notatki do zajęć (2) *9 października 2006 r.*

ALGORYTMY I STRUKTURY DANYCH

SORTOWANIE

Instytut Informatyki Uniwersytetu Wrocawskiego *Paweł Rzechonek*

Zadania porządkowe (ang. *ordering problems*) to klasa problemów, w których zbiór danych należy w pewien sposób uporządkować lub coś w tym zbiorze znaleźć. Do problemów porządkowych zalicza się przede wszystkim sortowanie i wybór mediany.

Dane są przeważnie pamiętane w tablicach lub w plikach, tak że z każdym elementem jest związana ***pozycja*** (ang. *position*) w zbiorze, na której on występuje.

Elementy zbioru danych pochodzą z określonego uniwersum z porządkiem liniowym, aby każde dwa można było porównać za pomocą relacji $\leq$. Czasami jeden element może być dużym agregatem różnych informacji. Wtedy wyróżnia się jedno pole zwane ***kluczem*** (ang. *key*), na którym jest określona relacja $\leq$ i które jest reprezentantem agregatu w operacji porównywania.

W problemach porządkowych nie zakłada się uporządkowania zbioru danych — przeważnie są to zbiory nieuporządkowane, albo częściowo uporządkowane. Spójrzmy na n-elementowy zbiór danych jako na ciąg elementów $A = (a_0, \ldots, a_{n-1})$. Miarą nieuporządkowania w takim ciągu jest liczba występujących w nim ***inwersji*** (ang. *inversion*), czyli takich par (i, j), że $0 \leq i < j < n$ oraz $a_i > a_j$:

$$|\{(i,j) \mid 0 \leq i < j < n\,,\, a_i > a_j\}|$$

O ciągu powiemy, że jest ***uporządkowany*** (ang. *ordered*), gdy jest niemalejący:

$$\forall\, 0 < i < n \,:\, a_{i-1} \leq a_i$$

1 Podstawowe pojęcia

Mówimy, że algorytm porządkujący działa ***w miejscu*** (ang. *in–place*), jeśli podczas działania algorytm zużywa tylko stałą liczbę dodatkowych komórek pamięci. Poza zbiór z danymi można więc wyprowadzić tylko stałą liczbę elementów w trakcie działania algorytmu.

Mówimy, że algorytm porządkujący jest ***stabilny*** (ang. *stable*), jeśli po zakończeniu działania algorytm zachowuje względne ustawienia elementów równych Algorytm taki może zmieniać pozycje elementów w zbiorze z danymi, ale elementy o równych kluczach zachowują początkowe ustawienia względem siebie.

O algorytmach porządkujących będziemy zakładali, że możemy mieć ***bezpośredni dostęp*** (ang. *random access*) do każdego elementu, że elementy mogą być jedynie ***porównywane*** (ang. *compare*) i ***przepisywane*** (ang. *move*) bądź ***zamieniane*** (ang. *exchange*) miejscami. Zauważmy, że jedna zamiana miejscami dwóch elementów wymaga trzech przepisań.

2 Sortowanie

Sortowanie (ang. *sorting*) to problem bardzo często rozwiązywany na komputerze. Wiąże się to z faktem, że dużo szybciej można znajdować informacje w zbiorach uporządkowanych niż w nieuporządkowanych.

Problem sortowania można zdefiniować następująco.

Dane: W n-elementowej tablicy $T[0 \ldots n-1]$ zapisany jest nieuporządkowany ciąg wartości $a_0, \ldots, a_{n-1}$.

Zadanie: Dane w tej tablicy należy tak poprzestawiać, aby $a_{\sigma_0} \leq \ldots \leq a_{\sigma_{n-1}}$, gdzie σ jest pewną permutacją zbioru n-elementowego.

Ograniczenia: Porządkując dane możemy jedynie porównywać elementy i przepisywać je bądź zamieniać miejscami.

2.1 Wyszukiwanie

Najczęściej wykonywaną operacją na zbiorach z danymi jest ***wyszukiwanie*** (ang. *searching*) informacji względem zadanego klucza. Problem ten można zdefiniować następująco.

Dane: Zadany jest klucz x oraz n-elementowa tablica $T[0 \ldots n-1]$ z elementami $a_0, \ldots, a_{n-1}$.

Zadanie: Należy znaleźć pozycję w tablicy, na której znajduje się element o wartości x.

