*Notatki z AiSD. Nr 1.* 22 lutego 2005

# PRELIMINARIA

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

## 1 Literatura

### 1.1 Podstawowa

1. T.H. Cormen, C.E. Leiserson i R.L. Rivest, *Wstęp do Algorytmów*, WNT, 1997 (i kolejne; późniejsze wydania są nieco zmienione)..
2. A.V. Aho, J.E. Hopcroft i J.D. Ullman, *Projektowanie i Analiza Algorytmów Komputerowych*, PWN, 1983 (oraz Helion 2003).
3. L. Banachowski, K. Diks i W. Rytter, *Algorytmy i Struktury Danych*, WNT, 1996.
4. G. Brassard i P. Bratley, *Algorithmics. Theory & Practice.*, Prentice Hall, 1988.

### 1.2 Uzupełniająca

1. D.E. Knuth, *The art of computer programming*, vol. I-III, Addison-Wesley, 1968-1973 (polskie wydanie WNT 2002).
2. D.C Kozen, *The design and analysis of algorithms*, Springer-Verlag, 1992.
3. S. Baase i A.vav Gelder, *Computer algorithms: introduction to design and analysis*, Addison-Wesley, 2000 .
4. L. Banachowski, A. Kreczmar i W. Rytter, *Analiza algorytmów i struktur danych*, WNT, 1987.
5. L. Banachowski, A. Kreczmar i W. Rytter, *Analysis of algorithms and data structures*, Addison-Wesley, 1991.
6. S.E. Goodman i S.T. Hedetniemi, *Introduction to the design and analysis of algorithms*, McGraw-Hill, 1977.
7. M.T.Goodrich i R.Tamassia, *Data structures and algorithms in JAVA,* Wiley, 1998.
8. E.M. Reingold, J. Nievergeld i N. Deo, *Algorytmy kombinatoryczne*, PWN, 1985.
9. R.Sedgewick, *Algorytmy w C++,* Wyd. ReadMe, 1999.
10. S.S.Skiena, *The algorithm design manual,* Springer-Verlag, 1997.
11. M.M.Sysło, N.Deo i J.S.Kowalik, *Algorytmy optymalizacji dyskretnej,* PWN, 1995.
12. J.P Tremblay i P.G Sorenson, *An introduction to data structures with applications*, McGraw-Hill, 1976.
13. M.A.Weiss, *Data structures and algorithm analysis,* Benjamin Cummings, 1992.
14. D.H. Greene i D.E. Knuth, *Mathematics for the analysis of algorithms*, Birkhäuser, 1982.

## 2 Problemy, algorytmy, programy

Zakładam, że wszyscy znają te pojęcia. Poniżej podajemy przykłady dwóch problemów oraz różnych algorytmów rozwiązujących je.

**Przykład 1** *Mnożenie liczb naturalnych.*

PROBLEM.
*dane*: $a, b \in \mathcal{N}$
*wynik*: iloczyn liczb $a$ i $b$

ALGORYTM 1. $a$ razy dodać do siebie liczbę $b$

ALGORYTM 2. "Pomnożyć pisemnie"

ALGORYTM 3. Mnożenie *"po rosyjsku"*

1. oblicz ciąg $a_1, a_2, ..., a_k$ taki, że $a_1 = a$, $a_k = 1$, $a_{i+1} = \lfloor \frac{a_i}{2} \rfloor$ (dla $i = 1, ..., k-1$),
2. oblicz ciąg $b_1, b_2, ..., b_k$ taki, że $b_1 = b$, $b_{i+1} = 2b_i$ (dla $i = 1, ..., k-1$),
3. oblicz $\sum_{\substack{i=1 \\ a_i \text{ nieparzyste}}}^{k} b_i$

□

UWAGA: Pó"xniej poznamy jeszcze dwa inne (niebanalne) algorytmy mnożenia liczb.

