Algorytmy i struktury danych (2022/2023)

Krzysztof Diks

Wykład 1 - algorytm - poprawność i złożoność obliczeniowa algorytmu, model obliczeń i złożoność problemu algorytmicznego, wybrane metody projektowania wydajnych algorytmów

Plan wykładu:

(01) 03.10.2022: algorytm - poprawność i złożoność obliczeniowa algorytmu, model obliczeń i złożoność problemu algorytmicznego, wybrane metody projektowania wydajnych algorytmów
(02) 10.10.2022: sortowanie przez porównania: Insertion Sort, Merge Sort, Heap Sort, Quick Sort
(03) 17.10.2022: Quick Sort - cd., sortowanie przez porównania, dolna granica w modelu drzew decyzyjnych, sortowanie w czasie "liniowym" i jego zastosowania
(04) 24.10.2022: statystyki pozycyjne i algorytm piątek; Algorytm Dijkstry i jego implementacje
(05) 07.11.2022: koszt zamortyzowany, kolejki priorytetowe: kopce zupełne, kopce dwumianowe, kopce Fibonacciego
(06) 14.11.2022: wyszukiwanie i słowniki, drzew wyszukiwań binarnych, zrównoważone drzewa wyszukiwań binarnych - AVL-drzewa, wzbogacanie struktur danych
(07) 21.11.2022: samoorganizujące się struktury danych - drzewa typu "splay", B-drzewa, drzewa czerwono-czarne
(08) 28.11.2022: haszowanie; grafowe algorytmy macierzowe
(09) 05.12.2022: przeszukiwania grafów i ich zastosowania
(10) 12.12.2022: przeszukiwania grafów i ich zastosowania cd., minimalne drzewo rozpinające
(11) 19.12.2022: problem sumowania zbiorów rozłącznych (Find-Union); algorytmy tekstowe: KMP, drzewa sufiksowe
(12) 09.01.2023: tablice i drzewa sufiksowe
(13) 16.01.2023: o pewnych problemach obliczeniowo trudnych
(14) 23.01.2023: ciekawostki ze świata algorytmiki lub odrabianie zaległości

Podstawowa literatura:

- http://www.smurf.mimuw.edu.pl
- Lech Banachowski, Krzysztof Diks, Wojciech Rytter, Algorytmy i struktury danych, PWN 2018
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein, Wprowadzenie do algorytmów, PWN 2012

Algorytm (nieformalnie): przepis postępowania, który pozwala na *automatyczne* rozwiązanie zadania

Podstawowe charakterystyki algorytmu:

- **poprawność**
- **złożoność obliczeniowa**
  - pamięciowa
  - czasowa

Specyfikacja algorytmu: para warunków $<\alpha,\beta>$

- α – warunek początkowy, który muszą spełniać dane wejściowe
- β – warunek końcowy, który muszą spełniać wyniki

Powiemy, że **algorytm A jest (całkowicie) poprawny** względem specyfikacji $<\alpha,\beta>$ (inaczej, zgodny z tą specyfikacją), jeżeli dla każdych danych spełniających warunek α obliczenie algorytmu A kończy się prawidłowo i wyniki spełniają warunek β.

Algorytm wraz ze specyfikacją zapisujemy w postaci $\{\alpha\}$ A $\{\beta\}$.

Algorytm 1.1 – potęgowanie binarne

Dane: n – dodatnia liczba całkowita {α}
x – liczba rzeczywista

Wynik: $y = x^n$ {β}

Algorytm:

```
begin
  z := x; y := 1; m := n;
  repeat
    if odd(m) then y := y*z;
    m := m div 2;
    z := z*z
  until m = 0
end
```

n	z	y	m
13	x	1	13
	x^2	x	6
	x^4	x	3
	x^8	x^5	1
	x^{16}	x^{13}	0

Dowodzenie poprawności algorytmów – metoda niezmienników

Powiemy, że warunek γ jest **niezmiennikiem** instrukcji I algorytmu A w wyróżnionym jej miejscu, przy warunku początkowym α, jeżeli dla każdego obliczenia algorytmu A dla danych spełniających α, ilekroć obliczenia dociera do wyróżnionego miejsca instrukcji I, spełniony jest warunek γ.

Powiemy, że algorytm A jest (**całkowicie**) **poprawny** względem specyfikacji <α,β>, jeśli spełnia 3 następujące warunki:

1. **[częściowa poprawność]** dla każdych danych wejściowych spełniających warunek początkowy α, jeśli obliczenie algorytmu A kończy się prawidłowo, wyniki spełniają warunek β

2. **[określoność obliczeń]** dla każdych danych wejściowych spełniających warunek początkowy α obliczenie algorytmu A nie jest przerwane

3. [**własność stopu]** dla każdych danych wejściowych spełniających warunek początkowy α obliczenie algorytmu A nie jest nieskończone