Ograniczenia: Można jedynie porównywać klucz z elementami w tablicy.

Najprostszym rozwiązanie jest przeglądanie tablicy od początku do końca i porównywanie jej elementów z kluczem, czyli ***wyszukiwanie sekwencyjne*** (ang. *sequence searching*).

```
function SEQ–SEARCH (keytype T[0 ... n − 1], key x) ↦ int
{
    for i = 0 ... n − 1 do
        if T[i].key = x then return i;
    return null;
}
```

Działanie procedury SEQ–SEARCH wymaga wykonania 1 porównania w najlepszym przypadku, n porównań w najgorszym przypadku, oraz $\sim \frac{n}{2}$ porównań w przypadku średnim gdy szukany element znajduje się w tablicy.

Dużo szybszy algorytm można skonstruować, gdy dane w tablicy są uporządkowane. Oto zmienione założenie dotyczące postaci danych.

Dane: Zadany jest klucz x oraz n-elementowa tablica $T[0 \ldots n-1]$ z elementami $a_0, \ldots, a_{n-1}$ uporządkowanymi w kolejności niemalejącej $a_0 \leq \ldots \leq a_{n-1}$.

Korzystając z informacji o uporządkowaniu danych można zastosować algorytm ***wyszukiwania binarnego*** (ang. *binary searching*).

```
function BIN–SEARCH (keytype T[0 ... n − 1], key x) ↦ int
{
    int a ← 0 ; // początek obszaru poszukiwań
    int b ← n ; // koniec obszaru poszukiwań
    int m ← ⌊(a+b)/2⌋ ; // środek obszaru poszukiwań
    while a < b do
    {
        compare T[m].key ? x
            case < then a ← m + 1 ;
            case > then b ← m ;
            case = then return m ;
        m ← ⌊(a+b)/2⌋ ;
    }
    return null;
}
```

Procedura BIN–SEARCH wykona nie więcej niż $\lfloor \log n + 1 \rfloor$ porównań w najgorszym przypadku.

2.2 Elementarne metody sortowania

2.2.1 Sortowanie bąbelkowe

Algorytm ***sortowania bąbelkowego*** (ang. *bubble sort*) jest najprostszym algorytmem sortowania. Idea sortowania bąbelkowego opiera się na redukcji rozmiaru problemu o 1 poprzez przesunięcie na koniec elementu maksymalnego. Przeglądając tablicę od początku do końca dbamy o to, aby element lokalnie największy zawsze znajdował się na lokalnym końcu. Operację przesuwania powtarzamy $n - 1$ razy, aż zadanie zredukuje się do rozmiaru 1.

```
procedure BUBBLE–SORT (key T[0 ... n − 1])
{
    for k = n − 1 ... 1 do
        for i = 1 ... k do
            if T[i − 1] > T[i] then EXCHANGE (T, i − 1, i) ;
}
```

W algorytmie sortowania bąbelkowego wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku i co najwyżej $\frac{n(n-1)}{2}$ zamian elementów. Czas działania algorytmu jest więc rzędu $\Theta(n^2)$.

2.2.2 Sortowanie przez zamianę

Algorytm ***sortowania przez zamianę*** (ang. *exchange sort*) jest naturalną modyfikacją sortowania bąbelkowego. Idea sortowania przez zamianę także polega na redukcji rozmiaru problemu o 1 poprzez ustawienie na końcu elementu maksymalnego. Teraz jednak zamiast przesuwać element lokalnie największy coraz dalej, to przeglądamy tablicę i po napotkaniu elementu większego od stojącego na końcu dokonujemy zamiany. Operację przenoszenia na koniec elementu maksymalnego powtarzamy $n - 1$ razy, aż zadanie zredukuje się do rozmiaru 1.

```
procedure EXCHANGE–SORT (key T[0 ... n − 1])
{
    for k = n − 1 ... 1 do
        for i = 0 ... k − 1 do
            if T[i] > T[k] then EXCHANGE (T, i, k);
}
```

W algorytmie sortowania przez zamianę wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku. Czas działania algorytmu jest więc rzędu $\Theta(n^2)$.

