

Rozdział 1

Lista 1

1.1.

1.2.

1.3.

1.4.

$$a_1 = a, \quad a_k = 1, \quad a_{i+1} = \lfloor \frac{a_i}{2} \rfloor$$

$$b_a = b, \quad b_{i+1} = 2b_i \Rightarrow b_i = 2^i b$$

Niech $a = \sum_{i=1}^k 2^{i-1} \bar{a}_i$, $\bar{a}_i \in \{0, 1\}$, czyli zapis w postaci binarnej. Wtedy

$$a_n = \sum_{i=1}^n 2^{i-1} \bar{a}_{i+(k-n)}$$

Dowód:

$$\sum_{i=1, \text{ nieparzyste } a_i}^k b_i = \sum_{i=1}^k \bar{a}_i 2^i b = b \sum_{i=1}^k 2^i \bar{a}_i = ab$$

Kryterium jednorodne - koszt każdej operacji jest jednostkowy. Zatem ile jedynek w zapisie binarnym liczby b , taką mamy złożoność $\Rightarrow O(\lceil \log_2 b \rceil)$

Kryterium logarytmiczne - koszt operacji zależy od odległości operandów. Dodawań mamy tyle samo $\Rightarrow O(\lceil \log_2 b \rceil \cdot \lceil \log_2 ab \rceil)$

1.5.

Niech $f_n = A_k f_{n-k} + A_{k-1} f_{n-k+1} + \dots + A_1 f_{n-1}$. Jeżeli f_n zależy od k poprzednich elementów to tworzymy następującą macierz $k \times k$ (x-kolumny, y-wiersze, indeksowanie od 0)

$$M_{x,y} = 1, \text{ dla } x = y + 1$$

$$M_{x,k} = A_{n-k+x}$$

Reszta elementów to 0. Przykład dla $k = 3$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ A_{n-3} & A_{n-2} & A_{n-1} \end{bmatrix}$$

Żeby policzyć f_n tworzymy wektor $F = (f_{n-k} \dots f_{n-1})$ i obliczamy $M \cdot F^T$.

Tworzymy macierz rozmiaru $(k + m) \times (k + m)$, k - ilość poprzednich wyrazów ciągu, m - stopień wielomianu. Dzielimy ją na cztery prostokąty. Lewy górny taki jak w poprzedniej części. Prawy górny wypełniamy zerami, oprócz ostatniego wiersza, który wypełniamy współczynnikami wielomianu. Lewy dolny wypełniamy zerami. Żeby wypełnić prawy dolny zastanówmy się najpierw jak uzyskać $(n + 1)^i$. Oczywiście z dwumianu newtona. prawy dolny prostokąt będzie miał postać

$$\begin{bmatrix} \binom{m}{m} & \dots & \dots & \binom{m}{0} \\ 0 & \binom{m-1}{m-1} & \dots & \binom{m-1}{0} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Znowu tworzymy sobie wektor $F = (f_{n-k} \dots f_{n-1}, n^m, n^{m-1}, \dots, 1)$ i obliczamy $M \cdot F^T$.

1.6.

1.7.

Mamy daną listę L . Dzielimy ją na podlisty długości \sqrt{n} . Tworzymy dodatkową listę K o długości \sqrt{n} , zawierającą wskaźniki na pierwsze elementy utworzonych wcześniej podlist. Przy wstawianiu elementu przeglądamy najpierw Listę K , a następnie listę L od miejsca, na które wskazywał wskaźnik z K . Maksymalnie przejrzymy $2\sqrt{n}$ elementów. Po wstawieniu elementu musimy ouaktualnić listę wskaźników K , czego koszt to znowu \sqrt{n}

1.8.

Wejście: Skierowany acykliczny graf.

Wyjście: Długość najdłuższej ścieżki.

LengthTo - tablica $|V(G)|$ elementów początkowo równych 0.

TopOrder(G) - posortowane topologicznie wierzchołki.

```

for each vertex V in topOrder(G) do
  for each edge (V, W) in E(G) do
    if LengthTo[W] <= LengthTo[V] + weight(G, (V,W)) then
      LengthTo[W] = LengthTo[V] + weight(G, (V,W))

return max(LengthTo[V] for V in V(G))

```

Sortowanie topologiczne działa w czasie $O(E + V)$, więc całość działa w czasie $O(E + V + E + V) = O(E + V)$. Żeby wypisać drogę musimy tylko zapamiętywać, dla których wierzchołków spełniony był IF.

