Ćwiczenia z ASD

Michał Knapik

Ostatnio kompilowane: 24 czerwca 2022

Początki

Zadanie

Rozważmy następujące funkcje:

```
FUN1(n):
1.      k = 1
2.      while (k < n):
3.            k := 2*k
4.      return k

FUN2(n):
1.       k = 1
2.       while (k*k < n):
3.             k := k + 1
4.       return k
```

Operacje dominujące w `FUN1` *i* `FUN2` *to działania arytmetyczne. Oblicz dokładną złożoność obliczeniową obu funkcji.*

Zadanie

Uszereguj niemalejąco względem rzędu następujące funkcje:

1. $n!$
2. $0.001n^5 + n^3$
3. $\log(n!)$
4. $n^{1000} - n^{999}$
5. $10000n^4 + 999n^3$,
6. $(1.000001)^n$
7. $\log(\log(n))$
8. $n\log(n)$
9. n^n

Zadanie

Skrót MIPS oznacza liczbę milionów operacji na sekundę. Procesor Zilog Z80 (opracowany w 1976) może wykonać 0.5 MIPS. Procesor AMD Ryzen Threadripper 3990X (2020) może wykonać 2300000 MIPS. Przyjrzymy się, jak brutalna siła obliczeniowa ma się do dobrego wyboru algorytmów.

1. *Dobry programista opracował algorytm sortujący w czasie nlogn i uruchomił go na maszynie z procesorem Z80. Słabszy programista napisał program sortujący w czasie n^2 i uruchomił go na maszynie z procesorem AMD Ryzen Threadripper 3990X.* **Znajdź wielkość tablicy którą Z80 posortuje szybciej niż AMD Ryzen.**
2. *Z poprzedniego punktu wynika, iż tablica taka będzie bardzo duża. Czy rzeczywiście jest to sukces? Ile czasu potrwa sortowanie tej tablicy przy pomocy maszyny AMD? A ile czasu potrwałoby, gdyby zastosowano na tej maszynie algorytm sortujący o złożoności ok. nlogn?*

Zadanie

Która z poniższych funkcji ma większą złożoność obliczeniową? Oszacuj z dołu wartość (nie złożoność) obu. Zaimplementuj.

```
FUN3(n):
1.      m = 1
2.      for i = 1 to 3
3.            m := m*m
4.            for j = 1 to n
5.                m := m+j
4.      return m

FUN4(n):
1.      m = 1
2.      for i = 1 to n
3.            m := m*m
4.            for j = 1 to 3
5.                m := m+j
4.      return m
```

Algorytm ma własność **pełnej poprawności**, gdy dla wszystkich danych wejściowych:

1. jeśli dane wejściowe są poprawne (zgodne ze specyfikacją wejścia), to algorytm po zatrzymaniu się zwróci poprawny wynik (zgodny ze specyfikacją),
2. jeśli dane wejściowe są poprawne, to algorytm w końcu zatrzyma się.

Algorytm jest **częściowo poprawny**, gdy spełniony jest pierwszy z powyższych warunków.

Zadanie

Czy poniższy algorytm jest poprawny częściowo dla dowolnych danych? Dlaczego nie zależy to od jego specyfikacji? A czy jest poprawny całkowicie?

```
Alg(x):
    while (true):
        pass
```

Metoda niezmienników służy do dowodzenia poprawności programów. Niezmiennik pętli to warunek (specyfikowany formalnie lub w języku bardziej naturalnym), który jest:

- prawdziwy przed pierwszą iteracją pętli (*inicjalizowanie*);
- oraz jeśli był prawdziwy przed daną iteracją pętli, to będzie prawdziwy i po niej (*utrzymanie*).

Jeśli własność jest niezmiennikiem pętli oraz pętla zatrzyma się po skończonej liczbie iteracji, to niezmiennik jest prawdziwy po zakończeniu pętli. Oczywiście musi być on zbudowany tak zręcznie, by wynikało z tego, że badany program jest poprawny.

Zadanie (Schemat Hornera)

Celem jest efektywne obliczenie wartości $f(x) = \sum_{i=0}^{n} a_i x^i$ *na podst. wzoru:*

$$f(x) = a_0 + x(a_1 + x(a_2 + \ldots x(a_{n-1} + xa_n))).$$

Jako dane wejściowe podana jest tablica współczynników wielomianu $A[0, \ldots, n]$*. Zapisz pseudokod algorytmu i zbadaj pełną poprawność algorytmu.*

Przykładowe rozwiązanie:

```
HORNER(A,x):
1.      res := 0
2.      i := len(A) - 1
3.
4.      while (i >= 0):
5.          res += A[i] + x*res
6.          i--
7.
8.      return res
```

```
HORNER(A,x):
1.    res := 0
2.    i := len(A) - 1
3.
4.    while (i >= 0):
5.        res += A[i] + x*res
6.        i--
7.
8.    return res
```

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:
  - Jeśli przed rozpoczęciem i–tej iteracji (lin. 4) mamy $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)} \ldots$

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:
  - Jeśli przed rozpoczęciem i–tej iteracji (lin. 4) mamy $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$ . . . to wykonanie lin. 5 zmodyfikuje zawartość zmiennej `res` następująco: $res = A[i] + x * (\sum_{j=i+1}^{n} A[j]x^{j-(i+1)}) =$

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:
  - Jeśli przed rozpoczęciem i–tej iteracji (lin. 4) mamy $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$ . . . to wykonanie lin. 5 zmodyfikuje zawartość zmiennej `res` następująco: $res = A[i] + x * (\sum_{j=i+1}^{n} A[j]x^{j-(i+1)}) = A[i] + \sum_{j=i+1}^{n} A[j]x^{j-i} = \sum_{j=i}^{n} A[j]x^{j-i}$.

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A}[\texttt{j}]\texttt{x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:
  - Jeśli przed rozpoczęciem i–tej iteracji (lin. 4) mamy $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$ . . . to wykonanie lin. 5 zmodyfikuje zawartość zmiennej `res` następująco: $res = A[i] + x * (\sum_{j=i+1}^{n} A[j]x^{j-(i+1)}) = A[i] + \sum_{j=i+1}^{n} A[j]x^{j-i} = \sum_{j=i}^{n} A[j]x^{j-i}$.
  - W lin. 6 mamy $i := i - 1$, więc powyższa zależność zapisana przy użyciu nowej wartości tej zmiennej ma postać: $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$.

```
HORNER(A,x):
1.     res := 0
2.     i := len(A) - 1
3.
4.     while (i >= 0):
5.         res += A[i] + x*res
6.         i--
7.
8.     return res
```

- **Warunek stopu**: oczywiste.
- **Niezmiennik pętli:**
  - Gdy $\texttt{i} = \texttt{n}$, to $\texttt{res} = 0$.
  - W przeciwnym przypadku $\texttt{res} = \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} \texttt{A[j]x}^{\texttt{j}-(\texttt{i}+1)}$.

Sprawdzamy utrzymanie niezmiennika:

- $i = n$: oczywiste.
- $-1 \leq i < n$:
  - Jeśli przed rozpoczęciem i–tej iteracji (lin. 4) mamy $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$ . . . to wykonanie lin. 5 zmodyfikuje zawartość zmiennej `res` następująco: $res = A[i] + x * (\sum_{j=i+1}^{n} A[j]x^{j-(i+1)}) = A[i] + \sum_{j=i+1}^{n} A[j]x^{j-i} = \sum_{j=i}^{n} A[j]x^{j-i}$.
  - W lin. 6 mamy $i := i - 1$, więc powyższa zależność zapisana przy użyciu nowej wartości tej zmiennej ma postać: $res = \sum_{j=i+1}^{n} A[j]x^{j-(i+1)}$.

Kończymy, przyglądając się zawartości zmiennej *res* po wykonaniu ostatniego kroku pętli. Krok ten zaczyna się z $i = 0$ a kończy z $i = -1$. Mamy wtedy:
$res = \sum_{j=-1+1}^{n} A[j]x^{j-(-1+1)} = \sum_{j=0}^{n} A[j]x^{j}$, czyli wartość $f(x)$.

Schemat Hornera pozwala na obliczenie wartości wielomianu w czasie liniowym. Naiwne podejście do tego problemu daje algorytm działający w czasie kwadratowym. Źródłem złożoności jest tu potęgowanie, realizowane przez szereg mnożeń. Spróbujemy teraz zaprojektować algorytm pozwalający na szybkie potęgowanie.

Zadanie

Wykorzystaj przedstawienie wykładnika w postaci binarnej do zaimplementowania efektywnego potęgowania i oszacuj złożoność algorytmu. Zapisz pseudokod funkcji $\texttt{FASTPOW}(\texttt{n},\texttt{k}) = \texttt{n}^{\texttt{k}}$. *Podpowiedź: zauważ, że* $n^{11_{dec}} = n^{1011_{bin}} = n^{1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0} = n^{2^3} \cdot n^{2^1} \cdot n^{2^0}$. *Zaobserwuj zgodnie z jaką regułą jest tworzony następujący ciąg:* $n^1, (n^2)^2 = n^{2^2}, ((n^2)^2)^2 = n^{2^3}, \ldots$

Przykładowe rozwiązanie (z przeliczaniem dec-bin w locie):

```
FASTPOW(n,k):
if k == 0 return 1
if (k mod 2 == 1) res := n
   else res := 1

k = k/2
mult = n*n
while (k > 0)
     if (k mod 2 == 1) res *= mult
     k = k/2
     mult = mult*mult

return res
```

Zadanie

Wykorzystaj zależność:

$$n^k = \begin{cases} n(n^{\frac{k-1}{2}})^2 & \text{dla } k \text{ nieparzystych}, \\ (n^{\frac{k}{2}})^2 & \text{dla } k \text{ parzystych} \end{cases}$$

do zaprojektowania algorytmu efektywnego potęgowania. Użyj rekursji (metoda dziel i rządź). Przygotuj równanie rekurencyjne na pesymistyczną złożoność czasową rozwiązania i oszacuj ją.

Przykładowe rozwiązanie:

```
RPOW(n,k):
if (k == 0) return 1
if (k mod 2 == 1) return n * (rpow(n, (k-1)/2)* rpow(n, (k-1)/2))
else return (rpow(n, k/2)* rpow(n, k/2))
```

Równanie rekurencyjne dla złożoności pesymistycznej:

$$T(k) = \begin{cases} 1 + T(\frac{k-1}{2}) & \text{dla } k \text{ nieparzystych}, \\ 1 + T(\frac{k}{2}) & \text{dla } k \text{ parzystych} \end{cases}$$

Zabawa: zapisać powyższy algorytm w Scheme/LISP!

Wyszukiwanie i początki sortowania

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := \mathit{len}(\mathbf{arr}) - 1$.

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := \mathit{len}(\mathbf{arr}) - 1$.
2. **e**, o ile jest w **arr**, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (tzn. $\mathbf{e} \notin \mathbf{arr}$).

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := \mathit{len}(\mathbf{arr}) - 1$.
2. **e**, o ile jest w **arr**, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (tzn. $\mathbf{e} \notin \mathbf{arr}$).
3. $\mathbf{med} := \mathbf{arr}[\lfloor \frac{l+p}{2} \rfloor]$. Jeśli $\mathbf{med} = \mathbf{e}$, zwróć $\lfloor \frac{l+p}{2} \rfloor$.

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := \mathit{len}(\mathbf{arr}) - 1$.
2. **e**, o ile jest w **arr**, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (tzn. $\mathbf{e} \notin \mathbf{arr}$).
3. $\mathbf{med} := \mathbf{arr}[\lfloor \frac{l+p}{2} \rfloor]$. Jeśli $\mathbf{med} = \mathbf{e}$, zwróć $\lfloor \frac{l+p}{2} \rfloor$.
4. Jeżeli $\mathbf{med} < \mathbf{e}$, niech $l := \lfloor \frac{l+p}{2} \rfloor + 1$ i wróć do 2.

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := \mathit{len}(\mathbf{arr}) - 1$.
2. **e**, o ile jest w **arr**, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (tzn. $\mathbf{e} \notin \mathbf{arr}$).
3. $\mathbf{med} := \mathbf{arr}[\lfloor \frac{l+p}{2} \rfloor]$. Jeśli $\mathbf{med} = \mathbf{e}$, zwróć $\lfloor \frac{l+p}{2} \rfloor$.
4. Jeżeli $\mathbf{med} < \mathbf{e}$, niech $l := \lfloor \frac{l+p}{2} \rfloor + 1$ i wróć do 2.
5. Jeżeli $\mathbf{med} > \mathbf{e}$, niech $p := \lfloor \frac{l+p}{2} \rfloor - 1$ i wróć do 2.

Binarne wyszukiwanie **e** w uporządkowanej tablicy **arr**:

1. Inicjalizacja $l := 0, p := len(\mathbf{arr}) - 1$.
2. **e**, o ile jest w **arr**, znajduje się w przedziale indeksowanym $[l, p]$. Jeśli $p < l$, zwróć -1 (tzn. $\mathbf{e} \notin \mathbf{arr}$).
3. $\mathbf{med} := \mathbf{arr}[\lfloor \frac{l+p}{2} \rfloor]$. Jeśli $\mathbf{med} = \mathbf{e}$, zwróć $\lfloor \frac{l+p}{2} \rfloor$.
4. Jeżeli $\mathbf{med} < \mathbf{e}$, niech $l := \lfloor \frac{l+p}{2} \rfloor + 1$ i wróć do 2.
5. Jeżeli $\mathbf{med} > \mathbf{e}$, niech $p := \lfloor \frac{l+p}{2} \rfloor - 1$ i wróć do 2.

Zadanie

Wypisz elementy z którym zostanie porównany klucz **e** *przy wyszukiwaniu go w ciągu* **arr***:*

- $\mathbf{e} = 251, \mathbf{arr} =$ $[0, 1, 3, 7, 11, 210, 113, 251, 325, 412, 1213, 2351]$.
- $\mathbf{e} = 2, \mathbf{arr} =$ $[0, 1, 3, 7, 11, 210, 113, 251, 325, 412, 1213, 2351]$.

Zadanie

Implementacja z wykładu wyszukiwania binarnego nie korzysta z rekurencji. Zaprojektuj rekurencyjną wersję wyszukiwania binarnego. Jakie są zalety a jakie wady rekurencji?

Zadanie

Implementacja z wykładu wyszukiwania binarnego nie korzysta z rekurencji. Zaprojektuj rekurencyjną wersję wyszukiwania binarnego. Jakie są zalety a jakie wady rekurencji?

```
Rec-BinSearch(s,l,p,e):
    if l > p return -1
    m = floor((l+p)/2)
    if s[m] = e return e
    if s[m] < e return Rec-BinSearch(s, m+1, p)
    if s[m] > e return Rec-BinSearch(s, l, m-1)
```

Zadanie

UVA: 10611 - The Playboy Chimp. Podpowiedź: zastosuj algorytm wyszukiwania binarnego zwracający ostatnio sprawdzany indeks zamiast -1 i obejrzyj elementy sąsiadujące ze zwróconym indeksem.

Zadanie

UVA: 10611 - The Playboy Chimp. Podpowiedź: zastosuj algorytm wyszukiwania binarnego zwracający ostatnio sprawdzany indeks zamiast -1 i obejrzyj elementy sąsiadujące ze zwróconym indeksem.

Zadanie

Metoda bisekcji służy do znajdowania pierwiastków równania $f(x) = 0$. *Zakładamy, że f jest funkcją rzeczywistą, ciągłą, określoną na przedziale* $[a, b]$ *i taką, że* $f(a) \cdot f(b) < 0$ *(dlaczego to wystarczy, by istniał pierwiastek w w/w przedziale?). Zaprojektować algorytm, który dla zadanego* $\epsilon > 0$ *znajduje przedział* $[c, d] \subseteq [a, b]$ *zawierający pierwiastek f i taki, że* $(d - c) \leq \epsilon$.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.
2. **Budujemy nowe piętro.** Iteruj po **parach** sąsiednich węzłów poprzedniego piętra. (tzn. rozważamy pary węzłów zawierających elementy $(\mathbf{arr}[0], \mathbf{arr}[1])$, $(\mathbf{arr}[2], \mathbf{arr}[3])$, $\ldots$). Z każdej takiej pary $(\mathit{node}(\mathbf{arr}[i]), \mathit{node}(\mathbf{arr}[i+1]))$ wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających $\mathbf{arr}[i]$ i $\mathbf{arr}[i+1]$. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.
2. **Budujemy nowe piętro.** Iteruj po **parach** sąsiednich węzłów poprzedniego piętra. (tzn. rozważamy pary węzłów zawierających elementy (**arr**[0], **arr**[1]), (**arr**[2], **arr**[3]), ...). Z każdej takiej pary (*node*(**arr**[i]), *node*(**arr**[$i+1$])) wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających **arr**[i] i **arr**[$i+1$]. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.
3. Jeżeli najnowsze piętro ma więcej niż jeden element, wróć do punktu 2.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.
2. **Budujemy nowe piętro.** Iteruj po **parach** sąsiednich węzłów poprzedniego piętra. (tzn. rozważamy pary węzłów zawierających elementy (**arr**[0], **arr**[1]), (**arr**[2], **arr**[3]), ...). Z każdej takiej pary (*node*(**arr**[i]), *node*(**arr**[$i+1$])) wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających **arr**[i] i **arr**[$i+1$]. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.
3. Jeżeli najnowsze piętro ma więcej niż jeden element, wróć do punktu 2.
4. Jeśli ma tylko jeden element, jest to element najmniejszy **mine**. Zejdź od korzenia w dół drzewa po węzłach zawierających **mine**, oglądając elementy porównywane z **mine**. Wybieramy najmniejszy z tych elementów.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.
2. **Budujemy nowe piętro.** Iteruj po **parach** sąsiednich węzłów poprzedniego piętra. (tzn. rozważamy pary węzłów zawierających elementy $(\mathbf{arr}[0], \mathbf{arr}[1])$, $(\mathbf{arr}[2], \mathbf{arr}[3])$, $\ldots$). Z każdej takiej pary $(\mathit{node}(\mathbf{arr}[i]), \mathit{node}(\mathbf{arr}[i+1]))$ wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających $\mathbf{arr}[i]$ i $\mathbf{arr}[i+1]$. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.
3. Jeżeli najnowsze piętro ma więcej niż jeden element, wróć do punktu 2.
4. Jeśli ma tylko jeden element, jest to element najmniejszy **mine**. Zejdź od korzenia w dół drzewa po węzłach zawierających **mine**, oglądając elementy porównywane z **mine**. Wybieramy najmniejszy z tych elementów.