Dwie podstawowe reguły Hoare'a wykorzystywane w dowodzeniu poprawności pętli:

$$\frac{\{\gamma \wedge W\}\ K\ \{\gamma\}}{\{\gamma\}\ \textbf{while}\ W\ \textbf{do}\ K\ \{\gamma \wedge \neg W\}} \qquad \frac{\alpha \Rightarrow \alpha'\ \{\alpha'\}\ K\ \{\beta'\}\ \beta' \Rightarrow \beta}{\{\alpha\}\ K\ \{\beta\}}$$

begin { n – dodatnia liczba całkowita, x – liczba rzeczywista }

```
  z := x; y := 1; m := n;
  repeat { γ: x^n = y*z^m ∧ m > 0}
    { m = 2k + [0/1] ∧ x^n = y*z^(2k + [0/1]) }
    if odd(m) then y := y*z;
    { x^n = y*z^2k }
    m := m div 2;
    { m = k ∧ x^n = y*z^2m = y*(z*z)^m }
    z := z*z
    { x^n = y*z^m ∧ m ≥ 0 }
  until m = 0;
end { y = x^n }
```

Dwie podstawowe metody dowodzenia własności stopu:

- metoda liczników iteracji
- metoda malejących wielkości

Problem (otwarty) Collatza:

Czy poniższy algorytm ma własność stopu?

```
{ n – dodatnia liczba całkowita }
begin
  x := n;
  while x ≠ 1 do
    if odd(x) then
      x := 3*x + 1
    else
      x := x div 2
end
```

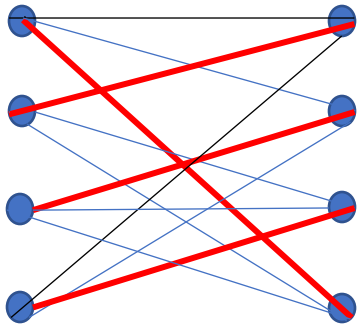
Dla wszystkich liczb $\leq 2^{69}$ odpowiedzią jest TAK!

https://pcbarina.fit.vutbr.cz/

Algorytm 1.2 – doskonałe skojarzenie w dwudzielnych grafach kubicznych (regularnych stopnia 3)

Dane: G = (V,E) – kubiczny (regularny stopnia 3) graf dwudzielny o co najmniej 6 wierzchołkach

Wynik: M ⊆ E – doskonałe skojarzenie w grafie G



Algorytm (Alexander Schrijver, 2001):

```
begin
  for each e ∈ E do w(e):= 1;
  while istnieje w G cykl (elementarny)
        złożony z krawędzi o wagach dodatnich do
  begin
    C := cykl(elementarny)złożony z krawędzi o wagach dodatnich;
    (M1,M2) := dwa, rozłączne, doskonałe skojarzenia w C;
    if W(M1) < W(M2) then M1 :=: M2; {W(M) = ∑e∈E w(e); :=: - zamiana}
    { W(M1) ≥ W(M2)}
    for each e ∈ M1 do w(e):= w(e)+1;
    for each e ∈ M2 do w(e):= w(e)-1
  end;
  M := {e ∈ E: w(e)> 0}
end
```

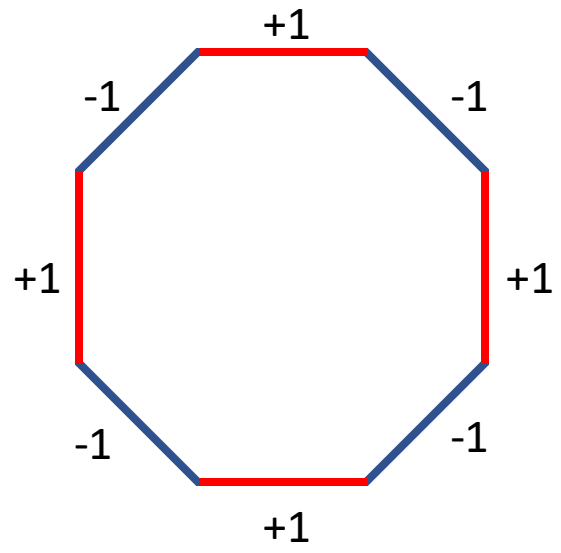
Poprawność algorytmu Schrijver'a – częściowa poprawność:

```
for each e ∈ E do w(e):= 1;

  while istnieje w G cykl (elementarny)
        złożony z krawędzi o wagach dodatnich do
  begin
    C := cykl(elementarny)złożony z krawędzi
         o wagach dodatnich;
    (M1,M2) := dwa, rozłączne,
               doskonałe skojarzenia w C;
    if W(M1) < W(M2) then M1 :=: M2;
    { W(M1) ≥ W(M2) }
    for each e ∈ M1 do w(e):= w(e)+1;
    for each e ∈ M2 do w(e):= w(e)-1
  end;
  M := {e ∈ E: w(e)> 0}
end
```

Niezmiennik pętli „while":

- waga każdej krawędzi jest nieujemną liczbą całkowitą
- dla każdego wierzchołka, suma wag krawędzi go opuszczających wynosi 3



Poprawność algorytmu Schrijver'a – częściowa poprawność:

niezmiennik pętli „while":

- waga każdej krawędzi jest nieujemną liczbą całkowitą
- dla każdego wierzchołka, suma wag krawędzi go opuszczających wynosi 3

oraz

zaprzeczenie dozoru pętli:

- podgraf H grafu G (V(H) = V(G)) rozpięty na krawędziach o dodatnich wagach nie zawiera pętli

ZATEM MAMY:

- H jest lasem
- stopień każdego wierzchołka w grafie H wynosi co najmniej 1, czyli z każdego wierzchołka wychodzi co najmniej 1 krawędź
- w H istnieje wierzchołek v stopnia 1, a jedyna krawędź v—u go opuszczająca ma wagę 3; tak więc u tez ma stopień 1
- z powyższego wynika, że stopień każdego wierzchołka w H jest stopnia 1 i krawędzie H tworzą doskonałe skojarzenie w G

Poprawność algorytmu Schrijver'a – warunek stopu:

```
for each e ∈ E do w(e):= 1;

  while istnieje w G cykl (elementarny)
        złożony z krawędzi o wagach dodatnich do
  begin
    C := cykl(elementarny)złożony z krawędzi
         o wagach dodatnich;
    (M1,M2) := dwa, rozłączne,
               doskonałe skojarzenia w C;
    if W(M1) < W(M2) then M1 :=: M2;
    { W(M1) ≥ W(M2) }
    for each e ∈ M1 do w(e):= w(e)+1;
    for each e ∈ M2 do w(e):= w(e)-1;
  end;
  M := {e ∈ E: w(e)> 0}
end
```

Z grafem G wiążemy zmieniającą się wartość $f =_{df} \sum_{e \in E} w^2(e)$.

Wartość f zmienia się w jednym obrocie pętli o:

$\sum_{e \in M_1}((w(e)+1)^2 - w(e)^2)$

+

$\sum_{e \in M_2}((w(e)-1)^2 - w(e)^2)$

=

$\sum_{e \in M_1} 2 * w(e) + |M_1| - \sum_{e \in M_2} 2 * w(e) + |M_2|$

$\geq \quad \{\sum_{e \in M_1} 2 * w(e) \geq \sum_{e \in M_2} 2 * w(e)\}$

$|M_1| + |M_2| = |C| > 0$

Mamy $3 * \frac{|V|}{2} \leq f \leq 9 * \frac{|V|}{2}$.

Złożoność obliczeniowa algorytmu:

- pamięciowa: ilość pamięci (komputera) niezbędna do wykonania algorytmu
  jednostka miary – słowo pamięci maszyny
- czasowa: czas pracy (komputera) niezbędny do zrealizowania algorytmu
  jednostka miary – operacja dominująca (liczba wszystkich operacji jednostkowych wykonywanych przez algorytm powinna być „proporcjonalna” do liczby wszystkich operacji dominujących)

Przykład:

```
begin {n – dodatnia liczba całkowita; x – liczba rzeczywista}
  z := x; y := 1; m := n;
  repeat
    if odd(m) then y := y*z;
    m := m div 2;
    z := z*z
  until m = 0;
end { y = x^n }
```

operacja dominująca: mnożenie „*”

$\lceil \log(n+1) \rceil <$ liczba operacji dominujących $\leq 2\lceil \log(n+1) \rceil$

wielkość pamięci - stała

Nie jest zazwyczaj możliwe wyznaczenie złożoności obliczeniowej jako funkcji danych wejściowych!

Złożoność obliczeniową mierzymy jako funkcję *rozmiaru* danych wejściowych, rozumianego jako (mówiąc ogólnie) liczbę „pojedynczych" danych na wejściu. Rozmiar powinien tak charakteryzować wielkość danych, żeby złożoność obliczeniowa rzeczywiście odpowiadała wydajności analizowanego algorytmu.

Złożoność czasowa algorytmu dla danych rozmiaru n:

Niech
D_n – zbiór możliwych danych wejściowych rozmiaru n
t(d) – liczba operacji dominujących dla zestawu danych d
X_n – zmienna losowa, której wartością jest t(d) dla $d \in D_n$
$p_{n,k}$ – prawdopodobieństwo, że dla danych rozmiaru n algorytm wykona k operacji dominujących

Przez **pesymistyczną złożoność czasową algorytmu** rozumie się funkcję
$W(n) = \sup \{t(d): d \in D_n\}$, gdzie sup oznacza kres górny zbioru.
Przez **oczekiwaną złożoność czasową algorytmu** rozumie się funkcję
$A(n) = \sum_{k \geq 0} k p_{n,k} = E[X_n]$, gdzie $E[X_n]$ oznacza wartość oczekiwaną zmiennej losowej X_n.

