Algorytmy i Struktury Danych

Złożoność Obliczeniowa Algorytmów

(c) Marcin Sydow

Pożądane cechy dobrego algorytmu

Dobry algorytm mający rozwiązywać jakiś problem powinien mieć 2 naturalne cechy:

1. (poprawność) zwracać **prawidłowy wynik** (dokładniej: zgodność z warunkiem końcowym specyfikacji algorytmu)
2. (szybkość) być jak najbardziej **wydajnym** (dokładniej: osiąganie wyniku przy możliwie minimalnym nakładzie zasobów: ilości pracy (w efekcie: czasu działania) i ilości używanej pamięci)

Pierwsza cecha odpowiada pojęciu **poprawności całkowitej** (poprzedni wykład)

Druga cecha odpowiada pojęciu złożoności obliczeniowej algorytmu (definicja podana będzie w niniejszym wykładzie)

Niższa **złożoność algorytmu** oznacza wydajniejszy algorytm.

Przykład: problem wyszukiwania klucza w tablicy

Specyfikacja:

`find(arr, len, key)`

wejście: arr - tablica liczb całkowitych, len - długość tablicy, key - liczba całkowita, której szukamy w tablicy
wyjście: liczba naturalna $i < len$ będąca wartością indeksu, pod którym w tablicy arr występuje szukany klucz

Przykład: problem wyszukiwania klucza w tablicy

Specyfikacja:

`find(arr, len, key)`

wejście: arr - tablica liczb całkowitych, len - długość tablicy, key - liczba całkowita, której szukamy w tablicy
wyjście: liczba naturalna $i < len$ będąca wartością indeksu, pod którym w tablicy arr występuje szukany klucz (czy ta specyfikacja jest pełna?)

Przykład: problem wyszukiwania klucza w tablicy

Specyfikacja:

`find(arr, len, key)`

wejście: arr - tablica liczb całkowitych, len - długość tablicy, key - liczba całkowita, której szukamy w tablicy
wyjście: liczba naturalna $i < len$ będąca wartością indeksu, pod którym w tablicy arr występuje szukany klucz (czy ta specyfikacja jest pełna?)
nie: nie wiadomo co zrobić jeśli nie ma klucza w tablicy. Uzupełnienie:

Przykład: problem wyszukiwania klucza w tablicy

Specyfikacja:

`find(arr, len, key)`

wejście: arr - tablica liczb całkowitych, len - długość tablicy, key - liczba całkowita, której szukamy w tablicy
wyjście: liczba naturalna $i < len$ będąca wartością indeksu, pod którym w tablicy arr występuje szukany klucz (czy ta specyfikacja jest pełna?)
nie: nie wiadomo co zrobić jeśli nie ma klucza w tablicy. Uzupełnienie:
lub wartość -1 jeśli pośród len pierwszych elementów nie ma klucza

Przykład: problem wyszukiwania klucza w tablicy

Specyfikacja:

`find(arr, len, key)`

wejście: arr - tablica liczb całkowitych, len - długość tablicy, key - liczba całkowita, której szukamy w tablicy
wyjście: liczba naturalna $i < len$ będąca wartością indeksu, pod którym w tablicy arr występuje szukany klucz (czy ta specyfikacja jest pełna?)
nie: nie wiadomo co zrobić jeśli nie ma klucza w tablicy. Uzupełnienie: **lub** wartość -1 jeśli pośród len pierwszych elementów nie ma klucza
pseudokod:

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Od czego zależy ilość pracy tego algorytmu?

"Szybkość" algorytmu

Jak mierzyć jak **"szybki"** jest algorytm ?

Wybierając miarę szybkości algorytmu należy mieć na uwadze 2 względy:

1. niezależność od języka programowania/platformy
2. możliwie duża niezależność od danych wejściowych

Pomysł?

"Szybkość" algorytmu

Jak mierzyć jak **"szybki"** jest algorytm ?

