Problemy porządkowe

r.

Problemy porządkowe

- 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 $\leq_{lex}$

Problemy porządkowe – tablica

- Dane są przechowywane w tablicy.
- Tablicę n-elementową oznaczamy przez A[n] albo precyzyjniej przez A[0…n-1] (elementy w tablicy są indeksowane od 0).
- 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-situ, in-place), jeśli korzysta tylko z pewnej stałej liczby O(1) dodatkowych komórek pamięci.

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: rozszerzanie rozwiązania o kolejne elementy w tablicy (pamiętamy rozwiązanie częściowe dla początkowego fragmentu tablicy)
- Złożoność czasowa: n-1 porównań O(n)
- Złożoność pamięciowa: 2 komórki O(1)

Znajdowanie wartości maksymalnej w tablicy – akceleracja

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

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.
- Idea rozwiązania: rozszerzanie rozwiązania o kolejne elementy
- Złożoność czasowa: n-1 porównań O(n)
- Złożoność pamięciowa: 2 komórki O(1)

Znajdowanie pozycji wartości maksymalnej w tablicy

```
func max-position (out comparable A[n]) => integer
{
   m := 0;
   for i = 1…n-1 do
         if A_i > A_m then m := i;
   return m;
}
```

Znajdowanie wartości maksymalnej w tablicy – dziel i zwyciężaj

```
func 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};
}
```

Czas: liczba porównań O(n)

Pamięć: głębokość wywołań rekurencyjnych O(log n)

Jednoczesne znajdowanie elementu minimalnego i maksymalnego

- Dane: nieuporządkowany zbiór n wartości umieszczonych w tablicy A[0…n-1].
- Zadanie: należy wyznaczyć w tablicy wartość największą i najmniejszą spośród danych.
- Idea rozwiązania: najpierw porównujemy parami wszystkie dane w tablicy – elementy mniejsze z każdej pary będą kandydatami na minimum a elementy większe będą kandydatami na maksimum; następnie wyznaczamy element minimalny spośród kandydatów na minimum i element maksymalny spośród kandydatów na maksimum.
- Złożoność czasowa: $\approx 3/2n$ porównań O(n)
- Złożoność pamięciowa: 5 komórek O(1)

Jednoczesne znajdowanie elementu minimalnego i maksymalnego

```
func min-max-value (out comparable A[n])
=> (comparable , comparable)
{
   p := 0, q := n-1;
   while p < q do
   {
         if A_p > A_q then A_p :=: A_q;
         p++, q—;
   }
   x := min-value(A[0…q]);
   y := max-value(A[p…n-1]);
   return (x, y);
}
```

Jednoczesne znajdowanie elementu minimalnego i maksymalnego – dziel i zwyciężaj

- Dane: nieuporządkowany zbiór n wartości umieszczonych w tablicy A[0…n-1].
- Zadanie: należy wyznaczyć w tablicy wartość największą i najmniejszą spośród danych.
- Idea rozwiązania: dla małych danych (1, 2 elementy) zadanie rozwiązujemy ad hoc; dla większych danych dzielimy je na dwie równoliczne częci i rozwiązujemy problemy rekurencyjnie, potem z obu wyników konstruujemy rozwiązanie.
- Złożoność czasowa: O(n)
- Złożoność pamięciowa: O(log n)

Jednoczesne znajdowanie elementu minimalnego i maksymalnego – dziel i zwyciężaj

```
func min-max-value (out comparable A[n])
=> (comparable , comparable)
{
   if (n = 1) return (A0, A0);
   if (n = 2)
         if (A0 < A1) return (A0, A1);
         else return (A1, A0);
   p := ⌊n/2⌋;
   (a, b) := min-max-value(A[0…p-1]);
   (c, d) := min-max-value(A[p…n-1]);
   return (min{a, c}, max{b, d});
}
```

Jednoczesne znajdowanie elementu minimalnego i maksymalnego – dziel i zwyciężaj

func min-max-value (out comparable A[n])
=> (comparable , comparable)
{
  if n = 1 **then return** (A_0, A_0);
  if n = 2 **then**
    if $A_0 < A_1$ **then return** (A_0, A_1);
    else return (A_1, A_0);
  p := $\lfloor n/2 \rfloor$;
  (m_1 , M_1) := min-max-value(A[0…p-1]);
  (m_2, M_2):= min-max-value(A[p…n-1]);
  return (min{m_1, m_2}, max{M_1, M_2});
}

Pamięć: głębokość rekurencji O(log n)

Czas: liczba porównań O(n)

Problem sortowania

- Dane: zbiór n wartości umieszczonych w tablicy A[0…n-1].
- Zadanie: należy tak poukładać dane w tablicy, aby występowały w porządku niemalejącym, czyli $A_0 \leq A_1 \leq \ldots \leq A_{n-1}$.
- 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.

