Poprawność i złożoność obliczeniowa algorytmów

Piotr Beling

Uniwersytet Łódzki

2020

http://pbeling.w8.pl

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Algorytm to:

- przepis, sposób postępowania prowadzący do rozwiązania (złożonego) problemu lub osiągnięcia celu,
- ciąg czynności elementarnych, czyli zrozumiałych dla wykonawcy algorytmu kroków.

Wykonawca algorytmu to zazwyczaj człowiek lub komputer.

Przykłady algorytmów:

- przepis kulinarny,
- algorytmy pisemnego dodawania, odejmowania, mnożenia oraz dzielenia liczb,
- algorytm Euklidesa do wyznaczania największego wspólnego dzielnika (jeden z najstarszych algorytmów, pojawił się w dziele „Elementy” w około 300r. p.n.e.; wciąż używany).

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Poprawność

Dalej zajmujemy się algorytmami wykonywanymi przez komputer.

Specyfikacja takiego algorytmu składa się z opisów:

- wejścia – jakie dane wejściowe są dopuszczalne i w jakim formacie (typy danych, zakresy, itp.);
- wyjścia – jaki wynik zwraca algorytm dla poszczególnych danych wejściowych i w jakim formacie.

Poprawny algorytm to taki, który, dla poprawnego wejścia:

- zakończy pracę w skończonym czasie (ma własność stopu),
- i zwróci wynik zgodny ze specyfikacją.

Poprawności algorytmów zazwyczaj dowodzi się:

- indukcyjnie,
- definiując i dowodząc tzw. niezmienniki, które są spełnione na różnych etapach wykonywania algorytmu.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Uwaga do pseudokodu: `|T|` oznacza długość tablicy `T`, zaś jej prawidłowe indeksy to: `0`, `1`, ..., `|T|-1`.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Uwaga do pseudokodu: `|T|` oznacza długość tablicy `T`, zaś jej prawidłowe indeksy to: `0`, `1`, ..., `|T|-1`.

Specyfikacja algorytmu:

- wejście: tablica `T` i wartość `e`.
- wyjście (wynik): liczba równa:
  - najmniejszemu indeksowi tablicy `T`, pod którym występuje wartość `e`,
  - albo `-1` gdy `T` nie zawiera `e`.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Uwaga do pseudokodu: `|T|` oznacza długość tablicy `T`, zaś jej prawidłowe indeksy to: `0`, `1`, ..., `|T|-1`.

Specyfikacja algorytmu:

- wejście: tablica `T` i wartość `e`.
- wyjście (wynik): liczba równa:
  - najmniejszemu indeksowi tablicy `T`, pod którym występuje wartość `e`,
  - albo `-1` gdy `T` nie zawiera `e`.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Uwaga do pseudokodu: `|T|` oznacza długość tablicy `T`, zaś jej prawidłowe indeksy to: `0`, `1`, ..., `|T|-1`.

Poprawność algorytmu:

- algorytm ma własność stopu gdyż jedyna pętla wykona się co najwyżej `|T|` razy

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wykażemy dwa niezmienniki P oraz K, które są spełnione (i zdefiniowane) we wskazanych miejscach pseudokodu
- P wykażemy indukcyjnie, przy okazji dowodząc K.
- W pierwszej iteracji (dla `i`=0) P jest pusto-spełniony.
- Dalej wykażemy, że jeśli
  - (założenie indukcyjne) P jest spełniony w i-tej iteracji (czyli `T[j]`$\neq$`e` dla wszystkich $j < i$),
  - to jest on też spełniony w iteracji $i + 1$
  - (przy okazji wykażemy K w i-tej iteracji).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu, krok indukcyjny:

- Jeśli zajdzie warunek `T[i] = e` to zostanie zwrócony indeks `i`, zgodny ze specyfikacją wyjścia (`T[i]` $=$ `e` z warunku, zaś `T[j]`$\neq$`e` dla wszystkich $j < i$ z założenia indukcyjnego o P).
- Jeśli warunek `T[i] = e` nie zajdzie, to sterowanie osiągnie miejsce niezmiennika K, spełnionego w bieżącej (i-tej) iteracji na podstawie założenia indukcyjnego (dla $j < i$) oraz niezajścia warunku (dla $j = i$).
- Zaś wprost z K w iteracji i wynika P w iteracji $i + 1$.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu, krok indukcyjny:

- Jeśli zajdzie warunek `T[i] = e` to zostanie zwrócony indeks `i`, zgodny ze specyfikacją wyjścia (`T[i]` $=$ `e` z warunku, zaś `T[j]`$\neq$`e` dla wszystkich $j < i$ z założenia indukcyjnego o P).
- Jeśli warunek `T[i] = e` nie zajdzie, to sterowanie osiągnie miejsce niezmiennika K, spełnionego w bieżącej (i-tej) iteracji na podstawie założenia indukcyjnego (dla $j < i$) oraz niezajścia warunku (dla $j = i$).
- Zaś wprost z K w iteracji i wynika P w iteracji $i + 1$.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu, krok indukcyjny:

- Jeśli zajdzie warunek `T[i] = e` to zostanie zwrócony indeks `i`, zgodny ze specyfikacją wyjścia (`T[i]` $=$ `e` z warunku, zaś `T[j]`$\neq$`e` dla wszystkich $j < i$ z założenia indukcyjnego o P).
- Jeśli warunek `T[i] = e` nie zajdzie, to sterowanie osiągnie miejsce niezmiennika K, spełnionego w bieżącej (i-tej) iteracji na podstawie założenia indukcyjnego (dla $j < i$) oraz niezajścia warunku (dla $j = i$).
- Zaś wprost z K w iteracji i wynika P w iteracji $i + 1$.