Wybierając miarę szybkości algorytmu należy mieć na uwadze 2 względy:

1. niezależność od języka programowania/platformy
2. możliwie duża niezależność od danych wejściowych

Pomysł?
Zliczać podstawowe **operacje** wykonywane przez algorytm

Operacje dominujące

Nie trzeba zliczać wszystkich operacji wykonywanych przez algorytm. Wystarczy zliczać operacje dominujące czyli te, które proporcjonalnie pokrywają całą pracę algorytmu.

Zbiór **operacji dominujących** danego algorytmu to zbiór takich operacji, których liczba jest proporcjonalna do liczby wszystkich operacji wykonanych przez cały algorytm.

Operacją dominującą nie jest więc np. operacja wykonywana tylko jednokrotnie. Natomiast każda pętla lub odgałęzienie instrukcji warunkowej powinna zawierać operację dominującą.

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie

porównanie `i < len` ?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie

porównanie `i < len` ? tak

porównanie `arr[i] == key` ?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie

porównanie `i < len` ? tak

porównanie `arr[i] == key` ? tak

obie naraz?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie

porównanie `i < len` ? tak

porównanie `arr[i] == key` ? tak

obie naraz? tak (choć nie jest to konieczne)

instrukcja `return i` ?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie
porównanie `i < len` ? tak
porównanie `arr[i] == key` ? tak
obie naraz? tak (choć nie jest to konieczne)
instrukcja `return i` ? nie (bo wykonywana najwyżej raz)
zwiększanie indeksu `i++` ?

Przykład: wyznaczanie operacji dominujących

Co może być **operacją dominującą w poniższym algorytmie?**

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

przypisanie `i = 0` ? nie
porównanie `i < len` ? tak
porównanie `arr[i] == key` ? tak
obie naraz? tak (choć nie jest to konieczne)
instrukcja `return i` ? nie (bo wykonywana najwyżej raz)
zwiększanie indeksu `i++` ? tak

Dwa kroki niezbędne do wykonania analizy złożoności

Zanim wykona się analizę złożoności danego algorytmu **należy wykonać 2 następujące kroki**:

1. wyznaczyć zbiór operacji dominujących
2. wyznaczyć funkcję argumentów (danych wejściowych), która stanowi rozmiar danych wejściowych

ad 1) zostało już omówione
ad 2) zostanie wyjaśnione w dalszej części wykładu

Złożoność czasowa algorytmu

Definition

Złożoność czasowa algorytmu to liczba operacji dominujących jakie wykona algorytm **jako funkcja rozmiaru danych**.

Złożoność czasowa algorytmu mierzy "jak wiele pracy" musi wykonać algorytm realizujący zadanie **w zależności** od tego jak duże są dane.

Niższa złożoność algorytmu oznacza "szybszy" algorytm i odwrotnie.

Przykład: wyznaczanie rozmiaru danych

Co stanowi **rozmiar danych** (wpływa na ilość pracy) w poniższym algorytmie?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Przykład: wyznaczanie rozmiaru danych

Co stanowi **rozmiar danych** (wpływa na ilość pracy) w poniższym algorytmie?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Rozmiar danych stanowi tu: **długość tablicy** arr

Przykład: wyznaczanie rozmiaru danych

Co stanowi **rozmiar danych** (wpływa na ilość pracy) w poniższym algorytmie?

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Rozmiar danych stanowi tu: **długość tablicy** arr

Operacją dominującą może być np. porównanie `arr[i] == key`

Wyznaczywszy rozmiar danych i operację dominującą, można wyznaczyć złożoność czasową tego algorytmu

Przykład - złożoność czasowa algorytmu

Dokonajmy analizy złożoności czasowej poniższego algorytmu:

```
find(arr, len, key){
  i = 0
  while(i < len){
    if(arr[i] == key)
      return i
    i++
  }
  return -1
}
```

Wykonujemy 2 kroki:

1. wyznaczamy **operację dominującą**: porównanie `arr[i] == key`
2. wyznaczamy co stanowi **rozmiar danych**: długość tablicy

Wtedy, zgodnie z definicją złożoności czasowej, liczba operacji dominujących wykonanych przez algorytm jako funkcja rozmiaru danych jest w zakresie:

- od **1**: kiedy klucz jest na pierwszej pozycji tablicy (wariant "optymistyczny")
- do **len**: kiedy klucz jest na końcu lub nieobecny (warant "pesymistyczny")

Widzimy więc pewien problem: nie ma pojedynczej funkcji, tylko ich zakres.