Sortowanie bąbelkowe (ang. bubble sort)

- Idea rozwiązania: porównuję 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.
- Złożoność czasowa: $n(n-1)/2$ porównań $\Theta(n^2)$
- Złożoność pamięciowa: 3 komórki $O(1)$
- Zastosowanie: dla małych danych.
- Technika redukcji.
- Algorytm jest stabilny i działa w miejscu.

Sortowanie bąbelkowe (ang. bubble sort)

```
proc bubble-sort (out comparable A[n])
{
   for k = n-1…1 do
         for j = 1…k do
               if A_{j-1} > A_j then A_{j-1} :=: A_j;
}
```

Sortowanie przez wybieranie (ang. selection sort)

- Idea rozwiązania: 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.
- Złożoność czasowa: n(n-1)/2 porównań $\Theta(n^2)$
- Złożoność pamięciowa: 3 komórki O(1)
- Zastosowanie: dla danych, które są duże (ich kopiowanie jest czasochłonne), ponieważ w trakcie działania algorytmu wykonamy co najwyżej n-1 zamian elementów.
- Technika redukcji.
- Algorytm nie jest stabilny ale działa w miejscu.

Sortowanie przez wybieranie (ang. selection sort)

```
proc selection-sort (out comparable A[n])
{
   for k = n-1…1 do
   {
         m := max-position(A[0…k]);
         if m ≠ k then A_m :=: A_k;
   }
}
```

Sortowanie przez wstawianie (ang. insertion sort)

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

Sortowanie przez wstawianie (ang. insertion sort)

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

Sortowanie przez zliczanie (ang. counting sort)

- Dane w tym algorytmie to liczby całkowite z małego k-elementowego przedziału (bez straty ogólności można założyć, że liczby te pochodzą ze zbioru {0, 1, …, k-1}).
- W algorytmie wykorzystujemy informacje o wartościach elementów (a nie tylko porównujemy je między sobą).

Sortowanie przez zliczanie (ang. counting sort)

- Idea rozwiązania prostego: tworzymy tablicę liczników wystąpień poszczególnych elementów; zerujemy te liczniki; następnie przeglądamy zawartość tablicy z danymi i dla każdego elementu zwiększamy licznik związany z jego wartością; na koniec wypisujemy po kolei taką liczbę wartości jaka jest zarejestrowana w licznikach.
- Złożoność czasowa: O(n + k) – każdy z n elementów raz odczytujemy, n razy zwiększmy licznik, n razy wypisujemy każdy element oraz każdy z k liczników należy na początku wyzerować a potem przeglądnąć.
- Złożoność pamięciowa: O(k) – tyle mamy liczników.
- Zastosowanie: dla liczb całkowitych z wąskiego przedziału.

Sortowanie przez zliczanie (ang. counting sort)

```
proc simple-counting-sort (out integer A[n])
{
   integer C[k];
   for j = 0…k-1 do C_j := 0;
   for i = 0…n-1 do C[A_i]++;
   i := 0;
   for j = 0…k-1 do
         while C_j > 0 do
         {
                  A_i := j;
                  i++;
                   C_j--;
         }
}
```

Sortowanie przez zliczanie (ang. counting sort)

- Idea rozwiązania: tworzymy tablicę kolejek wystąpień poszczególnych elementów i inicjalizujemy te kolejki (na początku każda kolejka jest pusta); następnie przeglądamy zawartość tablicy z danymi i każdy element umieszczamy w kolejce indeksowanej polem kluczowym tego elementu; na koniec przenosimy my po kolei wszystkie elementy z kolejek do tablicy z danymi.
- Złożoność czasowa: O(n + k) – każdy z n elementów raz odczytujemy, n razy zwiększmy licznik, n razy wypisujemy każdy element oraz każdy z k liczników należy na początku wyzerować a potem przeglądnąć.
- Złożoność pamięciowa: O(n + k) – mamy k kolejek i w sumie n elementów w tych kolejkach.
- Zastosowanie: dla danych z polami kluczowymi będącymi liczbami całkowitymi z wąskiego przedziału.
- Algorytm jest stabilny ale nie działa w miejscu.

Sortowanie przez zliczanie (ang. counting sort)

```
proc counting-sort (out object A[n])
{
    queue C[k];
    for j = 0…k-1 do C_j := ∅;
    for i = 1…n-1 do C[A_i.key].enqueue(A_i);
    i := 0;
    for j = 0…k-1 do
            while not C_j.empty() do
            {
                    A_i := C_j.dequeue();
                    i++;
            }
}
```

Wyszukiwanie binarne

- Dane: zbiór n wartości umieszczonych w tablicy A[0…n-1] w sposób uporządkowany (dane są posortowane, czyli $A_0 \leq A_1 \leq \ldots \leq A_{n-1}$) oraz wartość x.
- Zadanie: Chcemy wiedzieć czy x występuje w tablicy A.
- Zadanie równoważne: Chcemy wiedzieć ile występuje w tablicy A elementów <x.
- Ograniczenia: elementy z tablicy możemy tylko porównywać.

