

**PROJEKTOWANIE I ANALIZA  
ALGORYTMÓW**

**Wykład I**

**dr inż. Łukasz Jelen**  
Na podstawie wykładów dr. T. Fevensa

## INFORMACJE PODSTAWOWE

- Kontakt
  - e-mail: [lukasz.jelen@pwr.edu.pl](mailto:lukasz.jelen@pwr.edu.pl)
- E-portal
- Pok. 230, c-3 w godzinach konsultacji

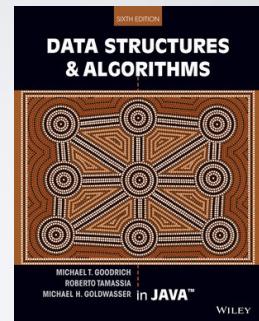
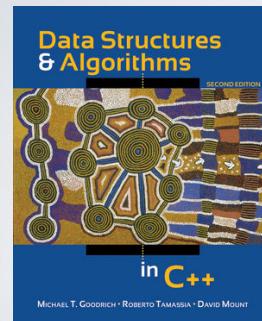
© 2004 Goodrich, Tamassia

## ZALICZENIE

- **Ocena z przedmiotu:**
  - **Egzamin (50%) + laboratorium (50%) - I termin**
    - **Warunkiem koniecznym podejścia do egzaminu jest pozytywna ocena z laboratorium**
  - **Egzamin** - test na e-portalu w sesji egzaminacyjnej - sala i czas: TBA
  - **Egzamin II termin** - w sesji poprawkowej
    - Wszystkie daty będą zgłoszone do dydaktyki oraz podane na e-portalu oraz stronie WIT
  - **Projekty** - szczegóły u prowadzących

© 2004 Goodrich, Tamassia

## LITERATURA



© 2004 Goodrich, Tamassia

## LITERATURA

- LITERATURA PODSTAWOWA:
  - [1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Wprowadzenie do algorytmów, WNT, W-wa.
  - [2] N.Wirth. Algorytmy+struktury danych = Programy, WNT, W-wa
- LITERATURA UZUPEŁNIAJĄCA:
  - [1] M.T. Goodrich, R. Tamassia, D. Mount, Data Structures and Algorithms, John Wiley and Sons, Inc., Hoboken, NJ, USA
  - [2] Aho A.V., Hopcroft J.E., Ullman J.D.: Projektowanie i analiza algorytmów komputerowych. PWN, W-wa

© 2004 Goodrich, Tamassia

## PROBLEM VS. ALGORYTM

- Ciasto:
  - 1 szklanka wody
  - 150g margaryny
  - 1 szklanka mąki pszennej
  - 5 jajek
  - szczypta soli
  - szczypta proszku do pieczenia

Co to jest?



© 2004 Goodrich, Tamassia

Mając dane wejściowe (składniki) mamy rozwiązać problem pieczenia ciasta

**Jak?**

## PRZEPIS NA CIASTO

### Przepis

<b>Przygotuj</b>	<b>Piekarnik : 180°</b>	<b>Woda: 250ml</b>	<b>Olej: 83ml</b>
<b>Instrukcje</b> Podgrzej piekarnik do 180°. Posmaruj formę margaryną			
<b>Ciało</b> W wysokim naczyniu wymieszać zawartość torebki z wodą i olejem. Ubijaj przez 2 minuty na wysokich obrotach			
<b>Pieczenie</b>	32x22 cm: 30 – 40 min.	Ø 20 cm: 30 – 40 min.	Forma do babeczek: 15 – 20 min.

1. Przygotować ciasto. Wodę zagotować z margaryną (...) po jednym jajku, szczypta soli i proszek do pieczenia.
2. Ciasto podzielić na 2 części.
3. Formę prostokątną o wymiarach ok. 35x 24cm wysmarować margaryną i posypać mąką. Połowę ciasta rozprowadzić łyżką w formie.
4. Piec w nagrzanym piekarniku, na złoty kolor, ok. 30min. w temperaturze 180°C. W ten sam sposób upiec drugą połowę ciasta.
5. Przygotować masę budyniową, 2 szklanki mleka i cukier zagotować (...) dodawać stopniowo zimny budyń.
6. Masę rozsmarować na jednym blacie ciasta. (...) Ciasto wstawić do lodówki, na co najmniej 2 godz.
7. Gotowe posypać cukrem pudrem.

© 2004 Goodrich, Tamassia

## DEFINICJA PROBLEMU

- Zbiór danych wraz z poleceniem wykonania
  - dane wejściowe
  - definicja danych
  - opis problemu
    - pytanie
    - polecenie



© 2004 Goodrich, Tamassia

## PRZEPIS NA CIASTO A PROGRAM KOMPUTEROWY

### Przepis

- daje nam specyfikację potrzebnych przedmiotów i przedstawia szczegółowy opis instrukcji ich użycia
- opis składników i czynności potrzebnych do upieczenia ciasta.
- rozwiązywanie problemu pieczenia ciasta
- Dokładne wykonanie instrukcji skutkuje poprawnie upieczonym ciastem.
- Producent rozwiązał za nas problem pieczenia ciasta.
- Program jest zestawem instrukcji potrzebnych do poprawnego rozwiązania problemu.
- Pisząc program analizujemy problem i rozwiązujeśmy go tworząc przepis (program).

© 2004 Goodrich, Tamassia

## PROGRAM KOMPUTEROWY (KOD) = STRUKTURA(Y) DANYCH + ALGORYTM(Y)

- Musimy znać podstawowe algorytmy i struktury danych aby móc tworzyć kod dobrej jakości
  - Przez „dobrą jakość” rozumieć będziemy program, który:
    - jest wydajny
    - rozwiązuje problem zgodnie z nałożonymi ograniczeniami sprzętowymi
- Nie można zostać dobrym programistą bez dobrej znajomości algorytmów i struktur danych

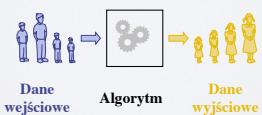
**Struktury danych** organizują informację

**Algorytm** przetwarza informację

© 2004 Goodrich, Tamassia

## ALGORYTM

- Jest zbiorem dobrze zdefiniowanych zasad niezbędnych do rozwiązania problemu.
- Przetwarza dane wejściowe problemu na dane wyjściowe, które ten problem rozwiązują.
- Problem może zawierać wiele algorytmów



© 2004 Goodrich, Tamassia

## ALGORYTM

- Właściwości:
  - Musi być poprawny
  - Musi się składać ze ściśle zdefiniowanych kroków
  - Kolejność kroków musi być ściśle określona
  - Musi się składać ze skończonej ilości kroków
  - Musi się zakończyć dla wszystkich danych wejściowych

**Program komputerowy jest instancją lub konkretną reprezentacją algorytmu w dowolnym języku programowania**



© 2004 Goodrich, Tamassia

## STRUKTURY DANYCH - DEFINICJE

- **Typ** jest to zbiór wartości
  - np.: Integer, Boolean
- **Typ danych** jest typem i zbiorem operacji, które przetwarzają ten typ.
  - np.: Suma, Iloczyn
- **Dana** jest elementem typu danych - informacja
  - Np.: Typy danych w C++:  
prosty: int, float, bool  
zagregowany: array, struct, vector, string
  - Zagregowany typ danych jest przykładem struktury danych składający się z:
    - prostych pól
    - powiązań między polami
    - operacji na strukturze danych, które pozwalają na manipulację tych pól

© 2004 Goodrich, Tamassia

## STRUKTURY DANYCH

- W językach programowania, niektóre struktury danych są wbudowane (np.: tablice, łańcuchy znakowe)
- Wiele innych struktur danych jest często niezbędnych
  - możemy je zaimplementować z wykorzystaniem wbudowanych struktur danych
- Nazywamy je strukturami zdefiniowanymi przez użytkownika (user-defined)
  - STL dla C++
  - Do zdefiniowania struktur danych w C++ wykorzystuje się klasy
- Na wykładzie poznamy struktury danych i algorytmy, które są najczęściej wykorzystywane w wielu aplikacjach

© 2004 Goodrich, Tamassia

## ABSTRACT DATA TYPE (ADT)

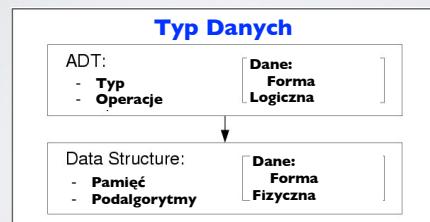
- Definicja typu danych tylko za pomocą wartości i operacji na tym typie danych
- Jest specyfikacją typu danych, która:
  - ukazuje istotne cechy
  - ukrywa detale implementacyjne
- Definiuje typ danych przez zależności **wejścia - wyjścia**
  - np.: każda operacja ADT jest zdefiniowana przez jej wejścia i wyjścia
- ADT radzą sobie ze złożonością poprzez abstrakcję: metafore



© 2004 Goodrich, Tamassia

## STRUKTURA DANYCH VS. ADT

- Struktura danych jest konkretną implementacją ADT



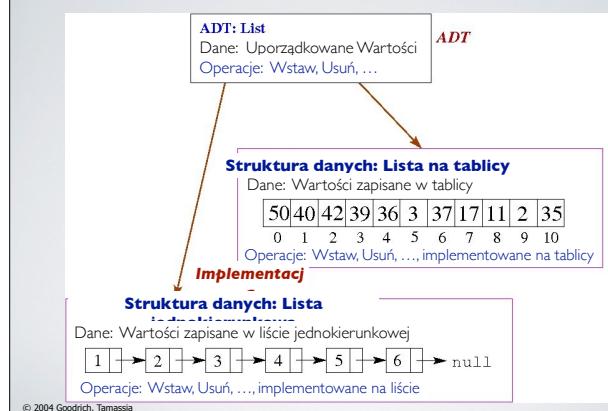
© 2004 Goodrich, Tamassia

## WYBÓR IMPLEMENTACJI ADT

- Wszystkie struktury danych posiadają swoje wady i zalety.
- Bardzo rzadko jedna struktura danych jest lepsza od innych we wszystkich sytuacjach.
- Struktura danych wymaga:
  - przestrzeni dla każdej przechowywanej danej
  - czasu do przeprowadzenia każdej podstawowej operacji
  - wysiłku programistycznego.

© 2004 Goodrich, Tamassia

## PRZYKŁAD: SD VS. ADT



© 2004 Goodrich, Tamassia

## WYBÓR IMPLEMENTACJI ADT

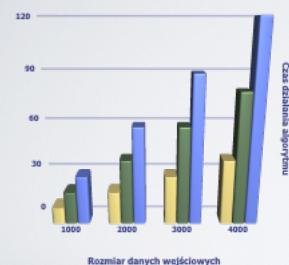
- Tylko po dokładnej analizie charakterystyki problemu możemy wybrać najlepszą strukturę danych dla danego zadania
- Przykład z banku:
  - Otwarcie konta: kilka minut
  - Transakcje: kilka sekund
  - Zamknięcie konta: doba



© 2004 Goodrich, Tamassia

## ZŁOŻONOŚĆ OBLCZENIOWA

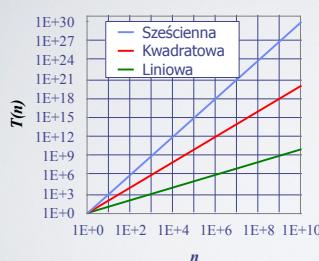
Najlepszy przypadek  
Średni przypadek  
Najgorszy przypadek



© 2004 Goodrich, Tamassia

- Większość algorytmów przekształca obiekty wejściowe w obiekty wyjściowe
- Czas działania (złożoność obliczeniowa) algorytmu zazwyczaj wzrasta wraz z rozmiarem danych wejściowych
- Średni czas działania jest najczęściej trudny do określenia
- koncentrujemy się na przypadku najgorszym
  - łatwiejszy do analizy
- Istotny w aplikacjach takich jak gry, finanse i robotyka

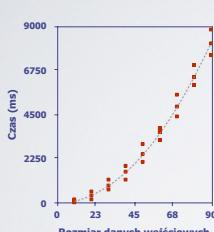
## SIEDEM WAŻNYCH FUNKCJI



© 2004 Goodrich, Tamassia

- Siedem funkcji często wykorzystywanych w analizie algorytmów:
  - Stała  $\approx 1$
  - Logarytmiczna  $\approx \log n$
  - Liniowa  $\approx n$
  - $N\text{-Log-}N \approx n \log n$
  - Kwadratowa  $\approx n^2$
  - Sześcienna  $\approx n^3$
  - Wykładnicza  $\approx 2^n$
- Na wykresie log-log, nachylenie linii świadczy o wzroście funkcji

## DOŚWIADCZENIA



© 2004 Goodrich, Tamassia

- Napisz program implementujący algorytm
- Przetestuj napisany program na danych o różnych rozmiarach
- Wykorzystaj metodę typu `System.currentTimeMillis()` do dokładnego oszacowania czasu działania algorytmu
- Zrób wykres dla otrzymanych wyników.

## OGRANICZENIE EKSPERYMENTÓW

- Niezbędne jest zaimplementowanie algorytmu, który może być trudny
- Wyniki złożoności obliczeniowej mogą nie być znaczące dla danych wejściowych, które nie były wykorzystywane w eksperymetach
- W celu porównania dwóch algorytmów należy korzystać z tego samego sprzętu i oprogramowania



© 2004 Goodrich, Tamassia

## ANALIZA TEORETYCZNA



- Wykorzystuje formalną reprezentację algorytmu zamiast implementacji
- Charakteryzuje złożoność obliczeniową jako funkcję rozmiaru danych wejściowych,  $n$
- Bierze pod uwagę wszystkie możliwe dane wejściowe
- Pozwala nam na ocenę szybkości działania algorytmu niezależnie od sprzętu/oprogramowania

© 2004 Goodrich, Tamassia

## DEFINICJA

- Złożoność obliczeniowa algorytmu A jest zdefiniowana przez:
  - $t$  - czas - ilość operacji niezbędnych do rozwiązania dowolnej instancji I problemu o rozmiarze  $N(I)$  przez algorytm A  $\Rightarrow N(I) = n$
  - $f_A(n) = \max(t)$
- Nas interesuje jak wygląda funkcja  $F_A$ , a nie jej wartości

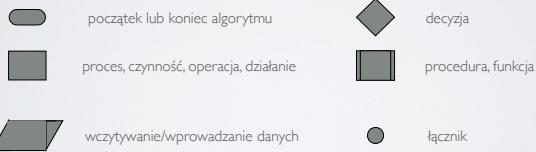
© 2004 Goodrich, Tamassia

## METODY REPREZENTACJI ALGORYTMÓW

- Pseudo kod

- Graficznie

- Schematy blokowe



© 2004 Goodrich, Tamassia

## PSEUDO KOD

- Uogólniony opis algorytmu
- Bardziej strukturalny niż opis w języku polskim
- Mniej szczegółowy od programu komputerowego
- Preferowana notacja do opisu algorytmów
- Ukrywa aspekty projektowania programu

© 2004 Goodrich, Tamassia

Przykład: znajdź element max tablicy

**Algorytm** *tabMax( $T, n$ )*  
Wejście tablica  $T$  zawierająca  $n$  integerów  
Wyjście element maksymalny  $T$   
 $biezacyMax \leftarrow T[0]$   
**for**  $i \leftarrow 1$  **do**  $n - 1$   
    **if**  $T[i] > biezacyMax$  **then**  
         $biezacyMax \leftarrow T[i]$   
**return**  $biezacyMax$

## DETALE PSEUDOCODU

- Kontrola działania

- if ... then ... [else ...]**

- Wypołanie metody

*zm.metoda (arg [, arg...])*

- while ... do ...**

- Zwracanie wartości

- repeat ... until ...**

**return** wyrażenie

- for ... do ...**

- Wyrażenia

- Wcięcia zastępują nawiasy

← Przypisanie  
(tak jak = w C++/Java)

- Deklaracja metod

= Testowanie równości  
(tak jak == w C++/Java)

- Algorytm** *metoda (arg [, arg...])*

**Wejście ...**  
**Wyjście**

→ Superskrypty i inne matematyczne

formatowanie jest dozwolone

© 2004 Goodrich, Tamassia

## OPERACJE PODSTAWOWE



- Podstawowe obliczenia są wykonywane przez algorytm
- Identyfikowane w pseudokodzie
- Niezależne od języka programowania
- Dokładna definicja nie jest istotna (poźniej zobaczymy dlaczego)
- Z założenia pobierają stałą ilość pamięci oraz wykonywane są w ścisłe określonym czasie

© 2004 Goodrich, Tamassia

- Przykłady:

- Wykonanie wyrażeń
- Przypisanie wartości do zmiennej
- Indeksowanie tablicy
- Wywołanie metody
- Powrót z metody

## ZLICZANIE OPERACJI PODSTAWOWYCH

- Badając pseudokod możemy określić maksymalną ilość operacji podstawowych wykonywanych przez algorytm w funkcji  $n$  - rozmiarze danych wejściowych

**Algorytm** *tabMax( $T, n$ )*  
 $biezacyMax \leftarrow T[0]$   
**for**  $i \leftarrow 1$  **do**  $n - 1$   
    **if**  $T[i] > biezacyMax$  **then**  
         $biezacyMax \leftarrow T[i]$   
    { zwiększanie licznika  $i$  }  
**return**  $biezacyMax$

il. operacji
2
2n
2(n - 1)
2(n - 1)
1
Suma 8n - 3

© 2004 Goodrich, Tamassia

## OKREŚLANIE ZŁOŻONOŚCI OBliczeniowej - CZASU DZIAŁANIA ALGORYTMU

- Algorytm tabMax wykonuje  $8n - 3$  operacji podstawowych w najgorszym przypadku. Zdefiniujmy:
    - $a$  = Czas wykonania najszybszej operacji podstawowej
    - $b$  = Czas wykonania najwolniejszej operacji podstawowej
  - Niech  $T(n)$  będzie najgorszym czasem tabMax. Wtedy
- $$a(8n - 3) \leq T(n) \leq b(8n - 3)$$
- Zatem, czas  $T(n)$  jest ograniczony przez dwie funkcje liniowe

© 2004 Goodrich, Tamassia

## WSPÓŁCZYNNIK WZROSTU ZŁOŻONOŚCI OBliczeniowej



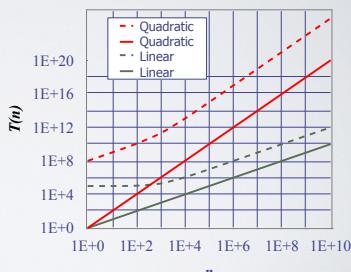
- Zmiana środowiska sprzętowego/oprogramowania
  - Ma stały wpływ na  $T(n)$ , ale
  - nie ma wpływu na współczynnik wzrostu  $T(n)$
- Liniowy wzrost czasu działania  $T(n)$  jest istotną właściwością algorytmu tabMax

© 2004 Goodrich, Tamassia

## SKŁADOWA STAŁA

- Asymptotyczny współczynnik wzrostu nie zależy od:

- składowych stałych lub
  - wyrażeń niższego rzędu
- Przykłady
    - $10^2n + 10^5$  jest funkcją liniową
    - $10^5n^2 + 10^8n$  jest funkcją kwadratową



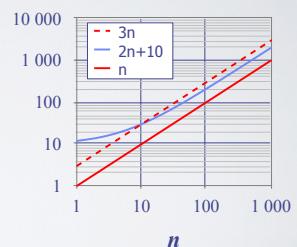
© 2004 Goodrich, Tamassia

## NOTACJA DUŻE O

- Mając daną funkcję  $f(n)$  i  $g(n)$  mówimy, że  $f(n)$  należy do  $\mathcal{O}(g(n))$  jeśli istnieją stałe nieujemne  $c$  i  $n_0$  takie, że

$$f(n) \leq cg(n) \text{ dla } n \geq n_0$$

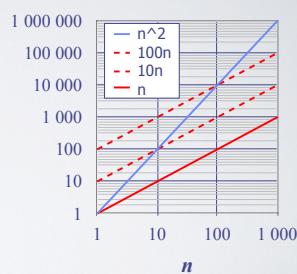
- Przykład:  $2n + 10$  jest w  $\mathcal{O}(n)$ 
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10/(c - 2)$
  - Weźmy  $c = 3$  i  $n_0 = 10$



© 2004 Goodrich, Tamassia

## PRZYKŁAD DUŻEGO O

- Przykład: funkcja  $n^2$  nie należy do  $\mathcal{O}(n)$ 
  - $n^2 \leq cn$
  - $n \leq c$
  - Powyższa nierówność nie może zostać spełniona ponieważ  $c$  musi być stała



© 2004 Goodrich, Tamassia

## WIĘCEJ PRZYKŁADÓW

### • $7n^2$

$7n^2$  jest w  $\mathcal{O}(n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $7n^2 \leq cn$  dla  $n \geq n_0$   
spełnione dla  $c = 7$  i  $n_0 = 1$

### • $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$  jest w  $\mathcal{O}(n^3)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $3n^3 + 20n^2 + 5 \leq cn^3$  dla  $n \geq n_0$   
spełnione dla  $c = 4$  i  $n_0 = 21$

### • $3 \log n + 5$

$3 \log n + 5$  jest w  $\mathcal{O}(\log n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $3 \log n + 5 \leq cn \log n$  dla  $n \geq n_0$   
spełniony dla  $c = 8$  i  $n_0 = 2$

© 2004 Goodrich, Tamassia

## DUŻE O I WSPÓŁCZYNNIK WZROSTU

- Notacja duże O daje nam górne ograniczenie współczynnika wzrostu funkcji.
- Określenie "f(n) jest w  $O(g(n))$ " oznacza, że współczynnik wzrostu funkcji f(n) jest nie większy niż współczynnik wzrostu funkcji g(n)
- Mogliśmy wykorzystać notację duże O do porównywania (stopniowania) funkcji względem ich współczynnika wzrostu

$f(n)$ jest w $O(g(n))$	$g(n)$ jest w $O(f(n))$	
Tak	Nie	$\rightarrow$ $g(n)$ rośnie szybciej
Nie	Tak	$\rightarrow$ $f(n)$ rośnie szybciej
Tak	Tak	$\rightarrow$ ten sam wzrost

© 2004 Goodrich, Tamassia

## ZASADY NOTACJI DUŻE O

- Jeśli f(n) jest wielomianem stopnia d, np.:
$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \dots + c_1 n^1 + c_0 n^0, \text{ to } f(n) \text{ jest w } O(n^d), \text{ np. :}$$
  - Pomiń wyrażenia niskiego stopnia
  - Pomiń stałe
- Wykorzystaj najmniejszą możliwą klasę funkcji
  - Powiemy " $2n$  jest w  $O(n)$ " zamiast " $2n$  jest w  $O(2n)$ "
  - Wykorzystaj najprostsze wyrażenie tej klasy
    - Powiemy " $3n + 5$  jest w  $O(n)$ " zamiast " $3n + 5$  jest w  $O(3n)$ "

© 2004 Goodrich, Tamassia

# PROJEKTOWANIE I ANALIZA ALGORYTMÓW

## TABLICE, LISTY, STOS

### Wykład 2

dr inż. Łukasz Jeleń  
Na podstawie wykładów dr. T. Fevensa

## SPRAWY ORGANIZACYJNE

- Egzamin: 28.06.2023, godz. 9:00, sala:TBA
- Konsultacje:
  - czwartki godz. 11:00 - 13:00, s. 230, c-3

© 2004 Goodrich, Tamassia

## POPRZEDNIO

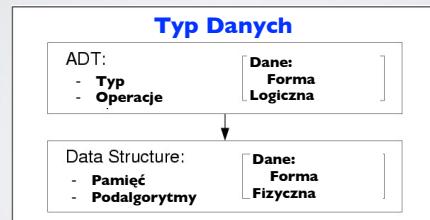


- Definicja Problemu
  - Zbiór danych wraz z poleceniem wykonania
- Definicja Algorytmu
  - Jest zbiorem dobrze zdefiniowanych zasad niezbędnych do rozwiązania problemu
- Struktury Danych
  - Typ danych wraz z polami, zależnościami między nimi oraz operacji na strukturze danych, które pozwalają na manipulację tych pól
- ADT
  - Definicja typu danych tylko za pomocą wartości i operacji na tym typie danych

© 2004 Goodrich, Tamassia

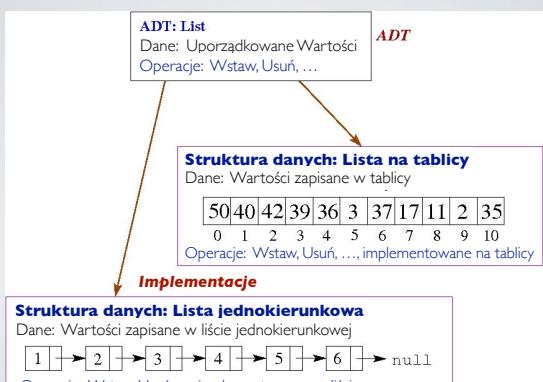
## STRUKTURA DANYCH VS. ADT

- Struktura danych jest konkretną implementacją ADT



© 2004 Goodrich, Tamassia

## PRZYKŁAD: SD VS. ADT



© 2004 Goodrich, Tamassia

## WYBÓR IMPLEMENTACJI ADT

- Tylko po dokładnej analizie charakterystyki problemu możemy wybrać najlepszą strukturę danych dla danego zadania
- Przykład z banku:
  - Otwarcie konta: kilka minut
  - Transakcje: kilka sekund
  - Zamknięcie konta: doba



© 2004 Goodrich, Tamassia

## TABLICE

arrays



© 2014 Goodrich, Tamassia, Goldwasser

## DEFINICJA

- Tablica jest sekwencyjnym zbiorem(kolekcja) zmennych tego samego typu.
- Każda tablica posiada:
  - komórkę/pole
  - indeks - unikalna referencia do wartości zapisanej na polu/w komórce.
  - 0, 1, 2, ..., n
- każda wartość zapisana w tablicy nazywana jest elementem



© 2004 Goodrich, Tamassia

- Deklaracja tablicy:

```
elementType[] arrayName = {initialValue0, initialValue1, ..., initialValueN-1};
```

- operator **new**:

- new elementType[rozmiar]
- zwraca referencia do nowej tablicy, która jest zazwyczaj zapisywana w zmiennej tablicowej

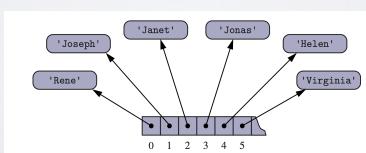
© 2004 Goodrich, Tamassia

## TABLICA ZNAKÓW VS. REFERENCJE DO OBIEKTÓW

- Tablica może przechowywać prymitywne elementy - znaki:



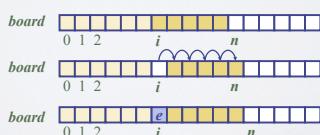
- może też przechowywać referencje do obiektów



© 2004 Goodrich, Tamassia

## PRZYKŁAD

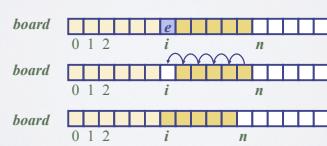
- Tablica przechowująca nazwiska graczy wraz z ich najlepszymi punktami w grze
- problem - dodawanie do tablicy
  - w celu dodania wpisu „e” do tablicy „board” na indeksie „i”, musimy zrobić dla niego miejsce poprzez przesunięcie **n-1** wpisów w tablicy



© 2004 Goodrich, Tamassia

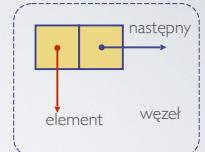
## PRZYKŁAD

- problem - usunięcie danych z tablicy
  - w celu usunięcia wpisu „e” z tablicy „board” na indeksie „i”, musimy przesunąć **n-1** wpisów w tablicy w celu załatwania dziury...



© 2004 Goodrich, Tamassia

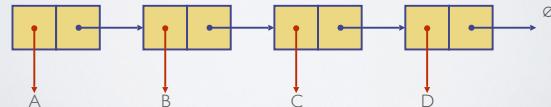
## LISTY



## LISTY JEDNOKIERUNKOWE

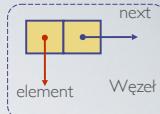
• Lista jednokierunkowa jest konkretną strukturą danych składającą się z sekwencji w\u0142\u00eaz\u0144

- Ka\u0142y w\u0142\u00eaz przechowuje:
  - element
  - link do nast\u0144ego w\u0142\u00eaz\u0144



© 2004 Goodrich, Tamassia

## DEFINICJA W\u0142\u00E3ZA DLA ELEMENT\u0144 LISTY



```
template <typename E>
class SNode{
    //w\u0142\u00eaz listy jednokier.
private:
    E elem; //warto\u0144 elementu listy
    SNode<E>* next; //nast\u0144eny elem. listy
    friend class SLinkedList<E>; //dost\u0144 do listy
};

//metody dost\u0144u do danych: //metody modyfikuj\u0144ce:
public E getElement()
{ return elem; }

public void setElement(E newE)
{ elem = newE; }

public SNode getNext()
{ return next; }

public void setNext(Node<E> newN)
{ next = newN; }
```

© 2004 Goodrich, Tamassia

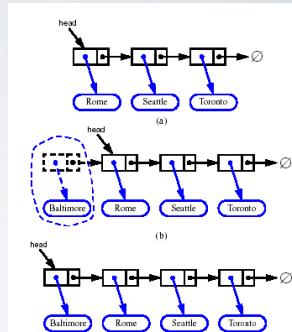
## INTERFEJS LISTY JEDNOKIERUNKOWEJ W C++

```
template <typename E>
class SLinkedList{ //lista jednokierunkowa
public:
    SLinkedList(); //konstruktor
    ~SLinkedList(); //destruktor
    bool empty() const; //sprawdzanie czy lista jest pusta
    const E& front() const; //zwraca pierwszy element
    void addFront(const E& e); //dodawanie na pocz\u0144tek listy
    void removeFront(); //usu\u0144 pierwszy element listy
private:
    SNode<E>* head; //pocz\u0144ek listy
};
```

© 2004 Goodrich, Tamassia

## DODAWANIE NA POCZ\u0144EKU

1. Alokacja nowego w\u0142\u00eaz\u0144
2. Dodanie nowego elementu
3. Doda\u0144 powi\u0144zanie tak, aby nowy w\u0142\u00eaz wskazywa\u0144 na pocz\u0144tek listy (stary head)
4. Uaktualni\u0144 head tak, aby wskazywa\u0144 na nowy w\u0142\u00eaz

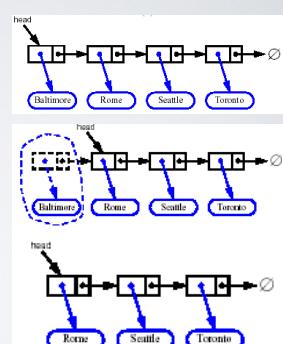


Przyk\u0144ad realizacji w materiałach na stronie

© 2004 Goodrich, Tamassia

## USUWANIE Z PRZODU LISTY

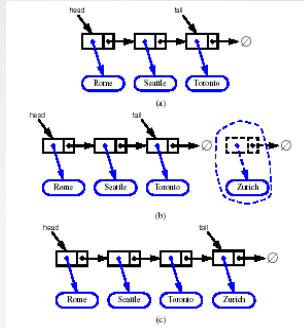
1. Uaktualni\u0144 head tak, aby wskazywa\u0144 na kolejny element w li\u0144cie
2. Usu\u0144\u0144 pierw\u0144szy w\u0142\u00eaz



© 2004 Goodrich, Tamassia

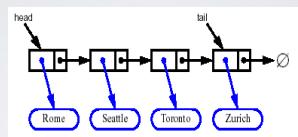
## DODAWANIE Z TYŁU LISTY

1. Alokacja nowego węzła
2. Dodanie nowego elementu
3. Dodać powiązanie tak, aby nowy węzeł wskazywał na **null**
4. Dodać powiązanie tak, aby ostatni element wskazywał na nowy węzeł
5. Uaktualnić *tail* tak, aby wskazywało na nowy węzeł



© 2004 Goodrich, Tamassia

## USUWANIE Z TYŁY LISTY



Usuwanie elementów z tyłu listy jednokierunkowej nie jest wydajne!

