Algorytmy i struktury danych

Sortowanie, statystyki pozycyjne

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Sortowanie

- dany jest ciąg elementów a_i, $i = 1, \ldots, n$
- należy tak poprzestawiać elementy (znaleźć taką permutację), aby a'_i było nie większe od a'_{i+1} dla $i = 1, \ldots, n-1$
- jeżeli "nie większe" znaczy $\leq$ sortujemy niemalejąco
- jeżeli "nie większe" znaczy $\geq$ sortujemy nierosnąco
- jeżeli "nie większe" znaczy "nie później" sortujemy chronologicznie
- itd.

Sortowanie

- oczekiwany porządek elementów możemy określić także poprzez zdefiniowanie *klucza*
- algorytm sortuje dane według nierosnących wartości klucza
- jeżeli kluczem jest a_i sortujemy niemalejąco
- jeżeli kluczem jest $-a_i$ sortujemy nierosnąco
- jeżeli kluczem struktury `Osoba` jest pole `wzrost` sortujemy według wzrostu (jeżeli −`wzrost`, to sortujemy według nierosnącego wzrostu)
- itd.

Sortowanie

- sortowanie stabilne
  - zachowuje porządek elementów o takich samych kluczach
  - np. sortujemy
    `(22, "Ewa"), (21, "Ala"), (21, "Ola"), (23, "Ula")`
  - jeżeli sortowanie jest stabilne otrzymamy:
    `(21, "Ala"), (21, "Ola"), (22, "Ewa"), (23, "Ula")`
  - jeżeli sortowanie nie jest stabilne możemy otrzymać:
    `(21, "Ola"), (21, "Ala"), (22, "Ewa"), (23, "Ula")`

Sortowanie

- sortowanie w miejscu
  - potrzebuje mniej niż $O(n)$ dodatkowej pamięci
  - całe sortowanie odbywa się w przekazanej do posortowania tablicy

Algorytmy sortowania

- algorytmy proste — $O(n^2)$, łatwe do implementacji;
  nie nadają się do danych liczniejszych niż 15–30
- algorytmy szybkie — $O(n \log n)$, trudniejsze do implementacji
- algorytmy liniowe — $O(n)$; do specyficznych zestawów danych
- algorytmy “inne”

Sortowanie bąbelkowe

- sortowanie w miejscu
- sortowanie stabilne
- w każdym etapie jeden, najcięższy "bąbelek" opada na dno (lub najlżejszy wypływa na wierzch)
- wykonujemy n etapów
- bąbelki gromadzą się na dnie (na końcu tablicy)
- złożoność $O(n^2)$

Prosta implementacja

```
1: for i = 1 do n do
2:         for j = 2, ..., n do
3:                 if mniejsze(a_j, a_{j-1}) then
4:                         zamień miejscami a_{j-1} i a_j
5:                 end if
6:         end for
7: end for
```

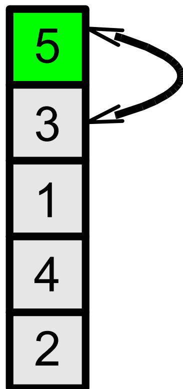
Prosta implementacja

```
1: for i = 1 do n do
2:         for j = 2, ..., n do
3:                 if KLUCZ(a_j) < KLUCZ(a_{j-1}) then
4:                         zamień miejscami a_{j-1} i a_j
5:                 end if
6:         end for
7: end for
```

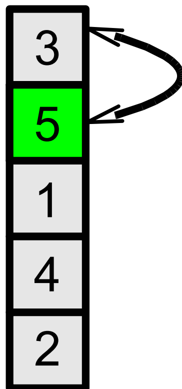
Prosta implementacja

```
1: for i = 1 do n do
2:     for j = 2, ..., n do
3:         if a_j < a_{j-1} then
4:             zamień miejscami a_{j-1} i a_j
5:         end if
6:     end for
7: end for
```