**Przykład 2** *Obliczanie n-tej liczby Fibonacciego.*

PROBLEM.
*dane*: $n \in \mathcal{N}$
*wynik*: wartość $n$-tej liczby Fibonacciego modulo stała c

ALGORYTM 1. Metoda rekurencyjna

```
fibrek(intn)
{   if (n ≤ 1) return 1;
    return (fibrek(n − 1) + fibrek(n − 2)) mod c;
}
```

ALGORYTM 2. Metoda iteracyjna

```
fibiter(intn)
{   inti, t, f0, f1;
    f0 ← f1 ← 1;
    for (i = 2; i ≤ n; i++)
       {t ← f0; f0 ← f1; f1 ← (t + f0) mod c; }
    return f1;
}
```

ALGORYTM 3. Metoda *"macierzowa"*

Korzystamy z tego, że

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_i \\ f_{i+1} \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_{i+2} \end{bmatrix}$$

Stąd

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix}$$

Wystarczy więc podnieść macierz do odpowiedniej potęgi (wykonując obliczenia modulo $c$) a następnie wynik przemnożyć przez wektor $[1, 1]^T$. $\square$

UWAGI:

- Będziemy zajmować się tylko problemami określonymi na nieskończonej dziedzinie (zbiorze danych). Dla problemu mnożenia jest nią $\mathcal{N}^2$, a dla problemu obliczania liczb Fibonacciego $\mathcal{N}$.
- Do zapisu algorytmów będziemy stosować różne formalizmy: od języka C, poprzez pseudopascal do opisu słownego.

# 3 Złożoność algorytmów i problemów

Efektywność (złożoność) algorytmów można porównywać empirycznie bąd"x teoretycznie. Wadą pierwszej metody jest jej zależność od implementacji oraz fakt, że zwykle można przetestować tylko niewielką grupę danych. Będziemy zajmować się głównie drugą metodą. Złożoność algorytmów będziemy określać funkcją rozmiaru danych.

**Przykład 3** *Przykłady określenia rozmiaru danych.*

| Problem | Rozmiar danych |
|---|---|
| *Wyszukiwanie elementu w ciągu* | *# elementów w ciągu* |
| *Mnożenie macierzy* | *Rozmiary macierzy* |
| *Sortowanie ciągu liczb* | *# elementów w ciągu* |
| *Przechodzenie drzewa binarnego* | *# węzłów w drzewie* |
| *Rozwiązywanie układu równań* | *# równań lub # zmiennych lub obie* |
| *Problemy grafowe* | *# wierzchołków lub # krawędzi lub obie.* |

□

Będzie nas interesować:

- *Złożoność czasowa* - liczba jednostek czasu potrzebnych na wykonanie algorytmu.
- *Złożoność pamięciowa* - liczba jednostek pamięci (np. komórek, bitów) potrzebnych na wykonanie algorytmu.

*Jednostka czasu* - czas potrzebny na wykonanie elementarnej operacji. Aby nasze rozważania były precyzyjne musimy określić model komputera. Dla nas podstawowym modelem będzie maszyna RAM (jej krótki opis zamieszczony jest na końcu notatki). Zwykle będziemy przyjmować następujące:

- *kryterium jednorodne* - koszt każdej operacji maszyny RAM jest jednostkowy.

Kryterium jednorodne jest nierealistyczne w przypadku algorytmów operujących na wielkich liczbach. W takich przypadkach będziemy posługiwać się:

- *kryterium logarytmicznym* - koszt operacji maszyny RAM jest równy sumie długości operandów.

UWAGA: Stosując kryterium logarytmiczne należy uwzględniać koszt obliczania adresu w trakcie wykonywania rozkazów stosujących adresowanie pośrednie.

Oczywiście analizując algorytmy nie będziemy ich zapisywać w języku maszyny RAM. Będzie ona jedynie naszym punktem odniesienia podczas analizy kosztów konstrukcji algorytmicznych wyższego rzędu.