Zadanie

Narysuj drzewko porównywań w algorytmie turniejowym dla ciągu $\mathbf{arr} = [7, 4, 9, 1, 8, 15]$.

Algorytm turniejowy: znaleźć **drugi najmniejszy** element **e** w tablicy **arr** (bez powtórzeń). Idea działania:

1. Niech **arr** będzie najniższym poziomem drzewa binarnego.
2. **Budujemy nowe piętro.** Iteruj po **parach** sąsiednich węzłów poprzedniego piętra. (tzn. rozważamy pary węzłów zawierających elementy $(\mathbf{arr}[0], \mathbf{arr}[1])$, $(\mathbf{arr}[2], \mathbf{arr}[3]), \ldots$). Z każdej takiej pary $(\mathit{node}(\mathbf{arr}[i]), \mathit{node}(\mathbf{arr}[i+1]))$ wybierz mniejszy element i umieść go w nowym węźle. Lewa i prawa gałąź nowego węzła powinny prowadzić do, odpowiednio, węzłów zawierających $\mathbf{arr}[i]$ i $\mathbf{arr}[i+1]$. Jeśli poprzednie piętro miało nieparzystą liczbę elementów, przepisz element, który nie wziął udziału w porównaniach do nowego piętra.
3. Jeżeli najnowsze piętro ma więcej niż jeden element, wróć do punktu 2.
4. Jeśli ma tylko jeden element, jest to element najmniejszy **mine**. Zejdź od korzenia w dół drzewa po węzłach zawierających **mine**, oglądając elementy porównywane z **mine**. Wybieramy najmniejszy z tych elementów.

Zadanie

Narysuj drzewko porównywań w algorytmie turniejowym dla ciągu $\mathbf{arr} = [7, 4, 9, 1, 8, 15]$.

Pytanie: dlaczego element znaleziony w czasie schodzenia w dół drzewka jest drugim najmniejszym w ciągu?

arr: ciąg (dla uproszczenia - różnych) nieuporządkowanych liczb. ***i*–tą statystyką pozycyjna** to *i*–ty najmniejszy element **arr**, gdzie $0 \leq i < |\mathbf{arr}|$ (tradycyjnie, liczymy od 0).

arr: ciąg (dla uproszczenia - różnych) nieuporządkowanych liczb. ***i*–tą statystyką pozycyjna** to *i*–ty najmniejszy element **arr**, gdzie $0 \leq i < |\mathbf{arr}|$ (tradycyjnie, liczymy od 0).

`Hoare-Partition(arr, p, r)`:

1. zwraca indeks q taki, że $p \leq q \leq r$, oraz
2. permutuje tablicę $\mathbf{arr}[p \ldots r]$ w taki sposób, że elementy $\mathbf{arr}[p \ldots (q-1)]$ są mniejsze od elementów $\mathbf{arr}[(q+1) \ldots r]$, zaś element $\mathbf{arr}[q]$ jest na właściwym miejscu (tam, gdzie znalazłby się po posortowaniu **arr**).

`Select(arr, p, r, i)` wykorzystuje podziały Hoare'a do obliczenia *i*–tej statystyki tablicy **arr** [p. . . r].

```
Select(arr, p, r, i):
    if p = r then return arr[p]

    q = Hoare-Partition(arr, p, r)
    k = q - p + 1
    if i = k then return arr[q]
    if i < k then return Select(arr, p, q-1, i)
    else return Select(arr, q+1, r, i-k)
```

`Select(arr, p, r, i)` wykorzystuje podziały Hoare'a do obliczenia *i*–tej statystyki tablicy **arr** [p. . . r].

```
Select(arr, p, r, i):
    if p = r then return arr[p]

    q = Hoare-Partition(arr, p, r)
    k = q - p + 1
    if i = k then return arr[q]
    if i < k then return Select(arr, p, q-1, i)
    else return Select(arr, q+1, r, i-k)
```

Zadanie

Znaleźć najgorszy przypadek dla procedury `Select` *i określić pesymistyczną złożoność.*

Zadanie

Zasymuluj działanie algorytmu InsertionSort na tablicy $\textbf{arr} = [4, 1, 5, 1, 6, 9, 10]$*. Określ pesymistyczny (największa liczba porównań) i optymistyczny (najmniejsza) przypadek dla tego algorytmu.*

Zadanie

Zasymuluj działanie algorytmu InsertionSort na tablicy $\textbf{arr} = [4, 1, 5, 1, 6, 9, 10]$*. Określ pesymistyczny (największa liczba porównań) i optymistyczny (najmniejsza) przypadek dla tego algorytmu.*

Zadanie

InsertionSort ma swoją nieco bardziej praktyczną wersję - sortowanie Shella (ShellSort). Zapoznaj się z opisem tego algorytmu i przetestuj kilka wybranych sekwencji wyboru kroku.

Zadanie

Która z wersji InsertionSort jest stabilna?

```
insertionSortA(arr, len){

    for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp < arr[curr - 1])){
        arr[curr] = arr[curr - 1];
        curr--;
    }

    arr[curr] = temp;
  }
}

insertionSortB(arr, len){

    for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp <= arr[curr - 1])){
        arr[curr] = arr[curr - 1];
        curr--;
    }

    arr[curr] = temp;
  }
}
```

Sortowanie, c.d.

Sortowanie przez scalanie (MergeSort). Sortujemy tablicę **arr** od indeksu p do r (tzn. poz. $p, p+1, \ldots, r-1$). Idea:

1. Znajdź indeks środkowy **med** $= \lfloor \frac{p+r}{2} \rfloor$. Wywołaj rekurencyjnie $L = \mathit{MergeSort}(\mathbf{arr}[p \ldots \mathbf{med}])$ i $R = \mathit{MergeSort}(\mathbf{arr}[\mathbf{med} \ldots r])$.
2. *Scal* posortowane podtablice L i R:
   - **2.1** Zadeklaruj nową tablicę pomocniczą **arrh** o wielkości $r - p$.
   - **2.2** Ustaw wskaźniki i, j na początkach tablic L i R.
   - **2.3** Wybierz mniejszy ze wskazywanych elementów (lub lewy - w przypadku równości, by utrzymać stabilność) i wstaw go do tablicy **arrh**. Zwiększ wykorzystany wskaźnik.
   - **2.4** Jeśli któryś ze wskaźników dotarł do krańca tablicy, wstaw pozostałe elementy drugiej tablicy do **arrh**.
   - **2.5** Jeśli zostało coś do przepisania, wróć do (c).
   - **2.6** Przepisz **arrh** do $\mathbf{arr}[p \ldots r]$.

Zadanie

Zasymuluj na tablicy $[10, 5, 1, 6, 1, 7, 8, 12, 3, 5, 13]$ *działanie algorytmu MergeSort: narysuj drzewo wywołań i drzewo złączania.*

Zadanie

Dodatkowa złożoność pamięciowa jest główną wadą MergeSort. Można jej uniknąć poprzez wykorzystanie list. Pojawia się tu kilka niuansów - jednym z nich jest problem znalezienia środka listy jednokierunkowej. Zaprojektować algorytm rozwiązujący ten problem wykorzystujący co najwyżej dwa wskaźniki i dokonujący tylko jednego przebiegu.

Hoare-Partition v.1. (wykład)

Pierwsza wersja podziału Hoare's przesuwa wskaźniki lewy i prawy od dwóch odp. skrajnych końców tablicy ku sobie. Utrzymywany jest warunek, iż na lewo od lewego wskaźnika znajdują się elementy niewiększe od dzielącego a na prawo od prawego niemniejsze.

```
Hoare-Partition(A,p,r):

x = A[p]
i = p + 1
j = r

do
    while (i < r and A[i] <= x) i++
    while (j > i and A[i] >= x) j--
    if i < j then swap(A, i, j)
while (i < j)

if A[i] > x then
    swap(A, p, i-1)
    return i - 1
else
    swap(p, i)
    return i
```

Zadanie

Zasymuluj na tablicy $[4, 2, 9, 1, 3, 8]$ *działanie algorytmu Hoare-Partition.*

Hoare-Partition v.2.

Działanie drugiej z procedur jest odmienne. Wskaźniki lewy i prawy wędrują od początku tablicy do jej końca. Utrzymywany jest pomiędzy nimi "mur" wartości większych od dzielącej. Gdy prawy wskaźnik napotyka element niewiększy od wartości dzielącej, "podaje" go lewemu, zamieniając go na pobrany od lewego element większy od wartości dzielącej. Następnie lewy wskaźnik robi jeden krok, zaś prawy rusza w dalszą drogę, aż do dotarcia na koniec tablicy lub ponownego trafienia na element niewiększy od dzielącego.

```
Hoare-Partition-Modern(A,p,r):

x = A[p]
i = p - 1

for j = p to r - 1
    if A[j] <= x then
        i++
        swap(A, i, j)

swap(A, i, p)

return i
```

Zadanie

Zasymuluj na tablicy $[4, 2, 9, 1, 3, 8]$ *działanie algorytmu Hoare-Partition-Modern.*

Zadanie

Zaimplementuj i przetestuj dowolną wersję podziału Hoare'a.

Zadanie

Zaimplementuj i przetestuj dowolną wersję podziału Hoare'a.

Przykładowe rozwiązanie w Pythonie:

```
def swap(A, i, j):
    h = A[i]
    A[i] = A[j]
    A[j] = h

def hoare_partition(A, p, r):

    x = A[p]
    i = p - 1

    for j in range(p, r):
        if A[j] <= x:
            i += 1
            swap(A, i, j)
    swap(A, i, p)

    return i
```

Mając w dłoni algorytm podziału Hoare'a, łatwo napisać Quicksort:

```
Quicksort(a, l, r):

   if l < r:
      k = partition(a, l, r)
      quicksort(a, l, k - 1)
      quicksort(a, k + 1, r)
```

Zadanie

Zaimplementuj i przetestuj w/w.

Mając w dłoni algorytm podziału Hoare'a, łatwo napisać Quicksort:

```
Quicksort(a, l, r):

   if l < r:
      k = partition(a, l, r)
      quicksort(a, l, k - 1)
      quicksort(a, k + 1, r)
```

Zadanie

Zaimplementuj i przetestuj w/w.

Przykładowe rozwiązanie w Pythonie:

```
def quicksort_rec(A, p, r):
    if p < r:
        good = hoare_partition(A, p, r)
        quicksort_rec(A, p, (good - 1))
        quicksort_rec(A, (good+1), r)

def quicksort(A):
    quicksort_rec(A, 0, len(A))
```

Sortowanie przez zliczanie (CountingSort):

```
countSort(A) {

    // declare
    max = maxValue(A)
    counts[max + 1]
    result[len(A)]

    // init
    for (i = 0; i < len(counts); i++) counts[i] = 0

    // count
    for (i = 0; i < len(A); i++) counts[A[i]]++
    for (i = 1; i < len(counts); i++) counts[i] += counts[i - 1]

    // sort
    for (i = len(A); i >= 0; i--) result[--counts[A[i]]] = A[i]

}
```

Uwaga: na wykładzie była odrobinę inna wersja, a w CLRS jest jeszcze trochę inna.

Sortowanie przez zliczanie (CountingSort):

```
countSort(A) {

    // declare
    max = maxValue(A)
    counts[max + 1]
    result[len(A)]

    // init
    for (i = 0; i < len(counts); i++) counts[i] = 0

    // count
    for (i = 0; i < len(A); i++) counts[A[i]]++
    for (i = 1; i < len(counts); i++) counts[i] += counts[i - 1]

    // sort
    for (i = len(A); i >= 0; i--) result[--counts[A[i]]] = A[i]

}
```

Uwaga: na wykładzie była odrobinę inna wersja, a w CLRS jest jeszcze trochę inna.
Jaka jest złożoność CountingSort względem max i len(A)?

Sortowanie przez zliczanie (CountingSort):

```
countSort(A) {

    // declare
    max = maxValue(A)
    counts[max + 1]
    result[len(A)]

    // init
    for (i = 0; i < len(counts); i++) counts[i] = 0

    // count
    for (i = 0; i < len(A); i++) counts[A[i]]++
    for (i = 1; i < len(counts); i++) counts[i] += counts[i - 1]

    // sort
    for (i = len(A); i >= 0; i--) result[--counts[A[i]]] = A[i]

}
```

Uwaga: na wykładzie była odrobinę inna wersja, a w CLRS jest jeszcze trochę inna.
Jaka jest złożoność CountingSort względem max i len(A)?
Odp.: $\Theta(max + len(A))$!

Zadanie

Przedstaw kolejne etapy sortowania tablicy **arr** *= [4,5,1,3,2,5,1,5] metodą* `CountingSort` *(w wersji stabilnej). Ściślej:*

- *Przedstaw wartości tablicy* `counts` *(1) po zliczaniu elementów, (2) po podsumowaniu przyrostowym, a następnie...*
- *przedstaw wartości tablic* `counts` *i* `results` *podczas wypełniania tej ostatniej elementami.*

Zadanie

Przedstaw kolejne etapy sortowania tablicy **arr** *= [4,5,1,3,2,5,1,5] metodą* `CountingSort` *(w wersji stabilnej). Ściślej:*

- *Przedstaw wartości tablicy* `counts` *(1) po zliczaniu elementów, (2) po podsumowaniu przyrostowym, a następnie...*
- *przedstaw wartości tablic* `counts` *i* `results` *podczas wypełniania tej ostatniej elementami.*

Zadanie

Pytanie: dlaczego w sortowaniu przez zliczanie wędrujemy od prawej strony tablicy wejściowej w lewo? Czy algorytm byłby poprawny, gdybyśmy wędrowali od lewej? A stabilny?

Sortowanie pozycyjne (RadixSort):
Idea: sortujemy tablicę A liczb o długości d cyfr pozycja po pozycji, zaczynając od najmniej znaczącej.

```
input: A – tablica liczb dziesiętnych o d cyfrach każda

RadixSort(A, d):
   for i = 1 to d:
       posortuj stabilnie A w/g pozycji i
```

Sortowanie pozycyjne (RadixSort):
Idea: sortujemy tablicę A liczb o długości d cyfr pozycja po pozycji, zaczynając od najmniej znaczącej.

```
input: A - tablica liczb dziesiętnych o d cyfrach każda

RadixSort(A, d):
   for i = 1 to d:
       posortuj stabilnie A w/g pozycji i
```

Zadanie
Zilustruj działanie RadixSort na liczbach: 314, 441, 328, 410, 128, 991.

Sortowanie pozycyjne (RadixSort):
Idea: sortujemy tablicę A liczb o długości d cyfr pozycja po pozycji, zaczynając od najmniej znaczącej.

```
input: A - tablica liczb dziesiętnych o d cyfrach każda

RadixSort(A, d):
   for i = 1 to d:
       posortuj stabilnie A w/g pozycji i
```

Zadanie
Zilustruj działanie RadixSort na liczbach: 314, 441, 328, 410, 128, 991.

Zadanie
Udowodnij (indukcyjnie) poprawność RadixSort i oszacuj złożoność względem wyboru podalgorytmu sortowania (jaki warto wybrać w tym przypadku?) oraz wielkości A.

Struktury danych

Tablice dynamiczne

Przykładowa strategia działania tablic dynamicznych:

- Jeśli po wykonaniu operacji *pushBack(e)* tablica jest pełna, alokujemy w pamięci miejsce na $A \times n$ elementów (typowo $A = 2$ - zwiększamy rozmiar dwukrotnie) i przepisujemy tablicę na nowe miejsce.
- Jeśli po wykonaniu operacji *popBack(e)* tablica ma wielkość logiczną równą $B\%$ wielkości fizycznej (typowo 25% - czyli $\frac{1}{4}$ tablicy zawiera elementy), również alokujemy w pamięci miejsce na $C\%$ fizycznej (typowo $C = 50\%$) wielkości tablicy i przepisujemy tablicę na nowe miejsce.

