Lista Zadań Nr 0

Algorytmy i Struktury Danych

Łukasz Stodółka

March 3, 2025

1. Złożoności (1 pkt)

Określ, z dokładnością do Θ złożoność (przy kryterium jednorodnym) następujących fragmentów kodu. Dla:

- $P(i, j) = \Theta(1)$,
- $P(i, j) = \Theta(j)$.

Rozwiązanie:

Kod 1:

Algorithm alg(n)

```
for i ← 1 to n do
    j ← i
    while j < n do
        sum ← P(i, j)
        j ← j + 1
    end while
end for
```

- W pierwszej pętli wykonujemy 1 operację:
  - przypisanie $j = i$, co kosztuje 1.
- Następnie wykonujemy pętlę, która iteruje $(n - i - 1)$ razy. W każdej iteracji wykonujemy 2 operacje:
  - przypisanie $P(i, j)$, które kosztuje $P(i, j)$,
  - przypisanie $j = j + i$, które kosztuje 1.
- Łączny koszt pętli wewnętrznej wynosi:

$$\sum_{j=i}^{n-1}(P(i, j) + 1) = n - i + \sum_{j=i}^{n-1}(P(i, j))$$

Czyli łącznie algorytm potrzebuje:

$$\sum_{i=1}^{n}(1+n-i+\sum_{j=i}^{n-1}(P(i,j))$$

Dla $P(i,j)$ w złożoności $\Theta(1)$ koszt $P(i,j)$ wynosi 1 więc algorytm wykonuje:

$$\sum_{i=1}^{n}(2n-2i+1) = 2n^2 - n^2 - n + n = n^2$$

operacji czyli w złożoności

$$\Theta(n^2).$$

Dla $P(i,j)$ w złożoności $\Theta(j)$ koszt $P(i,j)$ wynosi j a j jest uzależnione od iteracji pętli koszt P(i, j) dla ustalonego i wyniesie:

$$\sum_{j=i}^{n-1}(j) = \frac{(n-1)n}{2} - \frac{(i-1)i}{2}$$

Podstawiając:

$$\sum_{i=1}^{n}(1+n-i+\frac{(n-1)n}{2} - \frac{(i-1)i}{2}) = \frac{1}{2}\sum_{i=1}^{n}(n^2 - i^2 + n - i + 2) =$$

$$= \frac{1}{2}\left[n^3 - \frac{n(n+1)(2n+1)}{6} + n^2 - \frac{n(n+1)}{2} + 2n\right] =$$

$$= \frac{n^3+2n}{3}$$

operacji czyli w złożoności

$$\Theta(n^3).$$

Kod 2:

Algorithm alg(n)

```
1: for i ← 1 to n do
2:     j ← i
3:     while j < n do
4:         sum ← P(i, j)
5:         j ← j + j
6:     end while
7: end for
```

1. **Pętla zewnętrzna:**
   Dla każdego i od 1 do n wykonujemy przypisanie:
   $$j \leftarrow i,$$
   co kosztuje 1 operację.

2. **Pętla wewnętrzna:**
   Wewnątrz pętli mamy:
   - Operację: $sum \leftarrow P(i, j)$,
   - Operację: $j \leftarrow j + j$ (czyli podwajanie j).

   Warto zauważyć, że zamiast standardowej inkrementacji j, tutaj j przyjmuje kolejne wartości:
   $$i,\ 2i,\ 4i,\ 8i,\ \ldots$$
   Pętla wewnętrzna trwa tak długo, jak długo $j < n$. Liczbę iteracji dla danego i oznaczamy przez k i mamy:
   $$2^k \cdot i < n \quad \Longrightarrow \quad k < \log_2 \frac{n}{i}.$$
   Dlatego liczba iteracji wynosi:
   $$\lceil \log_2(n/i) \rceil.$$

3. **Koszt jednej iteracji wewnętrznej:**
   W każdej iteracji wykonujemy:
   $$P(i, j) \quad \text{oraz} \quad j \leftarrow j + j.$$
   Zakładamy, że koszt wywołania $P(i, j)$ jest ogólnie zależny od i i j i oznaczamy go przez $P(i, j)$, natomiast operacja przypisania $j \leftarrow j + j$ kosztuje 1.

4. **Całkowity koszt algorytmu:**
   Dla ustalonego i:
   - Koszt przypisania $j = i$ wynosi 1.
   - Koszt pętli wewnętrznej wynosi sumę kosztów poszczególnych iteracji. Ponieważ j przyjmuje wartości $j = 2^k \cdot i$ dla $k = 0, 1, 2, \ldots, \lceil \log_2(n/i) \rceil - 1$, to koszt każdej iteracji wynosi:
     $$P(i, 2^k \cdot i) + 1.$$

Zatem łączny koszt dla danego i to:

$$1 + \sum_{k=0}^{\lceil \log_2(n/i) \rceil - 1} \Big(P(i, 2^k \cdot i) + 1 \Big).$$

Sumując koszty dla wszystkich i od 1 do n, otrzymujemy ostateczny wzór:

$$\sum_{i=1}^{n} \left(1 + \sum_{k=0}^{\lceil \log_2(n/i) \rceil - 1} \Big(P(i, 2^k \cdot i) + 1 \Big) \right).$$