Nie da się przeprowadzić aktualnienia *tail* tak, aby wskazywał na węzeł poprzedzający w **stałym** czasie

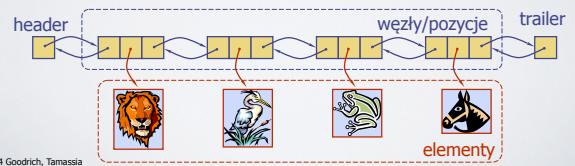
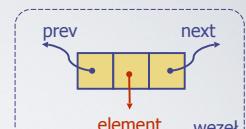
© 2004 Goodrich, Tamassia

## LISTY DWUKIERUNKOWE



## LISTA DWUKIERUNKOWA

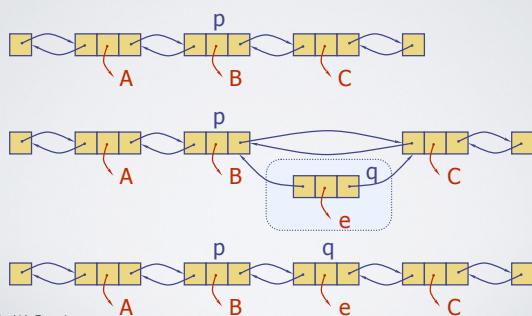
- Lista dwukierunkowa składa się z połączonych węzłów zawierających:
  - element
  - link do poprzedniego węzła
  - link do następnego węzła
- Zawiera dwa dodatkowe węzły
  - header
  - trailer



© 2004 Goodrich, Tamassia

## WSTAWIANIE

Wizualizacja metody `addAfter(p, e)` zwracającej pozycję *q*



© 2004 Goodrich, Tamassia

## WSTAWIANIE - ALGORYTM

**Algorytm** `addAfter(p, e)`:

Stwórz nowy węzeł *v*

*v.setElement(e)*

*v.setPrev(p) {link v do poprzednika}*

*v.setNext(p.getNext()) {link v do następcy}*

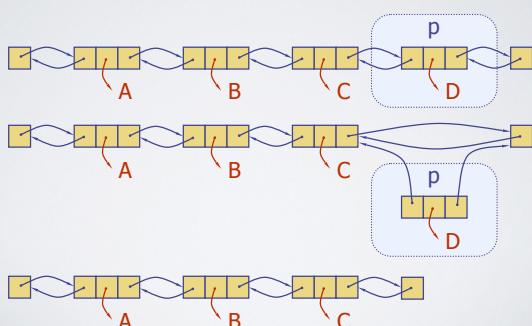
*(p.getNext()).setPrev(v) {link następca do v}*

*p.setNext(v) {link p do nowego następcy, v}*

© 2004 Goodrich, Tamassia

## USUWANIE

Wizualizacja `remove(p)`, gdzie `p = last()`



© 2004 Goodrich, Tamassia

## USUWANIE - ALGORYTM

**Algorytm** `remove(p):`

```
t ← p.element {tymczasowa zmienna do przechowywania  
zwracanej wartości}  
(p.getPrev()).setNext(p.getNext()) {usuwanie linków do p}  
(p.getNext()).setPrev(p.getPrev())  
p.setPrev(null) {usuwanie linków z p}  
p.setNext(null)  
return t
```

© 2004 Goodrich, Tamassia

## WYDAJNOŚĆ

- W implementacji listy dwukierunkowej stwierdzimy, że:
  - Wykorzystane przez listę miejsce dla n elementów jest w  $O(n)$
  - Miejsce wykorzystane przez każdy element listy jest w  $O(1)$

© 2004 Goodrich, Tamassia

## SEKWENCJA

- Sekwencja jest strukturą danych rozszerzającą definicję listy
- Zawiera funkcje pozwalające na dostęp do elementu poprzez podanie jego indeksu (jak dla wektora)
- Interfejs składa się z operacji na liście oraz:
  - `atIndex(i):` Zwraca element na pozycji i.
  - `indexOf(p):` Zwraca indeks elementu p.

```
class NodeSequence : public NodeList {  
public:  
    Iterator atIndex( int i ) const;  
        // zwraca element na pozycji i  
  
    int indexOf( const Iterator& p ) const;  
        // zwraca indeks elementu p  
};
```

© 2004 Goodrich, Tamassia

## STOSY



## ABSTRACT DATA TYPES (ADT)

Abstract data type (ADT) jest abstrakcją struktury danych

ADT definiuje:

Przechowywane dane  
Operacje na danych  
Definicję możliwych błędów tych operacji

Przykład: ADT modelujące prosty system obrotu papierami wartościowymi

Przechowywane dane to zamówienia kupna/sprzedaży

Dostępne operacje:

zamówienie kupna(akcja, udziały, cena)  
zamówienie sprzedaży(akcja, udziały, cena)  
void cancel(zamówienie)

Wyjątki:

Kupno/Sprzedaż nieistniejącą akcję  
Cancel nieistniejące zamówienie

© 2004 Goodrich, Tamassia

## STOS ADT



- ADT dla stosu przechowuje dowolne obiekty
  - Dodawanie i usuwanie wykonywane jest na zasadzie LIFO - Last-in First-out
  - Można przyrównać do zabawki dozującą cukierki (PEZ)
  - Operacje na stosie:
    - push(element): dodawanie elementu
    - element pop(): usuwa i zwraca element z wierzchu stosu
- Dodatkowe operacje na stosach:
  - object top(): zwraca ostatni element umieszczony na stosie bez jego usuwania
  - integer size(): zwraca ilość przechowywanych elementów
  - boolean isEmpty(): mówi, czy są jakieś elementy przechowywane na stosie

© 2004 Goodrich, Tamassia

## WYJĄTKI

- Próba wywołania operacji ADT może czasami prowadzić do wystąpienia błędów, które nazywamy wyjątkami
  - Exceptions
- Wyjątki są „wyrzucane” (thrown) przez operacje i nie mogą być wykonywane
- Dla stosu operacje pop i top nie mogą zostać wykonane, jeśli stos jest pusty
- Próba wywołania metody pop i top dla pustego stosu wyrzuci wyjątek EmptyStackException

© 2004 Goodrich, Tamassia

## ZASTOSOWANIA STOSÓW

### Zastosowania bezpośrednie

Historia odwiedzanych stron w przeglądarce internetowej  
Sekwencja operacji „cofnij” w edytorze tekstu  
Łańcuch wywołania metod w Wirtualnej maszynie Java

### Zastosowania pośrednie

Pomocnicza struktura danych dla algorytmów  
Składowa innych struktur danych

© 2004 Goodrich, Tamassia

## STOS BAZUJĄCY NA TABLICY

- Prostym sposobem implementacji Stosu jest zastosowanie tablicy
- Dodajemy elementy od lewej do prawej
- Dodatkowa zmienna kontrolująca indeks elementu na wierzchu stosu

**Algorytm** size()  
**return** t + 1

**Algorytm** pop()  
**if** isEmpty() **then**  
    **throw** EmptyStackException  
**else**  
    t  $\leftarrow$  t - 1  
    **return** S[t + 1]



© 2004 Goodrich, Tamassia

## STOS BAZUJĄCY NA TABLICY

- Tablica przechowująca elementy stosu może się zapieczać
- Operacja push() powinna zatem wyrzucić wyjątek FullStackException
- Ograniczenie związane z implementacją bazującą na tablicy
- Nie zawsze występujące przy implementacji stosu

**Algorytm** push(e)  
**if** t = S.length - 1 **then**  
    **throw** FullStackException  
**else**  
    t  $\leftarrow$  t + 1  
    S[t]  $\leftarrow$  e



© 2004 Goodrich, Tamassia

## WYDAJNOŚĆ I OGRANICZENIA

### Wydajność

Niech n będzie liczbą elementów na stosie

Wykorzystane miejsce będzie równe O(n)

Każda operacja będzie działała w czasie O(1)

### Ograniczenia

Maksymalny rozmiar stosu musi zostać zdefiniowany *a priori* i nie może zostać zmieniony

Próba dodania nowego elementu do pełnego stosu powoduje wyjątki typowe dla danej implementacji

© 2004 Goodrich, Tamassia

## STOS BAZUJĄCY NA POWIĘKSZANEJ TABLICY

- Wykonując operację push, kiedy tablica jest pełna, zamiast wyrzucania wyjątku, możemy zastąpić ją większą tablicą
- Jak duża powinna być nowa tablica?
  - Strategia inkrementalna: zwiększa rozmiar o stałą c
  - Strategia podwajania: podwój rozmiar tablicy

```
Algorytm push(e)
if t = S.length - 1 then
    A ← nowa tablica o rozmiarze...
    for i ← 0 to t do
        A[i] ← S[i]
        S ← A
        t ← t + 1
        S[t] ← e
```

© 2004 Goodrich, Tamassia

## PORÓWNANIE STRATEGII

- Porównamy strategię inkrementalną i podwajania poprzez analizę całkowitego czasu **T(n)** niezbędnego do wykonania n operacji push()
  - Zakładamy, że rozpoczynamy od pustego stosu reprezentowanego przez tablicę o rozmiarze 1
  - Czasem średnim będzie średni czas operacji push niezbędny do wykonania serii operacji, np.:  $T(n)/n$

© 2004 Goodrich, Tamassia

## STRATEGIA INKREMENTALNA

- Tablica zostanie zastąpiona  $k = n/c$  razy
- Całkowity czas  $T(n)$  wykonania n operacji push jest proporcjonalny do:
$$n + c + 2c + 3c + 4c + \dots + kc = \\ n + c(1 + 2 + 3 + \dots + k) = \\ n + ck(k + 1)/2$$
- Ponieważ c jest stałą,  $T(n)$  jest w  $O(n + k^2)$ , tj.  $O(n^2)$
- Średni czas operacji push jest w  $O(n)$

© 2004 Goodrich, Tamassia

## STRATEGIA PODWAJANIA

- Tablica zostanie zastąpiona  $k = \log_2 n$  razy
- Całkowity czas  $T(n)$  wykonania n operacji push jest proporcjonalny do:
$$n + 1 + 2 + 4 + 8 + \dots + 2^k = \\ n + 2^{k+1} - 1 = 2n - 1$$
- $T(n)$  jest w  $O(n)$
- Średni czas operacji push jest w  $O(1)$



© 2004 Goodrich, Tamassia

## INTERFEJS STOSU W C++

- Interfejs jest odniesieniem do ADT stosu
- Wymaga zdefiniowania klasy EmptyStackException
- Najbardziej zbliżoną konstrukcją jest vector

```
template <typename Object>
class Stack{
public:
    int size();
    bool isEmpty();
    Object& top()
        throw(EmptyStackException);
    void push(Object o);
    Object pop()
        throw(EmptyStackException);
}
```

© 2004 Goodrich, Tamassia

## SPRAWDZANIE DOPASOWANIA NAWIASÓW

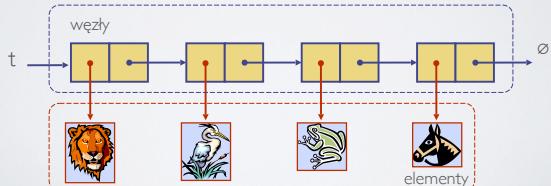
Każdy nawias "()", "{}", lub "[]" musi być sparowany z odpowiadającym mu nawiasem ")", "}" lub "]"

prawidłowo: ( )(( )){[([ )])}  
prawidłowo: (( )( )){[([ )])}  
 błędnie: )(( )){[([ )])}  
 błędnie: ({[ ]})  
 błędnie: (

© 2004 Goodrich, Tamassia

## LISTA JEDNOKIERUNKOWA JAKO STOS

- Możemy zaimplementować stos za pomocą listy jednokierunkowej
- Górnny element jest przechowywany w pierwszym węźle listy
- Wykorzystanie miejsca to  $O(n)$ , a każda operacja stosu jest wykonywana w czasie  $O(1)$



© 2004 Goodrich, Tamassia

# PROJEKTOWANIE I ANALIZA ALGORYTMÓW

KOLEJKA, KOLEJKA PRIORYTETOWA CZ.I,  
OPIS ALGORYTMÓW, ANALIZA ALGORYTMÓW CZ.I

**Wykład 3**

dr inż. Łukasz Jeleń  
Na podstawie wykładów dr. T. Fevensa

## OSTATNIO

- Tablice

S	A	M	P	L	E
0	1	2	3	4	5

- Listy jedno i dwukierunkowe

- Stos

Implementacja:

- na tablicy
- na liście jednokierunkowej

© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
A->setElement(new String("fred"));
```

© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
A->setElement(new String("fred"));
```

© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
A->setElement(new String("fred"));
Node<String> B = null;
```

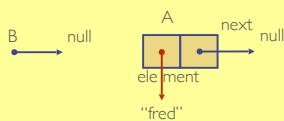
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;
```



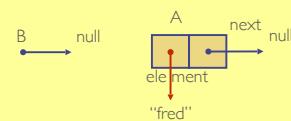
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;  
    B = A; // B jest referencją do obiektu A
```



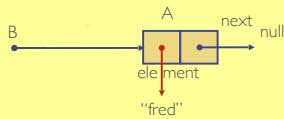
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;  
    B = A; // B jest referencją do obiektu A
```



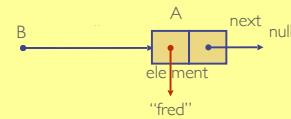
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;  
    B = A; // B jest referencją do obiektu A  
    A->setElement(new String("PAMSI"));
```



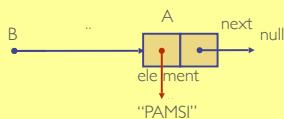
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;  
    B = A; // B jest referencją do obiektu A  
    A->setElement(new String("PAMSI"));
```



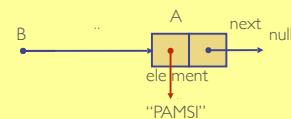
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j., śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
    A->setElement(new String("fred"));  
    Node<String> B = null;  
    B = A; // B jest referencją do obiektu A  
    A->setElement(new String("PAMSI"));  
    B->set(new String("Montréal"));
```



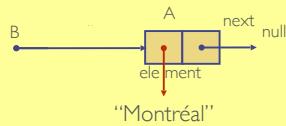
© 2004 Goodrich, Tamassia

## REFERENCJE

Krótkie przypomnienie referencji (t.j. śledzenie powiązań lub wskaźników).

```
Node<String> A = new Node<String>();
```

```
A->setElement(new String("fred"));
Node<String> B = null;
B = A; // B jest referencją do obiektu A
A->setElement(new String("PAMSI"));
B->setElement(new String("Montréal"));
```

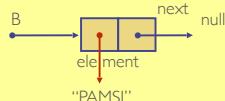


© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWVr",null)); // konstruktor
```

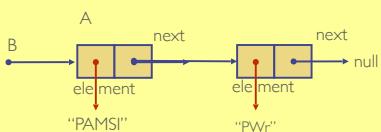


© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWVr",null)); // konstruktor
```



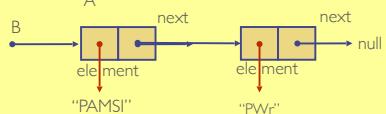
© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWVr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
```



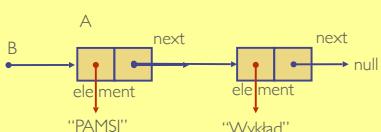
© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWVr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
```



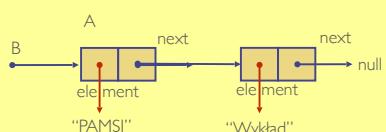
© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWVr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
B = B.getNext();
```



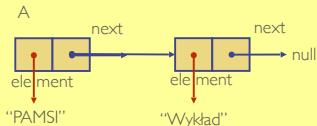
© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
B = B.getNext();
```

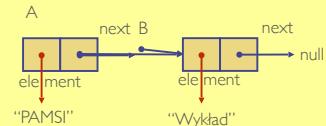

© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
B = B.getNext();
```

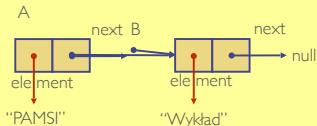

© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
B = B.getNext();
B.setNext(new Node<String>("Laboratorium",null));;
```

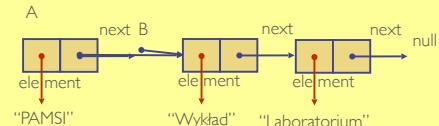

© 2004 Goodrich, Tamassia

## REFERENCJE II

Możemy także śledzić powiązania...

```
A->setNext(new Node<String>("PWr",null)); // konstruktor
```

```
B.getNext().setElement(new String("Wykład"));
B = B.getNext();
B.setNext(new Node<String>("Laboratorium",null));;
```


© 2004 Goodrich, Tamassia

## KOLEJKI



## KOLEJKA ADT

- Kolejka przechowuje dowolne obiekty
- Dodawanie i usuwanie jest wykonywane według zasad FIFO - first-in first-out
- Elementy dodawane są na końcu kolejki, a usuwane z przodu kolejki
- Główne operacje na kolejce:
  - enqueue(element): dodaje element na końcu kolejki
  - element dequeue(): usuwa i zwraca element z początku kolejki
- Dodatkowe operacje:
  - element front(): zwraca element na przodzie listy bez usuwania go
  - integer size(): zwraca ilość przechowywanych elementów
  - boolean isEmpty(): informuje czy w kolejce są przechowywane jakieś elementy
- Wyjątki
  - Próba wywołania dequeue lub front na pustej kolejce wyrzuca EmptyQueueException

© 2004 Goodrich, Tamassia

# ZASTOSOWANIA KOLEJEK

Zastosowanie bezpośrednie

Listy oczekujących,

Dostęp do zasobów współdzielonych (np.: drukarka),

Multiprogramming

Zastosowania pośrednie

Pomocnicza struktura danych dla algorytmów

Składowa innych struktur danych

© 2004 Goodrich, Tamassia

# KOLEJKA BAZUJĄCA NA TABLICY

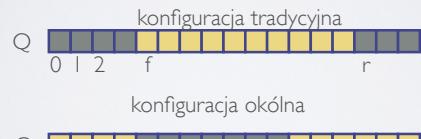
Zastosowanie tablicy o rozmiarze N w sposób okrężny/kolisty

Dwie zmienne kontrolują przód i tył kolejki

f indeks pierwszego elementu

r indeks下一个 do ostatniego elementu

Pozycja r w tablicy jest pusta



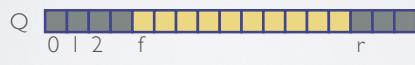
© 2004 Goodrich, Tamassia

# OPERACJE NA KOLEJCE

Wykorzystujemy operator modulo

**Algorytm** size()  
return  $(N - f + r) \bmod N$

**Algorytm** isEmpty()  
return  $(f = r)$

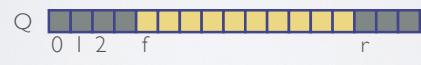


© 2004 Goodrich, Tamassia

# OPERACJE NA KOLEJCE

Operacja enqueue wyrzuca wyjątek jeśli tablica jest pełna. Wyjątek jest zależny od implementacji.

**Algorytm** enqueue(e)  
if size() = N - 1 then  
throw FullQueueException  
else  
Q[r] ← e  
r ← (r + 1) mod N



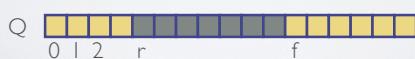
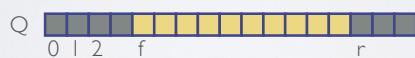
© 2004 Goodrich, Tamassia

# OPERACJE NA KOLEJCE

Operacja dequeue wyrzuca wyjątek jeśli kolejka jest pusta

Ten wyjątek jest określony w ADT dla kolejki

**Algorytm** dequeue()  
if isEmpty() then  
throw  
EmptyQueueException  
else  
temp ← Q[f]  
f ← (f + 1) mod N  
return temp

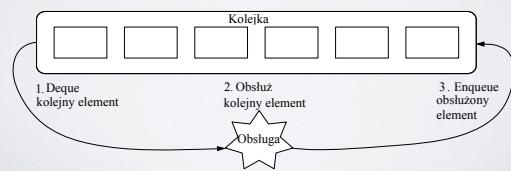


© 2004 Goodrich, Tamassia

# ZASTOSOWANIE : SYMULATOR SYSTEMU KOŁOWEGO

Możemy zaimplementować symulator systemu kołowego (tzw. systemu każdy z każdym) z zastosowaniem kolejki, Q, poprzez wielokrotne wywoływanie następujących kroków:

1. e ← Q.dequeue()
2. Obsługa elementu e
3. Q.enqueue(e)



© 2004 Goodrich, Tamassia

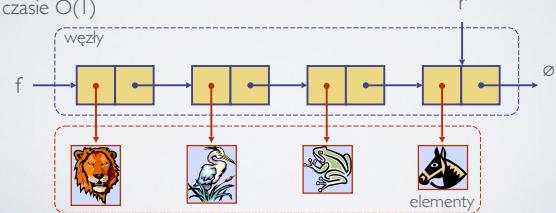
## KOLEJKA BAZUJĄCA NA POWIĘKSZANEJ TABLICY

- Wykonując operację enqueue, kiedy tablica jest pełna, zamiast wyrzucania wyjątku, możemy zastąpić ją większą tablicą
- Analogicznie do procedury, którą omawialiśmy w przypadku stosu
- Operacja enqueue ma średni czas działania:
  - $O(n)$  w przypadku strategii inkrementalnej
  - $O(1)$  w przypadku strategii podwajającej

© 2004 Goodrich, Tamassia

## KOLEJKA NA LIŚCIE JEDNOKIERUNKOWEJ

- Możemy zaimplementować kolejkę za pomocą listy jednokierunkowej
  - Element początkowy jest przechowywany w pierwszym węźle
  - Ostatni element jest przechowywany w ostatnim węźle
- Wykorzystane miejsce to  $O(n)$ , a każda operacja kolejki jest wykonywana w czasie  $O(1)$



© 2004 Goodrich, Tamassia

## KOLEJKI PRIORYTETOWE



## KOLEJKA PRIORYTETOWE - ADT

- Kolejka priorytetowa przechowuje kolekcję wpisów
  - Każdy element jest parą (klucz, wartość)
  - Główne metody kolejki priorytetowej
    - **insert(k, x)**  
dodaje element o kluczu k i wartości x
    - **removeMin()**  
usuwa i zwraca element o najmniejszym kluczu
- Dodatkowe metody:
  - **min()**  
zwraca, ale nie usuwa, element o najmniejszym kluczu
  - **size(), isEmpty()**
- Zastosowania:
  - Aukcje
  - Giełda papierów wartościowych

© 2004 Goodrich, Tamassia

## RELACJE UPORZĄDKOWANIA

- Klucze w kolejce priorytetowej mogą być dowolnymi obiektami, na których podstawie da się zdefiniować uporządkowanie
- Dwa różne wpisy w kolejce priorytetowej mogą posiadać ten sam klucz.
- Matematyczna koncepcja całkowitego uporządkowania  $\leq$ 
  - Właściwość refleksyjna:  
 $x \leq x$
  - Właściwość antysymetryczna:  
 $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Właściwość transzytywna:  
 $x \leq y \wedge y \leq z \Rightarrow x \leq z$

© 2004 Goodrich, Tamassia

## KOMPARATOR

- Komparator porównuje dwa obiekty zgodnie z koncepcją całkowitego uporządkowania
- Uogólniona postać kolejki priorytetowej wykorzystuje komparator
  - definicja sposobu porównywania obiektów
- Komparator jest niezależny od przechowywanych kluczy
  - takie same obiekty mogą zostać posortowane w różny sposób
  - zależny od komparatora

© 2004 Goodrich, Tamassia

## ZASTOSOWANIE KOMPARATORA W C++

- Klasa komparatora przeciąża operator "()" funkcją porównującą
- Przykład: Porównaj leksykograficznie dwa punkty na płaszczyźnie

```
class LexCompare{
public:
    int operator()(Point a, Point b){
        if (a.x < b.x) return -1
        else if (a.x > b.x) return 1
        else if (a.y < b.y) return -1
        else if (a.y > b.y) return 1
        else return 0;
    }
};
```

© 2004 Goodrich, Tamassia

- W celu wykorzystania komparatora należy zdefiniować obiekt tego typu i wywołać jego operator "()"
- Przykład:

```
Point p(2.3, 4.5);
Point q(1.7, 7.3);
LexCompare lexCompare;
if (lexCompare(p,q) < 0)
    cout<< "p jest mniejsze od q";
else if (lexCompare(p,q) == 0)
    cout<< "p jest równe q";
else if (lexCompare(p,q) > 0)
    cout<< "p jest większe od q";
```

## SORTOWANIE Z ZASTOSOWANIEM KOLEJEK PRIORYTETOWYCH

- Mogimy wykorzystać kolejkę priorytetową do posortowania zbioru porównywalnych elementów

- Pojedynczo umieść elementy w kolejce
- Usuń elementy z wykorzystaniem serii operacji removeMin

- Złożoność obliczeniowa takiego sortowania jest zależna od implementacji kolejki priorytetowej

### Algorytm PriorityQueueSort( $S, P$ )

**Wejście:** sekwencja  $S$ , kolejka priorytetowa  $P$  wykorzystująca metodę całkowitego uporządkowania kluczy  
**Output:** posortowana sekwencja  $S$  z zastosowaniem metody całkowitego uporządkowania

```
while !S.isEmpty() do
    e ← S.removeFirst()
    P.insert(e, null)
    while !P.isEmpty() do
        e ← P.removeMin().getKey()
        S.addLast(e)
```

© 2004 Goodrich, Tamassia

## KOLEJKA BAZUJĄCA NA LIŚCIE

- Implementacja z wykorzystaniem nieposortowanej listy



### Wydajność:

- umieszczań elementów zajmuje  $O(1)$
- możemy umieszczać elementy na początku i na końcu
- removeMin i min zajmują  $O(n)$
- musimy przeskanować całą listę w celu odnalezienia najmniejszego klucza

- Implementacja z wykorzystaniem posortowanej listy



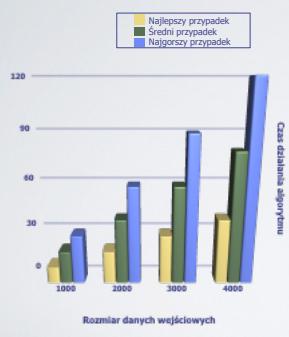
### Wydajność:

- umieszczań elementów zajmuje  $O(n)$
- musimy znaleźć miejsce gdzie możemy dodać nowy element
- removeMin i min zajmują  $O(1)$
- najmniejszy element znajduje się na początku listy

© 2004 Goodrich, Tamassia

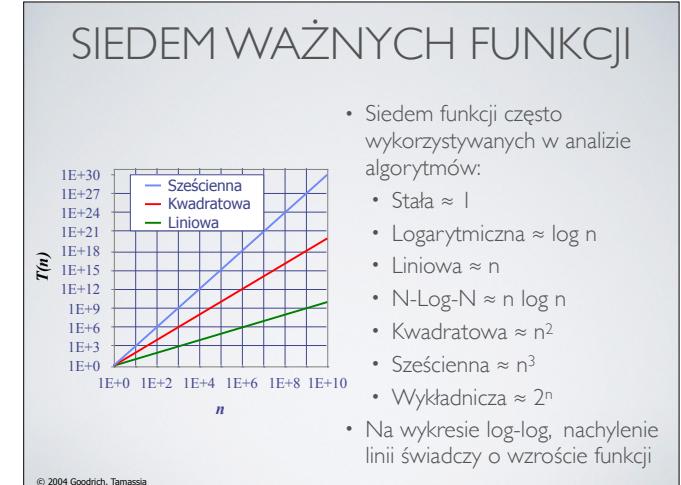
## ZŁOŻONOŚĆ OBLCZENIOWA

## ZŁOŻONOŚĆ OBLCZENIOWA



© 2004 Goodrich, Tamassia

- Większość algorytmów przekształca obiekty wejściowe w obiekty wyjściowe
- Czas działania (złożoność obliczeniowa) algorytmu zazwyczaj wzrasta wraz z rozmiarem danych wejściowych
- Średni czas działania jest najczęściej trudny do określenia
- koncentrujemy się na przypadku najgorszym
  - łatwiejszy do analizy
  - Istotny w aplikacjach takich jak gry, finanse i robotyka

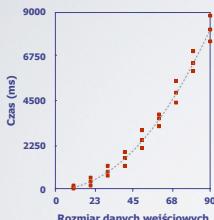


## SIEDEM WAŻNYCH FUNKCJI

- Siedem funkcji często wykorzystywanych w analizie algorytmów:
  - Stała  $\approx 1$
  - Logarytmiczna  $\approx \log n$
  - Liniowa  $\approx n$
  - $N\text{-Log-}N \approx n \log n$
  - Kwadratowa  $\approx n^2$
  - Sześcienna  $\approx n^3$
  - Wykładnicza  $\approx 2^n$
- Na wykresie log-log, nachylenie linii świadczy o wzroście funkcji

## DOŚWIADCZENIA

- Napisz program implementującą algorytm
- Przetestuj napisany program na danych o różnych rozmiarach
- Wykorzystaj metodę typu `System.currentTimeMillis()` do dokładnego oszacowania czasu działania algorytmu
- Zrób wykres dla otrzymanych wyników.



© 2004 Goodrich, Tamassia

## OGRANICZENIE EKSPERYMENTÓW

- Niezbędne jest zaimplementowanie algorytmu, który może być trudny
- Wyniki złożoności obliczeniowej mogą nie być znaczące dla danych wejściowych, które nie były wykorzystywane w eksperymentach
- **W celu porównania dwóch algorytmów należy korzystać z tego samego sprzętu i oprogramowania**

© 2004 Goodrich, Tamassia

## ANALIZA TEORETYCZNA



- Wykorzystuje formalną reprezentację algorytmu zamiast implementacji
- Charakteryzuje złożoność obliczeniową jako funkcję rozmiaru danych wejściowych,  $n$
- Bierze pod uwagę wszystkie możliwe dane wejściowe
- Pozwala nam na ocenę szybkości działania algorytmu niezależnie od sprzętu/oprogramowania

© 2004 Goodrich, Tamassia

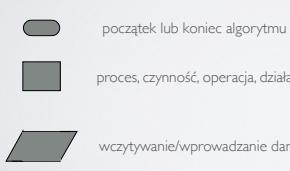
## DEFINICJA

- Złożoność obliczeniowa algorytmu  $A$  jest zdefiniowana przez:
  - $t$  - czas - ilość operacji niezbędnych do rozwiązania dowolnej instancji  $I$  problemu o rozmiarze  $N(I)$  przez algorytm  $A \Rightarrow N(I) = n$
  - $f_A(n) = \max(t)$
- Nas interesuje jak wygląda funkcja  $F_A$ , a nie jej wartości

© 2004 Goodrich, Tamassia

## METODY REPREZENTACJI ALGORYTMÓW

- Pseudo kod
- Graficznie
  - Schematy blokowe



© 2004 Goodrich, Tamassia

## PSEUDO KOD

- Uogólniony opis algorytmu
- Bardziej strukturalny niż opis w języku polskim
- Mniej szczegółowy od programu komputerowego
- Preferowana notacja do opisu algorytmów
- Ukrywa aspekty projektowania programu

Przykład: znajdź element max tablicy

**Algorytm** `tabMax(T, n)`  
**Wejście** tab  $T$  zawierająca  $n$  integerów  
**Wyjście** element maksymalny  $T$   
`biezaceMax  $\leftarrow T[0]$`   
`for  $i \leftarrow 1$  to  $n - 1$  do`  
    `if  $T[i] > biezaceMax$  then`  
        `biezaceMax  $\leftarrow T[i]$`   
`return biezaceMax`

© 2004 Goodrich, Tamassia

## DETALLE PSEUDOCODU

- Kontrola działania
    - **if ... then ... [else ...]**
    - **while ... do ...**
    - **repeat ... until ...**
    - **for ... do ...**
    - Wcięcia zastępują nawiasy
  - Deklaracja metod
  - **Algorytm** *metoda (arg [, arg...])*
    - Wejście ...**
    - Wyjście**
- © 2004 Goodrich, Tamassia

- Wywołanie metody
- *zm.metoda (arg [, arg...])*
- Zwracanie wartości
- **return wyrażenie**
- Wyrażenia
  - ← Przypisanie  
(tak jak = w C++/Java)
  - = Testowanie równości  
(tak jak == w C++/Java)
  - n<sup>2</sup>** Superskrypty i inne matematyczne formatowanie jest dozwolone

## OPERACJE PODSTAWOWE



- Podstawowe obliczenia są wykonywane przez algorytm
  - Identyfikowane w pseudokodzie
  - Niezależne od języka programowania
  - Dokładna definicja nie jest istotna (pozniej zobaczymy dlaczego)
  - Z założenia pobierają stałą ilość pamięci oraz wykonywane są w ścisłe określonym czasie
- © 2004 Goodrich, Tamassia
- Przykłady:
    - Wykonywanie wyrażeń
    - Przypisanie wartości do zmiennej
    - Indeksowanie tablicy
    - Wywołanie metody
    - Powrót z metody

## ZLICZANIE OPERACJI PODSTAWOWYCH

- Badając pseudokod możemy określić maksymalną ilość operacji podstawowych wykonywanych przez algorytm w funkcji n - rozmiaru danych wejściowych

<b>Algorytm</b> <i>tabMax(T, n)</i> <pre> biezacyMax ← T[0] for i ← 1 to n - 1 do   if T[i] &gt; biezacyMax then     biezacyMax ← T[i] { zwiększenie licznika i } return biezacyMax </pre>	il. operacji <table border="0"> <tr> <td>2</td> </tr> <tr> <td>2n</td> </tr> <tr> <td>2(n - 1)</td> </tr> <tr> <td>2(n - 1)</td> </tr> <tr> <td>1</td> </tr> </table> Suma $8n - 3$	2	2n	2(n - 1)	2(n - 1)	1
2						
2n						
2(n - 1)						
2(n - 1)						
1						

© 2004 Goodrich, Tamassia

## OKREŚLANIE ZŁOŻONOŚCI OBLCZENIOWEJ - CZASU DZIAŁANIA ALGORYTMU

- Algorytm tabMax wykonuje  $8n - 3$  operacji podstawowych w najgorszym przypadku. Zdefiniujmy:
    - a = Czas wykonania najszybszej operacji podstawowej
    - b = Czas wykonania najwolniejszej operacji podstawowej
  - Niech **T(n)** będzie najgorszym czasem tabMax. Wtedy
 
$$a(8n - 3) \leq T(n) \leq b(8n - 3)$$
  - Zatem, czas T(n) jest ograniczony przez dwie funkcje liniowe
- © 2004 Goodrich, Tamassia

## WSPÓŁCZYNNIK WZROSTU ZŁOŻONOŚCI OBLCZENIOWEJ

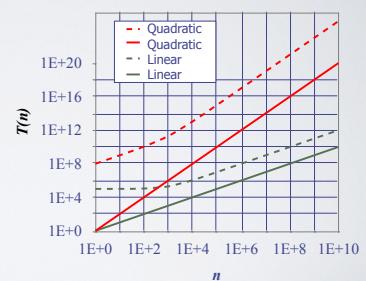


- Zmiana środowiska sprzętowego/oprogramowania
  - Ma stały wpływ na T(n), ale
    - nie ma wpływu na współczynnik wzrostu T(n)
- Liniowy wzrost czasu działania T(n) jest istotną właściwością algorytmu tabMax

© 2004 Goodrich, Tamassia

## SKŁADOWA STAŁA

- Asymptotyczny współczynnik wzrostu nie zależy od:
    - składowych stałych lub
    - wyrażeń niższego rzędu
  - Przykłady
    - $10^2n + 10^5$  jest funkcją liniową
    - $10^5n^2 + 10^8n$  jest funkcją kwadratową
- © 2004 Goodrich, Tamassia



## NOTACJA DUŻE O

- Mając daną funkcję  $f(n)$  i  $g(n)$  mówimy, że  $f(n)$  należy do  $\mathcal{O}(g(n))$  jeśli istnieją stałe nieujemne  $c$  i  $n_0$  takie, że

$$f(n) \leq cg(n) \text{ dla } n \geq n_0$$

- Przykład:  $2n + 10$  jest w  $\mathcal{O}(n)$ 
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10/(c - 2)$
  - Weźmy  $c = 3$  i  $n_0 = 10$

© 2004 Goodrich, Tamassia

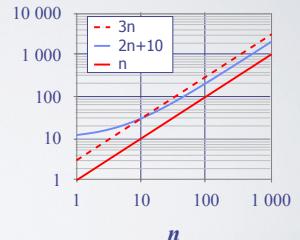
## NOTACJA DUŻE O

- Mając daną funkcję  $f(n)$  i  $g(n)$  mówimy, że  $f(n)$  należy do  $\mathcal{O}(g(n))$  jeśli istnieją stałe nieujemne  $c$  i  $n_0$  takie, że

$$f(n) \leq cg(n) \text{ dla } n \geq n_0$$

- Przykład:  $2n + 10$  jest w  $\mathcal{O}(n)$ 
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq 10/(c - 2)$
  - Weźmy  $c = 3$  i  $n_0 = 10$

© 2004 Goodrich, Tamassia



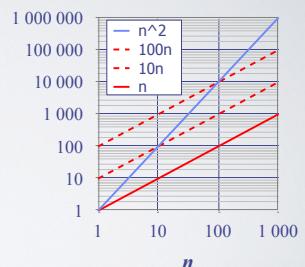
## PRZYKŁAD DUŻEGO O

- Przykład: funkcja  $n^2$  nie należy do  $\mathcal{O}(n)$ 
  - $n^2 \leq cn$
  - $n \leq c$
  - Powyższa nierówność nie może zostać spełniona ponieważ  $c$  musi być stałe

© 2004 Goodrich, Tamassia

## PRZYKŁAD DUŻEGO O

- Przykład: funkcja  $n^2$  nie należy do  $\mathcal{O}(n)$ 
  - $n^2 \leq cn$
  - $n \leq c$
  - Powyższa nierówność nie może zostać spełniona ponieważ  $c$  musi być stałe



## WIĘCEJ PRZYKŁADÓW

- $7n^2$
- $3n^3 + 20n^2 + 5$
- $3 \log n + 5$

© 2004 Goodrich, Tamassia

## WIĘCEJ PRZYKŁADÓW

- $7n^2$   
 $7n^2$  jest w  $\mathcal{O}(n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $7n^2 \leq cn$  dla  $n \geq n_0$   
spełnione dla  $c = 7$  i  $n_0 = 1$
- $3n^3 + 20n^2 + 5$
- $3 \log n + 5$

© 2004 Goodrich, Tamassia

## WIĘCEJ PRZYKŁADÓW

- **7n-2**

$7n-2$  jest w  $O(n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $7n-2 \leq c \cdot n$  dla  $n \geq n_0$   
spełnione dla  $c = 7$  i  $n_0 = 1$

- **$3n^3 + 20n^2 + 5$**

$3n^3 + 20n^2 + 5$  jest w  $O(n^3)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $3n^3 + 20n^2 + 5 \leq c \cdot n^3$  dla  $n \geq n_0$   
spełnione dla  $c = 4$  i  $n_0 = 21$

- **$3 \log n + 5$**

© 2004 Goodrich, Tamassia

## WIĘCEJ PRZYKŁADÓW

- **7n-2**

$7n-2$  jest w  $O(n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $7n-2 \leq c \cdot n$  dla  $n \geq n_0$   
spełnione dla  $c = 7$  i  $n_0 = 1$

- **$3n^3 + 20n^2 + 5$**

$3n^3 + 20n^2 + 5$  jest w  $O(n^3)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $3n^3 + 20n^2 + 5 \leq c \cdot n^3$  dla  $n \geq n_0$   
spełnione dla  $c = 4$  i  $n_0 = 21$

- **$3 \log n + 5$**

$3 \log n + 5$  jest w  $O(\log n)$   
potrzebujemy  $c > 0$  i  $n_0 \geq 1$  takie, że  $3 \log n + 5 \leq c \cdot \log n$  dla  $n \geq n_0$   
spełnione dla  $c = 8$  i  $n_0 = 2$

© 2004 Goodrich, Tamassia

## DUŻE O I WSPÓŁCZYNNIK WZROSTU

- Notacja duże O daje nam górne ograniczenie współczynnika wzrostu funkcji.
- Określenie “ $f(n)$  jest w  $O(g(n))$ ” oznacza, że współczynnik wzrostu funkcji  $f(n)$  jest nie większy niż współczynnik wzrostu funkcji  $g(n)$
- Mogliśmy wykorzystać notację duże O do porównywania (stopniowania) funkcji względem ich współczynnika wzrostu

$f(n)$ jest w $O(g(n))$	$g(n)$ jest w $O(f(n))$	
Tak	Nie	$g(n)$ rośnie szybciej
Nie	Tak	$f(n)$ rośnie szybciej
Tak	Tak	ten sam wzrost

© 2004 Goodrich, Tamassia

## DUŻE O I WSPÓŁCZYNNIK WZROSTU

- Notacja duże O daje nam górne ograniczenie współczynnika wzrostu funkcji.
- Określenie “ $f(n)$  jest w  $O(g(n))$ ” oznacza, że współczynnik wzrostu funkcji  $f(n)$  jest nie większy niż współczynnik wzrostu funkcji  $g(n)$
- Mogliśmy wykorzystać notację duże O do porównywania (stopniowania) funkcji względem ich współczynnika wzrostu

$f(n)$ jest w $O(g(n))$	$g(n)$ jest w $O(f(n))$	
Tak	Nie	$\rightarrow g(n)$ rośnie szybciej
Nie	Tak	$f(n)$ rośnie szybciej
Tak	Tak	ten sam wzrost

© 2004 Goodrich, Tamassia

## DUŻE O I WSPÓŁCZYNNIK WZROSTU

- Notacja duże O daje nam górne ograniczenie współczynnika wzrostu funkcji.
- Określenie “ $f(n)$  jest w  $O(g(n))$ ” oznacza, że współczynnik wzrostu funkcji  $f(n)$  jest nie większy niż współczynnik wzrostu funkcji  $g(n)$
- Mogliśmy wykorzystać notację duże O do porównywania (stopniowania) funkcji względem ich współczynnika wzrostu

$f(n)$ jest w $O(g(n))$	$g(n)$ jest w $O(f(n))$	
Tak	Nie	$\rightarrow g(n)$ rośnie szybciej
Nie	Tak	$\rightarrow f(n)$ rośnie szybciej
Tak	Tak	ten sam wzrost

© 2004 Goodrich, Tamassia

## DUŻE O I WSPÓŁCZYNNIK WZROSTU

- Notacja duże O daje nam górne ograniczenie współczynnika wzrostu funkcji.
- Określenie “ $f(n)$  jest w  $O(g(n))$ ” oznacza, że współczynnik wzrostu funkcji  $f(n)$  jest nie większy niż współczynnik wzrostu funkcji  $g(n)$
- Mogliśmy wykorzystać notację duże O do porównywania (stopniowania) funkcji względem ich współczynnika wzrostu

$f(n)$ jest w $O(g(n))$	$g(n)$ jest w $O(f(n))$	
Tak	Nie	$\rightarrow g(n)$ rośnie szybciej
Nie	Tak	$\rightarrow f(n)$ rośnie szybciej
Tak	Tak	ten sam wzrost

© 2004 Goodrich, Tamassia

## ZASADY NOTACJI DUŻE O

- Jeśli  $f(n)$  jest wielomianem stopnia  $d$ , np.:

$f(n) = c_d n^d + c_{d-1} n^{d-1} + \dots + c_1 n^1 + c_0 n^0$ , to  $f(n)$  jest w  $O(n^d)$ , np.:

1. Pomiń wyrażenia niskiego stopnia

2. Pomiń stałe

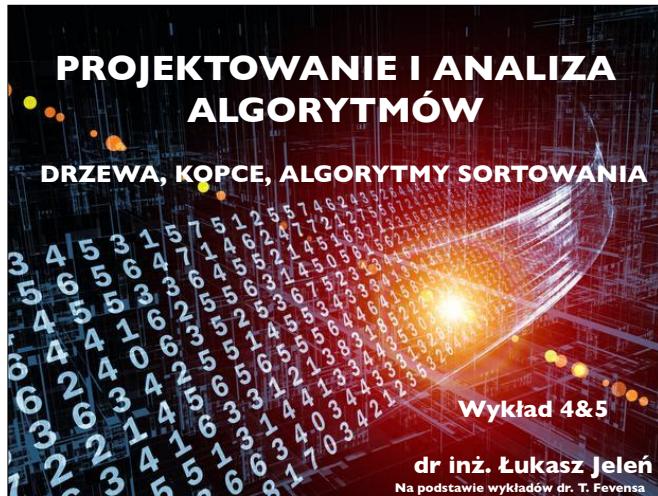
- Wykorzystaj najmniejszą możliwą klasę funkcji

- Powiemy "2n jest w  $O(n)$ " zamiast "2n jest w  $O(2n)$ "

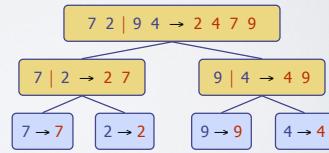
- Wykorzystaj najprostsze wyrażenie tej klasy

- Powiemy "3n + 5 jest w  $O(n)$ " zamiast "3n + 5 jest w  $O(3n)$ "

© 2004 Goodrich, Tamassia



## SORTOWANIE PRZEZ SCALANIE



## TECHNIKA DZIEL I ZWYCIĘŻAJ

**Dziel i zwyciężaj** jest ogólnym paradigmatem projektowania algorytmów:

- **Podział:** podziel dane wejściowe S na dwa rozłączne podzbiory S<sub>1</sub> i S<sub>2</sub>
- **Rekurencja:** rozwiąż problem dla S<sub>1</sub> i S<sub>2</sub>
- **Scalanie:** połącz rozwiązania dla S<sub>1</sub> i S<sub>2</sub> w jedno rozwiązanie dla S

Krokiem podstawowym rekurencji są podproblemy o rozmiarze 0 lub 1

Sortowanie **przez scalanie** jest algorytmem sortującym bazującym na technice dziel i zwyciężaj Podobnie jak dla sortowania kopcowego

- wykorzystuje komparator
- Posiada złożoność O(n log n)

Inaczej niż dla sortowania kopcowego

- Nie wykorzystuje zewnętrznej kolejki priorytetowej
- Pobiera dane w sposób sekwencyjny (odpowiedni do sortowania danych na dysku zewnętrzny)

**Algorytm mergeSort(S, C)**  
**Wejście** lista S z n elementami, komparator C  
**Wyjście** lista S posortowana zgodnie z komparatorem C