Rozdział 2

Lista 2

2.1.

2.2.

Sortujemy odcinki rosnąco wg. k_j . Wybieramy pierwszy, potem kolejny, który się zmieści po pierwszym.

Dowód: Niech nasz algorytm daje uporządkowanie U . Załóżmy, że istnieje lepsze, optymalne uporządkowanie S . Weźmy pierwszą parę odcinków, które różnią się w obu tych uporządkowaniach, tzn. $U_i \neq S_i$. Jeżeli odcinek S_i kończy się wcześniej niż U_i to zostałby on wybrany przez nasz algorytm. Jeśli kończy się na tej samej pozycji to nie ma znaczenia, który z nich został wybrany. Podobnie, indukcyjnie można zastosować to rozumowanie dla pozostałych par różnych odcinków w obu uporządkowaniach. Algorytm zachłanny daje więc rozwiązanie optymalne.

2.3.

Wytwarzanie ułamka egipskiego mniejszego od $\frac{x}{y}$ o największym mianowniku:

$$\frac{x}{y} \rightarrow \frac{1}{\lceil \frac{y}{x} \rceil}$$

Licznik wyrażenia $\frac{x}{y} - \frac{1}{\lceil \frac{y}{x} \rceil} = \frac{x\lceil \frac{y}{x} \rceil - y}{x\lceil \frac{y}{x} \rceil}$ będzie mała, lecz zawsze będzie ≥ 0 :

Sprawdźmy, czy faktycznie $0 \leq x\lceil \frac{y}{x} \rceil - y < x$

$$x\lceil \frac{y}{x} \rceil - y < x \Leftrightarrow x\lceil \frac{y}{x} \rceil - x < y \Leftrightarrow \lceil \frac{y}{x} \rceil - 1 < \frac{y}{x}.$$

Co jest oczywiście prawdą. Dodatkowo: $x\lceil \frac{y}{x} \rceil - y \geq 0$. Więc ułamek zawsze się zmniejsza, i zatrzyma się na 0.

Kontrprzykład na optymalność. Nasz da jakieś gówno, optymalny da:

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

2.4.

2.4.1. Lemat 1

W każdym momencie działania algorytmu, oraz po jego zakończeniu w E' nie będzie cyklu.

Dowód: Załóżmy nie wprost, że podczas działania algorytmu w którymś etapie pojawiła się spójna składowa, w której jest cykl. Oznaczmy ją jako S . Rozważmy następujące sytuacje:

- S powstała przez połączenie dwóch superwierzchołków v_1 i v_2 . Oznacza to, że do zbioru E' zostały dołączone krawędzie e_i i e_j . Ponieważ e_i została dołączona jako najbliższa krawędź incydentna do v_1 więc $C(e_i) < C(e_j)$. Ale skoro e_j została dołączona jako najbliższa krawędź incydentna do v_2 to musi zachodzić (Pamiętajmy, że w grafie nie ma krawędzi o takiej samej wadze) - sprzeczność.
- S powstała przez połączenie się trzech lub więcej superwierzchołków. Podzielmy powstały cykl C na następujące części: niech v_1, \dots, v_l będą kolejnymi superwierzchołkami należącymi do C a e_1, \dots, e_l będą kolejnymi krawędziami należącymi do C , które zostały dodane w zakończonym właśnie etapie algorytmu. W C krawędzie e_i oraz superwierzchołki v_i występują na przemian. Z zasady działania algorytmu możemy stwierdzić, że aby powstał taki cykl, musi zachodzić $C(e_1) < C(e_2) \dots < C(e_l) < C(e_1)$ - Sprzeczność.

2.4.2. Lemat 2

W każdym etapie działania algorytmu otrzymujemy dla każdego superwierzchołka minimalne drzewo rozpinające.

Dowód:

- Gdy zostanie zakończony etap 1:

Założmy, że istnieje taki superwierzchołek v_i , który nie jest minimalnym drzewem rozpinającym poddrzewa złożonego z wierzchołków należących do v_i . Weźmy więc takie minimalne drzewo rozpinające T . Istnieje krawędź e_i taka, że $e_i \in E(v_i)$ oraz $e_i \notin E(T)$. Dodajmy e_i . W T powstał cykl. Ponieważ e_i jest incydentna do pewnego wierzchołka z tego cyklu, istnieje więc inna krawędź e'_i incydentna do tego samego wierzchołka. Jednak z tego, że $e_i \in E(v_i)$ wynika, że $C(e_i) < C(e'_i)$. Jeśli usuniemy krawędź e'_i z T otrzymamy mniejsze drzewo rozpinające, co jest sprzeczne z założeniem o minimalności T .

