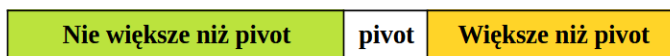
Jestem szybki, więc sortuję

#sortowanie #dziel_i_zwyciężaj
#rekurencja #pivot

Przyszedł czas na omówienie algorytmu sortowania szybkiego (*Quick Sort*) – rekurencyjnego algorytmu opartego na zasadzie **divide and conquer** (*dziel i zwyciężaj*). Sortowana tablica jest dzielona na dwie rozłączne części (odcinki) w oparciu o tak zwany **pivot** (ang. *sworzeń*). Wszystkie elementy tablicy będą porównane z *pivotem* i zaliczone do odpowiedniego odcinka:



Pivot pozostanie już na tej pozycji do końca sortowania całej tablicy, natomiast odcinki zaznaczone na kolor zielony i pomarańczowy zostaną przekazane do rekurencyjnych potomków funkcji sortującej, ponieważ kolejność elementów wewnątrz każdego z odcinków nie jest jeszcze ustalona i może być daleka od ostatecznego uporządkowania.

Algorytm sortowania szybkiego został po raz pierwszy opublikowany przez Tony'ego Hoare'a, a autor *Kodowania z Kocurrem* jest rówieśnikiem opisywanego algorytmu. Wiele zasług dla rozwoju tego algorytmu położył również Robert Sedgewick między innymi w swej pracy doktorskiej z 1975 roku.

Sortowanie szybkie przećwiczymy na poniższej tablicy o rozmiarze n (tutaj $n = 8$), a dokładniej – wektorze liczb całkowitych A – który docelowo posortujemy według porządku niemalejącego:

```
int n;
vector<int> A;
```

Obydwie te zmienne zadeklarawane są jako globalne, aby uprościć wywołania funkcji sortujących.

10	1	3	2	9	1	7	6
0	1	2	3	4	5	6	7

Główna procedura sortująca `q_sort()` jest rekurencyjna i jej argumenty oznaczają odcinek sortowanego wektora:

```
void q_sort(int start, int finish)
{
  // . . .
}
```

Na przykład dla $start = 1$ oraz $finish = 5$ sortowany będzie wycinek o indeksach od 1 do 4 (czyli $finish - 1$):

10	1	3	2	9	1	7	6
0	1	2	3	4	5	6	7
	start				finish		

Zatem posortowanie całego wektora odbędzie się, jeśli wywołamy funkcję `q_sort()` z następującymi argumentami:

```
q_sort(0, n);
```

Zadaniem tej funkcji będzie „kierowanie ruchem” podczas sortowania. Jako szanująca się funkcja rekurencyjna musi zawierać ona na początku sprawdzenie warunku zakończenia rekurencji. Można streścić to tak: *nie sortujemy, jeśli nie ma co sortować* – to znaczy jeśli odcinek wektora do posortowania jest pusty lub jednoelementowy:

```
if(start >= finish-1 || start < 0)
    return;
```

W przeciwnym razie wywoływana jest funkcja `q_partition()`, której rezultatem jest położenie (indeks) pewnego szczególnego elementu sortowanego wektora o wdzięcznej nazwie *pivot*:

```
int p = q_partition(start, finish);
```

Pivot stanowi punkt podziału wektora A: elementy nie większe od niego znajdują się na lewo, natomiast elementy większe zostały przesunięte na prawo.* Dzięki temu można kontynuować sortowanie oddzielnie w dwóch odcinkach – przed i za *pivotem*:

```
q_sort(start, p);
q_sort(p+1, finish);
```

Sam *pivot* posiada indeks p i jego pozycja nie ulegnie już zmianie do końca sortowania.

Widać zatem, że kluczowe dla poprawności i efektywności algorytmu jest działanie funkcji `q_partition()`, którą się teraz właśnie zajmiemy:

```
int q_partition(int start, int finish)
{
  // . . .
}
```

*Przypomina to nieco rolę spełnianą przez *medianę* zbioru danych. *Pivot* nie musi być równy medianie, choć dobrze jest, jeśli tak się dzieje – wtedy algorytm sortowania szybkiego działa najefektywniej (jeszcze do tego wrócimy).