Tablice dynamiczne

Przykładowa strategia działania tablic dynamicznych:

- Jeśli po wykonaniu operacji *pushBack(e)* tablica jest pełna, alokujemy w pamięci miejsce na $A \times n$ elementów (typowo $A = 2$ - zwiększamy rozmiar dwukrotnie) i przepisujemy tablicę na nowe miejsce.
- Jeśli po wykonaniu operacji *popBack(e)* tablica ma wielkość logiczną równą $B\%$ wielkości fizycznej (typowo 25% - czyli $\frac{1}{4}$ tablicy zawiera elementy), również alokujemy w pamięci miejsce na $C\%$ fizycznej (typowo $C = 50\%$) wielkości tablicy i przepisujemy tablicę na nowe miejsce.

Zadanie

Zasymuluj następującą kolejkę operacji, zaczynając od tablicy o wielkości 1:
$[pushBack(1), pushBack(2), popBack(), pushBack(3), pushBack(4), popBack()]$.

Tablice dynamiczne

Przykładowa strategia działania tablic dynamicznych:

- Jeśli po wykonaniu operacji *pushBack(e)* tablica jest pełna, alokujemy w pamięci miejsce na $A \times n$ elementów (typowo $A = 2$ - zwiększamy rozmiar dwukrotnie) i przepisujemy tablicę na nowe miejsce.
- Jeśli po wykonaniu operacji *popBack(e)* tablica ma wielkość logiczną równą $B\%$ wielkości fizycznej (typowo 25% - czyli $\frac{1}{4}$ tablicy zawiera elementy), również alokujemy w pamięci miejsce na $C\%$ fizycznej (typowo $C = 50\%$) wielkości tablicy i przepisujemy tablicę na nowe miejsce.

Zadanie

Zasymuluj następującą kolejkę operacji, zaczynając od tablicy o wielkości 1: $[pushBack(1), pushBack(2), popBack(), pushBack(3), pushBack(4), popBack()]$.

Zadanie

Przyjmijmy $A = 2$ *i* $B = C = 50\%$. *Innymi słowy, powiększamy tablicę dwukrotnie, gdy się wypełni i zmniejszamy o połowę gdy połowa miejsc jest wypełniona. To jest niedobra strategia i można zbudować sekwencję m operacji pushBack i popBack, której czas wykonania jest rzędu* $\Theta(m^2)$. *Zaproponuj taką sekwencję.*

Listy

Koszt czasowy dostępu do n-tego elementu jest rzędu $\Theta(n)$. Operacje *pushFront* (wstawienie elementu na początek, tj. przed głowę listy) i *popFront* (usunięcie głowy) działają w czasie $\Theta(1)$. W przypadku list jednokierunkowych operacje *pushBack* i *popBack* (wstawienie/usunięcie elementu na końcu listy) działają w czasie liniowym. Dwukierunkowe listy cykliczne: *pushBack* i *popBack* działają w czasie $\Theta(1)$.

```
SList {
      Type element;
      SList* next;
      }

DList {
      Type element;
      SList* next;
      SList* prev;
      }
```

Listy

Koszt czasowy dostępu do n-tego elementu jest rzędu $\Theta(n)$. Operacje *pushFront* (wstawienie elementu na początek, tj. przed głowę listy) i *popFront* (usunięcie głowy) działają w czasie $\Theta(1)$. W przypadku list jednokierunkowych operacje *pushBack* i *popBack* (wstawienie/usunięcie elementu na końcu listy) działają w czasie liniowym. Dwukierunkowe listy cykliczne: *pushBack* i *popBack* działają w czasie $\Theta(1)$.

```
SList {
      Type element;
      SList* next;
      }

DList {
      Type element;
      SList* next;
      SList* prev;
      }
```

Zadanie

Zaproponuj implementację (pseudokod) czterech wymienionych wyżej operacji dla list jednokierunkowych i dwukierunkowych list cyklicznych.

Listy

Koszt czasowy dostępu do n-tego elementu jest rzędu $\Theta(n)$. Operacje *pushFront* (wstawienie elementu na początek, tj. przed głowę listy) i *popFront* (usunięcie głowy) działają w czasie $\Theta(1)$. W przypadku list jednokierunkowych operacje *pushBack* i *popBack* (wstawienie/usunięcie elementu na końcu listy) działają w czasie liniowym. Dwukierunkowe listy cykliczne: *pushBack* i *popBack* działają w czasie $\Theta(1)$.

```
SList {
      Type element;
      SList* next;
      }

DList {
      Type element;
      SList* next;
      SList* prev;
      }
```

Zadanie

Zaproponuj implementację (pseudokod) czterech wymienionych wyżej operacji dla list jednokierunkowych i dwukierunkowych list cyklicznych.

Zadanie

Rozszerz implementację dla list jednokierunkowych o wskaźnik do ostatniego elementu i licznik liczby elementów.

Listy

Zadanie

Zaprojektuj algorytm, działający w czasie liniowym, usuwający z listy jednokierunkowej wszystkie pojawiające się pod rząd duplikaty. Np. lista
$1 \to 2 \to 2 \to 3 \to 4 \to 4 \to 4 \to 5$ *zostanie przekształcona na* $1 \to 2 \to 3 \to 4 \to 5$.

Listy

Zadanie

Zaprojektuj algorytm, działający w czasie liniowym, usuwający z listy jednokierunkowej wszystkie pojawiające się pod rząd duplikaty. Np. lista $1 \to 2 \to 2 \to 3 \to 4 \to 4 \to 4 \to 5$ *zostanie przekształcona na* $1 \to 2 \to 3 \to 4 \to 5$.

Zadanie

Zaprojektuj algorytm, który odwraca kierunek listy. Dozwolone jest tylko jedno przejście od początku do końca i $\Theta(1)$ *pamięci dodatkowej. Np. lista* $1 \to 2 \to 3 \to 4 \to 5$ *zostanie przebudowana na* $5 \to 4 \to 3 \to 2 \to 1$.

Kolejki: stosy

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij i zwroc element z wierzcholka stosu
      top(); //zwroc element z wierzcholka stosu bez usuwania
      empty(); //true wtw gdy stos jest pusty
};
```

Kolejki: stosy

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij i zwroc element z wierzcholka stosu
      top(); //zwroc element z wierzcholka stosu bez usuwania
      empty(); //true wtw gdy stos jest pusty
};
```

Zadanie

Zaproponuj implementację stosu przy użyciu list z dowiązaniami lub tablic.

Kolejki: stosy

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij i zwroc element z wierzcholka stosu
      top(); //zwroc element z wierzcholka stosu bez usuwania
      empty(); //true wtw gdy stos jest pusty
};
```

Zadanie

Zaproponuj implementację stosu przy użyciu list z dowiązaniami lub tablic.

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy i ew. jedną zmienną pomocniczą typu int. Na jednym znajduje się łańcuch znakowy **s**, *a drugi jest pusty. Sprawdź, czy* **s** *jest palindromem.*

Kolejki: stosy

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij i zwroc element z wierzcholka stosu
      top(); //zwroc element z wierzcholka stosu bez usuwania
      empty(); //true wtw gdy stos jest pusty
};
```

Zadanie

Zaproponuj implementację stosu przy użyciu list z dowiązaniami lub tablic.

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy i ew. jedną zmienną pomocniczą typu int. Na jednym znajduje się łańcuch znakowy **s**, *a drugi jest pusty. Sprawdź, czy* **s** *jest palindromem.*

Zadanie

Zapoznaj się z definicją Odwrotnej Notacji Polskiej (RPN). Napisz program, przyjmujący na wejściu łańcuch w RPN obsługujący podstawowe operacje arytmetyczne (40%) oraz unarne operacje takie jak silnia, funkcje trygonometryczne, logarytm naturalny (60%). Podpowiedź: wystarczy użyć jednego stosu i przeglądać wejściowy łańcuch od lewej.

Kolejki: stosy

```
Stack {
      push(element); //wstaw element na wierzcholek stosu
      pop(); //zdejmij i zwroc element z wierzcholka stosu
      top(); //zwroc element z wierzcholka stosu bez usuwania
      empty(); //true wtw gdy stos jest pusty
};
```

Zadanie
Zaproponuj implementację stosu przy użyciu list z dowiązaniami lub tablic.

Zadanie
Masz do dyspozycji tylko i wyłącznie dwa stosy i ew. jedną zmienną pomocniczą typu int. Na jednym znajduje się łańcuch znakowy **s**, *a drugi jest pusty. Sprawdź, czy* **s** *jest palindromem.*

Zadanie
Zapoznaj się z definicją Odwrotnej Notacji Polskiej (RPN). Napisz program, przyjmujący na wejściu łańcuch w RPN obsługujący podstawowe operacje arytmetyczne (40%) oraz unarne operacje takie jak silnia, funkcje trygonometryczne, logarytm naturalny (60%). Podpowiedź: wystarczy użyć jednego stosu i przeglądać wejściowy łańcuch od lewej.

Zadanie
Zaproponuj rekurencyjne rozwiązanie problemu wież Hanoi z dyskami umieszczanymi na stosach.

Kolejki: FIFO

```
Queue {
        inject(element); //wstaw element na koniec kolejki
        out(); //zdejmij element z początku kolejki
        front(); //zwroc element z początku kolejki bez usuwania
        empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Kolejki: FIFO

```
Queue {
      inject(element); //wstaw element na koniec kolejki
      out(); //zdejmij element z początku kolejki
      front(); //zwroc element z początku kolejki bez usuwania
      empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy. Zaimplementuj przy ich pomocy trzy główne operacje kolejki FIFO. Podaj złożoność czasową algorytmu.

Kolejki: FIFO

```
Queue {
      inject(element); //wstaw element na koniec kolejki
      out(); //zdejmij element z początku kolejki
      front(); //zwroc element z początku kolejki bez usuwania
      empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy. Zaimplementuj przy ich pomocy trzy główne operacje kolejki FIFO. Podaj złożoność czasową algorytmu.

Zadanie

Porównaj, krok po kroku, efekty wykonania następujących operacji przy wykorzystaniu stosu i kolejki FIFO: wstaw(1), zdejmij(), wstaw(2), wstaw(3), wstaw(4), zdejmij(), wstaw(5), zdejmij(), zdejmij().

Kolejki: FIFO

```
Queue {
      inject(element); //wstaw element na koniec kolejki
      out(); //zdejmij element z początku kolejki
      front(); //zwroc element z początku kolejki bez usuwania
      empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy. Zaimplementuj przy ich pomocy trzy główne operacje kolejki FIFO. Podaj złożoność czasową algorytmu.

Zadanie

Porównaj, krok po kroku, efekty wykonania następujących operacji przy wykorzystaniu stosu i kolejki FIFO: wstaw(1), zdejmij(), wstaw(2), wstaw(3), wstaw(4), zdejmij(), wstaw(5), zdejmij(), zdejmij().

Zadanie

Zaproponuj implementacje procedury search(e) wyszukujące element **e** *w (1) stosie (2) kolejce. Rozważ sytuacje, gdy jako struktury pomocniczej używasz tylko stosu lub tylko kolejki (wszystkie cztery możliwości).*

Kolejki: FIFO

```
Queue {
      inject(element); //wstaw element na koniec kolejki
      out(); //zdejmij element z początku kolejki
      front(); //zwroc element z początku kolejki bez usuwania
      empty(); //true wtw gdy kolejka jest pusta, opcjonalne
};
```

Zadanie

Masz do dyspozycji tylko i wyłącznie dwa stosy. Zaimplementuj przy ich pomocy trzy główne operacje kolejki FIFO. Podaj złożoność czasową algorytmu.

Zadanie

Porównaj, krok po kroku, efekty wykonania następujących operacji przy wykorzystaniu stosu i kolejki FIFO: wstaw(1), zdejmij(), wstaw(2), wstaw(3), wstaw(4), zdejmij(), wstaw(5), zdejmij(), zdejmij().

Zadanie

Zaproponuj implementacje procedury search(e) wyszukujące element **e** *w (1) stosie (2) kolejce. Rozważ sytuacje, gdy jako struktury pomocniczej używasz tylko stosu lub tylko kolejki (wszystkie cztery możliwości).*

Zadanie

Zaproponuj implementację tablicową kolejek (taka kolejka ma pewną wielkość maksymalną). Czy łatwo byłoby zmodyfikować ją tak, by uzyskać wersję dwukierunkową kolejki (tzn. uzupełnić o możliwość dodawania elementów na początku kolejki i zdejmowania elementów z końca). Wykonaj to samo zadanie dla implementacji listowej.

Kolejki priorytetowe

Przypomnienie: drzewa binarne

- Definicja drzewa binarnego.
- **Pytanie:** ile węzłow znajduje się na i-tym poziomie pełnego drzewa binarnego?
- **Pytanie:** ile węzłów znajduje się w pełnym drzewie o wysokości h?
- **Pytanie:** jakiej minimalnej wysokości musi być drzewo binarne, by pomieścić k elementów? (Podać dokładną wartość i asymptotyczny rząd wielkości.)

Kopce

- Kopiec to prawie pełne drzewo binarne, którego ścieżki tworzą niemalejące ciągi elementów.
- **Pytanie:** ile jest kopców o elementach 1,2,3,4,5?
- **Pytanie:** gdzie można się spodziewać drugiego najmniejszego elementu w kopcu? Gdzie może znaleźć się trzeci? A do jakiej głębokości można spodziewać się n-tego elementu?
- Implementacja tablicowa kopca, przypomnienie:
  - elementy złożone są do tablicy poziomami, idąc od korzenia i od lewej w prawo;
  - indeksujemy elementy tablicy od 1 (ew. ignorując indeks 0);
  - $parent(i) = \lfloor i/2 \rfloor$, $left(i) = 2 \cdot i$, $right(i) = 2 \cdot i + 1$.
- **Zadanie:** Zapisz w tablicy zadany w postaci graficznej kopiec. Zapisz w postaci graficznej kopiec podany w tablicy.

Kopce: downheap

- Downheap: odszukujemy indeks *min* wskazujący na najmniejszą wartość z trójki $i, left(i), right(i)$ i dokonujemy ew. zamiany wartości pod *min* i i, po czym wywołujemy downheap rekurencyjnie na *min*. Złożoność: $O(logn)$.
- **Pytanie:** dlaczego dokonujemy zamiany z mniejszym z dwójki dzieci?
- **Pytanie:** Jaka jest złożoność budowy kopca poprzez wstawianie n elementów jeden po drugim do początkowo pustego kopca?
- Przypomnienie operacji construct: wywołujemy downheap na indeksach od $n/2$ do 1. Złożoność: $O(n)$.
- **Zadanie:** Wykonaj operację construct na przykładowej tablicy elementów (np. [3,5,7,4,3,6,3,2,4]). Zapisz pośrednie wartości tablicy.
- **Pytanie:** Czy posortowana rosnąco tablica jest kopcem?

Kopce: delmin

- Delmin: usuwamy korzeń i wstawiamy w jego miejsce element kończący najniższy poziom a następnie wykonujemy na korzeniu downheap. Złożoność: jak downheap.
- **Zadanie:** Wykonaj na kopcu z poprzedniego zadania operację delmin aż do pełnego jego opróżnienia.
- Uwaga: w/w schemat: construct + delmin until empty jest realizacją sortowania przez kopcowanie (HeapSort). Jaka jest jego złożoność pesymistyczna?
- **Zadanie:** Jak usunąć dowolny (zadany poprzez indeks) element z kopca (operacja delete)? Jaka jest złożoność tej operacji? Zaprojektuj algorytm i uzasadnij jego poprawność. Podpowiedź: jak w przypadku usuwania korzenia, ale korzeniem będzie usunięty element.
- **Zadanie:** Zaprojektuj algorytm IncreaseKey(i), zwiększający wartość w zadanym węźle. Podpowiedź: po prostu użyj downheap.

Kopce: upheap i insert

- Przypomnienie działania upheap: porównujemy wartość węzła z rodzicem; jeśli zaburzony jest porządek, to zamieniamy wartości miejscami i wywołujemy upheap rekurencyjnie na rodzicu. Złożoność: $O(\mathit{logn})$.
- **Pytanie:** Często unikamy rekurencji ze względów pamięciowych (rosnący stos wywołań). Czy w przypadku kopca jest to duży problem? (Java bez problemu zniesie 5000 wywołań rekurencyjnych; Python powinien sobie poradzić do przynajmniej 900).
- Przypomnienie operacji insert: wstawiamy nowy element na pierwsze wolne miejsce na najniższym poziomie (pamiętamy o warunku prawie pełnego drzewa binarnego) i wywołujemy na nim procedurę upheap. Złożoność: jak upheap.
- **Zadanie:** Wstaw do początkowo pustego kopca elementy 3,5,7,4,3,6,3,2,4. Porównaj wynik z kopcem poprzednio zbudowanym przy użyciu operacji construct.

Kopce: zadania różne

- Otrzymujesz min-kopiec H zawierający liczby dodatnie. Korzystając tylko z operacji kopców wypisz elementy H w kolejności malejącej. Możesz modyfikować H.

Kopce: zadania różne

- Otrzymujesz min-kopiec H zawierający liczby dodatnie. Korzystając tylko z operacji kopców wypisz elementy H w kolejności malejącej. Możesz modyfikować H.
  Rozwiązanie: wystarczy zdejmować najmniejszy element z i wstawiać z powrotem jego odwrotność.