- Gdy zostanie zakończony etap 2:

Z poprawności etapu 1 wiemy, że istnieje takie wywołanie etapu 2, w którym każdy z superwierzchołków jest minimalnym drzewem rozpinającym. Jest to choćby pierwsze wywołanie. Założmy zatem, że dla pewnego wywołania tego etapu otrzymano superwierzchołki będące minimalnymi drzewami rozpinającymi, jednak scaliło przynajmniej dwa z nich w taki sposób, że dało się otrzymać mniejsze drzewo rozpinające. Niech etap k -ty będzie pierwszym takim etapem, w którym coś się popsło. Niech E'_1 będzie zbiorem krawędzi przed wywołaniem etapu k , a E'_2 będzie zbiorem krawędzi po jego wywołaniu. Niech T będzie minimalnym drzewem rozpinającym takim, że $V(T) = V(v_i)$, ale że $E(T) \neq E(v_i)$. Istnieje więc krawędź $e_i \in E(v_i)$ oraz $e_i \notin E(T)$.

Fakt: Krawędź dodana podczas k -tego wywołania. (Nie może należeć do E'_1 gdyż inaczej superwierzchołek do którego by należała nie byłby minimalnym drzewem rozpinającym, co jest sprzeczne z dowodem dla pierwszego etapu i założeniem, że wywołanie k -te jest najmniejszym wywołaniem, które zwróciło nieoptymalne drzewa) Dodajmy krawędź e_i do $E(T)$. W

T powstał cykl. Ponieważ e_i jest najmniejszą krawędzią incydentną do pewnego superwierzchołka z tego cyklu, istnieje więc inna krawędź incydenta do tego samego superwierzchołka. Jednak jej waga jest większa niż waga krawędzi e_i , zatem zastąpienie jej krawędzią e_i da nam mniejsze drzewo rozpinające co jest sprzeczne z założeniem o optymalności T .

2.5.

2.6.

Rozdział 3

Lista 3

3.1.

Zadanie to polega na skonstruowaniu szybkiego algorytmu obliczania największego wspólnego dzielnika dwóch dodatnich liczb całkowitych a i b . Przed podaniem algorytmu musimy udowodnić podane właściwości:

$$\gcd(a, b) = \begin{cases} 2 \cdot \gcd(\frac{a}{2}, \frac{b}{2}) & \text{a, b są parzyste;} \\ \gcd(a, \frac{b}{2}) & \text{a jest nieparzyste, b jest parzyste;} \\ \gcd(\frac{a-b}{2}, b) & \text{a, b są nieparzyste.} \end{cases}$$

3.1.1. a)

Jeżeli a i b są parzyste, 2 na pewno jest ich wspólnym dzielnikiem. Jeżeli a jest nieparzyste i b jest parzyste, wiemy że b dzieli się przez 2, a a nie. Więc $\gcd(a, b)$ pozostaje takie same dla a i $b/2$. Ostatnia własność wynika z faktu, że dla nieparzystych a i b , $(a - b)$ będzie parzyste. Ponieważ $\gcd(a - b, b) = \gcd(a, b)$ oraz $(a - b)$ jest teraz parzyste, możemy zastosować drugą własność.

3.1.2. b) algorytm rekurencyjny

```
procedure gcd(a, b)
Input: Two n-bit integers a, b
Output: GCD of a and b
if a = b:
    return a
else if (a is even and b is even):
    return 2*gcd(a/2, b/2)
else if (a is odd and b is even):
    return gcd(a, b/2)
else if (a is odd and b is odd and a > b):
    return gcd((a-b)/2, b)
else if (a is odd and b is odd and a < b):
    return gcd(a, (b-a)/2)
```

3.1.3. c) złożoność

Założmy, że a i b są n -bitowymi liczbami. Rozmiar a i b wynosi $2n$ bitów. Wszystkie z czterech ifów, oprócz przypadku gdzie a jest nieparzyste i b jest parzyste, zmniejsza rozmiar a i b do $2n - 2$ bitów, gdzie wcześniej wymieniony przypadek zmniejsza ilość bitów do $2n - 1$. Każda z operacji wykonuje się w czasie stałym ponieważ dzielimy lub mnożymy przez 2. Dla dwóch przypadków z

odejmowaniem, mamy odejmowanie dwóch n -bitowych liczb (złożoność wynosi $c \cdot n$ gdzie n jest wielkością operandu). Zatem najgorszy przypadek czwartego ifa algorytmu przedstawimy jako:

$$\begin{aligned} T(2n) &= T(n-1) + cn \\ T(2n-1) &= T(2n-2) + cn \\ T(2n-2) &= T(2n-3) + c(n-1) \text{ oba operandy mają długość } n-1 \\ T(2n-3) &= T(2n-4) + c(n-1) \\ &\dots \\ T(2) &= T(1) + c \end{aligned}$$