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wcześniej wykazaliśmy, że gdy w którejkolwiek iteracji zajdzie warunek `T[i] = e`, to zostanie zwrócony prawidłowy indeks.
- Jeśli warunek `T[i] = e` nigdy nie zajdzie, to pętla przebiegnie przez wszystkie indeksy tablicy, zakończy się, i zostanie zwrócone -1, zgodne ze specyfikacją wyjścia na podstawie:
- niezmiennika K w ostatniej iteracji pętli, tj. dla `i=|T|-1`,
- albo, gdy pętla nie wykonała żadnej iteracji, faktu, że tablica jest pusta (`|T|=0`).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wcześniej wykazaliśmy, że gdy w którejkolwiek iteracji zajdzie warunek `T[i] = e`, to zostanie zwrócony prawidłowy indeks.
- Jeśli warunek `T[i] = e` nigdy nie zajdzie, to pętla przebiegnie przez wszystkie indeksy tablicy, zakończy się, i zostanie zwrócone -1, zgodne ze specyfikacją wyjścia na podstawie:
- niezmiennika K w ostatniej iteracji pętli, tj. dla `i=|T|-1`,
- albo, gdy pętla nie wykonała żadnej iteracji, faktu, że tablica jest pusta (`|T|=0`).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wcześniej wykazaliśmy, że gdy w którejkolwiek iteracji zajdzie warunek `T[i] = e`, to zostanie zwrócony prawidłowy indeks.
- Jeśli warunek `T[i] = e` nigdy nie zajdzie, to pętla przebiegnie przez wszystkie indeksy tablicy, zakończy się, i zostanie zwrócone -1, zgodne ze specyfikacją wyjścia na podstawie:
- niezmiennika K w ostatniej iteracji pętli, tj. dla `i=|T|-1`,
- albo, gdy pętla nie wykonała żadnej iteracji, faktu, że tablica jest pusta (`|T|=0`).