```

if S.size() > 1
    (S1, S2) ← podziel(S, n/2)
    mergeSort(S1, C)
    mergeSort(S2, C)
    S ← pusta lista
        połącz(S1, S2, S)
return S

```

$$|S_1| = \left\lceil \frac{n}{2} \right\rceil \text{ and } |S_2| = \left\lfloor \frac{n}{2} \right\rfloor$$

© 2004 Goodrich, Tamassia

## SORTOWANIE PRZEZ SCALANIE

Sortowanie przez scalanie listy S zawierającej n elementów składa się z trzech kroków:

- **Podział:** podział S na dwie sekwencje S<sub>1</sub> i S<sub>2</sub> zawierającymi ok. n/2 elementów każda
- **Rekurencja:** posortuj rekurencyjnie S<sub>1</sub> i S<sub>2</sub>
- **Scalanie:** połącz S<sub>1</sub> i S<sub>2</sub> w jedną posortowaną listę

© 2004 Goodrich, Tamassia

## ŁĄCZENIE DWÓCH POSORTOWANYCH SEKWENCJI

Ostatni krok sortowania przez scalanie składa się ze scalania dwóch posortowanych sekwencji S<sub>1</sub> i S<sub>2</sub> zaimplementowanych jako lista w jedną posortowaną sekwencję S zawierającą połączenie elementów z S<sub>1</sub> i S<sub>2</sub>

Scalanie dwóch posortowanych sekwencji zawierających po n/2 elementów i zaimplementowane z zastosowaniem listy dwukierunkowej zabiera O(n) czasu

**Algorytm merge(S<sub>1</sub>, S<sub>2</sub>, S)**  
**Wejście** sekwencje S<sub>1</sub> i S<sub>2</sub> zawierające po n/2 elementów, pusta sekwencja S  
**Wyjście** posortowana sekwencja S: S<sub>1</sub> ∪ S<sub>2</sub>

```

while ~S1.isEmpty() ∧ ~S2.isEmpty()
    if S1.first().element() ≤ S2.first().element()
        S.addLast(S1.remove(S1.first()))
    else
        S.addLast(S2.remove(S2.first())))
    while ~S1.isEmpty()
        S.addLast(S1.remove(S1.first())))
    while ~S2.isEmpty()
        S.addLast(S2.remove(S2.first())))

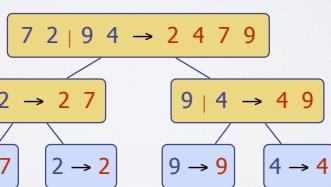
```

© 2004 Goodrich, Tamassia

## DRZEWO SORTOWANIA PRZEZ SCALANIE

Działanie sortowania przez scalanie jest zobrazowane przez drzewo binarne

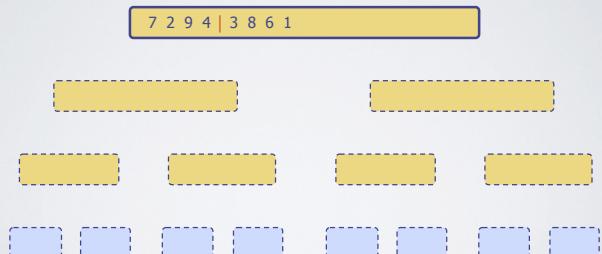
- każdy węzeł reprezentuje wywołanie rekurencyjne sortowania i zawiera
- nieposortowaną sekwencję przez wywołaniem i podziałem
- posortowaną sekwencję po zakończeniu wywołania
- korzeń jest początkowym wywołaniem
- liście są wywołaniami podsekwencji o rozmiarze 0 lub 1



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

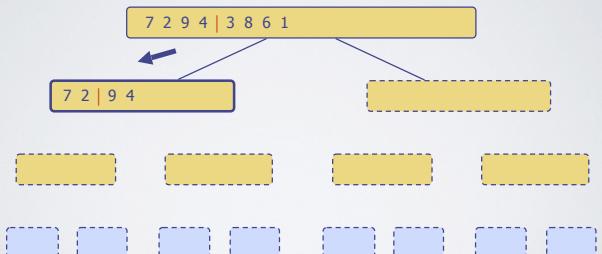
Podział



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

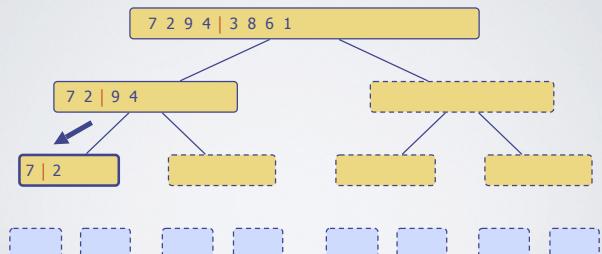
Wywołanie rekurencyjne, podział



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

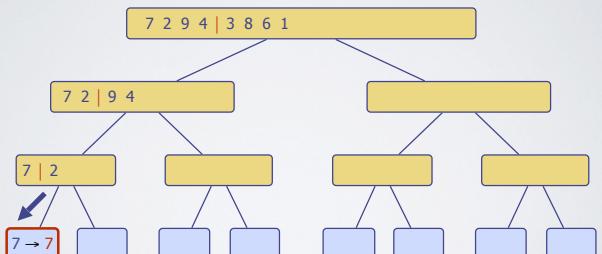
Wywołanie rekurencyjne, podział



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

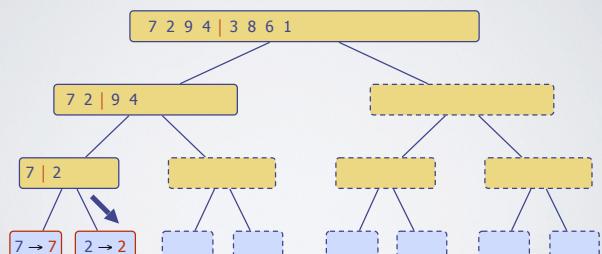
Wywołanie rekurencyjne, podział



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

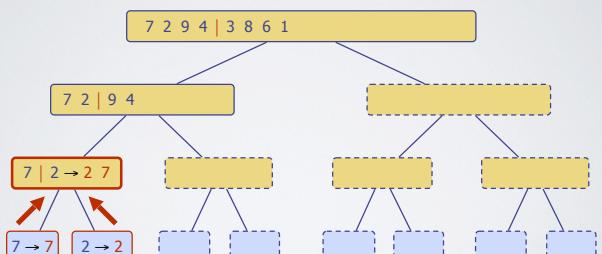
Wywołanie rekurencyjne, krok podstawowy



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

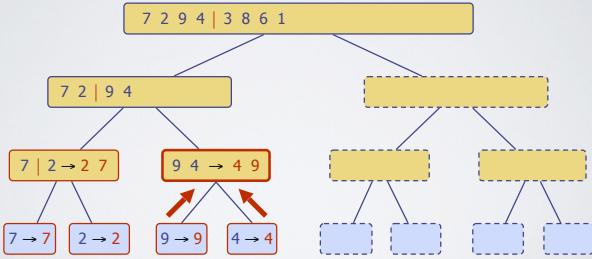
Scalanie



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

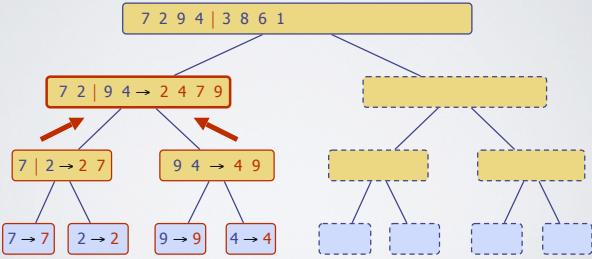
Wywołanie rekurencyjne, ..., krok podstawowy, scalanie



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

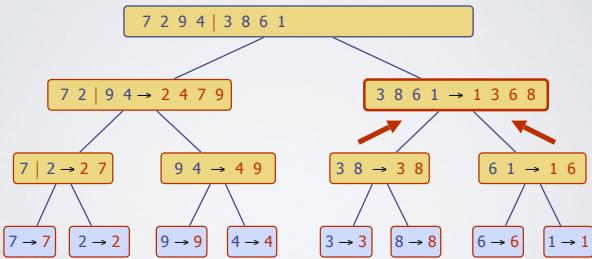
Scalanie



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

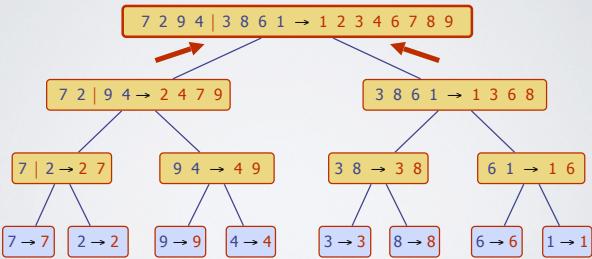
Wywołanie rekurencyjne, ..., krok podstawowy, scalanie



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

Scalanie



© 2004 Goodrich, Tamassia

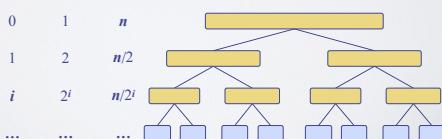
## ANALIZA SORTOWANIA PRZEZ SCALANIE



Wysokość drzewa sortowania to  $O(\log n)$

- przy każdym wywołaniu rekurencyjnym dzielimy sekwencję na pół
  - ilość operacji wykonywanych na węzłach na poziomie i to  $O(n)$
  - dzielimy i scalmy  $2^i$  sekwencji o rozmiarze  $n/2^i$
  - wykonujemy  $2^{i+1}$  wywołań rekurencyjnych
- Zatem, całkowita złożoność obliczeniowa sortowania przez scalanie wynosi  $O(n \log n)$

poziom #sekw rozmiar



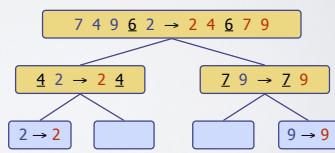
© 2004 Goodrich, Tamassia

## PODSUMOWANIE ALGORYTMÓW SORTOWANIA

Algorytm	Złożoność	Uwagi
przez wybór	$O(n^2)$	<ul style="list-style-type: none"> <li>• wolne</li> <li>• in-situ</li> <li>• dla małych tablic (&lt; 1K)</li> </ul>
przez wstawianie	$O(n^2)$	<ul style="list-style-type: none"> <li>• wolne</li> <li>• in-situ</li> <li>• dla małych tablic (&lt; 1K)</li> </ul>
przez kopcowanie	$O(n \log n)$	<ul style="list-style-type: none"> <li>• szybkie</li> <li>• in-situ (impl. na tablicy)</li> <li>• dla średnich tablic (1K – 1M)</li> </ul>
przez scalanie	$O(n \log n)$	<ul style="list-style-type: none"> <li>• szybkie</li> <li>• sekwencyjny dostęp do danych</li> <li>• dla b. dużych tablic (&gt; 1M)</li> </ul>

© 2004 Goodrich, Tamassia

## SORTOWANIE SZYBKIE



## SORTOWANIE SZYBKIE

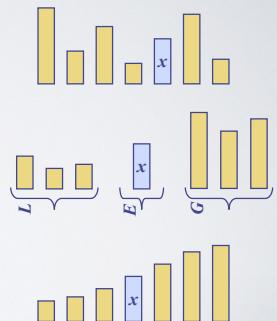
Sortowanie szybkie jest (losowym) algorytmem sortującym bazującym na technice dziel i zwycięzaj:

**Podział:** wybierz losowy element  $x$  (nazywany **piwotem**) i podziel  $S$  na

- $L$  - elementy mniejsze od  $x$
- $E$  - elementy równe  $x$
- $G$  - elementy większe od  $x$

**Rekurencja:** posortuj  $L$  i  $G$

**Scalanie:** połącz  $L$ ,  $E$  i  $G$



© 2004 Goodrich, Tamassia

## PODZIAŁ

Dzielimy sekwencję wejściową w następujący sposób:

- Usuwamy element  $y$  z  $S$  i
- Wstawiamy  $y$  do  $L$ ,  $E$  lub  $G$  w zależności od porównania z piwotem  $x$

Wszystkie operacje wstawiania i usuwania wykonywane są na początku lub na końcu sekwencji, a zatem ich czas działania wynosi  $O(1)$

Zatem czas działania kroku dzielącego sortowania szybkiego wyniesie  $O(n)$

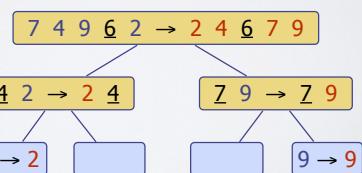
**Algorytm *partition*( $S, p$ )**  
**Wejście:** sekwencja  $S$ , pozycja  $p$  piwota  
**Wyjście:** podsekwencje  $L, E, G$  z elementami z  $S$  mniejszymi, równymi, lub większymi od piwota  
 $L, E, G \leftarrow$  puste sekwencje  
 $x \leftarrow S.remove(p)$   
**while**  $\sim S.isEmpty()$   
 $y \leftarrow S.remove(S.first())$   
**if**  $y < x$   
     $L.addLast(y)$   
**else if**  $y = x$   
     $E.addLast(y)$   
**else** {  $y > x$  }  
     $G.addLast(y)$   
**return**  $L, E, G$

© 2004 Goodrich, Tamassia

## DRZEWO SORTOWANIA

Działanie sortowania szybkiego może być zilustrowane za pomocą drzewa binarnego

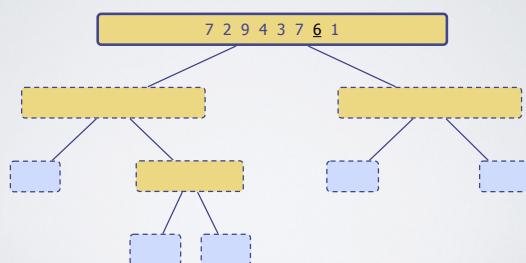
- Każdy węzeł reprezentuje wywołanie rekurencyjne i przechowuje:
  - Nieposortowaną sekwencję przed wywołaniem oraz piwot
  - Posortowaną sekwencję po wywołaniu
- Korzeń jest wywołaniem pierwotnym
- Liście są wywołaniami podsekwencji o rozmiarze 0 lub 1



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

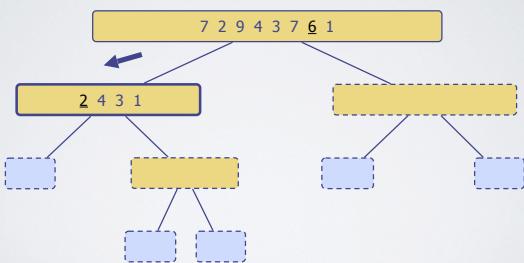
Wybór piwota



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

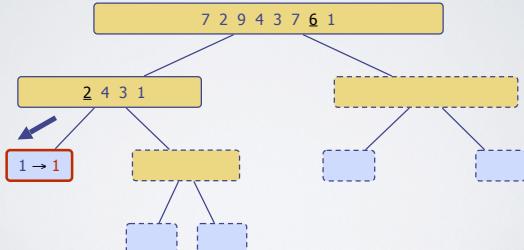
Podział, wywołanie rekurencyjne, wybór piwota



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

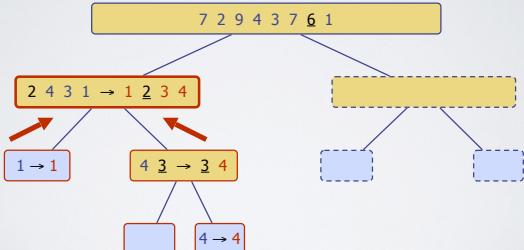
Podział, wywołanie rekurencyjne, wybór piwota



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

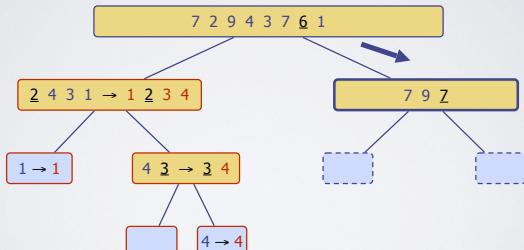
Wywołanie rekurencyjne, ..., krok podstawowy, scalanie



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

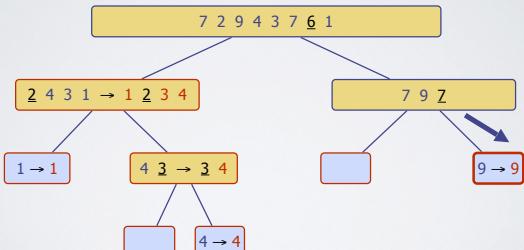
Wywołanie rekurencyjne, wybór piwota



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

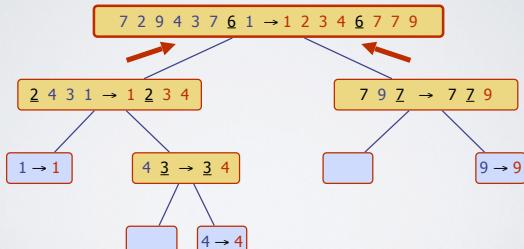
Podział, wywołanie rekurencyjne, wybór piwota



© 2004 Goodrich, Tamassia

## PRZYKŁAD DZIAŁANIA

Scalanie, scalanie



© 2004 Goodrich, Tamassia

## NAJGORSZA ZŁOŻONOŚĆ OBLCZENIOWA

Z najgorszym czasem sortowania szybkiego mamy do czynienia, gdy piwot jest unikalnym minimalnym lub maksymalnym elementem

Jedna z list L i G ma rozmiar  $n - 1$ , a druga ma rozmiar 0

Złożoność obliczeniowa jest proporcjonalna do sumy:

$$n + (n - 1) + \dots + 2 + 1$$

Zatem najgorszy czas tego sortowania to  $O(n^2)$

poziom czas

0 n

1 n - 1

...

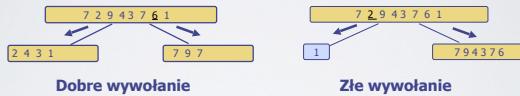
n - 1 1

© 2004 Goodrich, Tamassia

# OCZEKIWANA ZŁOŻONOŚĆ OBLCZENIOWA

Rozważmy wywołanie rekurencyjne sortowania szybkiego na sekwencji o rozmiarze s

- Dobre wywołanie: rozmiary L i G są mniejsze od  $3s/4$
- Złe wywołanie: jedna z sekwencji ma rozmiar większy niż  $3s/4$



Dobre wywołanie

Złe wywołanie

Wywołanie jest dobre z prawdopodobieństwem  $1/2$

- $1/2$  z możliwych piwotów skutkuje dobrym wywołaniem



Złe piwoty

Dobre piwoty

Złe piwoty

© 2004 Goodrich, Tamassia

# OCZEKIWANA ZŁOŻONOŚĆ OBLCZENIOWA

**Fakt probabilistyczny:** Oczekiwana ilość rzutów monetą wymagana do wyrzucenia k reszek wynosi  $2k$

Dla węzła na poziomie i, możemy oczekiwać:

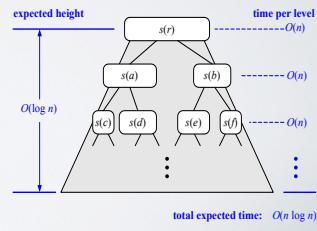
- $i/2$  potomków będzie dobrym wywołaniem
- Rozmiar sekwencji wejściowej dla danego wywołania wynosi najwyżej  $(3/4)^{i/2}$

Mamy zatem:

- Dla węzła o poziomie  $2\log_{4/3}n$ , oczekiwany rozmiar wejścia to jeden
- Oczekiwana wysokość drzewa sortowania to  $O(\log n)$

Ilość obliczeń wykonywanych w węzłach o tym samym poziomie wynosi  $O(n)$

Zatem oczekiwana złożoność obliczeniowa sortowania szybkiego jest w  $O(n \log n)$



© 2004 Goodrich, Tamassia

# SORTOWANIE SZYBKIE IN-SITU



Sortowanie szybkie może być zaimplementowane techniką in-situ.  
W kroku dzielącym zmodyfikujemy operacje tak, aby przeorganizować elementy w tablicy w taki sposób, że

- elementy  $\leq$  pivot będą posiadały indeks  $< i$
- pivot posiada indeks  $i$
- elementy  $\geq$  pivot będą posiadały indeks  $> i$

Wywołanie rekurencyjne będzie zawierać

- elementy z indeksami  $< i$
- elementy z indeksami  $> i$

**Algorytm *inPlaceQuickSort*( $S, a, b$ )**  
**Wejście:** lista  $S$ , indeksy  $a$  i  $b$   
**Wyjście:** lista  $S$  z elementami o indeksach między  $a$  i  $b$  przeorganizowana w sposób rosnący  
*if*  $a \geq b$   
*return*  
 $i \leftarrow$  losowy integer między  $a$  i  $b$   
*S.swapElements*( $i, b$ ) {pivot na końcu}  
 $i \leftarrow$  *inPlacePartition*( $a, b$ )  
*inPlaceQuickSort*( $S, a, i - 1$ )  
*inPlaceQuickSort*( $S, i + 1, b$ )

© 2004 Goodrich, Tamassia

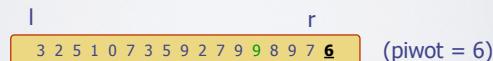
# PODZIAŁ IN-SITU

Najpierw należy wybrać pivot na indeksie  $i$  między  $a$  i  $b$ .



Zamień pivot z elementem na indeksie  $b$ .

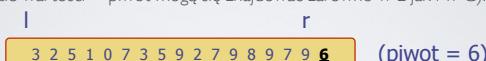
Umieść indeks startowy  $i$  na indeksie  $a$ , a indeks  $r$  na indeksie  $b - i$ .



© 2004 Goodrich, Tamassia

# PODZIAŁ IN-SITU

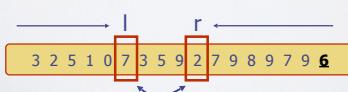
Dokonaj podziału z zastosowaniem dwóch indeksów do rozdzielenia  $S$  na  $L$  i  $G$  (pozostałe wartości = pivot mogą się znajdować zarówno w  $L$  jak i w  $G$ ).



(pivot = 6)

Powtarzaj dopóki  $i$  i  $r$  się nie przetną:

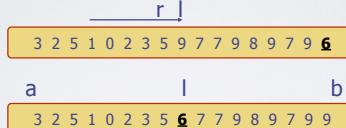
- Przeskanuj indeksy w prawo aż do odnalezienia elementu  $> x$ .
- Przeskanuj  $r$  w lewo aż do odnalezienia elementu  $< x$ .
- Jeśli  $i < r$ , zamień elementy na indeksach  $i$  i  $r$



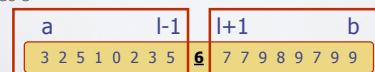
© 2004 Goodrich, Tamassia

# PODZIAŁ IN-SITU

Kiedy  $i$  i  $r$  miną się w taki sposób, że  $i > r$ , możemy zamienić pivot z elementem na pozycji  $|$



Wywołaj krok rekurencyjny dla podsekwiencji od indeksu  $a$  do  $i - 1$  i podsekwiencji dla indeksów  $i + 1$  do  $b$



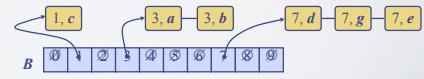
© 2004 Goodrich, Tamassia

## PODSUMOWANIE ALGORYTMÓW SORTOWANIA

Algorytm	Czas	Uwagi
przez wybór	$O(n^2)$	♦ in-situ ♦ wolne (dobre dla małych wejść)
przez wstawianie	$O(n^2)$	♦ in-situ ♦ wolne (dobre dla małych wejść)
szynkowe	$O(n \log n)$ oczekiwany $O(n^2)$ najgorszy	♦ in-situ, losowe ♦ najszybsza (dobra dla dużych danych)
przez kopcowanie	$O(n \log n)$	♦ in-situ ♦ szybka (dobra dla dużych danych)
przez scalanie	$O(n \log n)$	♦ sekwencyjny dostęp do danych ♦ szybka (dobra dla b. dużych danych)

© 2004 Goodrich, Tamassia

## SORTOWANIE KUBEŁKOWE I POZYCYJNE



## SORTOWANIE KUBEŁKOWE



Niech  $S$  będzie sekwencją wpisów (klucz, wartość) z kluczami w przedziale  $[0, N - 1]$

Sortowanie kubełkowe (Bucket-sort)

wykorzystuje klucze jako indeksy w zewnętrznej tablicy  $B$  sekwencji (kubelki)

**Faza 1:** Przenosimy wszystkie wpisy ( $k, o$ )  $S$  do kubla  $B[k]$

**Faza 2:** Dla  $i = 0, \dots, N - 1$ , przenieś wpisy kubla  $B[i]$  na koniec sekwencji  $S$

Analiza:

- Faza 1 zabiera  $O(n)$
- Faza 2 zabiera  $O(n + N)$

Sortowanie kubełkowe zabiera czas  $O(n + N)$

© 2004 Goodrich, Tamassia

**Algorytm *bucketSort*( $S, N$ )**  
**Wejście:** lista  $S$  z wpisami o kluczach w przedziale  $[0, N - 1]$   
**Wyjście:** sekwencja  $S$  posortowana rosnąco względem kluczy  
 $B \leftarrow$  tablica z  $N$  pustymi sekwencjami  
**for** każda pozycja  $p$  w  $S$  **do**  
     $e \leftarrow S.remove(p)$   
     $k \leftarrow e.getKey()$   
     $B[k].addLast(e)$   
**for**  $i \leftarrow 0$  to  $N - 1$  **do**  
    **for** każdy wpis  $e$  w  $B[i]$  **do**  
         $p \leftarrow B[i].first()$   
         $e \leftarrow B[i].remove(p)$   
         $S.addLast(e)$

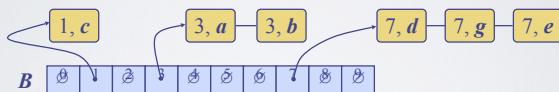
## PRZYKŁAD



Zakres kluczy  $[0, 9]$



Faza 1



Faza 2



© 2004 Goodrich, Tamassia

## WŁAŚCIWOŚCI I ROZSzerZENIA

Właściwości typu kluczy

- Klucze są wykorzystywane jako indeksy tablicy i nie mogą być dowolnymi obiektami
- Brak zewnętrznego komparatora
- Właściwość stałości sortowania
- Relatywne uporządkowanie dowolnych dwóch wpisów o tym samym kluczu jest zachowane po zakończeniu działania algorytmu

Rozszerzenia

- Klucze są integerami w przedziale  $[a, b]$ 
  - Umieść wpis  $(k, o)$  w kuble  $B[k - a]$
- Klucze z łańcuchami znakowymi ze zbioru  $D$  możliwych łańcuchów, gdzie  $D$  ma stały rozmiar (e.g., nazwy krajów członkowskich UE)
  - Posortuj  $D$  i wyznacz indeks  $r(k)$  dla każdego łańcucha  $k$  w  $D$  w posortowanej sekwencji
  - Umieść wpis  $(k, o)$  w kuble  $B[r(k)]$

© 2004 Goodrich, Tamassia

## UPORZĄDKOWANIE LEKSYKOGRAFICZNE



d-krotka jest sekwencją kluczy  $(k_1, k_2, \dots, k_d)$ , gdzie klucz  $k_i$  będzie  $i$ -tym wymiarem krotki

Przykład:

Współrzędne Kartezjańskie punktu w przestrzeni są 3-krotką

Uporządkowanie leksykograficzne dwóch d-krotek jest zdefiniowane rekurencyjnie w następujący sposób:

$$(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d)$$

↔

$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d)$$

Tzn., krotki są porównywane z przez pierwszy wymiar, potem przez drugi, itd.

© 2004 Goodrich, Tamassia

# SORTOWANIE LEKSYKOGRAFICZNE

Niech  $C$  będzie komparatorem, który porównuje dwie krotki  
Niech  $\text{stableSort}(S, C)$  będzie stałym algorytmem sortowania wykorzystującym komparator  $C$   
Sortowanie leksykograficzne sortuje sekwencję  $d$ -krotek w sposób leksykograficzny poprzez wywołanie  $d$  razy algorytmu  $\text{stableSort}$ . Raz dla każdego wymiaru.  
Sortowanie leksykograficzne działa w czasie  $O(dT(n))$ , gdzie  $T(n)$  jest czasem działania  $\text{stableSort}$

© 2004 Goodrich, Tamassia

**Algorytm *lexicographicSort(S)***  
**Wejście** sekwenja  $S$  z  $d$ -krotkami  
**Wyjście** sekwenja  $S$  posortowana leksykograficznie

```
for i ← d downto 1
    stableSort(S, Ci)
```

## Przykład:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)  
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)  
(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)  
(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

# SORTOWANIE POZYCYJNE

Sortowanie pozycyjne jest specyficzną odmianą sortowania leksykograficznego, które wykorzystuje sortowanie kubelkowe jako algorytm sortowania stałego. Sortowanie pozycyjne ma zastosowanie do krotek, których klucze w każdym wymiarze są integerami w przedziale  $[0, N - 1]$ . Sortowanie pozycyjne działa w czasie  $O(d(n + N))$

© 2004 Goodrich, Tamassia

**Algorytm *radixSort(S, N)***  
**Wejście** sekwenja  $S$  z  $d$ -krotkami takie, że  $(0, \dots, 0) \leq (x_1, \dots, x_d) \leq (N - 1, \dots, N - 1)$  dla każdej krotki  $(x_1, \dots, x_d)$  w  $S$   
**Wyjście** sekwenja  $S$  posortowana leksykograficznie