Podstawieniami możemy zapisać $T(2n)$ jako:

$$T(2n) = 2c \cdot \sum_{i=1}^n i$$

co daje nam $O(n^2)$ co w porównaniu do $O(n^3)$ czasu działania algorytmu euklidesa jest szybsze.

3.2.

PDF

3.3.

Średnicą drzewa nazywamy największą odległość między parami wierzchołków. Będziemy rozważali drzewa ukorzenione w wierzchołku r . Wprowadźmy pewne oznaczenia: analizujemy wierzchołek v , który ma m synów ponumerowanych od 0 do $m-1$. Niech $T(v)$ oznacza poddrzewo zaczepione w wierzchołku v , natomiast $s[v]$ jego rozmiar (liczbę wierzchołków).

Dla każdego wierzchołka będziemy pamiętali dwie liczby: $tak[v]$ - długość najdłuższej ścieżki z $T(v)$, która kończy się w wierzchołku v oraz $nie[v]$ - długość najdłuższej ścieżki z $T(v)$, która nie kończy się w v . Średnica drzewa jest równa $\max(tak[r], nie[r])$.

Tablice tak i nie wyliczamy za pomocą następującej rekurencji:

$$\begin{aligned} tak[v] &= \max_{0 \leq i < m} (1 + tak[i]) \\ nie[v] &= \max(\max_{0 \leq i < m} (nie[i]), \max_{0 \leq i < m} (tak[i]), \max_{0 \leq i < j < m} (tak[i] + 2 + tak[j])) \end{aligned}$$

Dla wierzchołka v obliczenia zabierają czas $O(m)$ (wyszukanie dwóch największych elementów w tablicy da się zrobić liniowo). Zatem dla całego drzewa czas wynosi $O(n)$.

3.4.

Ukorzeniamy drzewo w pewnym wierzchołku. Nazwijmy ten wierzchołek u , a jego dzieci v_1, v_2, \dots, v_n . Zauważamy, że ścieżka długości C albo przechodzi przez ten korzeń albo nie przechodzi:

- jeśli ścieżka nie przechodzi przez korzeń, to znajduje się z którymś z n niższych poddrzew z wierzchołkami v_1, v_2, \dots, v_n (innymi słowy dziel i zwyciężaj).
- jeśli ścieżka przechodzi przez korzeń to albo wychodzi bezpośrednio z korzenia i przechodzi przez jakiś wierzchołek v_i i idzie niżej (czyli ścieżka jedno-fałowa) albo łączy ze sobą dwa wierzchołki v_i i v_j i z dwóch stron idzie niżej (nazwijmy taki typ ścieżki dwu-fałowy). Dodatkowo musimy zapewniać, że połączone ścieżki są długości C .

Zacniemy od końca, czyli od punktu drugiego. Mamy już jakiś korzeń i chcemy policzyć, ile ścieżek przechodzący przez ten korzeń jest długości C . Zauważmy, że łatwo obliczyć odległość dowolnego wierzchołka do korzenia (wystarczy DFS). Teraz stwórzmy sobie strukturę, która potrafi nam coś zrobić:

- $ile(d, 1..i)$ - liczba ścieżek dochodzący do korzenia, które:
 - a) są długości d ;
 - b) przechodzą przez któryś z wierzchołków v_1, \dots, v_i .
- $wstaw(d, i+1)$ - dodaje informację do bazy o ścieżek do korzenia długości d , która, przechodzi przez wierzchołek v_{i+1} .