Dla $P(i,j)$ w złożoności $\Theta(1)$ koszt $P(i,j)$ wynosi 1 więc algorytm wykonuje:

$$\sum_{i=1}^{n} \left(1 + \sum_{k=0}^{\lceil \log_2(n/i) \rceil - 1} (2) \right) = n + 2 \sum_{i=1}^{n} (\lceil \log_2(n) - \log_2(i) \rceil) = n + 2n \lceil \log_2(n) \rceil - 2 \sum_{i=1}^{n} (\lfloor \log_2(i) \rfloor - \delta)$$

Dla $\{\log_2(i)\} < \{\log_2(n)\}$

$$\delta = 1$$

W przeciwnym razie

$$\delta = 0$$

Rozważmy

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor \leq \int_1^n \log_2 x \, dx = \frac{1}{\ln 2} \int_1^n \ln x \, dx = \frac{n \ln n - n + 1}{\ln 2} \approx \frac{n \ln n - n}{\ln 2}.$$

widać więc, że w ogólności algorytm będzie w złożoności:

$$\Theta(n log_2(n)).$$

Dla $P(i,j)$ w złożoności $\Theta(j)$ mamy:

$$\sum_{i=1}^{n} \left(1 + \sum_{k=0}^{\lceil \log_2(n/i) \rceil - 1} \Big(2^k \cdot i + 1 \Big) \right) = \frac{\left(2^{\lceil \log_2(n/i) \rceil} - 1 \right) n(n+1)}{2} + n \left(1 + \lceil \log_2(n/i) \rceil \right).$$

Rozważamy sumę

$$S(n) = \sum_{i=1}^{n} \Big(1 + \sum_{k=0}^{\lceil \log_2(\frac{n}{i}) \rceil - 1} \big(2^k \cdot i + 1 \big) \Big).$$

Niech

$$m = \left\lceil \log_2\left(\tfrac{n}{i}\right) \right\rceil.$$

Wówczas wewnętrzna suma:

$$\sum_{k=0}^{m-1} \big(2^k \cdot i + 1 \big) = \sum_{k=0}^{m-1} 2^k \cdot i + \sum_{k=0}^{m-1} 1 = i \sum_{k=0}^{m-1} 2^k + m = i \big(2^m - 1 \big) + m.$$

Zatem dla każdego i:

$$1 + \sum_{k=0}^{m-1} \left(2^k \cdot i + 1\right) = 1 + i\left(2^m - 1\right) + m.$$

Cała suma $S(n)$ staje się więc

$$S(n) = \sum_{i=1}^{n} \left(1 + i\left(2^{\lceil \log_2(\frac{n}{i}) \rceil} - 1\right) + \lceil \log_2(\tfrac{n}{i}) \rceil\right).$$

Największy wkład w tę sumę pochodzi od wyrażenia

$$i \cdot 2^{\lceil \log_2(\frac{n}{i}) \rceil}.$$

Zauważmy, że

$$\tfrac{n}{i} \;\leq\; 2^{\lceil \log_2(\frac{n}{i}) \rceil} \;\leq\; 2 \cdot \tfrac{n}{i}.$$

Stąd

$$n \;\leq\; i \cdot 2^{\lceil \log_2(\frac{n}{i}) \rceil} \;\leq\; 2n.$$

Czyli

$$n + n^2 + \sum_{i=1}^{n} \left(\lceil \log_2(n/i) \rceil\right) \leq S(n) \leq n + 2n^2 + \sum_{i=1}^{n} \left(\lceil \log_2(n/i) \rceil\right)$$

Zarówno dolne jak i górne ograniczenie ma złożoność $\Theta(n^2)$ A zatem algorytm również ma złożoność:

$$\Theta(n^2).$$

2. Szybkie potęgowanie (1 pkt)

Zapisz w pseudokodzie algorytm szybkiego potęgowania liczby x, który oblicza x^n przez wymnożenie odpowiednich potęg dwójkowych liczby x (tj. potęg postaci x^{2^k}). Zadbaj, by Twój algorytm używał stałej liczby komórek pamięci. Oszacuj jego złożoność przy kryterium jednorodnym i przy kryterium logarytmicznym.

Rozwiązanie:

```
1:  function FastPower(x, n)
2:      result ← 1
3:      power ← x
4:      while n > 0 do
5:          if n mod 2 = 1 then
6:              result ← result × power
7:          end if
8:          power ← power × power
9:          n ← ⌊n/2⌋
10:     end while
11:     return result
12: end function
```

Analiza złożoności

1. **Kryterium jednorodne** – każda operacja kosztuje 1.
2. **Kryterium logarytmiczne (bitowe)** – koszt zależy od wielkości liczb.

1. Analiza w modelu jednorodnym

W tym podejściu przyjmujemy, że:

- Każde przypisanie (np. `result <- 1`) kosztuje 1.
- Każda operacja arytmetyczna (mnożenie, dzielenie, `mod`) kosztuje 1.
- Każde porównanie (np. `n > 0`, `n mod 2 = 1`) kosztuje 1.