**Przykład 4** *Dwa algorytmy sortowania ciągu liczb.*

```
Procedure insert(T[1..n])
    for i ← 2 to n do
        x ← T[i]; j ← i − 1
        while j > 0 and x < T[j] do
            T[j + 1] ← T[j]
            j ← j − 1
        T[j + 1] ← x
```

```
Procedure select(T[1..n])
    for i ← 1 to n − 1 do
        minj ← i; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Idea tych algorytmów:

- *Insert*: w $i-$tej iteracji element $T[i]$ wstawiamy w odpowiednie miejsce do uporządkowanego ciągu $T[1], \ldots, T[i-1]$.
- *Select*: po $i-1$-szej iteracji elementy $T[1], \ldots, T[i-1]$ są uporządkowane i mniejsze od każdego elementu $T[i], \ldots, T[n]$; w $i$-tej iteracji wybieramy minimalny element spośród $T[i], \ldots, T[n]$ i wstawiamy go na pozycję $T[i]$.

Złożoność czasowa (przy kryterium jednorodnym):

- *Select* - zawsze rzędu $n^2$.
  UZASADNIENIE: Najpierw zauważamy, że instrukcja **if** sprowadza się do wykonania stałej liczby instrukcji maszyny RAM. Podobnie jest z inicjalizacją i jednokrotną iteracją pętli **for** . Instrukcje pętli wewnętrznej wykonują się $\Theta(n^2)$ razy. Każda ich iteracja to koszt stałej liczby instrukcji maszyny RAM, tak więc w sumie koszt wykonania tych instrukcji jest $\Theta(n^2)$. Koszt pozostałych instrukcji jest mniejszy i nie wyprowadza poza $\Theta(n^2)$.

- *Insert* - dla niektórych danych rzędu $n^2$, dla niektórych jedynie rzędu $n$.
  UZASADNIENIE: Koszt procedury zależy od początkowego uporządkowania tablicy $T$. W najgorszym przypadku element $T[i]$ wstawiany jest na początek tablicy co wymaga $i$ operacji przesunięcia. Łatwo sprawdzić, że gdy taka sytuacja ma miejsce dla każdego $i = 2, \ldots, n$ (tj. gdy początkowo tablica uporządkowana jest malejąco), to koszt procedury wynosi $\Theta(n^2)$. Z drugiej strony, gdy początkowo tablica jest uporz 'adkowana rosnąco, to dla każdego $i2, \ldots, n$ pętla **while** ma koszt stały, a więc cała procedura wykonuje się w czasie liniowym (tj. $\Theta(n)$).

□

Powyźsze spostrzeźenia prowadzą nas do pojęcia złożoności najgorszego i średniego przypadku:

- *złożoność najgorszego przypadku* (inaczej złożoność pesymistyczna) - maksimum kosztu obliczeń na danych rozmiaru $n$.
- *złożoność średniego przypadku* (inaczej złożoność oczekiwana) - średni koszt obliczeń na danych rozmiaru $n$.

UWAGA: Przy obliczaniu średniej złożoności należy uwzględniać rozkład prawdopodobieństwa z jakim algorytm będzie wykonywany na poszczególnych danych (zwykle jest to bardzo trudne do ustalenia).

Powracając do naszego przykładu widzimy, że złożoność pesymistyczna obydwu algorytmów sortowania jest rzędu $n^2$. Jak pó"xniej pokaźemy obydwa algorytmy mają także taką samą złożoność w średnim przypadku.

Stając przed dylematem wyboru któregoś spośród algorytmów należy postępować bardzo rozważnie i uwzględnić szereg czynników. W przypadku powyźszych algorytmów czynnikami takimi są m.in.:

- Rozkład danych. Co prawda w średnim przypadku *insert* nie jest lepszy od *select*, jednak jest stwierdzenie to jest prawdziwe przy założeniu jednostajnego rozkładu danych. W praktyce często mamy do czynienia z innymi rozkładami, w tym często z danymi prawie uporządkowanymi. Taka sytuacja przemawia za wyborem *insert*.
- Wielkość rekordów. Często sortujemy nie tyle same klucze, co rekordy, zawierające klucze jako jedno ze swych pól. Jeśli rekordy są duże, to należy uwzględnić fakt, że operacja przestawienia elementów jest kosztowna. Ponieważ *select* wykonuje zawsze $O(n)$ operacji

przestawienia elementów, a *insert* może ich wykonywać nawet $\Omega(n^2)$, więc taka sytuacja może przemawiać za wyborem *select*.
W takiej sytuacji można też rozważyć użyteczność wersji *insert* operującej na kopiach kluczy i wska"xnikach do rekordów. Wska"xniki te służą do przestawienia rekordów zgodnie z otrzymaną poprzez sortowanie permutacją kluczy.