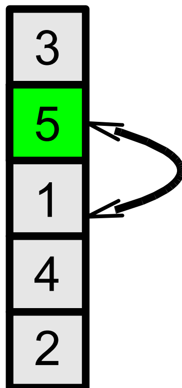
Sortowanie bąbelkowe



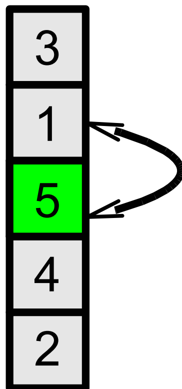
Sortowanie bąbelkowe



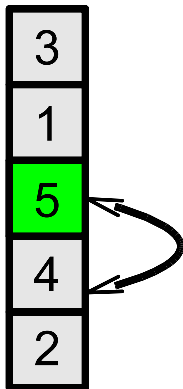
Sortowanie bąbelkowe



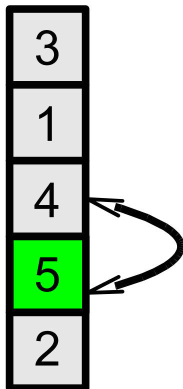
Sortowanie bąbelkowe



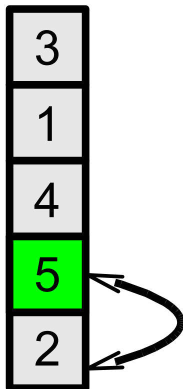
Sortowanie bąbelkowe



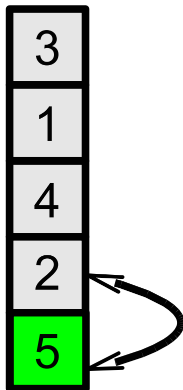
Sortowanie bąbelkowe



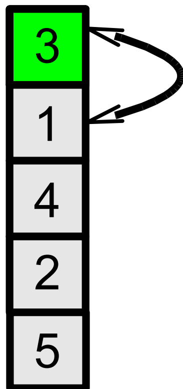
Sortowanie bąbelkowe



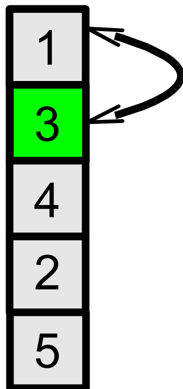
Sortowanie bąbelkowe



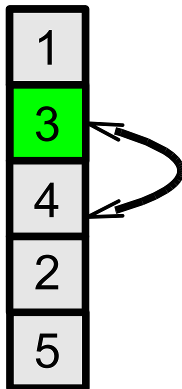
Sortowanie bąbelkowe



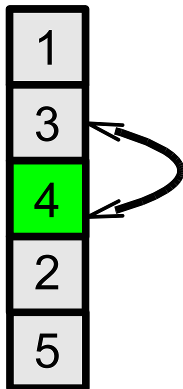
Sortowanie bąbelkowe



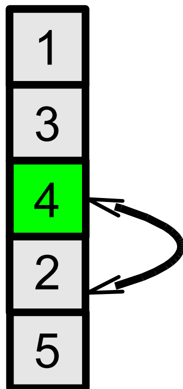
Sortowanie bąbelkowe



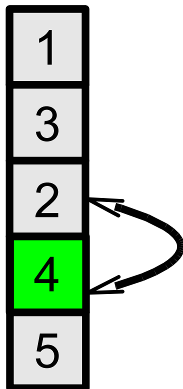
Sortowanie bąbelkowe



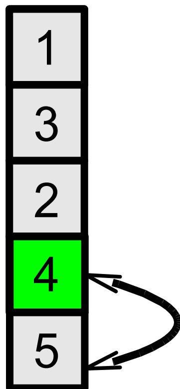
Sortowanie bąbelkowe



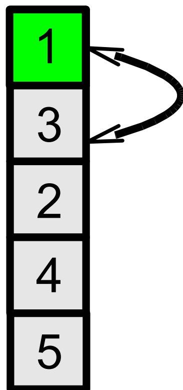
Sortowanie bąbelkowe



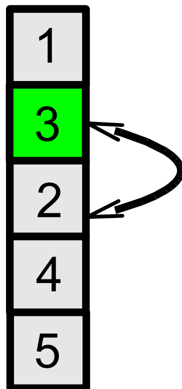
Sortowanie bąbelkowe



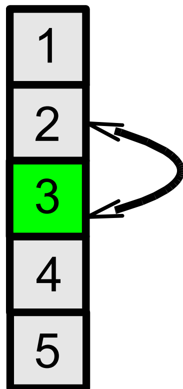
Sortowanie bąbelkowe



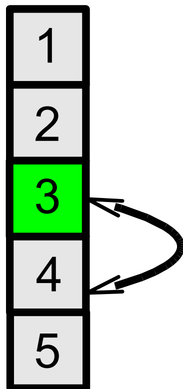
Sortowanie bąbelkowe



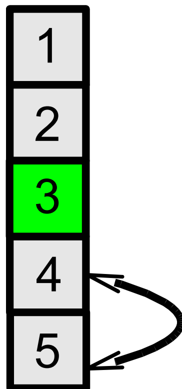
Sortowanie bąbelkowe



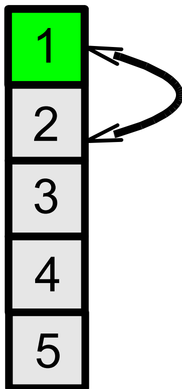
Sortowanie bąbelkowe



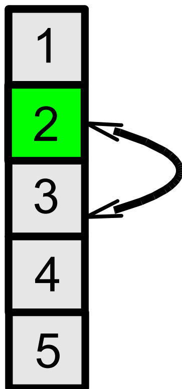
Sortowanie bąbelkowe



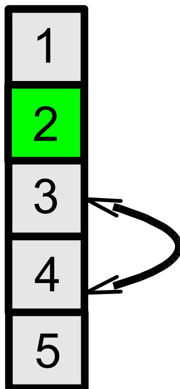
Sortowanie bąbelkowe



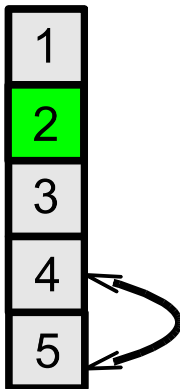
Sortowanie bąbelkowe



Sortowanie bąbelkowe



Sortowanie bąbelkowe



Lepsza implementacja

- zauważmy, że po i-tym kroku, i ostatnich elementów jest już na swoim miejscu

for $i = n, \ldots, 2$ **do**
  for $j = 2, \ldots, i$ **do**
    if $a_j < a_{j-1}$ **then**
      zamień miejscami a_{j-1} i a_j
    end if
  end for
end for

- oszczędzamy połowę czasu: zamiast $n \cdot (n-1)$ kroków mamy $1 + 2 + 3 + \ldots, +n - 1 = \frac{n \cdot (n-1)}{2}$

Jeszcze lepsza implementacja

- jeżeli zdarzy się, że w k-tym kroku wszystko jest już posortowane, możemy przerwać algorytm
- skąd będziemy wiedzieć? — w wewnętrznej pętli nie wystąpiła żadna zamiana
- niestety, złożoność ciągle $O(n^2)$

Jeszcze lepsza implementacja

for $i = n, \ldots, 2$ **do**
  $zmiana = false$
  for $j = 2, \ldots, i$ **do**
    if $a_j < a_{j-1}$ **then**
      zamień miejscami a_{j-1} i a_j
      $zmiana = true$
    end if
  end for
  if $zmiana = false$ **then**
    break
  end if
end for

Sortowanie przez wstawianie

- zaczynamy od pustego posortowanego ciągu
- kolejne elementy wstawiamy w odpowiednie miejsce w dotychczasowym posortowanym ciągu
- powtarzamy, aż wszystkie elementy wstawimy na swoje miejsce
- złożoność $O(n^2)$, ale wykonamy mniej zapisów do pamięci
- algorytm podobny do "sortowania" ręki podczas gry w karty

Sortowanie przez wstawianie

1: **for** $i = 2, \ldots, n$ **do**
2: $\quad t = a_i$
3: $\quad j = i - 1$
4: $\quad$ **while** $j > 0$ i $t < a_j$ **do**
5: $\quad\quad a_{j+1} = a_j$
6: $\quad\quad j = j - 1$
7: $\quad$ **end while**
8: $\quad a_{j+1} = t$
9: **end for**