Koszty inicjalizacji (linijki 2–3).

- `result <- 1` : 1 operacja (przypisanie).
- `power <- x` : 1 operacja (przypisanie).

Razem: 2 operacje przed startem pętli.

Pętla `while n > 0 do` (linijka 4). Pętla działa tak długo, aż `n` będzie równe 0. W każdej iteracji `n` jest dzielone przez 2 (zaokrąglenie w dół), więc łączna liczba iteracji wynosi w przybliżeniu

$$\lfloor \log_2(n) \rfloor + 1 \quad \approx \quad \Theta(\log n).$$

Wewnątrz pętli (każda iteracja):

1. `n > 0` – sprawdzenie warunku pętli: 1 operacja (porównanie).
2. `if n mod 2 = 1 then`:
   - `n mod 2` : 1 operacja,
   - porównanie z 1 : 1 operacja,
   - `result <- result * power` (jeśli warunek spełniony):
     - 1 operacja mnożenia,
     - 1 operacja przypisania.

     (łącznie 2 operacje, ale tylko gdy `n mod 2 = 1`).
3. `power <- power * power`:
   - 1 operacja mnożenia,
   - 1 operacja przypisania.
4. `n <- floor(n/2)`:
   - 1 operacja dzielenia całkowitego,
   - 1 operacja przypisania.

Podsumowując koszty w 1 iteracji (zakładając, że warunek `n mod 2 = 1` może być spełniony):

$$\underbrace{1}_{\text{war. while}} + \underbrace{(1+1)}_{\texttt{n mod 2, porównanie}} + \underbrace{(1+1)}_{\texttt{result <- ...}} + \underbrace{(1+1)}_{\texttt{power <- ...}} + \underbrace{(1+1)}_{\texttt{n <- floor(n/2)}} = 9.$$

Można zauważyć, że jeśli `n mod 2` nie jest równe 1, to oszczędzamy 2 operacje w danej iteracji, ale w analizie $\Theta(\cdot)$ i tak przyjmujemy stałą liczbę operacji na iterację.

Łączny koszt:

$$2 \;+\; 9 \cdot \big(\lfloor \log_2(n) \rfloor + 1\big) \in \Theta(\log n).$$

2. Analiza w modelu logarytmicznym (bitowym)

W tym modelu zakładamy, że:

- Koszt operacji na liczbach całkowitych zależy od rozmiaru tych liczb (w bitach).
- Mnożenie liczb o długości ℓ bitów może mieć koszt nawet $\Theta(\ell^2)$ albo $\Theta(\ell \log \ell)$.

Liczba iteracji pętli. Niezależnie od rozmiaru liczb, pętla `while` wykonuje się $\Theta(\log n)$ razy (bo `n` jest dzielone przez 2 w każdej iteracji).

Rozmiary liczb.

- `power` rośnie wykładniczo, ponieważ w każdej iteracji jest podnoszone do kwadratu: `power` $\leftarrow$ `power`^{2}.
- `result` w niektórych iteracjach (gdy `n mod 2 = 1`) jest mnożone przez `power`.

Zatem wartości `power` i `result` mogą stać się bardzo duże, a koszt mnożenia nie jest już stały.

Oszacowanie kosztu. Jeśli przyjmiemy, że `power` (i ewentualnie `result`) może osiągnąć wartości do $\approx x^n$ (w skrajnych przypadkach), to długość w bitach jest rzędu $\log(x^n) = n \log x$. Wówczas mnożenie może kosztować co najmniej $\Theta((n \log x)^\alpha)$ dla pewnej $\alpha \geq 1$, zależnie od implementacji mnożenia.

Ponieważ takich mnożeń (w pętli) mamy $\Theta(\log n)$, to końcowa złożoność może wynieść:

$$\Theta\big(\log n \times \text{koszt mnożenia liczb wielkości } x^n\big).$$

Dokładna postać zależy od szczegółów implementacyjnych i przyjętego modelu mnożenia.

Wniosek:

- W **kryterium jednorodnym** algorytm ma złożoność $\Theta(\log n)$.
- W **kryterium logarytmicznym/bitowym** ogólny koszt jest wyższy od $\Theta(\log n)$ i może zależeć od wartości `x` i `n`. Typowo spotyka się zapisy w stylu:

  $$\Theta\big(\log n \cdot M(\text{size of } x^n)\big),$$

  gdzie $M(\cdot)$ to koszt mnożenia liczb o danej długości bitowej.

3. Rekurencja w drzewach (1 pkt)

Napisz w pseudokodzie rekurencyjne funkcje w pseudokodzie, które dla danego drzewa binarnego T obliczają:

- liczbę wierzchołków w T;
- maksymalną odległość między wierzchołkami w T.