Warianty złożoności czasowej

Skoro więc liczba wykonanych operacji dominujących nawet dla ustalonego rozmiaru danych może się wahać w pewnym zakresie (nie ma pojedynczej funkcji, która to wyraża tylko ich zakres), wprowadza się kilka wariantów:

- **pesymistyczna złożoność czasowa**, oznaczana $W()$ (od ang. "worst"). Jest to funkcja wyrażająca kres górny możliwej liczby operacji dominujących dla ustalonego rozmiaru danych (wariant pesymistyczny)
- **przeciętna złożoność czasowa**, oznacza przez $A()$ (od ang. "average").Jest to funkcja wyrażająca *przeciętną* liczbę wykonanych operacji dominujących przy założeniu pewnego **modelu losowości danych** wejściowych

Są to najbardziej podstawowe warianty (w niektórych zagadnieniach rozważa się też np. amortyzowaną złożoność czasową, której definicji nie podajemy w tym wykładzie)

Formalna definicja pesymistycznej złożoności czasowej

Załóżmy następujące oznaczenia:
n - rozmiar danych
D_n - zbiór wszystkich możliwych danych wejściowych o rozmiarze n
$t(d)$ - liczba wykonanych operacji dominujących dla pewnych konkretnych danych wejściowych $d \in D_n$ o rozmiarze n

Definition

Pesymistyczna złożoność czasowa algorytmu:
$W(n) = sup\{t(d) : d \in D_n\}$

(W(n) - **W**orst)

Pesymistyczna złożoność czasowa wyraża liczbę wykonanych operacji dominujących w najgorszym (pesymistycznym) przypadku.

Ile wynosi pesymistyczna złożoność czasowa w naszym przykładzie? (czyli: jaka funkcją ją wyraża?)
$W(n) =$

Formalna definicja pesymistycznej złożoności czasowej

Załóżmy następujące oznaczenia:
n - rozmiar danych
D_n - zbiór wszystkich możliwych danych wejściowych o rozmiarze n
$t(d)$ - liczba wykonanych operacji dominujących dla pewnych konkretnych danych wejściowych $d \in D_n$ o rozmiarze n

Definition

Pesymistyczna złożoność czasowa algorytmu:
$W(n) = sup\{t(d) : d \in D_n\}$

(W(n) - **W**orst)

Pesymistyczna złożoność czasowa wyraża liczbę wykonanych operacji dominujących w najgorszym (pesymistycznym) przypadku.

Ile wynosi pesymistyczna złożoność czasowa w naszym przykładzie? (czyli: jaka funkcją ją wyraża?)
$W(n) = n$

Przeciętna złożoność czasowa algorytmu

załóżmy następujące oznaczenia:
n - rozmiar danych
X_n - zmienna losowa oznaczająca faktyczną liczbę wykonanych operacji dominujących dla losowych danych o rozmiarze n
p_{nk} - prawdopodobieństwo, że liczba wykonanych operacji dominujących wyniesie k dla losowych danych o rozmiarze n

Definition

Przeciętna złożoność czasowa algorytmu:
$A(n) = \sum_{k \geq 0} p_{nk} \cdot k = \sum P(X_n = k) \cdot k = E(X_n)$
Jest to wartość oczekiwana zmiennej losowej X_n

(A(n) **A**verage)

Uwaga: dla obliczenia przeciętnej złożoności czasowej niezbędne jest założenie pewnego modelu losowości danych, wyrażonego przez rozkład prawdopodobieństwa p_{nk}.

Przykład: wyznaczanie przeciętnej złożoności algorytmu

Wróćmy do naszego roboczego przykładu.