- można odpowiedniego miejsca poszukać binarnie, ale nie poprawi to złożoności (musimy zrobić miejsce na wstawienie elementu)

Sortowanie przez wybór

- w każdym kroku wybieramy najmniejszy element z ciągu i zamieniamy go z jego pierwszym elementem
- pomniejszamy ciąg o jeden (najmniejszy) element i powtarzamy operację

Sortowanie przez wybór

```
1: for i = 1, ..., n − 1 do
2:         m = i
3:         for j = i + 1, ... n do
4:                 if a_j < a_m then
5:                         m = j
6:                 end if
7:         end for
8:         if m ≠ i then
9:                 zamień miejscami a_m i a_i
10:        end if
11: end for
```

- $O(n^2)$ porównań, ale tylko $O(n)$ operacji kopiowania elementu

Sortowanie szybkie

- wybieramy pewien element z tablicy
- wszystkie elementy mniejsze od niego umieszczamy przed nim w tablicy
- wszystkie elementy większe od niego umieszczamy za nim w tablicy
- sortujemy (rekurencyjnie) wszystkie elementy mniejsze
- sortujemy (rekurencyjnie) wszystkie elementy większe
- w miejscu, nie jest stabilne

1: $l = left$, $p = right$
2: $m =$ `wybierz`(a, l, p)
3: **while** $l < p$ **do**
4:     **while** $l < m$ **and** $a_l < a_m$ **do** $l = l + 1$
5:     **while** $m < p$ **and** $a_m < a_p$ **do** $p = p + 1$
6:     zamień miejscami a_l z a_p
7:     **if** $l = m$ **then** $m = p$ **else if** $m = p$ **then** $m = l$
8:     **if** $l < m$ **then** $l = l + 1$
9:     **if** $m < p$ **then** $p = p - 1$
10: **end while**
11: **if** $left < m$ **then** `sortuj`$(a, left, m)$
12: **if** $m + 1 < right$ **then** `sortuj`$(a, m + 1, right)$

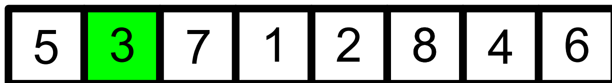
Sortowanie szybkie

- jeżeli nieszczęśliwie będziemy wybierać elementy do podziału (tj. element będzie nam dzielił wartości bardzo nierównomiernie) czas działania wydłuży się
- jak dobrze wybrać element do podziału?
  - losowo (nie ma przypadku pesymistycznego)
  - średni element z pierwszych trzech
  - mediana (ale jest to skomplikowane)

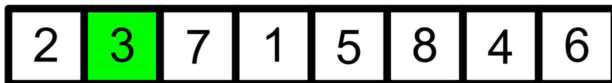
Sortowanie szybkie

- złożoność w średnim przypadku $O(n \log n)$
- złożoność w pesymistycznym przypadku $O(n^2)$ (za każdym razem dzielimy na przedział pusty i $n-1$ elementowy)
- złożoność w pesymistycznym przypadku $O(n \log n)$ jeżeli użyjemy mediany (w praktyce nie opłaca się)
- nie jest stabilny
- działa w miejscu

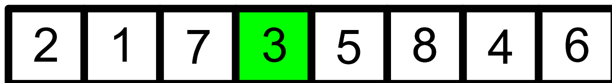
Sortowanie szybkie



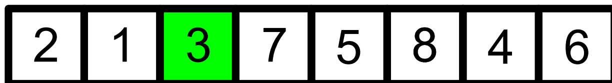
Sortowanie szybkie



Sortowanie szybkie



Sortowanie szybkie



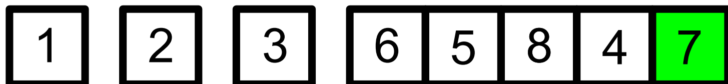
Sortowanie szybkie

Sortowanie szybkie

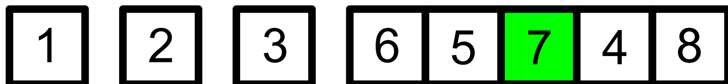
Sortowanie szybkie

Sortowanie szybkie

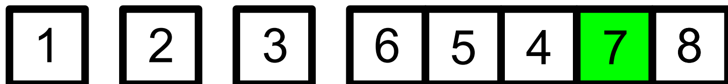
Sortowanie szybkie



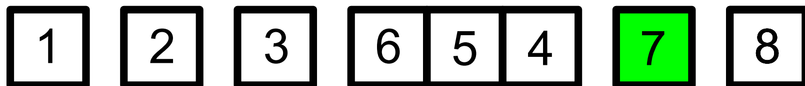
Sortowanie szybkie



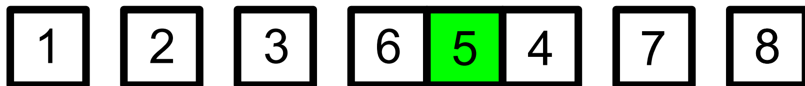
Sortowanie szybkie



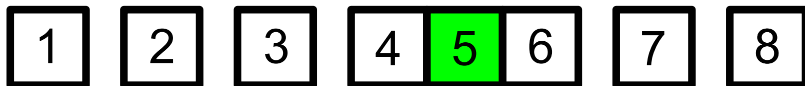
Sortowanie szybkie



Sortowanie szybkie



Sortowanie szybkie



Sortowanie szybkie

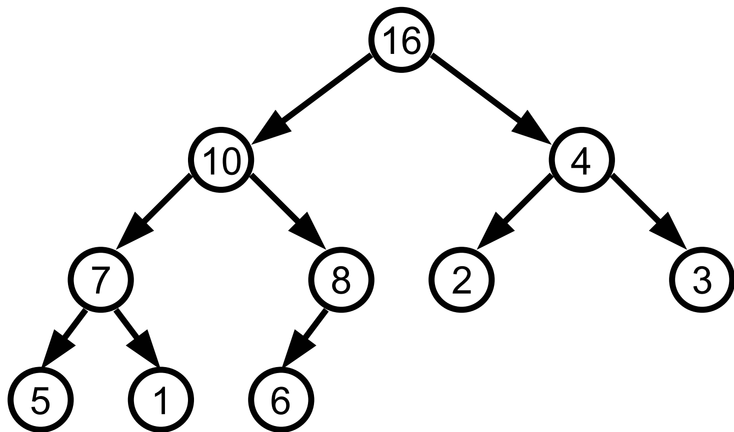
Sortowanie kopcowe

- algorytm o złożoności pesymistycznej $O(n \log n)$
- działa w miejscu
- nie jest stabilny
- wykorzystuje kopiec

