WOJSKOWA AKADEMIA TECHNICZNA
Wydział Cybernetyki

Algorytmy i struktury danych

Rok akademicki 2017/2018

Prowadzący: dr hab. inż. Kazimierz Worwa, prof. WAT

e-mail: kazimierz.worwa@wat.edu.pl

ALGORYTMY I STRUKTURY DANYCH

RAZEM	Wykłady	Ćwiczenia	Laboratoria
60x	24	16+	20+

Warunki zaliczenia przedmiotu

- Rygory:
  - zaliczenie ćwiczeń rachunkowych,
  - zaliczenie ćwiczeń laboratoryjnych,
  - egzamin.
- Warunkiem dopuszczenia do egzaminu jest zaliczenie ćwiczeń rachunkowych i laboratoryjnych
- Egzamin – pisemny test wielokrotnego wyboru

LITERATURA

1. **Aho A.V., Hopcroft J.E., Ullman J.D.: *Algorytmy i struktury danych*. Wydawnictwo Helion, Gliwice, 2003.**
2. **Aho A.V., Hopcroft J.E., Ullman J.D.: *Projektowanie i analiza algorytmów*. Wydawnictwo Helion, Gliwice, 2003.**
3. **Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*. WNT, Warszawa, 2011.**
4. **Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.: *Wprowadzenie do algorytmów*. PWN, Warszawa, 2017.**
5. **Drozdek A.: *C++. Algorytmy i struktury danych*. Wydawnictwo Helion, Gliwice, 2004.**
6. **Kotowski P.: *Algorytmy + struktury danych = abstrakcyjne typy danych*. Wydawnictwo BTC, Warszawa, 2006.**
7. **Harris S, Ross J.: *Algorytmy od podstaw*. Wydawnictwo Helion, Gliwice, 2006.**
8. **Wróblewski P.: *Algorytmy, struktury danych i techniki programowania*. Wydawnictwo Helion, Gliwice, 2015.**
9. **Wirth N.: *Algorytmy + struktury danych = programy*. WNT, Warszawa, 2004.**

Tematyka przedmiotu

- Złożoność obliczeniowa algorytmów
- Listy
- Kolejki (LIFO, FIFO, kolejki priorytetowe)
- Drzewa binarne
  - drzewa BST
  - drzewa AVL
  - drzewa czerwono-czarne
  - drzewa częściowo uporządkowane (kopce)
- Drzewa wielokierunkowe (B-drzewa)
- Algorytmy sortowania
  - wewnętrznego
  - zewnętrznego
- Tablice rozproszone (haszowanie)
- Problemy obliczeniowo trudne

Materiały do wykładów

- Materiały wykorzystywane na wykładach udostępniam na stronie https://cszis-elearning.wat.edu.pl:8443/WPB/
- logowanie:
  - nazwa użytkownika **AiSD**
  - Hasło **!Przedmiot3**
- Pobranie (lub wyświetlenie) pliku następuje po kliknięciu na link do pliku
- Jeżeli link jest nieaktywny, należy w opcjach przeglądarki dopuścić możliwość wyskakujących okienek.

Algorytmy i struktury danych

Wykład 1-2

Pojęcia podstawowe.
Złożoność obliczeniowa algorytmów.

Tematyka wykładu 1-2

- Pojęcia podstawowe
- Klasyfikacja algorytmów
- Własności algorytmów
- Metody oceny algorytmów
- Pojęcie złożoności obliczeniowej algorytmów
- Złożoność asymptotyczna: O-notacja, Ω-notacja, Θ-notacja
- Wyznaczanie złożoności czasowej algorytmów:
  - iteracyjnych
  - rekurencyjnych

Potoczne rozumienie pojęcia „algorytm”

- *Algorytmika* jest dziedziną wiedzy zajmującą się badaniem algorytmów
- Potocznie algorytm jest rozumiany jako pewien przepis na wykonanie zestawu czynności, prowadzących do osiągnięcia oczekiwanego, z góry określonego celu
- W informatyce algorytm jest pewną ściśle określoną procedurą obliczeniową, która dla zestawu właściwych danych wejściowych generuje określone dane wyjściowe
- Dzisiejsze, uogólnione znaczenie słowa ***algorytm***:

  Opis postępowania, umożliwiający rozwiązanie określonego problemu w skończonej liczbie kroków (w skończonym czasie)

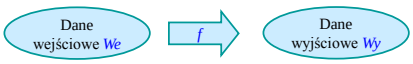
Pochodzenie słowa „algorytm”

- Słowo *„algorism”* zostało utworzone od nazwiska perskiego matematyka z IX wieku n.e., Muhameda ibu-Musy al-Choresmi, twórcy dziesiętnego systemu liczbowego
- Słowo „algorytm” pochodzi od łacińskiego słowa *„algorism”*, rozumianego w średniowieczu jako sztuka rachowania na liczbach w systemie dziesiętnym

Pojęciowy model algorytmu

Algorytm może być rozumiany jako pewne odwzorowanie ***f***, które dla określonego zestawu danych wejściowych ***We*** generuje określony zestaw danych wyjściowych ***Wy***:

$$f:\ We \longrightarrow Wy$$



Sposoby zapisu algorytmów

- Opis algorytmu obejmuje precyzyjny opis jego kolejnych kroków
- Do przedstawienia algorytmu stosuje się:
  - opis werbalny,
  - zapis formalny, np.:
    - zapisy graficzne (schematy blokowe),
    - formalne specyfikacje programów (np.VDM)
    - zapisy w postaci pseudokodu, z wykorzystaniem tzw. metajęzyka
    - zapis algorytmu w dowolnym *języku programowania*

Język programowania

- **Język programowania** jest środkiem umożliwiającym zapis algorytmów w postaci zrozumiałej dla człowieka, a równocześnie przetwarzalnej do postaci zrozumiałej dla komputera.
- Zapis algorytmu w języku programowania jest traktowany jako zapis formalny.
- Program komputerowy może być uznawany za jeden z rodzajów modeli matematycznych.

Klasyfikacja algorytmów

Rozważać będziemy następujące klasy algorytmów:

- algorytmy sekwencyjne - algorytmy równoległe;
- algorytmy numeryczne - algorytmy nienumeryczne;
- algorytmy rekurencyjne - algorytmy iteracyjne.