Najpierw należy założyć konkretny **model losowości danych**. Konkretnie sprowadza się to do założenia ile wynosi p_{nk} - prawdopodobieństwo, że wykonane będzie k operacji dominujących dla danych rozmiaru n.
Założymy tu najprostszy wariant, że wszystkie wartości k dla ustalonego rozmiaru danych są równe i wynoszą $1/n$. Jest to podobne do założenia, że klucz może być na każdym miejscu tablicy z takim samym prawdopodobieństwem:
$\forall_{0 \leq k < n} p_{nk} = P(X_n = k) = 1/n$

Otrzymujemy wtedy z definicji:
$A(n) = \sum_{k \geq 0} P(X_n = k) \cdot k = \sum_{0 \leq k < n} 1/n \cdot k = \frac{n+1}{2}$
Jest to zgodne z intuicją: klucz będzie wtedy "przeciętnie w połowie tablicy"

Pamięciowa złożoność algorytmu

Inną (obok szybkości działania) miarą efektywności algorytmu jest złożoność pamięciowa algorytmu. Jest ona zdefiniowana analogicznie do złożoności czasowej, tylko zamiast liczby operacji dominujących zliczamy liczbę jednostek pamięci.

Definition

Pamięciowa złożoność algorytmu: **S(n)** jest to liczba jednostek pamięci głównej użyta przez algorytm jako funkcja rozmiaru danych

Jak zauważymy później, po wprowadzeniu tzw. notacji asymptotycznej dla złożoności, nie będzie istotne o jakiej konkretnie jednostce pamięci mówimy (bit, bajt, MB, GB, etc.) gdyż będzie nas interesowało raczej tempo wzrostu zużycia pamięci przy wzroście rozmiaru danych wejściowych a nie konkretna ilość pamięci.

Możemy rozważać zarówno pesymistyczną jak i przeciętną złożoność pamięciową (SW(n), SA(n)).

Specjalnym przypadkiem jest, gdy algorytm zużywa zawsze *stałą* ilość pamięci niezależnie od danych wejściowych. W takim przypadku oznaczamy $S(n) = const$ lub $S(n) = O(1)$. Jest to dosyć częsty przypadek.

Pomijanie nieistotnych detali

Ostateczna szybkość **konkretnej implementacji** danego algorytmu zależy zawsze od użytego języka programowania i sprzętu, na którym jest realizowany. Np. 2 razy wolniejszy algorytm zaimplementowany na 6 razy szybszym sprzęcie będzie w sumie 3 razy szybszy! A interesuje nas szybkość samego algorytmu.
Dlatego obiektem zainteresowania podstawowej analizy algorytmów jest raczej **charakter tempa wzrostu** ilości operacji dominujących jako funkcja rosnącego rozmiaru danych (czyli rodzaj tej funkcji) niż konkretna funkcja.

Pomijanie nieistotnych detali, c.d.

Przy takim podejściu można pominąć nieistotne detale takie jak np. **stałe multiplikatywne i addytywne**.
W tym celu, do wyrażania funkcji złożoności algorytmów używana jest notacja asymptotyczna. Pozwala ona na uproszczenie zapisu przez pominięcie w/w nieistotnych szczegółów.

Np. dla poniższej funkcji złożoności czasowej:
$A(n) = 3.45 \cdot n + 2$
Nieistotny jest składnik "+2". Co więcej, jeśli interesuje nas tylko charakter tempa wzrostu (czyli **funkcja liniowa**), a nie konkretna stała (3.45) to użyta notacja powinna pozwolić pominąć te szczegóły.

Taką notacją jest **Notacja asymptotyczna**

Notacja asymptotyczna - "duże o"

Jest kilka wariantów tej notacji, najczęściej używanym jest wariant "duże o" (czytane jako litera alfabetu a nie jak cyfra zero)
Notacja "$O()$" służy do oznaczania **górnego ograniczenia** na tempo wzrostu danej funkcji.
Przykładowe użycie: "$3.45 \cdot n + 2 = O(n)$", co należy interpretować: funkcja "$3.45 \cdot n + 2 = O(n)$" ma charakter tempa wzrostu conajwyżej liniowy.

Definition

$f(n) = O(g(n)) \Leftrightarrow \exists_{c>0} \exists_{n_0} \forall_{n \geq n_0} f(n) \leq c \cdot g(n)$
Funkcja g(n) jest **ograniczeniem górnym rzędu wielkości** funkcji f(n).