Kopiec

- drzewo, w którym każdy węzeł może mieć dwoje, jednego lub zero potomków
- własność kopca: wartości w węzłach potomnych nie mogą być większe od wartości w węźle rodzicielskim (alternatywnie: nie mogą być mniejsze)
- umożliwia:
  - szybkie znalezienie największego (najmniejszego) elementu ($O(1)$)
  - dość szybkie usunięcie największego (najmniejszego) elementu ($O(\log n)$)
  - dość szybkie dodanie elementu ($O(\log n)$)
  - możemy w czasie liniowym ($O(n)$) z tablicy zbudować kopiec

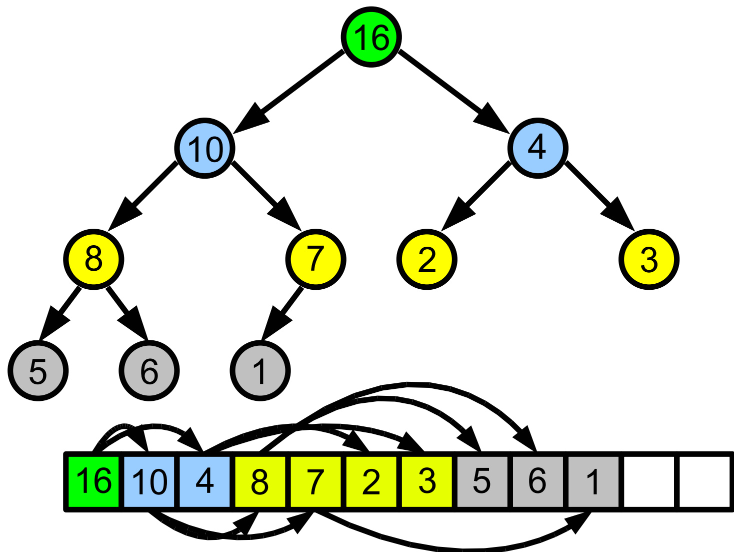
Kopiec



Kopiec

- mimo, że kopiec jest drzewem, zazwyczaj implementujemy go w tablicy
- pierwszy element (o numerze 1) jest korzeniem
- lewy potomek węzła o indeksie i ma indeks $2i$
- prawy potomek węzła o indeksie i ma indeks $2i + 1$
- rodzic węzła o indeksie i ma indeks $i/2$
- indeksując od 0, będzie to $2i + 1$, $2i + 2$, $(i - 1)/2$

Kopiec



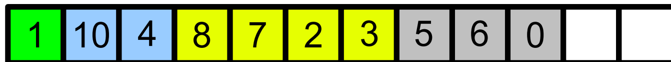
Przywracanie własności kopca

- lewy i prawy potomek pewnego węzła i są korzeniami poprawnych kopców
- dla węzła i własność kopca może nie być zachowana
- jak ją przywrócić?
- naprawa kopca w dół

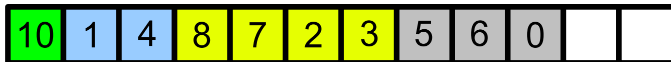
Przywracanie własności kopca

1: $l = 2 * i$, $p = 2 * i + 1$
2: **if** $l \leq n$ i $a_i < a_l$ **then**
3: $najw = l$
4: **else**
5: $najw = i$
6: **end if**
7: **if** $r \leq n$ i $a_{najw} < a_r$ **then**
8: $najw = r$
9: **end if**
10: **if** $najw \neq i$ **then**
11: zamień miejscami a_i i a_{najw}
12: przywróć własność kopca dla węzła $najw$
13: **end if**

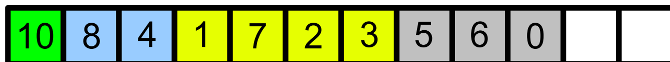
Przywracanie własności kopca



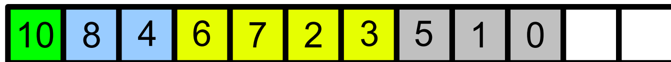
Przywracanie własności kopca



Przywracanie własności kopca



Przywracanie własności kopca



Największy element

- z własności kopca wynika, że największy element będzie zawsze na pierwszym miejscu tablicy

Usunięcie największego elementu

- "usuwamy" element z pierwszego miejsca tablicy (a_1)
- na jego miejsce wstawiamy element z końca tablicy (a_n)
- zmniejszamy rozmiar kopca: $n = n - 1$
- przywracamy własność kopca dla korzenia (a_1)

Usunięcie największego elementu

Usunięcie największego elementu

Usunięcie największego elementu

Usunięcie największego elementu

Usunięcie największego elementu

Usunięcie największego elementu

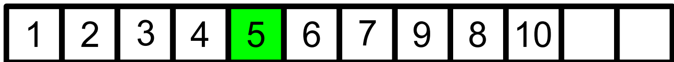
Budowa kopca z tablicy

- otrzymujemy tablicę $a_1, \ldots, a_n$
- mamy z niej zbudować poprawny kopiec
- elementy $a_{n/2+1}, \ldots, a_n$ są poprawnymi korzeniami kopców (nie mają żadnych potomków)
- rozpoczynamy od elementu $a_{n/2}$ i przywracamy dla niego własność kopca
- przesuwamy się do elementu wcześniejszego i dla niego przywracamy własność kopca
- powtarzamy proces aż do korzenia (elementu a_1)
- złożoność $O(n)$

