Sortowanie szybkie

Quick Sort

Algorytm sortowania szybkiego opiera się na strategii "dziel i zwyciężaj" (ang. divide and conquer), którą możemy krótko scharakteryzować w trzech punktach:

1. **DZIEL** - problem główny zostaje podzielony na podproblemy
2. **ZWYCIĘŻAJ** - znajdujemy rozwiązanie podproblemów
3. **POŁĄCZ** - rozwiązania podproblemów zostają połączone w rozwiązanie problemu głównego

Idea sortowania szybkiego jest następująca:

(DZIEL) :	najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej lewą partycją) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej prawą partycją).
(ZWYCIĘŻAJ) :	każdą z partycji sortujemy rekurencyjnie tym samym algorytmem.
(POŁĄCZ) :	połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany.

Sortowanie szybkie zostało wynalezione przez angielskiego informatyka, profesora Tony'ego Hoare'a w latach 60-tych ubiegłego wieku. W przypadku typowym algorytm ten jest najszybszym algorytmem sortującym z klasy złożoności obliczeniowej $O(n \log n)$ - stąd pochodzi jego popularność w zastosowaniach. Musimy jednak pamiętać, iż w pewnych sytuacjach (zależnych od sposobu wyboru piwotu oraz niekorzystnego ułożenia danych wejściowych) klasa złożoności obliczeniowej tego algorytmu może się degradować do $O(n^2)$, co więcej, poziom wywołań rekurencyjnych może spowodować przepełnienie stosu i zablokowanie komputera. Z tych powodów algorytmu sortowania szybkiego nie można stosować bezmyślnie w każdej sytuacji tylko dlatego, iż jest uważany za jeden z najszybszych

prof. Tony Hoare

algorytmów sortujących - zawsze należy przeprowadzić analizę możliwych danych wejściowych właśnie pod kątem przypadku niekorzystnego - czasem lepszym rozwiązaniem może być zastosowanie algorytmu **sortowania przez kopcowanie**, który nigdy nie degraduje się do klasy $O(n^2)$.

Tworzenie partycji

Do utworzenia partycji musimy ze zbioru wybrać jeden z elementów, który nazwiemy **piwotem**. W lewej partycji znajdą się wszystkie elementy niewiększe od piwotu, a w prawej partycji umieścimy wszystkie elementy niemniejsze od piwotu. Położenie elementów równych nie wpływa na proces sortowania, zatem mogą one występować w obu partycjach. Również porządek elementów w każdej z partycji nie jest ustalony.

Jako piwot można wybierać element pierwszy, środkowy, ostatni, medianę lub losowy. Dla naszych potrzeb wybierzemy element środkowy:

```
piwot  d[(lewy + prawy) div 2]
```

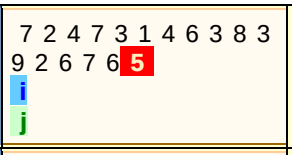
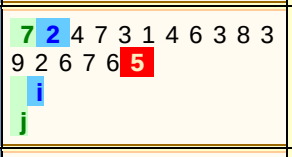
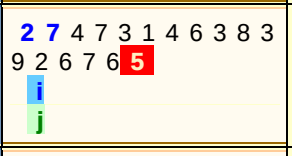
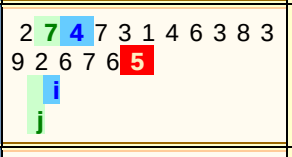
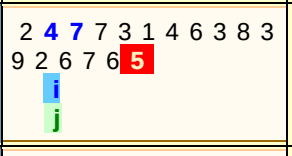
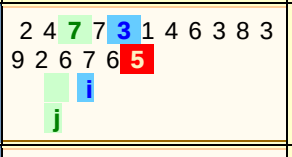
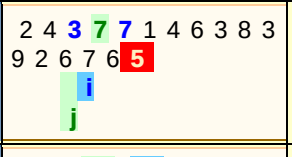
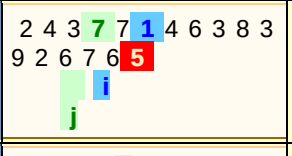
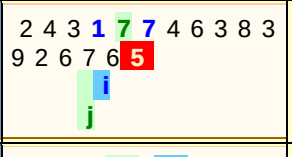
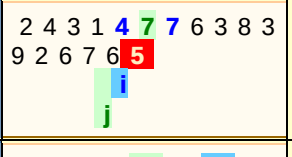
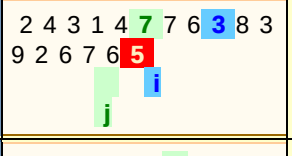
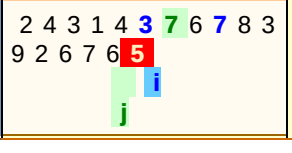
piwot - element podziałowy
d[ ] - dzielony zbiór
lewy - indeks pierwszego elementu
prawy - indeks ostatniego elementu

Dzielenie na partycje polega na umieszczeniu dwóch wskaźników na początku zbioru - *i* oraz *j*. Wskaźnik *i* przebiega przez zbiór poszukując wartości mniejszych od *piwotu*. Po znalezieniu takiej wartości jest ona wymieniana z elementem na pozycji *j*. Po tej operacji wskaźnik *j* jest przesuwany na następną pozycję. Wskaźnik *j* zapamiętuje pozycję, na którą trafi następny element oraz na końcu wskazuje miejsce, gdzie znajdzie się *piwot*. W trakcie podziału *piwot* jest bezpiecznie przechowywany na ostatniej pozycji w zbiorze.