Notacja O() dla rzędów wielkości funkcji odpowiada intuicyjnie symbolowi $\leq$ dla liczb.

Przykłady

Przeciętna złożoność czasowa z naszego przykładu jest liniowa: $A(n) = \frac{n+1}{2} = O(n)$

Przeciętna złożoność pamięciowa z naszego przykładu jest stała: $SA(n) = O(1)$ (od: "Space Average")

Przykład wyższego rzędu funkcji: $f(n) = 3n^2 + 2n - 7$ (rząd: kwadratowy)

Wtedy z definicji $f(n) = O(n)$ nie będzie prawdą, bo funkcja f ma wyższy (kwadratowy) rząd wzrostu (niż liniowy).
Można więc napisać $f(n) = O(n^2)$.
Można też napisać $f(n) = O(n^3)$, chociaż jest to mniej dokładne ograniczenie górne niż $O(n^2)$.

Ćwiczenie kontrolne: sprawdzić, że dla każdego z powyższych przypadków wynika to bezpośrednio z definicji matematycznej tej notacji (możliwość znalezienia lub nie odpowiednich stałych c i n_0)

Notacja asymptotyczna: $\Theta()$ ("duże teta")

Ważnym wariantem notacji asymptotycznej jest notacja $\Theta()$. Służy ona do wyrażania faktu, że funkcja ma **dokładnie** taki sam rząd wielkości jak inna funkcja. Odpowiada to więc znakowi "=" dla rzędów wielkości funkcyj. Jest to równoważne temu, że g jest zarazem ograniczeniem górnym f jak i na odwrót.

Definition

$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge g(n) = O(f(n))$ Funkcja f(n) ma **taki sam rząd wielkości** jak funkcja g(n)

Przykład: $f(n) = n^2 + n - 3 = \Theta(n^2)$
czyli: "funkcja $n^2 + n - 3$" ma **dokładnie kwadratowy rząd wielkości**.
Zauważmy, że pozostałe składniki: "n" oraz "-3" są zdominowane, przez składnik n^2. Do wyzaczania dokładnego rzędu funkcji będącej sumą wystarczy skupić się na **dominującym** składniku tej sumy.

Inne warianty notacji asymptotycznej

Tak jak występuje 5 symboli do porównywania wielkości liczb: $= \leq \geq < >$, stosowane jest też 5 wariantów notacji asymptotycznej.

(Uwaga: ta analogia nie jest jednak pełna, gdyż porządek na liczbach jest porządkiem liniowym)

1. Θ - "$=$" (równe rzędy wielkości funkcji)
2. O - "$\leq$" (ograniczenie górne nieostre rzędu wielkości)
3. Ω (wielkie "omega") - "$\geq$"(ograniczenie dolne nieostre)
4. o - "$<$" (ograniczenie górne ostre)
5. ω (małe "omega") - "$>$" (ograniczenie dolne ostre)

(Wielkie litery dotyczą ograniczeń z równością (nieostrych), małe ostrych)

Przykład:
W(n)=o(n) ("małe o")

znaczy: "rząd złożoności pesymistycznej jest istotnie mniejszy niż liniowy"

Uwagi odnośnie użycia notacji asymptotycznej

Uwaga: w notacji asymptotycznej, np. "f(n)=O(g(n))" symbol "=" nie ma zwykłego znaczenia "równości", jest to tylko rodzaj notacji. Wobec tego notacji takiej nie można traktować jako zwykłej równości i np. odejmować od obu stron symbolu "O(n)", etc. W szczególności notacji asymptotycznej używamy głównie po prawej stronie znaku "=".
Np. notacja: "O(f(n)) = n" lub "O(f(n)) = O(g(n))" **nie ma sensu**.