Kopce: zadania różne

- Otrzymujesz min-kopiec H zawierający liczby dodatnie. Korzystając tylko z operacji kopców wypisz elementy H w kolejności malejącej. Możesz modyfikować H. Rozwiązanie: wystarczy zdejmować najmniejszy element z i wstawiać z powrotem jego odwrotność.
- Wylosuj w sposób całkowicie jednorodny (tzn. każdy z węzłów powinien mieć taką samą szansę) węzeł z pełnego drzewa binarnego o wysokości h. Drzewo zadane jest w zapisie dowiązaniowym.

Kopce: zadania różne

- Otrzymujesz min-kopiec H zawierający liczby dodatnie. Korzystając tylko z operacji kopców wypisz elementy H w kolejności malejącej. Możesz modyfikować H.
  Rozwiązanie: wystarczy zdejmować najmniejszy element z i wstawiać z powrotem jego odwrotność.
- Wylosuj w sposób całkowicie jednorodny (tzn. każdy z węzłów powinien mieć taką samą szansę) węzeł z pełnego drzewa binarnego o wysokości h. Drzewo zadane jest w zapisie dowiązaniowym.
  Rozwiązanie: wystarczy wylosować liczbę z przedziału $[0, 2^{h+1})$ i potraktować jej binarny zapis jako opis ścieżki do węzła.

Słowniki

Słowniki pozwalają na przechowywanie par (k, v) złożonych z klucza k i wartości v. W ogólności, z danym kluczem może być powiązanych kilka wartości. Obsługiwane operacje to:

- *insert*(k, v): wstawienie elementu v pod kluczem k.
- *delete*(k, v): usunięcie elementu v związanego z kluczem k, jeśli taki zestaw istnieje. (W innej wersji ta operacja przyjmuje wskaźnik do obiektu do usunięcia jako argument.)
- *search*(k): zwraca zbiór elementów o kluczu k (często zakładamy, że klucze są unikalne).

Słowniki: adresowanie bezpośrednie

W adresowaniu bezpośrednim z każdym kluczem ze zbioru $0, \ldots, m$ związana jest dokładnie jedena wartość. Wartości przechowywane są w tablicy $tab[0 \ldots m]$. Wszystkie operacje działają w czasie stałym, ale struktura taka jest nieefektywna pamięciowo, bo zwykle większość kluczy nie jest używana (potrzebna pamięć to $|\mathcal{U}|$).

Zadanie

W tablicy $tab[0 \ldots m]$ *znajduje się m różnych elementów. Napisz algorytm wypisujący wszystkie ich podzbiory. Zilustruj jego działanie, pokazując dla tablicy* $tab = [a, b, c, d]$*, jakie zbiory będą po kolei wypisane.*

Słowniki: adresowanie bezpośrednie

W adresowaniu bezpośrednim z każdym kluczem ze zbioru $0, \ldots, m$ związana jest dokładnie jedena wartość. Wartości przechowywane są w tablicy $tab[0 \ldots m]$. Wszystkie operacje działają w czasie stałym, ale struktura taka jest nieefektywna pamięciowo, bo zwykle większość kluczy nie jest używana (potrzebna pamięć to $|\mathcal{U}|$).

Zadanie

W tablicy $tab[0 \ldots m]$ *znajduje się m różnych elementów. Napisz algorytm wypisujący wszystkie ich podzbiory. Zilustruj jego działanie, pokazując dla tablicy* $tab = [a, b, c, d]$*, jakie zbiory będą po kolei wypisane. Podpowiedź: skorzystaj z zapisu binarnego liczb od 0 do* $2^{m+1} - 1$*.*

Słowniki: funkcje mieszające, metoda łańcuchowa

Idea: chcemy wstawić do tablicy $tab[0, \ldots, m]$ elementy identyfikowane kluczami z (zazwyczaj dużego) uniwersum $\mathcal{U}$. Aby uniknąć narzutu pamięciowego adresowania bezpośredniego korzystamy z funkcji $h\colon \mathcal{U} \to [0, \ldots, m]$ przyporządkowującej kluczom pozycje w tablicy. Z racji wielkości uniwersum kolizje są niemal nieuniknione - tzn. dwa różne klucze zostaną przyporządkowane tej samej pozycji. Radzimy sobie z tym na kilka sposobów.

W **metodzie łańcuchowej** tablica *tab* zawiera dynamiczne struktury danych (np. kolejki), mogące pomieścić kolidujące elementy. Czasem nazywamy te struktury wiaderkami.

Zadanie

Rozważmy f. mieszającą $h_5(x) = x \mod 7$ *dla tablicy* $tab[0, \ldots, 6]$. *Załóżmy, że dla każdego i typ* $tab[i]$ *to (a) stos, (b) lista dwukierunkowa. Zasymuluj wstawienie do tablicy elementów* $(1, a), (8, b), (3, c), (6, d), (4, e), (10, f), (11, g)$. *Następnie zasymuluj wyszukanie elementów o kluczach* 8 *i* 13.

Słowniki: funkcje mieszające, metoda łańcuchowa, c.d.

Dobra praktyka: wartość m w funkcji $h_m(x) = x \mod m$ powinna być liczbą pierwszą możliwie odległą od potęg dwójki.

Przy założeniu, że dane wejściowe rozmieszczane są przez funkcję mieszającą w taki sposób, że n–ty element może pojawić się z takim samym prawdopodobieństwem na dowolnej z pozycji, średnia długość listy w tablicy $tab[0, \ldots, m]$ wynosi $\alpha = \frac{n}{m}$. Zatem wszystkie operacje słownika wymagają $\Theta(1)$ czasu na obliczenie $h_m(k)$ plus $\Theta(\frac{n}{m})$ na przeszukanie odpowiedniej listy.

Zadanie

Dobrą praktyką przy założeniu, że wstawiane dane są losowe jest utrzymywanie współczynnika obciążenia $\alpha = \frac{\textit{liczba wstawionych elementów}}{\textit{pojemność tablicy}}$ *poniżej* 0.75. *Zaprojektuj zarys implementacji metody insert, który będzie spełniał w/w wymagania.*

Słowniki: funkcje mieszające, metoda łańcuchowa, c.d.

Dobra praktyka: wartość m w funkcji $h_m(x) = x \mod m$ powinna być liczbą pierwszą możliwie odległą od potęg dwójki.

Przy założeniu, że dane wejściowe rozmieszczane są przez funkcję mieszającą w taki sposób, że n–ty element może pojawić się z takim samym prawdopodobieństwem na dowolnej z pozycji, średnia długość listy w tablicy $tab[0, \ldots, m]$ wynosi $\alpha = \frac{n}{m}$. Zatem wszystkie operacje słownika wymagają $\Theta(1)$ czasu na obliczenie $h_m(k)$ plus $\Theta(\frac{n}{m})$ na przeszukanie odpowiedniej listy.

Zadanie

Dobrą praktyką przy założeniu, że wstawiane dane są losowe jest utrzymywanie współczynnika obciążenia $\alpha = \frac{\textit{liczba wstawionych elementów}}{\textit{pojemność tablicy}}$ *poniżej* 0.75. *Zaprojektuj zarys implementacji metody insert, który będzie spełniał w/w wymagania. Podpowiedź: zastosuj podejście zbliżone do tablic dynamicznych.*

Słowniki: metoda mieszania wielokrotnego

W tej metodzie tablica $tab[0, \ldots, m]$ bezpośrednio przechowuje pary (k, v). Indeks do wstawienia elementu (k, v) obliczany jest przy użyciu *funkcji próbkującej* $h(k, i)$. Idea jest taka, że wstawiając element (k, v) do tablicy badamy po kolei jej zawartość pod indeksami $h(k, 0), h(k, 1), h(k, 2), \ldots$ i wstawiamy (k, v) w pierwsze ze znalezionych wolnych miejsc:

```
OPENHASH-INSERT(tab, (k,v)):

    for i = 0 to len(tab):
       index = h(k,i)
       if (tab[index] is NIL) then:
          tab[index] = (k,v)
          return index

    return -1 //table full
```

Zadanie

Jest wielu kandydatów na funkcje $h(k, \cdot)$ *(patrz CLRS). Rozważ funkcję próbkującą* $h(k, i) = (k + 3 \cdot i + 5 \cdot i^2) \mod 7$ *i wstaw do tablicy* $tab[0, \ldots 7]$ *elementy o kluczach* $5, 6, 11, 9, 2, 7$.

Binarne drzewa wyszukiwań

BST: połączenie funkcjonalności słowników i kolejek priorytetowych. Binarne drzewa, których węzły zawierają klucze oraz powiązane z nimi wartości.

Klucze rozmieszczone są w węzłach tak by spełnić warunek:

- węzły w poddrzewie o korzeniu *n.left* zawierają klucze nie większe od *n.key*;
- węzły w poddrzewie o korzeniu *n.right* zawierają klucze nie mniejsze od *n.key*,

dla każdego węzła *n*.

Operacje BST:

- *insert*(n), *delete*(n), *search*(k): normalne operacje słowników. Dla uproszczenia zakładamy, że *n* to struktura danych zawierająca pola *n.key*, *n.val*.
- *min*(), *max*(): zwrócenie najmniejszej i największej wartości w drzewie;
- *succ*(n): zwraca najmniejszy klucz spośród obecnych w drzewie i większych od *n.key*;
- *pred*(n): zwraca największy klucz spośród obecnych w drzewie i mniejszych od *n.key*.

Binarne drzewa wyszukiwań

BST: połączenie funkcjonalności słowników i kolejek priorytetowych. Binarne drzewa, których węzły zawierają klucze oraz powiązane z nimi wartości.

Klucze rozmieszczone są w węzłach tak by spełnić warunek:

- węzły w poddrzewie o korzeniu *n.left* zawierają klucze nie większe od *n.key*;
- węzły w poddrzewie o korzeniu *n.right* zawierają klucze nie mniejsze od *n.key*,

dla każdego węzła *n*.

Operacje BST:

- *insert*(n), *delete*(n), *search*(k): normalne operacje słowników. Dla uproszczenia zakładamy, że *n* to struktura danych zawierająca pola *n.key*, *n.val*.
- *min*(), *max*(): zwrócenie najmniejszej i największej wartości w drzewie;
- *succ*(n): zwraca najmniejszy klucz spośród obecnych w drzewie i większych od *n.key*;
- *pred*(n): zwraca największy klucz spośród obecnych w drzewie i mniejszych od *n.key*.

Zadanie

W jakiej kolejności wypisze węzły drzewa BST następujący program?

```
INORDER-WALK(n):

   if x != NIL then:
      INORDER-WALK(n.left)
      print(n.key)
      INORDER-WALK(n.right)
```

Binarne drzewa wyszukiwań

BST: połączenie funkcjonalności słowników i kolejek priorytetowych. Binarne drzewa, których węzły zawierają klucze oraz powiązane z nimi wartości.

Klucze rozmieszczone są w węzłach tak by spełnić warunek:

- węzły w poddrzewie o korzeniu *n.left* zawierają klucze nie większe od *n.key*;
- węzły w poddrzewie o korzeniu *n.right* zawierają klucze nie mniejsze od *n.key*,

dla każdego węzła *n*.

Operacje BST:

- *insert*(n), *delete*(n), *search*(k): normalne operacje słowników. Dla uproszczenia zakładamy, że *n* to struktura danych zawierająca pola *n.key*, *n.val*.
- *min*(), *max*(): zwrócenie najmniejszej i największej wartości w drzewie;
- *succ*(n): zwraca najmniejszy klucz spośród obecnych w drzewie i większych od *n.key*;
- *pred*(n): zwraca największy klucz spośród obecnych w drzewie i mniejszych od *n.key*.

Zadanie

W jakiej kolejności wypisze węzły drzewa BST następujący program?

```
INORDER-WALK(n):

   if x != NIL then:
      INORDER-WALK(n.left)
      print(n.key)
      INORDER-WALK(n.right)
```

Jak go przerobić, żeby wypisał węzły posortowane malejąco?

Binarne drzewa wyszukiwań: min, max, i succ

Przyjrzyjmy się zarysowi implementacji każdej z operacji BST:

- *min*(): przesuwaj wskaźnik *c* po drzewie, startując od korzenia i przestawiając go sukcesywnie na jego lewego syna - dopóki ten syn istnieje.
- *max*(): przesuwaj wskaźnik *c* po drzewie, startując od korzenia i przestawiając go sukcesywnie na jego prawego syna - dopóki ten syn istnieje.
- *succ*(n): jeśli *n.right* istnieje, zwróć *min*(*n.right*). Jeśli nie - sukcesywnie przesuwamy wskaźnik *c* po drzewie, startując od *n* i ustawiając go na jego własnego ojca. Powtarzamy przesuwanie, gdy poprzedni ruch był w lewo, kończymy gdy doszliśmy do NIL (brak sukcesora) lub wykonaliśmy ruch w prawo (zwracamy *c.key*).

```
BST-SUCCESSOR(n):

    if x.right != NIL then:
       return min(x.right)

    y = y.parent
    while y != NIL and x == y.right:
       x := y
       y := y.parent

    return y
```

- **Zadanie:** napisać *pred*(n).

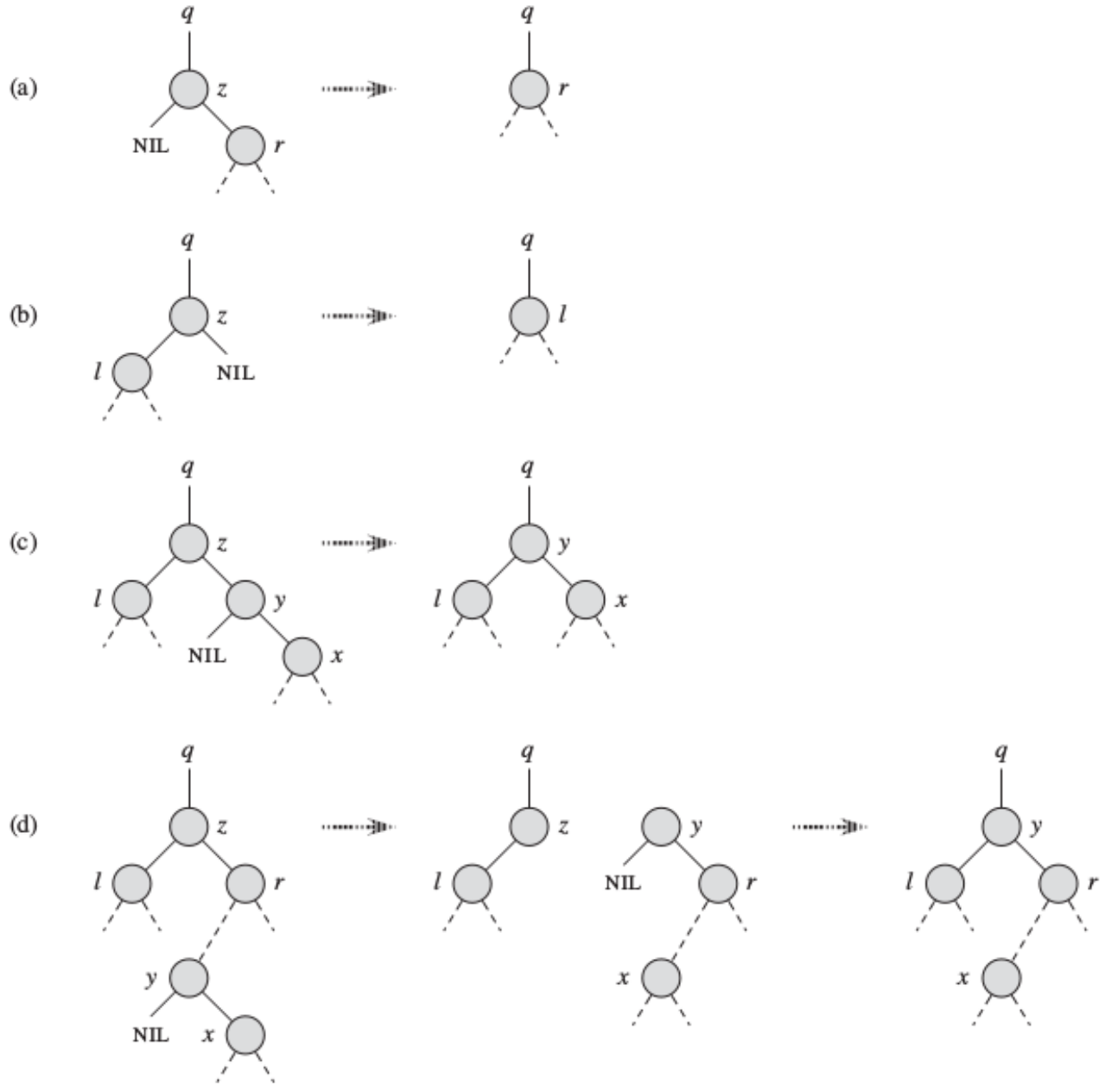
Binarne drzewa wyszukiwań: search i insert

- *search*(k): jeśli bieżący węzeł *n* (zaczynamy od korzenia) zawiera klucz *k*, to zwracamy go. Jeśli $n.key < k$, to kontynuujemy poszukiwanie w poddrzewie *n.right*. W przeciwnym przypadku poszukujemy w poddrzewie *n.left*. Jeśli w/w wybór poddrzewa jest niemożliwy, to klucz nie jest obecny w drzewie.
- *insert*(n): wykonujemy wyszukiwanie binarne (ignorując sytuację, gdy badany wierzchołek ma wartość równą *n.key*), dopóki nie znajdziemy węzła NIL. Wstawiamy *n* w miejsce tego węzła.

```
BST-INSERT(z)
    y = NIL
    x = root

    while x != NIL:
        y = x
        if z.key < x.key then:
           x = x.left
        else x = x.right

    z.p = y

    if y == NIL then:
       root = z
    elseif z.key < y.key then:
       y.left = z
    else y.right = z
```

Binarne drzewa wyszukiwań: delete

- *delete*(z): najbardziej złożony przypadek. Z CLRS:



Binarne drzewa wyszukiwań: zadania

Zadanie

Zaproponuj pseudokod procedury `BST-CUT(T,a,b)`*, która przycina drzewo T o unikalnych kluczach w taki sposób, że pozostają w nim tylko elementy o kluczach* $a \leq k \leq b$*. Podpowiedź: zacznij od przypadku, gdy korzeń* $a \leq root.key \leq b$*.*

Algorytmy grafowe

Rozgrzewka z drzewami

Zadanie

Drzewo binarne T dane jest w postaci dowiązaniowej. Zaproponuj rekurencyjną implementację funkcji HEIGHT(T), obliczającej wysokość drzewa. Jak może wyglądać wersja iteracyjna?