Drzewo łatwo sobie na rysować na kartce. Jeśli popatrzymy na ścieżkę, to jakiś fragment idzie tak jakby z lewej na prawo (lub na odwrót, zależy jak popatrzymy). Nas będzie interesować ten wierzchołek na końcu ścieżki, który jest najbardziej na prawo, oznaczmy go przez x . Niech $d(x)$ to odległość do korzenia. Po drodze zauważamy, że ścieżka przechodzi przez wierzchołek v_j , a potem idzie jeszcze bardziej w lewo przez jakiś wierzchołek v_i , czyli musi być, że $i < j$. Łatwo zauważyć, że wszystkich ścieżek długości C , a kończących się na x jest dokładnie tyle ile jest "lewych" ścieżek dochodzących do korzenia, ale długości $C - d(x)$, czyli $ile(C - d(x), 1..j - 1)$. Można się domyślić, że algorytm będzie wyglądał tak:

1. początkowo $ile(d, 1..0) = 0$, bo wiemy, że nic nie wiemy, za wyjątkiem $ile(0, 1..0) = 1$, bo jest jedna ścieżka długości 0.
2. Dla kolejnych $i = 1, 2, \dots, n$, wykonujemy poniższe czynności:
 - a) obliczamy odległości do korzenia wierzchołków w poddrzewie v_i
 - b) do wyniku dodajemy $ile(C - d(x), 1..(i - 1))$ (x - wierzchołek w poddrzewie)
 - c) uaktualnimy ile poprzez wywołanie $wstaw(d(x), i + 1)$ dla wszystkich x w poddrzewie v_i

Łatwo zauważyć, że parametr i jest niepotrzebny. Całą strukturę łatwo można stworzyć na jakimś drzewie zrównoważonym (czyli mapie). Wykonujemy dokładnie $O(m)$ zapytań, które kosztują nas $O(\log m)$, bo w mapie może być co najwyżej m różnych liczb, bo jest co najwyżej m ścieżek do korzenia, zatem złożoność wynosi $O(m \log m)$.

Teraz powróćmy do punktu pierwszego, cały algorytm będzie wyglądał tak:

1. Znajdź jakiś fajny wierzchołek, który dzieli graf na "połowę". (zrobimy to w czasie $O(m)$)
2. Policz ile jest ścieżek przechodzących przez korzeń o długości C (opisane wyżej). (Czas $O(m \log m)$)
3. Wykonaj rekurencyjnie to samo, ale na poddrzewach.

Teraz jak znaleźć fajny wierzchołek. Okazuje się, że istnieje taki wierzchołek, że jeśli go ukorzenimy, to każde poddrzewo jest wielkości co najwyżej $\lceil n/2 \rceil$. A jak go znaleźć: bierzemy wierzchołek v , jeśli nie spełnia założeń, to wchodzimy do największego poddrzewa (dowód poprawności jest podobny, nie wprost, pokazujemy, że można zejść niżej).

Całkowita złożoność wynosi $O(m \log^2 m)$.

3.5.

3.5.1. a)

W wektorze pamiętamy pierwszą kolumnę oraz pierwszy wiersz bez pierwszego wyrazu. Wektor ten ma rozmiar $2n - 1$, więc dodawanie dwóch takich wektorów mamy w $O(n)$.

3.5.2. b)

Macierz Toeplitza ma następującą postać blokową:

$$T = \begin{bmatrix} A & B \\ C & A \end{bmatrix}$$

Zadanie polega na pomnożeniu macierzy T przez wektor blokowy $T = \begin{bmatrix} x \\ y \end{bmatrix}$ w czasie mniejszym niż n^2 . Korzystając z dwóch (wzajemnie dualnych) obserwacji:

$$Ax = A(x + y - y) = A(x + y) - Ay$$

$$Ay = A(x + y - x) = A(x + y) - Ax$$

mamy:

$$\begin{aligned} T \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} A & B \\ C & A \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Ay \end{bmatrix} = \\ &= \begin{bmatrix} A(x + y) - Ay + By \\ Cx + Ay \end{bmatrix} = \begin{bmatrix} A(x + y) + (B - A)y \\ Cx + Ay \end{bmatrix} = \\ &= \begin{bmatrix} A(x + y) + (B - A)y \\ Cx + A(x + y) - Ax \end{bmatrix} = \begin{bmatrix} A(x + y) + (B - A)y \\ A(x + y) + (C - A)x \end{bmatrix} \end{aligned}$$

I zauważamy, że złożoność czasowa to $T(n) = 3T(n/2) + O(n)$ gdzie $O(n)$ zamyka sumę liniowych czasów wszystkich dodawań.

Rozwiązując typowe równanie rekurencyjne mamy $T(n) = O(n^{\log_2 3}) < O(n^2)$.

3.6.

3.7.

Rozdział 4

Lista 4

Rozdział 5

Lista 5