to: x i y, oraz wynik jaki chcemy uzyskać ($\{-1, 0, 1\}$).

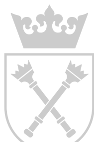
Powyższe programowanie dynamiczne działa w czasie $\mathcal{O}(n + q^4)$. Niestety, uruchamianie go dla każdego t byłoby zbyt wolne.

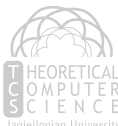
Niech $x_1 < x_2 < ... < x_l$ to *zbiór* znanych nam wartości ciągu.

Rozważmy $\mathcal{O}(n)$ przedziałów postaci $[0, x_1 - 1]$, $[x_1, x_1]$, $[x_1 + 1, x_2 - 1]$, $[x_2, x_2]$, ..., $[x_l + 1, m - 1]$. Zauważmy, że jeśli p jest dowolnym z nich, to każde $t \in p$ da nam ten sam przebieg dynamika.



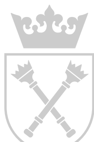
Ponadto, można pokazać, że tylko $\mathcal{O}(q)$ przedziałów ma szansę dać nam niezerową kontrybucję do odpowiedzi (intuicyjnie, mediana ciągu z uzupełnionymi nieznanymi wartościami musi być blisko mediany multizbioru wpisanych już wartości).

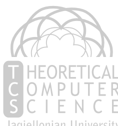




Ponadto, można pokazać, że tylko $\mathcal{O}(q)$ przedziałów ma szansę dać nam niezerową kontrybucję do odpowiedzi (intuicyjnie, mediana ciągu z uzupełnionymi nieznanymi wartościami musi być blisko mediany multizbioru wpisanych już wartości).

Czas wywoływania programowania dynamicznego spada już do $\mathcal{O}(nq + q^5)$, co jest już akceptowalne.

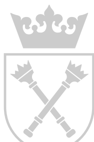




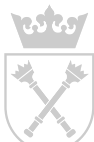
Ponadto, można pokazać, że tylko $\mathcal{O}(q)$ przedziałów ma szansę dać nam niezerową kontrybucję do odpowiedzi (intuicyjnie, mediana ciągu z uzupełnionymi nieznanymi wartościami musi być blisko mediany multizbioru wpisanych już wartości).

Czas wywoływania programowania dynamicznego spada już do $\mathcal{O}(nq + q^5)$, co jest już akceptowalne.

Pozostaje jeden problem: gdy ustalaliśmy dokładne t, do odpowiedzi dodawaliśmy składniki $t^x(m-1-t)^y$. Teraz musimy umieć szybko sumować takie wyrażenia po $t \in [L, R]$ (dla wielu różnych x, y, L, R).

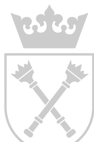


Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.



Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.

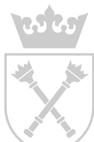
Bez straty ogólności $L = 1$.

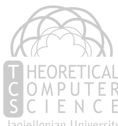


Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.

Bez straty ogólności $L = 1$.

Przy ustalonych x oraz y, $f(R) = \sum_{t=1}^{t=R} t^x(m-1-t)^y$ jest wielomianem stopnia q.



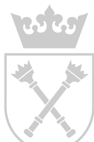


Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.

Bez straty ogólności $L = 1$.

Przy ustalonych x oraz y, $f(R) = \sum_{t=1}^{t=R} t^x(m-1-t)^y$ jest wielomianem stopnia q.

Możemy więc na przykład wyznaczyć jego q pierwszych wartości, i dokonać (naiwnej) interpolacji wielomianowej.



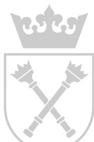
Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.

Bez straty ogólności $L = 1$.

Przy ustalonych x oraz y, $f(R) = \sum_{t=1}^{t=R} t^x(m-1-t)^y$ jest wielomianem stopnia q.

Możemy więc na przykład wyznaczyć jego q pierwszych wartości, i dokonać (naiwnej) interpolacji wielomianowej.

Możemy też sprowadzić problem do obliczania $\sum_{t=1}^{t=R} t^c$; te sumy dla wszystkich $c = 1, ..., q$ można wyznaczyć programowaniem dynamicznym.



Sumy tej postaci liczymy $\mathcal{O}(q^3)$ razy, więc musimy je liczyć w $\mathcal{O}(q^2)$ żeby nie zdominowało to złożoności rozwiązania.

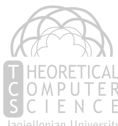
Bez straty ogólności $L = 1$.

Przy ustalonych x oraz y, $f(R) = \sum_{t=1}^{t=R} t^x(m-1-t)^y$ jest wielomianem stopnia q.

Możemy więc na przykład wyznaczyć jego q pierwszych wartości, i dokonać (naiwnej) interpolacji wielomianowej.

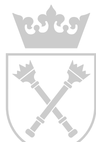
Możemy też sprowadzić problem do obliczania $\sum_{t=1}^{t=R} t^c$; te sumy dla wszystkich $c = 1, ..., q$ można wyznaczyć programowaniem dynamicznym.

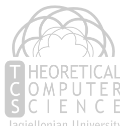
Finalnie mamy złożoność $\mathcal{O}(nq + q^5)$, i dostajemy zasłużone OK.



Jury zawodów

Lech Duraj
Krzysztof Maziarz
Krzysztof Kleiner
Daniel Goc
Mateusz Radecki





Betatesterzy

Rafał Burczyński
Marcin Briański
Kamil Rajtar
Witold Jarnicki
Jan Tułowiecki
Adam Szady
Mateusz Radecki
Kamil Dębowski
Marek Sommer

Dziękujemy!