Rozgrzewka z drzewami

Zadanie

Drzewo binarne T dane jest w postaci dowiązaniowej. Zaproponuj rekurencyjną implementację funkcji HEIGHT(T), obliczającej wysokość drzewa. Jak może wyglądać wersja iteracyjna?

Zadanie

Drzewo binarne T dane jest w postaci dowiązaniowej. Zaproponuj rekurencyjną i iteracyjną implementację funkcji BINS(T), obliczającej liczbę węzłów, które mają dwójkę dzieci (tzn. żadne z dzieci nie jest null).

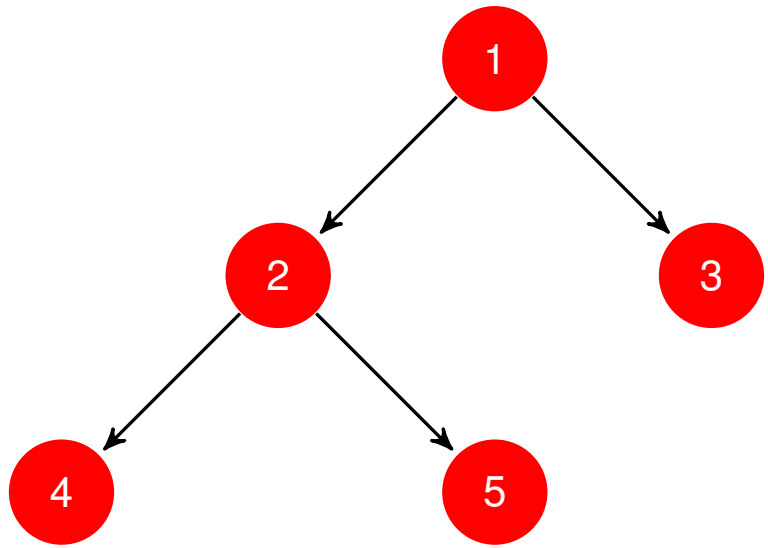
O porządkach w drzewach

Visit(BinaryTree t)

- print(t.element) //preorder
- Visit(t.left)

- Visit(t.right)

Zadanie



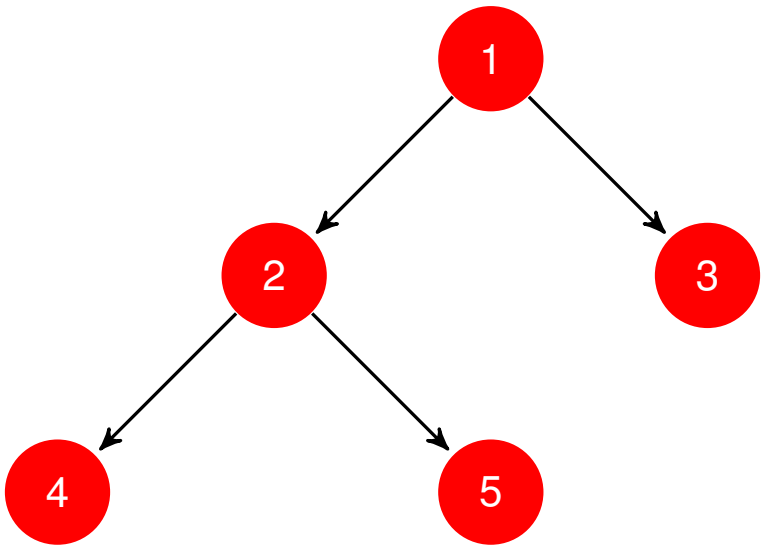
Wypisz elementy drzewa w kolejności pre/in/postorder.

O porządkach w drzewach

Visit(BinaryTree t)

- Visit(t.left)
- print(t.element) //inorder
- Visit(t.right)

Zadanie



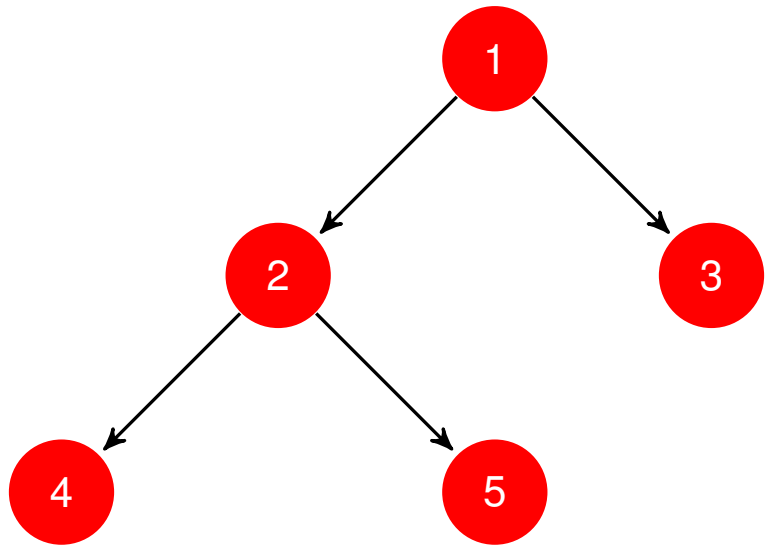
Wypisz elementy drzewa w kolejności pre/in/postorder.

O porządkach w drzewach

Visit(BinaryTree t)

- Visit(t.left)
- Visit(t.right)
- print(t.element) //postorder

Zadanie



Wypisz elementy drzewa w kolejności pre/in/postorder.

O porządkach w drzewach, c.d.

Zadanie

Zbuduj drzewo, którego wierzchołki wypisane w porządku inorder tworzą ciąg 11, 8, 7, 9, 4, 3, 5 zaś w porządku preorder tworzą ciąg 4, 7, 8, 11, 9, 5, 3.

O porządkach w drzewach, c.d.

Zadanie

Zbuduj drzewo, którego wierzchołki wypisane w porządku inorder tworzą ciąg 11, 8, 7, 9, 4, 3, 5 zaś w porządku preorder tworzą ciąg 4, 7, 8, 11, 9, 5, 3.

Zadanie

Zbuduj drzewo, którego wierzchołki wypisane w porządku inorder tworzą ciąg 7, 3, 11, 9, 8, 10 zaś w porządku postorder tworzą ciąg 7, 11, 9, 10, 8, 3.

O porządkach w drzewach, c.d.

Zadanie

Zbuduj drzewo, którego wierzchołki wypisane w porządku inorder tworzą ciąg 11, 8, 7, 9, 4, 3, 5 zaś w porządku preorder tworzą ciąg 4, 7, 8, 11, 9, 5, 3.

Zadanie

Zbuduj drzewo, którego wierzchołki wypisane w porządku inorder tworzą ciąg 7, 3, 11, 9, 8, 10 zaś w porządku postorder tworzą ciąg 7, 11, 9, 10, 8, 3.

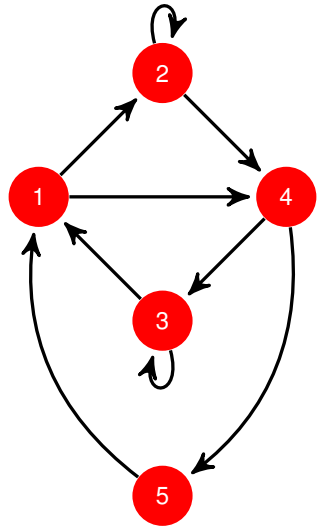
Zadanie

Para porządków (inorder, postorder) lub (inorder, preorder) jednoznacznie określa drzewo. Czy jest tak w przypadku pary (preorder, postorder)?

O reprezentacjach grafów

W większości przypadków będziemy reprezentować grafy w postaci list sąsiedztwa, ale warto wiedzieć coś i o macierzach sąsiedztwa.

Zadanie



Zbuduj reprezentację grafu w postaci (1) listy sąsiedztwa; (2) macierzy sąsiedztwa.

O reprezentacjach, c.d.

Zadanie

Niech $G = (V, E)$ będzie grafem zadanym w postaci (1) listy sąsiedztwa; (2) macierzy sąsiedztwa. Niech $G^T = (V, E^T)$ będzie grafem powstałym poprzez odwrócenie strzałek w G, tzn. $(u, v) \in E^T$ wtw $(v, u) \in E$. Jak obliczyć odpowiednie reprezentacje G^T w obu przypadkach? Czy dla przypadku (2) można efektywnie uzyskać nową reprezentację w czasie stałym? Jak?

O reprezentacjach, c.d.

Zadanie

Niech $G = (V, E)$ będzie grafem zadanym w postaci (1) listy sąsiedztwa; (2) macierzy sąsiedztwa. Niech $G^T = (V, E^T)$ będzie grafem powstałym poprzez odwrócenie strzałek w G, tzn. $(u, v) \in E^T$ wtw $(v, u) \in E$. Jak obliczyć odpowiednie reprezentacje G^T w obu przypadkach? Czy dla przypadku (2) można efektywnie uzyskać nową reprezentację w czasie stałym? Jak?

Zadanie

Zlewisko w grafie skierowanym $G = (V, E)$ to taki wierzchołek z którego nie wychodzi żadna krawędź (tzn. stopień wyjściowy jest równy 0) a do którego wchodzą krawędzie ze wszystkich pozostałych (tzn. stopień wejściowy jest równy $|V| - 1$). Zaproponuj i zaimplementuj algorytm działający w czasie $O(|V|)$, który sprawdza, czy G zawiera zlewisko. **Graf zadany jest w postaci macierzy sąsiedztwa!**

BFS

```
BFS(G, s):
1.  //Inicjalizacja
2.  for kazdy wierzcholek u in V[G] \ {s}
3.      color[u] := BIALY
4.      d[u] := INFTY
5.      pi[u] := NULL

6.  color[s] := SZARY
7.  d[s] := 0
8.  pi[s] := NULL
9.  Q = new Q
10. Q.ENQUEUE(s)

11. while Q is not empty:
12.     u := Q.DEQUEUE()
13.     for kazdy v in Adj[u]
14.         if color[v] = BIALY then
15.             color[v] := SZARY
16.             d[v] := d[u] + 1
17.             pi[v] := u
18.             Q.ENQUEUE(v)
19.     color[u] := CZARNY
```

BFS, c.d.

Zadanie

(Cormen) W zapasach amerykańskich zapaśnicy dzielą się na "dobrych" i "złych". Zapaśnicy walczą parami. Załóżmy, że mamy n zapaśników i listę r pojedynków (par zawodników). Zaprojektuj algorytm działający w czasie $O(n+r)$, który stwierdzi, czy zapaśników można podzielić na "dobrych" i "złych" w taki sposób, żeby w każdym pojedynku "dobry" zapaśnik walczył ze "złym". Jeśli taki podział jest możliwy, to Twój algorytm powinien go podać.

BFS, c.d.

Zadanie

(Cormen) W zapasach amerykańskich zapaśnicy dzielą się na "dobrych" i "złych". Zapaśnicy walczą parami. Załóżmy, że mamy n zapaśników i listę r pojedynków (par zawodników). Zaprojektuj algorytm działający w czasie $O(n+r)$, który stwierdzi, czy zapaśników można podzielić na "dobrych" i "złych" w taki sposób, żeby w każdym pojedynku "dobry" zapaśnik walczył ze "złym". Jeśli taki podział jest możliwy, to Twój algorytm powinien go podać.

Zadanie

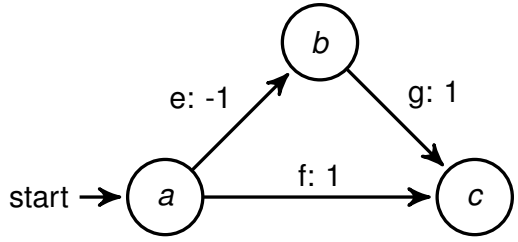
Udowodnij, że graf spójny jest drzewem wtedy i tylko wtedy, gdy różnica liczby wierzchołków i liczby krawędzi wynosi 1. Podpowiedź: indukcja + drzewo rozpinające (o których będzie później, ale coś już można powiedzieć).

DFS

```
DFS(G)
1. for kazdy wierzcholek u in V[G]
2.     color[u] = BIALY
3.     pi[u] = NULL
4.     d[u] := INFTY
5. time = 0

6. for kazdy wierzcholek u in V[G]
7.     if color[u] = BIALY then DFS-VISIT(u)


DFS-VISIT(u)
1. color[u] = SZARY
2. ++time
3. d[u] = time
4. for kazdy v in Adj[u]
5.     if color[v] = BIALY
6.         pi[v] = u
7.         DFS-VISIT(v)
8. color[u] = CZARNY
9. f[u] = ++time
```

Najkrótsze ścieżki: relaksacje

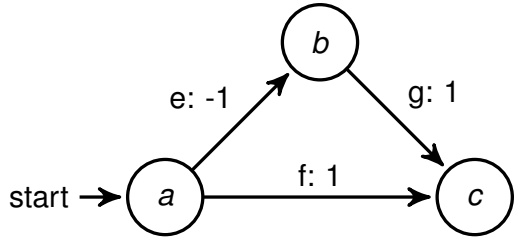
Idea: każdy z węzłów zapamiętuje poprzednik π w budowanym drzewie najkrotszych znanych ścieżek ze źródła s oraz długość d najkrótszej znanej ścieżki ze źrodła s. Rozpatrując nową krawędź (u, v) oceniamy, czy pozwoli ona na skrócenie ścieżki do v.



node	a	b	c
d	0	∞	∞
π	N	N	N

Najkrótsze ścieżki: relaksacje

Idea: każdy z węzłów zapamiętuje poprzednik π w budowanym drzewie najkrotszych znanych ścieżek ze źródła s oraz długość d najkrótszej znanej ścieżki ze źrodła s. Rozpatrując nową krawędź (u, v) oceniamy, czy pozwoli ona na skrócenie ścieżki do v.

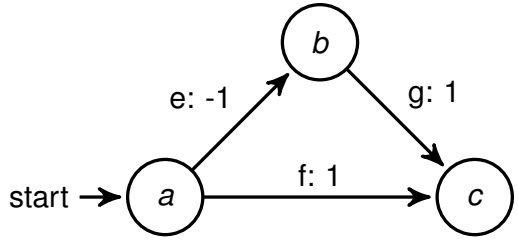


node	a	b	c
d	0	∞	1
π	N	N	a

relax(f)

Najkrótsze ścieżki: relaksacje

Idea: każdy z węzłów zapamiętuje poprzednik π w budowanym drzewie najkrotszych znanych ścieżek ze źródła s oraz długość d najkrótszej znanej ścieżki ze źrodła s. Rozpatrując nową krawędź (u, v) oceniamy, czy pozwoli ona na skrócenie ścieżki do v.

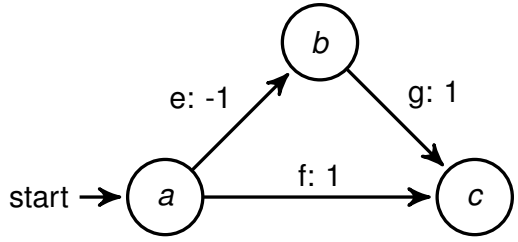


node	a	b	c
d	0	-1	1
π	N	a	a

relax(e)

Najkrótsze ścieżki: relaksacje

Idea: każdy z węzłów zapamiętuje poprzednik π w budowanym drzewie najkrotszych znanych ścieżek ze źródła s oraz długość d najkrótszej znanej ścieżki ze źrodła s. Rozpatrując nową krawędź (u, v) oceniamy, czy pozwoli ona na skrócenie ścieżki do v.

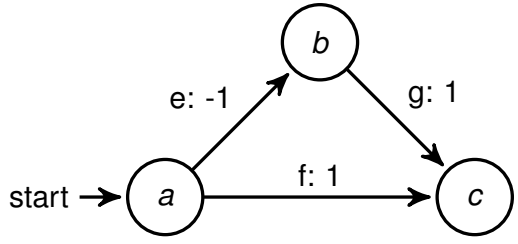


node	a	b	c
d	0	-1	0
π	N	a	b

relax(g)

Najkrótsze ścieżki: relaksacje

Idea: każdy z węzłów zapamiętuje poprzednik π w budowanym drzewie najkrotszych znanych ścieżek ze źródła s oraz długość d najkrótszej znanej ścieżki ze źrodła s. Rozpatrując nową krawędź (u, v) oceniamy, czy pozwoli ona na skrócenie ścieżki do v.



