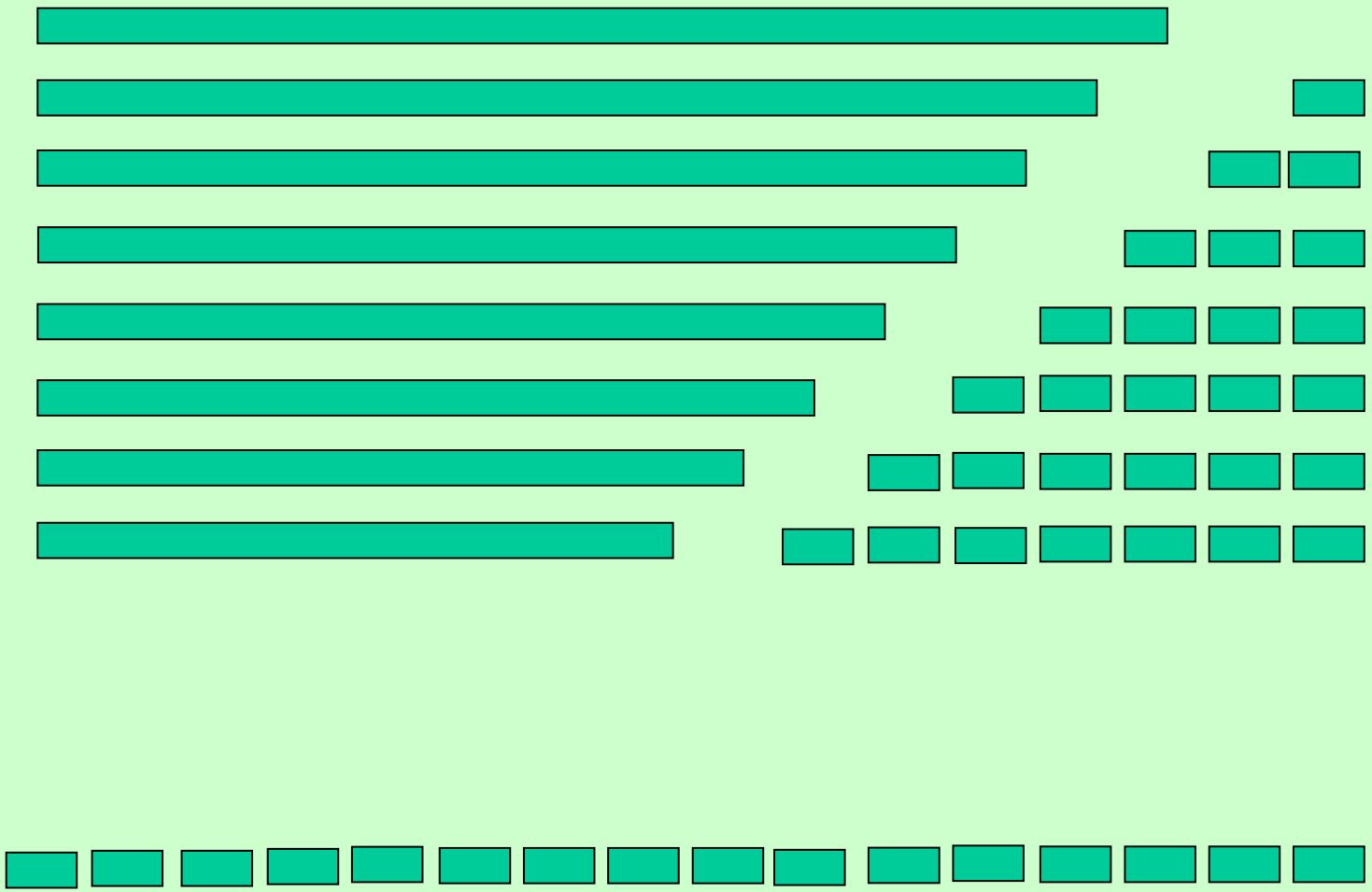
Dzielić i zwyciężać

Czyli jak szybko pokroić szczypiorek i nie tylko …