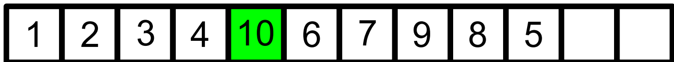
Budowa kopca z tablicy

1: **for** $i = n/2, \ldots, 1$ **do**
2: przywróć własność kopca dla węzła i
3: **end for**

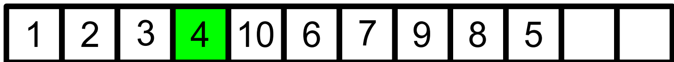
Budowa kopca z tablicy



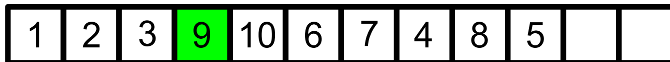
Budowa kopca z tablicy



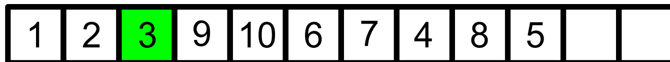
Budowa kopca z tablicy



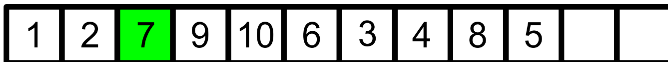
Budowa kopca z tablicy



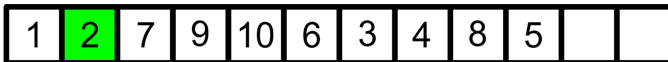
Budowa kopca z tablicy



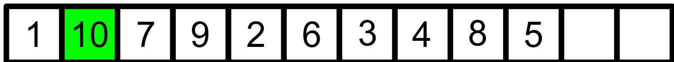
Budowa kopca z tablicy



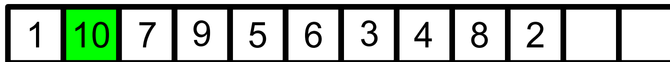
Budowa kopca z tablicy



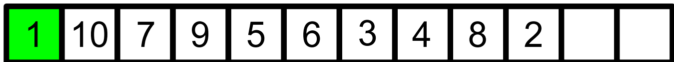
Budowa kopca z tablicy



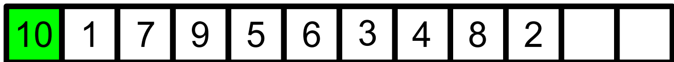
Budowa kopca z tablicy



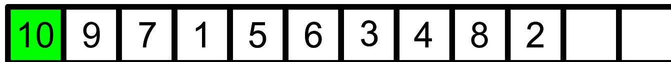
Budowa kopca z tablicy



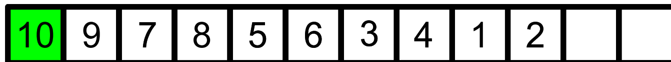
Budowa kopca z tablicy



Budowa kopca z tablicy



Budowa kopca z tablicy



Sortowanie kopcowe

- w tablicy do posortowania $a_1, \ldots, a_n$ budujemy kopiec
- usuwamy największy element (zapamiętując jego wartość)
- kopiec zmniejszył się o 1 element — zrobiło się miejsce na końcu tablicy
- w to miejsce wstawiamy usunięty element (ostatnie miejsce w tablicy)

Sortowanie kopcowe

- usuwamy z kopca kolejny największy element
- kopiec ponownie zmniejszył się o 1 element — zrobiło się kolejne wolne miejsce (przedostatnie pole tablicy)
- wstawiamy w to miejsce usunięty element
- powtarzamy operację aż usuniemy wszystkie elementy z kopca
- otrzymamy posortowaną tablicę

Kolejka priorytetowa

- kopiec można również wykorzystać do implementacji kolejki priorytetowej
- jest to struktura o właściwościach podobnych do zwykłej kolejki
- do zwykłej kolejki możemy dodawać elementy i je wyjmować — najpierw wyjmujemy najwcześniej dodany
- z kolejki priorytetowej wyjmujemy najpierw najważniejszy element (o najwyższym priorytecie)

Kolejka priorytetowa

- mając kopiec, potrafimy z niego szybko usunąć najważniejszy (największy) element
- jak dodać element e do kopca?
- naprawa kopca w górę:

```
1: zwiększ rozmiar kopca o 1 (n = n + 1)
2: i = n
3: while i > 1 i e > a_{i/2} do
4:         a_i = a_{i/2}
5:         i = i/2
6: end while
7: a_i = e
```

Kopiec

- jeżeli mamy wskazany element w kopcu, jak go usunąć?

Kopiec

- jeżeli mamy wskazany element w kopcu, jak go usunąć?
- zamieniamy go miejscami z ostatnim i zmniejszamy rozmiar o jeden

Kopiec

- jeżeli mamy wskazany element w kopcu, jak go usunąć?
- zamieniamy go miejscami z ostatnim i zmniejszamy rozmiar o jeden
- jeżeli wstawiony element jest większy od swojego rodzica, naprawiamy kopiec w górę
- w przeciwnym razie naprawiamy kopiec w dół

Kopiec

- gdy potrzebujemy kopca minimalizującego (takiego który pozwala na szybkie poznanie wartości najmniejszej) zamieniamy wszystkie porównania elementów z $<$ na $>$

Sortowanie Shella

- sortowanie przez wstawianie działa szybko dla danych "prawie" uporządkowanych
- sortowany ciąg dzielimy na podciągi, wybierając co k-ty element
- sortujemy wszystkie podciągi z krokiem k
- zmniejszamy krok
- powtarzamy operacje aż dojdziemy z krokiem do 1

Sortowanie Shella

- dla $k = n/2^i$ pesymistyczna złożoność wynosi $O(n^2)$
- dla $k = 2^i - 1$ pesymistyczna złożoność wynosi $O(n^{3/2})$
- dla $k = 4^i - 3 \cdot 2^i + 1$ pesymistyczna złożoność to $O(n^{4/3})$
- dla $k = 2^i \cdot 3^j$ pesymistyczna złożoność wynosi $O(n \log^2 n)$ (lepiej być nie może)