Techniki algorytmiczne

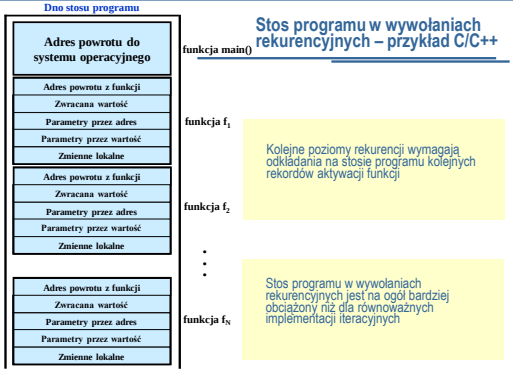
Podstawowe techniki wykorzystywane w budowie algorytmów:

- Iteracja
- Rekurencja

Rekurencja

- **Rekurencja** oznacza wywołanie funkcji (procedury) przez tę samą funkcję (procedurę)
- Niepożądana cecha definicji rekurencyjnych: aby wyznaczyć n-tą wartość trzeba najpierw wyznaczyć wszystkie „wcześniejsze" wartości
- Ważne jest, aby kolejne wywołania funkcji (procedury) rekurencyjnej były realizowane dla kolejnych wartości parametrów formalnych w taki sposób, aby nie doszło do zjawiska „nieskończonej pętli rekurencyjnych wywołań funkcji"

Stos programu w wywołaniach rekurencyjnych – przykład C/C++



Funkcja rekurencyjna – ciąg Fibonacciego

- n-ty wyraz ciągu Fibonacciego jest wyliczany wg formuły, n≥0:

$$Fib(n) = \begin{cases} n & \text{dla } n<2 \\ Fib(n-2) + Fib(n-1) & \text{dla } n>=2 \end{cases}$$

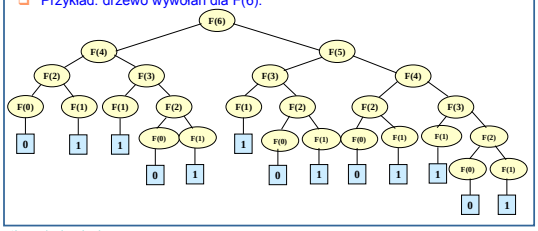
- Rekurencyjna implementacja w języku C:

```
long int Fib (int n)
{
  if (n<2)
    return n;
  else
    return Fib(n-2) + Fib (n-1);
}
```

Dla dużych wartości *n* stos programu prawdopodobnie „nie wytrzyma" takiej realizacji funkcji *Fib*

Efektywność rekurencyjnego wykonania funkcji Fibonacciego (cd.)

- Rekurencyjna implementacja funkcji Fibonacciego jest bardzo nieefektywna.
- Stos programu nie jest praktycznie w stanie zrealizować tego algorytmu już dla liczb większych od 50.
- Oznacza to, że program ma zbyt dużą „złożoność pamięciową".
- Przykład: drzewo wywołań dla F(6):



Efektywność rekurencyjnego wykonania funkcji Fibonacciego

n	Liczba dodawań	Liczba wywołań
6	12	25
10	88	177
15	986	1 973
20	10 945	21 891
25	121 392	242 785
30	1 346 268	2 692 537

Iteracyjne wykonanie rekurencyjnej funkcji Fibonacciego

- Bardziej efektywna jest iteracyjna implementacja funkcji Fibonacciego. Nie przepełniamy wtedy stosu programu i wykonujemy znacznie mniejszą liczbę przypisań wartości niż w implementacji rekurencyjnej
- Przykład iteracyjnej implementacji funkcji Fibonacciego:

```
long int IteracyjnyFib(int n)
{
   int i=2, last=0, tmp;  long int current =1;
   if (n<2)
     return n;
   else {
        for (  ; i<=n; i++)  {
            tmp = current;
            current += last;
            last = tmp;
        }
    return current;
   }
}
```


Efektywność iteracyjnego wykonania rekurencyjnej funkcji Fibonacciego

n	Liczba przypisań w algorytmie iteracyjnym	Liczba przypisań (wywołań) w algorytmie rekurencyjnym
6	15	25
10	27	177
15	42	1 973
20	57	21 891
25	72	242 785
30	87	2 692 537

Program 1.1b | Program 1.1a

Pułapki rekurencji

Co będzie wynikiem wykonania poniższej funkcji?

```
int jakas_funkcja(int n)
{
  if  (n==1)
     return 1;
  else
    if ((n%2)==0)       // czy n jest parzyste?
      return jakas_funkcja(n-2)*n;
    else
      return jakas_funkcja(n-1)*n;
}
```

Program 1.2

Jak porównywać programy (algorytmy)?

Idealny program to taki, który:

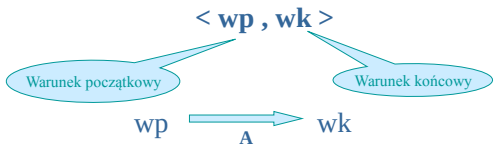
- ma czytelny i zrozumiały kod,
- jest napisany w ogólnie dostępnym języku programowania,
- jest efektywny obliczeniowo (szybko liczy, nie wymaga dużej pamięci),
- zawsze daje poprawne wyniki.

Jak porównywać programy – przykładowe kryteria

- prostota algorytmu,
- czytelność kodu,
- długość kodu,
- poprawność wyników,
- czas realizacji (obliczeń),
- zajętość pamięci.

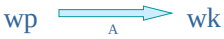
Częściowa poprawność algorytmu

- Specyfikacją algorytmu nazywamy zbiór par warunków (własności):



Algorytm *A* wykorzystujący strukturę danych *S* jest częściowo poprawny ze względu na specyfikację <*wp, wk*>, jeżeli dla wszystkich danych spełniających warunek początkowy i dla których algorytm zatrzyma się, uzyskane wyniki spełniają warunek końcowy.

Całkowita poprawność algorytmu



Mówimy, że algorytm *A* wykorzystujący strukturę danych *S* jest całkowicie poprawny ze względu na specyfikację <*wp, wk*> jeżeli dla wszystkich danych w strukturze *S* spełniających warunek początkowy *wp*, algorytm zatrzymuje się i daje wyniki spełniające warunek końcowy *wk*.

Złożoność obliczeniowa algorytmów

- **Złożoność obliczeniowa** - miara służąca do porównywania efektywności obliczeniowej algorytmów.
- Mamy dwa zasadnicze kryteria efektywności obliczeniowej: czas i pamięć, stąd mówimy o złożoności czasowej i pamięciowej.