- Stabilność. Czasami zależy nam, by rekordy o jednakowych kluczach pozostawały w tablicy wynikowej w takim samym względnym porządku w jakim były początkowo. O procedurach sortowania zachowujących taki porządek mówimy, że są *stabilne*. Jak łatwo sprawdzić *insert* jest stabilny a *Select* nie.

- Intensywność wykorzystania algorytmu. Jeśli algorytm ma być bardzo intensywnie wykorzystywany, np. jako część składowa większego systemu, wówczas nawet drobne usprawnienia mogą prowadzić do istotnej poprawy efektywności całego systemu. W tym przypadku warto zwrócić uwagę na możliwość dokonania takich usprawnień w algorytmach jak redukcja liczby rozkazów w najbardziej wewnętrznych pętlach algorytmu, możliwość zastosowania szybkich operacji maszynowych, itp... Warto też duży nacisk położyć na porównanie empiryczne algorytmów.

Na zakończenie jeszcze uwaga o *złożoności problemów*. Definiuje się ją jako złożoność najlepszego algorytmu rozwiązującego dany problem. Zwykle jest ona dużo trudniejsza do określenia niż złożoność konkretnego algorytmu.
Załóżmy, że chcemy określić złożoność problemu $\mathcal{P}$. Skonstruowanie algorytmu $\mathcal{A}$ rozwiązującego $\mathcal{P}$ pozwala nam jedynie na sformułowanie wniosku, że złożoność $\mathcal{P}$ jest nie większa niż złożoność $\mathcal{A}$ (mówimy, że algorytm $\mathcal{A}$ wyznacza *granicę górną* na złożoność $\mathcal{P}$). Aby dokładnie wyznaczyć złożoność $\mathcal{P}$ należy ustalić jeszcze granicę dolną, a więc wykazać, że żaden algorytm nie jest w stanie rozwiązać $\mathcal{P}$ szybciej. Twierdzenia tego typu są bardzo trudne i stanowią prawdziwe wyzwanie dla naukowców. W trakcie wykładu zapoznamy się tylko z kilkoma przykładami takich twierdzeń i to tylko dla ograniczonego (w stosunku do maszyny RAM) modelu obliczeń.

## 4 Notacja dla rzędów funkcji

OZNACZENIA:
$\mathcal{R}^*$ - zbiór nieujemnych liczb rzeczywistych,
$\mathcal{R}^+$ - zbiór dodatnich liczb rzeczywistych,
analogiczne oznaczenia dla $\mathcal{N}$.

**Definicja 1** *Niech $f : \mathcal{N} \to \mathcal{R}^*$ będzie dowolną funkcją.*

- $O(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c \in \mathcal{R}^+} \exists_{n_0 \in \mathcal{N}} \forall_{n \geq n_0} \quad t(n) \leq cf(n).\}$
- $\Omega(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c \in \mathcal{R}^+} \exists_{n_0 \in \mathcal{N}} \forall_{n \geq n_0} \quad t(n) \geq cf(n).\}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

UWAGA: Czasami $\Omega(f(n))$ jest definiowana jako $\{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c \in \mathcal{R}^+} \forall_{n_0 \in \mathcal{N}} \exists_{n \geq n_0} \quad t(n) \geq cf(n).\}$

## 5 Podstawowe struktury danych

Zakładam znajomość takich struktur danych (i ich implementacji) jak: tablice, rekordy, listy, kolejki, stos, drzewa, grafy (listy incydencji, macierz sąsiedztwa),... .

## 6 Dodatek: Krótki opis maszyny RAM

CZĘŚCI SKŁADOWE:

- *taśma wejściowa* - ciąg liczb całkowitych; dostęp jednokierunkowy;
- *taśma wyjściowa*
- *pamięć:* komórki adresowane liczbami naturalnymi; każda komórka może pamiętać dowolną liczbę całkowitą.
- *akumulator* - komórka o adresie 0.
- procesor.

INSTRUKCJE:

| | | | |
|---|---|---|---|
| LOAD | *argument* | STORE | *argument* |
| ADD | *argument* | SUB | *argument* |
| MULT | *argument* | DIV | *argument* |
| READ | *argument* | WRITE | *argument* |
| JUMP | *etykieta* | JGTZ | *etykieta* |
| JZERO | *etykieta* | HALT | |

Operacje przesłania i arytmetyczne mają dwa argumenty - drugim jest akumulator. W nim umieszczany jest wynik operacji arytmetycznych.

RODZAJE ARGUMENTÓW:

| postać | znaczenie |
|---|---|
| $=$*liczba* | stała |
| *liczba* | adres |
| $\star$ *liczba* | adresowanie pośrednie |