Rozwiązanie:

Algorithm Zliczanie wierzchołków

```
1: function CountNodes(T)
2:     if T = null then
3:         return 0
4:     else
5:         return 1 + CountNodes(T.left) + CountNodes(T.right)
6:     end if
7: end function
```

Algorithm Obliczanie średnicy drzewa

```
1: function diameterHelper(root, diameter)
2:     if root = null then
3:         return 0
4:     end if
5:     leftHeight ← diameterHelper(root.left, diameter)
6:     rightHeight ← diameterHelper(root.right, diameter)
7:     diameter ← max(diameter, leftHeight + rightHeight)
8:     return 1 + max(leftHeight, rightHeight)
9: end function
10: function diameter(root)
11:     dia ← 0
12:     diameterHelper(root, dia)
13:     return dia
14: end function
```

4. Operacje na drzewie BST (1 pkt)

Napisz w pseudokodzie procedury, które dla danego drzewa binarnych przeszukiwań T:

- wstawiają zadany klucz do T;
- usuwają wierzchołek z zadanym kluczem z T;
- dla danego klucza znajdują następny co do wielkości klucz w drzewie.

Rozwiązanie:

Algorithm Wstawianie klucza do drzewa BST

```
1:  function INSERT(root, key)
2:      if root == null then
3:          return nowy węzeł(key)
4:      end if
5:      if key < root.key then
6:          root.left = Insert(root.left, key)
7:      else
8:          if key > root.key then
9:              root.right = Insert(root.right, key)
10:         end if
11:     end if
12:     return root
13: end function
```

Algorithm Znalezienie następnego klucza w BST

```
1:  function FINDSUCCESSOR(root, key)
2:      successor = null
3:      while root != null do
4:          if key < root.key then
5:              successor = root
6:              root = root.left
7:          else
8:              root = root.right
9:          end if
10:     end while
11:     return successor
12: end function
```

Algorithm Usuwanie klucza z drzewa BST

```
1:  function REMOVE(root, key)
2:      if root == null then
3:          return root
4:      end if
5:      if key < root.key then
6:          root.left = Remove(root.left, key)
7:      else
8:          if key > root.key then
9:              root.right = Remove(root.right, key)
10:         else
11:             if root.left == null then
12:                 return root.right
13:             else
14:                 if root.right == null then
15:                     return root.left
16:                 end if
17:             end if
18:             temp = MinValueNode(root.right)
19:             root.key = temp.key
20:             root.right = Remove(root.right, temp.key)
21:         end if
22:     end if
23:     return root
24: end function
```

5. Algorytm macierzowy obliczania rekurencji

Rozważmy ciąg Fibonacciego zadany rekurencją:

$$a_n = a_{n-1} + a_{n-2}.$$

Rekurencję tę można przedstawić w postaci macierzowej. Definiujemy wektor:

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix},$$

oraz macierz przejścia:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Wówczas mamy zależność:

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = A \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix}.$$

Podnosząc macierz A do potęgi $n-1$, otrzymujemy:

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = A^{n-1} \begin{bmatrix} a_1 \\ a_0 \end{bmatrix}.$$

Uogólnienie dla ciągów liniowych

Rozważmy ciąg zadany rekurencją liniową rzędu k:

$$b_n = a_1 b_{n-1} + a_2 b_{n-2} + \cdots + a_k b_{n-k}.$$

Definiujemy wektor:

$$\begin{bmatrix} b_n \\ b_{n-1} \\ \vdots \\ b_{n-k+1} \end{bmatrix}.$$

Macierz towarzysząca ma postać:

$$A_k = \begin{bmatrix} a_1 & a_2 & \dots & a_k \\ 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}.$$

Rekurencję można zapisać w postaci macierzowej:

$$\begin{bmatrix} b_n \\ b_{n-1} \\ \vdots \\ b_{n-k+1} \end{bmatrix} = A_k \begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_{n-k} \end{bmatrix}.$$

Podnosząc macierz A_k do potęgi $n-k+1$, otrzymujemy:

$$\begin{bmatrix} b_n \\ b_{n-1} \\ \vdots \\ b_{n-k+1} \end{bmatrix} = A_k^{n-k+1} \begin{bmatrix} b_k \\ b_{k-1} \\ \vdots \\ b_1 \end{bmatrix}.$$

Rozszerzenie macierzy dla rekurencji z dodatkową stałą lub wielomianem

W standardowym rozwiązaniu rekurencji liniowych (bez składników stałych) stosuje się macierz o wymiarze $k \times k$. Aby uwzględnić w równaniu dodatkowy składnik (np. stałą lub ogólnie wielomian $P(n)$), rozszerzamy wektor stanu o dodatkowe elementy.

1. Rozszerzenie wektora i macierzy:
Zamiast wektora o k elementach, rozważamy wektor o $k+1$ elementach, w którym na końcu umieszczamy liczbę 1. Odpowiadająca macierz przejścia staje się wtedy macierzą o wymiarze $(k+1) \times (k+1)$. W takiej macierzy:

- Pierwszy wiersz jest modyfikowany tak, aby uwzględniał dodatkowy składnik,
- Ostatni wiersz zawiera jedynkę, co powoduje, że przy mnożeniu macierzy przez wektor, dolny element (stała 1) pozostaje niezmieniony.