2.2.3 Sortowanie przez wybieranie

Algorytm ***sortowania przez wybieranie*** (ang. *selection sort*) jest z kolei rozwinięciem sortowania przez zamianę. Idea sortowania przez wybieranie polega na redukcji rozmiaru problemu o 1 poprzez wstawienie na koniec elementu maksymalnego. Najpierw przeglądamy całą tablicę aby wyznaczyć pozycję elementu największego a następnie zamieniamy go z ostatnim. Operację wstawiania na koniec elementu maksymalnego powtarzamy $n - 1$ razy, aż zadanie zredukuje się do rozmiaru 1.

```
procedure SELECTION–SORT (key T[0 ... n − 1])
{
    for k = n − 1 ... 1 do
    {
        int m ← 0; // pozycja elementu maksymalnego
        for i = 1 ... k do
            if T[i] > T[m] then m ← i;
        if m < k then EXCHANGE (T, m, k);
    }
}
```

W algorytmie sortowania przez wybieranie wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku, ale tylko co najwyżej $n - 1$ zamian elementów. Czas działania algorytmu jest więc rzędu $\Theta(n^2)$.

2.2.4 Sortowanie przez wstawianie

Algorytm ***sortowania przez wstawianie*** (ang. *insertion sort*) polega na wstawianiu kolejnych elementów do uporządkowanego fragmentu. Ideą sortowania przez wstawianie jest wepchnięcie na właściwą pozycję ostatniego elementu do posortowanej początkowej części. W ten sposób zwiększamy za każdym razem długość posortowanej części tablicy o 1. Operację wstawiania elementu do posortowanego początkowego fragmentu tablicy powtarzamy $n - 1$ razy, aż posortujemy wszystkie elementy.

```
procedure INSERTION–SORT (key T[0 ... n − 1])
{
    for k = 1 ... n − 1 do
    {
        int p ← k ;
        while p > 0 ∧ T[p − 1] > T[p] do EXCHANGE (T, p − 1, p) ;
    }
}
```

W algorytmie sortowania przez wstawianie wykonujemy $\frac{n(n-1)}{2}$ porównań w najgorszym przypadku i około $\frac{n^2}{4}$ w przypadku średnim. Czas działania algorytmu jest więc rzędu $\Theta(n^2)$.

2.3 Sortowanie Shella

Sortowanie Shella (D. L. Shell, 1959) jest prostym rozszerzeniem sortowania przez wstawianie, które przyspiesza proces sortowania dzięki przestawianiu odległych a nie tylko sąsiadujących elementów.

Definicja 1 *h-pociąg ciągu $A = (a_0, a_1, \ldots)$ to podciąg złożony z elementów oddalonych od siebie o wielokrotność liczby h.*

Definicja 2 *Ciąg $A = (a_0, a_1, \ldots)$ jest h-posortowany, gdy wszystkie jego h-podciągi są posortowane.*

Ciąg n-elementowy zawiera co najwyżej h różnych h-podciągów: gdy $n < h$ to ciąg zawiera tylko n podciągów jednoelementowych, w przeciwnym przypadku jest ich dokładnie h.

Twierdzenie 3 *Przeprowadzenie g-sortowania w ciągu h-posortowanym nie zniszczy h-posortowania.*

Twierdzenie to jest podstawą sortowania Shella. Otwartym problemem pozostaje odpowiedni dobór przyrostów, czyli ciągu wartości $H = (h_0, h_1, \ldots)$, według którego należy przeprowadzać kolejne sortowania, przy czym $h_0 = 1$ jest ostatnim przyrostem w sortowaniu.

```
procedure SHELL–SORT (key T[0 ... n − 1])
{
    H ← (h_0, h_1, ...) ; // ciąg przyrostów
    k ← max{i | h_i < n} ;
    while k ≥ 0 do
    {
        h ← h_k ;
        for g = 0 ... h − 1 do
            ⟨posortuj metodą INSERTION–SORT h-podciąg rozpoczynający się od T[g]⟩ ;
        k ← k − 1 ;
    }
}
```

Twierdzenie 4 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się dwuelementowy ciąg przyrostów* $(1, \lfloor \sqrt[3]{n} \rfloor)$ *będzie działało w czasie* $O(n^{5/3})$.

Stosując sortowanie Shella z tylko dwoma skokami można istotnie przyspieszyć proces sortowania w stosunku do sortowania przez proste wstawianie.