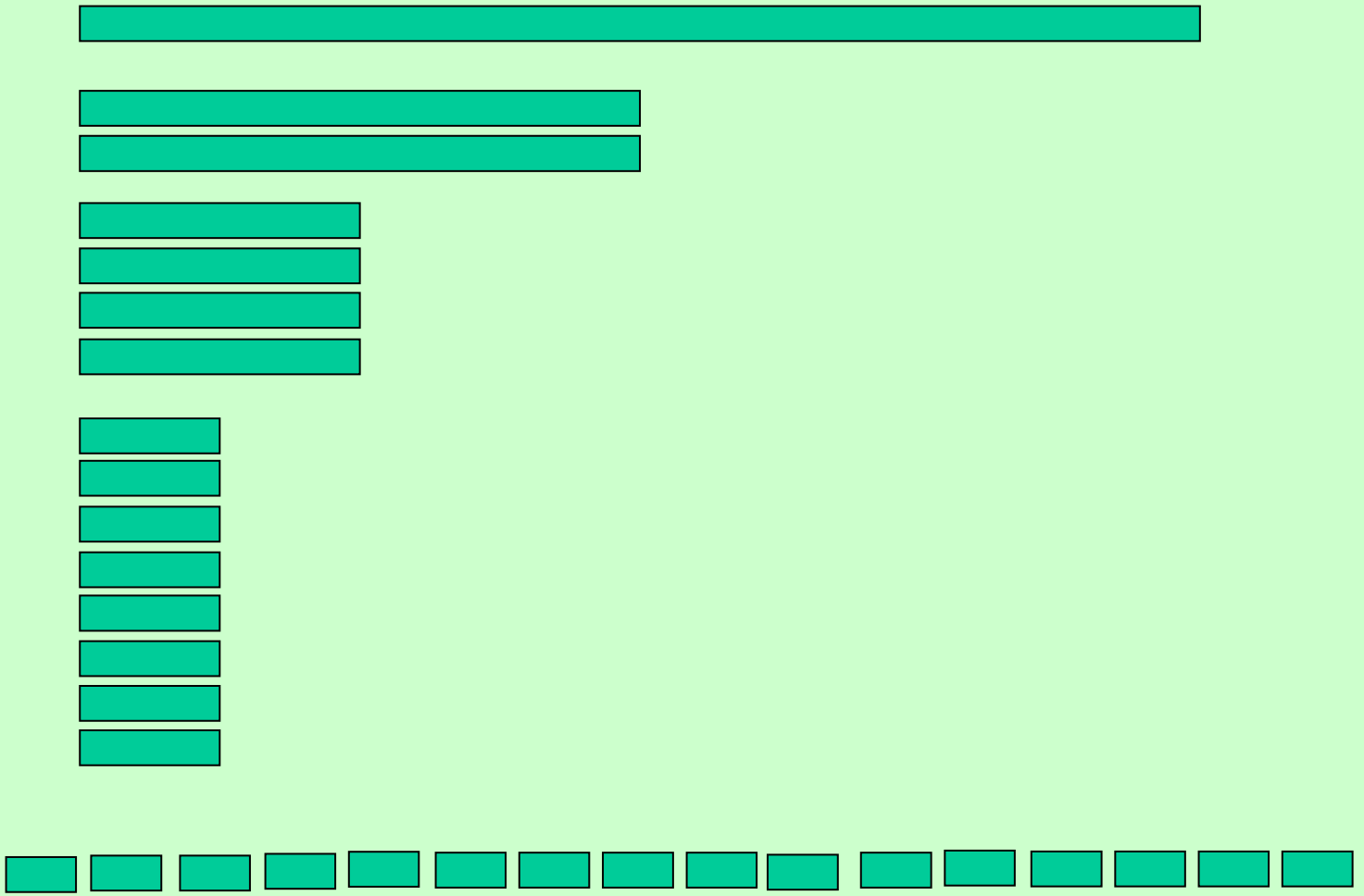
dr Beata Laszkiewicz

Czy szczypiorek i jego krojenie ma coś wspólnego z informatyką?