Rodzaje złożoności obliczeniowej algorytmów

- **Złożoność pamięciowa** - wyrażana w skali zajętości pamięci (PAO, pamięci zewnętrznej), niezbędnej dla realizacji algorytmu
- **Złożoność czasowa** - wyrażająca liczbę kroków (np. liczbę wymaganych iteracji) wykonania algorytmu (rzadziej w skali czasu)
- Na ogół (obecnie) w ocenie złożoności obliczeniowej algorytmów złożoność czasowa jest istotniejsza od złożoności pamięciowej

Czynniki wpływające na czas wykonania programu

- Rozmiar danych wejściowych algorytmu
- Jakość kodu wynikowego generowanego przez kompilator (język kodu źródłowego)
- Architektura komputera, w tym szybkość procesora, na którym program jest wykonywany
- Efektywność wykorzystanego algorytmu (jego złożoność obliczeniowa, zazwyczaj głównie czasowa)

Złożoność czasowa algorytmu

Definicja

Miarą złożoności czasowej jest liczba operacji podstawowych (dominujących), wykonywanych przez algorytm w czasie jego realizacji, wyrażona jako funkcja rozmiaru danych.

Oznaczenie

Złożoność czasowa algorytmu *A* jako funkcja rozmiaru danych *n*:

$T(A, n)$ lub $T(n)$

Złożoność czasowa algorytmu

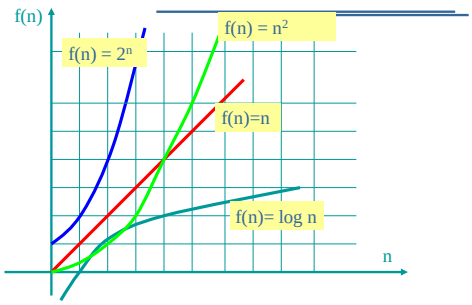
- Do oszacowania czasu realizacji algorytmu nie powinno się używać zwykłych jednostek czasu
- Zamiast tego powinny być stosowane jednostki logiczne, określające związek między rozmiarem danych wejściowych ***n*** a czasem ***t*** potrzebnym na ich przetworzenie
- Funkcje opisujące związek między *T(n)* a ***n*** mogą być bardzo złożone; w praktyce upraszczamy je, pozostawiając tzw. składowe dominujące, tj. składowe mające największy wpływ na wartości *T(n)*

Przykład

- Niech zależność czasu realizacji algorytmu od rozmiaru danych wejściowych opisuje funkcja

$f(n) = n^2 + 100n + \log n + 1000$

Porównanie szybkości wzrostu funkcji



$f(n) = n^2 + 100n + \log n + 1000$

Szybkość wzrostu poszczególnych składników funkcji

Funkcja: $f(n) = n^2 + 100*n + \log_{10} n + 1000$

n	f(n)	n^2	100*n	$\log_{10} n$	1000
1	1 101	0.1%	9%	0.0 %	91%
10	2 101	4.8%	48%	0.05%	48%
100	21 002	48%	48%	0.001%	4.8%
10^3	1 101 003	91%	9%	0.0003%	0.09%
10^4		99%	1%	0.0%	0.001%
10^5		99,9%	0.1%	0.0%	0.0000%

Dla dużych wartości ***n*** składnikiem dominującym jest $\mathbf{n^2}$, tzn. funkcja rośnie jak $\mathbf{n^2}$; pozostałe składniki mogą być pominięte;
n – liczba wykonywanych operacji

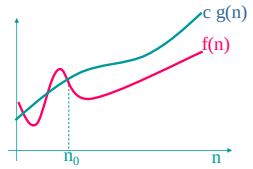
Asymptotyczna złożoność obliczeniowa

- Funkcja wyrażająca zależność miedzy ***n*** a ***T*** jest zwykle bardzo skomplikowana, a jej dokładne obliczenie ma znaczenie jedynie w odniesieniu do specyficznych problemów
- Przybliżoną miarą efektywności najczęściej stosowaną w praktyce jest tzw. **asymptotyczna złożoność czasowa**, tzn. złożoność czasowa wyznaczona dla odpowiednio dużych (ale skończonych) wartości *n*
- Asymptotyczna złożoność czasowa wyrażana jest w oparciu o specjalne notacje
- W praktyce najczęściej wykorzystuje się O-notację, Ω-notację oraz Θ-notację

O-notacja

- Definicja:
  Funkcja *f(n)* jest funkcją o złożoności *O(g(n))*, jeżeli istnieją takie liczby dodatnie *c* i n_o , że dla każdego $n \geq n_0$ zachodzi
  $$f(n) \leq c\, g(n)$$
- Zgodnie z powyższą definicją, związek między funkcjami *f* i *g* można wyrazić stwierdzając, że *g(n)* jest kresem górnym dla *f(n)*
- *Uwaga*: powyższa definicja nie daje żadnych wskazówek co do tego, jak wyznaczyć stałe *c* i n_0 (takich par stałych może być wiele)
- Interpretacja:
  O(g(n)) = { f(n): istnieją dodatnie liczby c i n_0 takie, że dla każdego $n \geq n_0$ zachodzi $0 \leq f(n) \leq c\, g(n)$ }
- Piszemy: $f(n) \in O(g(n))$ lub częściej $f(n) = O(g(n))$

O-notacja – ilustracja graficzna



f(n)=O(g(n))

Przykład

- Niech zależność czasu realizacji algorytmu od rozmiaru danych wejściowych *n* opisuje funkcja
  $$f(n) = n^2 + 100n + \log n + 1000$$
- Wówczas, wykorzystując O-notację, można napisać, że
  $$n^2 + 100n + \log n + 1000 \in O(n^2)$$
- Zatem, dla badanej funkcji **$T(n) = O(n^2)$**
- Tak określona złożoność czasową algorytmu nazywa się **asymptotyczną złożonością czasową**
- W praktyce wykorzystuje się także pojęcia **optymistycznej**, **pesymistycznej** oraz **średniej** złożoności czasowej algorytmu

Własności O-notacji

Własność 1 (przechodniość)
Jeśli f(n) jest O(g(n)) i g(n) jest O(h(n)), to f(n) jest O(h(n)).

Własność 2:
Jeśli f(n) jest O(h(n)) i g(n) jest O(h(n)), to f(n)+g(n) jest O(h(n)).

Własność 3:
Funkcja an^k jest $O(n^k)$.

Własność 4:
Funkcja n^k jest $O(n^{k+j})$ dla dowolnego nieujemnego j.

Z powyższych własności wynika, że dowolny wielomian jest „wielkie O" dla n podniesionego do najwyższej w nim potęgi, czyli
$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k)$$
(jest też oczywiście $O(n^{k+j})$ dla dowolnego nieujemnego j).