Twierdzenie 5 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się dwuelementowy ciąg przyrostów* $(1, \lfloor \sqrt[3]{n} \rfloor)$ *będzie działało w czasie* $O(n^{5/3})$.

Jeszcze lepsze wyniki można uzyskać wprowadzając większą ilość skoków. Bardzo dobrze zachowuje się sekwencja skoków 1, 3, 7, 15, 31... (T. N. Hibbard, 1963), którą można zapisać następującym wzorem:

$$h_i \quad = \quad 2_{i+1} - 1 \quad \text{dla} \quad i \geq 0$$

Twierdzenie 6 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Hibbarda (1, 3, 7, 15, 31...) będzie działało w czasie* $O(n^{3/2})$.

Inny dobry ciąg to sekwencja skoków 1, 4, 13, 40, 121... (D. E. Knuth, 1969), którą można zdefiniować rekurencyjnie:

$$\begin{aligned} h_0 &= 1 \\ h_i &= 3\,h_{i-1} + 1 \quad \text{dla} \quad i \geq 1 \end{aligned}$$

Twierdzenie 7 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Knutha (1, 4, 13, 40, 121...) będzie działało w czasie* $O(n^{3/2})$.

Działanie algorytmu Shella można jeszcze poprawić stosując sekwencję skoków 1, 5, 19, 41, 109... (R. Sedgewick, 1986), która wyraża się wzorem:

$$h_i \; = \; \begin{cases} 9 \cdot 2^i - 9 \cdot 2^{i/2} + 1 & \text{dla parzystych } i \\ 8 \cdot 2^i - 6 \cdot 2^{(i+1)/2} + 1 & \text{dla nieparzystych } i \end{cases}$$

Twierdzenie 8 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Sedgewicka (1, 5, 19, 41, 109...) będzie działało w czasie* $O(n^{4/3})$.

Dużo lepszy wynik można uzyskać, gdy wykorzysta się fakt, że sortowanie przez wstawianie zastosowane do ciągu 2-uporządkowanego i 3-uporządkowanego będzie działało w liniowym czasie.

Fakt 9 *Jeśli zastosujemy metodę sortowania przez proste wstawianie do ciągu, który jest jednocześnie 2-posortowany i 3-posortowany, to każdy element przesunie się co najwyżej o jedną pozycję.*

Jeśli ciąg jest 4- i 6-uporządkowany, to 2-sortowanie wykona się na tym ciągu w liniowym czasie; a jeśli ciąg jest 6- i 9-uporządkowany, to również w liniowym czasie wykona się na tym ciągu 3-sortowanie. Kontynuując to rozumowanie można skonstruować następującą sekwencję przyrostów (V. R. Pratt, 1971): 1, 2, 3, 4, 6, 9, 8, 12, 18, 27...

						1								
				2				3						
		4				6				9				
	8			12				18				27		
16		24				36				54				81

Wartości ciągu przyrostów Pratta można odczytać z powyższego trójkąta, a ogólna postać każdego wyrazu to:

$$h_{(i^2+i)/2+j} \;=\; 2^{i-j} \cdot 3^j \quad \text{dla } i = 0, 1 \ldots \text{ oraz } j = 0 \ldots i$$

Twierdzenie 10 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Pratta (1, 2, 3, 4, 6, 9, 8, 12, 18, 27...) będzie działało w czasie* $O(n \log^2 n)$.

3 Scalanie

Scalanie (ang. *merging*) to łączenie dwóch posortowanych ciągów w jeden posortowany ciąg. Problem ten można zdefiniować następująco.

Dane: Mamy dwie tablice $V[0 \ldots p-1]$ i $W[0 \ldots q-1]$ i elementy w obu tablicach są uporządkowane, czyli $V[0] \leq \ldots \leq V[p-1]$ i $W[0] \leq \ldots \leq W[q-1]$.

Zadanie: Dane z obu tablic należy uporządkować.

Ograniczenia: Porządkując dane możemy jedynie porównywać elementy i przepisywać je bądź zamieniać miejscami.