Złożoność czasowa algorytmu potęgowania binarnego:

```
begin {n – dodatnia liczba całkowita; x –
liczba rzeczywista}
  z := x; y := 1; m := n;
  repeat
    if odd(m) then y := y*z;
    m := m div 2;
    z := z*z
  until m = 0;
end { y = x^n }
```

rozmiar danych: r = $\lceil \log(n+1) \rceil$

r < W(r) ≤ 2r

Algorytm 1.3 – sortowanie przez wstawianie (Insertion Sort)
Dane: dodatnia liczba całkowita n
tablica a[1..n] = [e_1, e_2, …, e_n] elementów z uniwersum z liniowym porządkiem (U,≤)
Wynik: tablica a[1..n] = [$e_{i_1} \leq e_{i_2} \leq \ldots \leq e_{i_n}$]

Algorytm:

```
begin
  a[0] := -∞; { strażnik }
  for i ∈ [2,..,n] do { i przebiega kolejne liczby w przedziale }
  begin
    v := a[i]; j := i-1;
    while v < a[j] do                    { operacja dominująca }
    begin a[j+1]:= a[j]; j ;= j-1 end;
    a[j] := v
  end
end
```

Sortowanie przez wstawianie – analiza złożoności

Inwersją w ciągu (tablicy) a = [e_1, e_2, …, e_n] nazywamy każdą parę indeksów (i,j),
$1 \le i < j \le n$, taką że $e_i > e_j$. Inaczej, inwersja to para nieuporządkowanych elementów w ciągu.
Liczbę inwersji w ciągu (tablicy) a oznaczamy przez Inv(a).

$$0 \le \text{Inv(a)} \le n(n-1)/2$$

Złożoność czasowa algorytmu sortowania przez wstawianie dla danej tablicy a[1..n], mierzona liczbą porównań „v < a[j]”, wynosi

$$n\text{-}1 + \text{Inv(a)}$$

Pesymistyczna złożoność czasowa algorytmu sortowania przez wstawianie wynosi

$$W(n) = n\text{-}1 + n(n\text{-}1)/2$$

Algorytm sortowania przez wstawianie jest **algorytmem w miejscu** – poza pamięcią na przechowywanie danych, rozmiar dodatkowej pamięci niezbędnej do jego realizacji jest stały, niezależny od rozmiaru danych.

Algorytm sortowania przez wstawianie jest **algorytmem stabilnym** – zachowuje względny porządek elementów o tych samych wartościach.

Notacja asymptotyczna:

Poniżej rozważamy funkcje o argumentach będących nieujemnymi liczbami całkowitymi i o nieujemnych, rzeczywistych wartościach.

Dla danej funkcji g(n) przez O(g(n)) oznaczamy zbiór funkcji

$O(g(n)) = \{f(n)$: istnieją dodatnie stałe c i n_0 takie, że $0 \leq f(n) \leq cg(n)$ dla każdego $n \geq n_0$.

Piszemy $f(n) = O(g(n))$, gdy $f(n) \in O(g(n))$.

Dla danej funkcji g(n) przez Ω(g(n)) oznaczamy zbiór funkcji

$\Omega(g(n)) = \{f(n)$: istnieją dodatnie stałe c i n_0 takie, że $0 \leq cg(n) \leq f(n)$ dla każdego $n \geq n_0$.

Piszemy $f(n) = \Omega(g(n))$, gdy $f(n) \in \Omega(g(n))$.

Dla danej funkcji g(n) przez Θ(g(n)) oznaczamy zbiór funkcji

$\Theta(g(n)) = \{f(n)$: istnieją dodatnie stałe c_1, c_2 i n_0 takie, że $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla każdego $n \geq n_0$.

Piszemy $f(n) = \Theta(g(n))$, gdy $f(n) \in \Theta(g(n))$.

Notacja asymptotyczna:

Dla danej funkcji g(n) przez o(g(n)) oznaczamy zbiór funkcji
o(g(n)) = {f(n): dla każdej dodatniej stałej c istnieje dodatnia stała n_0, taka że $0 \leq f(n) < cg(n)$ dla każdego $n \geq n_0$.
Piszemy f(n) = o(g(n)), gdy $f(n) \in o(g(n))$.

Dla danej funkcji g(n) przez ω(g(n)) oznaczamy zbiór funkcji
ω(g(n)) = {f(n): dla każdej dodatniej stałej c istnieje dodatnia stała n_0, taka że $0 \leq cg(n) < f(n)$ dla każdego $n \geq n_0$.
Piszemy f(n) = ω(g(n)), gdy $f(n) \in \omega(g(n)$.

Notacja asymptotyczna:

Potęgowanie binarne: $W(n) = \Theta(\log n)$

Sortowanie przez wstawianie: $W(n) = \Theta(n^2)$

Analiza średniego czasu działania sortowania przez wstawianie:

Czas działania algorytmu Insertion Sort, liczony liczbą porównań, zależy od liczby inwersji Inv w sortowanym ciągu, $0 \le Inv \le n(n-1)/2$, i wynosi $n-1 + Inv$.

Model probabilistyczny: na wejściu pojawia się **losowa permutacja** liczb 1, 2, …, n.

Ile jest średnio inwersji w losowej permutacji $p = [e_1, e_2, \ldots, e_n]$?

X_n – zmienna losowa, której wartością jest liczba inwersji w p

Ile wynosi $E[X_n]$?