```
for i ← d downto 1
    bucketSort(S, N)
```

Tutaj 18.04

# PRZYKŁAD



Posortuj podane wartości z zastosowaniem sortowania pozycyjnego:  
 $\{21, 38, 241, 973, 100, 333\}$

© 2004 Goodrich, Tamassia

# SORTOWANIE POZYCYJNE DLA LICZB BINARNYCH



Rozważ sekwencję  $n$ -bitowych integerów  $x = x_{b-1} \dots x_0$   
Reprezentujemy każdy element jako  $b$ -krotkę integerów z przedziału  $[0, 1]$  i zastosujemy sortowanie pozycyjne z  $N = 2$ . Ta modyfikacja algorytmu pozycyjnego działa w czasie  $O(bn)$ . Dla przykładu, możemy posortować sekwencję 32-bitowych integerów w liniowym czasie

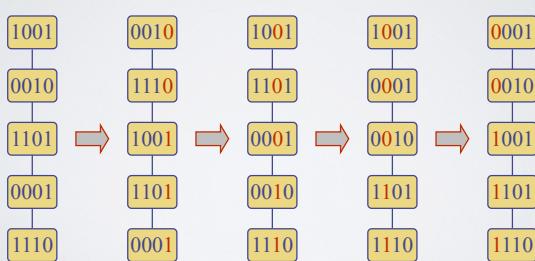
**Algorytm *binaryRadixSort(S)***  
**Wejście** sekwenja  $S$  z  $b$ -bitowymi integerami  
**Wyjście** posortowana sekwenja  $S$   
zamień każdy element  $x$  z  $S$  na elementem  $(0, x)$   
**for**  $i \leftarrow 0$  **to**  $b - 1$   
    zamień klucz  $k$  każdego elementu  $(k, x)$  w  $S$   
    z bitem  $x_i$   
**bucketSort(S, 2)**

© 2004 Goodrich, Tamassia

# PRZYKŁAD



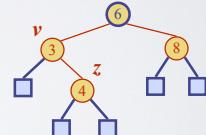
Sortowanie sekwenacji 4-bitowych integerów



© 2004 Goodrich, Tamassia

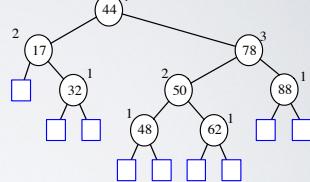
# DRZEWA AVL

Adel'son-Vel'skii & Landis



## DEFINICJA DRZEW AVL

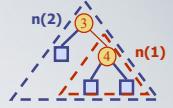
- Drzewa AVL są zbalansowane
- Drzewo AVL jest **binarnym drzewem przeszukiwań**, w którym dla każdego węzła wewnętrznego v wysokość jego potomków może się różnić najwyżej o 1.



Przykład drzewa AVL z zaznaczonymi wysokościami

© 2004 Goodrich, Tamassia

## WYSOKOŚĆ DRZEWY AVL



**Fakt:** Wysokość drzewa AVL przechowującego n kluczy wynosi  $O(\log n)$

**Dowód:** Wprowadźmy ograniczenie **n(h)**: najmniejsza ilość węzłówewnętrznych drzewa AVL o wysokości h.

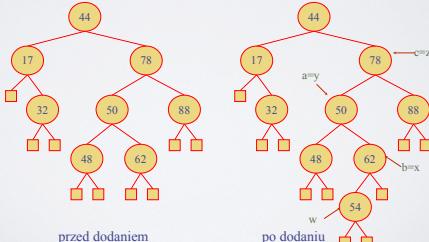
- Widzimy zatem, że  $n(1) = 1$  i  $n(2) = 2$
- Dla  $n > 2$ , drzewo AVL o wysokości h zawiera korzeń, jedno poddrzewo AVL o wys.  $h-1$  i drugie poddrzewo o wys.  $h-2$
- zatem  $n(h) = 1 + n(h-1) + n(h-2)$
- Wiedząc, że  $n(h-1) > n(h-2)$ , mamy  $n(h) > 2n(h-2)$ . Zatem  
•  $n(h) > 2n(h-2)$ ,  $n(h) > 4n(h-4)$ ,  $n(h) > 8n(h-6)$ , ... (przez indukcję)  
•  $n(h) > 2^{h-2}$
- rozwiązuje krok podstawowy ( $h-2i = 2$ ) otrzymamy  $n(h) > 2^{h/2-1}$
- nakładając logarytm  $h < 2\log n(h) + 2$
- Zatem wysokość drzewa AVL jest w  $O(\log n)$

© 2004 Goodrich, Tamassia

## DODAWANIE ELEMENTU DO DRZEWY AVL

- Dodawanie elementu jest takie samo jak w binarnym drzewie przeszukiwań
- Zawsze wykonywane poprzez rozszerzenie węzła zewnętrznego

Przykład:



przed dodaniem

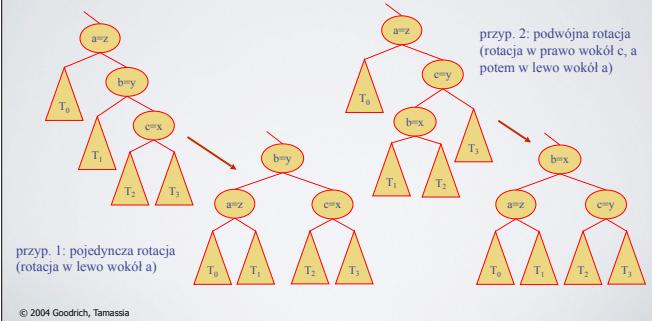
po dodaniu w

po dodaniu

© 2004 Goodrich, Tamassia

## ROTACJA WĘZŁÓW AVL

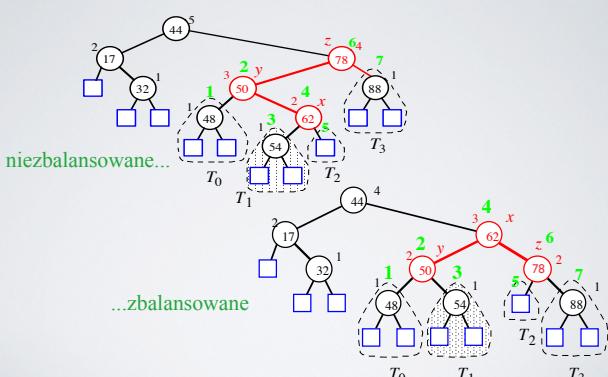
- niech  $(a, b, c)$  będzie listą ścieżki inorder dla x, y, z
- wykonaj rotacje niezbędne do umieszczenia b na górze drzewa



przyk. 1: pojedyncza rotacja (rotacja w lewo wokół a)

przyk. 2: podwójna rotacja (rotacja w prawo wokół a, a potem w lewo wokół a)

## DODAWANIE - C.D.

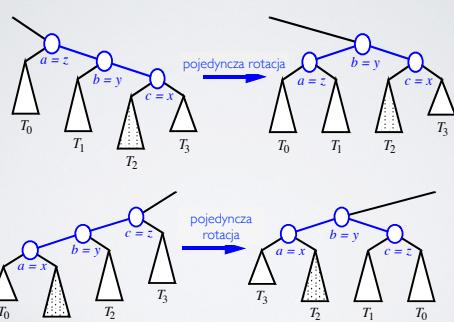


niebalansowane...

...zbalansowane

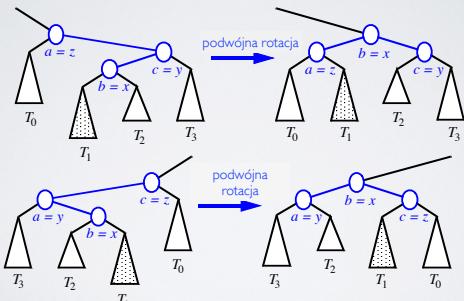
© 2004 Goodrich, Tamassia

## RESTRUKTURYZACJA - JAKO POJEDYNCZA ROTACJA



© 2004 Goodrich, Tamassia

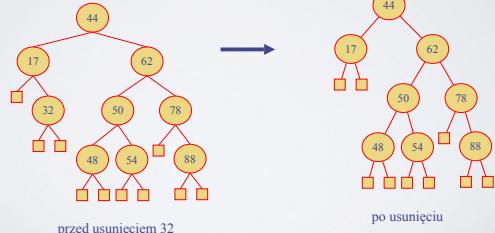
## RESTRUKTURYZACJA - JAKO PODWÓJNA ROTACJA



© 2004 Goodrich, Tamassia

## USUWANIE Z DRZEWĄ AVL

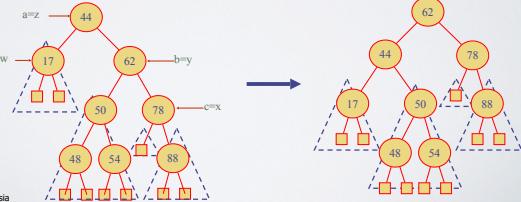
- Usuwanie rozpoczyna się w taki sam sposób jak w binarnym drzewie przeszukiwań
- usunięty węzeł będzie pustym zewnętrznym węzłem
- jego ojciec, w, może spowodować brak zbalansowania
- Przykład:



© 2004 Goodrich, Tamassia

## BALANSOWANIE DRZEWĄ PO USUNIĘCIU

- Niech z będzie **pierwszym niebalansowanym** węzłem napotkanym przy travescerowaniu drzewa w góre począwszy od węzła w. Jednocześnie niech y będzie synem z o większej wysokości i niech x będzie synem y o większej wysokości
- wywołujemy **restructure(x)** do przywrócenia balansu w z.
- Ponieważ restrukturyzacja może naruszyć balans w węzłach na wyższych poziomach drzewa musimy kontynuować sprawdzanie aż do osiągnięcia korzenia.



© 2004 Goodrich, Tamassia

## ZŁOŻONOŚĆ OBLCZENIOWA DRZEW AVL

- pojedyncza zmiana -  $O(1)$ 
  - stosując strukturę listy dla drzewa binarnego
- szukanie -  $O(\log n)$ 
  - wysokość drzewa -  $O(\log n)$ , brak restrukturyzacji
- wstawianie -  $O(\log n)$ 
  - wstępne szukanie -  $O(\log n)$
  - Restrukturyzacja dla zachowania wysokości -  $O(\log n)$
- usuwanie -  $O(\log n)$ 
  - wstępne szukanie -  $O(\log n)$
  - Restrukturyzacja dla zachowania wysokości -  $O(\log n)$

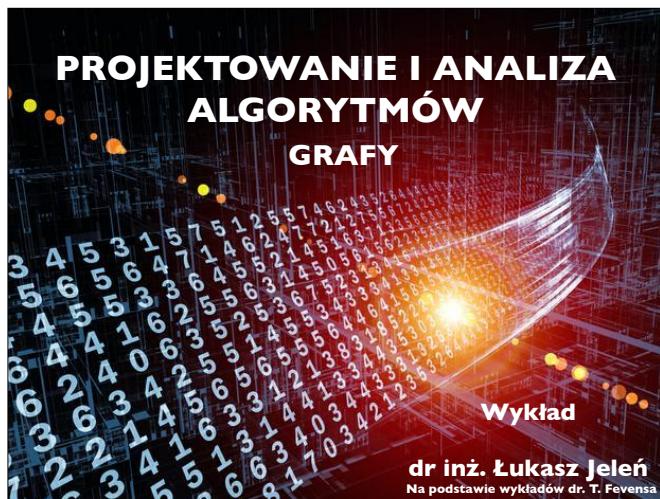


© 2004 Goodrich, Tamassia

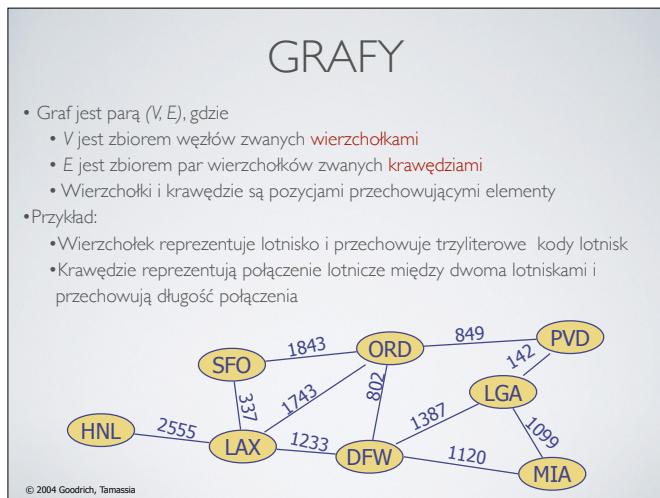
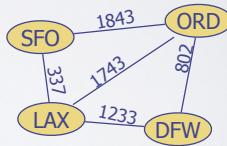
## PORÓWNANIE IMPLEMENTACJI SŁOWNIKÓW

Metoda	Tablica Haszująca (później)	Tablica przeszukiwań (Lista uporządkowana)	BST $\log n \leq h \leq n$	AVL
<code>size, isEmpty</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>entries</code>	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<code>find</code>	$O(1) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(\log n) \text{ oczekiwany}, O(\log n) \text{ przyp. najgorszy}$	$O(h)$	$O(\log n)$
<code>findAll</code>	$O(1 + s) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(\log n + s) \text{ oczekiwany}, O(\log n + s) \text{ przyp. najgorszy}$	$O(h + s)$	$O(\log n + s)$
<code>insert</code>	$O(1) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(n) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(h)$	$O(\log n)$
<code>remove</code>	$O(1) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(n) \text{ oczekiwany}, O(n) \text{ przyp. najgorszy}$	$O(h)$	$O(\log n)$

© 2004 Goodrich, Tamassia

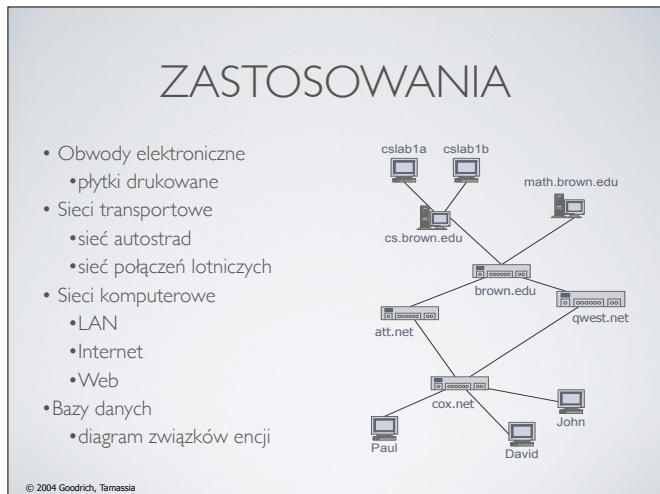


## GRAFY



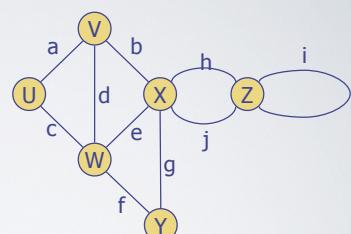
## TYPY KRAWĘDZI

- Krawędź skierowana
  - skierowana para wierzchołków  $(u, v)$
  - pierwszy wierzchołek  $u$  jest początkiem
  - drugi wierzchołek  $v$  jest celem/końcem
  - np.: lot
- Krawędź nieskierowana
  - nieskierowana para wierzchołków  $(u, v)$
  - np.: połączenie lotnicze
- Graf skierowany
  - wszystkie krawędzie są skierowane
  - np.: mapa połączeń
- Graf nieskierowany
  - wszystkie krawędzie są nieskierowane



## TERMINOLOGIA

- Wierzchołki końcowe krawędzi
    - $U$  i  $V$  są końcami krawędzi  $a$
  - Krawędzie incydentne do wierzchołków
    - $a, b$  są incydentne do  $V$
  - Wierzchołki sąsiednie
    - $U$  i  $V$  są sąsiednie
  - Stopień wierzchołka
    - $X$  ma stopień 5
  - Krawędzie równoległe
    - $h$  i  $j$  są równoległe
  - Pętla
    - $i$  jest pętlą
- Gęstość grafu:  
• Stosunek liczby krawędzi do max. liczby krawędzi
- $$\frac{2|E|}{|V|(|V| - 1)}$$

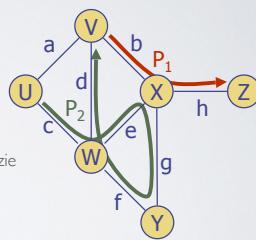


© 2004 Goodrich, Tamassia

## TERMINOLOGIA 2

### Ścieżka

- sekwencja kolejnych wierzchołków i krawędzi
- rozpoczyna się od wierzchołka
- kończy się na wierzchołku
- każda krawędź poprzedza i następuje jej wierzchołek końcowy



### Prosta ścieżka

- ścieżka, w której wszystkie wierzchołki i krawędzie różnią się od siebie

### Przykłady:

- $P_1 = (V, b, X, h, Z)$  jest prostą ścieżką
- $P_2 = (U, c, W, e, X, g, Y, f, V, d, V)$  nie jest ścieżką prostą

© 2004 Goodrich, Tamassia

## TERMINOLOGIA 3

### Cykł

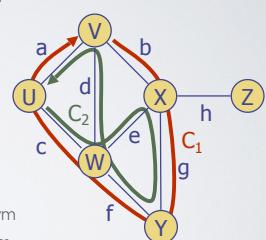
- okrężna sekwencja kolejnych wierzchołków i krawędzi
- każdą krawędź poprzedza i następuje jej wierzchołek końcowy

### Cykł prosty

- cykl, w którym wszystkie wierzchołki i krawędzie różnią się od siebie

### Przykłady:

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, d)$  jest cyklem prostym
- $C_2 = (U, c, W, e, X, g, Y, f, V, d, V, a, d)$  nie jest cyklem prostym



© 2004 Goodrich, Tamassia

## WŁAŚCIWOŚCI

### Właściwość 1

$$\sum \deg(v) = 2m$$

Dowód: każda nieskierowana krawędź jest liczona dwa razy

### Właściwość 2

W grafie nieskierowanym bez pętli i wielokrotnych krawędzi

$$m \leq n(n-1)/2$$

Dowód: każdy wierzchołek ma stopień najwyżej  $(n-1)$

### Notacja

n	ilość wierzchołków
m	ilość krawędzi
$\deg(v)$	stopień wierzchołka v



### Przykład

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

© 2004 Goodrich, Tamassia

## GŁÓWNE METODY GRAFÓW ADT

### Wierzchołki i krawędzie

- są pozycjami
- przechowują elementy

### Metody dostępu:

- `endVertices(e)`: tablica dwóch końcowych wierzchołków e
- `opposite(v, e)`: przeciwny wierzchołek do v względem e
- `areAdjacent(v, w)`: prawda iff v i w sąsiadnie
- `replace(v, x)`: zastąp element w wierzchołku v na x
- `replace(e, x)`: zastąp element na krawędzi e na x

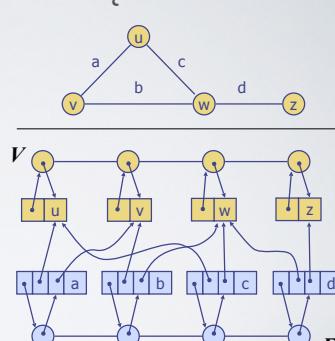
### Metody uaktualniające

- `insertVertex(o)`: dodaj wierzchołek przechowujący element o
- `insertEdge(v, w, o)`: dodaj krawędź (v, w) przechowującą element o
- `removeVertex(v)`: usuń wierzchołek v (oraz przylegające krawędzie)
- `removeEdge(e)`: usuń krawędź e
- Metody iterujące
- `incidentEdges(v)`: krawędzie przylegające do v
- `vertices()`: wszystkie wierzchołki w grafie
- `edges()`: wszystkie krawędzie w grafie

© 2004 Goodrich, Tamassia

## LISTA KRAWĘDZI

- Obiekt - wierzchołek
  - element
  - referencja do pozycji w liście wierzchołków
- Obiekt - krawędź
  - element
  - obiekt - wierzchołek początkowy
  - obiekt - wierzchołek końcowy
  - referencja do pozycji w liście krawędzi
- Lista wierzchołków
  - sekwencja obiektów wierzchołka
- Lista krawędzi
  - sekwencja obiektów krawędzi



© 2004 Goodrich, Tamassia

## LISTA SĄSIEDZTWA

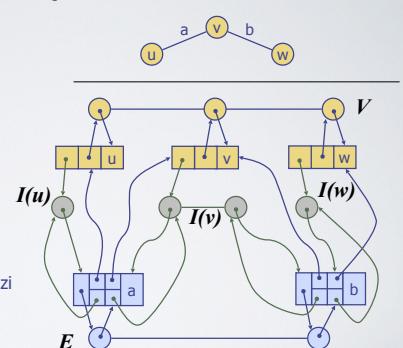
### Struktura listy krawędzi

### Lista incydencji dla każdego wierzchołka, $I(v)$

- sekwencja referencji do obiektów krawędzi incydentnych

### Rozszerzone obiekty krawędzi

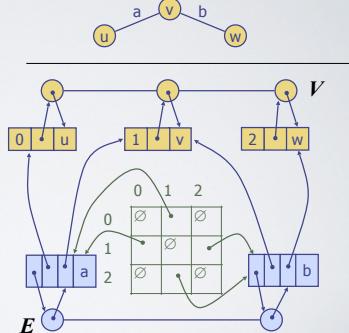
- referencje do listy sąsiedztwa wierzchołków końcowych



© 2004 Goodrich, Tamassia

## MACIERZ SĄSIEDZTWA

- Struktura listy krawędzi



© 2004 Goodrich, Tamassia

- Rozszerzony obiekt wierzchołka
  - klucz - integer key (indeks) - powiązany z wierzchołkiem
- Tablica 2D sąsiedztwa
  - Referencja do obiektu krawędzi dla sąsiednich wierzchołków
  - Nul dla wierzchołków niesąsiadujących

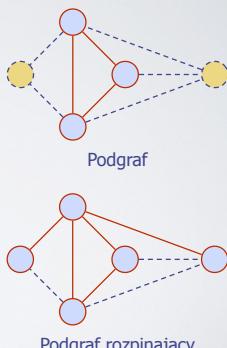
## ZŁOŻONOŚĆ ASYMPTOTYCZNA

	Lista krawędzi	Lista sąsiedztwa	Macierz sąsiedztwa
Miejsce	$n + m$	$n + m$	$n^2$
incidentEdges( $v$ )	$m$	$\deg(v)$	$n$
areAdjacent ( $v, w$ )	$m$	$\min(\deg(v), \deg(w))$	1
insertVertex( $o$ )	1	1	$n^2$
insertEdge( $v, w, o$ )	1	1	1
removeVertex( $v$ )	$m$	$\deg(v)$	$n^2$
removeEdge( $e$ )	1	1	1

© 2004 Goodrich, Tamassia

## PODGRAFY

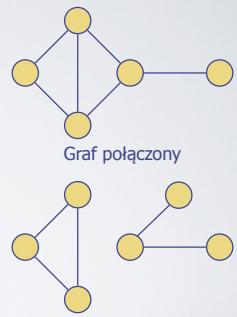
- Podgraf  $S$  grafu  $G$  jest takim grafem, że
  - wierzchołki  $S$  są podzbiorem wierzchołków  $G$
- Podgraf rozpinający grafu  $G$  jest podgrafem, który zawiera wszystkie wierzchołki  $G$



© 2004 Goodrich, Tamassia

## ŁĄCZNOŚĆ

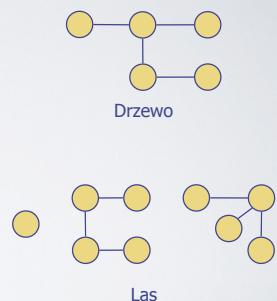
- Graf jest połączony (spójny) jeśli istnieje ścieżka między każdą parą wierzchołków.
- Elementem połączonym grafu  $G$  jest maksymalny podgraf połączony grafu  $G$



© 2004 Goodrich, Tamassia

## DRZEWA I LASY

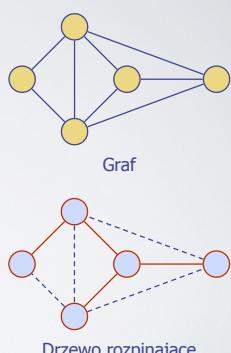
- Drzewo jest grafem nieskierowanym takim, że
  - $T$  jest połączone
  - $T$  nie zawiera cykli
  - Jest to inna definicja drzewa niż w przypadku drzewa ukorzenionego
- Lasem jest graf nieskierowany bez cykli
- Komponenty połączone lasu są drzewami



© 2004 Goodrich, Tamassia

## DRZEWA I LASY ROZPINAJĄCE

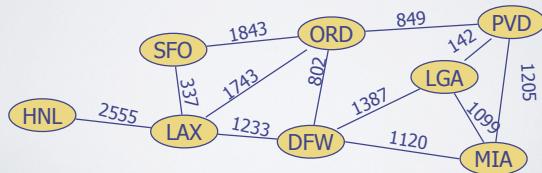
- Drzewo rozpinające grafu połączonego jest podgrafem połączonym, które jest drzewem
- Drzewo rozpinające nie jest unikalne dopóki graf nie jest drzewem
- Drzewa rozpinające mają zastosowanie w sieciach komunikacyjnych
- Las rozpinający grafu jest podgrafem rozpinającym, który jest lasem



© 2004 Goodrich, Tamassia

## GRAF WAŻONY

- W grafie ważonym każda krawędź ma przypisaną wartość liczbową, tzw. wagę krawędzi
- Wagi krawędzi mogą reprezentować odległość, koszt, czas, itp.
- Przykład:
  - W grafie tras lotniczych waga krawędzi reprezentuje odległość między lotniskami końcowymi wyrażoną w milach



© 2004 Goodrich, Tamassia

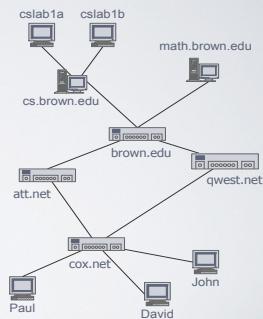
## PROBLEM

- Załóżmy, że chcemy połączyć wszystkie komputery w nowo tworzonym laboratorium/biurze
  - minimalna ilość kabla - koszty

### Rozwiązywanie:

- Model grafu ważonego ( $G$ )
  - wierzchołki - komputery
  - krawędzie - wszystkie możliwe pary  $(u, v)$  komputerów
  - $\deg(u, v) = w(u, v)$  - odpowiada długości kabla potrzebnego do połączenia komputera  $u$  z komputerem  $v$
- Moglibyśmy wyznaczyć najkrótszą drogę od wierzchołka  $v$ 
  - nieoptymalne
- Znajdziemy drzewo  $T$ , które zawiera wszystkie wierzchołki  $G$  i posiada najmniejsze łączne wagi ze wszystkich drzew rozpinających.

© 2004 Goodrich, Tamassia



## MINIMALNE DRZEWKA ROZPINAJĄCE

- Mając dany nieskierowany graf  $G$ , chcemy znaleźć drzewo  $T$ , które zawiera wszystkie wierzchołki i minimalizuje sumę:

$$w(T) = \sum_{((v,u) \in T)} w((v,u))$$

- Problem wyznaczania drzewa rozpinającego o najmniejszej wadze nazywa się problemem minimalnego drzewa rozpinającego (MST).

© 2004 Goodrich, Tamassia

## ALGORYTM KRUSKALA

- Buduje minimalne drzewo rozpinające z zastosowaniem klastrów
  - grupowania węzłów
- Początkowo wszystkie węzły stanowią osobne klastry
- Krawędzie przechowywane w kolejce priorytetowej
  - wagi są kluczami
- Dla wszystkich krawędzi:
  - $Q.removeMin();$
  - Jeśli  $u$  i  $v$  nie należą do tego samego klastra to dodajemy  $(v, u)$  do  $T$
  - Łączymy klastry zawierające  $u$  i  $v$  w jeden

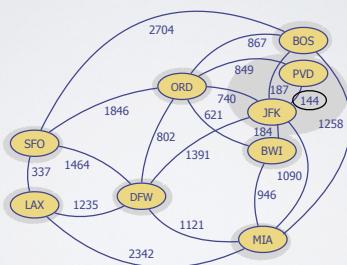
```

Algorytm Kruskala( $G$ )
Wejście: Graf ważony  $G$  z  $n$  wierzchołkami i  $m$  krawędziami
Wyjście: Minimalne drzewo rozpinające
T dla grafu G
for każdy wierzchołek  $v$  w  $G$ 
   $C(v) \leftarrow \{v\}$ 
   $Q \leftarrow \{E\}$  // E - lista krawędzi
   $T \leftarrow \emptyset$ 
while  $T.size() < n-1$  do
   $(u, v) \leftarrow Q.removeMin()$ 
  if  $C(v) \neq C(u)$ 
     $T.Add(v, u)$ 
    Merge( $C(v)$ ,  $C(u)$ )
return  $T$ 

```

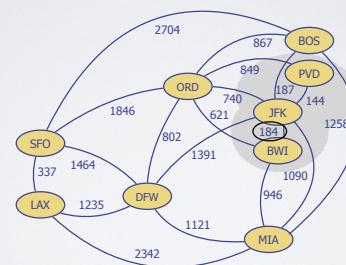
© 2004 Goodrich, Tamassia

## PRZYKŁAD



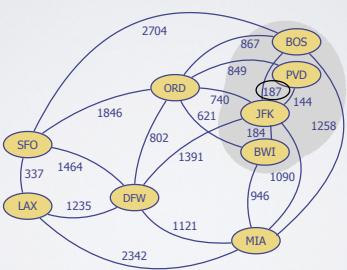
© 2004 Goodrich, Tamassia

## PRZYKŁAD



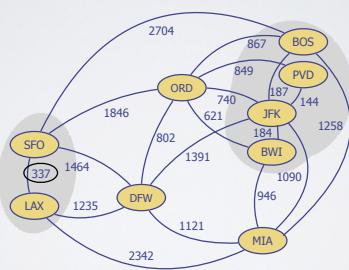
© 2004 Goodrich, Tamassia

## PRZYKŁAD



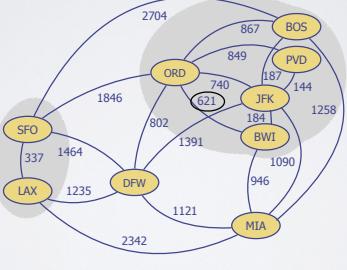
© 2004 Goodrich, Tamassia

## PRZYKŁAD



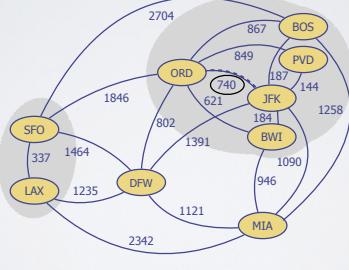
© 2004 Goodrich, Tamassia

## PRZYKŁAD



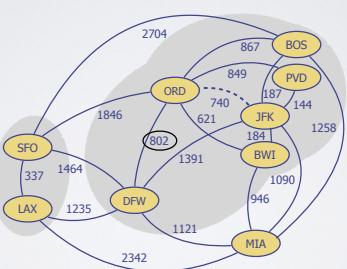
© 2004 Goodrich, Tamassia

## PRZYKŁAD



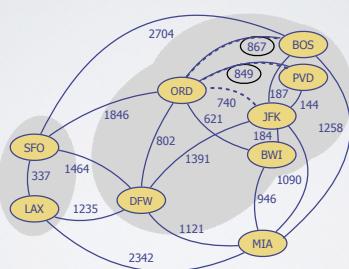
© 2004 Goodrich, Tamassia

## PRZYKŁAD



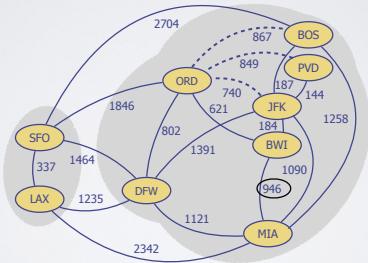
© 2004 Goodrich, Tamassia

## PRZYKŁAD



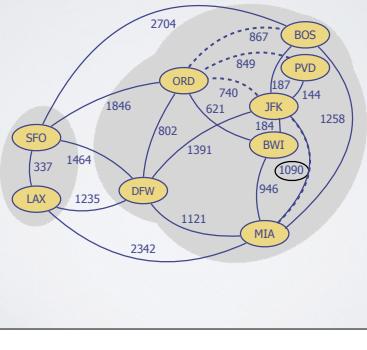
© 2004 Goodrich, Tamassia

## PRZYKŁAD



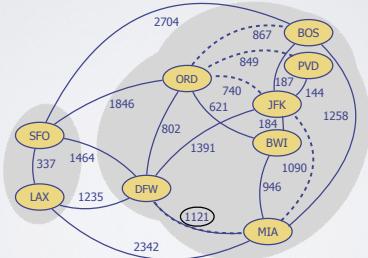
© 2004 Goodrich, Tamassia

## PRZYKŁAD



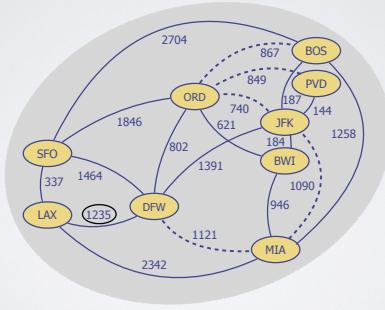
© 2004 Goodrich, Tamassia

## PRZYKŁAD



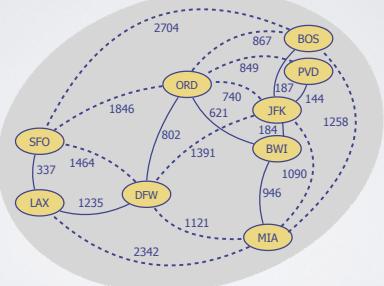
© 2004 Goodrich, Tamassia

## PRZYKŁAD



© 2004 Goodrich, Tamassia

## PRZYKŁAD



© 2004 Goodrich, Tamassia

## ZŁOŻONOŚĆ ALGORYTMU KRUSKALA

- Możemy zaimplementować kolejkę priorytetową za pomocą kopca
- $O(m \log n)$  - sukcesywne wstawianie do kolejki
- $O(m)$  - techniką bottom-up
- Usuwanie:
  - $O(\log m) \Rightarrow O(\log n)$  (graf jest prosty)
  - Zatem czas to  $O(m \log n)$
- Całkowita złożoność algorytmu Kruskala:
  - Czas pętli **while**:
  - $\sum_{(l \in G)} (l + \deg(v)) \log n$
- co daje czas  $O((n + m) \log n)$
- po uproszczeniu  **$O(m \log n)$**

© 2004 Goodrich, Tamassia

## ALGORYTM PRIMA - JARNIKA

- Buduje minimalne drzewo rozpinające z zastosowaniem klastrów
  - rozpoczyna od dowolnego wierzchołka v
  - korzenia
- Idea zbliżona do algorytmu Dijkstry (później)
- Dodajemy v do drzewa T, a krawędzie incydentne do v umieszczaamy w kolejce priorytetowej
  - wagi są kluczami
- Q.removeMin():
  - jeśli wierzchołek z  $\not\in$  MST to dodajemy z do MST
  - dodajemy do Q krawędzie incydentne do z
  - jeśli T zawiera wszystkie wierzchołki grafu to koniec

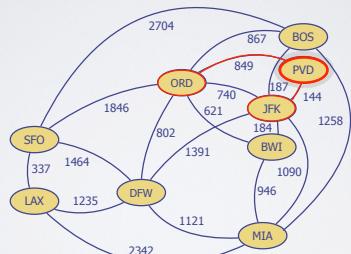
### Algorytm PrimJarnik( $G$ )

Wejście: Graf ważony  $G$  z n wierzchołkami i m krawędziami  
Wyjście: Minimalne drzewo rozpinające T dla grafu G

