

Problemy porządkowe

Problemy porządkowe – zadania

- Problemy porządkowe to zbiór różnych zadań obliczeniowych związanych z porządkowaniem zbioru danych i wyszukiwaniem informacji na takim zbiorze.
- Rodzaje zadań porządkowych:
 - sortowanie
 - scalanie
 - podział
 - wyszukiwanie wartości w zbiorze
 - element minimalny/maksymalny
 - k-ty co do wielkości element w zbiorze

Problemy porządkowe – dane

- Dane w problemach porządkowych pochodzą z określonego **uniwersum z porządkiem liniowym** – zbiór z relacją porządkującą $\leq$ (przechodnia, antysymetryczna i zwrotna i dotyczącą każdej pary elementów z tego zbioru).
- Przykłady zbiorów z porządkiem liniowym:
 - zbiór liczb całkowitych z relacją $\leq$
 - zbiór liczb rzeczywistych z relacją $\leq$
 - zbiór ciągów znakowych z porządkiem leksykograficznym $\leq_{\text{lex}}$

Problemy porządkowe – tablica

- Dane dla rozważanych problemów porządkowych są przechowywane w **tablicy**.
- Tablicę n -elementową oznaczamy przez $A[0...n-1]$ (elementy w tablicy są indeksowane od 0).
- Niepusty fragment tablicy z danymi oznaczamy $T[a...b]$, gdzie $0 \leq a \leq b < n$ (fragment taki zawiera $b - a + 1$ elementów).

Stabilność i działanie w miejscu

- Algorytm jest **stabilny** (ang. stable), gdy dane o takich samych wartościach zachowują pierwotne uporządkowanie.
- Algorytm **działa w miejscu** (ang. in-place), jeśli korzysta tylko z pewnej stałej liczby $O(1)$ dodatkowych komórek pamięci do rozwiązania zadania (co wyklucza głęboką rekurencję).

Inwersje

- Dwa elementy a_i i a_j w ciągu $A = (a_0, a_1, \dots, a_{n-1})$ są w **inwersji**, gdy nie są uporządkowane, czyli: $a_i > a_j$ dla $i < j$.
- Liczba inwersji występująca w ciągu jest **miarą uporządkowania** tego ciągu.
- W każdym ciągu może być od 0 (ciąg uporządkowany) do $n(n-1)/2$ (ciąg odwrotnie uporządkowany) inwersji.

Techniki rozwiązywania zadań obliczeniowych

Dziel i zwyciężaj oraz redukcja

Dziel i zwyciężaj

- Technika rozwiązywania problemów metodą „dziel i zwyciężaj” została zastosowana przez Napoleona w bitwie pod Austerlitz w 1805 r.
- Problem obliczeniowy wykazuje **własność podstruktury**, jeśli jego rozwiązanie można wyrazić funkcją jednego lub **kilku** rozwiązań jego podproblemów.
- Aby zastosować **technikę dziel i zwyciężaj** problem obliczeniowy musi wykazywać własność podstruktury.
- Technika dziel i zwyciężaj jest techniką budowania rozwiązania od góry **top-down**.

Dziel i zwyciężaj

- Idea metody dziel i zwyciężaj: podzielić problem na mniejsze podproblemy i na podstawie ich rozwiązań wyprowadzić rozwiązanie pierwotnego problemu.
- W rozwiązywaniu problemów techniką dziel i zwyciężaj w naturalny sposób wykorzystuje się rekurencję.
- Przykłady problemów rozwiązywanych metodą dziel i zwyciężaj:
 - sortowanie przez scalanie,
 - sortowanie szybkie (przez podział).

Dziel i zwyciężaj

- Schemat użycia strategii dziel i zwyciężaj do rozwiązywania problemów obliczeniowych:
 - dla małych problemów udziel odpowiedzi z definicji (wylicz rozwiązanie ręcznie);
 - większy problem podziel na mniejsze podproblemy;
 - rozwiąż mniejsze podproblemy (rekurencja);
 - połącz rozwiązania podproblemów w rozwiązanie problemu pierwotnego.

Redukcja

- Problem obliczeniowy wykazuje **własność redukcji**, jeśli jego rozwiązanie można obliczyć na podstawie rozwiązania **jednego** podproblemu tego samego typu.
- Aby zastosować **technikę redukcji** problem obliczeniowy musi wykazywać własność redukcji.
- Technika redukcji jest techniką budowania rozwiązania od góry **top-down**.
- Technika redukcji jest szczególnym przypadkiem techniki dziel i zwyciężaj, w której występuje tylko jeden podproblem.