Dla $1 \le i < j \le n$ niech $X_{i,j}$ będzie zmienną losową przyjmująca wartość 1, gdy $e_i > e_j$, natomiast wartość 0, gdy $e_i < e_j$.

$$X_n = \sum_{1 \le i < n} \sum_{i < j \le n} X_{i,j}$$

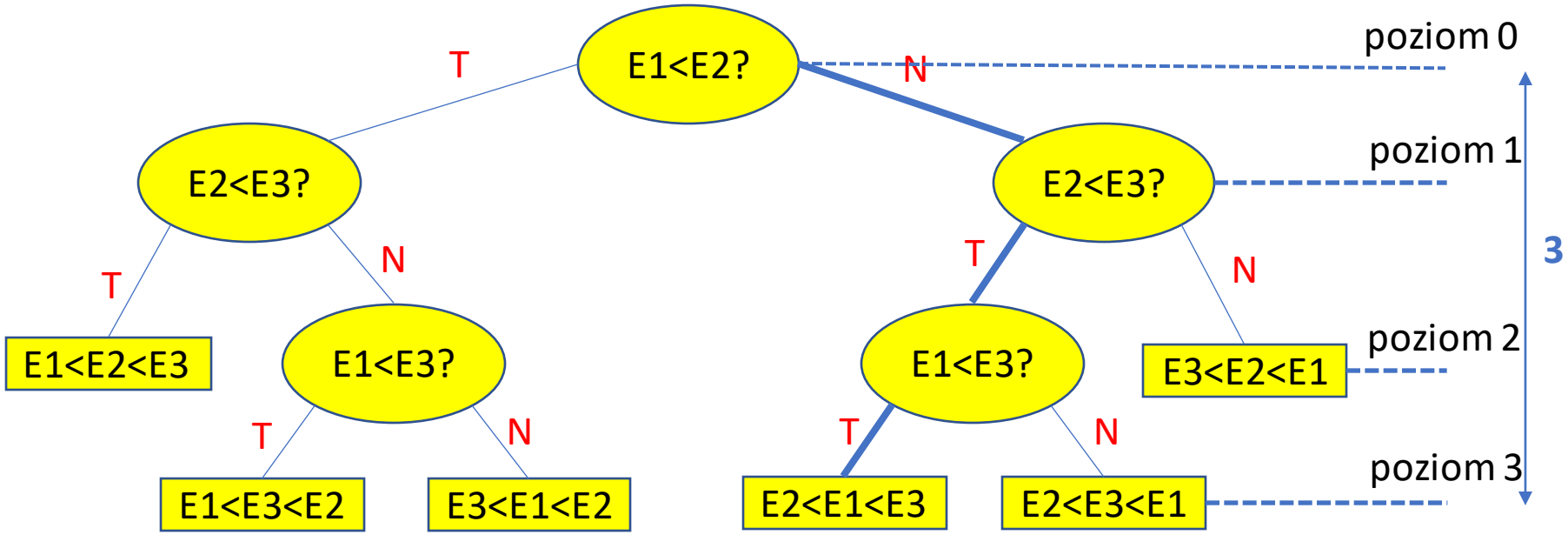
$$E[X_n] = \sum_{1 \le i < n} \sum_{i < j \le n} E[X_{i,j}] = \sum_{1 \le i < n} \sum_{i < j \le n} \frac{1}{2} = n(n-1)/4$$

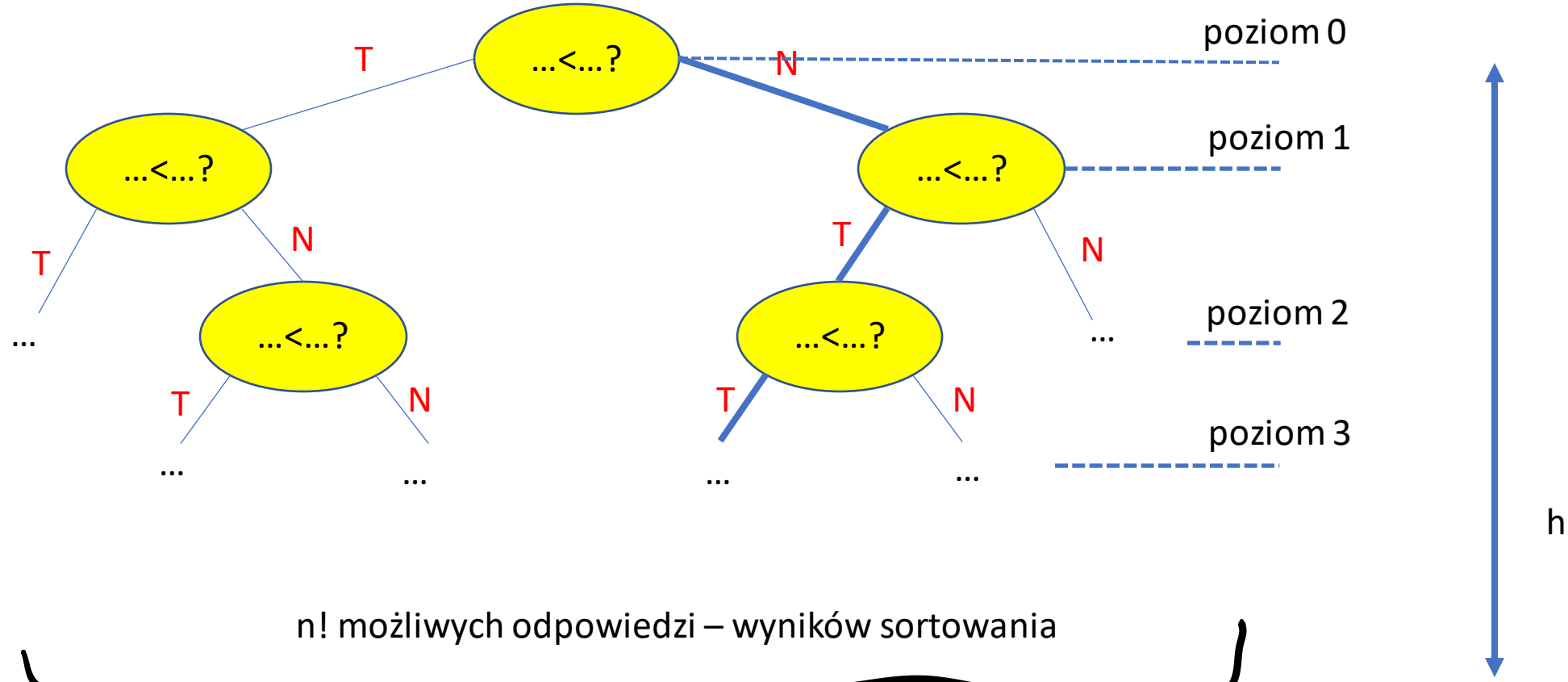
Zatem w modelu losowych permutacji średni czas działania algorytmu sortowania przez wstawianie, mierzony liczbą porównań, wynosi