Własności O-notacji (cd.)

Własność 5:
Jeśli f(n)=c g(n), to f(n) jest O(g(n))

Własność 6:
Funkcja $\log_a n$ jest $O(\log_b n)$ dla dowolnych a>1, b>1

Własność 7:
$\log_a n$ jest $O(\log_2 n)$ dla dowolnego dodatniego a

Wniosek:
Wszystkie funkcje logarytmiczne (niezależnie od podstawy logarytmu) są sobie równoważne w sensie O-notacji

Własności O-notacji (cd.)

Własność 8 *(reguła sumy)*

Jeśli $T_1(n)=O(f(n))$ i $T_2(n)=O(g(n))$ są złożonościami czasowymi dwóch fragmentów algorytmu, to łączna złożoność czasowa algorytmu (będącego sekwencją obydwu fragmentów) wynosi:

$$T_1(n) + T_2(n) = O(\max\{ f(n), g(n)\})$$

Własność 9 *(reguła iloczynu)*

Niech $T_1(n)=O(f(n))$ i $T_2(n)=O(g(n))$ są złożonościami czasowymi dwóch fragmentów algorytmu. Wówczas:

$$T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$$

Własności O-notacji

- Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest funkcja logarytmiczna.
- Jeżeli można wykazać, że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry.
- Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak O(log log n) czy O(1) ma praktyczne znaczenie.

Ω - notacja

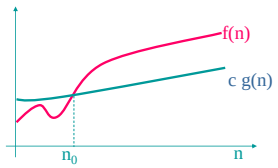
- O-notacja dotyczy kresu górnego funkcji
- Istnieje symetryczna definicja kresu dolnego w postaci Ω-notacji

- *Definicja*:
  f(n) jest funkcją o złożoności $\Omega(g(n))$ jeżeli istnieją takie liczby dodatnie *c* i n_0, że dla każdego $n \geq n_0$ zachodzi
  $$f(n) \geq c\, g(n)$$
- Wniosek
  Funkcja *f(n)* ma złożoność $\Omega(g(n))$ wtedy i tylko wtedy, gdy *g(n)* ma złożoność $O(f(n))$

Ω - notacja

- Interpretacja:
  $\Omega(g(n)) = \{ f(n)$: istnieją dodatnie liczby c i n_0 takie, że dla każdego $n \geq n_0$ zachodzi $0 \leq c\, g(n) \leq f(n) \}$
- Piszemy: $f(n) \in \Omega(g(n))$ lub częściej $f(n) = \Omega(g(n))$

Ω-notacja – ilustracja graficzna



$f(n)=\Omega(g(n))$

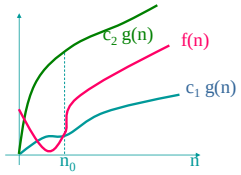
Θ - notacja

- Definicja
  f(n) jest funkcją o złożoności $\Theta(g(n))$ jeżeli istnieją liczby dodatnie c_1, c_2 i n_0 takie, że dla każdego $n \geq n_0$ zachodzi
  $$c_1\, g(n) \leq f(n) \leq c_2\, g(n)$$
- Wniosek
  Funkcja *f(n)* ma złożoność $\Theta(g(n))$ wtedy i tylko wtedy, gdy *f(n)* ma złożoność $O(g(n))$ i *f(n)* ma złożoność $\Omega(g(n))$

Θ - notacja

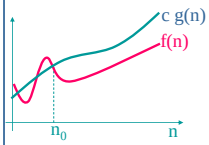
- Interpretacja:
  $\Theta(g(n)) = \{ f(n)$: istnieją dodatnie liczby c_1, c_2 i n_0 takie, że dla każdego $n \geq n_0$ zachodzi $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \}$
- Piszemy: $f(n) \in \Theta(g(n))$ lub częściej $f(n) = \Theta(g(n))$

Θ-notacja – ilustracja graficzna

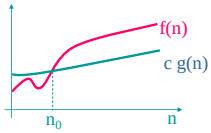


f(n)=Θ(g(n))

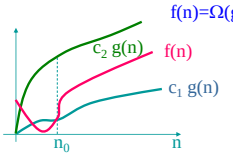
O-notacja, Ω-notacja, Θ-notacja – porównanie



f(n)=O(g(n))



f(n)=Ω(g(n))



f(n)=Θ(g(n))

Notacja asymptotyczna - podsumowanie

Niech $f, g : N \to R^+$.

Mówimy, że funkcja *f* jest co najwyżej rzędu *g*, jeśli $(\exists c>0, \exists n_0 \in N)\ \forall n>n_0 \quad f(n) \le c\ g(n)$ **f = O (g)**

Mówimy, że funkcja *f* jest co najmniej rzędu *g*, jeśli $(\exists c>0, \exists n_0 \in N)\ \forall n>n_0 \quad c\ g(n) \le f(n)$ **f = Ω (g)**

Mówimy, że funkcja *f* jest rzędu *g*, jeśli $(\exists c_1>0, c_2>0, \exists n_0 \in N)\ \forall n>n_0 \quad c_1\ g(n) \le f(n) \le c_2\ g(n)$ **f = Θ (g)**

Porównanie szybkości wzrostu funkcji

Mówimy, że algorytm A ma złożoność czasową:

- ❑ logarytmiczną jeśli $T(A, n) = \Theta(\log n)$
- ❑ wielomianową jeśli $T(A, n) = \Theta(n^\alpha)$ $\alpha \in N$
  - liniową jeśli $T(A, n) = \Theta(n)$
  - kwadratową jeśli $T(A, n) = \Theta(n^2)$
- ❑ wykładniczą jeśli $T(A, n) = \Theta(a^n)$ $a \in R^+$

Porównywanie rzędów funkcji

Przykład 1 Niech $f(n)=100n$, $g(n)= 2n+100$, $h(n) = 0.1\ n^2 + n$

Mamy: $f = O(n)$, $f = \Omega(n)$, $g = O(n)$ ale także $g = O(n^2)$, $g = \Theta(n)$
$h = O(n^2)$, $h \neq O(n)$, $h = \Omega(n)$

Lemat (o porównywaniu rzędów funkcji)
Niech $\lim_{n\to\infty} [f(n)/g(n)] = c$. Wówczas:
1. Jeżeli $c \neq 0$ to f i g są tego samego rzędu.
2. Jeżeli $c = 0$, to $f = O(g)$ oraz $f \neq \Omega(g)$.
3. Jeżeli $c = +\infty$, to f ma rząd większy niż g, $g = O(f)$ i $g \neq \Omega(f)$.