```
v ← losowy wierzchołek z G
D(v) ← 0 // bieżące MST
for każdy wierzchołek u ≠ v
  D(u) ← +∞
  Q ← {E} // E - lista krawędzi
  T ← ∅
while Q.isEmpty() = false do
  (u, e) ← Q.removeMin()
  T.Add(u, e)
  for każdy wierzchołek z ∈ Q sąsiedni do u
    if w(u, z) < D(z)
      D(z) ← w(u, z)
      Wstaw (u, z) do Q z kluczem D(z)
return T
```

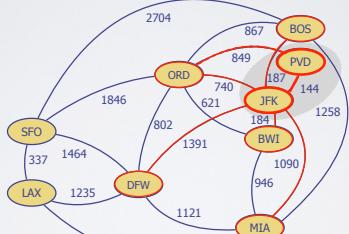
© 2004 Goodrich, Tamassia

## PRZYKŁAD



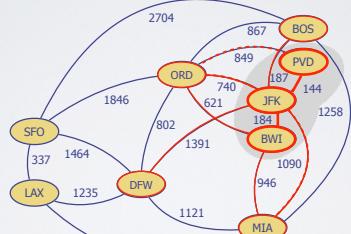
© 2004 Goodrich, Tamassia

## PRZYKŁAD



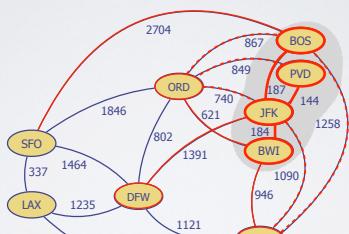
© 2004 Goodrich, Tamassia

## PRZYKŁAD



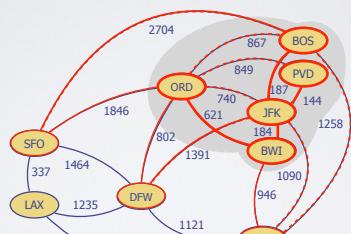
© 2004 Goodrich, Tamassia

## PRZYKŁAD



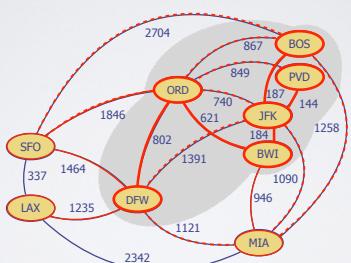
© 2004 Goodrich, Tamassia

## PRZYKŁAD



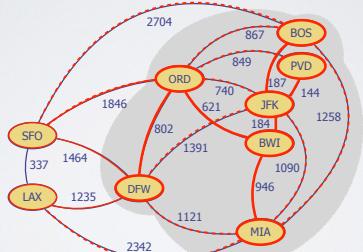
© 2004 Goodrich, Tamassia

## PRZYKŁAD



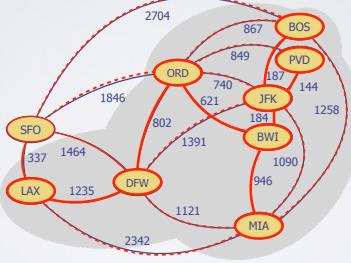
© 2004 Goodrich, Tamassia

## PRZYKŁAD



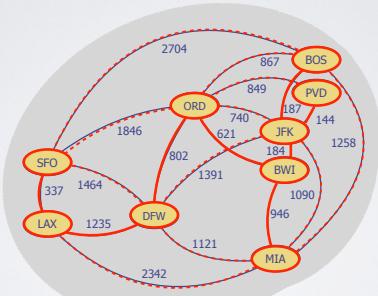
© 2004 Goodrich, Tamassia

## PRZYKŁAD



© 2004 Goodrich, Tamassia

## PRZYKŁAD



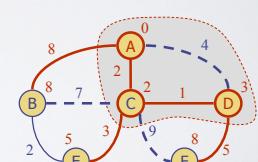
© 2004 Goodrich, Tamassia

## ZŁOŻONOŚĆ

- Możemy zaimplementować kolejkę priorytetową za pomocą kopca
- Usuwanie:
  - $O(\log n)$
  - Zatem czas to  $O(m \log n)$  (pętla **while**)
- Uaktualnianie:
  - max.  $O(\log n)$  dla każdej krawędzi  $(u, z)$
  - Pozostałe operacje wykonywane w czasie  $O(1)$
- Zatem całkowity czas to  $O((n + m) \log n)$
- po uproszczeniu  **$O(m \log n)$**

© 2004 Goodrich, Tamassia

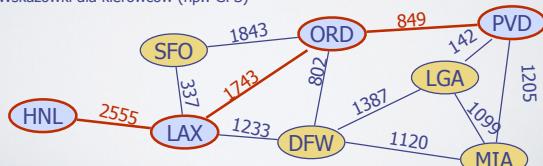
## NAJKRÓTSZA ŚCIEŻKA



## NAJKRÓTSZA ŚCIEŻKA

- Mając dany graf ważony i dwa wierzchołki  $u$  i  $v$  chcemy wyznaczyć ścieżkę między nimi o najmniejszej całkowitej wadze.

- Długość ścieżki jest sumą wag jej krawędzi.
- Przykład:**
  - Najkrótsza ścieżka między Providence i Honolulu
- Zastosowania**
  - Przekierowywanie pakietów Internetowych
  - Rezerwacje lotów
  - Wskazówki dla kierowców (np.: GPS)



## WŁAŚCIWOŚCI

### Właściwość 1:

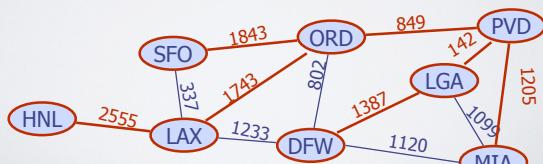
Podścieżka najkrótszej ścieżki jest najkrótszą ścieżką samą w sobie

### Właściwość 2:

Istnieje drzewo najkrótszych ścieżek poczawszym od węzła startowego do wszystkich pozostałych węzłów

### Przykład:

Drzewo najkrótszych ścieżek z Providence



## ALGORYTM DIJKSTRY

- Odległość wierzchołka  $v$  od wierzchołka  $s$  jest długością najkrótszej ścieżki między  $s$  i  $v$
- Algorytm Dijkstry wyznacza odległości wszystkich wierzchołków poczawszym od wierzchołka startowego  $s$
- Założenia:**
  - graf jest spójny/połączony
  - krawędzie są nieskierowane
  - wagi krawędzi są nieujemne

- Podobnie jak w przypadku algorytmu Prima będziemy tworzyć "chmurę" ze wszystkich wierzchołków (MST) poczawszym od  $s$
- Przechowujemy każdy wierzchołek  $v$ , etykietę  $d(v)$  reprezentującą odległość  $v$  od  $s$  w podgrafie zawierającym MST wraz z sąsiednimi wierzchołkami
- Przy każdej iteracji
  - Dodajemy wierzchołek  $u$  nieznajdujący się w MST o najmniejszej odległości,  $d(u)$
  - Uaktualniamy etykiety wierzchołków sąsiednich do  $u$

© 2004 Goodrich, Tamassia

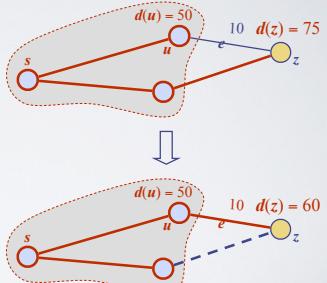
## RELAKSACJA WIERZCHOŁKÓW

- Weźmy krawędź  $e = (u,z)$  taką, że

- $u$  jest wierzchołkiem ostatnio dodanym
- $z$  jest poza MST/chmurą

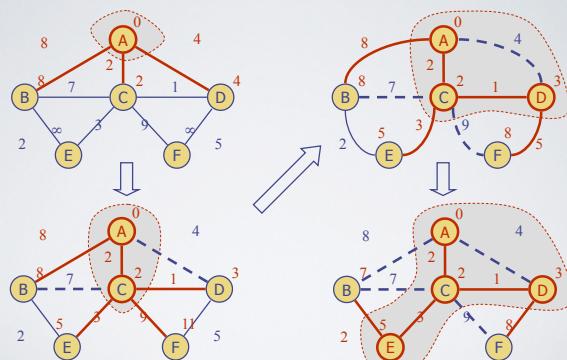
- Relaksacja krawędzi  $e$  aktualizuje odległość  $d(z)$  następująco:

$$d(z) \leftarrow \min\{d(z), d(u) + \text{waga}(e)\}$$

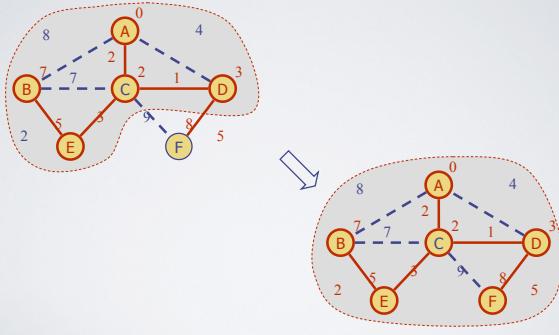


© 2004 Goodrich, Tamassia

## PRZYKŁAD



## PRZYKŁAD



# ALGORYTM DIJKSTRA

- Kolejka priorytetowa przechowuje wierzchołki nieznajdujące się w bieżącym MST
  - Klucz: odległość
  - Element: wierzchołek
  - *insert(k,e)* zwraca lokalizację
  - *replaceKey(l,k)* zamienia klucz dla danego elementu
- Przechowujemy dwie etykiety dla każdego wierzchołka:
  - Odległość ( $d(v)$ )
  - lokalizacje w kolejce priorytetowej

```
Algorithm DijkstraDistances(G, s)
    Q ← nowa kolejka priorytetowa bazująca
        na kopcu
    for all  $v \in G.vertices()$ 
        if  $v = s$ 
            setDistance( $v, 0$ )
        else
            setDistance( $v, \infty$ )
             $l \leftarrow Q.insert(getDistance(v), v)$ 
            setLocator( $v,l$ )
    while  $\sim Q.isEmpty()$ 
         $u \leftarrow Q.removeMin()$ 
        for all  $e \in G.incidentEdges(u)$ 
            {relaksacja krawędzi  $e$ }
             $z \leftarrow G.opposite(u,e)$ 
             $r \leftarrow getDistance(u) + weight(e)$ 
            if  $r < getDistance(z)$ 
                setDistance( $z,r$ )
                Q.replaceKey(getLocator( $z$ ), $r$ )
```

© 2004 Goodrich, Tamassia

# ZŁOŻONOŚĆ

- Operacje grafowe
  - Metoda *incidentEdges* jest wywoływana raz dla każdego wierzchołka
- Operacje etykietowania
  - Ustawiamy/pobieramy odległość i lokalizację wierzchołka z  $O(\deg(z))$  razy
  - Ustawianie/pobieranie etykiety zajmuje czas  $O(1)$
- Operacje na kolejce priorytetowej
  - Każdy wierzchołek jest umieszczany i usuwany z kolejki tylko raz. Każda taka operacja zajmuje czas  $O(\log n)$
  - Klucz wierzchołka jest modyfikowany najwyżej  $\deg(w)$  razy. Każda zmiana klucza zajmuje czas  $O(\log n)$
- Algorytm Dijkstra działa w czasie  $O((n + m) \log n)$  pod warunkiem, że graf jest zaimplementowany za pomocą listy sąsiedztwa
- Po uproszczeniu  $O(m \log n)$  - graf jest spójny

© 2004 Goodrich, Tamassia

# PROJEKTOWANIE I ANALIZA ALGORYTMÓW

## GRAFY CZ.2, SZTUCZNA INTELIGENCJA

**Wykład**

dr inż. Łukasz Jeleń

Część slajdów pochodzi z wykładow prof. Christiana Jacoba z Uniwersytetu w Calgary oraz wykładow prof. Włodzisława Ducha z UMK i dr. inż. D. Banasiaka, PWr

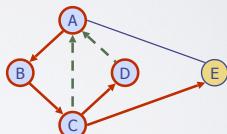
## ALGORYTM DIJKSTRA

- Kolejka priorytetowa przechowuje wierzchołki nieznajdujące się w bieżącym MST
  - Klucz: odległość
  - Element: wierzchołek
  - *insert(k,e)* zwraca lokalizację
  - *replaceKey(l,k)* zamienia klucz dla danego elementu
- Przechowujemy dwie etykiety dla każdego wierzchołka:
  - Odległość ( $d(v)$ )
  - lokalizacje w kolejce priorytetowej

```
Algorithm DijkstraDistances( $G, s$ )
 $Q \leftarrow$  nowa kolejka priorytetowa bazująca na kopcu
for all  $v \in G.vertices()$ 
  if  $v = s$ 
    setDistance( $v, 0$ )
  else
    setDistance( $v, \infty$ )
     $l \leftarrow Q.insert(getDistance(v), v)$ 
    setLocator( $v, l$ )
while  $\sim Q.isEmpty()$ 
   $u \leftarrow Q.removeMin()$ 
  for all  $e \in G.incidentEdges(u)$ 
    {relaksacja krawędzi  $e$ }
     $z \leftarrow G.opposite(u, e)$ 
     $r \leftarrow getDistance(u) + weight(e)$ 
    if  $r < getDistance(z)$ 
      setDistance( $z, r$ )
       $Q.replaceKey(getLocator(z), r)$ 
```

© 2004 Goodrich, Tamassia

## PRZESZUKIWANIE W GŁĘB



## PRZESZUKIWANIE W GŁĘB (DFS)

- DFS jest techniką travwersowania grafów
- DFS
  - Odwiedza wszystkie wierzchołki i krawędzie grafu  $G$
  - Sprawdza czy  $G$  jest spójny
  - Wyznacza komponenty połączone grafu
  - Wyznacza las rozpinający grafu
- Czas DFS dla grafu z  $n$  wierzchołkami i  $m$  krawędziami to  $O(n + m)$
- DFS może być rozszerzony do rozwiązywania innych problemów związanych z grafami
  - Znajdywanie ścieżki między dwoma zadanymi wierzchołkami
  - Znajdywanie cykli w grafie
- Przeszukiwanie w głąb dla grafu jest równoważne ze ścieżką Eulera dla drzew binarnych

© 2004 Goodrich, Tamassia

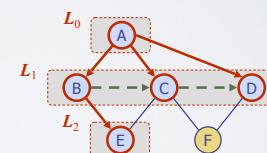
## SZUKANIE ŚCIEŻKI

- Możemy zmodyfikować algorytm DFS tak, aby znajdował ścieżkę między dwoma wierzchołkami  $u$  i  $z$  z zastosowaniem metody szablonowej
- Wykorzystujemy stos  $S$  do śledzenia ścieżki między wierzchołkiem startowym, a bieżącym
- W momencie osiągnięcia wierzchołka końcowego  $z$  zwracamy ścieżkę jako zawartość stosu