Redukcja

- Idea metody redukcji: wykonaj pewne obliczenia, które pozwolą na zredukowanie pierwotnego problemu do jego podproblemu.
- W rozwiązywaniu problemów techniką redukcji wykorzystuje się w sposób naturalny rekurencję, którą można jednak łatwo zastąpić iteracją.
- Przykłady problemów rozwiązywanych metodą redukcji:
 - rekurencyjne obliczanie silni,
 - szybkie potęgowanie.

Redukcja

- Schemat użycia strategii redukcji do rozwiązywania problemów obliczeniowych:
 - dla małych problemów udziel odpowiedzi z definicji (wylicz rozwiązanie ręcznie);
 - większy problem zredukuj do mniejszego podproblem;
 - wylicz rozwiązanie dla mniejszego problemu;
 - rozwiązanie podproblemu przekształć na rozwiązanie pierwotnego problemu.

Problemy porządkowe

Znajdowanie maksimum i minimum

Znajdowanie wartości maksymalnej (minimalnej) w tablicy

- Dane: zbiór n wartości umieszczonych w tablicy $A[0...n-1]$; dane nie są uporządkowane.
- Zadanie: należy wyznaczyć wartość największą spośród danych.
- Idea rozwiązania: wyznaczamy element maksymalny dla każdego prefiksu (początkowego fragmentu) tablicy i rozszerzamy to rozwiązanie o kolejny element.

Znajdowanie wartości maksymalnej w tablicy – progresja

```
function max-value (out comparable A[n]) => comparable  
{  
    m := A0;  
    for i = 1...n-1 do  
        if Ai > m then m := Ai;  
    return m;  
}
```

- Złożoność czasowa (liczba porównań): n-1 porównań $O(n)$
- Złożoność pamięciowa: $O(1)$

Znajdowanie wartości maksymalnej (minimalnej) w tablicy

- Idea rozwiązania techniką dziel i zwyciężaj:
wyznaczamy element maksymalny dla pierwszej i element maksymalny dla drugiej połowy tablicy i na tej podstawie wyznaczamy element maksymalny dla całej tablicy.

Znajdowanie wartości maksymalnej w tablicy – dziel i zwyciężaj

```
function max-value (out comparable A[n]) => comparable  
{  
  if (n = 1) then return A0;  
  p := ⌊n/2⌋;  
  m1 := max-value(A[0...p-1]);  
  m2 := max-value(A[p...n-1]);  
  return max{m1, m2};  
}
```

- Złożoność czasowa (liczba porównań): n-1 porównań $O(n)$
- Złożoność pamięciowa (głębokość rekurencji): $O(\log n)$

Znajdowanie pozycji wartości maksymalnej (minimalnej) w tablicy

- Dane: zbiór n wartości umieszczonych w tablicy $A[0...n-1]$; dane nie są uporządkowane.
- Zadanie: należy wyznaczyć pozycję w tablicy wartości największej spośród danych.
- Zaleta: na podstawie pozycji można wyznaczyć wartość elementu największego.
- Idea rozwiązania techniką przyrostową: rozszerzanie rozwiązania o kolejne elementy tablicy (pamiętamy rozwiązanie częściowe, czyli pozycję z wartością maksymalną dotychczas znalezioną, dla początkowego fragmentu tablicy).
- Zadanie to można również rozwiązać techniką dziel i zwyciężaj.

Znajdowanie pozycji wartości maksymalnej w tablicy – progresja

```
function max-position (out comparable A[n]) => integer
{
    p := 0;
    for i = 1...n-1 do
        if  $A_i > A_p$  then p := i;
    return p;
}
```

- Złożoność czasowa (liczba porównań): n-1 porównań $O(n)$
- Złożoność pamięciowa: $O(1)$

Problemy porządkowe

Sortowanie

Problem sortowania

- Dane: zbiór n wartości umieszczonych w tablicy $A[0...n-1]$.
- Zadanie: należy tak poukładać dane w tablicy (dokonać permutacji), aby występowały w porządku niemalejącym, czyli $A_0 \leq A_1 \leq \dots \leq A_{n-1}$.
- Ograniczenia: elementy z tablicy możemy tylko porównywać i kopiować.
- Każdy algorytm sortujący wykorzystujący tylko porównania elementów wykona w najgorszym przypadku $\Omega(n \cdot \log n)$ porównań (dowód: drzewa decyzyjne).