2. Uogólnienie na wielomian $P(n)$:
Jeśli mamy do czynienia z dodatkowym składnikiem w postaci wielomianu $P(n)$ o stopniu $\deg P$, możemy rozważać wektor stanu i macierz o wymiarach odpowiednio:

$$k + \deg P + 1 \quad \text{oraz} \quad (k + \deg P + 1) \times (k + \deg P + 1).$$

W wyniku działania A^n na taki wektor, na dole uzyskamy dodatkowo wartości:

$$1,\ n,\ n^2,\ \ldots,\ n^{\deg P}.$$

Pierwszy wiersz macierzy zostaje ponownie odpowiednio zmodyfikowany, a dodatkowe wyrazy, takie jak $(n+1)^l$, można wyrazić jako kombinację liniową potęg $n^l,\ n^{l-1},\ \ldots,\ n,\ 1$.

3. Metoda różniczkowania (odejmowania) rekurencji:
Alternatywnie, można postąpić następująco:

- Odejmujemy równanie rekurencyjne dla n od równania dla $n+1$.
- Definiujemy nowy ciąg:
$$b_n := a_{n+1} - a_n.$$
- Otrzymaną rekurencję dla b_n (która nie zawiera dodatkowych stałych) rozwiązujemy standardowo.
- Na końcu sumujemy ciąg b_n, aby odzyskać ciąg a_n.

Można też zapisać wielomian charakterystyczny dla nowej rekurencji na a_n jako iloczyn wielomianu dla b_n oraz czynnika $\pm(x-1)$.

4. Ogólna postać rozwiązania i metoda anihilatorów:
Rozwiązanie rekurencji dla b_n ma postać kombinacji liniowej wyrazów $n^l\lambda^n$, gdzie λ są pierwiastkami wielomianu charakterystycznego. W wyniku, ciąg a_n przyjmuje podobną postać, ale może zawierać dodatkowy składnik postaci n^l. Dodatkowe wyrazy pojawiają się wtedy, gdy w rozwiązaniu b_n występują składniki 1, n, $\ldots$, n^{l-1} (np. gdy b_n nie zawiera stałej – wtedy 1 nie jest pierwiastkiem wielomianu charakterystycznego).

Metoda ta jest przykładem stosowania anihilatorów. Wprowadzamy operator przesunięcia E, który przesuwa ciąg o jedno miejsce (czyli $Ea_n = a_{n+1}$). Szukamy wielomianu Q tak, aby ciąg a_n należał do jądra operatora $Q(E)$. Procedura przebiega w dwóch krokach:

1. Najpierw zastosujemy operator $(E-1)$ dokładnie $\deg P + 1$ razy, co redukuje problem do rekurencji bez dodatkowego składnika wielomianowego.

2. Następnie, do rozwiązania zwykłej rekurencji stosujemy wielomian charakterystyczny, podstawiając operator E w miejsce zmiennej.

Iloczyn obu wielomianów daje operator $Q(E)$. Rozkładając Q na czynniki liniowe (nad $\mathbb{C}$), otrzymujemy, że ciąg a_n jest kombinacją liniową wyrazów postaci:

$$n^l\lambda^n,$$

gdzie λ to pierwiastki Q, a l jest liczbą mniejszą od krotności danego pierwiastka.

Podsumowanie:

- Rozszerzenie wektora stanu (o dodatkowe elementy) i macierzy (do wymiaru $(k+\deg P+1)\times(k+\deg P+1)$) pozwala uwzględnić dodatkowe składniki wielomianowe w rekurencji.
- Metoda różniczkowania (odejmowania) upraszcza rekurencję przez wyeliminowanie stałych, co umożliwia jej standardowe rozwiązanie, a następnie odzyskanie ciągu a_n przez sumowanie.
- Stosując metodę anihilatorów i operator przesunięcia E, uzyskujemy ostateczną postać rozwiązania jako kombinację liniową wyrazów $n^l\lambda^n$.

Ogólny przypadek rekurencji z wielomianem

Rozważmy rekurencję

$$c_n = a_1 c_{n-1} + a_2 c_{n-2} + \cdots + a_k c_{n-k} + \Big(b_0 n^l + b_1 n^{l-1} + \cdots + b_l\Big),$$

gdzie wielomian

$$P(n) = b_0 n^l + b_1 n^{l-1} + \cdots + b_l$$

ma stopień l.

Aby sprowadzić tę rekurencję do postaci macierzowej, rozszerzamy wektor stanu do wymiaru

$$k + l + 1:$$

$$\mathbf{v}_n = \begin{pmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_{n-k+1} \\ n^l \\ n^{l-1} \\ \vdots \\ 1 \end{pmatrix}.$$

Przyjmujemy, że wektor stanu dla $n-1$ ma postać

$$\mathbf{v}_{n-1} = \begin{pmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_{n-k} \\ (n-1)^l \\ (n-1)^{l-1} \\ \vdots \\ 1 \end{pmatrix}.$$

Chcemy skonstruować macierz przejścia A tak, aby

$$\mathbf{v}_n = A\,\mathbf{v}_{n-1}.$$

Macierz A budujemy blokowo:

1. **Część dotycząca ciągu c_n:**
   Pierwszy wiersz macierzy odpowiada równaniu rekurencyjnemu

   $$c_n = a_1 c_{n-1} + a_2 c_{n-2} + \cdots + a_k c_{n-k} + P(n).$$

   Zatem pierwszy wiersz macierzy A ma postać

   $$\big[a_1,\ a_2,\ \ldots,\ a_k,\ s_0,\ s_1,\ \ldots,\ s_l\big],$$

gdzie współczynniki $s_0, s_1, \ldots, s_l$ są dobrane tak, aby (przy uwzględnieniu aktualizacji części wielomianowej, opisanej niżej)

$$s_0 (n-1)^l + s_1 (n-1)^{l-1} + \cdots + s_l = b_0 n^l + b_1 n^{l-1} + \cdots + b_l.$$

Współczynniki s_i wyznacza się poprzez rozwinięcie dwumianowe

$$n^j = ((n-1)+1)^j = \sum_{i=0}^{j} \binom{j}{i} (n-1)^i,$$

tzn. wyrażamy każdą potęgę n^j w bazie $\{(n-1)^l,\ (n-1)^{l-1}, \ldots, 1\}$.

2. **Część przesunięcia wyrazów ciągu c_n:**
Kolejne $k-1$ wierszy macierzy odpowiadają „przesunięciu" wyrazów ciągu:

$$\begin{array}{ll} \text{wiersz } 2: & (1,\ 0,\ \ldots,\ 0,\ 0, \ldots, 0), \\ \text{wiersz } 3: & (0,\ 1,\ \ldots,\ 0,\ 0, \ldots, 0), \\ & \vdots \\ \text{wiersz } k: & (0,\ 0,\ \ldots,\ 1,\ 0, \ldots, 0). \end{array}$$

3. **Część dotycząca aktualizacji wielomianu:**
Wiersze od $k+1$ do $k+l+1$ odpowiadają aktualizacji stanu wielomianowego. Skoro dla dowolnego wykładnika j mamy

$$n^j = ((n-1)+1)^j = \sum_{i=0}^{j} \binom{j}{i} (n-1)^i,$$

to macierz aktualizująca tę część – oznaczona przez B – ma postać górnotrójkątną:

$$B = \begin{pmatrix} \binom{l}{l} & \binom{l}{l-1} & \cdots & \binom{l}{0} \\ 0 & \binom{l-1}{l-1} & \cdots & \binom{l-1}{0} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Wiersze te umieszczamy w macierzy A jako blok dolny, tj. mają postać

$$(\underbrace{0, \ldots, 0}_{k \text{ zer}}, \text{wiersze macierzy } B).$$

Stąd ogólna postać macierzy przejścia A wygląda następująco:

$$A = \begin{pmatrix} a_1 & a_2 & \cdots & a_k & s_0 & s_1 & \cdots & s_l \\ 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ \multicolumn{4}{c}{\mathbf{0}} & \multicolumn{4}{c}{B} \end{pmatrix}.$$

Dla $n \geq k$ (przy ustalonych warunkach początkowych $c_0, c_1, \ldots, c_{k-1}$ oraz odpowiednich wartościach początkowych dla części wielomianowej) rozwiązanie rekurencji otrzymujemy jako

$$\mathbf{v}_n = A^{(n-k)}\,\mathbf{v}_k,$$

a wartość c_n znajduje się jako pierwszy element wektora $\mathbf{v}_n$.

Podsumowanie:

- Rozszerzamy wektor stanu do wymiaru $k+l+1$ poprzez dołączenie elementów odpowiadających potęgom n od n^l do 1.
- Pierwszy wiersz macierzy przejścia składa się z dwóch części: współczynniki rekurencji $a_1, \ldots, a_k$ oraz współczynniki $s_0, \ldots, s_l$ dobrane tak, aby po aktualizacji części wielomianowej uzyskać $P(n) = b_0 n^l + \cdots + b_l$.
- Kolejne $k-1$ wierszy służą do przesunięcia wyrazów ciągu c_n.
- Blok dolny – macierz B – aktualizuje stan wielomianowy według wzoru

$$n^j = ((n-1)+1)^j = \sum_{i=0}^{j} \binom{j}{i} (n-1)^i.$$

Powyższy zapis przedstawia ogólną metodę macierzową dla rozwiązywania rekurencji niehomogenicznych, gdzie dodatkowy składnik jest wielomianem funkcji n.

—

W praktycznych zastosowaniach współczynniki $s_0, s_1, \ldots, s_l$ wyznacza się przez przepisanie wyrażenia $P(n)$ (dla n w nowej postaci) w bazie $\{(n-1)^l, (n-1)^{l-1}, \ldots, 1\}$ przy użyciu rozwinięcia dwumianowego.

6. Sprawdzanie ścieżki (1.5 pkt)

Ułóż algorytm, który dla drzewa $T = (V, E)$ oraz listy par wierzchołków $\{v_i, u_i\}$ $(i = 1, \ldots, m)$, sprawdza, czy v_i leży na ścieżce z u_i do korzenia. Przyjmij, że drzewo zadane jest jako lista $n - 1$ krawędzi (p_i, a_i), takich, że p_i jest ojcem a_i w drzewie.