Przykład: algorytm wyszukujący wartości `e` w tablicy `T`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        // P: T[j] ≠ e dla j = 0, 1, ..., i-1
        if T[i] = e: return i
        // K: T[j] ≠ e dla j = 0, 1, ..., i
    return -1
```

Poprawność algorytmu:

- Wcześniej wykazaliśmy, że gdy w którejkolwiek iteracji zajdzie warunek `T[i] = e`, to zostanie zwrócony prawidłowy indeks.
- Jeśli warunek `T[i] = e` nigdy nie zajdzie, to pętla przebiegnie przez wszystkie indeksy tablicy, zakończy się, i zostanie zwrócone -1, zgodne ze specyfikacją wyjścia na podstawie:
- niezmiennika K w ostatniej iteracji pętli, tj. dla `i=|T|-1`,
- albo, gdy pętla nie wykonała żadnej iteracji, faktu, że tablica jest pusta (`|T|=0`).

Motywacja

Opracowujemy nowe algorytmy aby:

- rozwiązywać nowe, dotychczas nierozwiązane problemy;
- rozwiązywać problemy zużywając przy tym mniej zasobów.

W przypadku wykonywania algorytmu przez komputer, zużywane zasoby to **czas** i **pamięć**.

Cechę algorytmu charakteryzującą ilość używanych przez niego zasobów nazywamy jego **złożonością obliczeniową**, odpowiednio (w zależności od zasobu) **czasową** albo **pamięciową**.

Motywacja

Opracowujemy nowe algorytmy aby:

- rozwiązywać nowe, dotychczas nierozwiązane problemy;
- rozwiązywać problemy zużywając przy tym mniej zasobów.

W przypadku wykonywania algorytmu przez komputer, zużywane zasoby to **czas** i **pamięć**.

Cechę algorytmu charakteryzującą ilość używanych przez niego zasobów nazywamy jego **złożonością obliczeniową**, odpowiednio (w zależności od zasobu) **czasową** albo **pamięciową**.

Motywacja

Opracowujemy nowe algorytmy aby:

- rozwiązywać nowe, dotychczas nierozwiązane problemy;
- rozwiązywać problemy zużywając przy tym mniej zasobów.

W przypadku wykonywania algorytmu przez komputer, zużywane zasoby to **czas** i **pamięć**.

Cechę algorytmu charakteryzującą ilość używanych przez niego zasobów nazywamy jego **złożonością obliczeniową**, odpowiednio (w zależności od zasobu) **czasową** albo **pamięciową**.

Motywacja

Opracowujemy nowe algorytmy aby:

- rozwiązywać nowe, dotychczas nierozwiązane problemy;
- rozwiązywać problemy zużywając przy tym mniej zasobów.

W przypadku wykonywania algorytmu przez komputer, zużywane zasoby to **czas** i **pamięć**.

Cechę algorytmu charakteryzującą ilość używanych przez niego zasobów nazywamy jego **złożonością obliczeniową**, odpowiednio (w zależności od zasobu) **czasową** albo **pamięciową**.

Motywacja

Opracowujemy nowe algorytmy aby:

- rozwiązywać nowe, dotychczas nierozwiązane problemy;
- rozwiązywać problemy zużywając przy tym mniej zasobów.

W przypadku wykonywania algorytmu przez komputer, zużywane zasoby to **czas** i **pamięć**.

Cechę algorytmu charakteryzującą ilość używanych przez niego zasobów nazywamy jego **złożonością obliczeniową**, odpowiednio (w zależności od zasobu) **czasową** albo **pamięciową**.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara.
Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara. Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara.
Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara.
Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara.
Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara. Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność czasowa

Złożoność czasowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- liczbę operacji elementarnych (albo, zazwyczaj, **operacji dominujących**) wykonanych przez algorytm dla tych danych.

Operacje dominujące algorytmu to

- wyróżnione operacje elementarne,
- których liczba wykonań jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.
- np. mnożenie dla algorytmu potęgowania

Złożoność czasową można by mierzyć za pomocą zegara.
Wtedy wynik zależałby jednak nie tylko od samego algorytmu, ale także od szybkości wykonującego go komputera.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pamięciowa

Złożoność pamięciowa algorytmu to funkcja przyporządkowująca

- rozmiarowi danych wejściowych,
- ilość pamięci (bitów lub słów maszynowych) użytej do wykonania algorytm dla tych danych,
- (domyślnie) nie wliczając w to pamięci użytej do zapisania danych wejściowych i wyjściowych (wyniku).

Przyjmujemy, że obliczenia prowadzone są w tzw. modelu *Word RAM*, zakładającym (w uproszczeniu) operowanie słowami, które:

- mają stałą wielkość (np. 64 bity),
- są na tyle duże, by móc adresować całą pamięć komputera,
- i na których możemy wykonywać podstawowe operacje w stałym czasie.

Współczesne komputery w przybliżeniu realizują ten model.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Złożoność pesymistyczna, optymistyczna, oczekiwana

Ilość użytych zasobów (czasu lub pamięci) może być różna, dla różnych zestawów danych o tym samym rozmiarze n.

Dlatego wyróżniamy różne warianty złożoność obliczeniowej:

- **pesymistyczny** – maksymalny, uzyskany dla „najtrudniejszego" zestawu danych o rozmiarze n;
- **optymistyczny** – minimalny, uzyskany dla „najłatwiejszego" zestawu danych o rozmiarze n;
- **oczekiwany** (średni) – suma złożoności wszystkich zestawów danych rozmiaru n, ważona prawdopodobieństwami ich wystąpienia na wejściu.
  Często obliczana przy założeniu, że poszczególne zestawy danych są równie prawdopodobne.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Rozmiar danych wejściowych = wielkość tablicy `T`, tj. $n = |T|$.

Operacja dominująca: porównanie elementów (`T[i]=e`)

- jest wykonywana w każdym przebiegu jedynej pętli,
- więc liczba jej wykonań jest proporcjonalna do liczby wszystkich operacji.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Rozmiar danych wejściowych = wielkość tablicy `T`, tj. $n = |T|$.