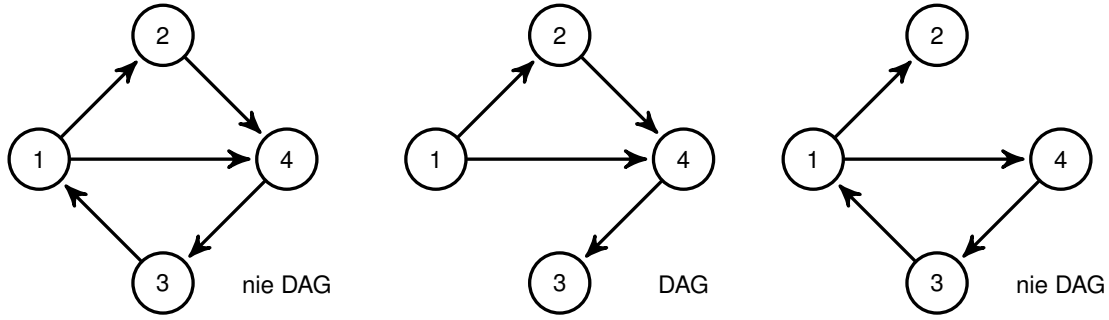
node	a	b	c
d	0	-1	0
π	N	a	b

relax(g)

Zadanie

Kolejność relaksacji f,e,g pozwoliła na znalezienie najkrótszych ścieżek z a. Czy każda kolejność to gwarantuje?

Najkrótsze ścieżki: skierowany graf acykliczny (DAG)

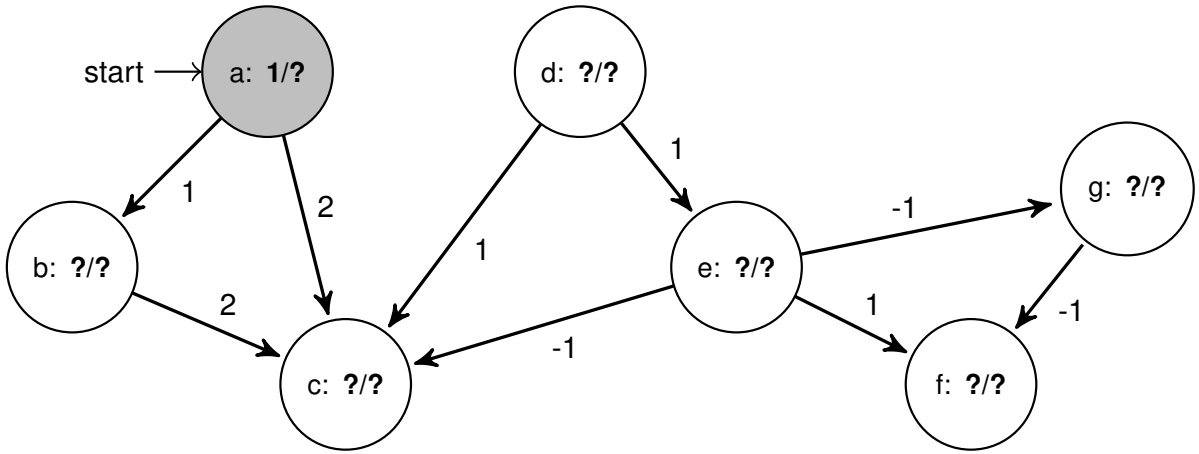


Aby stwierdzić, czy graf jest DAG-iem, wystarczy wykonać DFS z wykrywaniem krawędzi wstecznej, modyfikując DFS-VISIT:

```
1.  is_dag = true

DFS-VISIT(u)
2.  color[u] = SZARY
3.  ++time
4.  d[u] = time
5.  for kazdy v in Adj[u]
6.      if color[v] = SZARY then is_dag = false
7.      if color[v] = BIALY
8.         pi[v] = u
9.         DFS-VISIT(v)
10. color[u] = CZARNY
11. f[u] = ++time
```

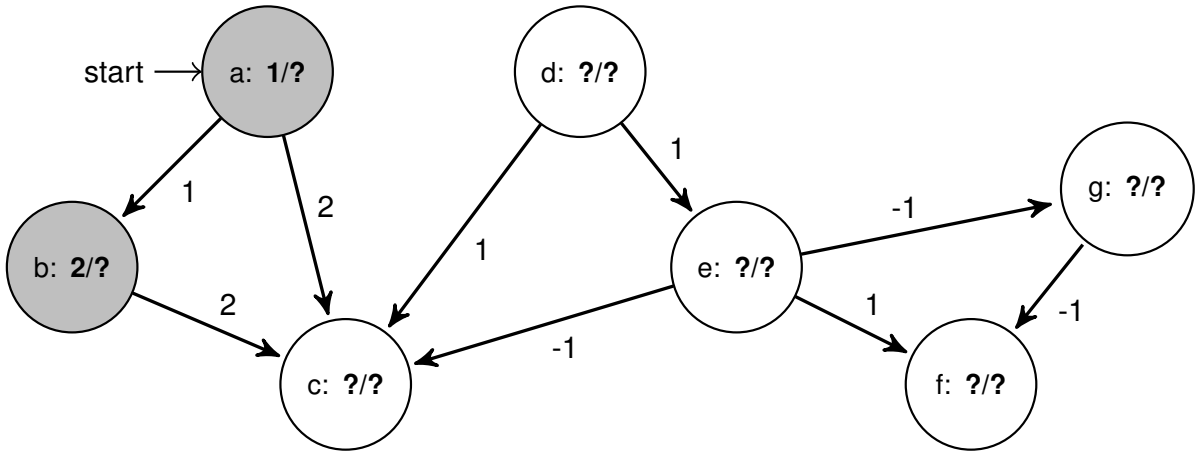
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



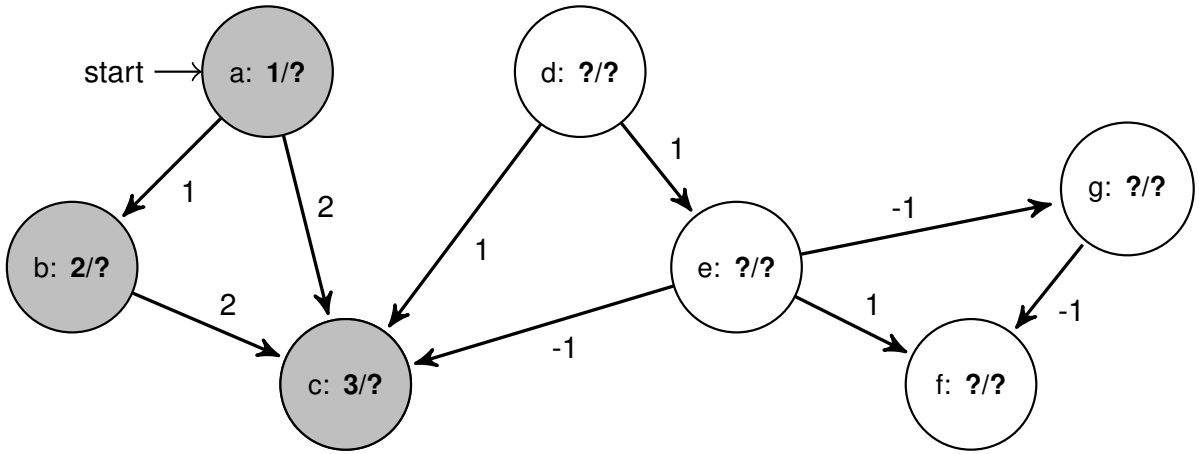
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



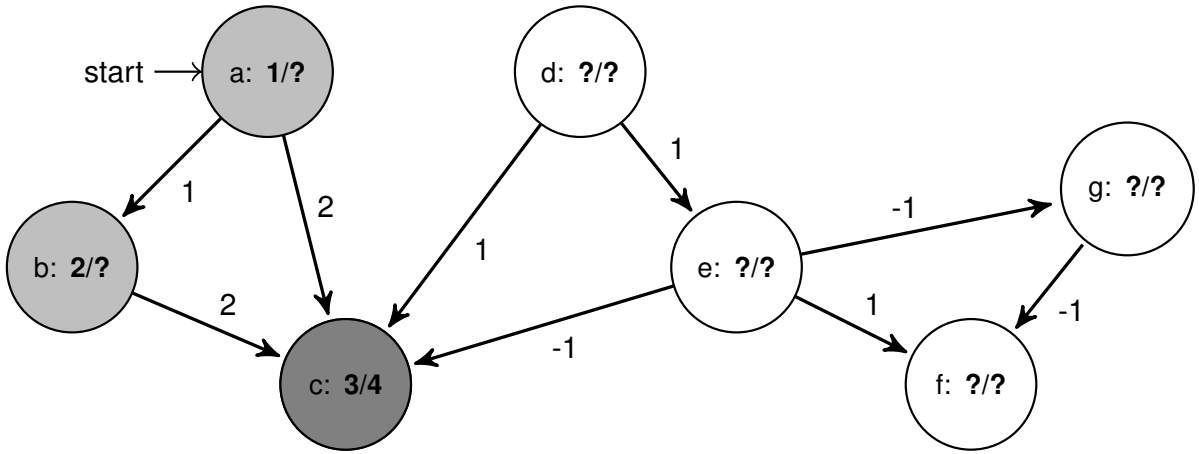
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



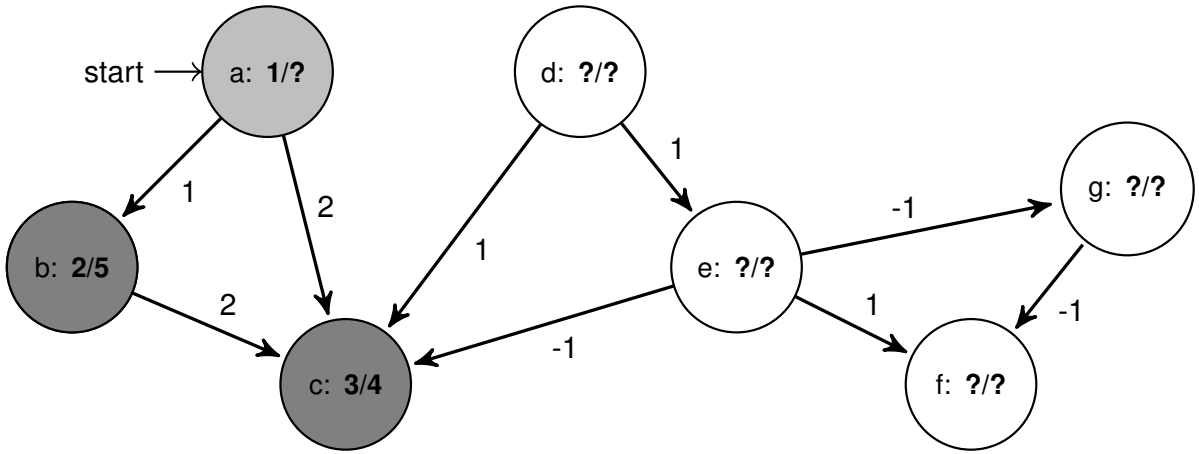
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



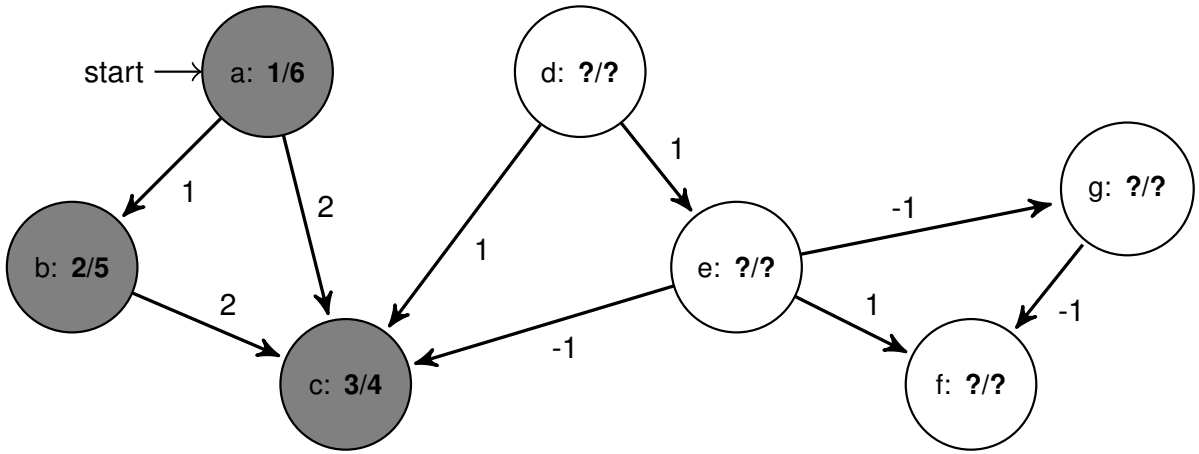
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



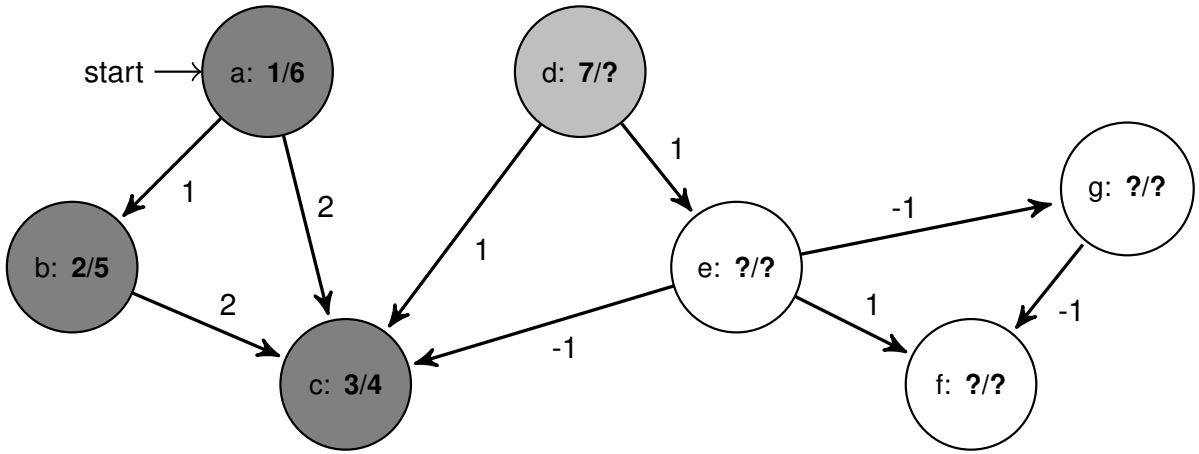
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



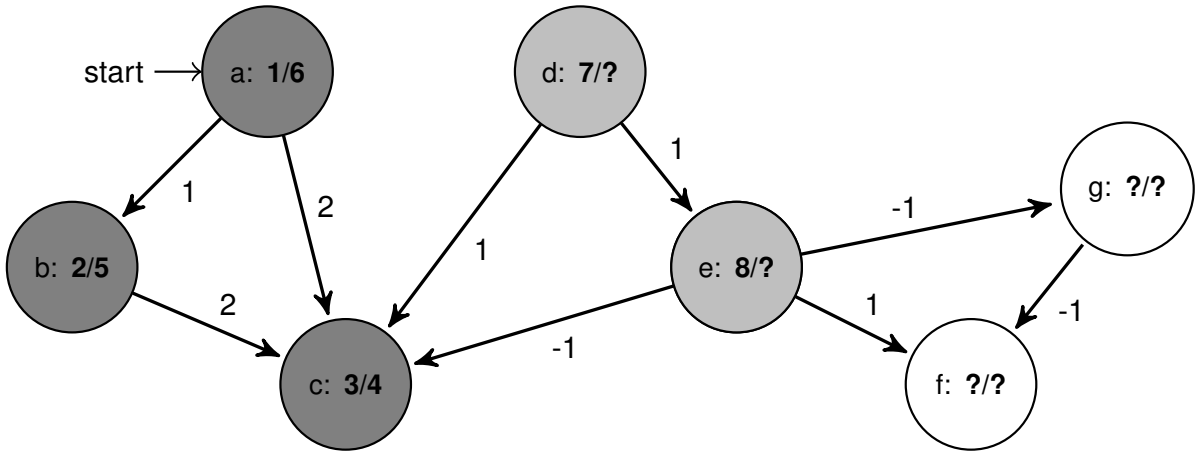
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