Przykład 2
$f(n) = 0,3\ n^3 + 10n + 100$
$g(n)= n^3$
$h(n) = \log n$

$\lim_{n\to\infty} f(n)/g(n)= 0,3$
Zatem $f = g = \Theta(n^3)$

$\lim_{n\to\infty} f(n)/h(n) = +\infty$
Zatem $h = O(f) = O(n^3)$, $h \neq \Omega(f)$.

Złożoność a rozmiar i czas

Ile czasu potrzeba na rozwiązanie zadania o ustalonym rozmiarze i złożoności?

wymiar \ T(A,n)	lg n	n	n lg n	n^2	n^3	2^n
$n=10^2$	6.6 μs	0.1 ms	0.7 ms	10 ms	1s	10^6 lat
$n= 10^4$	13.3 μs	10 ms	0.1 s	100 s	11dni	10^{100} lat

Jaki jest maksymalny rozmiar problemu *n*, który można rozwiązać w ustalonym czasie, znając złożoność algorytmu?

czas \ T(A,n)	lg n	n	n lg n	n^2	n^3	2^n
1 s	$2^{1000000}$	10^6	$63*10^3$	10^3	10^2	19
1 godz.		$36*10^8$	$13*10^7$	$60*10^3$	$15*10^2$	31

Czy szybkość może przezwyciężyć złożoność?

- Mamy 5 algorytmów A_1, A_2, A_3, A_4, A_5 rozwiązujących ten sam problem. Niech s_i oznacza maksymalny rozmiar problemu, który można rozwiązać na *komputerze 1* przy pomocy algorytmu A_i w ustalonym czasie *t*.
- Jaki jest maksymalny rozmiar problemu, który można rozwiązać w tym samym czasie *t* na *komputerze 2*, który jest 10 razy szybszy?

	lg n	n	n^2	n^3	2^n
Komputer 1	s_1	s_2	s_3	s_4	s_5
Komputer 2	s_1^{10}	$10*s_2$	$3{,}16*s_3$	$2{,}15*s_4$	$3{,}32 + s_5$

Dla *komputera 1*: $T(A, s_5) = 2^{s_5} \rightarrow t$

Dla *komputera 2*: $T(A, s_5) = 2^{s_5} \rightarrow t/10$

Szukamy takiego x, że $T(A, x) = 2^x \rightarrow t$

Zatem: $t = 10 \cdot 2^{s_5}$, $t = 2^x$, $2^x = 10 \cdot 2^{s_5}$

czyli $x = 3{,}32 + s_5$

MS Excel

Wpływ dziesięciokrotnego przyspieszenia komputera na wzrost rozmiaru zadania

Jaki jest maksymalny rozmiar problemu, który można rozwiązać w tym samym czasie *t* na *komputerze 2*, który jest 10 razy szybszy?

Złożoność czasowa *T(n)*	Komputer 1	Komputer 2 (dziesięciokrotnie szybszy)	Względny przyrost rozmiaru problemu
100n	10	100	10,0
$5n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3

Wyznaczanie złożoności czasowej - przykłady

Przykład 1

Pojedyncza pętla wyznaczająca sumę elementów wektora

```
for (i=sum=0; i<n; i++)
        sum=sum+ a[i];
```

- Liczba przypisań w całej pętli: ***2+2n***
- Złożoność czasowa: ***T(n)=O(n)***

Wyznaczanie złożoności czasowej - przykłady

Przykład 2

Pętla podwójna wyznaczająca sumę elementów tablicy

```
for (i=0; i<n; i++) {
    for (j=1, sum=a[0]; j<=i; j++)
        sum=sum+ a[j];
    printf(„\n Suma podtablicy %d" , sum)
}
```

- Liczba przypisań w całej pętli:

$$1 + 3n + 2\sum_{i=0}^{n-1} i = 1 + 3n + 2(1 + 2 + \ldots + n - 1) = 1 + 3n + n(n-1) = O(n^2)$$

- Złożoność czasowa: ***T(n)=O(n²)***

Wyznaczanie złożoności czasowej - przykłady

Przykład 3

Zerowanie elementów tablicy leżących na i pod główną przekątną

```
int tab[n][n];
void zerowanie();
{
int i,j;
i=0;                    // 1
while (i<n)             // 1
 {
    j=0;                // 1
    while (j<=i)        // 1
    {
        tab[i][j]=0;    // 1
        j=j+1;          // 1
    }
    i=i+1;              // 1
 }
}
```

Złożoność czasowa (z uwagi na wszystkie instrukcje):

$$T(n) = 1 + \sum_{i=0}^{n-1}(3 + \sum_{j=0}^{i} 3) = 1 + \sum_{i=0}^{n-1}(3 + 3(i+1)) =$$

$$= 1 + 3n + 3\frac{n(n+1)}{2} = O(n^2)$$

Złożoność czasowa algorytmów rekurencyjnych

- Kiedy algorytm zawiera *rekurencyjne wywołanie samego siebie*, złożoność czasową jego działania można często opisać zależnością rekurencyjną (rekurencją), wyrażającą tę złożoność dla problemu rozmiaru *n* za pomocą złożoności czasowej podproblemów mniejszych rozmiarów.
- Możemy więc użyć narzędzi matematycznych, aby rozwiązać rekurencję i w ten sposób otrzymać oszacowanie złożoności czasowej algorytmu.

Rekurencja dla algorytmu typu “dziel i zwyciężaj”

- Rekurencja odpowiadającą czasowi działania algorytmu typu “dziel i zwyciężaj” opiera się na podziale jednego poziomu rekurencji na trzy etapy
- Niech **T(n)** będzie czasem działania algorytmu dla problemu rozmiaru **n**
- Jeśli rozmiar problemu jest odpowiednio mały, powiedzmy **$n \le c$** dla pewnej stałej **c**, to jego rozwiązanie zajmuje stały czas, co zapisujemy jako **$\Theta(1)$**
- Załóżmy że dzielimy problem rozmiaru **n** na **a** podproblemów, każdy rozmiaru **n/b**
- Jeśli **D(n)** jest czasem dzielenia problemu na podproblemy, a **C(n)** czasem scalania rozwiązań poszczególnych podproblemów w pełne rozwiązanie problemu wyjściowego, to otrzymujemy rekurencję

$$T(n) = \begin{cases} \Theta(1) & \text{dla } n \le c \\ aT(n/b) + D(n) + C(n) & \text{dla } n > c \end{cases}$$