Można natomiast użyć notacji asymptotycznej w nieco rozszerzonej formie np.:
$f(n) = g(n) + O(h(n))$
co oznacza: f(n) - g(n) = O(h(n)) (czyli, że rzędy wielkości funkcyj f i g różnią się co najwyżej o składnik liniowy)

Alternatywna definicja notacji asymptotycznej

W niektórych zastosowaniach (np. porównywania rzędów wielkości dwóch funkcji) ma zastosowanie inna wersja definicji. Nie używa ona kwantyfikatorów lecz pojęcia przejścia granicznego z analizy matematycznej (ale można udowodnić, że jest matematycznie równoważna):

Obliczamy wtedy granicę ilorazu dwóch funkcyj f i g:
(zakładamy, że obie funkcje mają wartości dodatnie)
$lim_{n\to\infty}\frac{f(n)}{g(n)}$

jeśli granica istnieje, to możliwe są 3 przypadki jej wartości:

1. ∞ - w tym przypadku f ma **wyższy** rząd wielkości od g: f(n)=$\omega(g(n))$
2. stała dodatnia - w tym przypadku rzędy wielkości f i g są **równe**: f(n)=$\Theta(g(n))$
3. zero - w tym przypadku f ma **niższy** rząd wielkości niż g: f(n)=o(g(n))

Najczęściej spotykane rzędy wielkości

Poniżej wymienione są często spotykane rzędy złożoności algorytmów:

- **stała** (np. $S(n) = 3 = \Theta(1)$)
- **logarytmiczna** (np. $W(n) = 2 + lg_2 n = \Theta(\log(n))$)
- **liniowa** (np. $A(n) = 2n + 1 = \Theta(n)$)
- **liniowo-logarytmiczna** (np. $A(n) = 1.44 \cdot n \log(n) = \Theta(n \log(n))$)
- **kwadratowa** (e.g. $W(n) = 3n^2 + 4 = \Theta(n^2)$)
- **sześcienna** (e.g. $A(n) = \Theta(n^3)$)
- pod-wykładniczna (e.g. $A(n) = \Theta(n^{\log(n)})$)
- **wykładnicza** (e.g. $A(n) = \Theta(2^n)$)
- silniowa (e.g. $W(n) = \Theta(n!)$)

Algorytmy o złożoności czasowej wyższej niż wielomianowa uważane są za niepraktyczne poza przypadkami danych wejściowych o małym rozmiarze.

Praktyczne reguły przy ustalaniu rzędu wielkości funkcji

- Jeśli funkcja jest w postaci sumy kilku czynników, to rząd wielkości wyznaczony jest przez dominujący składnik.
- Każdy wielomian ma dokładnie taki rząd wielkości jaki wyznaczony jest przez składnik o najwyższym stopniu
- Każda funkcja potęgowa (n^α gdzie $\alpha \in R$) ma inny rząd, zależny od wykładnika α.
- Jest tylko jeden rząd logarytmiczny (tzn. wszystkie logarytmy mają ten sam rząd wielości niezależnie od podstawy logarytmu)
- Rząd logarytmiczny $log_\beta n$ jest ostro niższy od dowolnej funkcji potęgowej n^α (nawet dla np. $\alpha = 0.0001$, etc.)
- każda funkcja wykładnicza γ^n ma inny rząd w zależności od podstawy γ
- każda funkcja wykładnicza γ^n (dla dowolnego $\gamma \in R^+$) ma rząd ostro większy od każdej funkcji potęgowej n^α.

Pytania/Zadania kontrolne

- czym mierzymy "szybkość" algorytmu?
- jakie 2 kroki muszą być wykonane zanim przystąpimy do analizy złożoności czasowej algorytmu?
- Znajomość na pamięć definicji i umiejętność wyznaczenia dla danego algorytmu:
  - operacji dominującej
  - rozmiaru danych
  - złożoności czasowej pesymistycznej
  - złożoności czasowej przeciętnej (na razie tylko dla bardzo prostych algorytmów)
  - złożoności pamięciowej
- jaki jest cel notacji asymptotycznej?
- definicja (na pamięć) i interpretacja notacji asymptotycznej w 5 wariantach
- umieć udowodnić z definicji, że dane wyrażenie zawierające notację asymptotyczną jest prawdziwe lub fałszywe

Dziękuję za uwagę