Sortowanie list

- z przedstawionych algorytmów jedynie sortowanie bąbelkowe nadaje się do sortowania list
- do list dwukierunkowych nadaje się też sortowanie przez wstawianie
- jak szybko ($O(n \log n)$) posortować listę?
- można ją przepisać do tablicy, posortować tablicę, na nowo utworzyć listę, ale jest to pamięcio- i czasochłonne
- rozwiązanie: sortowanie przez scalanie

Sortowanie przez scalanie

- dzielimy listę na 2 prawie równe części (różniące się długością o co najwyżej jeden)
- sortujemy (rekurencyjne) obie listy
- posortowane listy scalamy w jedną

Scalanie list

- chcemy scalić dwie uporządkowane listy w jedną, także uporządkowaną
- wybieramy mniejszą głowę, lub jedyną dostępną, gdy jedna z list się skończy
- dołączamy na koniec nowej listy
- operację powtarzamy

Scalanie list

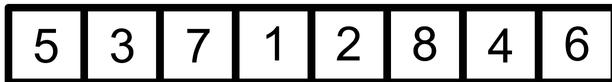
1: **if** $a.dana < b.dana$ **then** $glowa = a$; $a = a.next$
2: **else** $glowa = b$; $b = b.next$
3: $koniec = glowa$
4: **while** $a \neq nullptr$ i $b \neq nullptr$ **do**
5: **if** $a.dana < b.dana$ **then**
6: $koniec.next = a$
7: $a = a.next$
8: **else**
9: $koniec.next = b$
10: $b = b.next$
11: **end if**
12: $koniec = koniec.next$
13: **end while**
14: **if** $a \neq nullptr$ **then** $koniec.next = a$
15: **else if** $b \neq nullptr$ **then** $koniec.next = b$
16: **else** $koniec.next = nullptr$

Sortowanie przez scalanie

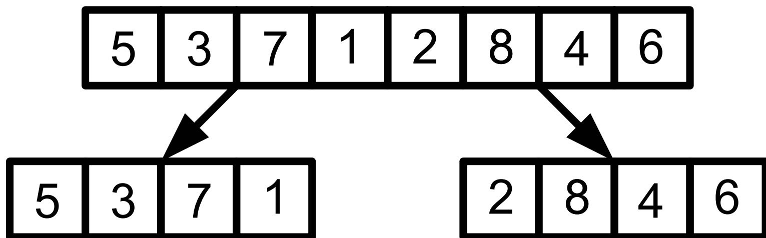
- SORTUJLISTE(*lista*)

```
1:  if lista.next = nullptr then return lista
2:  b = lista
3:  t = lista
4:  while t.next ≠ nullptr i t.next.next ≠ nullptr do
5:          b = b.next
6:          t = t.next.next
7:  end while
8:  t = b.next
9:  b.next = nullptr
10: b = t
11: a = SORTUJLISTE(lista)
12: b = SORTUJLISTE(b)
13: wynik = SCAL(a, b)
14: return wynik
```