Sortowanie bąbelkowe

(ang. bubble sort)

- Idea rozwiązania techniką redukcji: porównujemy po kolei sąsiednie pary elementów porządkując je jednocześnie; po napotkaniu elementu maksymalnego będzie on przepychany w kierunku końca tablicy; po takim cyklu element maksymalny znajdzie się na końcu tablicy, redukując rozmiar problemu o 1; takich cykli wystarczy więc wykonać $n-1$ aby cała tablica została posortowana.
- Zastosowanie: dla małych danych.
- Algorytm jest stabilny i działa w miejscu.

Sortowanie bąbelkowe (ang. bubble sort)

```
procedure bubble-sort (out comparable A[n])  
{  
    for k = n...2 do // k – długość prefiksu  
        for j = 1...k-1 do  
            if  $A_{j-1} > A_j$  then  $A_{j-1} := A_j$ ;  
}
```

- Złożoność czasowa: $n(n-1)/2$ porównań $\Theta(n^2)$
- Złożoność pamięciowa: $O(1)$

Sortowanie bąbelkowe – modyfikacje

- Jeśli w cyklu przesiewania nie wykonano żadnej zamiany, to ciąg jest już posortowany (można przerwać algorytm).
- Jeśli ostatnia zamiana wykonana w cyklu przesiewania wystąpiła na pozycji $j-1$ i j -tej, to końcowe elementy od j włącznie są już na właściwych pozycjach.
- Można przesiewać raz w prawą stronę a potem w lewą stronę – sortowanie koktajlowe.

Sortowanie koktajlowe (ang. cocktail-sort)

```
procedure cocktail-sort (out comparable A[n])  
{  
  a := 0, b := n-1;  
  while a < b do {  
    pos := a;  
    for j = a...b-1 do  
      if Aj > Aj+1 then Aj := Aj+1; pos := j;  
    b := pos;  
    for j = b...a+1 do  
      if Aj-1 > Aj then Aj-1 := Aj; pos := j;  
    a := pos;  
  }  
}
```

- Złożoność czasowa: $\leq n(n-1)/2$ porównań $O(n^2)$
- Złożoność pamięciowa: $O(1)$

Sortowanie przez wstawianie (ang. insertion sort)

- Idea rozwiązania techniką przyrostową: do posortowanego początkowego fragmentu tablicy wstawiam następny element przez porównania i zamiany, aż nie znajdzie się na odpowiedniej pozycji; zaczynamy od fragmentu jednoelementowego (pierwszy element w tablicy); po wykonaniu $n-1$ wstawień kolejnych elementów tablica będzie posortowana.
- Zastosowanie: do danych, które są częściowo uporządkowane albo prawie uporządkowane.
- Algorytm jest stabilny i działa w miejscu.

Sortowanie przez wstawianie (ang. insertion sort)

```
proc insertion-sort (out comparable A[n])  
{  
  for k = 2...n do  
    for j = k-1...1 do  
      if  $A_{j-1} > A_j$  then  $A_{j-1} := A_j$ ;  
      else break;  
}
```

- Złożoność czasowa: $n(n-1)/2$ porównań w najgorszym przypadku (dane są odwrotnie uporządkowane) czyli $O(n^2)$; $n-1$ porównań w najlepszym przypadku (dane są już uporządkowane) czyli $\Omega(n)$.
- Złożoność pamięciowa: $O(1)$

Sortowanie przez wybieranie

(ang. selection sort)

- Idea rozwiązania techniką redukcji: znajdujemy pozycję elementu maksymalnego i przenosimy go na koniec tablicy poprzez zamianę, redukując w ten sposób rozmiar problemu o 1; takich czynności wystarczy więc wykonać $n-1$ aby cała tablica została posortowana.
- Zastosowanie: dla danych, które są duże (ich kopiowanie jest czasochłonne), ponieważ w trakcie działania algorytmu wykonamy co najwyżej $n-1$ zamian elementów.
- Algorytm nie jest stabilny ale działa w miejscu.