Dla przykładu podzielimy na partycje zbiór:

{ 7 2 4 7 3 1 4 6 5 8 3 9 2 6 7 6 3 }

Lp.	Operacja	Opis
1.	7 2 4 7 3 1 4 6 **5** 8 3 9 2 6 7 6 3	Wyznaczamy na piwot element środkowy.
2.	7 2 4 7 3 1 4 6 **3** 8 3 9 2 6 7 6 **5**	Piwot wymieniamy z ostatnim elementem zbioru

3.		Na początku zbioru ustawiamy dwa wskaźniki. Wskaźnik *i* będzie przeglądał zbiór do przedostatniej pozycji. Wskaźnik *j* zapamiętuje miejsce wstawiania elementów mniejszych od piwotu
4.		Wskaźnikiem *i* szukamy elementu mniejszego od piwotu
5.		Znaleziony element wymieniamy z elementem na pozycji *j*-tej. Po wymianie wskaźnik *j* przesuwamy o 1 pozycję.
6.		Szukamy
7.		Wymieniamy i przesuwamy *j*.
8.		Szukamy
9.		Wymieniamy i przesuwamy *j*.
10.		Szukamy
11.		Wymieniamy i przesuwamy *j*.
12.		Szukamy
13.		Wymieniamy i przesuwamy *j*.
14.		Szukamy
15.		Wymieniamy i przesuwamy *j*.

16.	2 4 3 1 4 3 **7** 6 7 8 **3** 9 2 6 7 6 **5** i j	Szukamy
17.	2 4 3 1 4 3 **3** **6** 7 8 **7** 9 2 6 7 6 **5** i j	Wymieniamy i przesuwamy *j*.
18.	2 4 3 1 4 3 3 **6** 7 8 7 9 **2** 6 7 6 **5** i j	Szukamy
19.	2 4 3 1 4 3 3 **2** **7** 8 7 9 **6** 6 7 6 **5** i j	Wymieniamy i przesuwamy *j*.
20.	2 4 3 1 4 3 3 2 **5** 8 7 9 6 6 7 6 7 ^ i Lewa partycja j Prawa partycja	Brak dalszych elementów do wymiany. *Piwot* wymieniamy z elementem na pozycji *j*-tej. Podział na partycje zakończony.

Po zakończeniu podziału na partycje wskaźnik *j* wyznacza pozycję piwotu. Lewa partycja zawiera elementy mniejsze od *piwotu* i rozciąga się od początku zbioru do pozycji *j* - 1. Prawa partycja zawiera elementy większe lub równe *piwotowi* i rozciąga się od pozycji *j* + 1 do końca zbioru. Operacja podziału na partycje ma liniową klasę złożoności obliczeniowej - $O(n)$.

Specyfikacja problemu

Sortuj_szybko(*lewy, prawy*)

Dane wejściowe

d[ ] - Zbiór zawierający elementy do posortowania. Zakres indeksów elementów jest dowolny.

lewy - indeks pierwszego elementu w zbiorze, lewy $\in$ C

prawy - indeks ostatniego elementu w zbiorze, prawy $\in$ C

Dane wyjściowe

d[ ] - Zbiór zawierający elementy posortowane rosnąco

Zmienne pomocnicze

piwot - element podziałowy

i, j - indeksy, i, j ∈ C

Lista kroków

Algorytm sortowania szybkiego wywołujemy podając za *lewy* indeks pierwszego elementu zbioru, a za *prawy* indeks elementu ostatniego (czyli *Sortuj_szybko*(1,*n*)). Zakres indeksów jest dowolny - dzięki temu ten sam algorytm może również sortować fragment zbioru, co wykorzystujemy przy sortowaniu wyliczonych partycji.