Strategia I krojenia szczypiorku o dł. n = 16



Strategia II krojenia szczypiorku o dł. n = 16



Ile operacji krojenia wykonaliśmy?

Metoda 1

- w każdym kroku odcinaliśmy jedną jednostkę;
- dla n=16 wykonaliśmy 15 operacji.

Metoda 2

- w każdym kroku, wykonując jedno krojenie, zmniejszaliśmy długość szczypiorku o połowę;
- dla n=16 wykonaliśmy 4 operacje.

Ile operacji wykonujemy?

n	Metoda 1	Metoda 2
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
…	…	…
1024	1023	10
…	…	…
100 000	???	???

99 999

~ 17

Słowo wstępu...

Wiele ważnych algorytmów ma strukturę rekurencyjną: W celu rozwiązania danego problemu algorytm wywołuje sam siebie przy rozwiązywaniu podobnych podproblemów.

Naturalnym podejściem w rozwiązywaniu problemów jest dążenie do wykorzystania znanych rozwiązań problemów.

Metoda rozwiązania problemu

- podziel problem na podproblemy,
- znajdź rozwiązania podproblemów,
- połącz rozwiązania podproblemów w rozwiązanie głównego problemu.

Metoda Dziel i Zwyciężaj

Aby skonstruowany algorytm był efektywny, należy dodać kilka warunków:

- problem jest dzielony na takie same lub bardzo podobne podproblemy,
- liczba podproblemów wynosi co najmniej 2,
- podproblemy są rozwiązywane na podzbiorach zbioru danych, w których liczba elementów jest niemal jednakowa i stanowi stałą cześć (np. połowę) całego zbioru danych rozwiązywanego problemu.

Potęgowanie liczb

Problem: Oblicz a^n, dla $n \in N$.

Algorytm naiwny: $a^n = \underbrace{a \cdot a \cdot a \ldots \cdot a}_{n}$

```
wynik ← 1;
for i ← 1, 2, 3, ..., n
    wynik ← wynik * a;
```

Złożoność: *n*

Potęgowanie liczb

$$a^n = \begin{cases} 1, & gdy\ n = 0 \\ a^{n/2} * a^{n/2}, & gdy\ n\ jest\ parzyste \\ a^{(n-1)/2} * a^{(n-1)/2} * a, & gdy\ n\ jest\ nieparzyste \end{cases}$$

Czas działania: $T(n) = T\left(\frac{n}{2}\right) + kn$

Złożoność: $\log_2 n$ - logarytmiczna

Potęgowanie dla n=16

Metoda klasyczna

$$a^{16} = a^{15} * a$$
$$\hookrightarrow a^{14} * a$$
$$\hookrightarrow a^{13} * a$$
$$\hookrightarrow a^{12} * a$$
$$\hookrightarrow \ldots$$
$$\hookrightarrow a^{2} * a$$
$$\hookrightarrow a^{1} * a$$
$$\hookrightarrow a^{0} * a$$
$$\hookrightarrow 1$$

Metoda D & Z

$$a^{16} = a^{8} * a^{8}$$
$$\hookrightarrow a^{4} * a^{4}$$
$$\hookrightarrow a^{2} * a^{2}$$
$$\hookrightarrow a^{1} * a^{1}$$
$$\hookrightarrow a^{0} * a$$
$$\hookrightarrow 1$$

Ile jest mnożeń?

Inne klasyczne przykłady

- jednoczesne wyszukiwanie minimum i maksimum w n-elementowym nieuporządkowanym np. ciągu liczb,
- sortowanie przez scalanie,
- i wiele, wiele innych.