Sortowanie przez wybieranie (ang. selection sort)

procedure selection-sort (out comparable A[n])

{

for k = n...2 **do**

 {

 p := max-position(A[0...k-1]);

if p $\neq$ k-1 **then** A_p := A_{k-1};

 }

}

- Złożoność czasowa: $n(n-1)/2$ porównań $\Theta(n^2)$
- Złożoność pamięciowa: $O(1)$

Problemy porządkowe

Wyszukiwanie binarne

Wyszukiwanie binarne

- Dane: zbiór n wartości umieszczonych w tablicy $A[0...n-1]$ w sposób uporządkowany (dane są posortowane, czyli $A_0 \leq A_1 \leq \dots \leq A_{n-1}$) oraz wartość x .
- Zadanie: chcemy wiedzieć czy x występuje w tablicy A .
- Ograniczenia: elementy w tablicy możemy tylko porównywać ze sobą i z wartością x .
- Inne wersje tego zadania:
 - podawanie pozycji znalezionej elementu;
 - podanie ile jest elementów $< x$ lub $\leq x$;
 - jeśli wartości x nie ma w tablicy, to jaka jest najbliższa wartość do x ?

Wyszukiwanie binarne

- Analogia do szukania słowa w słowniku.
- Idea rozwiązania: patrzymy na element środkowy i porównujemy go z wartością x – jeśli $< x$ od tego elementu to dalsze poszukiwania zawężamy do pierwszej części zbioru, jeśli $> x$ od tego elementu to dalsze poszukiwania zawężamy do drugiej części zbioru a w przypadku gdy $x =$ elementowi środkowemu to znaleźliśmy szukaną wartość.

Wyszukiwanie binarne – wersja rekurencyjna

```
function binary-search (out comparable A[n], comparable x)
    => boolean
{
    if n = 0 then return false;
    if n = 1 then return A0 = x;
    m := ⌊n/2⌋;
    if x < Am then return binary-search(A[0...m-1], x);
    if Am < x then return binary-search(A[m+1...n-1], x);
    return true;
}
```

Wyszukiwanie binarne – wersja iteracyjna

```
function binary-search (out comparable A[n], comparable x)
    => boolean
{
    a := 0, b := n-1;
    while a ≤ b do
    {
        m := ⌊(a+b)/2⌋;
        if x < Am then b := m-1;
        else if Am < x then a := m+1;
        else return true;
    }
    return false;
}
```

Wyszukiwanie binarne

- Złożoność algorytmu rekurencyjnego:
 - złożoność pamięciowa – głębokość wywołań rekurencyjnych $O(\log n)$ – rozmiar danych w każdym wywołaniu zmniejsza się dwukrotnie;
 - złożoność czasowa $O(\log n)$ – głębokość rekurencji.
- Złożoność algorytmu iteracyjnego:
 - złożoność pamięciowa $O(1)$ – kilka komórek pamięci;
 - złożoność czasowa $O(\log n)$ – liczba iteracji.
- Technika redukcji.

Problemy porządkowe

Scalanie i sortowanie przez scalanie

Scalanie posortowanych ciągów

- Dane: dwa posortowane ciągi umieszczone w tablicach $A[0...n-1]$ i $B[0...m-1]$ (dane są uporządkowane, czyli $A_0 \leq A_1 \leq \dots \leq A_{n-1}$ oraz $B_0 \leq B_1 \leq \dots \leq B_{m-1}$).
- Zadanie: Mamy połączyć dane z obu ciągów w taki sposób, aby dane te były ostatecznie uporządkowane.
- Ograniczenia: elementy z tablicy możemy tylko porównywać i kopiować.
- Inne warianty tego problemu:
 - liczba ciągów do scalenia może być większa niż 2,
 - dwa ciągi do scalenia są umieszczone w jednej tablicy jeden za drugim.

Scalanie posortowanych ciągów

- Spostrzeżenie: minimum spośród pierwszych elementów obu ciągów $\min(A_0, B_0)$ jest elementem minimalnym całego zbioru danych.
- Idea algorytmu: wyznaczamy minimum z obu ciągów i przenosimy go do tablicy wynikowej; proces ten powtarzamy dopóki nie wyczerpią się dane.