Rekurencja dla algorytmu typu “dziel i zwyciężaj”

Przykład: algorytm sortowania przez scalanie

- znajdujemy środek przedziału, zajmuje to czas stały D(n)=Θ(1)
- rozwiązujemy rekurencyjnie dwa podproblemy, każdy rozmiaru n/2, co daje czas działania 2 T(n/2)
- łączymy dwa uporządkowane podciągi w jeden ciąg (uporządkowany) w czasie Θ(n), a więc C(n)= Θ(n).
- Ostatecznie

$$T(n) = \begin{cases} \Theta(1) & \text{dla } n = 1 \\ 2T(n/2) + \Theta(1) + \Theta(n) & \text{dla } n > 1 \end{cases}$$

- Można pokazać, że rozwiązaniem tej rekurencji jest **$T(n) = \Theta(n \log n)$**

Wyznaczanie złożoności czasowej algorytmów rekurencyjnych

Metody rozwiązywania rekurencji:

1. **Metoda podstawiania**: zgadujemy oszacowanie, a następnie wykorzystujemy indukcję matematyczną.
2. **Metoda iteracyjna**: przekształcamy rekurencję na sumę (korzystamy z technik ograniczania sum).
3. **Metoda drzewa rekursji**: uzupełniająca metodę podstawiania
4. **Metoda rekurencji uniwersalnej**: stosujemy oszacowanie na rekurencje mające postać

$$T(n) = a\,T(n/b) + f(n),$$

gdzie $a \ge 1$, $b > 1$, a $f(n)$ jest daną funkcją.

Wyznaczanie złożoności czasowej algorytmów rekurencyjnych

Metoda podstawiania:

- Polega na odgadnięciu postaci rozwiązania, a następnie wykazaniu przez indukcję, że jest ono poprawne.
- Trzeba także znaleźć wartości odpowiednich stałych.
- Metoda jest bardzo skuteczna, ale może być stosowana tylko w przypadkach, kiedy można przewidzieć postać rozwiązania.

Metoda podstawiania

Oznaczenia:
$\lfloor a \rfloor$ - największa liczba całkowita x taka, że x<=a
$\lceil a \rceil$ - najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **$T(n) = 2T(\lfloor n/2 \rfloor) + n$**

Metoda podstawiania

Oznaczenia:
$\lfloor a \rfloor$ - największa liczba całkowita x taka, że x<=a
$\lceil a \rceil$ - najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **$T(n) = 2T(\lfloor n/2 \rfloor) + n$**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...

Metoda podstawiania

Oznaczenia:
⌊a⌋- największa liczba całkowita x taka, że x<=a
⌈a⌉- najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **T(n) = 2T(⌊n/2⌋) + n**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...
- Przewidywane rozwiązanie: **T(n) = O(n lg n),** tzn. $\exists c > 0$ $T(n) \leq c\, n \lg n$

Metoda podstawiania

Oznaczenia:
⌊a⌋- największa liczba całkowita x taka, że x<=a
⌈a⌉- najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **T(n) = 2T(⌊n/2⌋) + n**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...
- Przewidywane rozwiązanie: **T(n) = O(n lg n),** tzn. $\exists c > 0$ $T(n) \leq c\, n \lg n$
- Założenie: dla n = ⌊n/2⌋ zachodzi: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$

Metoda podstawiania

Oznaczenia:
⌊a⌋- największa liczba całkowita x taka, że x<=a
⌈a⌉- najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **T(n) = 2T(⌊n/2⌋) + n**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...
- Przewidywane rozwiązanie: **T(n) = O(n lg n),** tzn. $\exists c > 0$ $T(n) \leq c\, n \lg n$
- Założenie: dla n = ⌊n/2⌋ zachodzi: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$
- Indukcja: $T(n) \leq 2[c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)] + n \leq \mathbf{c\, n \lg(n/2) + n} =$

$= c\, n \lg n - c\, n \lg 2 + n = c\, n \lg n - cn + n$

Metoda podstawiania

Oznaczenia:
⌊a⌋- największa liczba całkowita x taka, że x<=a
⌈a⌉- najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **T(n) = 2T(⌊n/2⌋) + n**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...
- Przewidywane rozwiązanie: **T(n) = O(n lg n),** tzn. $\exists c > 0$ $T(n) \leq c\, n \lg n$
- Założenie: dla n = ⌊n/2⌋ zachodzi: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$
- Indukcja: $T(n) \leq 2[c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)] + n \leq \mathbf{c\, n \lg(n/2) + n} =$

$= c\, n \lg n - c\, n \lg 2 + n = c\, n \lg n - cn + n$

- Jeśli $c \geq 1$ zachodzi: $c\, n \lg n - cn + n \leq c\, n \lg n$ **Dlaczego?**
  czyli $\mathbf{T(n) \leq c\, n \lg n}$

Metoda podstawiania

Oznaczenia:
⌊a⌋- największa liczba całkowita x taka, że x<=a
⌈a⌉- najmniejsza liczba całkowita x taka, że x>=a

Przykład:

- Postać rekurencji: **T(n) = 2T(⌊n/2⌋) + n**
- Zachodzi: T(1)=1; T(2)=4; T(3)=5, T(4)=12 ...
- Przewidywane rozwiązanie: **T(n) = O(n lg n),** tzn. $\exists c > 0$ $T(n) \leq c\, n \lg n$
- Założenie: dla n = ⌊n/2⌋ zachodzi: $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$
- Indukcja: $T(n) \leq 2[c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)] + n \leq \mathbf{c\, n \lg(n/2) + n} =$

$= c\, n \lg n - c\, n \lg 2 + n = c\, n \lg n - cn + n$

- Jeśli $c \geq 1$ zachodzi: $c\, n \lg n - cn + n \leq c\, n \lg n$ **Dlaczego?**
  czyli $\mathbf{T(n) \leq c\, n \lg n}$
- Rozwiązanie **T(n) = O(n lg n)** jest poprawne dla $c \geq 2$ i $n \geq 2$ **Dlaczego?**

Metoda iteracyjna

Metoda iteracyjna

- Polega na rozwijaniu (iterowaniu) rekurencji i wyrażanie jej jako sumy składników zależnych tylko od warunków brzegowych.
- Następnie mogą być użyte techniki sumowania do oszacowania rozwiązania.