K01: $i \leftarrow \left[\frac{lewy + prawy}{2}\right]$

K02: piwot ← d[i]; d[i] ← d[prawy]; j ← lewy

K03: Dla i = lewy, lewy + 1, ..., prawy - 1: wykonuj kroki K04...K05

K04: Jeśli d[i] ≥ piwot, to wykonaj kolejny obieg pętli K03

K05: d[i] ↔ d[j]; j ← j + 1

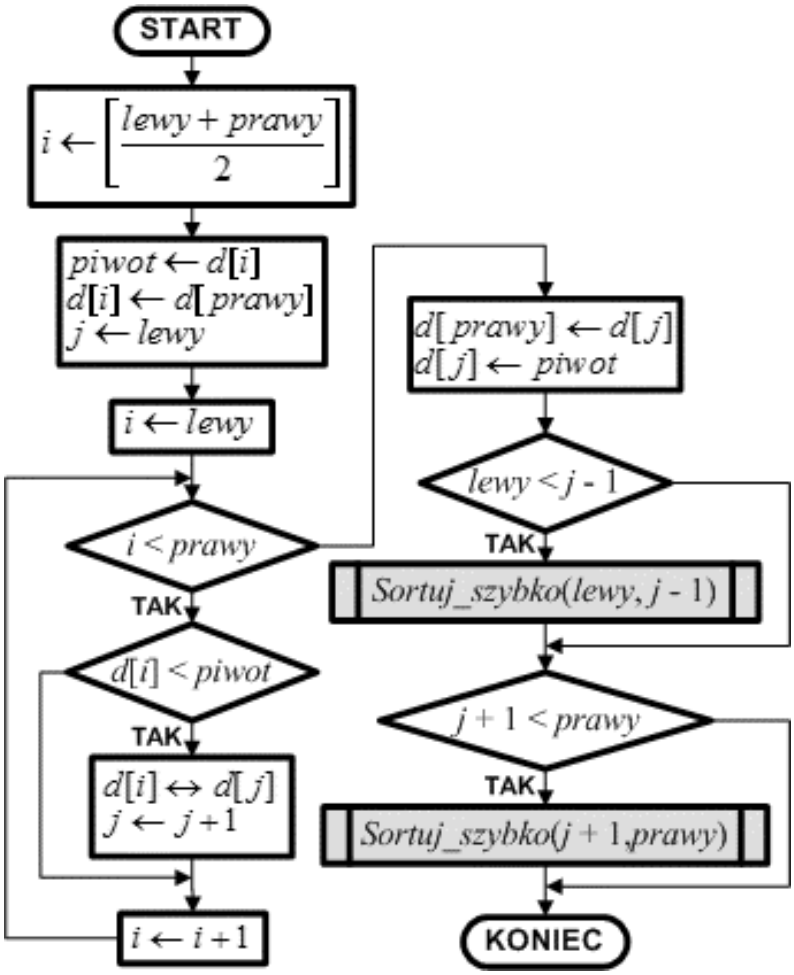
K06: d[prawy] ← d[j]; d[j] ← piwot

K07: Jeśli lewy < j - 1, to wykonaj rekurencyjnie **Sortuj_szybko**(lewy, j - 1)

K08: Jeśli j + 1 < prawy, to wykonaj rekurencyjnie **Sortuj_szybko**(j + 1, prawy)

K09: Zakończ algorytm

Schemat blokowy



Na element podziałowy wybieramy element leżący w środku dzielonej partycji. Wyliczamy jego pozycję i zapamiętujemy ją tymczasowo w zmiennej *i*. Robimy to po to, aby dwukrotnie nie wykonywać tych samych rachunków.

Element *d*[*i*] zapamiętujemy w zmiennej piwot, a do *d*[*i*] zapisujemy ostatni element partycji. Dzięki tej operacji *piwot* został usunięty ze zbioru.

Ustawiamy zmienną *j* na początek partycji. Zmienna ta zapamiętuje pozycję podziału partycji.

W pętli sterowanej zmienną *i* przeglądamy kolejne elementy od pierwszego do przedostatniego (ostatni został umieszczony na pozycji piwotu, a piwot zapamiętany). Jeśli *i*-ty element jest mniejszy od *piwotu*, to trafia on na początek partycji - wymieniamy ze sobą elementy na pozycjach *i*-tej i *j*-tej. Po tej operacji przesuwamy punkt podziałowy partycji *j*.

Po zakończeniu pętli element z pozycji *j*-tej przenosimy na koniec partycji, aby zwolnić miejsce dla *piwotu*, po czym wstawiamy tam *piwot*. Zmienna *j* wskazuje zatem wynikową pozycję *piwotu*. Pierwotna partycja została podzielona na dwie partycje:

partycja lewa od pozycji *lewy* do *j* - 1 zawiera elementy mniejsze od piwotu
partycja prawa od pozycji *j* + 1 do pozycji *prawy* zawiera elementy większe lub równe piwotowi.

Sprawdzamy, czy partycje te obejmują więcej niż jeden element. Jeśli tak, to wywołujemy rekurencyjnie algorytm sortowania szybkiego przekazując mu granice wyznaczonych partycji. Po powrocie z wywołań rekurencyjnych partycja wyjściowa jest posortowana rosnąco. Kończymy algorytm.