Operacja dominująca: porównanie elementów (`T[i]=e`)

- jest wykonywana w każdym przebiegu jedynej pętli,
- więc liczba jej wykonań jest proporcjonalna do liczby wszystkich operacji.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Rozmiar danych wejściowych = wielkość tablicy `T`, tj. $n = |T|$.
Operacja dominująca: porównanie elementów (`T[i]=e`)

- jest wykonywana w każdym przebiegu jedynej pętli,
- więc liczba jej wykonań jest proporcjonalna do liczby wszystkich operacji.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Rozmiar danych wejściowych = wielkość tablicy `T`, tj. $n = |T|$.
Operacja dominująca: porównanie elementów (`T[i]=e`)

- jest wykonywana w każdym przebiegu jedynej pętli,
- więc liczba jej wykonań jest proporcjonalna do liczby wszystkich operacji.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Czasowa złożoność obliczeniowa to:

- optymistyczna: 1 porównanie (gdy `T[0]=e`);
- pesymistyczna: $n = |T|$ porównań (gdy `e` występuje w `T` jedynie na ostatniej pozycji albo wcale);
- oczekiwana: $\sum_{i=1}^{n} ip_i + np_{\notin}$ porównań, gdzie:
  - p_i - prawdopodobieństwo, że `e` w `T` występuje po raz pierwszy na pozycji o indeksie $i-1$,
  - $p_{\notin}$ - prawdopodobieństwo, że `e` nie ma w `T`.

Przykładowo, dla $p_1 = \cdots = p_n = 1/n$ i $p_{\notin} = 0$ otrzymujemy: $\sum_{i=1}^{n} i\frac{1}{n} + n \cdot 0 = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{(1+n)n}{2} = \frac{n+1}{2}$.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Czasowa złożoność obliczeniowa to:

- optymistyczna: 1 porównanie (gdy `T[0]=e`);
- pesymistyczna: $n = |T|$ porównań (gdy `e` występuje w `T` jedynie na ostatniej pozycji albo wcale);
- oczekiwana: $\sum_{i=1}^{n} i p_i + n p_{\notin}$ porównań, gdzie:
  - p_i - prawdopodobieństwo, że `e` w `T` występuje po raz pierwszy na pozycji o indeksie $i-1$,
  - $p_{\notin}$ - prawdopodobieństwo, że `e` nie ma w `T`.

Przykładowo, dla $p_1 = \cdots = p_n = 1/n$ i $p_{\notin} = 0$ otrzymujemy: $\sum_{i=1}^{n} i\frac{1}{n} + n \cdot 0 = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{(1+n)n}{2} = \frac{n+1}{2}$.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Czasowa złożoność obliczeniowa to:

- optymistyczna: 1 porównanie (gdy `T[0]=e`);
- pesymistyczna: $n = |T|$ porównań (gdy `e` występuje w `T` jedynie na ostatniej pozycji albo wcale);
- oczekiwana: $\sum_{i=1}^{n} ip_i + np_{\notin}$ porównań, gdzie:
  - p_i - prawdopodobieństwo, że `e` w `T` występuje po raz pierwszy na pozycji o indeksie $i-1$,
  - $p_{\notin}$ - prawdopodobieństwo, że `e` nie ma w `T`.

Przykładowo, dla $p_1 = \cdots = p_n = 1/n$ i $p_{\notin} = 0$ otrzymujemy: $\sum_{i=1}^{n} i\frac{1}{n} + n \cdot 0 = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{(1+n)n}{2} = \frac{n+1}{2}$.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Czasowa złożoność obliczeniowa to:

- optymistyczna: 1 porównanie (gdy `T[0]=e`);
- pesymistyczna: $n = |T|$ porównań (gdy `e` występuje w `T` jedynie na ostatniej pozycji albo wcale);
- oczekiwana: $\sum_{i=1}^{n} ip_i + np_{\notin}$ porównań, gdzie:
  - p_i - prawdopodobieństwo, że `e` w `T` występuje po raz pierwszy na pozycji o indeksie $i-1$,
  - $p_{\notin}$ - prawdopodobieństwo, że `e` nie ma w `T`.

Przykładowo, dla $p_1 = \cdots = p_n = 1/n$ i $p_{\notin} = 0$ otrzymujemy: $\sum_{i=1}^{n} i\frac{1}{n} + n \cdot 0 = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{(1+n)n}{2} = \frac{n+1}{2}$.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Czasowa złożoność obliczeniowa to:

- optymistyczna: 1 porównanie (gdy `T[0]=e`);
- pesymistyczna: $n = |T|$ porównań (gdy `e` występuje w `T` jedynie na ostatniej pozycji albo wcale);
- oczekiwana: $\sum_{i=1}^{n} i p_i + n p_{\notin}$ porównań, gdzie:
  - p_i - prawdopodobieństwo, że `e` w `T` występuje po raz pierwszy na pozycji o indeksie $i-1$,
  - $p_{\notin}$ - prawdopodobieństwo, że `e` nie ma w `T`.

  Przykładowo, dla $p_1 = \cdots = p_n = 1/n$ i $p_{\notin} = 0$ otrzymujemy: $\sum_{i=1}^{n} i\frac{1}{n} + n \cdot 0 = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{(1+n)n}{2} = \frac{n+1}{2}$.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Pamięciowa złożoność obliczeniowa: algorytm używa stałej ilości pamięci (1 słowa) do zapamiętania indeksu i.

Uwaga:

- W ogólności potrzebujemy $\lceil \log_2(n) \rceil$ bitów pamięci by zapisać liczbę z zakresu od 0 do $n-1$. Można by więc argumentować, że pamięciowa złożoność algorytmu to $\lceil \log_2(n) \rceil$ bitów.
- Ponieważ jednak w założonym modelu *Word RAM* operujemy stałej wielkości słowami na tyle dużymi, by móc adresować całą pamięć komputera, to stałej wielkości są także indeksy dowolnej tablicy mieszczącej się w jego pamięci.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Pamięciowa złożoność obliczeniowa: algorytm używa stałej ilości pamięci (1 słowa) do zapamiętania indeksu i.

Uwaga:

- W ogólności potrzebujemy $\lceil \log_2(n) \rceil$ bitów pamięci by zapisać liczbę z zakresu od 0 do $n-1$. Można by więc argumentować, że pamięciowa złożoność algorytmu to $\lceil \log_2(n) \rceil$ bitów.
- Ponieważ jednak w założonym modelu *Word RAM* operujemy stałej wielkości słowami na tyle dużymi, by móc adresować całą pamięć komputera, to stałej wielkości są także indeksy dowolnej tablicy mieszczącej się w jego pamięci.

Przykład: algorytm wyszukiwania w tablicy

Rozważmy algorytm zwracający najmniejszy indeks tablicy `T`, pod którym występuje wartość `e`, albo `-1` gdy `T` nie zawiera `e`:

```
fun find_index(T, e):
    for i ← 0, 1, ..., |T|-1:
        if T[i] = e: return i
    return -1
```

Pamięciowa złożoność obliczeniowa: algorytm używa stałej ilości pamięci (1 słowa) do zapamiętania indeksu i.

Uwaga:

- W ogólności potrzebujemy $\lceil \log_2(n) \rceil$ bitów pamięci by zapisać liczbę z zakresu od 0 do $n-1$. Można by więc argumentować, że pamięciowa złożoność algorytmu to $\lceil \log_2(n) \rceil$ bitów.
- Ponieważ jednak w założonym modelu *Word RAM* operujemy stałej wielkości słowami na tyle dużymi, by móc adresować całą pamięć komputera, to stałej wielkości są także indeksy dowolnej tablicy mieszczącej się w jego pamięci.

Przybliżone wartości wybranych funkcji:

n	$\log_2(n)$	$\sqrt{n}$	$n\log_2(n)$	n^2	2^n	$n!$
1	1	1	1	1	1	1
10	3	3	33	100	10^3	10^6
100	7	10	664	10^4	10^{30}	10^{158}
10^3	10	32	10^4	10^6	10^{301}	10^{2567}
10^4	13	100	10^5	10^8	10^{3011}	10^{35659}
10^5	17	316	10^6	10^{10}	10^{30103}	10^{456573}
10^6	20	10^3	10^7	10^{12}	10^{301030}	$10^{5565709}$
10^9	30	10^4	10^{10}	10^{18}	dużo	dużo

Czas wykonania n instrukcji przez procesor wykonujący 10^9 instrukcji/sekundę ($1GHz$):

n	czas	n	czas	n	czas	n	czas
10^9	1s	10^{13}	3h	10^{16}	116d	10^{20}	3171r
10^{11}	2m	10^{14}	28h	10^{17}	3r	10^{30}	10^{13}r
10^{12}	17m	10^{15}	12d	10^{18}	33r	10^{50}	10^{33}r

s - sekunda, m - minuta, h - godzina, d - doba, r - rok

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = 2n$.
- Czy można powiedzieć, że pierwszy z nich jest szybszy?
- A co jeśli jego operacja dominująca wykonuje się (np. 2x) wolniej niż drugiego?
- Albo gdy wykonuje on więcej operacji niedominujących?
- Lub jest uruchomiony na wolniejszym komputerze?
- W związku z powyższymi wątpliwościami, często zasadne jest używanie notacji pomijającej takie szczegóły jak stała 2 we wzorze $f_2(n) = 2n$.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

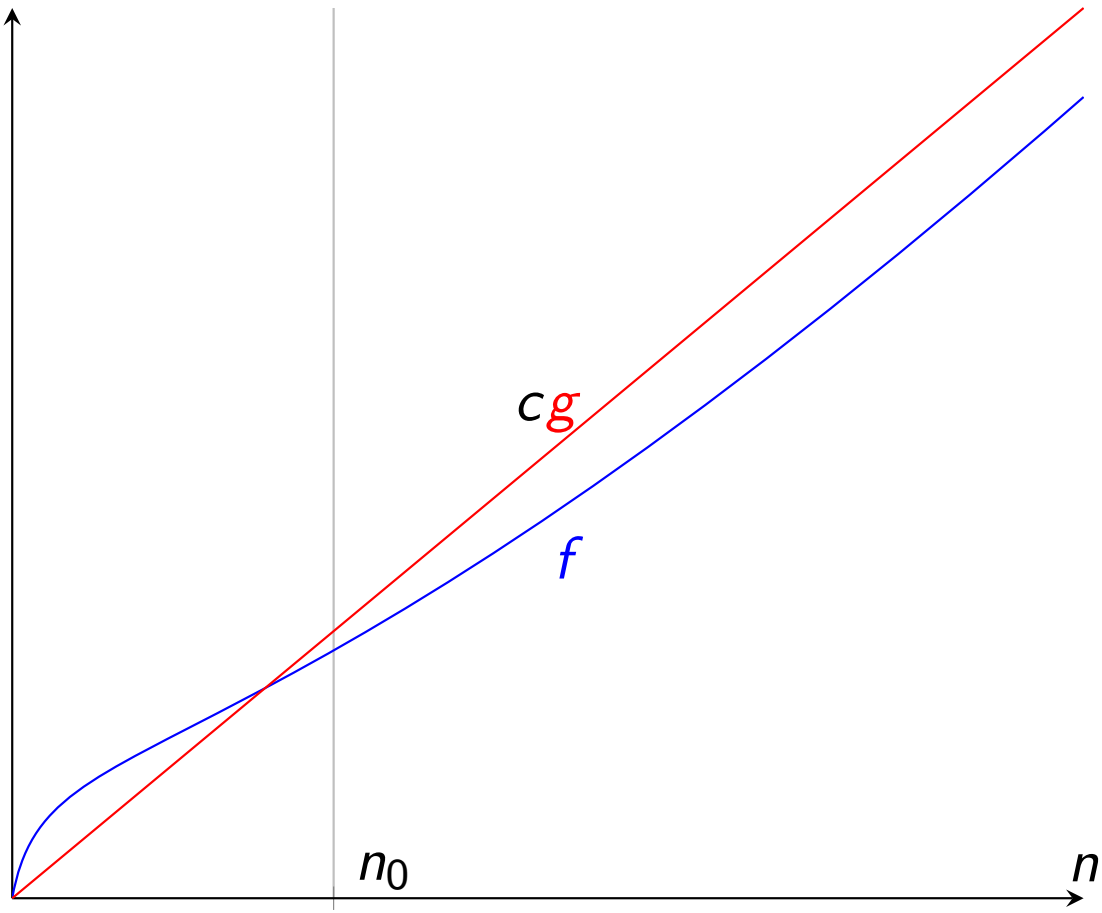
- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

Przykład: który algorytm jest szybszy?

- Porównajmy inne dwa algorytmy, których złożoności czasowe to kolejno $f_1(n) = n$ i $f_2(n) = n^2$.
- Czy można powiedzieć, że pierwszy algorytm jest szybszy?
- Załóżmy, że jego operacja dominująca wykonuje się c razy wolniej (dla pewnej stałej $c > 1$, np. $c = 1000$),
- albo że jest on uruchamiamy na c razy wolniejszym komputerze od drugiego