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

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

Wyszukiwanie binarne – wersja iteracyjna

```
func binary-search (out comparable A[n], comparable x) =>
   boolean
{
   a := 0, b := n-1;
   while a ≤ b do
   {
         m := ⌊(a+b)/2⌋;
         if x < A_m then b := m-1;
         else if A_m < 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) – ilość wywołań rekurencyjnych.
- Złożoność algorytmu iteracyjnego:
  - złożoność pamięciowa O(1) – 3 komórki pamięci.
  - złożoność czasowa O(log n) – liczba iteracji.
- Technika redukcji.

Scalanie posortowanych ciągów

- Dane: dwa posortowane ciągi umieszczone w tablicach A[0…n-1] i B[0…m-1] (dane są uporządkowane, czyli $A_0 \leq A_1 \leq \ldots \leq A_{n-1}$ oraz $B_0 \leq B_1 \leq \ldots \leq B_{m-1}$).
- 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.

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

```
func merge (comparable A[n], comparable B[m]) =>
    comparable[n+m]
{
    comparable C[n+m];
    i := 0, j:= 0;
    while i < n and j < m do
            if A_i ≤ 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 przepisań 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: dziele dane wejściowe na dwa rozłączne równoliczne podzbiory (z dokładnością do 1), następnie sortuję każdy z tych podzbiorów i za pomocą scalania łączę wyniki w jeden posortowany ciąg.

Sortowanie przez scalanie (ang. merge sort)

```
proc merge-sort (out comparable A[n])
{
   if n jest małe np. <5 then { insertion-sort(A); return; }
   m := ⌊n/2⌋;
   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, gdy 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!

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.
- Szczególne 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

Algorytm Lomuta (elementem dzielącym jest dowolna wartość):

```
func partition-Lomuto
(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;
}
```

Podział danych w ciągu

- Idea algorytmu Sedgewicka:
  szukamy od początku tablicy elementu >p i od końca tablicy elementu <p, następnie zamieniamy te elementy miejscami; po tej operacji problem redukuje się do elementów umieszczonych pomiędzy tymi zamienionymi.

Podział danych w ciągu

Algorytm Sedgewicka (elementem dzielącym jest dowolna wartość) – wersja rekurencyjna:

```
func partition-Sedgewick (out comparable A[n], comparable p) => integer
{
    if n≤0 then return 0;
    b := 0;
    while b<n and A_b<p do b++;
    if b=n then return n;
    e := n-1;
    while e≥0 and A_e≥p do e--;
    if e<b then return b;
    A_b :=: A_e;
    k := partition-Sedgewick(A[b+1…e-1], p);
    return b+1+k;
}
```

Podział danych w ciągu

Algorytm Sedgewicka (elementem dzielącym jest dowolna wartość) – wersja iteracyjna:

```
func partition-Sedgewick (out comparable A[n], comparable p) => integer
{
    b := 0;
    while b<n and A_b<p do b++;
    if b=n then return n;
    e := n-1;
    while e≥0 and A_e≥p do e--;
    if e<b then return b;
    do {
            A_b :=: A_e;
            do b++; while A_b<p;
            do e--; while A_e≥p;
    } while b<e;
    return b;
}
```

Podział danych w ciągu

- Algorytm podziału Lomuta jest iteracyjny.
- Algorytm podziału Sedgewicka jest iteracyjny (można go też zaprogramować rekurencyjnie).
- Złożoność algorytmu Lomuta i Sedgewicka w wersji iteracyjnej:
  - złożoność pamięciowa O(1);
  - złożoność czasowa O(n) – liczba porównań elementów.
- Algorytm Sedgewicka wykonuje minimalną liczbę zamian elementów.
- Algorytmy te nie są stabilne.

Sortowanie przez podział (szybkie)

- 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 przez podział (szybkie)

```
func qiuck-sort (out comparable A[n])
{
   if n jest małe then insertion-sort(A);
   i := losowa wartość ze zbioru {0,…,n-1};
   p := A_i;
   m := partition(A, p);
   qiuck-sort(A[0…m-1]);
   qiuck-sort(A[m+1…n-1]);
}
```

Sortowanie przez podział (szybkie)

- 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:
    $T(n) = T(k) + T(n-k) + O(n)$

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 w tablicy tej istnieje k elementów <x.
- Ograniczenia: elementy z tablicy możemy tylko porównywać i kopiować.

Wyszukiwanie k-tego elementu

- Idea algorytmu Hoare'a: 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 wyznaczania k-tego co do wielkości elementu w tablicy:

func kth-element (out comparable A[n], integer k) => comparable

{

if k=0 **then** { m := find-min-pos(A); A_0 :=: A_m; **return** A_0; }

if k=n-1 **then** { m := find-max-pos(A); A_m :=: A_{n-1}; **return** A_{n-1}; }

i := losowa wartość ze zbioru {0,…,n-1};

p := A_i;

m := partition(A, p);

if k<m **then return** kth-element(A[0…m-1], k);

else if k>m **then return** kth-element(A[m+1…n-1], k-m-1);
else return A_m;

}

Wyszukiwanie k-tego elementu

- 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.