Scalanie posortowanych ciągów

```
function merge (comparable A[n], comparable B[m]) =>
    comparable[n+m]
{
    new comparable C[n+m];
    i := 0, j := 0;
    while i < n and j < m do
        if  $A_i \leq B_j$  then {  $C_{i+j} := A_i$ ; i++; }
        else {  $C_{i+j} := B_j$ ; j++; }
    while i < n do {  $C_{i+j} := A_i$ ; i++; }
    while j < m do {  $C_{i+j} := B_j$ ; j++; }
    return C;
}
```


Scalanie posortowanych ciągów

- Algorytm scalania jest iteracyjny.
- Złożoność algorytmu scalania:
 - złożoność pamięciowa $O(n+m)$ – 2 komórki pamięci + tablica na wynik.
 - złożoność czasowa $O(n+m)$ – liczba przepisania elementów do tablicy wynikowej.
- Algorytm scalania jest stabilny.
- Inne wersje danych do tego algorytmu:
 - dane znajdują się w jednej tablicy i wynik też ma się tam znaleźć;
 - procedura scalająca ma podany przez parametr bufor do scalania wyników.

Sortowanie przez scalanie

- Idea: dzieli dane wejściowe na dwa rozłączne równoliczne podzbiory (z dokładnością do 1), następnie sortuje każdy z tych podzbiorów i za pomocą scalania łączy wyniki w jeden posortowany ciąg.

Sortowanie przez scalanie (ang. merge sort)

```
procedure merge-sort (out comparable A[n])  
{  
    // if n jest małe np. <5 then { insertion-sort(A); return; }  
    if n = 1 then return;  
    m :=  $\lfloor n/2 \rfloor$ ;  
    merge-sort(A[0...m-1]);  
    merge-sort(A[m...n-1]);  
    C := merge(A[0...m-1], A[m...n-1]);  
    A := C;  
}
```

Sortowanie przez scalanie

- Sortowanie przez scalanie korzysta z techniki dziel i zwyciężaj.
- Algorytm ten jest stabilny, ponieważ korzysta ze stabilnego scalania.
- Algorytm ten nie działa w miejscu, gdyż korzysta z rekurencji i używa tablic pomocniczych do scalania.
- Złożoność algorytmu sortowania przez scalanie:
 - złożoność pamięciowa $O(n)$ – na każdym poziomie rekurencji używamy tablic pomocniczych do umieszczenia wyniku scalania;
 - złożoność czasowa $O(n \cdot \log n)$ – wynika z zależności rekurencyjnej:
$$T(n) = 2 T(n/2) + O(n)$$
- Algorytm jest optymalny czasowo!

Problemy porządkowe

Podział i sortowanie szybkie (przez podział)

Podział danych w ciągu

- Dane: nieuporządkowany ciąg umieszczony w tablicy $A[0...n-1]$ oraz wartość p zwana piwotem (ang. pivot – oś) albo elementem dzielącym.
- Zadanie: Mamy podzielić dane w ciągu na elementy $\leq p$ i $\geq p$ w taki sposób, aby elementy mniejsze od piwota znalazły się na początku tablicy a elementy większe na końcu.
- Ograniczenia: elementy z tablicy możemy tylko porównywać i kopiować.
- Szczególna wersja tego problemu:
 - element dzielący jest jednym z elementów tablicy (wiemy którym) i po podziale piwot jest umieszczany na granicy podziału;
 - wartości elementów w tablicy często się powtarzają i należy dokonać trójpodziału czyli podzielić elementy na wartości $< p$, $= p$ i $> p$, przy czym wartości równe piwotowi umieszczamy pośrodku.

Podział danych w ciągu

- Idea algorytmu Lomuta:
jeśli dokonaliśmy podziału $n-1$ elementów w tablicy i elementy $>p$ zaczynają się od pozycji k , n -ty element możemy prosto dołączyć do rozwiązania pozostawiając go na tej samej pozycji gdy jest $>p$ albo zamieniając z k -tym elementem gdy jest $\leq p$.

Podział danych w ciągu

```
function Lomuto-partition (out comparable A[n], comparable p)
    => integer
{
    g := 0;
    for i = 0...n-1 do
        if  $A_i < p$  then {  $A_g := A_i$ ; g++; }
    return g;
}
```

- W algorytmie Lomuta elementem dzielącym może być dowolna wartość.

Podział danych w ciągu

- Algorytm podziału Lomuta jest iteracyjny.
- Złożoność algorytmu Lomuta :
 - złożoność pamięciowa: $O(1)$
 - złożoność czasowa (liczba porównań): $O(n)$
- Algorytm jest stabilny.