3.1 Scalanie tradycyjne

Tradycyjna procedura scalająca działa podobnie do sortowania przez wybór. W każdym kroku wybierany jest najmniejszy element spośród obu ciągów i przenoszony do tablicy wynikowej; w ten sposób rozmiar problemu jest redukowany o 1 w każdym przebiegu pętli.

```
procedure TRADIT–MERGE (key V[0...p−1], key W[0...q−1], key R[0...p+q−1])
{
    i ← 0 ; // wartownik w tablicy V
    j ← 0 ; // wartownik w tablicy W
    while i < p ∧ j < q do
        if V[i] ≤ W[j]
            then { R[i+j] ← V[i] ; i++; }
            else { R[i+j] ← W[j] ; j++; }
    while i < p do { R[i+j] ← V[i] ; i++; }
    while j < q do { R[i+j] ← W[j] ; j++; }
}
```

Scalanie tradycyjne jest stabilne, ale nie działa w miejscu. Podczas działania tego algorytmu wykonywanych jest $p+q$ przepisań elementów i co najwyżej $p+q-1$ porównań. Czas działania scalania tradycyjnego jest więc liniowy $\Theta(p+q)$.

3.2 Scalanie Kronroda

Oryginalna procedura Kronroda scala ciągi o dowolnych długościach. Tutaj zostanie przedstawiona uproszczona wersja tego algorytmu, co pozwoli skupić się na idei samego scalania a nie na szczegółach technicznych.

Dane: Mamy tablicę $T[0 \ldots p+q-1]$, w której dwie części tej tablicy są posortowane, czyli $T[0] \leq \ldots \leq T[p-1]$ oraz $T[p] \leq \ldots \leq T[p+q-1]$. Dodatkowo $r = \sqrt{p+q}$ jest liczbą naturalną, oraz wielkości posortowanych fragmentów są podzielne przez r, czyli $r|p$ i $r|q$.

---

procedure SIMPLE–KRONROD–MERGE (**key** $T[0 \ldots p+q-1]$)
{
  $r \leftarrow \sqrt{p+q}$; // *wielkość bloku*
  ⟨*podziel dane w tablicy* T *na bloki o rozmiarze* r*,*
    polem kluczowym w każdym bloku niech będzie pierwszy najmniejszy element⟩;
  ⟨*posortuj bloki metodą* SELECTION–SORT ⟩;
  for $i = 1 \ldots r-3$ **do**
    ⟨*scal dwa sąsiednie bloki* $(i-1)$*-szy oraz* i*-ty,*
      wykorzystując dwa ostatnie bloki jako bufor na wynik ⟩;
  ⟨*posortuj trzy ostatnie bloki metodą* SELECTION–SORT⟩;
}

Scalanie Kronroda działa w miejscu, ale nie jest stabilne. Czas działania algorytmu jest liniowy $\Theta(p+q)$.

3.3 Sortowanie przez scalanie

Sortowanie przez scalanie (ang. *merge–sort*) jest algorytmem rekurencyjnym. Metoda ta dzieli dane na dwie równe (z dokładnością do 1) części, sortuje każdą z nich (rekurencyjnie) a na koniec scala posortowane części (wykorzystując określoną procedurę scalającą).

---

procedure MERGE–SORT (**key** $T[0 \ldots n-1]$)
{
  if $n < 2$ **then return**;
  $m \leftarrow \lfloor \frac{n}{2} \rfloor$; // *środek tablicy*
  MERGE–SORT $(T[0 \ldots m-1])$;
  MERGE–SORT $(T[m \ldots n-1])$;
  MERGE $(T[0 \ldots m-1], T[m \ldots n-1])$; // *określona procedura scalająca*
}

Wiele własności sortowania przez scalanie zależy od zastosowanej procedury scalającej. Stabilność tego algorytmu zależy od stabilności scalania. Czas potrzebny na wykonanie sortowania przez scalanie także zależy od czasu scalania; gdy założymy, że scalanie działa w czasie $O(n)$ to czas działania sortowania wyraża się następującą zależnością rekurencyjną:

$$\begin{cases} T(0) = T(1) &= O(1) \\ T(n) &= T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n) \quad \text{dla} \quad n \geq 1 \end{cases}$$

Tak więc czas działania tego algorytmu wynosi $O(n \log n)$.

4 Podział

Podział (ang. *partition*) danych według zadanej wartości polega na odseparowaniu elementów mniejszych od wartości dzielącej i elementów większych od niej. Wartość, według której dzielimy zbiór z danymi nazywa się ***piwotem*** (ang. *pivot*). Dokładna definicja tego problemu jest następująca.