Jako *pivot* wybieramy ostatni element w rozważanym odcinku wektora.[†] Pomocnicza zmienna i wskaże nam indeks miejsca, gdzie będziemy wstawiać kolejny element nieprzewyższający *pivota* – na początku ustawiamy ją *przed* indeksem *start*:

```
int pivot = A[finish-1];
int i = start-1;
```

Pierwsze wywołanie funkcji odbywa się dla całego wektora, zatem $start = 0$ oraz $finish = 8$. *Pivot* oznaczony jest kolorem różowym z czerwoną wartością 6, natomiast indeks i oznaczony jest pomarańczową gwiazdką:

	10	1	3	2	9	1	7	6
*	0	1	2	3	4	5	6	7

Głównym składnikiem funkcji jest pętla przebiegająca po wszystkich elementach odcinka wektora: od początku do *pivota* (ale z jego wyłączeniem):

```
for(int j=start; j<finish-1; j++)
  // . . .
```

Elementy wskazywane przez indeks j są oznaczone kolorem niebieskim. Dla $j = 0$ mamy:

	10	1	3	2	9	1	7	6
*	0	1	2	3	4	5	6	7

Każdy taki element $A[j]$ jest porównywany z *pivotem*: jeśli jest mniejszy lub mu równy, wtedy jest przesuwany na pozycję wskazywaną przez indeks i (który *przed* tą operacją jest zwiększany o 1):

```
for(int j=start; j<finish-1; j++)
  if(A[j] <= pivot)
  {
    i++;
    swap(A[i], A[j]);
  }
```

W tym przypadku porównywany element ma wartość 10, więc nic się nie dzieje i przechodzimy do następnego obiegu pętli ($j = 1$). Tutaj mamy wartość $A[1] = 1$, mniejszą od *pivota*, zatem powiększamy zmienną i do wartości 0 (gwiazdka przesuwa się w prawo)...

10	1	3	2	9	1	7	6
0 *	1	2	3	4	5	6	7

[†]Będziemy jeszcze o tym mówić.

...zamieniamy miejscami $A[0] \leftrightarrow A[1]$ i przechodzimy do następnego obiegu pętli:

1	10	3	2	9	1	7	6
0	1*	2	3	4	5	6	7

Element oznaczony kolorem zielonym znajduje się już na właściwym miejscu, a dokładniej: nie będzie przesuwany aż do końca pracy tej pętli. Tutaj mamy $j = 2$ oraz $A[2] = 3$, co jest mniejsze od *pivota*, więc gwiazdka znów nam przeskoczyła w prawo ($i = 1$) i musimy zamienić miejscami $A[1] \leftrightarrow A[2]$.

W następnym obiegu pętli $j = 3$ i znów mamy element $A[3] = 2$ mniejszy od *pivota*:

1	3	10	2	9	1	7	6
0	1	2*	3	4	5	6	7

Gwiazdka (indeks i) przemieściła się w prawo, zamieniamy miejscami $A[2] \leftrightarrow A[3]$ i mamy następny obieg pętli:

1	3	2	10	9	1	7	6
0	1	2*	3	4	5	6	7

Teraz mamy na tapecie element $A[4]$ o wartości 9, co jest mniejsze od *pivota*, więc gładko przechodzimy do następnego obiegu pętli ($j = 5$):

1	3	2	10	9	1	7	6
0	1	2	3*	4	5	6	7

Gwiazdka przeskoczyła nam na pozycję 3 i zamieniamy miejscami $A[3] \leftrightarrow A[5]$. W następnym obiegu pętli widzimy wartość 7, zatem nie mamy przestawienia elementów:

1	3	2	1	9	10	7	6
0	1	2	3*	4	5	6	7

Pętla po zmiennej j się kończy i teraz musimy ustawić *pivot* na właściwym miejscu. Już poza pętlą wykonujemy następujące instrukcje:

```
i++;
swap(A[i], A[finish-1]);
return i;
```