- i w związku z tym czasy wykonania algorytmów wynoszą odpowiednio cn oraz n^2 jednostek czasu (np. sekund).
- Pomimo „spowolnienia" pierwszego algorytmu c razy, wykona się on szybciej od drugiego, gdy tylko dane wejściowe będą dostatecznie duże (konkretnie gdy $n > c$).
- Tym razem pierwszy algorytm jest więc istotnie szybszy od drugiego, przynajmniej w sensie który sformalizujemy dalej.

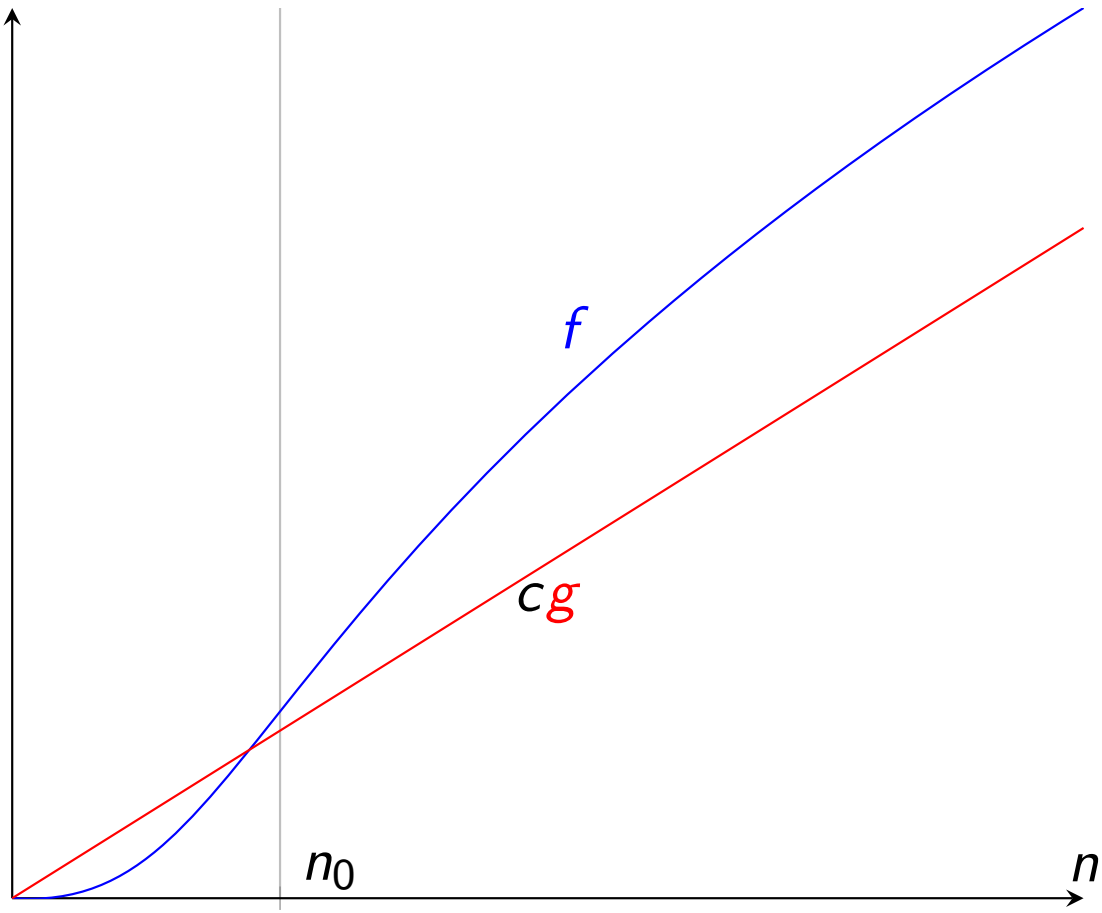
Definicja notacji O („jest co najwyżej rzędu”)

Niech $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ – funkcje. Mówimy, że f **jest co najwyżej rzędu** g, co zapisujemy $f(n) = O(g(n))$, gdy istnieją takie stałe $n_0 \in \mathbb{N}$, oraz $c > 0$, że $\forall n \geq n_0 : f(n) \leq cg(n)$.



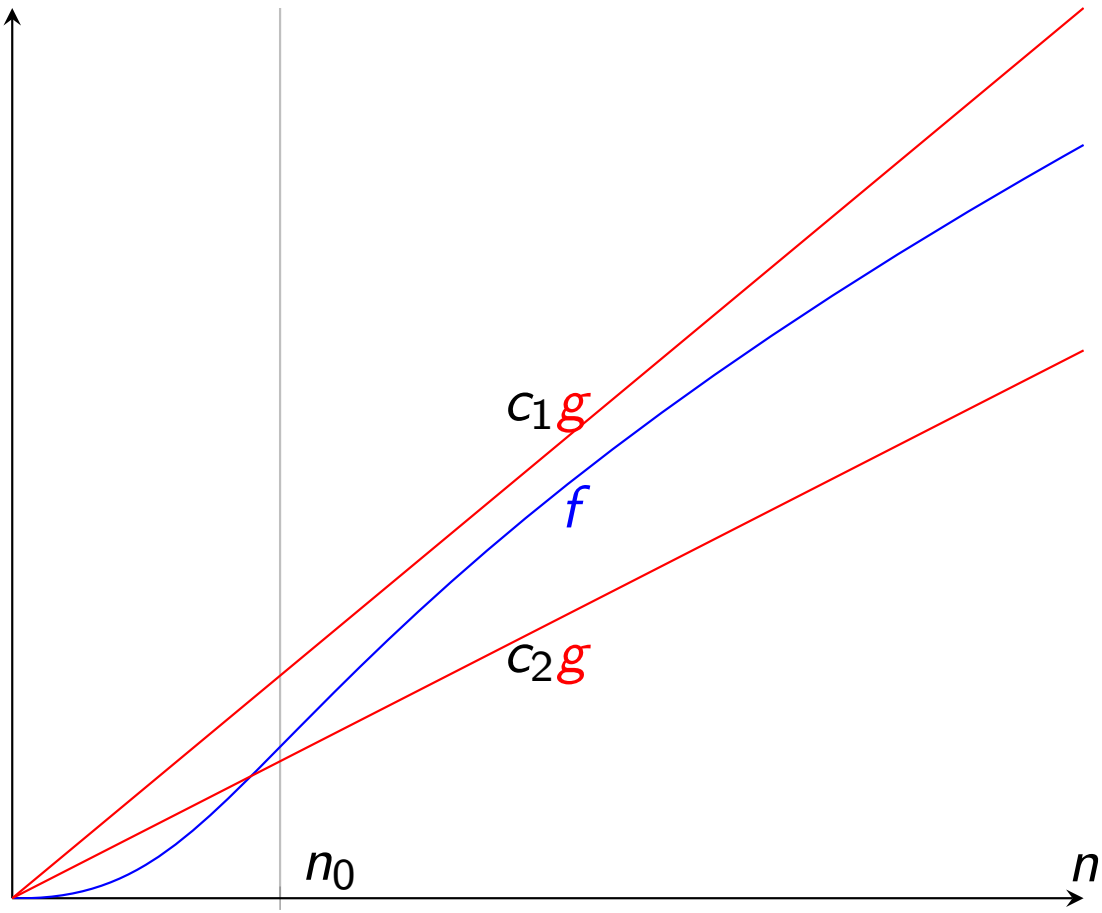
Definicja notacji Ω („jest co najmniej rzędu”)

Niech $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ – funkcje. Mówimy, że f **jest co najmniej rzędu** g, co zapisujemy $f(n) = \Omega(g(n))$, gdy istnieją takie stałe $n_0 \in \mathbb{N}$, oraz $c > 0$, że $\forall n \geq n_0 : f(n) \geq cg(n)$.



Definicja notacji Θ („jest dokładnie rzędu”)

Niech $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ – funkcje. Mówimy, że f **jest dokładnie rzędu** g, co zapisujemy $f(n) = \Theta(g(n))$, gdy $f(n) = O(g(n))$ i równocześnie $f(n) = \Omega(g(n))$.



Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Intuicje stojące za notacjami O, Ω, Θ

- O, Ω, Θ opisują asymptotyczne tempo wzrostu funkcji,
- określają jak szybko rośnie funkcja wraz ze wzrostem jej argumentu, ukrywając przy tym pewne szczegóły,
- opisują zachowanie funkcji dla dużych argumentów, $n \to +\infty$.
- Zapisy te mówią, że f w stosunku do g rośnie asymptotycznie:
  - $f(n) = O(g(n))$ – nie szybciej,
  - $f(n) = \Omega(g(n))$ – nie wolniej,
  - $f(n) = \Theta(g(n))$ – równie szybko.
- Jeśli $f(n) = O(g(n))$ i równocześnie nie zachodzi $f(n) = \Theta(g(n))$, to g rośnie asymptotycznie szybciej od f,
- wtedy algorytm o złożoności czasowej f, nawet uruchomiony na wielokrotnie wolniejszym komputerze, wykona się szybciej od g, gdy tylko dane wejściowe będą dostatecznie duże.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany tego samego stopnia

Niech $f(n) = 5n^3 + 10n + 4$. Pokażemy że $f(n) = \Theta(n^3)$.

Najpierw dowiedziemy, że $f(n) = O(n^3)$.
By to uczynić, wskażemy stałą c taką, że $f(n) \leq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \leq 5n^3 + 10n^3 + 4n^3 = 19n^3$, to wystarczy położyć $c = 19$.