$$A(n) = n - 1 + \frac{n(n-1)}{4} = \Theta(n^2)$$

Czy można sortować szybciej przez porównania?

Model drzew decyzyjnych – sortowanie trzech elementów E1, E2, E3





n! możliwych odpowiedzi – wyników sortowania

$$2^h \geq n! \quad \xrightarrow{\text{stąd}} \quad h \geq \lceil \log n! \rceil = n \log n - \frac{n}{\ln 2} - \frac{1}{2} \log n + O(1)$$

Każdy algorytm sortujący przez porównania wykonuje w pesymistycznym przypadku $\Omega(n \log n)$ porównań.

Czy można przyśpieszyć sortowanie przez wstawianie przez zmniejszenie liczby wykonywanych porównań?

```
begin
  a[0] := -∞;
  for i ∈ [2,..,n] do
  begin
    v := a[i];

    j := i-1;
    while v < a[j] do
    begin a[j+1]:= a[j]; j ;= j-1 end;

    a[j] := v
  end
end
```

a[0..i-1] jest posortowana
miejsce wstawienia v = a[i]
można wyznaczyć z pomocą wyszukiwania binarnego

```
BS(i,v)::
{ szukamy miejsca wstawienia v
  w uporządkowaną tablicę a[0..i-1], 1 < i ≤ n, a[0] = -∞ }
begin
  l := 1; p := i-1;
  while l ≤ p do
  begin
  {a[0..i-1] – uporządkowana, 1 ≤ l ≤ p ≤ i-1,
  a[l-1] ≤ v ≤ a[p+1]}
    s := (l+p) div 2;
    if v ≥ a[s] then
      l := s+1
    else
      p := s-1
  end;
  return l
end
```

Liczba porównań jest równa $\lceil \log i \rceil$

Czy można przyśpieszyć sortowanie przez wstawianie przez zmniejszenie liczby wykonywanych porównań?

Algorytm 1.3, sortowanie przez wstawianie z wyszukiwaniem binarnym

```
begin
  a[0] := -∞;
  for i ∈ [2,..,n] do
  begin
    v := a[i];

    j := i-1;
    while v < a[j] do
    begin a[j+1]:= a[j]; j := j-1 end;

    a[j] := v
  end
end
```

```
j := BS(i,v);
for k ∈ [i-1..j] do
  a[k+1] := a[k];
```

Łączna liczba porównań: $\sum_{i=2}^{n} \lceil \log i \rceil = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$

Co z przesunięciami „a[k+1] := a[k]”?

Trzy wybrane metody projektowania wydajnych algorytmów z przykładami:

- metoda „dziel i zwyciężaj”: mnożenie macierzy algorytmem Strassena
- programowanie dynamiczne: optymalne mnożenie łańcucha macierzy
- metoda zachłanna: system różnych reprezentantów dla przedziałów

Metoda „dziel i zwyciężaj”, algorytm Strassena mnożenia macierzy:

Niech A = $(a_{i,j})$ i B = $(b_{i,j})$ będą macierzami (liczb rzeczywistych) o wymiarach nxn. Iloczyn C=A*B macierzy A i B definiujemy jako macierz C = $(c_{i,j})$, gdzie $c_{i,j} =_{df} \sum_{k=1}^{n} a_{i,k} * b_{k,j}$.

Z definicji obliczenie $c_{i,j}$ wymaga n mnożeń i n-1 dodawań. Zatem koszt obliczenia macierzy C to n^3 mnożeń i $(n-1)n^2$ dodawań. Zatem koszt wymnożenia dwóch macierzy nxn wynosi $\Theta(n^3)$ operacji arytmetycznych.