Funkcja zwraca wartość indeksu odpowiadającego położeniu *pivota* – tutaj jest to wartość 4. Zauważmy, że na prawo od niego znajdują się elementy większe od 6 (niekoniecznie posortowane):

1	3	2	1	6	10	7	9
0	1	2	3	4*	5	6	7

Dla porządku przedstawimy teraz funkcje `q_partition()` oraz `q_sort()` w całości, po czym prześledzimy, jak radzą sobie rekurencyjni potomkowie tych funkcji:

```
int n;
vector<int> A;

int q_partition(int start, int finish)
{
  int pivot = A[finish-1];
  int i = start-1;
  for(int j=start; j<finish-1; j++)
    if(A[j] <= pivot)
    {
      i++;
      swap(A[i], A[j]);
    }
  i++;
  swap(A[i], A[finish-1]);
  return i;
}

void q_sort(int start, int finish)
{
  if(start >= finish-1 || start < 0)
    return;
  int p = q_partition(start, finish);
  q_sort(start, p);
  q_sort(p+1, finish);
}
```

Ustalenie rozmiaru wektora A oraz wczytanie danych do niego należy przeprowadzić przed rozpoczęciem sortowania.

Pivot ulokowany jako $A[4]$ podczas pierwszego wywołania funkcji `q_partition()` pozostanie już na tym miejscu do samego końca algorytmu, a następna generacja funkcji potomnych będzie operować na odcinkach odpowiednio: od $A[0]$ do $A[3]$ włącznie oraz od $A[5]$ do $A[7]$ włącznie.

Pierwszy potomek `q_partition()` otrzymuje argumenty $start = 0$ oraz $finish = 4$. W roli *pivota* występuje ostatni element o wartości 1:

	1	**3**	**2**	**1**
*	0	1	2	3

Gwiazdka ustawiona przed indeksem *start* i rusza pętla po j:

1	**3**	**2**	**1**
0*	1	2	3

Element $A[0]$ nie przewyższa *pivota*, więc przesuwamy gwiazdkę i zamieniamy miejscami $A[0] \leftrightarrow A[0]$, czyli nie mamy żadnego przestawienia i przechodzimy do $j = 1$:

1	**3**	**2**	**1**
0*	1	2	3

Element $A[1] = 2$ jest większy od *pivota*, więc gwiazdka zostaje tam, gdzie była i przechodzimy do $j = 2$:

1	**3**	**2**	**1**
0*	1	2	3

Znowu to samo: element $A[2] = 3$ przewyższa *pivota*, więc pętla po j dochodzi do końca. Przesuwamy gwiazdkę w prawo i zamieniamy miejscami *pivot* i element oznaczony gwiazdką:

1	**1**	**2**	**3**
0	1*	2	3

Rekurencyjni potomkowie tej instancji funkcji `q_sort()` będą sortować odpowiednio: wycinek od $A[0]$ do $A[0]$ oraz od $A[2]$ do $A[3]$. Pierwszy wymieniony wycinek jest jednoelementowy, a drugi jest już posortowany, więc nic się w nich nie będzie działo (co łatwo prześledzić).

Pozostaje nam przeanalizować działanie funkcji `q_sort()` na prawym wycinku wektora: *start* $= 5$ oraz *finish* $= 8$. Mamy tutaj trzy elementy, a *pivot* ma wartość 9:

	10	**7**	**9**
*	5	6	7

Zmienna j zaczyna od wartości 5 i nic się nie dzieje, bo $A[5] = 10$ jest większe od *pivota*:

	10	**7**	**9**
*	5	6	7

Za to dla $j = 6$ mamy ruch w interesie: gwiazdka przesuwa się w prawo...