```
Algorytm pathDFS( $G, v, z$ )
setLabel( $v, ODWIEDZONY$ )
S.push( $v$ )
if  $v = z$ 
  return S.elements()
for all  $e \in G.incidentEdges(v)$ 
  if getLabel( $e$ ) = NIEZNALĘZIONE
     $w \leftarrow opposite(v, e)$ 
    if getLabel( $w$ ) = NIEZNALĘZIONE
      setLabel( $w, ZNALEZIONE$ )
      S.push( $e$ )
      pathDFS( $G, w, z$ )
      S.pop( $e$ )
    else
      setLabel( $e, POWROTNY$ )
      S.pop( $v$ )
```

© 2004 Goodrich, Tamassia

## PRZESZUKIWANIE WSZERZ

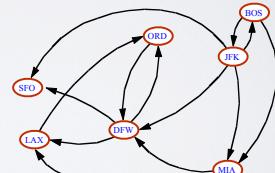


## PRZESZUKIWANIE WSZERZ (BFS)

- BFS jest również techniką traversowania grafu
- BFS
  - Odwiedza wszystkie wierzchołki i krawędzie
  - Sprawdza czy G jest spójny
  - Wyznacza komponenty połączone grafu
  - Wyznacza las rozpinający grafu
- Czas BFS dla grafu z  $n$  wierzchołkami i  $m$  krawędziami to  $O(n + m)$
- BFS może być rozszerzony do rozwiązywania innych problemów związanych z grafami
  - Znajdywanie ścieżki o minimalnej ilości krawędzi między dwoma wierzchołkami
  - Znajdywanie prostych cykli, jeśli takie istnieją

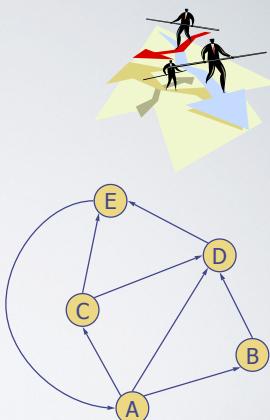
© 2004 Goodrich, Tamassia

## GRAFY SKIEROWANE



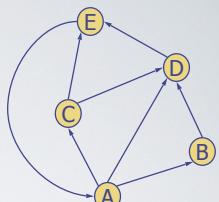
## DIGRAF

- Digraph** jest grafem, którego wszystkie krawędzie są skierowane
  - Skrót od "directed graph"
- Zastosowanie
  - ulice jednokierunkowe
  - loty
  - szeregowanie/planowanie zadań



© 2004 Goodrich, Tamassia

## WŁAŚCIWOŚCI

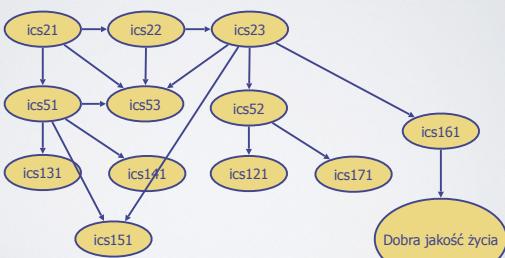


- Graf  $G=(V,E)$  taki, że
  - Każda krawędź jest skierowana:
    - Krawędź  $(a,b)$  idzie od  $a$  do  $b$ , ale nie od  $b$  do  $a$ .
- Jeśli  $G$  jest prosty, to  $m \leq n*(n-1)$ .
- Jeśli będziemy przechowywać krawędzie wejściowe i wyjściowe w osobnych listach sąsiedztwa, to możemy wymienić/wypisać te krawędzie czasie proporcjonalnym do rozmiaru list.

© 2004 Goodrich, Tamassia

## ZASTOSOWANIE DIGRAFU

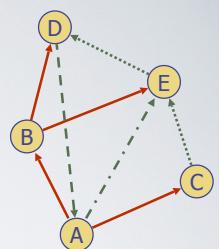
- Planowanie: krawędź  $(a,b)$  oznacza, że zadanie  $a$  musi zostać zakończone przed rozpoczęciem zadania  $b$



© 2004 Goodrich, Tamassia

## DFS DLA DIGRAFU

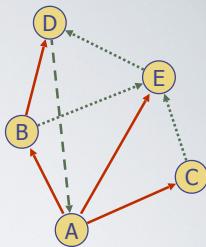
- Mogliśmy zmodyfikować algorytmy traversowania (DFS i BFS) dla digrafu w taki sposób aby traversowały krawędzie tylko zgodnie z ich kierunkiem



- W digrafach algorytm DFS będzie zawierał 4 rodzaje krawędzi:
  - krawędzie odnalezione → (wskaże na poprzednika)
  - krawędzie powrotnie ↗ (wskaże na następnego)
  - krawędzie następujące → (wskaże na każdy inny wierzchołek)
  - krawędzie poprzeczne ..... (wskaże na każdy inny wierzchołek)
- DFS dla digrafu rozpoczynając w wierzchołku  $s$  wyznacza wierzchołki osiągalne z  $s$

© 2004 Goodrich, Tamassia

## BFS DLA DIGRAFU



- W digrafach algorytm BFS będzie zawierał 3 rodzaje krawędzi:
  - krawędzie odnalezione → (wskazuje na poprzednika)
  - krawędzie powrotne - - - (wskazuje na każdy inny wierzchołek)
  - krawędzie poprzeczne ..... (wskazuje na każdy inny wierzchołek)
- BFS dla digrafu rozpoczynając w wierzchołku  $s$  także wyznacza wierzchołki osiągalne z  $s$

© 2004 Goodrich, Tamassia

## SZTUCZNA INTELIGENCJA

## SZTUCZNA INTELIGENCJA

Metody i algorytmy pozwalające na symulowanie inteligentnego zachowania komputera

zbliżające jego sposób podejmowania decyzji do ludzkiego.

Termin zaproponowany przez Johna McCarthy'ego w roku 1955.

## REPREZENTACJA WIEDZY

Problem oraz jego rozwiązanie jest zazwyczaj przedstawiane w sposób nieformalny

„dostarcz przesyłkę zaraz po jej otrzymaniu”

„napraw to co się zepsuło w zasilaniu kuchenki”



## SCHEMAT ROZWIAZYWANIA PROBLEMÓW PRZEZ KOMPUTER

- W celu rozwiązania problemu, programista musi:
  - uszczegółowić zadanie i wyznaczyć rozwiązanie;
  - przedstawić problem w języku, w którym komputer może dokonać wnioskowania;
  - wykorzystać komputer do obliczenia wyjścia
    - odpowiedz przedstawiona użytkownikowi
    - sekwenca niezbędnych do przeprowadzenia akcji
  - zinterpretować wyjście jako rozwiązanie problemu.

## ...ALE OD POCZĄTKU...

Idea programowania komputerów w celu „inteligentnego” zachowania.

- Po raz pierwszy zaproponowana przez...

kogóż innego jak nie Alan Turing, 1950

Pojęcie „sztucznej inteligencji” zdefiniowane przez John'a McCarthy'iego w 1955.

„Każdy aspekt uczenia lub innych cech inteligencji może być opisany w sposób, który maszyna jest w stanie zasymulować.”



J. McCarthy, '51

## CELE SZTUCZNEJ INTELIGENCJI

### Hipoteza „słaba” AI:

- Maszyny mogą zostać zaprogramowane tak, aby **wykazywały** inteligentne zachowania.
- prawdziwe: dowód Deep Blue, TD-Gammon, i inne.
- Programy wykorzystują metody znacznie różniące się od ludzkich.  
– Złożoność (zadań) vs. Symulacja (metody ludzkie).

### Hipoteza „silna” AI:

- Maszyny mogą zostać zaprogramowane tak, aby **posiadały** inteligentne zachowania.
- Czy muszą wykorzystywać metody zbliżone do mózgu (np. SN)

Searle zastosował eksperyment Chińskiego pokoju jako ścisły dowód istnienia silnej AI

Ale wielu się z tym nie zgadza!

## TEST TURINGA

### Sędzia wpisuje pytania

- Komputer twierdzi, że jest człowiekiem
- tak jak i człowiek



Jeśli sędzia nie potrafi odgadnąć za którym komputerem siedzi człowiek, to mówimy, że komputer przeszedł test i jest „inteligentny”

## EKSPERYMENT CHIŃSKIEGO POKOJU

### Załóżmy, że:

- Nie znamy języka Chińskiego.
- Jesteśmy sami w pokoju zawierającym szufladki „wejście” i „wyjście”.
- Posiadamy księgi pisma Chińskiego
- Posiadamy instrukcję po polsku (ale nie tłumaczenie), która objaśnia co należy napisać na karcie umieszczonej w szufladce „wyjście” w odpowiedzi na różne wejścia

### Następnie:

- Chińczyk po drugiej stronie ściany wkłada do szufladki „wejście” kartkę zapisaną po chińsku. Oni wiedzą, że to są pytania, my nie.
- Po zapoznaniu się z instrukcją wyznaczamy właściwą chińską odpowiedź, zapisujemy i oddajemy na „wyjście”

### I teraz pytanie:

- Ludzie na zewnątrz myślą, że znamy chiński. A znamy?
- Jeśli komputer zachowywałby się tak samo, to znałby chiński?

## PROBLEMY SZTUCZNEJ INTELIGENCJI

- rozwiązywanie problemów (gry np. szachy, zagadki logiczne, itp.)
- rozumowanie logiczne (dowodzenie twierdeń, projektowanie układów logicznych)
- przetwarzanie języka naturalnego (rozumienie mowy, tłumaczenie maszynowe, rozumienie języka, prowadzenie konwersacji)
- automatyczne programowanie (opis algorytmów przy pomocy języka naturalnego)

dr inż. D. Banasiak: Wykład z AI

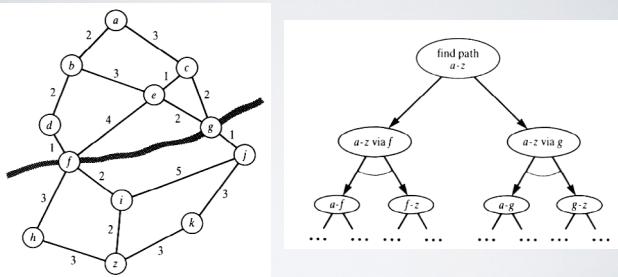
## STRATEGIE ROZWIAZYWANIA PROBLEMÓW

- Proste techniki przeszukiwania
- Dekompozycja problemu
  - grafu AND/OR
- Przeszukiwanie zależne od domeny problemu

## PRZESZUKIWANIE PROSTE

- Przeszukiwanie zachłanne
- Wzrost i spadek gradientu
- Przeszukiwanie stochastyczne
  - symulowane wyżarzanie
  - przeszukiwanie ewolucyjne
- Przeszukiwanie w głąb i wszerz

## DEKOMPOZYCJA PROBLEMU I GRAFY AND/OR



[Bratko 2001]

## PRZYKŁAD - GRA W PUZZLE

Przesuwaj kafelki dopóki nie będą poukładane w kolejności

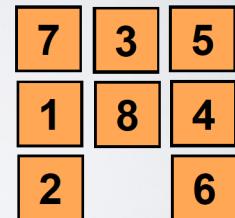
Oczywiście znajdziemy tę grę w sieci

Ale z jakiej korzystają strategii?

- Brak jednoznacznego algorytmu

Z tej pozycji,

- trzy kolejne pozycje
- z każdej z nich mamy
  - dwa, trzy lub cztery kolejne ruchy
  - i tak dalej

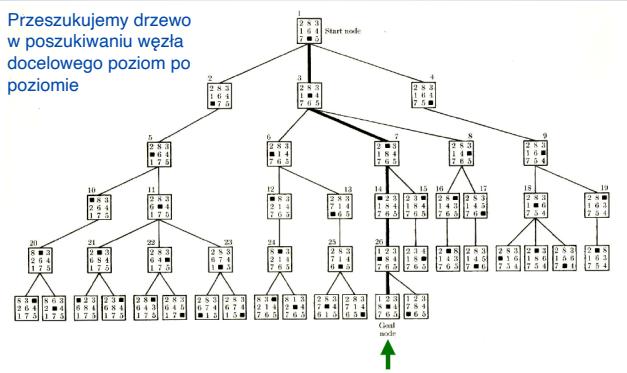


Czy to nie przypomina drzewa?

- oczywiście

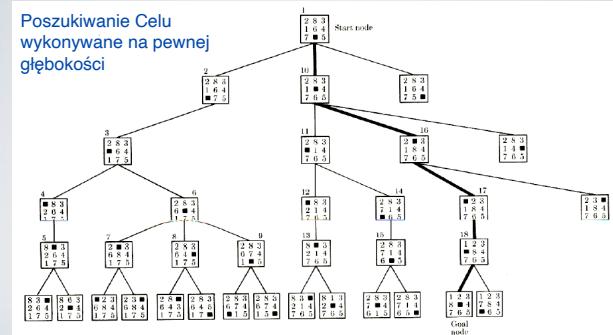
## PUZZLE: PRZESZUKIWANIE WSZERZ

Przeszukujemy drzewo w poszukiwaniu węzła docelowego poziom po poziomie



## PUZZLE: PRZESZUKIWANIE W GŁĘBÓ

Poszukiwanie Celu wykonywane na pewnej głębokości



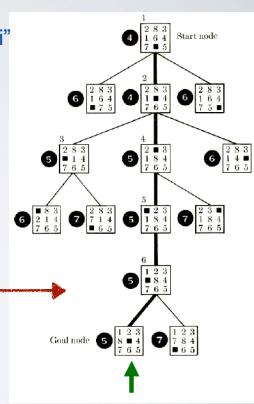
## PUZZLE: PRZESZUKIWANIE HEURYSTYCZNE

BFS i DFS są poszukiwaniami „ślepymi”

- Wymagające metody odnajdywania ścieżki
- Często niemożliwe
  - zbyt duża ilość rozwinięć węzłów
  - Ale sukces gwarantowany

Przeszukiwanie heurystyczne

- Wykorzystuje „funkcje oszacowania” do rankingu węzłów; wybieramy najlepszy
- Brak gwarancji sukcesu
- Np.: wykorzystaj odległość od startu + ilość kafelek nie na miejscu
- Wiele możliwych funkcji



## FUNKCJA HEURYSTYCZNA

- Funkcja  $h : \Psi \rightarrow \mathbb{R}$ , gdzie  $\Psi$  to zbiór dozwolonych stanów,  $\mathbb{R}$  to liczby rzeczywiste, odwzorowuje stany  $s$  zbioru  $\Psi$  na wartości  $h(s)$  służące do oceny względnych kosztów lub zysków rozwijania dalszej drogi przez węzeł odpowiadający  $s$ .

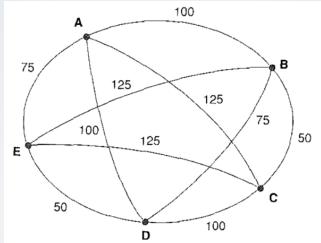
Węzeł A ma 3 potomków.

$$h(s1)=0.8, h(s2)=2.0, h(s3)=1.6$$

Wartości = koszty utworzenia węzła; najtańszej jest utworzyć węzeł  $s1$  i ten z punktu widzenia danej heurystyki jest najlepszym kandydatem.

prof. W. Duch, UMK

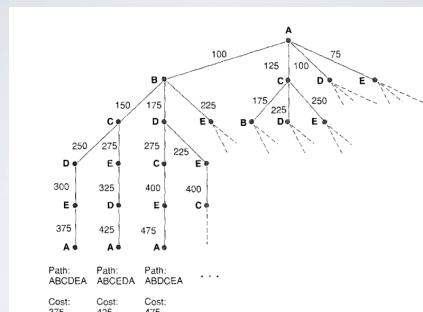
## PROBLEM KOMIWOJAZERA



Co zostanie wybrane?

prof. W. Duch, UMK

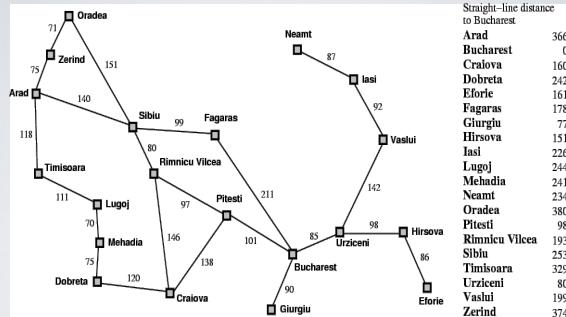
## DRZEWO PK



Drzewo DFS. Mamy  $(N-1)!$  dróg dla wszystkich permutacji, zakładając N miejsc i powrót do tego samego miejsca.

prof. W. Duch, UMK

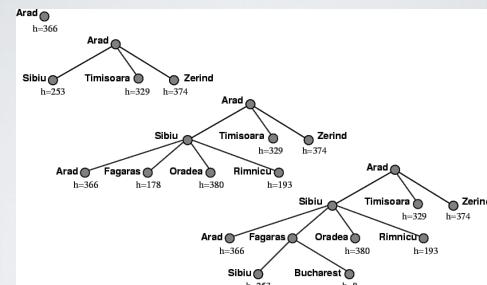
## ALGORYTMY ZACHŁANNE



Odległości miast na mapie Rumunii od Bukaresztu;  
 $h_{SLD}(n)$  = odległość w linii prostej do miasta n.

prof. W. Duch, UMK

## SZUKANIE ZACHŁANNE



$h(n)$  - odległość mierzona w linii prostej od Bukaresztu - podana w węzłach.

prof. W. Duch, UMK

## SZUKANIE A\*

- Szukanie zachłanne minimalizuje koszty dojścia do celu  $\rightarrow h(n)$
- nie jest to algorytm zupełny ani optymalny
- Lekarstwem będzie wprowadzenie drugiej funkcji  $g(n)$  i jej minimalizacja
  - minimalizacja kosztów dojścia do danego węzła
  - metoda kompletna, optymalna, ale mało efektywna
- Metoda A\* wykorzystuje obie te funkcje ( $h(n)$  i  $g(n)$ ) w jednej funkcji heurystycznej
  - $f(n) = g(n) + h(n)$
  - ocenia koszty najbliższego rozwiązania przechodzącego przez węzeł

prof. W. Duch, UMK

## ALGORYTM A\*

1. Począwszy od węzła początkowego tworzymy nowe węzły  $\{n\}$  do momentu osiągnięcia celu
2. Sortujemy nowe węzły z zastosowaniem funkcji:
 
$$f(n) = g(n) + h(n)$$
  - ścieżek zapętlonych nie bierzymy pod uwagę
  - Wybieramy najlepszy węzeł  $n'$
  - Zostawiamy tylko najbliższą ścieżkę do  $n'$
  - Jeśli  $n'$  jest celem  $\Rightarrow$  koniec
  - else  $\Rightarrow$  rozwijaj dalsze węzły łącznie z  $n'$

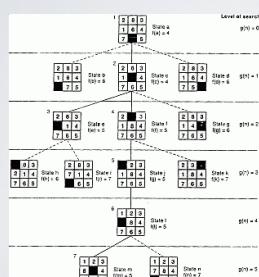
prof. W. Duch, UMK

## WŁAŚCIWOŚCI

- Ponieważ wybierana jest najtańsza droga do danego węzła n, to żadna inna droga nie może obniżyć całkowitego kosztu (monotoniczność).
- $h(n)$  powinno być wiarygodną oceną kosztów dojścia do celu
  - monotoniczne zaniedżenie wszystkich kosztów nie przeszkadza.
- Algorytm A\* jest w tym przypadku optymalny.

prof. W. Duch, UMK

## PRZYKŁAD A\* DLA PUZZLI



Przestrzeń stanów utworzona w czasie heurystycznego szukania 8-ki.

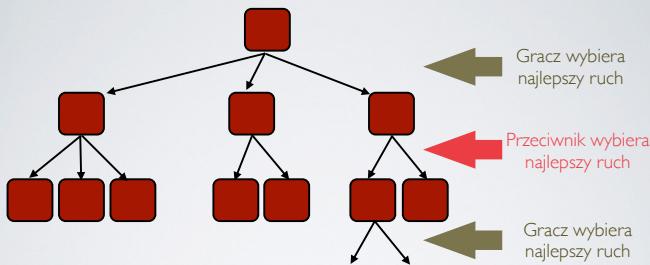
$$f(n) = g(n) + h(n)$$

$g(n)$  = odległość od startu do stanu n.

$h(n)$  = liczba elementów na złym miejscu.

prof. W. Duch, UMK

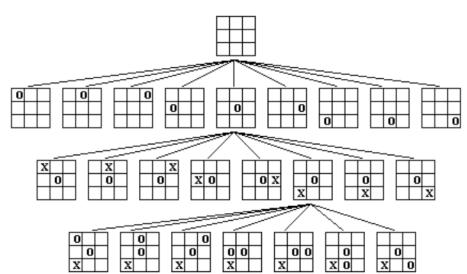
## A CO JEŚLI MAMY PRZECIWNIKA?



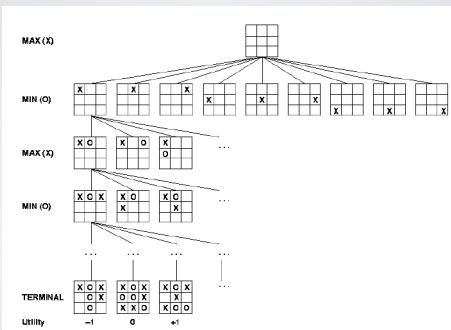
Powinniśmy wykonać ruch minimalizujący największe szanse przeciwnika

- A przeciwnik powinien zrobić to samo
- Metoda „Minimax” wykorzystuje heurystykę, która zakłada, że wykonujemy ruch najlepszy dla siebie (max) a przeciwnik wykona ruch najgorszy dla nas (min)

## PRZYKŁAD KÓŁKO I KRZYŻYK



## MINIMAX DLA KÓŁKA I KRZYŻYKA



## TWIERDZENIE MINIMAX

Każda skończona gra dwuosobowa o sumie zerowej ma co najmniej jedno rozwiązanie, które określa wartość gry i optymalne strategie graczy.

von Neumann (1928)

## STRATEGIA MINIMAXU

1. Utwórz drzewo dla gry do maksymalnej głębokości.
  - tak głębokie na ile damy rade
2. Przypisz liściom wartości f. heurystycznej.
3. Cofnij się o jeden poziom i dokonaj oceny węzłów.
  - wierzchołkom na kolejnych poziomach nadajemy wartości maks. dla gracza oraz min. dla przeciwnika
4. Po osiągnięciu korzenia wybierz decyzję maksymalizującą zyski.

## MINIMAX - ANALIZA

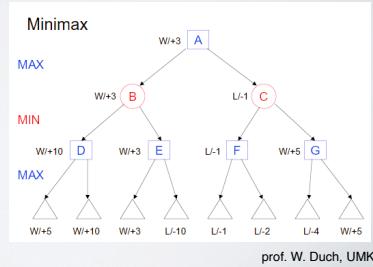
- Bazuje na teorii gier von Neumanna, Morgensterna z 1944
- Jest decyzją kompletną dla drzew skończonych
- Złożoność czasowa:  $O(b^m)$
- Złożoność pamięciowa:  $O(bm)$  - przeszukiwanie DFS

## MINIMAX - PRZYKŁAD

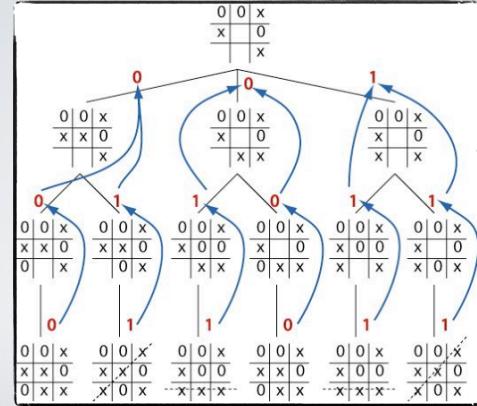
Wybór najlepszej drogi

W/g strategii, oceniamy informacje z liści i przechodzimy do kolejnego poziomu przenosząc najniższe lub najwyższe oceny na węzły z ruchami dla MAX i MIN

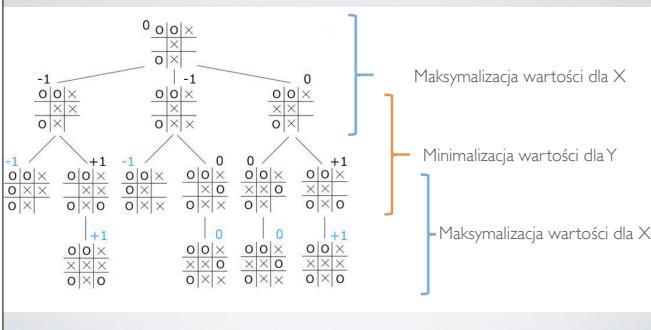
MAX powinien przejść do sytuacji B, skąd ma szansę dotrzeć do węzła  $W_{i+1} + 10$ , ale MIN zredukuje ją do  $W_{i+1}$ ; wybór przejścia do C daje MIN szansę zredukowania uzyskania przewagi o 1 punkt, czyli  $L-1$ .



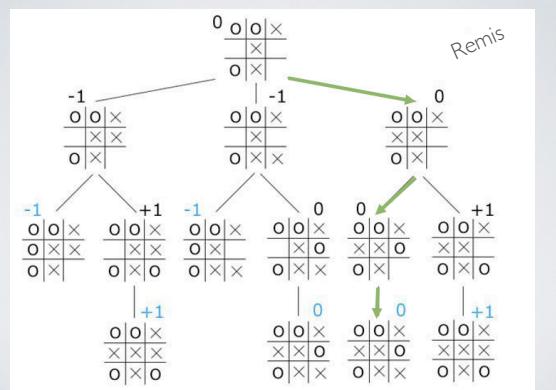
## MINIMAX - PRZYKŁAD



## MINIMAX - PRZYKŁAD

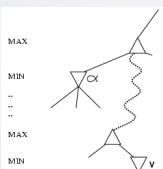


## MINIMAX - PRZYKŁAD



## CIĘCIA ALFA-BETA

- Jest modyfikacją strategii minmax
- Węzły nie mające wpływu na wartość przypisywaną do przodków są eliminowane z dalszej analizy
  - oszczędność czasu (mniejsze drzewo gry)
- Dwa rodzaje cięć:
  - cięcie  $\alpha$  - najlepsza wartość dla MAXa dla kolejnych kroków;
    - porzuć V jeśli istnieją lepsze wartości  $a$  w innej części grafu (dla MAX);
  - cięcie  $\beta$  - najlepsza wartość dla MINa dla kolejnych kroków;

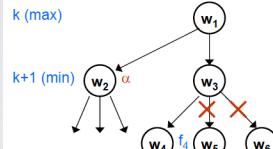


Złożoność czasowa:  $O(b^{m/2})$

prof. W. Duch, UMK

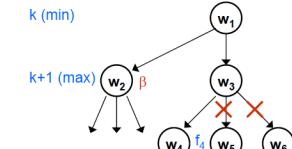
## CIĘCIA ALFA-BETA

### Cięcie $\alpha$ :



jeżeli  $f_4 \leq \alpha$ , to wierzchołków  
w<sub>5</sub> i w<sub>6</sub> nie musimy rozpatrywać

### Cięcie $\beta$ :

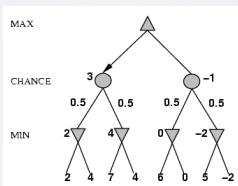


jeżeli  $f_4 \geq \beta$ , to wierzchołków  
w<sub>5</sub> i w<sub>6</sub> nie musimy rozpatrywać

dr inż. D. Banasiak: Wykład z AI

## GRY NIEDETERMINISTYCZNE

- Strategia min-max połączona z oceną probabilistyczną szans na generowanie kolejnego ruchu
- Obcinanie  $\alpha$ - $\beta$  można zastosować, ale przestaje być efektywne,
  - wzrasta liczba możliwych rozgałęzień.



prof. W. Duch, UMK

# PROJEKTOWANIE I ANALIZA ALGORYTMÓW

## SZTUCZNA INTELIGENCJA

**Wykład**  
dr inż. Łukasz Jeleń

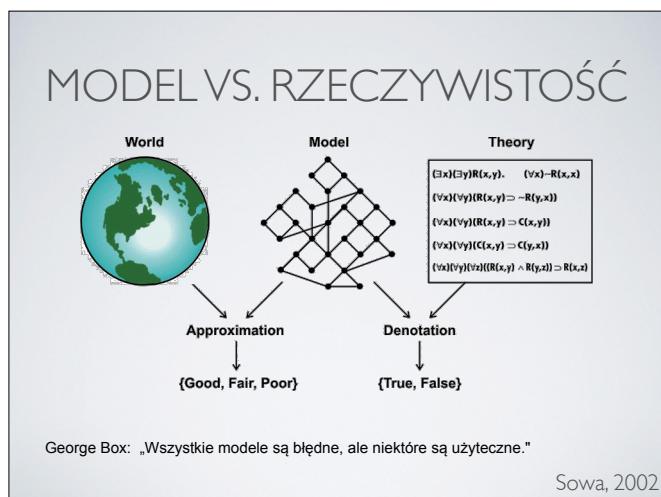
Część slajdów pochodzi z wykładu prof. Christiana Jacoba z Uniwersytetu w Calgary oraz wykładów prof. Włodzisława Ducha z UMK i dr. inż. D. Banasiaka, PWr

## SZTUCZNA INTELIGENCJA

Metody i algorytmy pozwalające na symulowanie intelligentnego zachowania komputera

zblążające jego sposób podejmowania decyzji do ludzkiego.

Termin zaproponowany przez Johna McCarthy'ego w roku 1955.

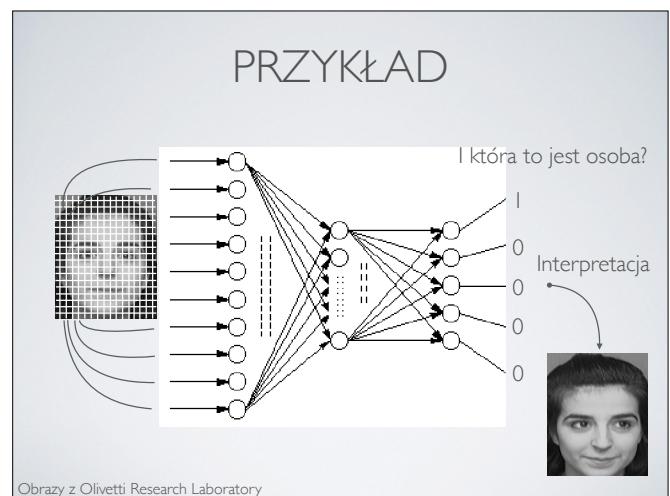


## REPREZENTACJA WIEDZY

Problem oraz jego rozwiązanie jest zazwyczaj przedstawiane w sposób nieformalny  
 „dostarcz przesyłkę zaraz po jej otrzymaniu”  
 „napraw to co się zepsuło w zasilaniu kuchenki”



- ## SCHEMAT ROZWIAZYWANIA PROBLEMÓW PRZEZ KOMPUTER
- W celu rozwiązania problemu, programista musi:
    - uszczegółowić zadanie i wyznaczyć rozwiązanie;
    - przedstawić problem w języku, w którym komputer może dokonać wnioskowania;
    - wykorzystać komputer do obliczenia wyjścia
      - odpowiedź przedstawiona użytkownikowi
      - sekwenca niezbędnych do przeprowadzenia akcji
    - zinterpretować wyjście jako rozwiązanie problemu.



## WIEDZA

- Jest informacją o domenie, która może zostać wykorzystana do rozwiązywania problemów w tej dziedzinie
- Rozwiązanie problemu wymaga reprezentacji wiedzy przez komputer
  - Programista musi zdefiniować jak będzie ona reprezentowana
  - Reprezentacja jest formą wiedzy wykorzystywaną jako agent
- Schemat reprezentacji jest specyficzną formą wiedzy
- Dobra reprezentacja wiedzy stanowi kompromis między różnymi celami

## DANE, INFORMACJA, WIEDZA



A. Straszak: Społeczeństwo oparte na wielkich zasobach wiedzy

## PRZYKŁAD

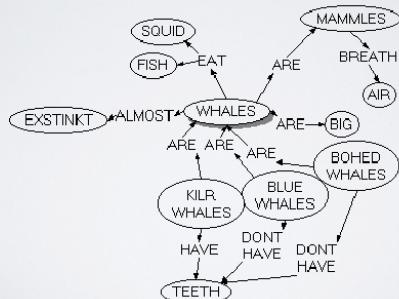
Jabłko jest owocem. Owoc jest słodki



Seth Russell

## PRZYKŁAD 2

Przykład stworzony przez sześciolatka opisujący jego stan wiedzy o wielorybach



Seth Russell

## ...ALE OD POCZĄTKU...

Idea programowania komputerów w celu „inteligentnego” zachowania.

- Po raz pierwszy zaproponowana przez...  
kogóż innego jak nie Alan Turing, 1950

Pojęcie „sztucznej inteligencji” zdefiniowane przez John'a McCarthy'iego w 1955.

„Każdy aspekt uczenia lub innych cech inteligencji może być opisany dokładnie tak, aby maszyna była w stanie to zasymulować.”



J. McCarthy, '51

## CELE SZTUCZNEJ INTELIGENCJI

### Hipoteza „słaba” AI:

- Maszyny mogą zostać zaprogramowane tak, aby wykazywały inteligentne zachowania.
- prawdziwe: dowód Deep Blue, TD-Gammon, i inne.
- Programy wykorzystują metody znacznie różniące się od ludzkich.
  - Złożoność (zadań) vs. Symulacja (metody ludzkie).

### Hipoteza „silna” AI:

- Maszyny mogą zostać zaprogramowane tak, aby posiadały inteligentne zachowania.
- Czy muszą wykorzystywać metody zbliżone do mózgu (np. SN)

Searle zastosował eksperyment Chińskiego pokoju jako ścisły dowód istnienia silnej AI

Ale wielu się z tym nie zgadza!

## TEST TURINGA

Sędzia wpisuje pytania

- Komputer twierdzi, że jest człowiekiem
- tak jak i człowiek



Jeśli sędzia nie potrafi odgadnąć za którym komputerem siedzi człowiek, to mówimy, że komputer przeszedł test i jest „inteligentny”

## EKSPERYMENT CHIŃSKIEGO POKOJU

Założymy, że:

- Nie znamy języka Chińskiego.
- Jesteśmy sami w pokoju zawierającym szufladki „wejście” i „wyjście” .
- Posiadamy księgi pisma Chińskiego
- Posiadamy instrukcję po polsku (ale nie tłumaczenie), która objaśnia co należy napisać na kartce umieszczanej w szufladce „wyjście” w odpowiedzi na różne wejścia

Następnie:

- Chińczyk po drugiej stronie ściany wkłada do szufladki „wejście” kartkę zapisaną po chińsku. Oni wiedzą, że to są pytania, my nie.
- Po zapoznaniu się z instrukcją wyznaczamy właściwą chińską odpowiedź, zapisujemy i oddajemy na „wyjście”

I teraz pytanie:

- Ludzie na zewnątrz myślą, że znamy chiński. A znamy?
- Jeśli komputer zachowywałby się tak samo, to znałby chiński?

## PROBLEMY SZTUCZNEJ INTELIGENCJI

- rozwiązywanie problemów (gry np. szachy, zagadki logiczne, itp.)
- rozumowanie logiczne (dowodzenie twierdzeń, projektowanie układów logicznych)
- przetwarzanie języka naturalnego (rozumienie mowy, tłumaczenie maszynowe, rozumienie języka, prowadzenie konwersacji)
- automatyczne programowanie (opis algorytmów przy pomocy języka naturalnego)

dr inż. D. Banasiak: Wykład z AI

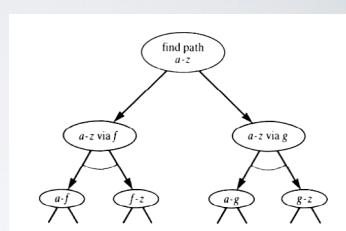
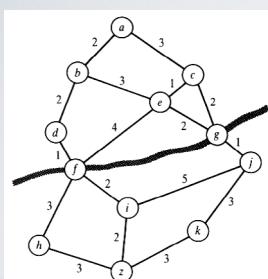
## STRATEGIE ROZWIAZYWANIA PROBLEMÓW

- Proste techniki przeszukiwania
- Dekompozycja problemu
  - grafu AND/OR
- Przeszukiwanie zależne od domeny problemu

## PRZESZUKIWANIE PROSTE

- Przeszukiwanie zachłanne
- Wzrost i spadek gradientu
- Przeszukiwanie stochastyczne
  - symulowane wyżarzanie
  - przeszukiwanie ewolucyjne
- Przeszukiwanie w głąb i wszerz

## DEKOMPOZYCJA PROBLEMU I GRAFY AND/OR



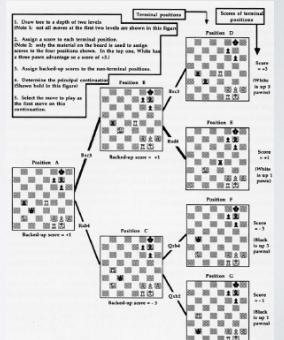
[Bratko 2001]

## PRZESZUKIWANIE ZALEŻNE OD DOMENY PROBLEMU



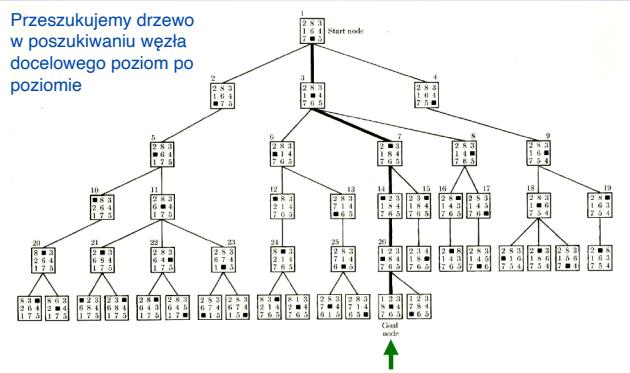
Anatoli Karpow i Garii Kasparow

[Christian Jacob]



## PUZZLE: PRZESZUKIWANIE WSZERZ

Przeszukujemy drzewo w poszukiwaniu węzła docelowego poziom po poziomie



## PRZYKŁAD - GRA W PUZZLE

Przesuwaj kafelki dopóki nie będą poukładane w kolejności

Oczywiście znajdziemy tę grę w sieci

Ale z jakiej korzystają strategii?

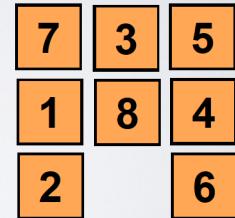
- Brak jednoznacznego algorytmu

Z tej pozycji,

- trzy kolejne pozycje
- z każdej z nich mamy
  - dwa, trzy lub cztery kolejne ruchy
  - i tak dalej

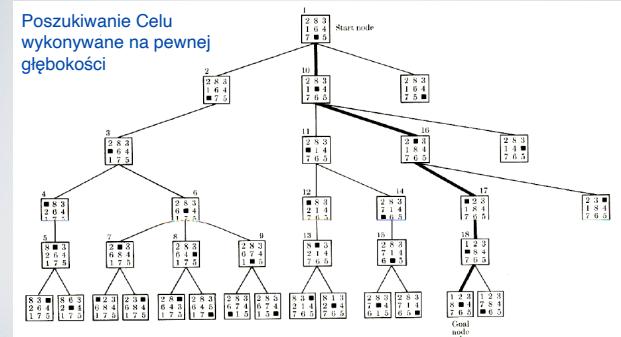
Czy to nie przypomina drzewa?

- oczywiście



## PUZZLE: PRZESZUKIWANIE W GŁĘBÓKU

Poszukiwanie Celu wykonywane na pewnej głębokości



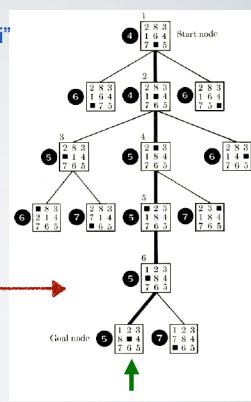
## PUZZLE: PRZESZUKIWANIE HEURYSTYCZNE

BFS i DFS są poszukiwaniami „słepymi”

- Wymagające metody odnajdywania ścieżki
- Często niemożliwe
  - zbyt duża ilość rozwinięć węzłów
- Ale sukces gwarantowany

Przeszukiwanie heurystyczne

- Wykorzystuje „funkcje oszacowania” do rankingu węzłów; wybieramy najlepszy
- Brak gwarancji sukcesu
- Np.: wykorzystaj odległość od startu + ilość kafelek nie na miejscu
- Wiele możliwych funkcji



## FUNKCJA HEURYSTYCZNA

- Funkcja  $h : \Psi \rightarrow \mathbb{R}$ , gdzie  $\Psi$  to zbiór dozwolonych stanów,  $\mathbb{R}$  to liczby rzeczywiste, odwzorowuje stany  $s$  zbioru  $\Psi$  na wartości  $h(s)$  służące do oceny względnych kosztów lub zysków rozwijania dalszej drogi przez węzeł odpowiadający  $s$ .

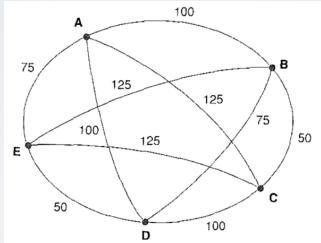
Węzeł A ma 3 potomków.

$$h(s1)=0.8, h(s2)=2.0, h(s3)=1.6$$

Wartości = koszty utworzenia węzła; najtańszej jest utworzyć węzeł  $s1$  i ten z punktu widzenia danej heurystyki jest najlepszym kandydatem.

prof. W. Duch, UMK

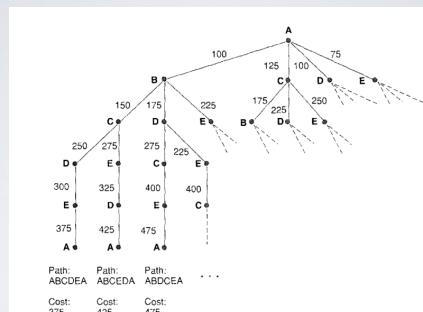
## PROBLEM KOMIWOJAZERA



Co zostanie wybrane?

prof. W. Duch, UMK

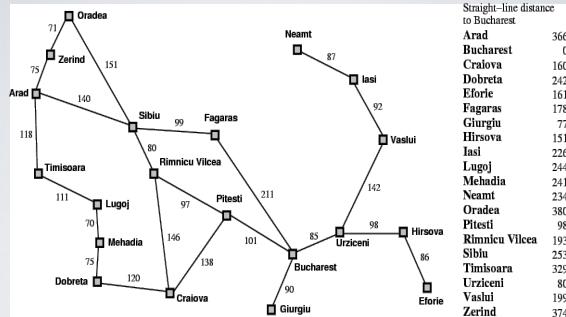
## DRZEWO PK



Drzewo DFS. Mamy  $(N-1)!$  dróg dla wszystkich permutacji, zakładając N miejsc i powrót do tego samego miejsca.

prof. W. Duch, UMK

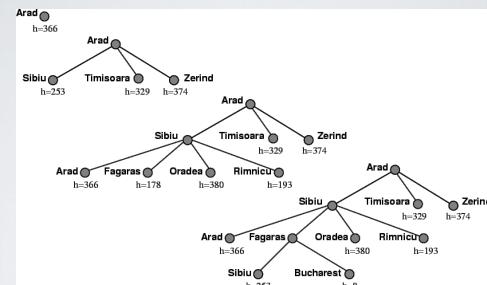
## ALGORYTMY ZACHŁANNE



Odległości miast na mapie Rumunii od Bukaresztu;  
 $h_{SLD}(n)$  = odległość w linii prostej do miasta n.

prof. W. Duch, UMK

## SZUKANIE ZACHŁANNE



$h(n)$  - odległość mierzona w linii prostej od Bukaresztu - podana w węzłach.

prof. W. Duch, UMK

## SZUKANIE A\*

- Szukanie zachłanne minimalizuje koszty dojścia do celu  $\rightarrow h(n)$
- nie jest to algorytm zupełny ani optymalny
- Lekarstwem będzie wprowadzenie drugiej funkcji  $g(n)$  i jej minimalizacja
  - minimalizacja kosztów dojścia do danego węzła
  - metoda kompletna, optymalna, ale mało efektywna
- Metoda A\* wykorzystuje obie te funkcje ( $h(n)$  i  $g(n)$ ) w jednej funkcji heurystycznej
  - $f(n) = g(n) + h(n)$
  - ocenia koszty najbliższego rozwiązania przechodzącego przez węzeł

prof. W. Duch, UMK

## ALGORYTM A\*

1. Począwszy od węzła początkowego tworzymy nowe węzły  $\{n\}$  do momentu osiągnięcia celu
2. Sortujemy nowe węzły z zastosowaniem funkcji:
 
$$f(n) = g(n) + h(n)$$
  - ścieżek zapętlonych nie bierzymy pod uwagę
  - Wybieramy najlepszy węzeł  $n'$
  - Zostawiamy tylko najbliższą ścieżkę do  $n'$
  - Jeśli  $n'$  jest celem  $\Rightarrow$  koniec
  - else  $\Rightarrow$  rozwijaj dalsze węzły łącznie z  $n'$

prof. W. Duch, UMK

## WŁAŚCIWOŚCI

- Ponieważ wybierana jest najtańsza droga do danego węzła n, to żadna inna droga nie może obniżyć całkowitego kosztu (monotoniczność).
- $h(n)$  powinno być wiarygodną oceną kosztów dojścia do celu
  - monotoniczne zaniedżenie wszystkich kosztów nie przeszkadza.
- Algorytm A\* jest w tym przypadku optymalny.

prof. W. Duch, UMK

## A\* ITERACYJNIE POGŁĘBIANIE

### IDA\*

- Wykorzystujemy algorytm DFS
- Oceniamy całkowity koszt  $f(n) = g(n) + h(n)$  heurystyką A\*
- If  $f(n) > T$  (próg) => cofnij się
- else => rozwiązywanie nieznalezione, zwiększamy T i powtarzamy

#### Wady:

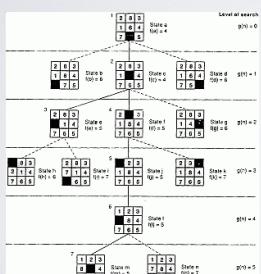
- Część ścieżek jest odwiedzana więcej niż raz
- końcowe szukanie zajmuje dużo czasu

#### Zalety:

- Mała pamięć

prof. W. Duch, UMK

## PRZYKŁAD A\* DLA PUZZLI



Przestrzeń stanów utworzona w czasie heurystycznego szukania 8-ki.

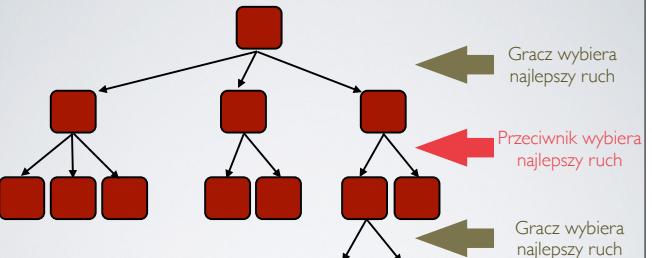
$$f(n) = g(n) + h(n)$$

$g(n)$  = odległość od startu do stanu n.

$h(n)$  = liczba elementów na złym miejscu.

prof. W. Duch, UMK

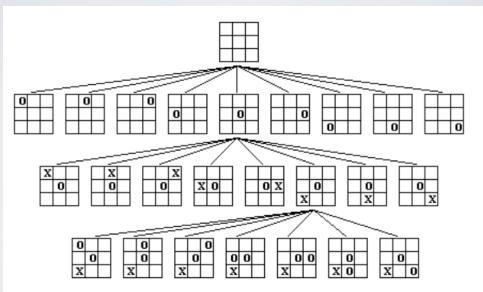
## A CO JEŚLI MAMY PRZECIWNIKĄ?



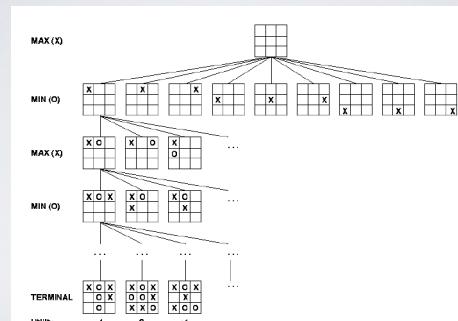
Powinniśmy wykonać ruch minimalizujący największe szanse przeciwnika

- A przeciwnik powinien zrobić to samo
- Metoda „Minimax” wykorzystuje heurystykę, która zakłada, że wykonujemy ruch najlepszy dla siebie (max) a przeciwnik wykona ruch najgorszy dla nas (min)

## PRZYKŁAD KÓŁKO I KRZYŻYKA



## MINIMAX DLA KÓŁKA I KRZYŻYKA



## TWIERDZENIE MINIMAX

Każda skończona gra dwuosobowa o sumie zerowej ma co najmniej jedno rozwiązanie, które określa wartość gry i optymalne strategie graczy.

von Neumann (1928)

## STRATEGIA MINIMAXU

1. Utwórz drzewo dla gry do maksymalnej głębokości.

- tak głębokie na ile damy radę

2. Przypisz liściom wartości f. heurystycznej.

3. Cofnij się o jeden poziom i dokonaj oceny węzłów.

- wierzchołkom na kolejnych poziomach nadajemy wartości maks. dla gracza oraz min. dla przeciwnika

4. Po osiągnięciu korzenia wybierz decyzję maksymalizującą zyski.

## MINIMAX - ANALIZA

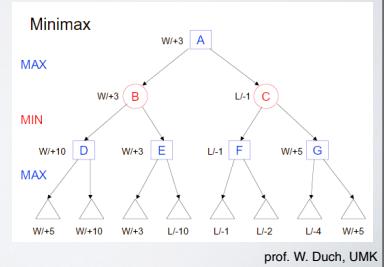
- Bazuje na teorii gier von Neumanna, Morgensterna z 1944
- Jest decyzją kompletną dla drzew skończonych
- Złożoność czasowa:  $O(b^m)$
- Złożoność pamięciowa:  $O(bm)$  - przeszukiwanie DFS

## MINIMAX - PRZYKŁAD

Wybór najlepszej drogi

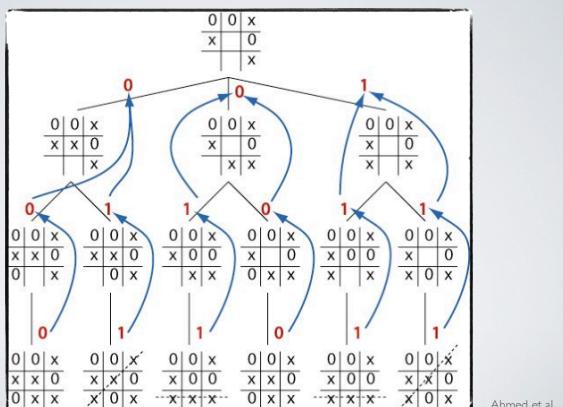
W/g strategii, oceniamy informacje z liści i przechodzimy do kolejnego poziomu przenosząc najniższe lub najwyższe oceny na węzły z ruchami dla MAX i MIN

MAX powinien przejść do sytuacji B, skąd ma szansę dotrzeć do węzła W/+10, ale MIN zredukuje ją do W/+3; wybór przejścia do C daje MIN szansę zredukowania uzyskania przewagi o 1 punkt, czyli L-1.



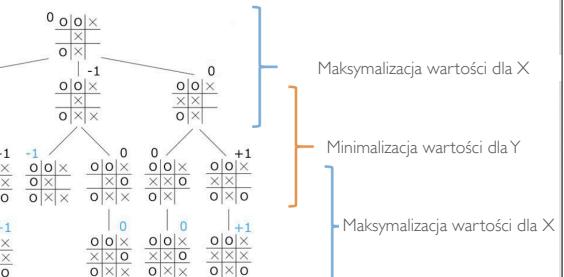
prof. W. Duch, UMK

## MINIMAX - PRZYKŁAD



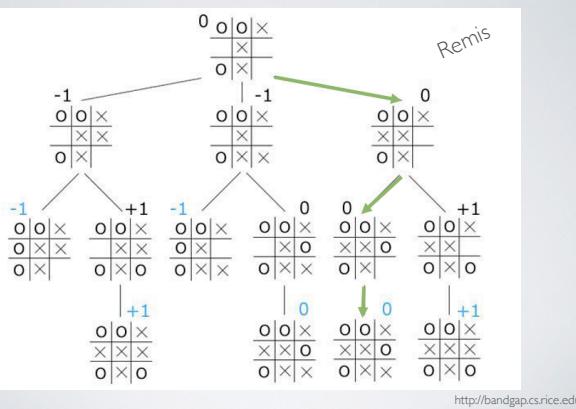
Ahmed et al.

## MINIMAX - PRZYKŁAD



<http://bandgap.cs.rice.edu>

## MINIMAX - PRZYKŁAD



<http://bandgap.cs.rice.edu>

## PRZYKŁAD - SZACHY

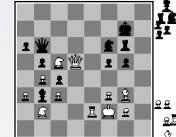
- Ludzie oceniają jakościowo, wystarczą względne porównania.
- Na podstawie doświadczenia tworzy się złożone funkcje oceny.



Ruch czarnych  
Przewaga białych



Ruch białych  
Czarne wygrywają



Ruch czarnych  
Białe bliskie  
przegranej

prof. W. Duch, UMK

## FUNKCJE OCENY

- Koszty:
  - pionek=1, skoczek=3, gonięcy=3, wieża=5...
- Pozycja figur: ocena zależna od sytuacji
  - pionek przy końcu planszy jest istotniejszy niż zablokowany
- Ocena kolejnego ruchu, zagrożeń dla figur
- Złożone oceny konfiguracji wielu figur
- Połączenie liniowe różnych elementów oceny  $f_i$  z wagami  $W_i$

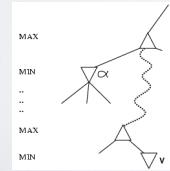
$$E(f) = \sum_i W_i f_i$$

- $W_i$  jest dobierany w celu maksymalizacji zysków
  - nieliniowe funkcje mogą być lepsze

prof. W. Duch, UMK

## CIĘCIA ALFA-BETA

- Jest modyfikacją strategii minmaxs
- Węzły nie mające wpływu na wartość przypisywaną do przodków są eliminowane z dalszej analizy
  - oszczędność czasu (mniejsze drzewo gry)
- Dwa rodzaje cięć:
  - cięcie  $\alpha$  - najlepsza wartość dla MAXa dla kolejnych kroków;
    - porzuć V jeśli istnieją lepsze wartości  $\alpha$  w innej części grafu (dla MAX);
  - cięcie  $\beta$  - najlepsza wartość dla MINa dla kolejnych kroków;

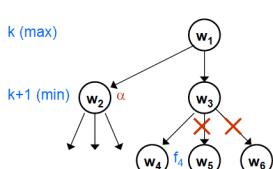


Złożoność czasowa:  $O(b^{m/2})$

prof. W. Duch, UMK

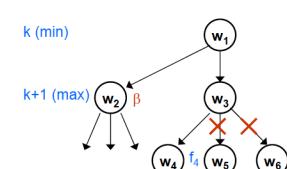
## CIĘCIA ALFA-BETA

### Cięcie $\alpha$ :



jeżeli  $f_4 \leq \alpha$ , to wierzchołki  
 $w_5$  i  $w_6$  nie musimy rozpatrywać

### Cięcie $\beta$ :

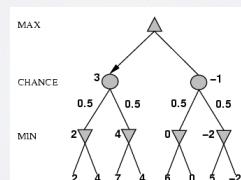


jeżeli  $f_4 \geq \beta$ , to wierzchołki  
 $w_5$  i  $w_6$  nie musimy rozpatrywać

dr inż. D. Banasiak: Wykład z AI

## GRY NIEDETERMINISTYCZNE

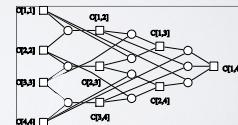
- Strategia min-max połączona z oceną probabilistyczną szans na generowanie kolejnego ruchu
- Obcinanie  $\alpha$ - $\beta$  można zastosować, ale przestaje być efektywne,
  - wzrasta liczba możliwych rozgałęzień.



prof. W. Duch, UMK



## PROGRAMOWANIE DYNAMICZNE



### CIĄG FIBONACCIEGO

- Ciąg Fibonacciego: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
 
$$F_i = i \quad \text{jeśli } i \leq 1$$

$$F_i = F_{i-1} + F_{i-2} \quad \text{jeśli } i \geq 2$$
- Rozwiązywanie w sposób rekurencyjny:
- Wiele powtarzanych obliczeń
- Powinien być rozwiązywany z pomocą jednej pętli

<http://algorytmika.wikidot.com/>

## PROGRAMOWANIE DYNAMICZNE

- Metoda projektowania algorytmów wykorzystywana w sytuacjach kiedy dany problem może zostać przedstawiony jako sekwencja decyzji
- rozwiązywanie problemu nie jest traktowane jako pojedynczy problem
  - wiele podproblemów
    - muszą być niezależne od siebie
    - rozpoczynamy od najprostszych
- matematyczna technika wyznaczania rozwiązań optymalnych

### CIĄG FIBONACCIEGO

Wyznaczanie n-tego wyrazu CF

wersja rekurencyjna

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

fib(3)=fib(1)+fib(2)=fib(1)+fib(0)+fib(1)
      =1+0+1=2
```