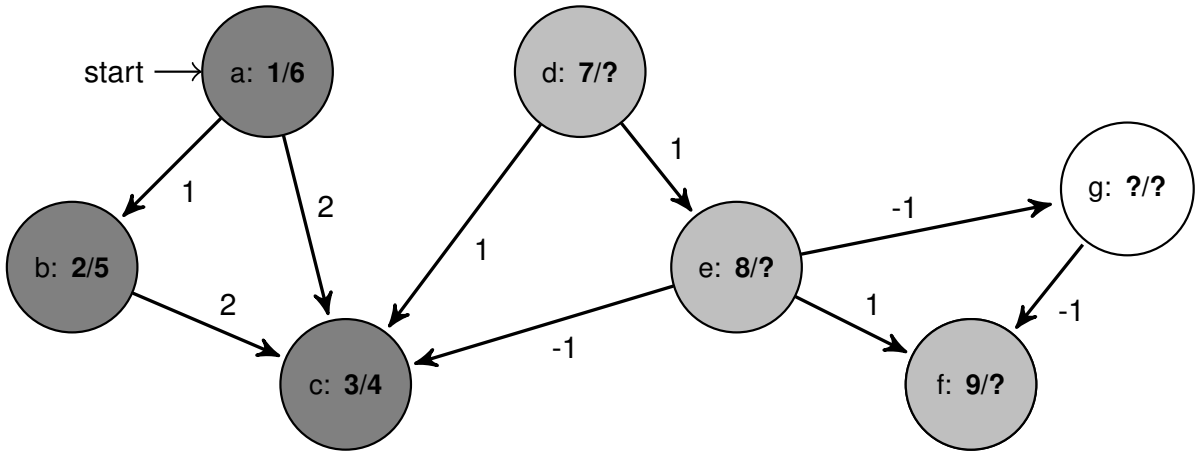
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



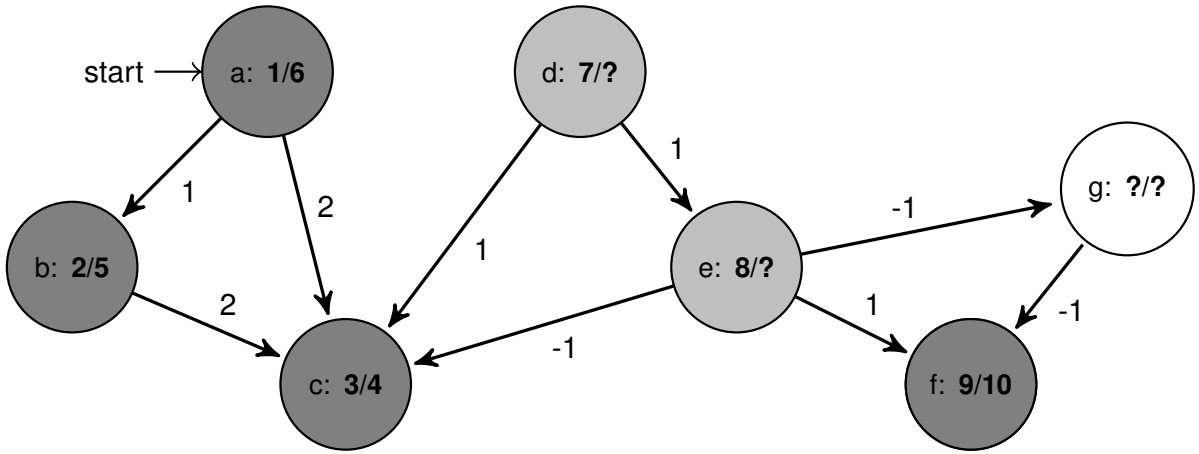
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



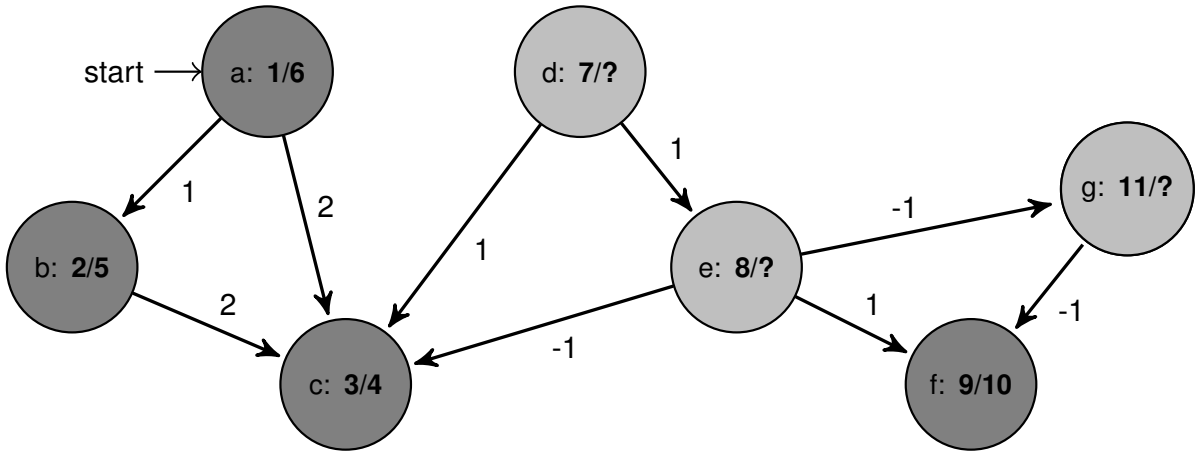
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



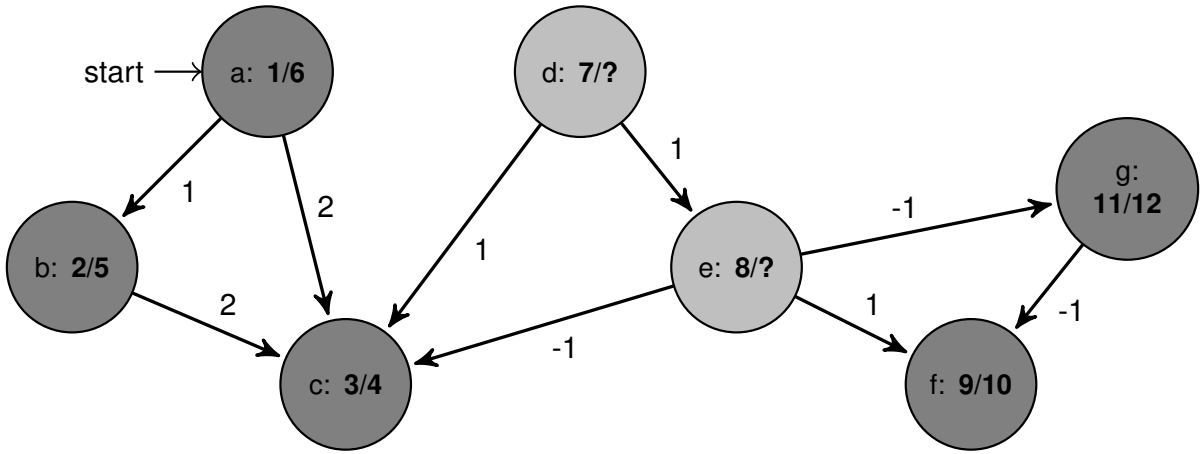
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



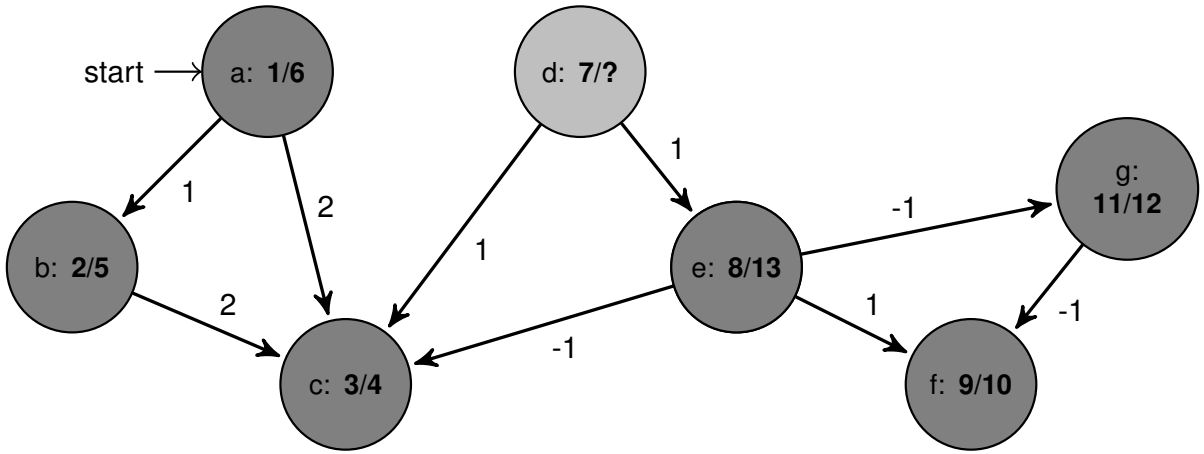
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



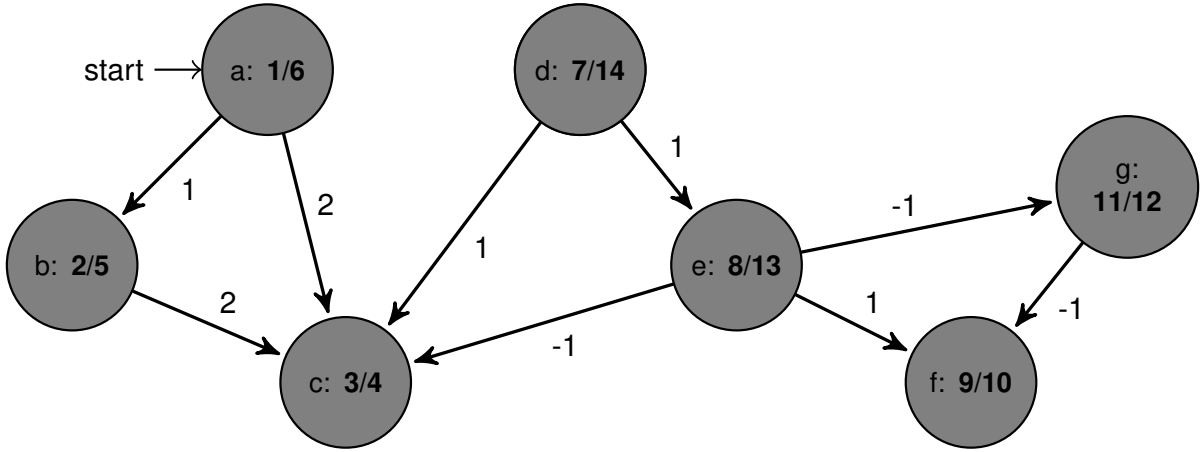
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.



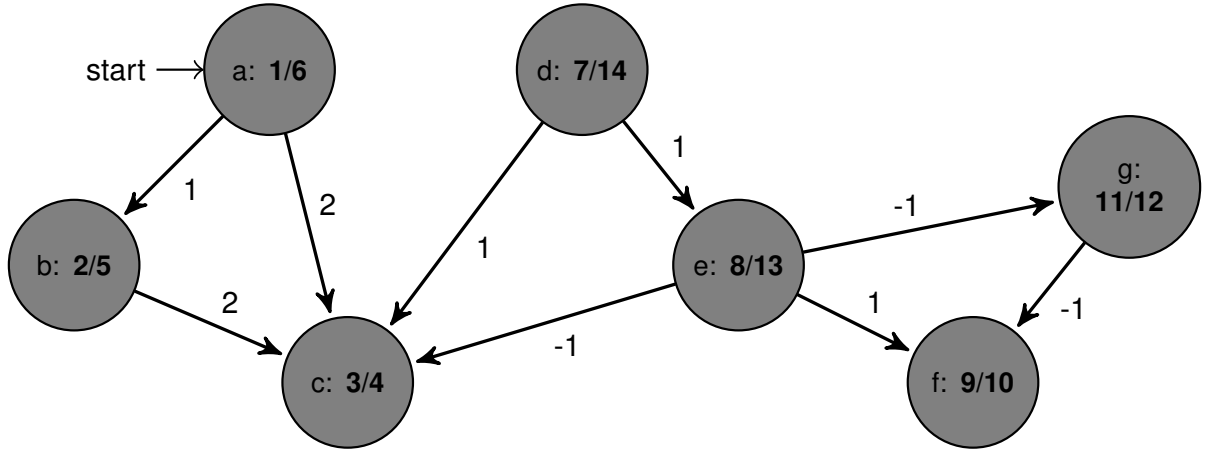
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.

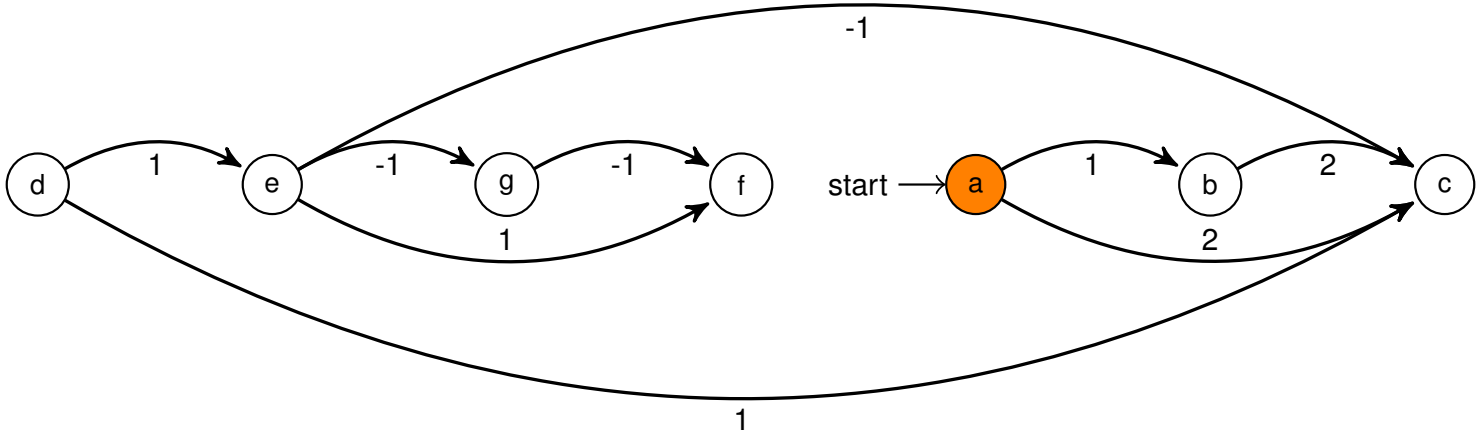


Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Sortowanie topologiczne: jeśli graf jest DAG-iem, to możemy posortować jego wierzchołki malejąco względem końca czasu przetwarzania.

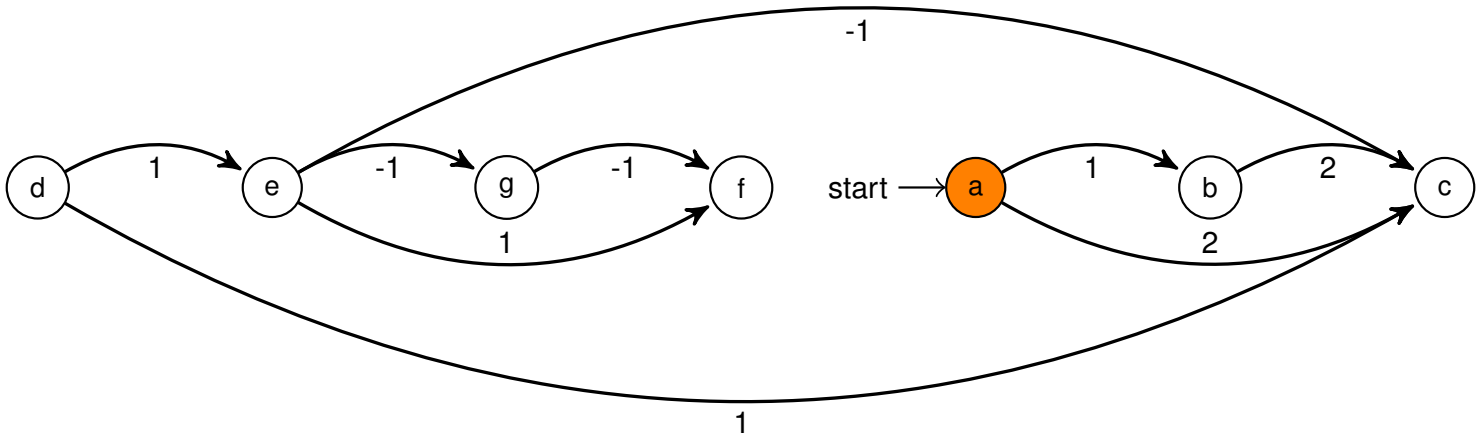


Posortowane:



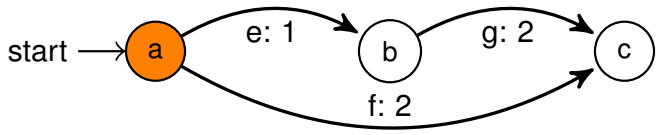
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

W posortowanym DAG-u wierzchołki, które poprzedzają źródło są nieosiągalne i można je usunąć:



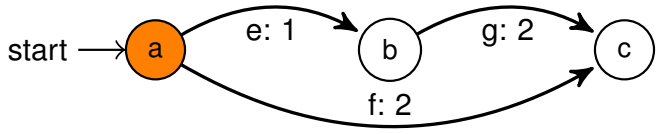
Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).



Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

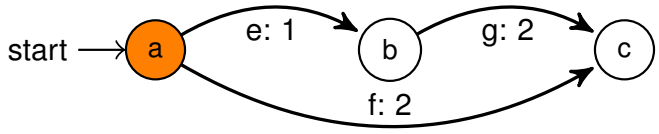
Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).



node	a	b	c
d	0	∞	∞
π	N	N	N

Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).