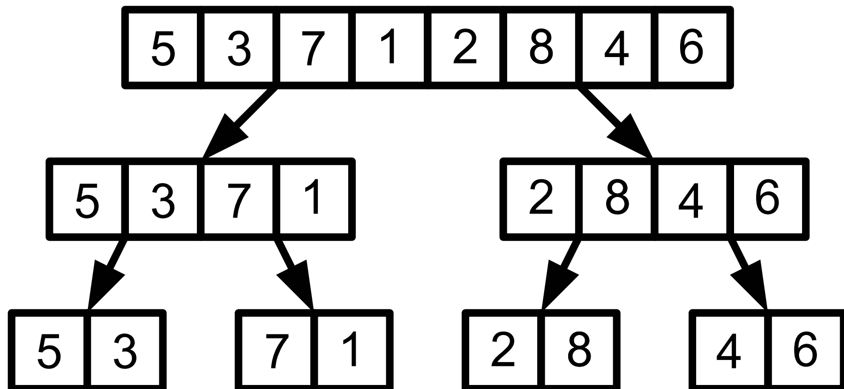
Sortowanie przez scalanie



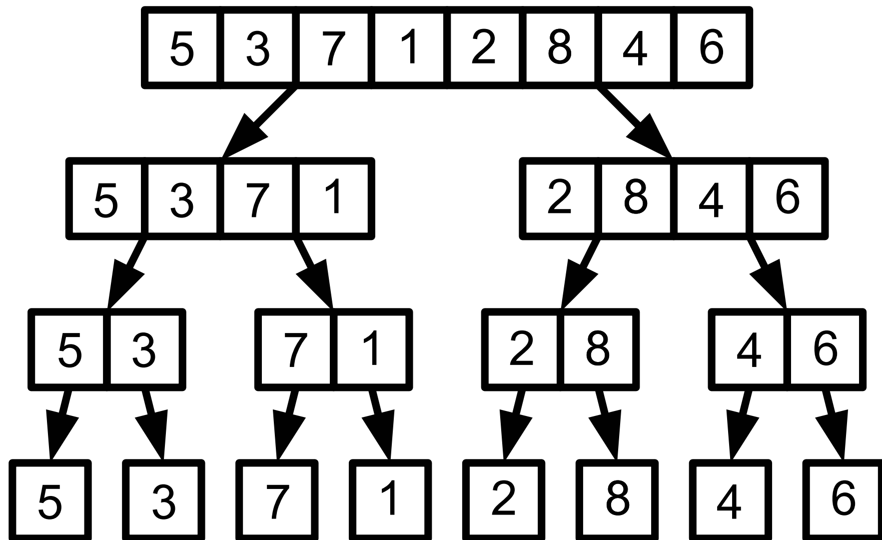
Sortowanie przez scalanie



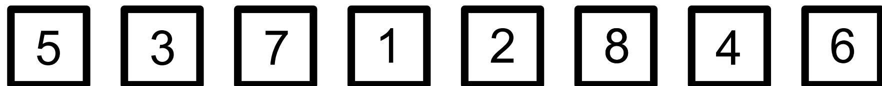
Sortowanie przez scalanie



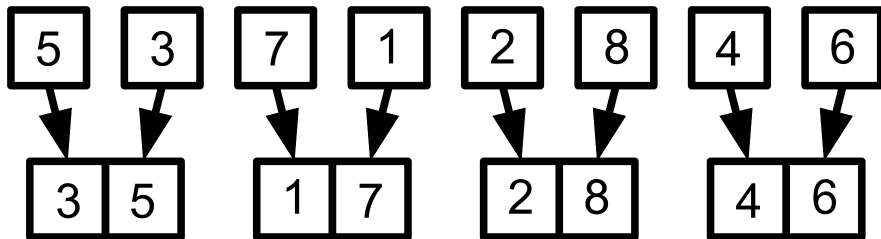
Sortowanie przez scalanie



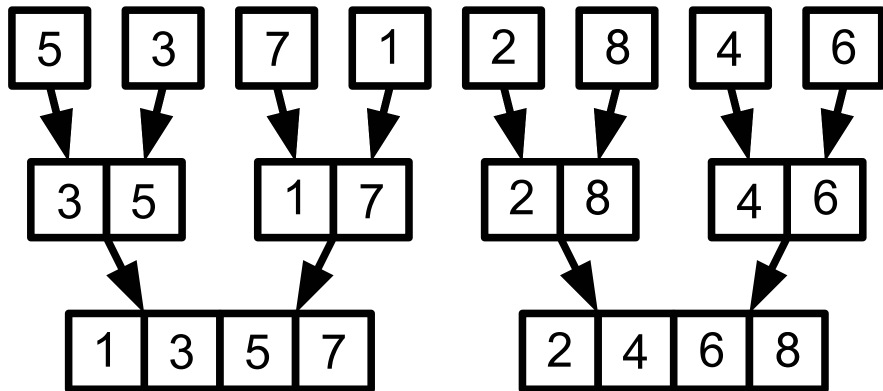
Sortowanie przez scalanie

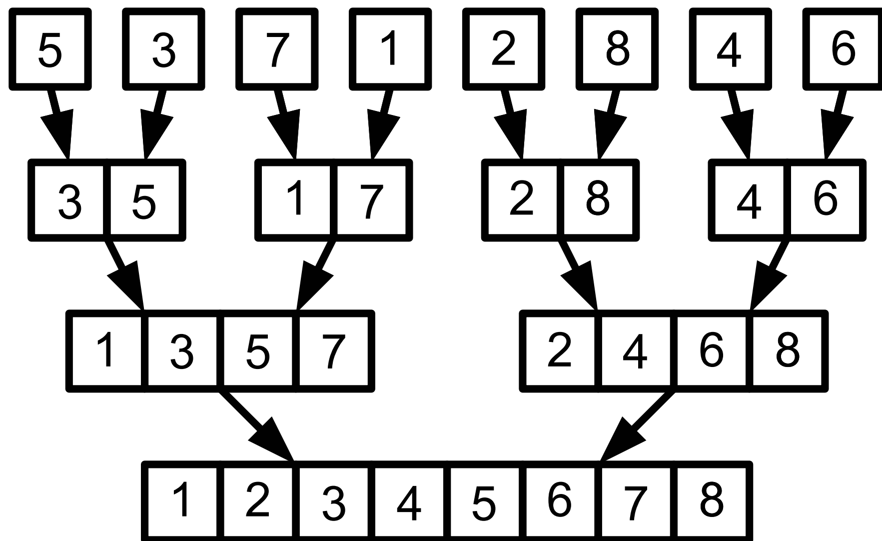


Sortowanie przez scalanie



Sortowanie przez scalanie




5
3
7
1
2
8
4
6
3 5
1 7
2 8
4 6
1 3 5 7
2 4 6 8
1 2 3 4 5 6 7 8

Sortowanie bez porównań

- algorytmy sortowania wykorzystujące porównanie nie mają szans działać szybciej niż $O(n \log n)$
- pewne specyficzne kategorie danych można jednak posortować szybciej

Sortowanie przez zliczanie

- sortujemy liczby z niewielkiego zakresu $(0, \ldots, k)$
- liczymy ilość wystąpień każdej z liczby
- wyznaczamy pozycje w których rozpoczynają się kolejne ciągi takich samych liczb
- rekonstruujemy posortowaną tablicę
- wymaga $O(n+k)$ czasu i $O(n+k)$ dodatkowej pamięci (w niektórych implementacjach wystarczy $O(k)$ dodatkowej pamięci)
- aby sortowanie było stabilne, należy "odwrócić" ostatnią pętlę w kodzie

Sortowanie przez zliczanie

1: $ile[i] =$ dla $i = 0, \ldots, k$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad ile[a_i] += 1$
4: **end for**
5: **for** $i = 1, \ldots, k$ **do**
6: $\quad ile[i] = ile[i-1] + ile[i]$
7: **end for**
8: **for** $i = 1, \ldots, n$ **do**
9: $\quad p = ile[a_i]$
10: $\quad ile[a_i] -= 1$
11: $\quad b_p = a_i$
12: **end for**

Sortowanie kubełkowe

- sprawdza się dla jednostajnie rozłożonych, niezbyt dużych zakresów danych
- złożoność $O(n)$, pesymistyczna $O(n^2)$

1: podziel dany przedział liczb na n podprzedziałów (kubełków) o równej długości
2: przypisz sortowane dane do odpowiednich kubełków
3: posortuj dane w każdym z kubełków (np. przez wstawianie)
4: przepisz to tablicy wynikowej zawartości kolejnych niepustych kubełków