Część I:

JEDNOCZESNE WYSZUKIWANIE MIN-MAX

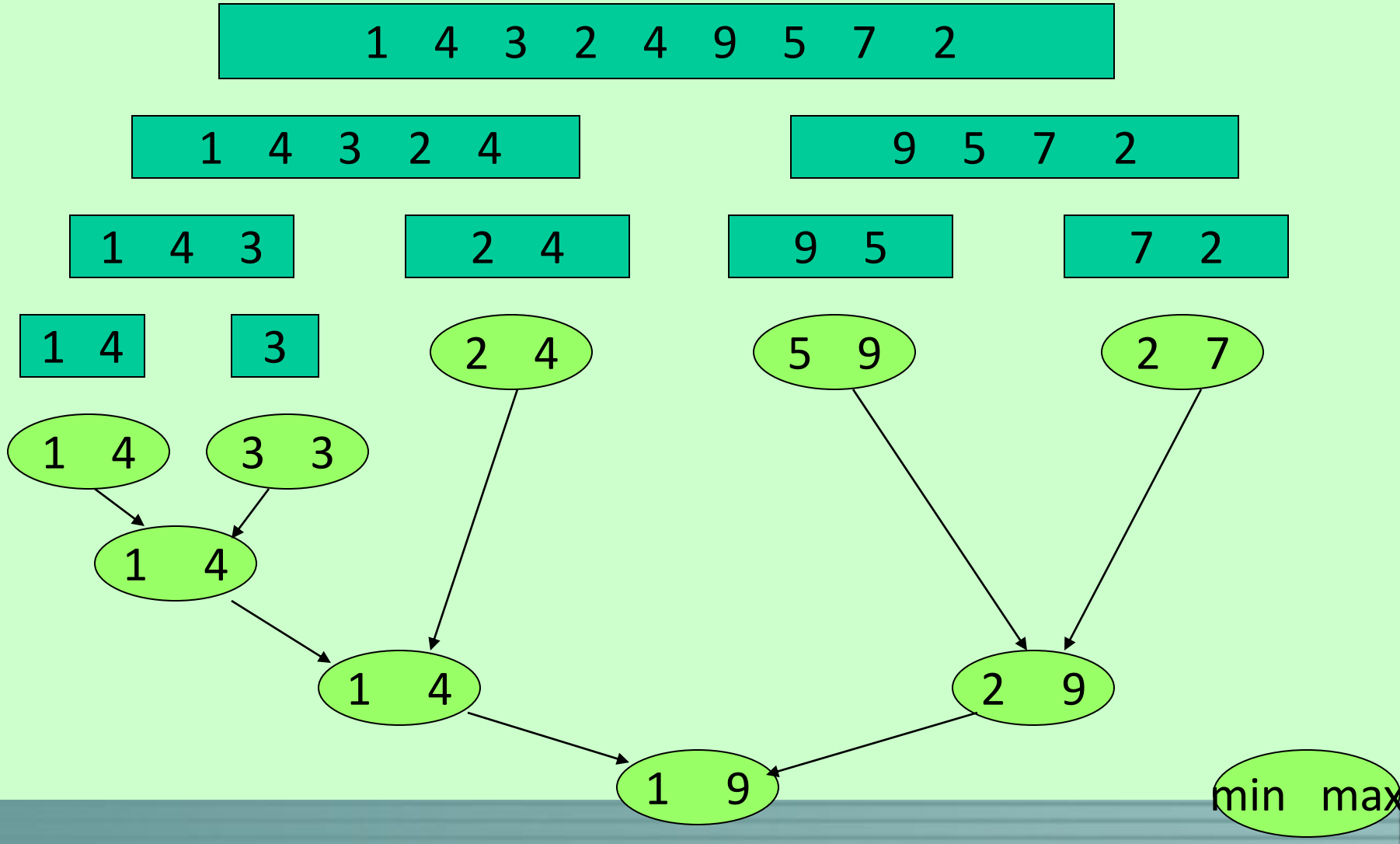
Jak znaleźć najmniejszy i największy element w ciągu *n* elementów?