Teraz dowiedziemy, że $f(n) = \Omega(n^3)$.
W tym celu wystarczy wskazać stałą c taką, że $f(n) \geq cn^3$ dla wszystkich $n \geq 1$.
Ponieważ $5n^3 + 10n + 4 \geq 5n^3$, to wystarczy położyć $c = 5$.

Ponieważ $f(n) = O(n^3)$ i $f(n) = \Omega(n^3)$, to $f(n) = \Theta(n^3)$.

Analogicznie można pokazać, że **każdy wielomian $w(n)$ stopnia s o dodatnim współczynniku przy najwyższej potędze, jest dokładnie rzędu n^s (tj. $w(n) = \Theta(n^s)$)**.

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Przykład: wielomiany różnych stopni

Pokażemy że $5n^2 = O(n^3)$ i równocześnie, że nie prawdą jest $5n^2 = \Omega(n^3)$ (czyli nie jest prawdą także $5n^2 = \Theta(n^3)$).

$5n^2 = O(n^3)$ bo dla $c = 5$ i wszystkich $n \geq 0$ mamy $5n^2 \leq cn^3$.

Dowiedziemy, że nie prawdą jest $5n^2 = \Omega(n^3)$:

- (przez zaprzeczenie) załóżmy że $5n^2 = \Omega(n^3)$ jest prawdą,
- czyli że istnieją stałe $n_0 \in \mathbb{N}$ oraz $c > 0$, takie że:
- $5n^2 \geq cn^3$ dla wszystkich $n \geq n_0$.
- Wtedy jednak, dla wszystkich $n \geq n_0$ musiałoby zachodzić $5 \geq cn$, co nie może być prawdą, gdyż cn dąży do nieskończoności gdy $n \to \infty$
- ($5 \geq cn$ nie zachodzi dla żadnego $n > \frac{5}{c}$).

Logarytmy

- Dla dowolnych $a > 1$, $b > 1$, zachodzi $\log_a(n) = \Theta(log_b(n))$.
- Powyższy fakt wynika wprost ze wzoru na zmianę podstawy logarytmu

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = c \log_b(n)$$

  dla stałej $c = 1/\log_b(a)$.
- Inaczej mówiąc, wszystkie logarytmy (o podstawie większej niż 1) mają takie same asymptotyczne tempo wzrostu.
- Dlatego w asymptotycznych notacjach (O, Ω, Θ) pomija się w zapisie podstawy logarytmów, np. $log_2(n) = \Theta(log(n))$.

Logarytmy

- Dla dowolnych $a > 1$, $b > 1$, zachodzi $\log_a(n) = \Theta(log_b(n))$.
- Powyższy fakt wynika wprost ze wzoru na zmianę podstawy logarytmu

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = c \log_b(n)$$

  dla stałej $c = 1/\log_b(a)$.
- Inaczej mówiąc, wszystkie logarytmy (o podstawie większej niż 1) mają takie same asymptotyczne tempo wzrostu.
- Dlatego w asymptotycznych notacjach (O, Ω, Θ) pomija się w zapisie podstawy logarytmów, np. $log_2(n) = \Theta(log(n))$.

Logarytmy

- Dla dowolnych $a > 1$, $b > 1$, zachodzi $\log_a(n) = \Theta(log_b(n))$.
- Powyższy fakt wynika wprost ze wzoru na zmianę podstawy logarytmu