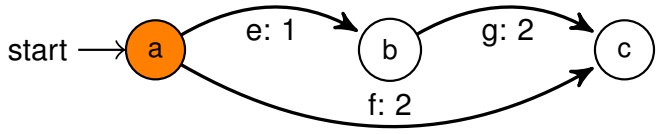


node	a	b	c
d	0	1	∞
π	N	a	N

relax(e)

Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).

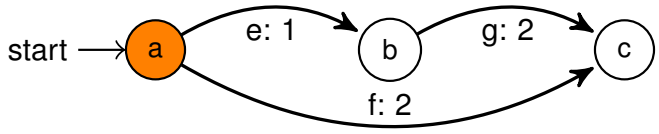


node	a	b	c
d	0	1	2
π	N	a	a

relax(f)

Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).

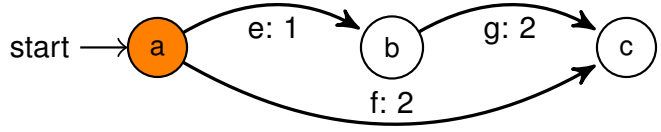


node	a	b	c
d	0	1	2
π	N	a	a

relax(g) – brak zmiany

Najkrótsze ścieżki: skierowany graf acykliczny (DAG), c.d.

Teraz wystarczy dokonać relaksacji pozostałych krawędzi zgodnie z porządkiem sortowania (zaczynając od źródła).



node	a	b	c
d	0	1	2
π	N	a	a

relax(g) – brak zmiany

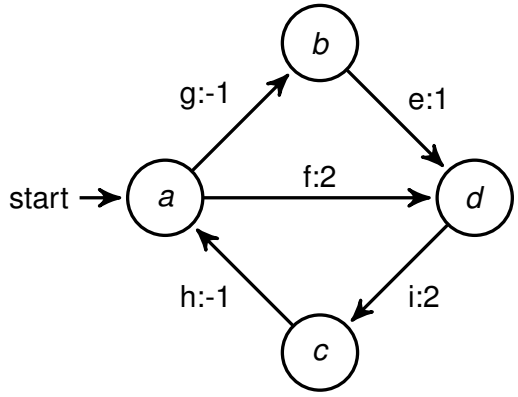
Złożoność wzgl. sumy liczby krawędzi i węzłow:
liniowa (DFS z wykrywaniem cykli) + liniowa (relaksacje) = liniowa.

Najkrótsze ścieżki: algorytm Bellmana-Forda

```
Bellman-Ford(s):
// init
1.    for kazdy wierzcholek u in V[G]
2.       pi[u] := NULL
3.       d[u] := INFTY
6.    color[s] := SZARY
7.    d[s] := 0

// relax
8.    for i = 1 to (|V| - 1):
9.       for each edge (u,v) in E:
10.         relax(u,v)

// test for negative cycles
11.   for each edge (u,v) in E:
12.      if d[v] > d[u] + w(u,v) return False

13. return True
```

Idea: dokonaj relaksacji każdej z krawędzi i powtórz to o jeden mniej razy, niż jest wierzchołków.
Działa (przy braku ujemnych cykli), bo: każda z najkrótszych ścieżek ma co najwyżej $|V| - 1$ krawędzi (zasada szufladkowa).
Złożoność: $O(|V| \cdot |E|)$.

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.

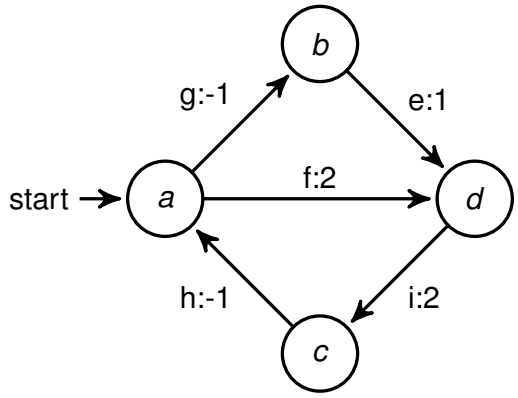


W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	∞	∞	∞
π	N	N	N	N

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



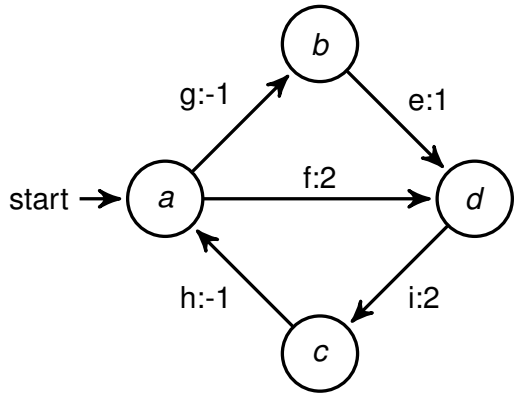
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	∞	∞	∞
π	N	N	N	N

relax(e) – b/z

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



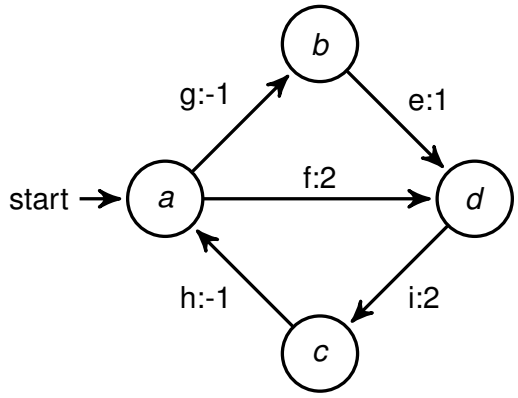
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	∞	∞	2
π	N	N	N	a

relax(f)

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



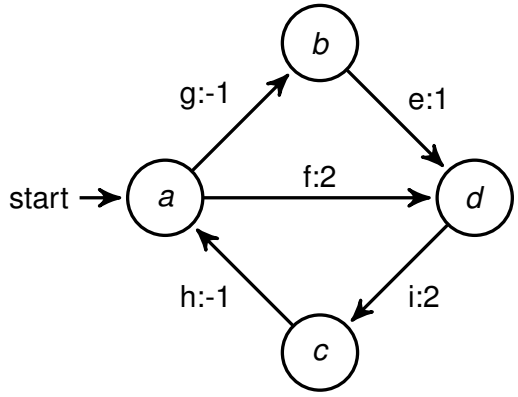
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	-1	∞	2
π	N	a	N	a

relax(g)

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



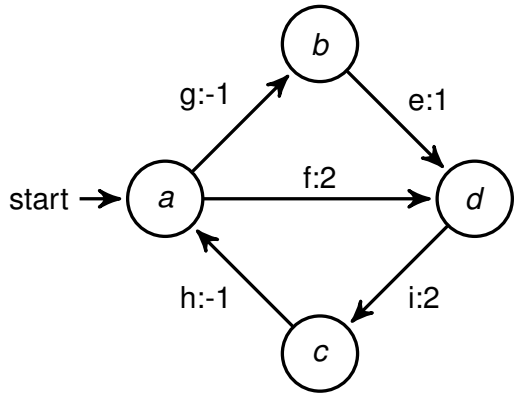
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	-1	∞	2
π	N	a	N	a

relax(h) – b/z

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



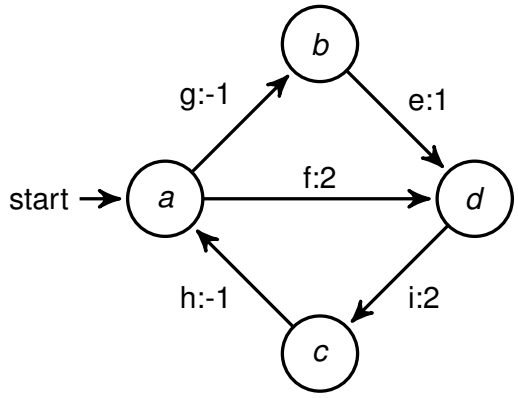
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(1) runda:

node	a	b	c	d
d	0	-1	4	2
π	N	a	d	a

relax(i)

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



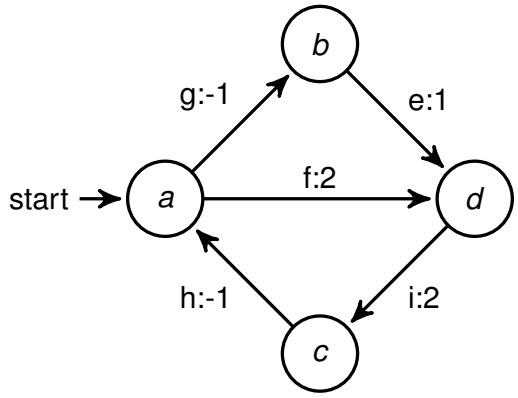
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(2) runda:

node	a	b	c	d
d	0	-1	4	0
π	N	a	d	b

relax(e)

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



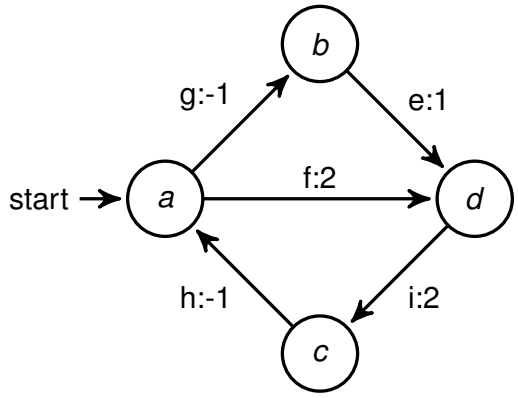
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(2) runda:

node	a	b	c	d
d	0	-1	4	0
π	N	a	d	b

relax(f) – b/z

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



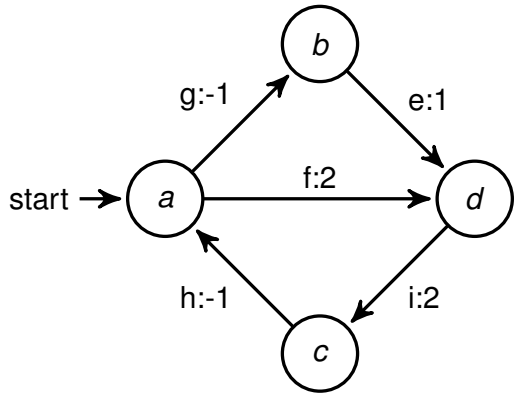
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(2) runda:

node	a	b	c	d
d	0	-1	4	0
π	N	a	d	b

relax(g) – b/z

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



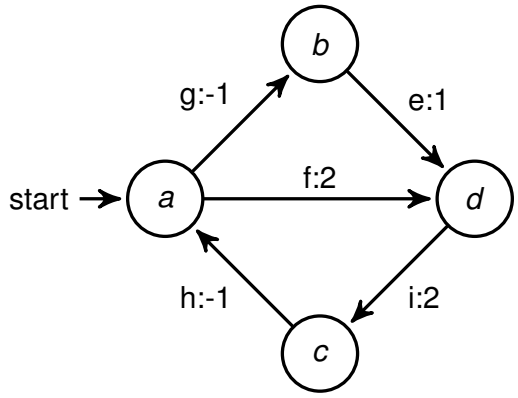
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(2) runda:

node	a	b	c	d
d	0	-1	4	0
π	N	a	d	b

relax(h) – b/z

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



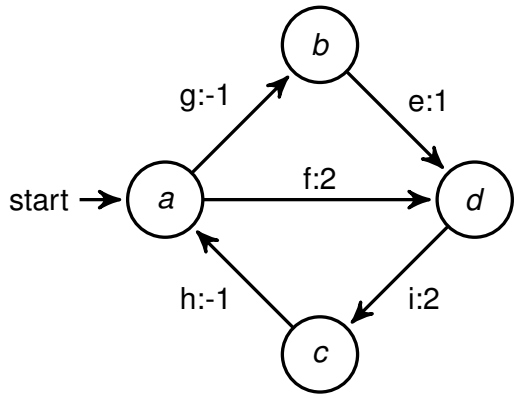
W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(2) runda:

node	a	b	c	d
d	0	-1	2	0
π	N	a	d	b

relax(i)

Najkrótsze ścieżki: algorytm Bellmana-Forda, c.d.



W każdej z 3 rund zachowamy kolejność relaksacji: e,f,g,h,i.

(3) runda:

node	a	b	c	d
d	0	-1	2	0
π	N	a	d	b

b/z

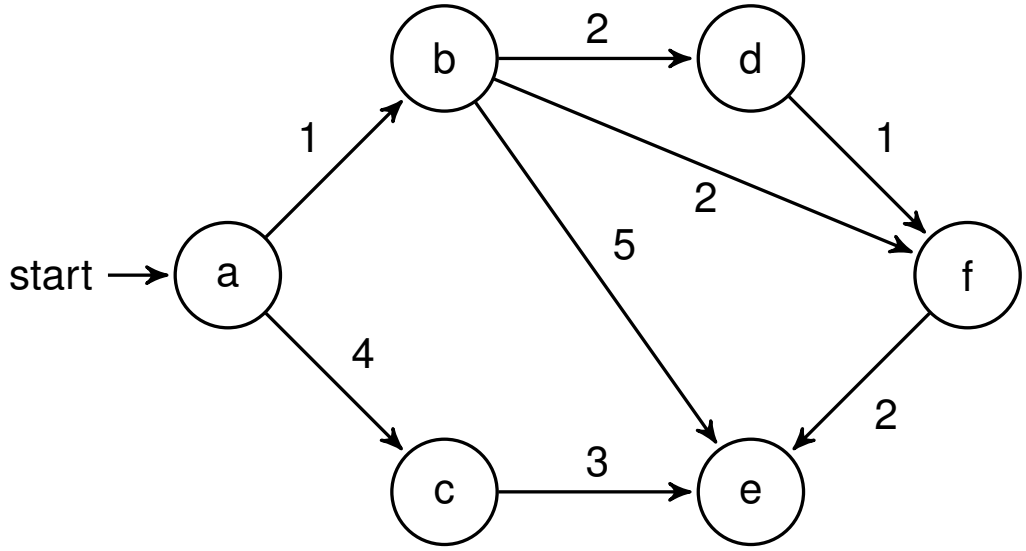
(ale i tak musimy policzyć)

Najkrótsze ścieżki: algorytm Dijkstry

```
Dijkstra(s):
// init
1.   for kazdy wierzcholek u in V[G]
2.      pi[u] := NULL
3.      d[u] := INFTY
6.   color[s] := SZARY
7.   d[s] := 0

8.   S := empty // lista przetworzonych wierzchołków
9.   Q := min_pq(V[G]) // kolejka priorytetowa wierzchołków wzg. wart. d

// obliczenie najkrótszych ścieżek
10.  while Q not empty:
11.        u = Q.delmin()
12.        S.add(u)
13.        for kazdy v in Adj[u] relax(u,v)
```

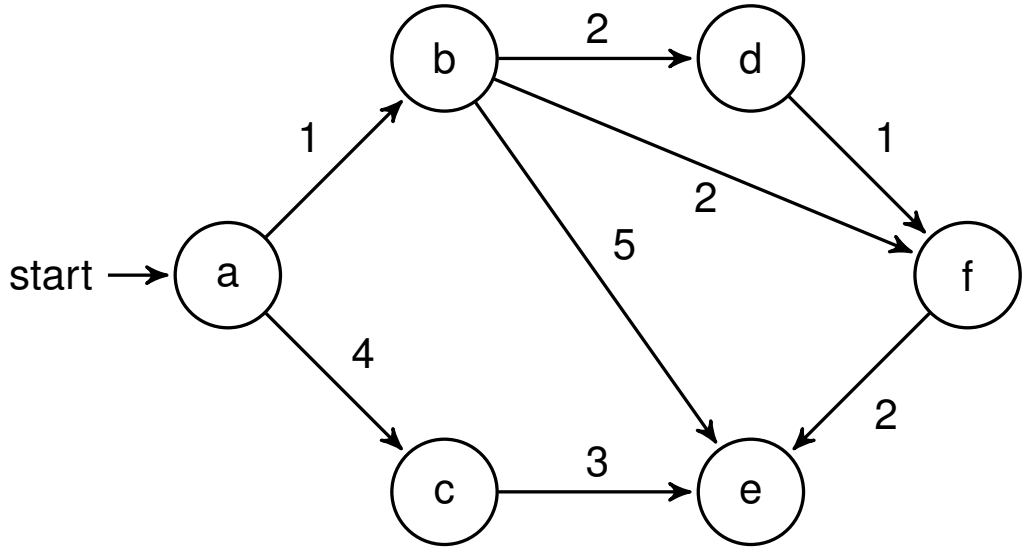
Złożoność: $O((|V| + |E|) \cdot log|V|)$ przy wykorzystaniu kopca jako kolejki pomocniczej.

Najkrótsze ścieżki: algorytm Dijkstry, c.d.



N	d(b),$\pi(b)$	d(c),$\pi(c)$	d(d),$\pi(d)$	d(e),$\pi(e)$	d(f),$\pi(f)$

Najkrótsze ścieżki: algorytm Dijkstry, c.d.



N	**d(b),**$\pi(b)$	**d(c),**$\pi(c)$	**d(d),**$\pi(d)$	**d(e),**$\pi(e)$	**d(f),**$\pi(f)$
a	<u>1a</u>	4a	∞	∞	∞
ab		4a	<u>3b</u>	6b	3b
abd		4a		6b	<u>3b</u>
abdf		<u>4a</u>		5f	
abdfc				<u>5f</u>	
abdfce					