Najprostsze rozwiązanie:

- Znaleźć minimum przeglądając liniowo ciąg danych
- Znaleźć maksimum przeglądając liniowo ciąg danych

Koszt: (n-1)+(n-1) = 2(n-1)

Jednoczesne wyszukiwanie min i max



Schemat działania

Jeśli w ciągu jest jeden element, to ustaw go jako `min` i jako `max`.

Jeśli w ciągu są 2 elementy, to ustal, który będzie `min`, a który `max`.

Jeśli w ciągu są więcej niż 2 elementy, to:

- podziel ciąg na 2 części: Z_1 oraz Z_2
- wykonaj ten sam algorytm dla Z_1, ustal $\texttt{min}_1$ oraz $\texttt{max}_1$
- wykonaj ten sam algorytm dla Z_2, ustal $\texttt{min}_2$ oraz $\texttt{max}_2$
- wybierz `min` z $\texttt{min}_1$, $\texttt{min}_2$
- wybierz `max` z $\texttt{max}_1$, $\texttt{max}_2$

Algorytm MinMax

Dane: n – liczba elementów
T – n-elementowy nieuporządkowany ciąg liczb
p – początek szukania
k – koniec szukania

Wynik: min – najmniejszy element ciągu
max – największy element ciągu

Uwagi:
T[i] oznacza i-ty element ciągu liczb

```
MinMax(T,p,k,min,max)
if (k-p=0) then
    min ← T[p]
    max ← T[k]
else
    if (k-p=1) then
        if (T[k]<T[p]) then
            min ← T[k]
            max ← T[p]
        else
            min ← T[p]
            max ← T[k]
    else
       s ←(p+k) div 2;
       MinMax(T,p,s,min1, max1)
       MinMax(T,s+1,k,min2,max2)
       if (min1<min2) then min ← min1 else min ← min2
       if (max1>max2) then max ← max1 else max ← max2
```

Złożoność algorytmu MinMax

Niech $T(n)$ oznacza złożoność algorytmu MinMax. Wtedy:

$$T(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 1 & \text{dla } n = 2 \\ 2T\left(\frac{n}{2}\right) + 2 & \text{dla } n > 2 \end{cases}$$

Dla n będącego potęgą liczby 2 ($n=2^k$) złożoność MinMax jest równa:

$$T(n) = \frac{3}{2}n - 2$$

Obliczenie złożoności:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 =$$

$$= 2\left(2T\left(\frac{n}{4}\right) + 2\right) + 2 = 4T\left(\frac{n}{4}\right) + 4 + 2 =$$

$$= 4\left(2T\left(\frac{n}{8}\right) + 2\right) + 4 + 2 = 8T\left(\frac{n}{8}\right) + 8 + 4 + 2 = \ldots$$

$$= 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \ldots + 2^3 + 2^2 + 2^1 =$$

$$= 2^{k-1}T\left(\frac{2^k}{2^{k-1}}\right) + 2^k - 2 =$$

$$= 2^{k-1} \cdot 1 + n - 2 = n \cdot 2^{-1} + n - 2 = \frac{3}{2}n - 2$$

Część II:

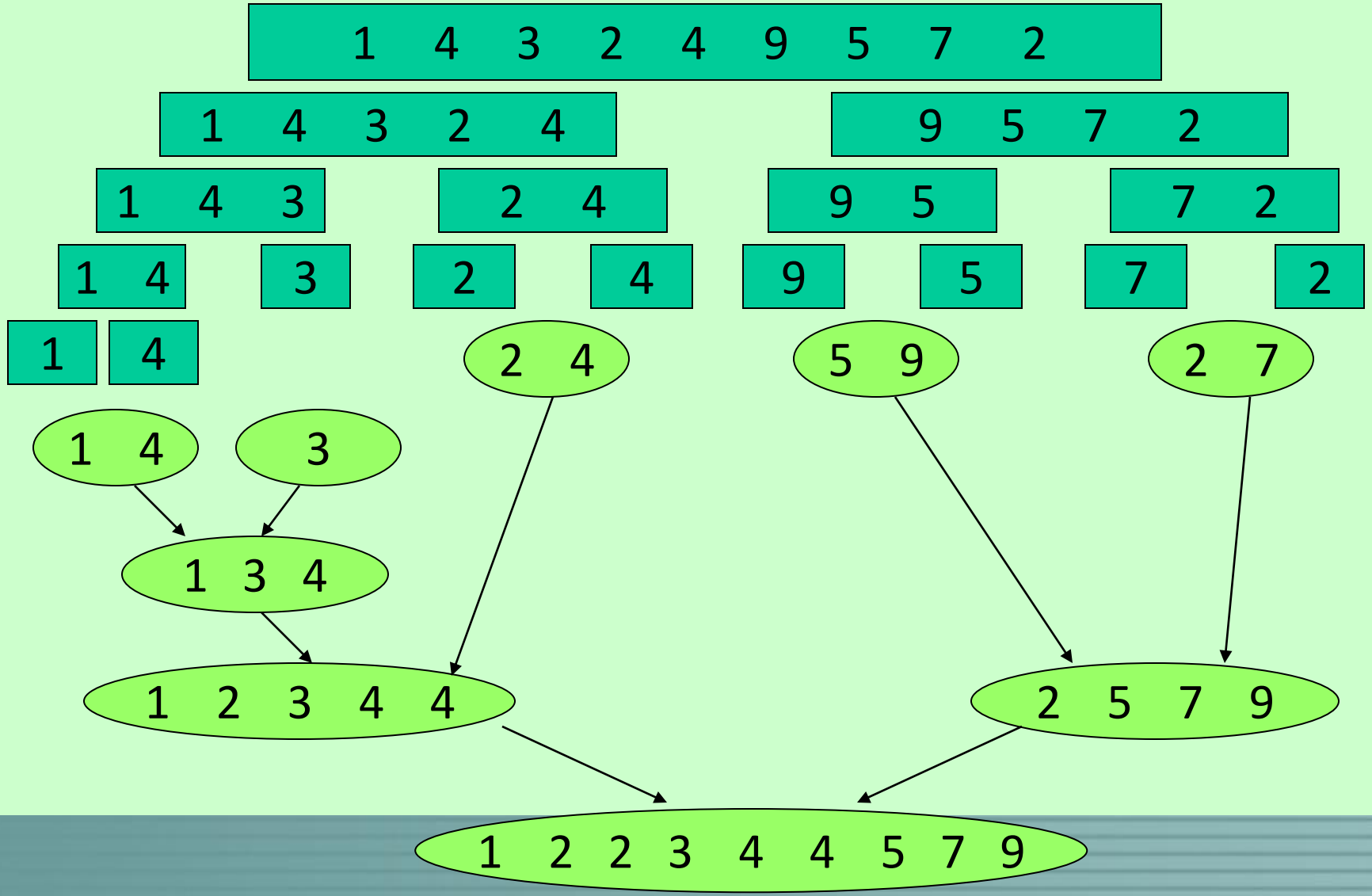
SORTOWANIE PRZEZ SCALANIE

Sortowanie

- przez wybór: $O(n^2)$
- przez wstawianie : $O(n^2)$
- bąbelkowe : $O(n^2)$
- quicksort : $O(n^2)$;
- przez scalanie : $O(nlog_2 n)$

Uwaga: Wszystkie złożoności podane w wersji pesymistycznej! Średnia złożoność quicksort to $O(nlog_2 n)$.

Sortowanie przez scalanie



Schemat działania

Jeśli w ciągu jest jeden element, to nic nie rób.