10	**7**	**9**
*5	6	7

...i zamieniamy miejscami $A[6] \leftrightarrow A[5]$:

7	**10**	**9**
*5	6	7

Pętla po j się kończy, gwiazdkę przesuwamy w prawo i zamieniamy miejscami *pivot* z elementem $A[6]$:

7	9	10
5	*6	7

Do potomnych funkcji powędrują jednoelementowe wycinki wektora (od $A[5]$ do $A[5]$ oraz od $A[7]$ do $A[7]$), zatem kwestię sortowania należy uznać za rozwiązaną:

1	1	2	3	6	7	9	10
0	1	2	3	4	5	6	7

Złożoność obliczeniowa tej metody sortowania istotnie zależy od sposobu wybierania kluczowego elementu, jakim jest *pivot*. Jeśli dane w sortowanym wektorze ułożone są losowo, wtedy możemy spodziewać się dobrej złożoności $O(n \lg n)$ (nie będziemy tego dowodzić). Znamy inne algorytmy sortowania o podobnej złożoności (*Merge Sort* czy *Heap Sort*) i w tej kategorii *Quick Sort* jest w czołówce, zwłaszcza dla danych rozproszonych. Jednak w sytuacji, gdy wektor z danymi jest już posortowany (lub prawie posortowany), wtedy algorytm „kwadraci się”, czyli działa ze złożonością zbliżoną do $O(n^2)$. W następnej króciutkiej sekcji podajemy sugestie, jak sobie poradzić w tym problemem.

Dodajmy jeszcze, że algorytm sortowania szybkiego **nie jest** algorytmem stabilnym, to znaczy, że elementy o tej samej wartości mogą znaleźć się w końcowym porządku w innej kolejności, niż były w początkowym, nieposortowanym wektorze. Jeśli koniecznie zależy nam na stabilności sortowania, wtedy powinniśmy dołożyć dodatkowe kryterium przy porównywaniu elementów (*mniejszy-większy*) – na przykład wartość indeksu w sortowanej tablicy.

Wybór *pivota*

Jeśli jako *pivot* wybieramy zawsze ostatni element w przetwarzanym odcinku wektora, wtedy ryzykujemy nieefektywność naszej metody. Nietrudno zauważyć, że *pivot* powinien mieć wartość zbliżoną do *mediany* zbioru sortowanych elementów – wtedy na koniec działania funkcji `q_sort()` byłby umieszczany mniej więcej w środku odcinka wektora, a funkcje potomne otrzymywałyby „połówki” odcinka do posortowania.[‡]

Jeden sposób polega na losowym wyborze indeksu z zakresu *start* oraz $finish - 1$. Element o wybranym indeksie będzie naszym *pivotem* – możemy go zamienić miejscami z ostatnim elementem w odcinku i zastosować resztę przedstawionego wyżej algorytmu bez zmian. Nawet, jeśli dane są początkowo ułożone według jakiegoś porządku, wtedy losowość w wyborze *pivota* powinna dać znaczącą poprawę efektywności algorytmu.

[‡]Zasada *divide and conquer* prowadzi do najefektywniejszego algorytmu, jeśli dzielimy zbiór danych na połowy – no, w przybliżeniu na połowy.

Idealnie byłoby, gdybyśmy mogli szybko znaleźć medianę naszego odcinka wektora i uznać ją za *pivot*. Niestety znajdowanie mediany w ogólnym przypadku ma złożoność liniową, a nam potrzebna jest złożoność $O(1)$. Można wszelako znaleźć taką *medianę dla ubogich*:[§] wziąć pierwszy, środkowy i ostatni element z naszego odcinka i z nich wyciągnąć medianę. Mogłoby to wyglądać tak:

```
int center = (start + finish - 1) / 2
if(A[center] < A[start])
    swap(A[start], A[center]);
if(A[finish-1] < A[start])
    swap(A[start], A[finish-1]);
if(A[center] < A[finish-1])
    swap(A[center], A[finish-1]);
int pivot = A[finish-1];
```

Tak na szybko, to byłoby na tyle.

[§]Zwaną oryginalnie *median-of-three*. Tę metodę zaproponował Robert Sedgewick w 1998 roku.