Wyznaczanie złożoności czasowej algorytmów rekurencyjnych

Rozwinięcie rekurencji

- Jest uproszczoną wersją metody iteracyjnej
- Polega na:
  - rozpisaniu równania rekurencyjnego dla kolejnych wartości **n,**
  - dodaniu otrzymanych równań stronami,
  - zredukowaniu jednakowych wyrazów i przekształceniu otrzymanej zależności tak, aby uzyskać jawną zależność funkcji **T(n)** od **n**
- Metoda jest skuteczna jedynie w odniesieniu do niektórych postaci rekurencji

Metoda iteracyjna – rozwinięcie rekurencji

Przykład 1. Funkcja Silnia

```
int Silnia(int n) {
if (n==0)
   return 1;
else
   return n*Silnia(n-1);
}
```

Czas wykonania jednej instrukcji porównania wartości ; ogólnie: **Θ(1)**

Złożoność czasowa:

$$T(n)=\begin{cases}1 & \text{dla } n=0\\ T(n-1)+1 & \text{dla } n\geq 1\end{cases}$$

Metoda iteracyjna - rozwinięcie rekurencji

Przykład 1 (cd.)

$$T(n)=\begin{cases}1 & \text{dla } n=0\\ T(n-1)+1 & \text{dla } n\geq 1\end{cases}$$

$$T(n)=T(n-1)+1$$
$$T(n-1)=T(n-2)+1$$
$$\dots$$
$$T(1)=T(0)+1$$
$$T(0)=1$$

dodajemy stronami

$$T(n)+T(n-1)+\dots+T(1)+T(0)=n+1+T(n-1)+\dots+T(1)+T(0)$$

Zatem:

T(n)= n+1=O(n)

Metoda iteracyjna

Przykład 2:

- Postać rekurencji: **T(n) = 3T(n/4) + n; T(1)=1**
- Iterujemy: T(n) = n + 3T(n/4) =
  = n + 3[n/4 +3T(n/16)] =
  = n + 3 n/4 + 9T(n/16) =
  = n + 3 n/4 + 9[n/16 + 3T(n/64)] =
  = n + 3 n/4 + 9 n/16 + 27 T(n/64) = ...

$$S_k=a_1\frac{1-q^k}{1-q},\quad q\neq 1$$

$$S_\infty=\frac{a_1}{1-q},\quad q\neq 1$$

- Spostrzeżenie: i-ty składnik ciągu wynosi $(\frac{3}{4})^i$ **n, i=0, 1, 2 ,…**
- Iterowanie kończymy, gdy $\mathbf{n/4^x = 1}$, czyli $\mathbf{x = \log_4 n}$
- Zatem

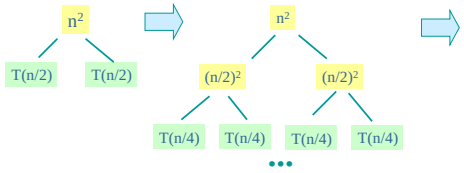
$$T(n)=n+3\frac{n}{4}+3^2\frac{n}{4^2}+3^3\frac{n}{4^3}+\dots+3^{\log_4 n}\frac{n}{4^{\log_4 n}}=n\sum_{k=0}^{\log_4 n}(\frac{3}{4})^k=\Theta(n)$$

Metoda drzewa rekursji

- Drzewo rekursji pozwala w dogodny sposób zilustrować rozwijanie rekurencji, jak również ułatwia stosowanie aparatu algebraicznego, służącego do rozwiązywania tej rekurencji
- Metoda szczególnie użyteczna gdy rekurencja opisuje algorytm typu "dziel i zwyciężaj"
- Każdy węzeł drzewa reprezentuje czas wykonania podproblemu
- **Sumując czasy na kolejnych poziomach drzewa otrzymujemy czas łączny**
- Drzewo rekursji może stanowić pomoc przy odgadywaniu rozwiązania (w metodzie podstawienia)

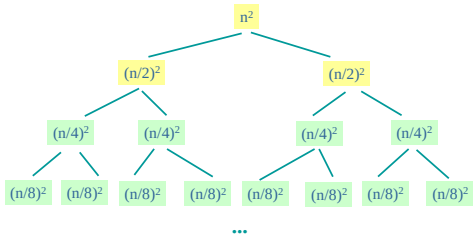
Drzewo rekursji

Przykład 1 **T(n) = 2 T(n/2) + n², T(1)=1**



Drzewo rekursji

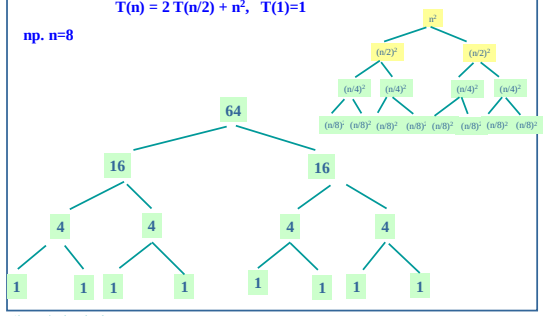
T(n) = 2 T(n/2) + n², T(1)=1



Drzewo rekursji

T(n) = 2 T(n/2) + n², T(1)=1

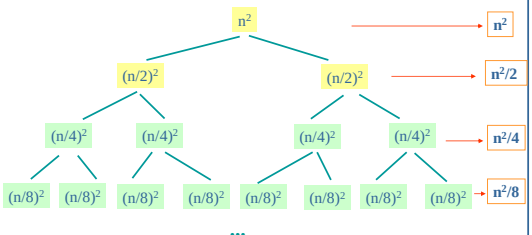
np. n=8



Drzewo rekursji

$$S_k = a_1 \frac{1-q^k}{1-q}, \quad q \neq 1$$

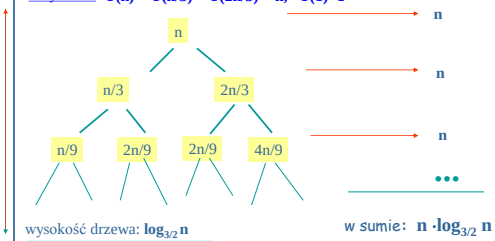
T(n) = 2 T(n/2) + n², T(1)=1



Dlaczego? **w sumie: Θ(n²)**

Drzewo rekursji

Przykład 2 **T(n) = T(n/3) + T(2n/3) + n, T(1)=1**



wysokość drzewa: $\log_{3/2} n$

w sumie: $n \cdot \log_{3/2} n$

Dlaczego?