Jeśli w ciągu jest więcej niż jeden element, to:

- podziel ciąg Z na 2 części: Z_1 oraz Z_2
- wykonaj ten sam algorytm dla Z_1 (posortuj tą samą metodą)
- wykonaj ten sam algorytm dla Z_2 (posortuj tą samą metodą)
- scal Z_1 oraz Z_2 w jeden posortowany ciąg Z

Algorytm MergeSort

Dane: n – liczba elementów
T – n elementowy nieuporządkowany ciąg liczb
p – początek sortowania
k – koniec sortowania

Wynik: T – n-elementowy uporządkowany ciąg liczb

```
MergeSort(T,p,k)
if (k-p=0) then ;
else
    s ←(p+k) div 2
    MergeSort(T,p,s)
    MergeSort(T,s+1,k)
    Merge(T,p,s,k)
```

Złożoność algorytmu MergeSort

Niech $T(n)$ oznacza złożoność algorytmu MergeSort. Wtedy:

$$T(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 1 & \text{dla } n = 2 \\ 2T\left(\frac{n}{2}\right) + n - 1 & \text{dla } n > 2 \end{cases}$$

Dla n będącego potęgą liczby 2 ($n=2^k$) złożoność MergeSort jest równa:

$$T(n) = n \log_2 n - n + 1$$

Obliczenie złożoności:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 =$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2} - 1\right) + n - 1 = 4T\left(\frac{n}{4}\right) + n - 2 + n - 1 =$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + n - 2 + n - 1 =$$

$$= 8T\left(\frac{n}{8}\right) + n - 4 + n - 2 + n - 1 = \ldots$$

$$= 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) + (k-1)n - \left(2^{k-2} + 2^{k-3} + \ldots + 2^1 + 2^0\right) =$$

$$= 2^{k-1}T\left(\frac{2^k}{2^{k-1}}\right) + (k-1)n - 2^{k-1} + 1 =$$

$$= 2^{k-1} \cdot 1 + kn - n - 2^{k-1} + 1 = kn - n + 1 = n\log_2 n - n + 1$$

Nasz problem:

Jak scalić 2 uporządkowane ciągi w 1 uporządkowany ciąg?

KIEDYŚ może to się tu znajdzie….

Część III:

PRZESZUKIWANIE BINARNE

Przeszukiwanie binarne

- wykorzystuje strukturę ciągu: dane są uporządkowane
- wykorzystuje metodę dziel i zwyciężaj
- Liczba elementów w ciągu przeszukiwanym jest połową (jest zmniejszana) w każdym kroku algorytmu

Przeszukiwanie binarne - algorytm

Dane: t – ciąg n-elementowy
e – element, którego szukamy
Wynik: s, 1<=s<=n, taki że t[s]=e lub
s = -1, jeśli element y nie występuje w ciągu t

Algorytm rekurencyjny

```
BinSearch(t,p,k,e)
if (k<p) return -1;
else
     s = (p+k)/2
     if (t[s]==e) return s;
     else
          if (t[s]>e) return BinSearch(t,p,s-1,e)
          else return BinSearch(t,s+1,k,e)
```

Algorytm iteracyjny

```
s = (p+k)/2;
while (k>=p and t[s]!=e)
      if (t[s]>e)
            k = s-1;
      else
            p = s+1;
      s = (p+k)/2;
if (k<p) return -1; else return s;
```

Złożoność algorytmu

Odpowiedź na pytanie o liczbę podziałów ciągu w najgorszym przypadku tzn.
Jak wiele razy musimy odrzucać połowę aktualnego ciągu, aby otrzymać jeden element?

Ciąg *n*-elementowy może być połowiony co najwyżej $\log_2 n$ razy.