Czy można szybciej mnożyć macierze?

Dla dalszych rozważań (bez straty ogólności) załóżmy, że n jest potęgą dwójki.

Jeśli n = 1 to mnożenie macierzy sprowadza się do jednego mnożenia. Załóżmy zatem, że n > 1.
Dzielimy macierze na cztery podmacierze o wymiarach n/2 x n/2.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Wówczas $C_{i,j} = A_{i,1} * B_{1,j} + A_{i,2} * B_{2,j}$.

Niech T(n) oznacza koszt rekurencyjnego wymnożenia macierzy.

Mamy $T(1) = \Theta(1)$ oraz $T(n) = 8T(n/2) + \Theta(n^2)$, dla n > 1.

Twierdzenie o rekurencji uniwersalnej

Niech $a \geq 1$ i $b > 1$ będą stałymi, niech $f(n)$ będzie pewną funkcją o wartościach nieujemnych i niech $T(n)$ będzie zdefiniowane dla nieujemnych liczb całkowitych przez rekurencję

$$T(n) = aT(n/b) + f(n)$$

gdzie n/b bierzemy z podłogą lub sufitem. Wówczas $T(n)$ może być ograniczone asymptotycznie w następujący sposób:

1. Jeśli $f(n) = O(n^{\log_b a - \varepsilon})$ dla pewnej stałej $\varepsilon > 0$, to $T(n) = \Theta(n^{\log_b a})$.
2. Jeśli $f(n) = \Theta(n^{\log_b a})$, to $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Jeśli $f(n) = \Omega(n^{\log_b a + \varepsilon})$ dla pewnej stałej $\varepsilon > 0$, oraz $af\left(\frac{n}{b}\right) \leq cf(n)$ dla pewnej stałej $c < 1$ i wszystkich dostatecznie dużych n, to $T(n) = \Theta(f(n))$.

Rekurencyjne mnożenie macierzy: $T(n) = 8T(n/2) + \Theta(n^2)$

Mamy $a = 8$, $b = 2$, $f(n) = O(n^2)$, $\varepsilon = 1$. Zatem $T(n) = \Theta(n^3)$.

Algorytm 1.4, Strassen, 1969

$M_1 = (A_{1,2} - A_{2,2})*(B_{2,1} + B_{2,2})$
$M_2 = (A_{1,1} + A_{2,2})*(B_{1,1} + B_{2,2})$
$M_3 = (A_{1,1} - A_{2,1})*(B_{1,1} + B_{1,2})$
$M_4 = (A_{1,1} + A_{1,2})*B_{2,2}$
$M_5 = A_{1,1}*(B_{1,2} - B_{2,2})$
$M_6 = A_{2,2}*(B_{2,1} - B_{1,1})$
$M_7 = (A_{2,1} + A_{2,2})*B_{1,1}$

Teraz

$C_{1,1} = M_1 + M_2 - M_4 + M_6$
$C_{1,2} = M_4 + M_5$
$C_{2,1} = M_6 + M_7$
$C_{2,2} = M_2 - M_3 + M_5 - M_7$

Zatem
$T(n) = 7T(n/2) + \Theta(n^2)$ z twierdzenie o rekurencji uniwersalnej (punkt 1) mamy

$$T(n) = O\left(n^{\log_2 7}\right) = O(n^{2,81})$$

Na dzisiaj najszybszy algorytm działa w czasie $O(n^{\omega})$, gdzie $\omega < 2.3728596$ (Josh Alman, Virginia Vassilevska, 2014).

Programowanie dynamiczne – optymalne wymnożenie łańcucha macierzy

Dane: dodatnia liczba całkowita n

n macierzy M_1, …, M_n o wymiarach odpowiednio r_{i-1} x r_i, dla i = 1, …, n

Wynik: optymalny ze względu na łączną liczbę mnożeń sposób obliczenia iloczynu

M_1*…* M_n

Przykład

Załóżmy, że chcemy obliczyć iloczyn 3 macierzy A, B, C o wymiarach odpowiednio 5x10, 10x3 oraz 3x4.
Mamy dwa sposoby obliczenia iloczynu tych macierzy: (A*B)*C oraz A*(B*C).
Koszt obliczeń pierwszym sposobem wynosi 5*10*3+5*3*4 = 210.
Koszt obliczeń drugim sposobem wynosi 10*3*4 + 5*10*4 = 320.
Pierwszy sposób jest lepszy.

Liczba możliwych sposobów wymnożenia n macierzy wynosi C_{n-1}, gdzie C_n jest n-tą liczbą Catalana

$$C_n = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Niech $w_{i,j}$ oznacza optymalny koszt obliczenia iloczynu $M_{i+1}*\ldots* M_j$ dla $0 \le i < j \le n$.
Zastanówmy się w jaki sposób wyznaczyć (obliczyć) $w_{0,n}$. Pomaga rekurencja!

$$w_{i,j} = \begin{cases} 0 & i+1 = j \\ MIN_{i<k<j}\left(w_{i,k} + w_{k,j} + r_i * r_k * r_j\right) & i+1 < j \end{cases}$$

$$\underset{r_i \times r_k}{((M_{i+1} * \ldots * M_k)} * \underset{r_k \times r_j}{(M_{k+1} * \ldots * M_j))}$$

Rekurencyjne obliczenie $w_{0,n}$ wymaga policzenia kosztu wymnożenia macierzy dla każdego możliwego sposobu – rozstawienia nawiasów!