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = c \log_b(n)$$

  dla stałej $c = 1/\log_b(a)$.
- Inaczej mówiąc, wszystkie logarytmy (o podstawie większej niż 1) mają takie same asymptotyczne tempo wzrostu.
- Dlatego w asymptotycznych notacjach (O, Ω, Θ) pomija się w zapisie podstawy logarytmów, np. $log_2(n) = \Theta(log(n))$.

Logarytmy

- Dla dowolnych $a > 1$, $b > 1$, zachodzi $\log_a(n) = \Theta(log_b(n))$.
- Powyższy fakt wynika wprost ze wzoru na zmianę podstawy logarytmu

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = c \log_b(n)$$

  dla stałej $c = 1/\log_b(a)$.
- Inaczej mówiąc, wszystkie logarytmy (o podstawie większej niż 1) mają takie same asymptotyczne tempo wzrostu.
- Dlatego w asymptotycznych notacjach (O, Ω, Θ) pomija się w zapisie podstawy logarytmów, np. $log_2(n) = \Theta(log(n))$.

Wybrane funkcje wypisane w takiej kolejności, że każda kolejna jest co najmniej (ale nie dokładnie) rzędu poprzedniej:

funkcja	nazwy powiązanej złożoności obliczeniowej i przykłady algorytmów o złożoności czasowej takiego rzędu
1	stała; niezależna od wielkości danych wejściowych
$\log(n)$	logarytmiczna; np. wyszukiwanie binarne
$\sqrt{n}$	
n	liniowa; np. wyszukiwanie w nieuporządkowanej tablicy
$n\log(n)$	liniowo-logarytmiczna; np. minimalna złożoność czasowa sortowania przez porównywanie
n^2	kwadratowa; np. naiwne algorytmy sortowania
n^3	sześcienna
2^n	wykładnicza (o podstawie 2)
3^n	wykładnicza (o podstawie 3)
$n!$	

Ogólnie funkcje (oraz złożoności) postaci n^c (dla $c > 0$) nazywamy wielomianowymi (mówimy: algorytm wielomianowy; działa w czasie wielomianowym), zaś c^n (dla $c > 1$) wykładniczymi.

Bibliografia

- Thomas H. Cormen, Charles E. Leiserson, Roland L. Rivest, Clifford Stein *Wprowadzenie do algorytmów*
- L. Banachowski, K. Diks, W. Rytter *Algorytmy i struktury danych*, Wydawnictwa Naukowo-Techniczne, 2006.
- K. Diks, A. Malinowski, W. Rytter, T. Waleń *Algorytmy i struktury danych/Wstęp: poprawność i złożoność algorytmu*, WWW: http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Wst%C4%99p:_poprawno%C5%9B%C4%87_i_z%C5%82o%C5%BCono%C5%9B%C4%87_algorytmu
- M. Sydow *Algorytmy i Struktury Danych*, prezentacje:
  - *Poprawność Algorytmów*, WWW: http://users.pja.edu.pl/~msyd/wyka-pl/correctness1-pl.pdf
  - *Złożoność Obliczeniowa Algorytmów*, WWW: http://users.pja.edu.pl/~msyd/wyka-pl/complexity2-pl.pdf