Część IV (materiał dodatkowy):

PRZESZUKIWANIE INTERPOLACYJNE

Przeszukiwanie interpolacyjne

- Przeszukiwanie binarne może nie być tak szybkie, jak byśmy oczekiwali; przykład: ”nazwisko na B” w książce telefonicznej
- Jak szukamy:

Przeszukiwanie binarne:

- porównujemy czy element *e* jest większy, mniejszy czy równy elementowi ciągu

Przeszukiwanie interpolacyjne:

- Sprawdzamy i korzystamy z informacji jak bardzo element *e* jest większy, mniejszy od elementu w ciągu

Porównanie

Przeszukiwanie binarne:

$$s \leftarrow \frac{p+k}{2} = p + \frac{1}{2}(k-p)$$

Przeszukiwanie interpolacyjne:

$$s \leftarrow p + \frac{y - t[p]}{t[k] - t[p]}(k-p)$$

Dzielimy ciąg danych, ale próbujemy „strzelić” bliżej części ciągu, którą jesteśmy zainteresowani

Algorytm przeszukiwania interpolacyjnego

Dane, wyniki: jak w przeszukiwaniu binarnym
Metoda: jak w przeszukiwaniu binarnym, z wyjątkiem obliczania *s*

Uwaga: w praktyce komputerowa realizacja przeszukiwania interpolacyjnego nie wygrywa z przeszukiwaniem binarnym:

- dla małych *n* liczba *log n* jest mała
- Inne operacje zwiększają łączną liczbę operacji

Złożoność algorytmu: $O(log_2(log_2 n))$

Twierdzenie o rekurencji uniwersalnej

Jeżeli $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ dla pewnych stałych $a > 0, b > 1$ oraz $d \geq 0$, to

$$T(n) = \begin{cases} O\left(n^d\right), & gdy\ d > \log_b a, \\ O(n^d \log n), & gdy\ d = \log_b a, \\ O\left(n^{\log_b a}\right), & gdy\ d < \log_b a. \end{cases}$$

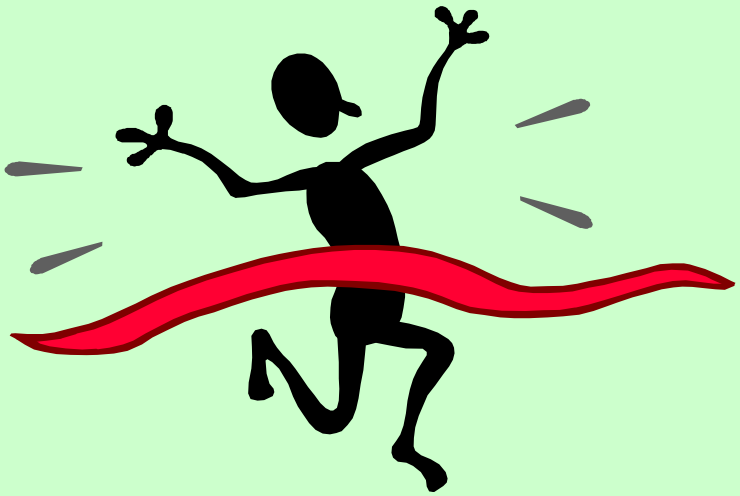
To twierdzenie mówi nam o czasach działania większości programów typu „dziel i zwyciężaj”, z którymi się spotkaliśmy i spotkamy.

Zapamiętaj!

- Jedną z technik projektowania jest metoda dziel i zwyciężaj:
  - problem jest dzielony na takie same lub bardzo podobne podproblemy,
  - liczba podproblemów wynosi co najmniej 2,
  - podproblemy są rozwiązywane na podzbiorach zbioru danych, w których liczba elementów jest niemal jednakowa i stanowi stałą cześć (np. połowę) całego zbioru danych rozwiązywanego problemu.
  - struktura algorytmu opartego na tej technice jest często rekurencyjna.

Informatyka jest jak kurz…
jest wszędzie – i już!

Dziękuję za uwagę!



Beata Laszkiewicz