Czas:  $O(2^n)$

<http://algorytmika.wikidot.com/>

### CIĄG FIBONACCIEGO

Wyznaczanie n-tego wyrazu CF  
z zastosowaniem programowania dynamicznego

```
int fibonacci(int n) {
    int a, b;
    if(n == 0) return 0;
    a = 0; b = 1;
    for(int i=0; i<(n-1); i++)
    {
        b += a;
        a = b-a;
    }
    return b;
}
```

Czas:  $O(n)$

<http://algorytmika.wikidot.com/>

## PRZYKŁAD - OBLCZANIE SYMBOLU NEWTONA

wersja rekurencyjna

$$\binom{n}{k}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$\binom{n}{k} = \begin{cases} 1 & \text{dla } k = 0 \text{ lub } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{dla } 0 < k < n \end{cases}$$

nieoptymalne - prowadzi do czasu wykładniczego

$$\binom{n-1}{k-1}$$

$$i$$

$\binom{n-1}{k}$  może powodować wywoływanie wykładniczą ilość rekurencji

<http://algorytmika.wikidot.com/>

## PRZYKŁAD - OBLCZANIE SYMBOLU NEWTONA

programowanie dynamiczne      wersja iteracyjna  
mała liczba podproblemów ->  $O(n^2)$

```
for ( int i = 1; i<= n; i++)
    tab[i][0] = 1;
.....
function symbol(int n,int k) {
    for (int j = 1; j<=k; j++)
        tab[j][j]= 1;
    for (int i = j + 1; i<=n; i++)
        tab[i][j] = tab[i-1][j-1] + tab[i-1][j];
    return tab[n][k];
}
```

## KONCEPCJA PROGRAMOWANIA DYNAMICZNEGO

programowanie dynamiczne -> rozwiązywanie problemu "od końca"

• Kroki:

- rozwiąż problem dla jednego elementu
  - dla różnych wartości parametru sterującego
  - zapamiętaj wyniki,
- dodaj kolejny element do problemu
- zbuduj rozwiązanie problemu powiększonego o nowy składnik
  - wybór dokonywany optymalnie w oparciu o wcześniejsze rozwiązania (zpisane)

## ZASADA OPTYMALIZACJI

- Założmy, że podczas rozwiązywania problemu mamy dokonać decyzji  $D_1, D_2, \dots, D_n$ .
- Jeśli sekwencja ta jest optymalna to ostatnie k decyzji,  $1 < k < n$  musi być optymalne
  - np.: problem znajdywania najkrótszej ścieżki
  - Jeśli  $i_1, i_2, \dots, j$  jest najkrótszą ścieżką od  $i$  do  $j$ , to  $i_1, i_2, \dots, j$  musi być najkrótszą ścieżką od  $i_1$  do  $j$
- Jeżeli problem może być opisany przez wielopoziomowy graf, to może zostać rozwiązany z zastosowaniem programowania dynamicznego

## PROGRAMOWANIE DYNAMICZNE

• Dwa podejścia:

- Postępujące (Forward)
  - jeśli problem jest zdefiniowany postępująco, to relacje są rozwiązywane wstecznie
    - rozpoczynając od ostatniej decyzji
  - wsteczne (Backward)
    - analogicznie, jeśli relacje są formułowane wstecznie, to będą rozwiązywane postępująco

• Do rozwiązania problemu z zastosowaniem programowania dynamicznego musimy:

- Znaleźć relacje rekurencyjne
- Przedstawić problem w postaci grafu wielopoziomowego

## PROBLEM ALOKACJI ZASOBÓW

mamy  $m$  zasobów i  $n$  projektów

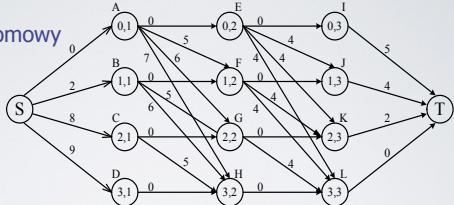
zysk  $z(i, j)$ :  $j$  zasobów zostało przydzielonych do projektu  $i$

maksymalizujemy zysk całkowity

Resource Project			
	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

## ROZWIĄZANIE GRAFOWE

Graf wielopoziomowy



- Problem alokacji zasobów może być przedstawiony w formie grafu wielopoziomowego
- $(i, j)$ :  $i$  zasobów przydzielonych do projektów  $1, 2, \dots, j$
- n.p.: węzeł H =  $(3, 2)$ : 3 zasoby przydzielone do projektów 1, 2

## PROBLEM KOMIWOJAZERA

Komiwojażer chce odwiedzić dany zbiór miast i powrócić do początku przemierzając najmniejszy możliwy dystans



Problem łatwy do opisania

ale...

trudny do rozwiązania

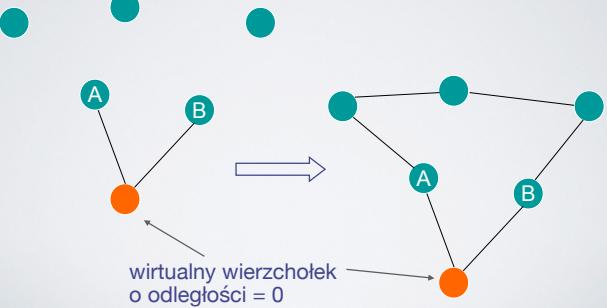
## PROBLEM KOMIWOJAZERA

Nawet jeżeli nie mamy wymogu powrotu do początku, to i tak problem może być traktowany jako PK

Założymy, że chcemy przejechać z miasta A do B odwiedzając wszystkie miasta

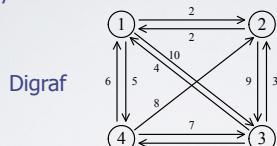


## PROBLEM KOMIWOJAZERA



## PROBLEM KOMIWOJAZERA

Inny przykład



Macierz kosztów

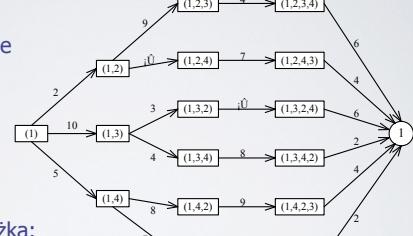
	1	2	3	4
1	$\infty$	2	10	5
2	2	$\infty$	9	$\infty$
3	4	3	$\infty$	4
4	6	8	7	$\infty$

## ROZWIĄZANIE GRAFOWE

Móže opisać wszystkie możliwe przejścia digrafu

Najkrótsza ścieżka:

$(1, 4, 3, 2, 1)$   
 $5+7+3+2=17$



## REPREZENTACJA WIERZCHOŁKA

- Założymy, że w grafie mamy 6 wierzchołków.
  - Możemy połączyć  $\{1, 2, 3, 4\}$  i  $\{1, 3, 2, 4\}$  w jeden wierzchołek.
- (a)

combine

(b)
- $(3), (4, 5, 6)$  oznacza, że ostatni odwiedzony wierzchołek to 3, a pozostałe wierzchołki do odwiedzenia to  $(4, 5, 6)$

## PROGRAMOWANIE DYNAMICZNE

- Niech  $g(i, S)$  będzie długością najkrótszej ścieżki począwszy od wierzchołka  $i$ , przechodzącą przez wszystkie wierzchołki  $S$  i skończywszy w wierzchołku 1.

- Długość optymalnej podróży:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\}$$

- W formie ogólnej:

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

- Złożoność czasowa:

$$\begin{aligned} & n + \sum_{k=2}^n (n-1) \binom{n-2}{n-k} (n-k) \\ & = O(n^2 2^n) \end{aligned}$$

## TEORIA ZŁOŻONOŚCI



## TEORIA ZŁOŻONOŚCI



**Teoria złożoności** poszukuje rozwiązań dla problemów, które są algorytmicznie trudne do rozwiązania

W części poświęconej **Algorytmom i Strukturam Danych** dowiedzieliśmy się w jaki sposób mierzyć/oszacowywać złożoność konkretnych algorytmów poprzez asymptotyczny pomiar ilości kroków

W dziedzinie **teorii złożoności** możemy mieć do czynienia z problemami, które nie będą rozwiązywalne algorytmicznie.

## PROBLEMY I ALGORYTMY

Sortowanie przez wstawianie działa w czasie  $O(n^2)$ , a Sortowanie przez Scalanie jest algorymem o złożoności  $O(n \log n)$ .

Pierwsza część zdania oznacza:

Jeśli policzymy ilość kroków wykonywanych przez algorytm sort. przez wstawianie dla wejścia o rozmiarze  $n$ , biorąc największą liczbę spośród wszystkich wejść o takim rozmiarze, to tak zdefiniowana funkcja  $n$  będzie ograniczona przez stałą wielokrotność  $n^2$ .

Sensownym jest porównywanie tych dwóch algorytmów - poszukując rozwiązań tego samego problemu

ale...

jaka jest złożoność **problemu sortowania???**

## OGRANICZENIE DOLNE I GÓRNE

Jaka jest złożoność czasowa najszybszego algorytmu sortującego listę?

Przez analizę algorytmu sort. przez scalanie wiemy, że nie jest gorsza niż  **$O(n \log n)$** .

Złożoność konkretnego algorytmu wyznacza **górne ograniczenie** złożoności problemu

Do wyznaczenia **ograniczenia dolnego** potrzebujemy wykazać, że nie istnieje algorytm (nawet taki, który nie został jeszcze wymyślony), który może być lepszy.

W przypadku sortowania możemy wyznaczyć ograniczenie dolne jako  $\Omega(n \log n)$ . Sortowanie jest rzadkim przypadkiem, gdzie znane ograniczenia dolne i górne są takie same.

## PRZYPOMNIENIE

Złożoność algorytmu (bez znaczenia, czy czasowa, czy pamięciowa) jest przeważnie opisywana asymptotycznie:

Definicja:

Dla funkcji  $f: \mathbb{N} \rightarrow \mathbb{N}$  i  $g: \mathbb{N} \rightarrow \mathbb{N}$  to powiemy, że

- $f = O(g)$  jeśli istnieje  $n_0 \in \mathbb{N}$  i stała  $c$  taka, że dla wszystkich  $n > n_0$ ,  $f(n) \leq c g(n)$ ;
- $f = \Omega(g)$  jeśli istnieje  $n_0 \in \mathbb{N}$  i stała  $c$  taka, że dla wszystkich  $n > n_0$ ,  $f(n) \geq c g(n)$ ;
- $f = \Theta(g)$  jeśli  $f = O(g)$  i  $f = \Omega(g)$

## PROBLEM KOMIWOJAŻERA

Dane:

- $V$  - zbiór wierzchołków
- $c: V \times V \rightarrow \mathbb{N}$  - macierz kosztu

Znajdź ciąg  $v_1, \dots, v_n$  z  $V$ , dla których całkowity koszt:

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

był jak najmniejszy

## ZŁOŻONOŚĆ P.K.

Algorytm:

Sprawdź wszystkie możliwe ciągi w  $V$  i znajdź ten o najniższym koszcie

Ograniczenie dolne:

Analiza taka jak dla sortowania wykazuje ograniczenie  $\Omega(n \log n)$

Ograniczenie górne:

Aktualnie najszybszy znany algorytm posiada złożoność  $O(n^2 2^n)$

## ALGORYTMY - FORMALNIE

W celu wykazania ograniczenia dolnego dla danego problemu, a nie dla konkretnego algorytmu, musimy udowodnić stwierdzenie dla **wszystkich** algorytmów rozwiązujących ten problem

W celu udowodnienia faktów dot. algorytmów potrzebujemy dokładnej matematycznej definicji algorytmu

Wykorzystamy do tego **Maszynę Turinga**

## MASZYNA TURINGA



opracowane na podstawie slajdów Dr. C. Busch'a

## MASZYNA TURINGA

Uproszczenie w postaci Maszyny Turinga nie pozwala na reprezentację algorytmów, ale jest bardzo dobrym narzędziem do prowadzenia dowodów dla wszystkich algorytmów.

## JĘZYKI REGULARNE

Co to jest język?

Język jest zbiorem łańcuchów znakowych na przestrzeni pewnego zbioru  $\Sigma$  znaków/symboli zwanych alfabetem

łańcuch/wyraz: Sekwencja liter/symboli

Przykład: „kot”, „pies”, „dom”, ...

łańcuchy na przestrzeni alfabetu:

$$\Sigma = \{a, b, c, \dots, z\}$$

## ALFABET I ŁAŃCUCHY ZNAKÓW

będziemy wykorzystywali symbole alfabetu (litery) pisane małą literą

Dla danego języka  $\Sigma^* = \{a, b\}$  mamy następujące łańcuchy:

a

ab

$u = ab$

abba

$v = bbbaaa$

baba

$w = abba$

aaabbbaabbab

## JĘZYK REGULARNY

- Założmy, że  $L \subseteq \Sigma^*$  jest dowolnym językiem
  - słowo  $u \in \Sigma^*$  rozróżnia dwa słowa  $v$  i  $w \in \Sigma^*$  względem języka  $L$  jeżeli jedno ze słów  $u, v, w$  należy do  $L$  a drugie nie.
  - nie istnieje słowo  $u$ , które rozróżnia słowa  $v$  i  $w$ ,
  - słowa są nieroróżnialne

opracowane częściowo na podstawie materiałów Dr. T. Szymorczuka

## JĘZYK REGULARNY

Definicja

- według Myhill-Nerode:
  - Język  $L$  jest językiem regularnym  $\Leftrightarrow$  zbiór  $\Sigma^*$  ma skończenie wiele klas nieroróżnialnych słów.
- według Wikipedii:
  - Język regularny (ang. regular language) to język formalny taki, że istnieje automat o skończonej liczbie stanów potrafiący zdecydować, czy dane słowo należy do języka. Również, taki, że istnieje dlań gramatyka regularna.

opracowane częściowo na podstawie materiałów Dr. T. Szymorczuka

## PRZYKŁADY JĘZYKÓW REGULARNYCH

zbiór wszystkich słów alfabetu {0, 1}

zbiór wszystkich słów alfabetu {0, 1} o długości n

zbiór wszystkich słów alfabetu {0, 1} o parzystej długości

zbiór wszystkich słów alfabetu {0, 1} zaczynających się od zera

zbiór wszystkich słów alfabetu {0, 1} nie zaczynających się od zera

zbiór wszystkich słów alfabetu {0, 1} w których na przemian występują zera i jedynki

zbiór wszystkich słów alfabetu {0, 1} w których na przemian występują zera, jedynki i dwójki

Zbiór wszystkich języków regularnych oznacza się przez REG.

w/g Wikipedii

## GRAMATYKA REGULARNA

Gramatyka:

- systematyczny opis wybranego języka naturalnego
  - składnia - syntaktyka
    - reguły określają regularności rządzące kombinacjami słów
  - znaczenie - semantyka
    - bada znaczenie słów i zdań
  - fonologia - dźwiękowy system języka
    - wyróżnia dźwięki i ich dopuszczalne zestawienia w opisywanym języku

## DLACZEGO SIĘ O TYM UCZYMY?

- Teoria języków formalnych tworzy pewne modele obliczeń
  - systemy opisu języków
    - gramatyki i automaty
  - systemy mają spełniać warunki efektywności
    - analitycznej
      - prowadzi do algorytmu, który w skończonej liczbie kroków rozstrzyga, czy dowolne słowo należy do danego języka, czy nie
    - syntetycznej
      - daje w rezultacie algorytm, który umożliwia wygenerowanie wszystkich słów danego języka.

## DLACZEGO SIĘ O TYM UCZYMY?

- Idee Chomsky'ego przeniknęły do innych dziedzin nauki.
- Jego teoria znalazła istotne zastosowanie w badaniach nad językami programowania.
- Gramatyki Chomsky'ego służą również do budowania modeli procesów biologicznych, czy też procesów badanych przez nauki o społeczeństwie.
- Teoria gramatyk rozwinęła się w wielu kierunkach, służąc jako formalny opis sekwencyjnych zmian różnorakich obiektów, takich jak, termy, grafy, obrazy, czy fraktały.

## ŁAŃCUCHY ZNAKÓW

- Inaczej łańcuch, słowo lub string
  - jest dowolną skońzoną sekwencją znaków nad alfabetem  $\Sigma$
  - jeśli  $\Sigma = \{0, 1\}$ , to łańcuch 0101 jest słowem nad alfabetem  $\Sigma$

a	u = ab
ab	
abba	v = bbbaaa
baba	
aaabbbaabbab	w = abba



## TEORIA ZŁOŻONOŚCI



**Teoria złożoności** poszukuje rozwiązań dla problemów, które są algorytmicznie trudne do rozwiązania

W części poświęconej **Algorytmom i Strukturom Danych** dowiedzieliśmy się w jaki sposób mierzyć/oszacowywać złożoność konkretnych algorytmów poprzez asymptotyczny pomiar ilości kroków

W dziedzinie **teorii złożoności** możemy mieć do czynienia z problemami, które nie będą rozwiązywalne algorytmicznie.

## PROBLEMY I ALGORYTMY

Sortowanie przez wstawianie działa w czasie  $O(n^2)$ , a Sortowanie przez Scalanie jest algorymem o złożoności  $O(n \log n)$ .

Pierwsza część zdania oznacza:

Jeśli policzymy ilość kroków wykonywanych przez algorytm sort. przez wstawianie dla wejścia o rozmiarze  $n$ , biorąc największą liczbę spośród wszystkich wejść o takim rozmiarze, to tak zdefiniowana funkcja  $n$  będzie ograniczona przez stałą wielokrotność  $n^2$ .

Sensownym jest porównywanie tych dwóch algorytmów - poszukując rozwiązań tego samego problemu

ale...

jaka jest złożoność **problemu sortowania???**

## OGRANICZENIE DOLNE I GÓRNE

Jaka jest złożoność czasowa najszybszego algorytmu sortującego listę?

Przez analizę algorytmu sort. przez scalanie wiemy, że nie jest gorsza niż  **$O(n \log n)$** .

Złożoność konkretnego algorytmu wyznacza **górne ograniczenie** złożoności problemu

Do wyznaczenia **ograniczenia dolnego** potrzebujemy wykazać, że nie istnieje algorytm (nawet taki, który nie został jeszcze wymyślony), który może być lepszy.

W przypadku sortowania możemy wyznaczyć ograniczenie dolne jako  $\Omega(n \log n)$ . Sortowanie jest rzadkim przypadkiem, gdzie znane ograniczenia dolne i górne są takie same.

## PRZYPOMNIENIE

Złożoność algorytmu (bez znaczenia, czy czasowa, czy pamięciowa) jest przeważnie opisywana asymptotycznie:

Definicja:

Dla funkcji  $f: \mathbb{N} \rightarrow \mathbb{N}$  i  $g: \mathbb{N} \rightarrow \mathbb{N}$  to powiemy, że

- $f = O(g)$  jeśli istnieje  $n_0 \in \mathbb{N}$  i stała  $c$  taka, że dla wszystkich  $n > n_0$ ,  $f(n) \leq c g(n)$ ;
- $f = \Omega(g)$  jeśli istnieje  $n_0 \in \mathbb{N}$  i stała  $c$  taka, że dla wszystkich  $n > n_0$ ,  $f(n) \geq c g(n)$ ;
- $f = \Theta(g)$  jeśli  $f = O(g)$  i  $f = \Omega(g)$

## PROBLEM KOMIWOJAŻERA

Dane:

•  $V$  - zbiór wierzchołków

•  $c: V \times V \rightarrow \mathbb{N}$  - macierz kosztu

Znajdź ciąg  $v_1, \dots, v_n$  z  $V$ , dla których całkowity koszt:

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

był jak najmniejszy

## ZŁOŻONOŚĆ P.K.

Algorytm:

Sprawdź wszystkie możliwe ciągi wV i znajdź ten o najniższym koszcie

Ograniczenie dolne:

Analiza taka jak dla sortowania wykazuje ograniczenie  $\Omega(n \log n)$

Ograniczenie górne:

Aktualnie najszybszy znany algorytm posiada złożoność  $O(n^2 2^n)$

## ALGORYTMY - FORMALNIE

W celu wykazania ograniczenia dolnego dla danego problemu, a nie dla konkretnego algorytmu, musimy udowodnić stwierdzenie dla **wszystkich** algorytmów rozwiązujących ten problem

W celu udowodnienia faktów dot. algorytmów potrzebujemy dokładnej matematycznej definicji algorytmu

Wykorzystamy do tego **Maszynę Turinga**

## MASZYNA TURINGA



opracowane na podstawie slajdów Dr. C. Busch'a

## MASZYNA TURINGA

Uproszczenie w postaci Maszyny Turinga nie pozwala na reprezentację algorytmów, ale jest bardzo dobrym narzędziem do przeprowadzania dowodów dla wszystkich algorytmów.

## JĘZYKI REGULARNE

Co to jest język?

Język jest zbiorem łańcuchów znakowych na przestrzeni pewnego zbioru  $\Sigma$  znaków/symboli zwanych alfabetem

łańcuch/wyraz: Sekwencja liter/symboli

Przykład: „kot”, „pies”, „dom”, ...

łańcuchy na przestrzeni alfabetu:

$\Sigma = \{a, b, c, \dots, z\}$

## ALFABET I ŁAŃCUCHY ZNAKÓW

będziemy wykorzystywali symbole alfabetu (litery) pisane małą literą

Dla danego języka  $\Sigma^* = \{a, b\}$  mamy następujące łańcuchy:

a

ab

abba

baba

$u = ab$

$v = bbbaaa$

$w = abba$

$aaabbbaabbab$

## JĘZYK REGULARNY

- Założymy, że  $L \subseteq \Sigma^*$  jest dowolnym językiem
  - słowo  $u \in \Sigma^*$  rozróżnia dwa słowa  $v$  i  $w \in \Sigma^*$  względem języka  $L$  jeżeli jedno ze słów  $u, v, w$  należy do  $L$  a drugie nie.
  - nie istnieje słowo  $u$ , które rozróżnia słowa  $v$  i  $w$ ,
  - słowa są nieroróżnialne

opracowane częściowo na podstawie materiałów Dr. T. Szymoricka

## JĘZYK REGULARNY

### Definicja

- według Myhill-Nerode:
  - Język  $L$  jest językiem regularnym  $\Leftrightarrow$  zbiór  $\Sigma^*$  ma skończenie wiele klas nieroróżnialnych słów.
- według Wikipedii:
  - Język regularny (ang. regular language) to język formalny taki, że istnieje automat o skończonej liczbie stanów potrafiący zdecydować, czy dane słowo należy do języka. Również, taki, że istnieje dlań gramatyka regularna.

opracowane częściowo na podstawie materiałów Dr. T. Szymoricka

## PRZYKŁADY JĘZYKÓW REGULARNYCH

zbiór wszystkich słów alfabetu  $\{0, 1\}$

zbiór wszystkich słów alfabetu  $\{0, 1\}$  o długości  $n$

zbiór wszystkich słów alfabetu  $\{0, 1\}$  o parzystej długości

zbiór wszystkich słów alfabetu  $\{0, 1\}$  zaczynających się od zera

zbiór wszystkich słów alfabetu  $\{0, 1\}$  nie zaczynających się od zera

zbiór wszystkich słów alfabetu  $\{0, 1\}$  w których na przemian występują zera i jedynki

zbiór wszystkich słów alfabetu  $\{0, 1\}$  w których na przemian występują zera, jedynki i dwójki

Zbiór wszystkich języków regularnych oznacza się przez REG.

w/g Wikipedia

## GRAMATYKA REGULARNA

### Gramatyka:

- systematyczny opis wybranego języka naturalnego
  - składnia - syntaktyka
    - reguły określają regularności rządzące kombinacjami słów
  - znaczenie - semantyka
    - bada znaczenie słów i zdań
  - fonologia - dźwiękowy system języka
    - wyróżnia dźwięki i ich dopuszczalne zestawienia w opisywanym języku

## DLACZEGO SIĘ O TYM UCZYMY?

- Teoria języków formalnych tworzy pewne modele obliczeń
  - systemy opisu języków
    - gramatyki i automaty
  - systemy mają spełniać warunki efektywności
    - analitycznej
      - prowadzi do algorytmu, który w skończonej liczbie kroków rozstrzyga, czy dowolne słowo należy do danego języka, czy nie
    - syntetycznej
      - daje w rezultacie algorytm, który umożliwia wygenerowanie wszystkich słów danego języka.

## DLACZEGO SIĘ O TYM UCZYMY?

- Idee Chomsky'ego przeniknęły do innych dziedzin nauki.
- Jego teoria znalazła istotne zastosowanie w badaniach nad językami programowania.
- Gramatyki Chomsky'ego służą również do budowania modeli procesów biologicznych, czy też procesów badanych przez nauki o społeczeństwie.
- Teoria gramatyk rozwinięła się w wielu kierunkach, służyąc jako formalny opis sekwencyjnych zmian różnorakich obiektów, takich jak, terminy, grafy, obrazy, czy fraktale.

# ŁAŃCUCHY ZNAKÓW

- Inaczej łańcuch, słowo lub string
  - jest dowolną skońzoną sekwencją znaków nad alfabetem  $\Sigma$
  - jeśli  $\Sigma = \{0, 1\}$ , to łańcuch 0101 jest słowem nad alfabetem  $\Sigma$

a	u = ab
ab	
abba	v = bbbaaa
baba	
aaabbbaabbab	w = abba

# WYBRANE OPERACJE NA ŁAŃCUCHACH

## OPERACJE NA ŁAŃCUCHACH

$w = a_1a_2\dots a_n$	abba
$v = b_1b_2\dots b_m$	bbbaaa
<b>Konkatenacja</b>	
$wv = a_1a_2\dots a_n b_1b_2\dots b_m$	abbabbbaaa
$w = a_1a_2\dots a_n$	ababaaabbb
<b>Odwrotność</b>	
$w^R = a_n \dots a_2a_1$	bbbbaababa

## DŁUGOŚĆ ŁAŃCUCHA I KONKATENACJI

$w = a_1a_2\dots a_n$	
<b>Długość:</b>	$ w  = n$
<b>Przykłady:</b>	$ abba  = 4$ $ aa  = 2$
$ uv  =  u  +  v $	
<b>Przykłady:</b>	
$u = aab,  u  = 3$	
$v = abaab,  v  = 5$	
$ uv  =  aababaab  = 8$	
$ uv  =  u  +  v  = 3 + 5 = 8$	

## PUSTY ŁAŃCUCH

Pustym łańcuchem nazywamy słowo nie zawierające ani jednej litery, oznaczany przez  $\lambda$

### Obserwacje:

$$|\lambda| = 0$$

$$\lambda w = w\lambda = w$$

$$\lambda abba = abba\lambda = abba$$

## OPERACJA POWIELENIA

$$w^n = \underbrace{ww\dots w}_n$$

$$\text{Przykład: } (abba)^2 = abbaabba$$

$$\text{Definicja: } w^0 = \lambda$$

$$(abba)^0 = \lambda$$

## OPERACJA \*

$\Sigma^*$  : jest zbiorem wszystkich możliwych słów nad alfabetem  $\Sigma$

$$\Sigma = \{a, b\}$$

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

## OPERACJA +

$\Sigma^+$ : jest zbiorem wszystkich możliwych słów nad alfabetem  $\Sigma$  z wyłączeniem  $\lambda$

$$\Sigma = \{a, b\}$$

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

## UWAGA!!!

Zbiór pusty:  $\emptyset$

Pusty łańcuch:  $\lambda$        $\emptyset = \{\} \neq \lambda$

Rozmiar zbioru:       $|\{\}| = |\emptyset| = 0$

Rozmiar zbioru:       $|\{\lambda\}| = 1$

Długość łańcucha:       $|\lambda| = 0$

## OPERACJE NA JĘZYKACH

Typowe operacje na zbiorach:

$$\{a, ab, aaaa\} \cup \{bb, ab\} = \{a, ab, bb, aaaa\}$$

$$\{a, ab, aaaa\} \cap \{bb, ab\} = \{ab\}$$

$$\{a, ab, aaaa\} - \{bb, ab\} = \{a, aaaa\}$$

Dopełnienie:       $L = \Sigma^* - L$

$$\overline{\{a, ba\}} = \{\lambda, b, aa, ab, bb, aaa, \dots\}$$

## OPERACJE NA JĘZYKACH

Odwrotność

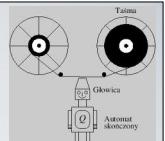
Definicja:  $L^R = \{w^R : w \in L\}$

Przykłady:  $\{ab, aab, baba\}^R = \{ba, baa, abab\}$

$$L = \{a^n b^n : n \geq 0\}$$

$$L^R = \{b^n a^n : n \geq 0\}$$

## AUTOMATY



Są drugim, obok gramatyki, modelem obliczeń będącym przedmiotem badań nad teorią języków i automatów

model realizujący warunek efektywności analitycznej

na jego podstawie możliwe jest sformułowanie algorytmu rozstrzygającego w skończonej liczbie kroków, czy dowolne słowo należy, czy też nie należy do języka rozpoznawanego przez ten automat

taki automat, który daje algorytm efektywnie rozstrzygający, czy dowolne obliczenie sformułowane nad alfabetem automatu jest poprawne

# AUTOMAT SKOŃCZENIE STANOWY

jest jednym z najprostszych modeli obliczeń



model z bardzo istotnie ograniczoną pamięcią

działanie sprowadza się do zmiany stanu pod wpływem określonego zewnętrznego sygnału czy impulsu

Urządzenia oparte o taki model automatów są często spotykane  
drzwi

automaty do napoi

winda

urządzenia sterujące taśmą produkcyjną



## PRZYKŁAD

Automatycznie otwierane drzwi sterowane automatem

działania automatu opisane jako:

WE - symbol odpowiadający sytuacji, w której osoba chce wejść do pomieszczenia zamkнутego przez takie drzwi

identyfikacja przez odpowiedni czujnik

WY - symbol opisujący zamiar wyjścia z pomieszczenia

WEWY - symbol związany z równoczesnym zamiarem wejścia jakiejś osoby i wyjścia innej

BRAK - oznacza brak osób, które chcą wejść lub wyjść

Zatem

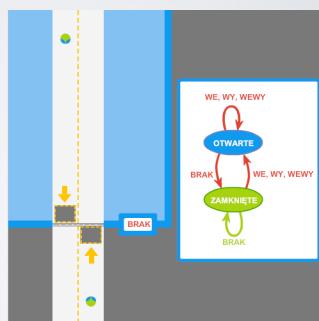
$$\Sigma = \{WE, WY, WEWY, BRAK\}$$

alfabet nad którym określmy automat o stanach:

OTWARTE

ZAMKNIĘTE

## PRZYKŁAD

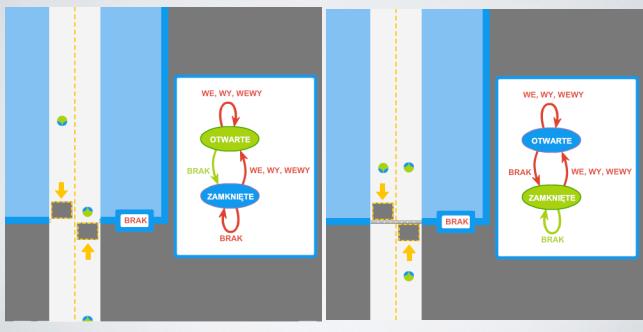


## PRZYKŁAD

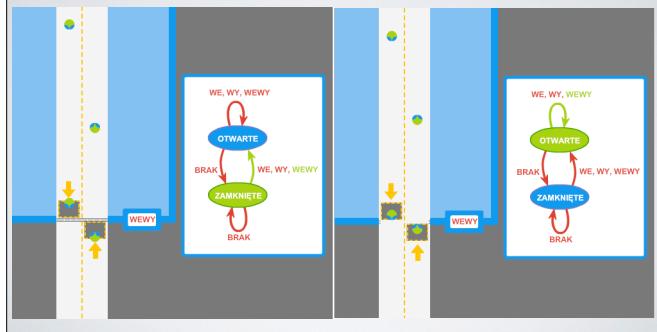


Animacja na stronie: [http://wazniak.mimuw.edu.pl/Wyklad\\_3\\_Automat\\_skoczenie\\_stanowy](http://wazniak.mimuw.edu.pl/Wyklad_3_Automat_skoczenie_stanowy)

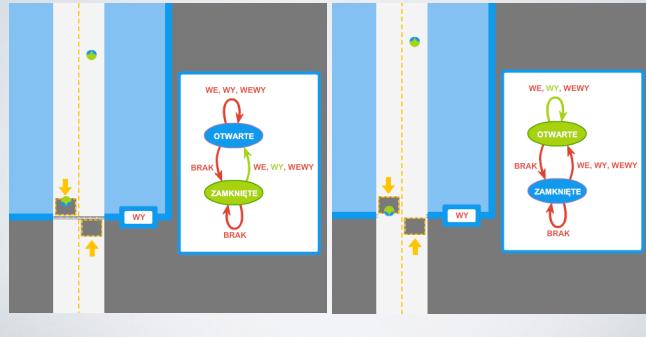
## PRZYKŁAD



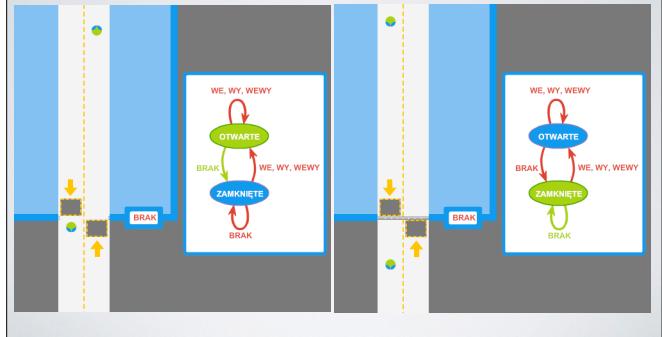
## PRZYKŁAD



## PRZYKŁAD



## PRZYKŁAD



## WNIOSKI

Automaty reagują na określone sygnały zewnętrzne zmieniając swój stan reprezentowane przez litery alfabetu  $\Sigma$

Jeśli ustalimy stan początkowy automatu oraz dopuszczalne stany końcowe automat będzie testował dowolne słowo z  $\Sigma^*$  rozpoczynając od stanu początkowego

Jeśli końcowym rezultatem działania automatu będzie stan końcowy słowo rozpoznawane przez automat obliczenia określone tym słowem są poprawne

Automaty można graficznie reprezentować jako etykietowane grafy skierowane wierzchołki odpowiadają stanowi automatu strzałki między wierzchołkami wraz z etykietami oznaczają przejście automatu z jednego stanu do innego pod wpływem litery

## JĘZYKI BEZKONTEKSTOWE

### Twierdzenie

$$\left\{ \begin{array}{l} \text{Języki} \\ \text{bezkontekstowe} \\ (\text{Gramatyki}) \end{array} \right\} = \left\{ \begin{array}{l} \text{Języki} \\ \text{akceptowane przez} \\ \text{automaty skończone} \\ \text{ze stosem} \end{array} \right\}$$

## DOWÓD - KROK 1

$$\left\{ \begin{array}{l} \text{Języki} \\ \text{bezkontekstowe} \\ (\text{Gramatyki}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Języki} \\ \text{akceptowane przez} \\ \text{automaty skończone} \\ \text{ze stosem} \end{array} \right\}$$