Spostrzeżenie: wielokrotnie liczymy to samo!
Dla przykładu: $w_{0,3}$ obliczamy licząc rekurencyjnie $w_{0,4}$, $w_{0,5}$, $w_{0,6}$ itd.

Pomysł: tablicujmy to co już policzyliśmy!
Gdyby $w_{i,k}$, $w_{k,j}$ były już policzone dla k = i+1, …,j-1, to policzenie $w_{i,j}$ kosztowałoby tylko $\Theta(j-i)$ operacji arytmetycznych!

Algorytm 1.5, obliczanie kosztu optymalnego wymnażania łańcucha macierzy

```
begin
  for i ∈ [0..n-1] do
    w[i,i+1]:= 0;
  for s ∈ [2..n] do
    for i ∈ [0..n-s] do
    begin
      j := i+s;
      w[i,j] := +∞;
      for k ∈ [i+1..j-1]do
        w[i,j] := MIN(w[i,j],w[i,k]+w[k,j]+r_i*r_k*r_j);
    end
end
```

Ile razy wykonujemy tę instrukcję?

Złożoność czasowa tego algorytmu wynosi $\Theta(n^3)$.

Algorytm 1.6, optymalne wymnażanie łańcucha macierzy

Algorytm 1.5 służył do wyznaczenia kosztu optymalnego wymnożenia łańcucha macierzy.

Przyjrzyjmy się raz jeszcze rekurencji na optymalny koszt $w_{i,j}$.

$$w_{i,j} = \begin{cases} 0 & i+1 = j \\ \mathrm{MIN}_{i<k<j}\left(w_{i,k} + w_{k,j} + r_i * r_k * r_j\right) & i+1 < j \end{cases}$$

$$\begin{matrix} ((M_{i+1} * \ \ldots * \ M_k) & * & (M_{k+1} * \ \ldots * \ M_j)) \\ r_i \times r_k & & r_k \times r_j \end{matrix}$$

Znajomość wartości k, która minimalizuje wartość $w_{i,k} + w_{k,j} + r_i * r_k * r_j$ pozwala prawidłowo rozstawić nawiasy i obliczyć najpierw iloczyn $L = (M_{i+1} * \ \ldots * \ M_k)$, potem P = $(M_{k+1} * \ \ldots * \ M_j)$ i Na koniec L*P.

Przyjmijmy, że podczas obliczania $w_{i,j}$ wartość k minimalizująca wyrażenie $w_{i,k} + w_{k,j} + r_i * r_k * r_j$ zapamiętamy na $w_{j,i}$.

```
Iloczyn(i,j)::
{ optymalne obliczanie M_{i+1}*…*M_j }
begin
  if j = i+1 then
    return M_j
  else
  begin
    k := w[j,i];
    L := Iloczyn(i,k);
    P := Iloczyn(k,j);
    return L*P
  end
end
```

Metoda zachłanna, system różnych reprezentantów dla przedziałów liczbowych

Dane: dodatnia liczba całkowita n

n domkniętych przedziałów $[a_1, b_1], \ldots, [a_n, b_n]$, $1 \leq a_i \leq b_i \leq n$ dla $i = 1, \ldots, n$

Wynik: (SRR) ciąg $x_1, \ldots, x_n$ (o ile istnieje) taki, że $x_i \in [a_i, b_i]$ oraz $x_i \neq x_j$ dla $i \neq j$

Przykład

n = 5

[2,4], [1,2], [4,5], [3,4], [1,3]

2 1 5 4 3

[1,4], [2,3], [3,3], [2,2], [4,5]

SRR nie istnieje

Algorytm 1.7, zachłanne wyznaczanie SRR dla przedziałów

```
begin
  podziel przedziały na zbiory X_i, gdzie                    {sortowanie}
  X_i to zbiór przedziałów o początku w i;
  X := ∅;
  for i ∈ [1..n] do
  begin
    X := X ∪ X_i;
    if X = ∅ then
      exit(„SRR nie istnieje")
    else
    begin
      J := przedział w X z najmniejszym prawym końcem;       {zachłanny wybór}
      x_indeks(J) := i;
      X := X\{J};
      X := X\{wszystkie przedziały z X o końcu w i}
    end
end
```

Implementację algorytmu 1.7 i analizę złożoności pozostawiamy na dalsze wykłady.

Wykład opracowano między innymi na podstawie książek:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein, Wprowadzenie do algorytmów, PWN 2012
- Lech Banachowski, Krzysztof Diks, Wojciech Rytter, Algorytmy i struktury danych, PWN 2018
- Lech Banachowski, Antoni Kreczmar, Elementy analizy algorytmów, WNT 1989