Rozwiązanie:

```cpp
#include <iostream>
using namespace std;
const int SIZE = 1e6;
int p[SIZE];

bool check(int u, int v) {
    while (u != 1) {
        if (u == v)
            return true;
        u = p[u];
    }
    return false;
}

int main() {
    int n; cin >> n;
    for (int i = 0; i < n - 1; i++) {
        int a, b; cin >> a >> b;
        p[b] = a;
    }
}
```

Ten algorytm działa w złożoności $O(n*m)$, ale można zrobić to szybciej, używając algorytmu LCA w $O(n \log n + m * \log n)$:

```cpp
#include <bits/stdc++.h>
using namespace std;

static const int MAXN = 100000;
static const int LOGN = 17;

vector<int> adj[MAXN + 1];
int parent[MAXN + 1][LOGN + 1];
int depth[MAXN + 1];

void dfs(int v, int p) {
    parent[v][0] = p;
    depth[v] = (v == p ? 0 : depth[p] + 1);
    for (int nxt : adj[v]) {
        if(nxt != p)
            dfs(nxt, v);
    }
}

void buildParent(int n) {
    for(int k = 1; k <= LOGN; k++) {
        for(int v = 1; v <= n; v++) {
            parent[v][k] = parent[ parent[v][k-1] ][k-1];
        }
```

```
    }
}

int LCA(int x, int y) {
    if(depth[x] < depth[y]) swap(x, y);
    int diff = depth[x] - depth[y];
    for(int k = 0; k <= LOGN; k++) {
        if(diff & (1 << k)) {
            x = parent[x][k];
        }
    }
    if(x == y) return x;
    for(int k = LOGN; k >= 0; k--) {
        if(parent[x][k] != parent[y][k]) {
            x = parent[x][k];
            y = parent[y][k];
        }
    }
    return parent[x][0];
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < n-1; i++){
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    dfs(1, 1);
    buildParent(n);
    while(m--){
        int v, u;
        cin >> v >> u;
        int l = LCA(v, u);
        if(l == v) cout << "TAK\n";
        else cout << "NIE\n";
    }
    return 0;
}
```

7. Liniowy koszt budowy kopca (1pkt)

Niech A będzie tablicą n elementów. Procedura `Build-Heap(`A`)`, która buduje kopiec (minimalny lub maksymalny) z elementów w A, działa w czasie $O(n)$. Dokładniej, całkowita liczba operacji wywołanych przez `Build-Heap(`A`)` jest ograniczona przez pewną stałą pomnożoną przez n.

Proof. Rozważmy standardową procedurę `Build-Heap(`A`)`, która dla tablicy $A[1..n]$ (indeksowanej od 1) przebiega w następujący sposób:

1. Dla $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 wykonaj:
   - `Heapify(`A`,` i`,` n`)`,

gdzie `Heapify(`A`,` i`,` n`)` zapewnia utrzymanie własności kopca poprzez ewentualne zepchnięcie elementu $A[i]$ w dół drzewa.

Idea dowodu Wykorzystujemy fakt, że wywołanie `Heapify` na węźle o indeksie i może spowodować przesunięcie elementu $A[i]$ o pewną liczbę poziomów w dół. Maksymalna wysokość kopca to $O(\log n)$, ale liczba węzłów, które mogą być zepchnięte o wiele poziomów, jest niewielka. Dokładne zsumowanie kosztu pozwala uzyskać wynik $O(n)$.

Analiza kosztu wywołań `Heapify` Niech $h(i)$ oznacza wysokość poddrzewa z korzeniem w węźle i. W najgorszym przypadku każde wywołanie `Heapify(`A`,` i`,` n`)` może zepchnąć element w dół o co najwyżej $h(i)$ poziomów. Jednakże:

- Węzły będące liśćmi (tj. $i > \lfloor n/2 \rfloor$) nie wymagają praktycznie żadnych przesunięć.
- Węzły w wyższych poziomach mają większy potencjał przesunięcia w dół, ale jednocześnie tych węzłów jest mniej.

Aby oszacować łączny koszt, sumujemy czas `Heapify` dla wszystkich węzłów $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1$.

Sumowanie po poziomach Wyobraźmy sobie drzewo kopca jako drzewo binarne wysokości $\lfloor \log n \rfloor$. Oznaczmy przez k poziom w drzewie (korzeń jest na poziomie 0). Liczba węzłów na poziomie k wynosi co najwyżej 2^k. Każdy taki węzeł może być zepchnięty w dół o co najwyżej $\lfloor \log n \rfloor - k$ poziomów.