Dane: W n-elementowej tablicy $T[0 \ldots n-1]$ zapisany jest nieuporządkowany ciąg wartości $a_0, \ldots, a_{n-1}$. Zadany jest także piwot *piv*, według którego będzie dokonywany podział tablicy.

Zadanie: Dane w tej tablicy należy rozdzielić na elementy mniejsze od piwota i większe.

Ograniczenia: Rozdzielając dane możemy jedynie porównywać elementy i przepisywać je bądź zamieniać miejscami.

Rozważając dalej problem podziału, ograniczymy postać danych do przypadku, w którym piwot jest pierwszym elementem tablicy.

Dane: W n-elementowej tablicy $T[0 \ldots n-1]$ zapisany jest nieuporządkowany ciąg wartości $a_0, \ldots, a_{n-1}$. Piwot, według którego będzie dokonywany podział tablicy, jest pierwszym elementem w tablicy $piv = T[0]$.

4.1 Podział Lomuta

Bardzo prosty algorytm przeglądający tablicę z jednej strony. W trakcie działania przenosi on elementy mniejsze od piwota do bloku na początku tablicy.

```
procedure LOMUTO–PARTITON (key T[0 ... n-1]) ↦ int
{
    piv ← T[0] ; // piwot
    l ← 1 ; // wartownik bloku elementów mniejszych od piwota
    for i = 1 ... n-1 do
        if T[i] < piv then { T[l] ↔ T[i] ; l++ ; }
    T[0] ↔ T[l-1] ;
    return l-1 ;
}
```

Algorytm Lomuta nie jest stabilny i działa w miejscu. Wykonuje on $n-1$ porównań, więc jego czas działania jest liniowy $O(n)$.

4.2 Podział Sedgewicka

Prosty algorytm przeglądający tablicę z obu stron. W trakcie działania przenosi on elementy mniejsze od piwota do bloku na początku tablicy a większe do bloku na końcu tablicy.

```
procedure SEDGEWICK–PARTITON (key T[0 ... n − 1]) ↦ int
{
    piv ← T[0] ; // piwot
    l ← 0 ; // wartownik bloku elementów nie większych od piwota
    g ← n ; // wartownik bloku elementów nie mniejszych od piwota
    loop
    {
        do g − − ; while T[g] > piv;
        if g = l then break;
        do l + + ; while T[l] < piv;
        if g ≤ l then break;
        T[l] ↔ T[g] ;
    }
    T[0] ↔ T[g] ;
    return g ;
}
```

Algorytm Sedgewicka nie jest stabilny i działa w miejscu. Wykonuje on co najwyżej $n + 1$ porównań, więc jego czas działania jest liniowy $O(n)$.

4.3 Sortowanie szybkie (przez podział)

Sortowanie szybkie (ang. *quick–sort*) nazywane również ***sortowaniem przez podział*** jest algorytmem rekurencyjnym. Metoda ta najpierw dokonuje podziału danych (wykorzystując określoną procedurę dzielącą) względem losowo wybranej spośród danych wartości, a potem każdą z wydzielonych części sortuje (rekurencyjnie).

```
procedure QUICK–SORT (key T[0 ... n − 1])
{
    if n < 2 then return;
    piv ← T[⟨losowa w artość z zakresu 0 ... n − 1⟩] ; // losowy wybór piwota
    m ← PARTITION (T, piv) ; // podział tablicy według piwota
    QUICK–SORT (T[0 ... m − 1]) ;
    QUICK–SORT (T[m + 1 ... n − 1]) ;
}
```

Wiele własności sortowania szybkiego zależy od zastosowanej procedury dzielącej. Stabilność tego algorytmu zależy od stabilności podziału. Czas potrzebny na wykonanie sortowania przez podział także zależy od czasu dzielenia; gdy założymy, że dzielenie działa w czasie $O(n)$ to czas działania sortowania wyraża się następującą zależnością rekurencyjną:

$$\begin{cases} T(0) = T(1) &= O(1) \\ T(n) &= T(m) + T(n-m-1) + O(n) \quad \text{dla} \quad n \geq 1 i m \in \{0, \ldots, n-1\} \end{cases}$$

Tak więc czas działania sortowania szybkiego wynosi w najgorszym przypadku $O(n^2)$, ale w przypadku średnim (każdy możliwy podział jest jednakowo prawdopodobny) wynosi $O(n \log n)$.