Zatem, ostateczny wynik:

T(n) = Θ(n log n)

Metoda rekurencji uniwersalnej

- Metoda rekurencji uniwersalnej podaje "uniwersalny przepis" rozwiązywania równania rekurencyjnego postaci

  $T(n) = a\,T(n/b) + f(n)$

  gdzie $a \geq 1$ i $b > 1$ są stałymi, a $f(n)$ jest funkcją asymptotycznie dodatnią.
- Za wartość n/b przyjmujemy najbliższą liczbę całkowitą (mniejszą lub większą od wartości dokładnej) tj. $n/b = \lfloor n/b \rfloor$ lub $n/b = \lceil n/b \rceil$

Metoda rekurencji uniwersalnej

- Rekurencja opisuje czas działania algorytmu, który dzieli problem rozmiaru ***n*** na ***a*** problemów, każdy rozmiaru ***n/b***, gdzie ***a*** i ***b*** są dodatnimi stałymi.
- Każdy z ***a*** problemów składowych jest rozwiązywany rekurencyjnie w czasie ***T(n/b)***.
- Koszt dzielenia problemu oraz łączenia rezultatów częściowych jest opisany funkcją ***f(n)***

Dowód twierdzenia o rekurencji uniwersalnej:
patrz: T.H. Cormen, Ch.E.Leiserson, R.L.Rivest , C Stein: *Wprowadzenie do algorytmów*

Metoda rekurencji uniwersalnej

- Niech $a \geq 1$ i $b > 1$ będą stałymi, niech $f(n)$ będzie pewną funkcją i niech $T(n)$ będzie zdefiniowane rekurencyjnie, dla nieujemnych liczb całkowitych:

  $T(n) = a\,T(n/b) + f(n)$,

  gdzie n/b oznacza najbliższą liczbę naturalną (mniejszą lub większą od wartości dokładnej) tj. $n/b = \lfloor n/b \rfloor$ lub $n/b = \lceil n/b \rceil$
- Wtedy funkcja $T(n)$ może być ograniczona asymptotycznie w następujący sposób:
  1. Jeśli $f(n) = O(n^{\log_b a - \varepsilon})$ dla pewnej stałej $\varepsilon > 0$, to $T(n) = \Theta(n^{\log_b a})$
  2. Jeśli $f(n) = \Theta(n^{\log_b a})$ to $T(n) = \Theta(n^{\log_b a} \log n)$
  3. Jeśli $f(n) = \Omega(n^{\log_b a + \varepsilon})$ dla pewnej stałej $\varepsilon > 0$ i jeśli **$a\,f(n/b) \leq c\,f(n)$** dla pewnej stałej $c < 1$ i wszystkich dostatecznie dużych n, to

     $T(n) = \Theta(f(n))$

     tzw. warunek regularności

Metoda rekurencji uniwersalnej

Interpretacja

- W każdym z trzech przypadków porównujemy funkcję $f(n)$ z funkcją $n^{\log_b a}$
- Rozwiązanie rekurencji zależy od większej z tych dwóch funkcji
- Jeśli funkcja $n^{\log_b a}$ jest wielomianowo większa od $f(n)$ (Przypadek 1), to rozwiązaniem rekurencji jest

  $T(n) = \Theta(n^{\log_b a})$
- Jeśli funkcja $f(n)$ jest wielomianowo większa od $n^{\log_b a}$ (Przypadek 3), to rozwiązaniem jest

  $T(n) = \Theta(f(n))$
- Jeśli funkcje są tego samego rzędu (Przypadek 2), to rozwiązaniem jest

  $T(n) = \Theta(n^{\log_b a} \log n)$, czyli $T(n) = \Theta(f(n) \log n)$

Metoda rekurencji uniwersalnej

Przykład 1

Określić oszacowanie asymptotyczne dla rekurencji:

$T(n) = 9\,T(n/3) + n$

- Mamy: $a=9$, $b=3$, $f(n)=n$, a zatem $n^{\log_b a} = n^{\log_3 9} = n^2$
- Ponieważ $f(n) = O(n^{\log_3 9 - \varepsilon})$, gdzie $\varepsilon = 1$, możemy zastosować przypadek 1 twierdzenia o rekurencji uniwersalnej i wnioskować że rozwiązaniem jest

  $T(n) = \Theta(n^2)$

Metoda rekurencji uniwersalnej

Przykład 2

Określić oszacowanie asymptotyczne dla rekurencji:

$T(n) = T(2n/3) + 1$

- Mamy: $a=1$, $b=3/2$, $f(n)=1$, a zatem $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- Stosujemy przypadek 2, gdyż $f(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1)$ a zatem rozwiązaniem rekurencji jest

  $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_{3/2} 1} \log n) = \Theta(\log n)$

Metoda rekurencji uniwersalnej

Przykład 3

Określić oszacowanie asymptotyczne dla rekurencji:

$T(n) = 3T(n/4) + n \log n$

- Mamy: $a=3$, $b=4$, $f(n)=n \log n$, a zatem $n^{\log_b a} = n^{\log_4 3} = n^{0,793}$
- Ponieważ $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, gdzie $\varepsilon \sim 0.2$, więc stosuje się tutaj przypadek 3, jeśli możemy pokazać, że dla $f(n)$ zachodzi warunek regularności
- Dla dostatecznie dużych n możemy napisać: **$a\,f(n/b) \leq c\,f(n)$**

  $a\,f(n/b) = 3\,(n/4) \log (n/4) \leq (3/4)\, n \log n = c\,f(n)$, przy czym $c = 3/4 < 1$
- Warunek regularności jest zatem jest spełniony i możemy napisać, że rozwiązaniem rekurencji jest **$T(n) = \Theta[f(n)] = \Theta(n \log n)$**

Metoda rekurencji uniwersalnej

Przykład 4

Określić oszacowanie asymptotyczne dla rekurencji:

$T(n) = 2T(n/2) + n \log n$

- Mamy: $a=2$, $b=2$, $f(n)=n \log n$, a zatem $n^{\log_b a} = n$

  Wydaje się, że powinien to być przypadek 3, gdyż $f(n)=n \log n$ jest asymptotycznie większe niż $n^{\log_b a} = n$ (ale nie wielomianowo!)
- Stosunek $f(n)/n^{\log_b a} = n \log n / n$ jest asymptotycznie mniejszy niż n^ε dla każdej dodatniej stałej ε.
- W konsekwencji rekurencja ta "wpada" w lukę między przypadkiem 2 i 3.
- Nie można zatem zastosować twierdzenia o rekurencji uniwersalnej

Dziękuję za uwagę