Możemy zatem zapisać oszacowanie następująco:

$$T(n) \;\leq\; \sum_{k=0}^{\lfloor \log n \rfloor} \Big(2^k\Big) \cdot \Big(\lfloor \log n \rfloor - k\Big).$$

Niech $h = \lfloor \log_2 n \rfloor$. Wówczas koszt budowy kopca oszacujemy jako

$$T(n) \leq \sum_{k=0}^{h} 2^k \, (h - k).$$

Możemy zapisać tę sumę w postaci:

$$T(n) = h \sum_{k=0}^{h} 2^k - \sum_{k=0}^{h} k \, 2^k.$$

Pierwszy składnik to suma geometryczna:

$$\sum_{k=0}^{h} 2^k = 2^{h+1} - 1.$$

Drugi składnik to znana suma ważona, dla której zachodzi:

$$\sum_{k=0}^{h} k\,2^k = (h-1)2^{h+1} + 2.$$

Podstawiając, otrzymujemy:

$$T(n) = h\,(2^{h+1} - 1) - \Big[(h-1)2^{h+1} + 2\Big].$$

Rozwijając i upraszczając, mamy:

$$\begin{aligned} T(n) &= h\,2^{h+1} - h - (h-1)2^{h+1} - 2 \\ &= \Big[h\,2^{h+1} - (h-1)2^{h+1}\Big] - (h+2) \\ &= 2^{h+1} - (h+2). \end{aligned}$$

Skoro $2^{h+1} \leq 2n$ (ponieważ $2^h \leq n < 2^{h+1}$), mamy ostatecznie:

$$T(n) \leq 2n - (\lfloor \log_2 n \rfloor + 2),$$

co implikuje

$$T(n) = O(n).$$

Wniosek Całkowity koszt wywołań `Heapify` w procedurze `Build-Heap` jest zatem $O(n)$. W praktyce można wykazać (szczegółowymi rachunkami), że dokładna stała przed n w oszacowaniu jest niewielka (np. ≤ 2), co kończy dowód. □

8. Dijkstra dla wag wierzchołków (1pkt)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll INF = 1e18;
vector<pair<int, ll>> adj[1000000];
ll distt[1000000];
int backTrack[1000000];
int n, m, s, t;
vector<ll> weightt;
void dijkstra(int start) {
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll
        , int>>> pq;
    distt[start] = weightt[start];
    pq.push({distt[start], start});
    while(!pq.empty()){
        auto [currDist, u] = pq.top();
        pq.pop();
        if(currDist > distt[u]) continue;
        for(auto &edge : adj[u]){
            int v = edge.first;
            ll cost = edge.second;
            if(distt[u] + cost < distt[v]){
                distt[v] = distt[u] + cost;
                backTrack[v] = u;
                pq.push({distt[v], v});
            }
        }
    }
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cin >> n >> m >> s >> t;
    weightt.resize(n+1);
    for (int i = 1; i <= n; i++){
        cin >> weightt[i];
    }
    for (int i = 0; i < m; i++){
        int a, b;
        cin >> a >> b;
        adj[a].push_back({b, weightt[b]});
        adj[b].push_back({a, weightt[a]});
    }
    for (int i = 1; i <= n; i++){
        distt[i] = INF;
        backTrack[i] = -1;
    }
    dijkstra(s);
    if(distt[t] == INF){
        cout << -1 << "\n";
    } else {
        vector<int> path;
        for(int cur = t; cur != -1; cur = backTrack[cur])
            path.push_back(cur);
        reverse(path.begin(), path.end());
```

```
55          cout << distt[t] << "␣" << path.size() << "\n";
56          for(auto v : path)
57              cout << v << "␣";
58          cout << "\n";
59      }
60      return 0;
61  }
```

9. Dowód poprawności algorytmu Dijkstry w przypadku 1 ujemnej krawędzi przy startowym wierzchołku (1.5pkt)

Niech $G = (V, E)$ będzie grafem spełniającym następujące założenia:

1. $s \in V$ jest wierzchołkiem źródłowym.
2. Istnieje dokładnie jedna krawędź $(s, v) \in E$ o ujemnej wadze, tzn.
$$w(s, v) < 0.$$
3. Dla każdej innej krawędzi wychodzącej ze źródła s, czyli dla $(s, u) \in E$ przy $u \neq v$, zachodzi
$$w(s, u) \geq 0.$$
4. Dla każdej krawędzi $(x, y) \in E$ przy $x \neq s$ mamy:
$$w(x, y) \geq 0.$$

Inicjalizacja

Algorytm Dijkstry ustawia:

$$d(s) = 0 \quad \text{oraz} \quad d(u) = \infty \quad \text{dla każdego } u \in V,\ u \neq s.$$

Podczas procesowania krawędzi wychodzących z s:

- Dla krawędzi (s, v) o ujemnej wadze mamy:
$$d(v) = d(s) + w(s, v) = 0 + w(s, v) = w(s, v) < 0.$$
- Dla każdej innej krawędzi (s, u) przy $u \neq v$:
$$d(u) = 0 + w(s, u) \geq 0.$$

Porządkowanie w kolejce priorytetowej

Kolejka priorytetowa w algorytmie Dijkstry sortuje wierzchołki według ich bieżących wartości $d(u)$. W związku z tym wierzchołek v (dla którego $d(v) < 0$) będzie miał najmniejszą wartość i zostanie jako pierwszy wyjęty z kolejki. Oznacza to, że od razu zostanie sfinalizowany z wartością
$$d(v) = w(s, v).$$

Dalsze przetwarzanie grafu

Po przetworzeniu wierzchołków s i v, wszystkie pozostałe krawędzie w grafie mają nieujemne wagi. Zatem klasyczna poprawność algorytmu Dijkstry (dla grafów o nieujemnych wagach) zachodzi dla pozostałych wierzchołków.