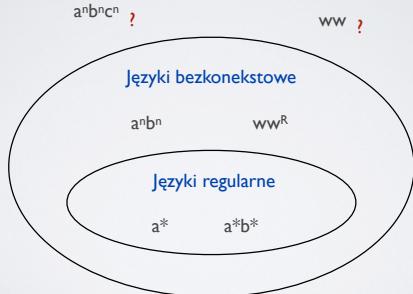
Należy zamienić dowolną gramatykę bezkontekstową  $G$  na automat skończony ze stosem  $M$  gdzie  $L(G) = L(M)$

## DOWÓD - KROK 2

$$\left\{ \begin{array}{l} \text{Języki} \\ \text{bezkontekstowe} \\ (\text{Gramatyki}) \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Języki} \\ \text{akceptowane przez} \\ \text{automaty skończone} \\ \text{ze stosem} \end{array} \right\}$$

Należy zamienić dowolny automat skończony  $M$  ze stosem na gramatykę bezkontekstową  $G$  gdzie  $L(G) = L(M)$

## HIERARCHIA JĘZYKÓW

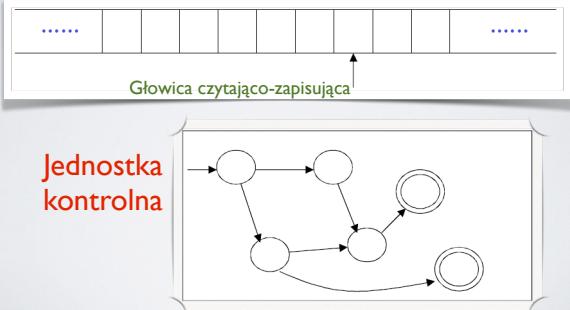


## HIERARCHIA JĘZYKÓW



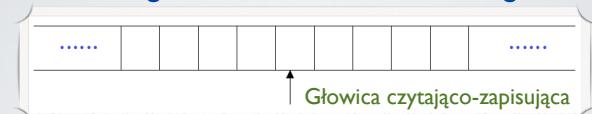
## MASZYNA TURINGA

Taśma



## TAŚMA

Brak ograniczeń - nieskończona długość



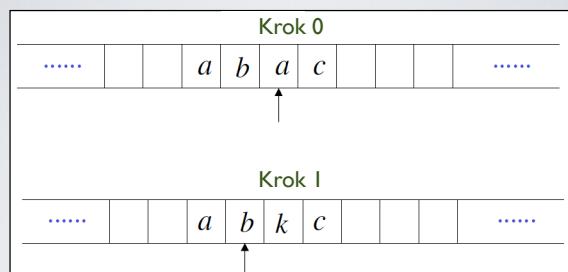
Główica porusza się w prawo lub w lewo



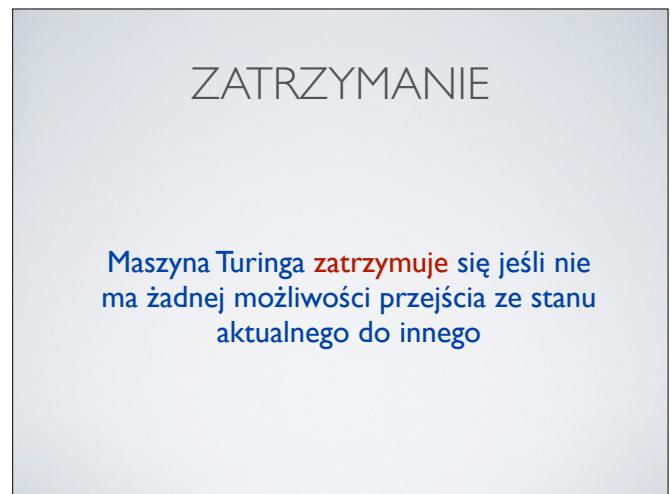
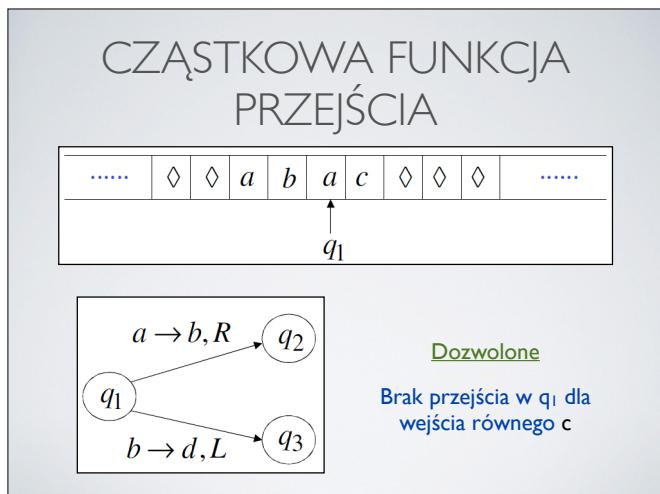
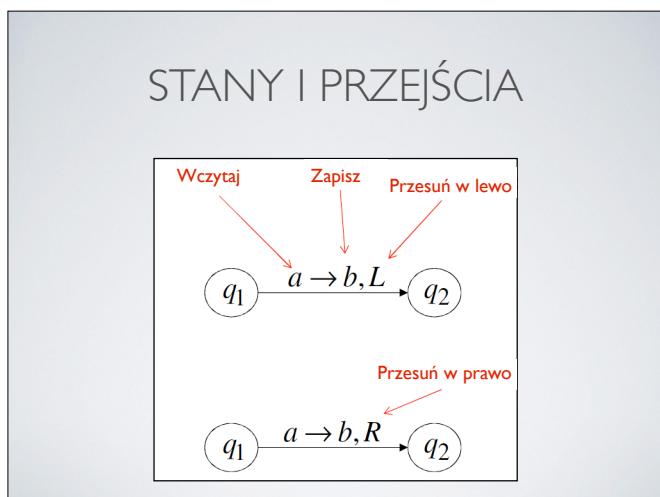
Przy każdym kroku głowica wykonuje następujące czynności:

1. Wczytuje znak
2. Zapisuje znak
3. Przesuwa się w lewo lub w prawo

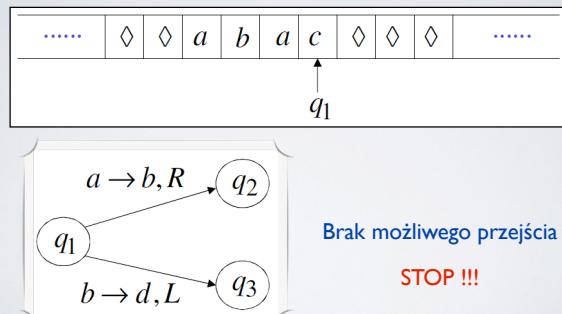
## PRZYKŁAD



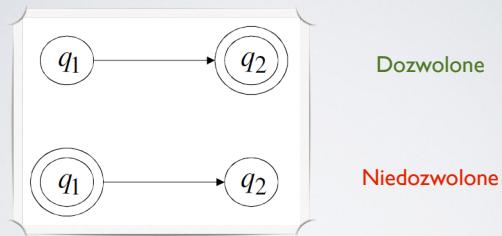
1. Wczytuje a
2. Zapisuje k
3. Przesuwa się w lewo



## PRZYKŁAD



## STANY KOŃCOWE



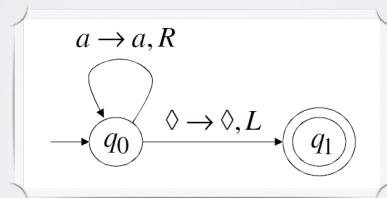
- Stany końcowe nie posiadają przejść wychodzących
- W stanie końcowym maszyna się zatrzymuje

## AKCEPTOWALNOŚĆ



## MASZYNA TURINGA - PRZYKŁAD

Maszyna Turinga, która akceptuje język:  $a^*$



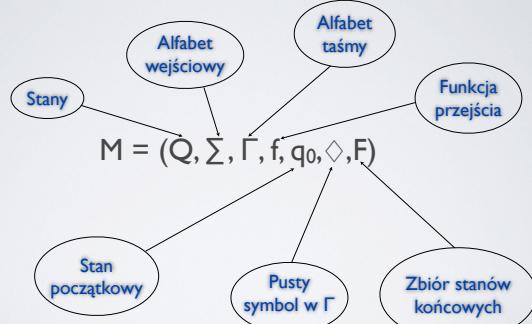
## FORMALNA DEFINICJA MASZYNY TURINGA

## FUNKCJA PRZEJŚCIA

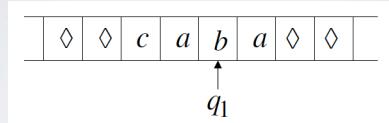
$$(q_1) \xrightarrow{a \rightarrow b, R} (q_2) \quad f(q_1, a) = (q_2, b, R)$$

$$(q_1) \xrightarrow{c \rightarrow d, L} (q_2) \quad f(q_1, c) = (q_2, d, L)$$

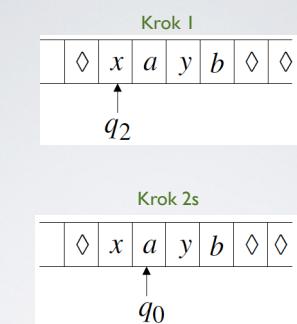
## MASZYNA TURINGA



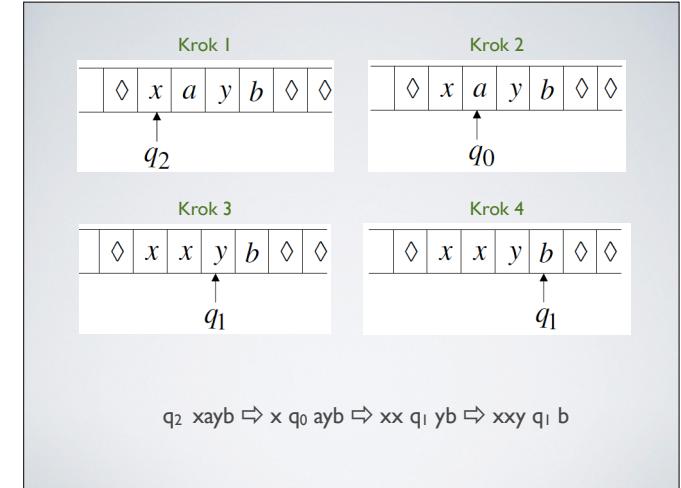
## KONFIGURACJA



Opis chwilowy: ca q<sub>1</sub> ba



Przejście:  $q_2 \ xayb \Rightarrow x \ q_0 \ ayb$



$q_2 \ xayb \Rightarrow x \ q_0 \ ayb \Rightarrow xx \ q_1 \ yb \Rightarrow xxy \ q_1 \ b$

## AKCEPTOWANY JĘZYK

Dla dowolnej maszyny Turinga  $M$

$$L(M) = \{w: q_0 \xrightarrow{*} x_1 q_f x_2\}$$

↓ Stan początkowy                    ↓ Stan końcowy

## STANDARDOWA MASZYNA TURINGA

Maszyna, którą opisaliśmy jest standardową maszyną Turinga:

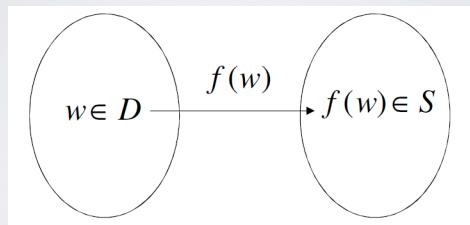
- Deterministyczna
- Nieskończona taśma w obu kierunkach
- Taśma stanowi plik wejścia/wyjścia

## OBLCZANIE FUNKCJI ZA POMOCĄ MASZYN TURINGA

Funkcja  $f(w)$  posiada:

Dziedzina:  $D$

Przestrzeń rozwiązań:  $S$



Funkcja może posiadać wiele parametrów/argumentów:

Przykład: Funkcja sumowania

$$f(x, y) = x + y$$

## DOMENA INTEGERÓW

Dziesiętnie: 5

Binarne: 101

Unarne: 11111

Preferowany jest zapis unarny:

łatwiejszy w operacjach z Maszyną Turinga

### Definicja

Funkcja  $f$  jest obliczalna jeśli istnieje Maszyna Turinga  $M$  taka, że:

Konfiguracja początkowa

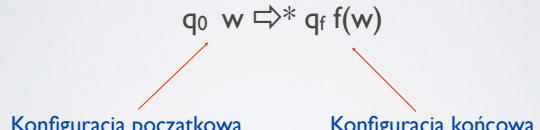
Konfiguracja końcowa



Dla wszystkich  $w \in D$

### Innymi słowy

Funkcja  $f$  jest obliczalna jeśli istnieje Maszyna Turinga  $M$  taka, że:



$\forall w \in D$

## PRZYKŁAD

Funkcja  $f(x, y) = x + y$

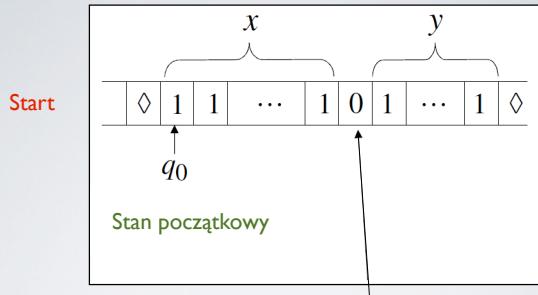
gdzie  $x, y$  są liczbami całkowitymi

**Jest obliczalna**

Maszyna Turinga M dla f:

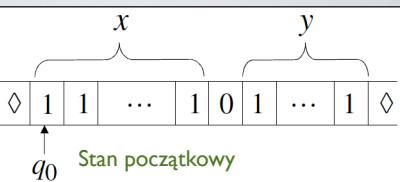
Łańcuch wejściowy:  $x0y$  unarnie

Łańcuch wyjściowy:  $xy0$  unarnie

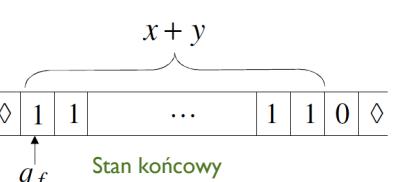


0 jest wartością (delimiterem) oddzielającą wejścia x i y

Start

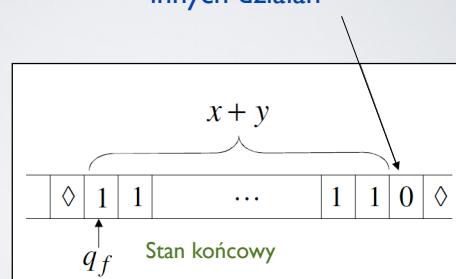


Koniec



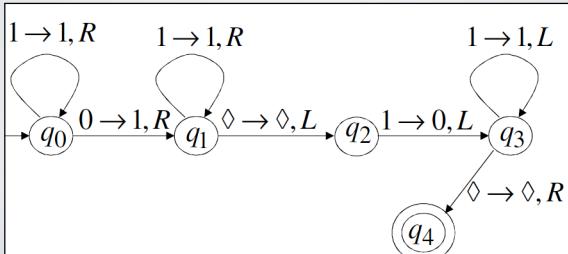
0 może pomóc jeśli będziemy chcieli wykorzystać wynik do innych działań

Koniec



Maszyna Turinga M dla f:

$$f(x, y) = x + y$$

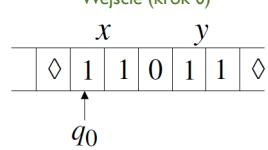


Przykład działania

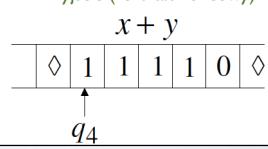
$$x = 11 \quad (2)$$

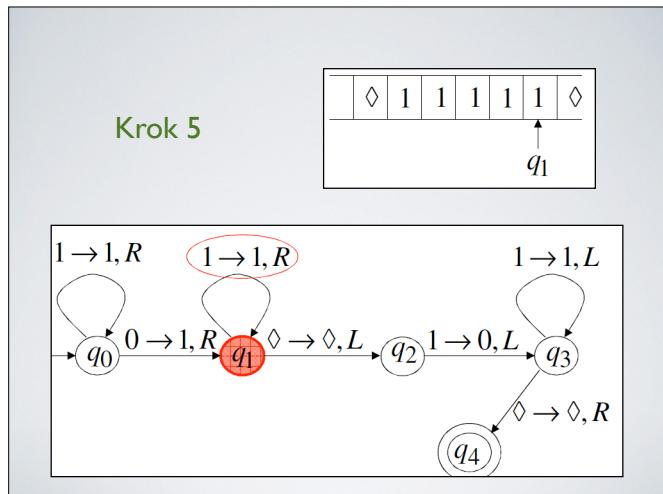
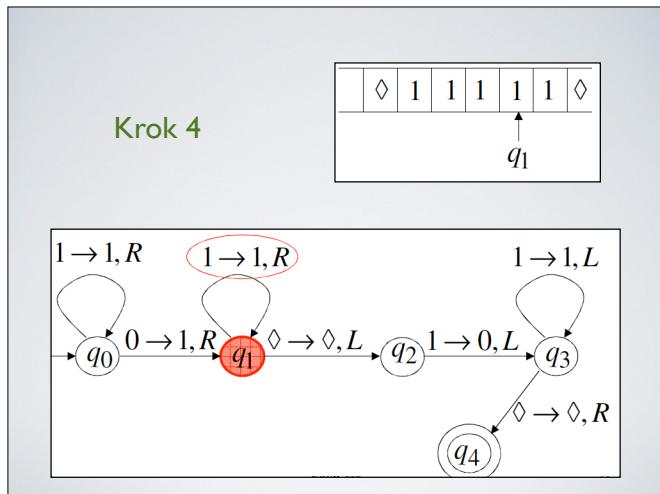
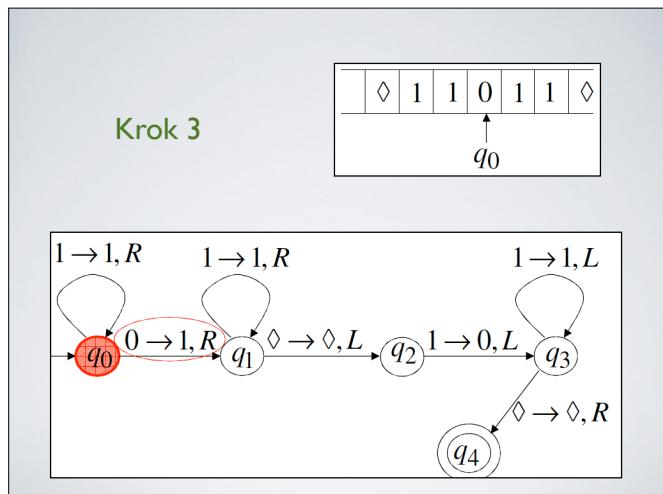
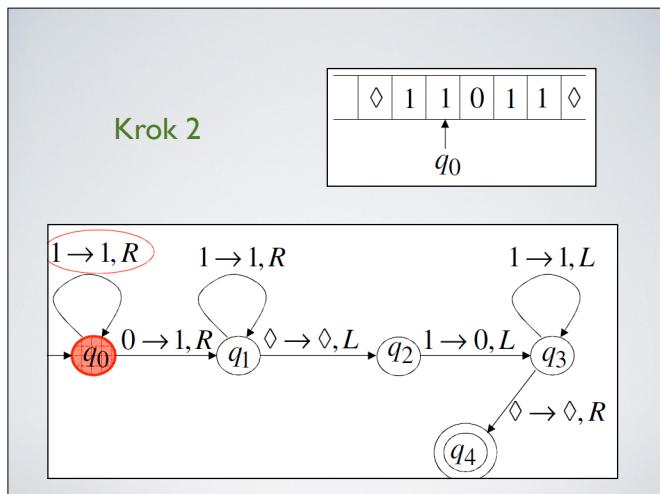
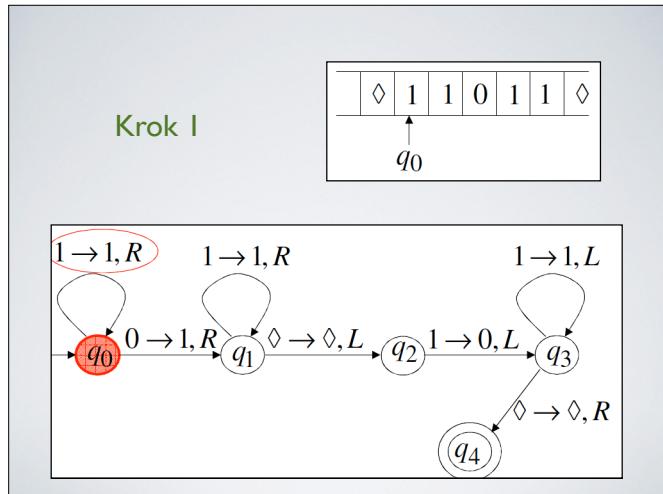
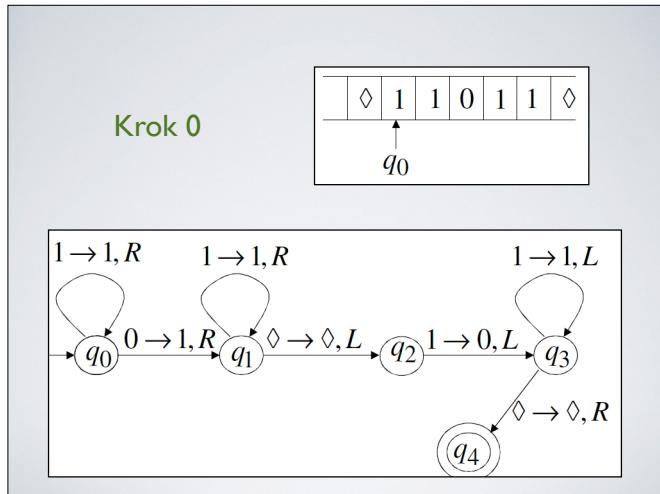
$$y = 11 \quad (2)$$

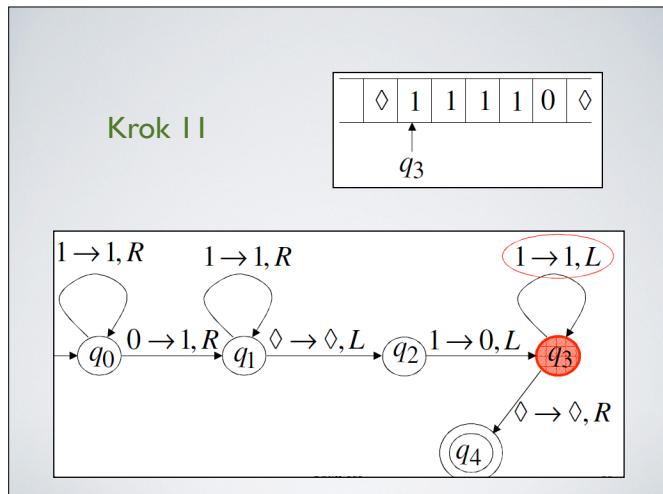
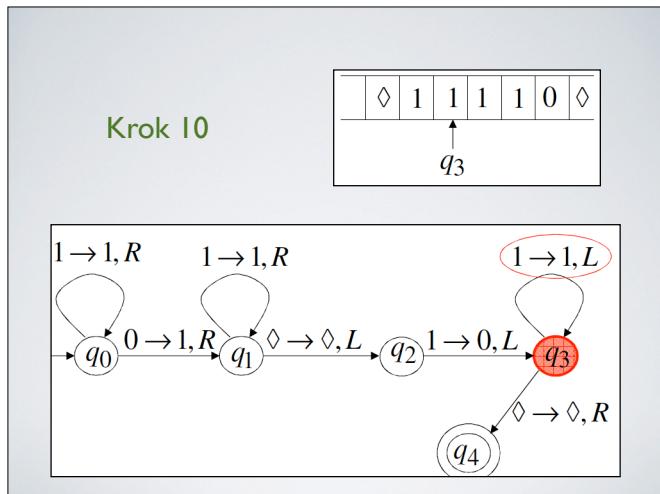
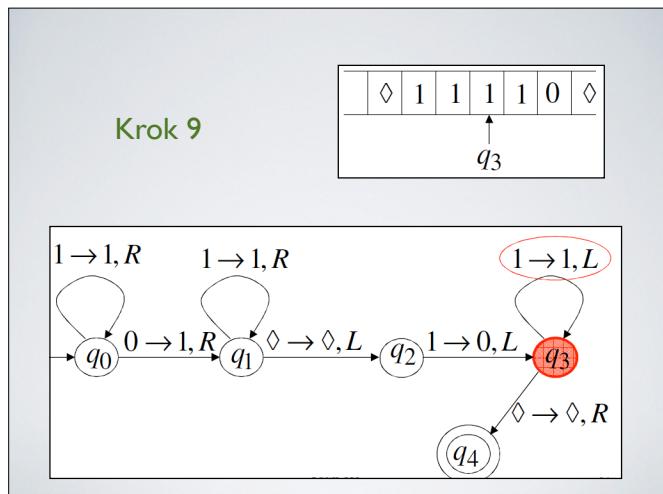
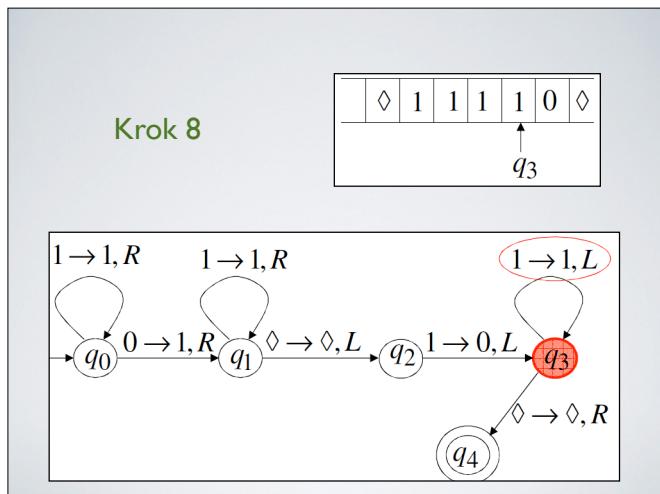
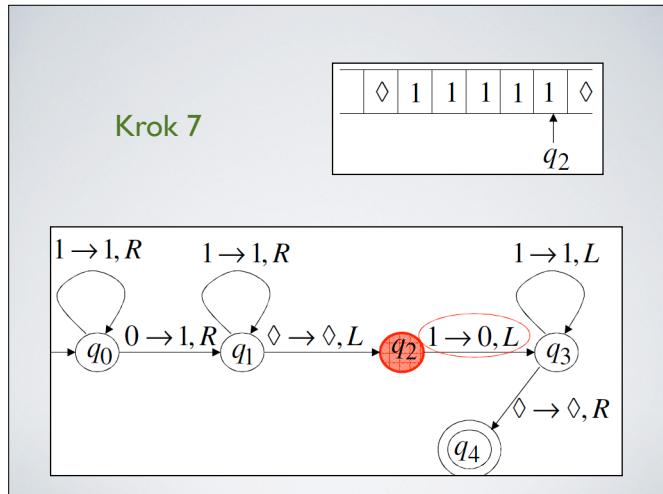
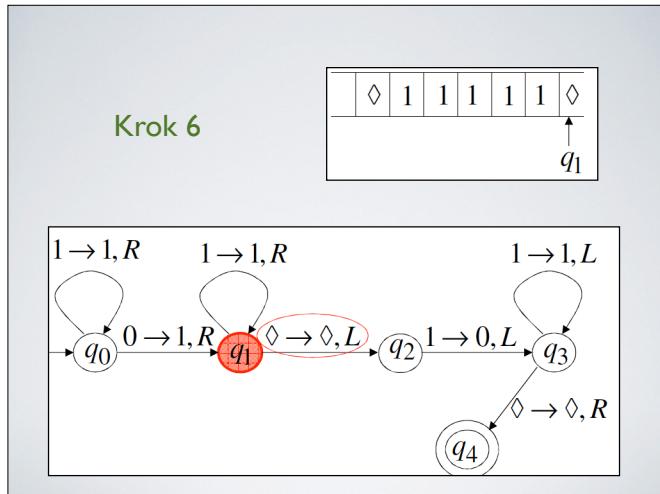
Wejście (krok 0)



Wyjście (rezultat końcowy)

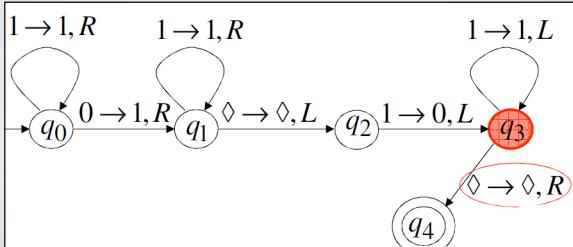






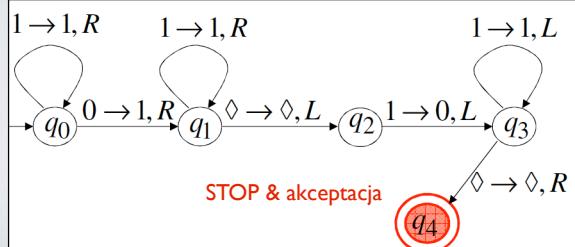
### Krok 12

$\diamond$	1	1	1	1	0	$\diamond$
$q_3$						



### Krok 13

$\diamond$	1	1	1	1	0	$\diamond$
$q_4$						

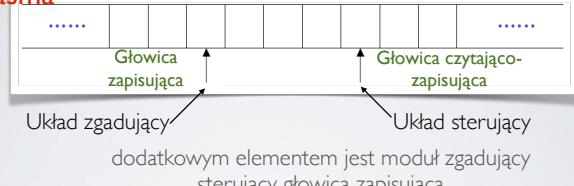


## MASZYNA TURINGA - WNIOSKI

- Pozwala rozwiązywać problemy decyzyjne
  - model maszyny uniwersalnej
- Jeśli czas ( $t$ ) działania algorytmu A jest ograniczony wielomianem ( $t \leq p(|L|)$ ), to powiemy, że jest on **algorytmem wielomianowym**
  - $t \leq p(|L|)$  - wielomian zależny od długości języka
- w przeciwnym wypadku algorytm będzie **algorytmem ponadwielomianowym**.

## NIEDETERMINISTYCZNA MASZYNA TURINGA

### Taśma



Moduł zgadujący odpowiada za zapisanie odgadniętego rozwiązania

- wykorzystywany tylko do tego
- np.: ścieżka w P.K., upakowanie plecaka, itp.

## NIEDETERMINISTYCZNA MASZYNA TURINGA - WNIOSKI

- Dla ustalonego  $L$  niedeterministyczna maszyna Turinga może wyznaczyć inne odpowiedzi
  - inne łańcuchy wyjściowe
  - może istnieć nieskończenie wiele procesów obliczeń dla  $L$
- Powiemy, że algorytm A na niedeterministycznej MT będzie akceptował język  $L$  jeśli przynajmniej jeden z tych procesów obliczeń dla  $L$  jest akceptowałny
- Niedeterministyczna maszyna Turinga jest nierealistycznym modelem komputera

## KLASY PROBLEMÓW DECYZYJNYCH

- Klasa problemów P (polynomial-time)
  - (Oryginalna definicja)
    - Problemy, które mogą zostać rozwiązane przez deterministyczną maszynę Turinga w czasie wielomianowym.
  - (Definicja równoważna)
    - Problemy, które są rozwiązywalne w czasie wielomianowym.
- Klasa problemów NP (non-deterministic polynomial-time)
  - (Oryginalna definicja)
    - Problemy, które mogą zostać rozwiązane przez niedeterministyczną maszynę Turinga w czasie wielomianowym.
  - (Definicja równoważna)
    - Problemy, które są weryfikowalne w czasie wielomianowym.
      - Dla danego rozwiązania istnieje algorytm wielomianowy sprawdzający, czy to rozwiązanie jest poprawne.

## PROBLEMY P VS. NP

Są dzisiaj chyba jednym z największych otwartych problemów w naukach informatycznych (i matematycznych!).

Doczekały się nawet roli w serialu :) - „Wzór” (NUMB3RS)

No dobra, ale co to są te problemy P-NP???

## CZAS WIELOMIANOWY

- Czas wielomianowy: złożoność czasowa to  $O(n^k)$  - k jest stałą
- Czy poniższe przykłady mają złożoność wielomianową?
  - $T(n) = 3$  tak
  - $T(n) = n$  tak
  - $T(n) = n \log(n)$  tak
  - $T(n) = 5^n$  nie
  - $T(n) = n!$  nie

## PROBLEMY P VS. NP

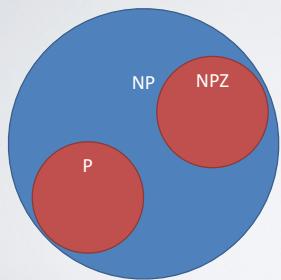
- Weryfikacja wielomianowa może zostać wykorzystana do sprawdzenia, czy problem jest NP
- n.p:
  - Sortowanie, n-integerów
    - Kandydat: tablica
    - Weryfikacja: przeskanuj raz
  - Kopiec maksymalny, n-węzłów:
    - Kandydat: pełne binarne drzewo przeszukiwań
    - Weryfikacja: przeskanuj raz wszystkie węzły

## PROBLEMY NP - ZUPEŁNE

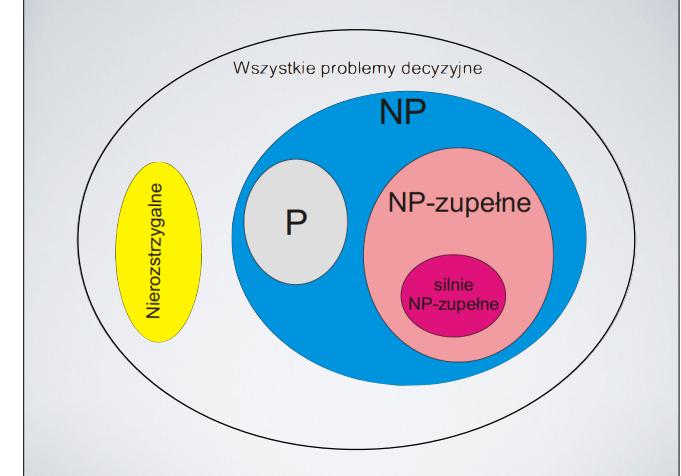
- Problem NP zupełny (NPZ) jest:
  - problemem NP
  - trudniejszy
    - wszystkie problemy równe problemom NP
- Innymi słowy NPZ jest najtrudniejszym problemem NP
- Do tej pory nie odkryto algorytmu wielomianowego dla żadnego problemu NPZ

## PROBLEMY NP - ZUPEŁNE

Większość naukowców wierzy, że: Jednak, nie jest wykluczone, że



Problemy NP-trudne: problemy trudniejsze lub równe problemom NPZ



## PO CO UCZYĆ SIĘ O NPZ

- Jeden z najistotniejszych powodów to:
  - Jeśli widzimy, że problem jest NPZ, to możemy przestać zwracać sobie głowę tworzeniem algorytmu wielomianowego do jego rozwiązania.
- Należy poinformować szefa, że mamy do czynienia z problemem NPZ
- No dora, ale jak udowodnić, że jest to problem NPZ?

## JAK UDOWODNIĆ, ŻE PROBLEM JEST NPZ?

- Popularną metodą jest udowodnienie, że nie jest prostszy niż znany problem NPZ.
- Aby udowodnić, że problem A jest problemem NPZ:
  - Wybierz problem B (także NPZ)
  - Znajdź transformację wielomianową zamieniającą A w B
- Algorytm redukcyjny
  - Jeśli A może zostać rozwiązane w czasie wielomianowym, to B może zostać rozwiązane także w czasie wielomianowym, co jest zaprzeczeniem, że B jest problemem NPZ.
  - Zatem, A nie może być rozwiązywalne w czasie wielomianowym, aby być problemem NPZ.

## A CO JEŚLI MUSIMY ROZWIĄZAĆ PROBLEM NPZ?

- Kup droższe i lepsze maszyny i... poczekaj
  - ale to może zająć nawet setki lat...
- Zastosuj algorytm aproksymujący
  - Dający prawie optymalne rozwiązanie