Sortowanie pozycyjne

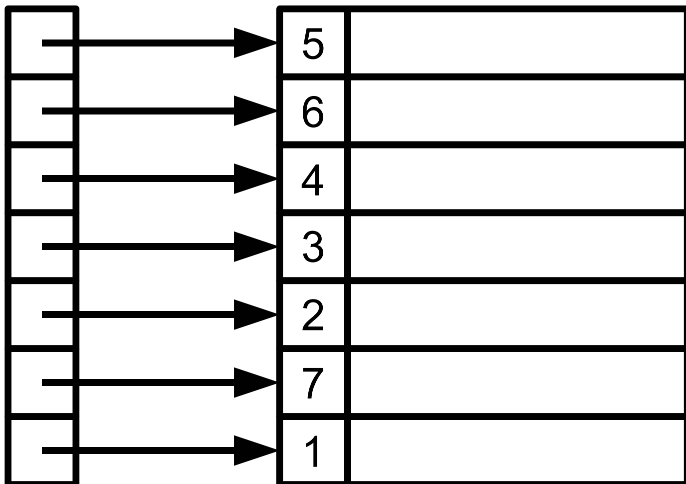
- nadaje się do sortowania danych składających się z niedużej liczby "liter" (np. słowa, liczby)
- złożoność czasowa $O(d(n+k))$, pamięciowa $O(n+k)$ gdzie d to liczba "liter" w wyrazach zaś k jest liczbą "liter" w alfabecie
- niech 1 będzie numerem "najstarszej" litery, n — najmłodszej

```
1: for i = n, ..., 1 do
2:        posortuj dane stabilnie według i-tej litery
          (np. przez zliczanie)
3: end for
```

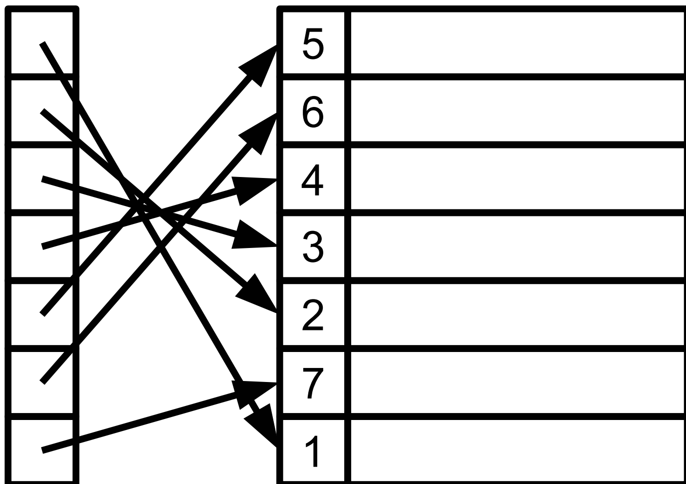
Sortowanie wskaźników

- jeżeli mamy tablicę dużych struktur bądź obiektów, bezpośrednie sortowanie jej może okazać się czasochłonne (duża liczba przepisywania dużych obiektów w pamięci)
- rozwiązaniem jest sortowanie wskaźników (lub indeksów)
- tworzymy tablicę wskaźników na kolejne elementy naszej listy
- sortujemy tablicę wskaźników (kluczem jest tu klucz obiektu wskazywanego przez wskaźnik!)
- sortowanie przestawia w pamięci jedynie wskaźniki (liczby)

Sortowanie wskaźników



Sortowanie wskaźników



Algorytmy sortowania

- sortowanie bąbelkowe
- sortowanie przez wstawianie
- sortowanie gnoma
- sortowanie przez wybór
- sortowanie biblioteczne
- sortowanie szybkie
- sortowanie introspektywne
- sortowanie koktajlowe

Algorytmy sortowania

- sortowanie Shella
- sortowanie grzebieniowe
- sortowanie kopcowe
- sortowanie przez scalanie
- sortowanie kubełkowe
- sortowanie pozycyjne
- sortowanie przez zliczanie
- bogosort (stupidsort)
- ...

Statystyki pozycyjne

- i-tą statystyką pozycyjną pewnego ciągu nazywamy liczbę, która po jego posortowaniu znalazła by się na i-tym miejscu
- np. minimum jest pierwszą statystyką pozycyjną
- np. maksimum jest n-tą statystyką pozycyjną
- np. mediana jest $n/2$-tą statystyką pozycyjną
- w czasie $O(n \log n)$ i-ą statystykę pozycyjną można znaleźć sortując ciąg i patrząc na i-tą pozycję

Znajdowanie maksimum

- maksimum (minimum) możemy znaleźć szybciej ($O(n)$):

1: $max = a_1$
2: **for** $i = 2, \ldots, n$ **do**
3:     **if** $a_i > max$ **then**
4:         $max = a_i$
5:     **end if**
6: **end for**

Znajdowanie maksimum i minimum

- jeżeli chcemy znaleźć równocześnie i maksimum i minimum, możemy zrobić to w jednym przebiegu:

1: $max = a_1$
2: $min = a_1$
3: **for** $i = 3, \ldots, n$ **step** 2 **do**
4: **if** $a_{i-1} < a_i$ **then**
5: $max = MAX(a_i, max)$
6: $min = MIN(a_{i-1}, min)$
7: **else**
8: $max = MAX(a_{i-1}, max)$
9: $min = MIN(a_i, min)$
10: **end if**
11: **end for**

i-ta statystyka pozycyjna

- jeżeli i jest bardzo bliskie 1 lub n (2, 3, ew. 4) jesteśmy w stanie sobie jeszcze poradzić, modyfikując powyższy algorytm
- ale jak znaleźć i-tą statystykę pozycyjną w czasie $O(n)$ dla dowolnego i?
- szybciej niż $O(n)$ się nie da

i-ta statystyka pozycyjna

1: podziel elementy $a_1, \ldots, a_n$ na *podloga*$(n/5)$ grup po 5 elementów i jedną grupę z pozostałymi elementami
2: wyznacz medianę z każdej grupy (np. sortując; z grupy o parzystej liczbie elementów należy wybrać większą z median)
3: rekurencyjnie znajdź medianę zbioru median $\to x$
4: podziel tablicę a względem x (tak jak w sortowaniu szybkim) niech k będzie liczbą elementów w lewej częsci a $n - k$ liczbą elementów w prawej części
5: **if** $i \leq k$ **then**
6: znajdź rekurencyjnie i-tą statystykę pozycyjną lewej części przedziału
7: **else**
8: znajdź rekurencyjnie $i - k$-tą statystykę pozycyjną prawej części przedziału
9: **end if**