Sortowanie szybkie (przez podział)

- Idea: dokonuję się podziału tablicy względem losowo wybranego elementu (podział musi być nietrywialny), następnie sortuje się rekurencyjnie część z elementami mniejszymi od piwota i potem z elementami większymi od piwota.
- Wskazówka: procedurę podziału można tak zmodyfikować, aby w punkcie podziału znalazł się piwot, który jest jednym z elementów zbioru.
- Uwaga: losowy wybór piwota spośród danych w tablicy gwarantuje nam uniknięcie złych danych.

Sortowanie szybkie (przez podział)

```
function quick-sort (out comparable A[n])  
{  
    if n jest małe then insertion-sort(A);  
    i := losowa wartość ze zbioru {0,...,n-1};  
    p := Ai;  
    m := partition(A, p); // na pozycji m-tej jest piwot  
    quick-sort(A[0...m-1]);  
    quick-sort(A[m+1...n-1]);  
}
```

Sortowanie szybkie (przez podział)

- Sortowanie przez podział korzysta z techniki dziel i zwyciężaj.
- Algorytm ten nie jest stabilny, gdyż korzysta z niestabilnego podziału danych.
- Algorytm ten nie działa w miejscu, gdyż korzysta z rekurencji.
- Złożoność algorytmu sortowania przez podział:
 - złożoność pamięciowa $O(n)$ w najgorszym przypadku i $O(\log n)$ w oczekiwanym przypadku – głębokość rekurencji;
 - złożoność czasowa $O(n^2)$ w najgorszym przypadku i $O(n \cdot \log n)$ w oczekiwanym przypadku – wynika z zależności rekurencyjnej:
$$T(n) = T(k) + T(n-k) + O(n)$$

Problemy porządkowe

Wyszukiwanie k-tego co do wielkości
elementu

Wyszukiwanie k-tego elementu

- Dane: nieuporządkowany ciąg umieszczony w tablicy $A[0...n-1]$ oraz wartość k z zakresu od 0 do $n-1$.
- Zadanie: Mamy wyznaczyć k -tą co do wielkości wartość w tej tablicy, czyli taką wartość x z tablicy A , że po posortowaniu elementów tablicy na pozycji k -tej znajdzie się element o wartości x (w tablicy tej istnieje k elementów $< x$ gdy elementy te są parami różne).
- Ograniczenia: elementy z tablicy możemy tylko porównywać i kopiować.
- Szczególne przypadki:
 - minimum/maksimum,
 - mediana.

Wyszukiwanie k-tego elementu

– algorytm Hoare'a

- Idea algorytmu Hoare'a (1961): po dokonaniu podziału danych względem losowo wybranej wartości z tablicy, k-ty co do wielkości element będzie się znajdował tylko w jednej z części po podziale.

Wyszukiwanie k-tego elementu – algorytm Hoare'a

```
function kth-element (out comparable A[n], integer k) =>
    comparable
{
    if k=0 then { m := find-min-pos(A); A0 ::= Am; return A0; }
    if k=n-1 then { m := find-max-pos(A); Am ::= An-1; return An-1; }
    i := losowa wartość ze zbioru {0,...,n-1};
    p := Ai;
    m := partition(A, p); // na m-tej pozycji znajduje się p
    if k < m then return kth-element(A[0...m-1], k);
    else if m < k then return kth-element(A[m+1...n-1], k-m-1);
    else return p;
}
```


Wyszukiwanie k-tego elementu

– algorytm Hoare'a

- Wyszukiwanie k-tego co do wielkości elementu korzysta z techniki redukcji.
- Algorytm ten nie jest stabilny, gdyż korzysta z niestabilnego podziału danych.
- Algorytm ten nie działa w miejscu, gdyż jest rekurencyjny.
- Efektem ubocznym algorytmu Hoare'a jest podział danych w tablicy na elementy mniejsze od k-tej wartości (po lewej stronie) i elementy większe (po prawej stronie).
- Złożoność algorytmu wyszukiwania k-tego co do wielkości elementu z wykorzystaniem podziału:
 - złożoność pamięciowa $O(n)$ w najgorszym przypadku i $O(\log n)$ w oczekiwanym przypadku – głębokość rekurencji;
 - złożoność czasowa $O(n^2)$ w najgorszym przypadku i $O(n)$ w przypadku oczekiwanym.