

sortowania elementów przechowywanych w pamięci; jeśli chodzi o sortowanie elementów zapisanych w plikach dyskowych, to istnieją inne algorytmy, które operacje te wykonują bardziej efektywnie). Algorytm quicksort został opracowany przez C.A.R Hoare w 1962 roku.

Aby zrozumieć ten sposób sortowania, należy znać algorytm podziału danych, przedstawiony w poprzedniej części rozdziału. Najprościej rzecz ujmując, zasada działania quicksortu opiera się na podziale tablicy na dwie podtablice oraz rekurencyjnym wywołaniu tej samej metody w celu posortowania uzyskanych podtablic. Ten prosty schemat można wzbogacić pewnymi dodatkowymi elementami. Są one związane z wyborem wartości osiowej oraz sposobem sortowania niewielkich grup danych. Wszystkie te dodatkowe usprawnienia zostały omówione w dalszej części rozdziału, po przedstawieniu podstawowej wersji głównego algorytmu.

Bardzo trudno jest zrozumieć, co robi algorytm quicksort, jeśli wcześniej nie dowiemy się, jak operacje te są wykonywane. Dlatego zmienimy standardową kolejność prezentowania zagadnień i w pierwszej kolejności zostanie przedstawiony kod algorytmu quicksort, a dopiero potem aplet demonstracyjny *QuickSort1*.

Algorytm quicksort

Podstawowa wersja rekurencyjnej metody implementującej algorytm quicksort jest stosunkowo prosta. Poniżej został przedstawiony jej przykładowy kod:

```
public void recQuickSort(int left, int right)
{
    if(right-left <= 0) // jeśli rozmiar <= 1,
        return; // już posortowane
    else // rozmiar wynosi 2 lub więcej
    {
        // dzielimy zakres
        int partition = partitionIt(left, right);
        recQuickSort(left, partition-1); // sortujemy lewą stronę
        recQuickSort(partition+1, right); // sortujemy prawą stronę
    }
} // end recQuickSort()
```

Jak widać, działanie algorytmu składa się z trzech podstawowych etapów:

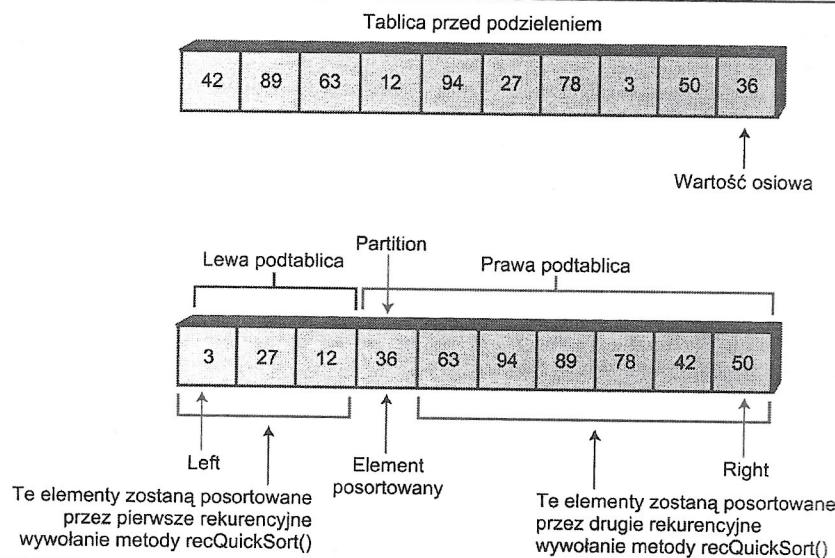
- (1) Podziału tablicy lub podtablicy na dwie grupy — lewą oraz prawą. Pierwsza z nich zawiera klucze mniejsze, a druga — większe.
- (2) Rekurencyjnego wywołania samej siebie w celu posortowania lewej grupy.
- (3) Rekurencyjnego wywołania samej siebie w celu posortowania prawej grupy.

Po wykonaniu operacji podziału wszystkie elementy znajdujące się w prawej podgrupie będą mniejsze od wszystkich elementów należących do grupy prawej. Jeśli zatem w kolejnych etapach działania algorytmu zostaną posortowane obie podtablice, to będzie to równoznaczne z posortowaniem całej tablicy. A w jaki sposób można posortować podtablice? Oczywiście wywołując rekurencyjnie tę samą metodę.

Argumenty przekazywane w wywołaniu metody `recQuickSort()` określają lewy i prawy kraniec tablicy (podtablicy), którą należy posortować. W pierwszej kolejności metoda sprawdza, czy sortowany zakres tablicy zawiera tylko jeden element. Jeśli tak, to tablica jest z definicji posortowana,

a działanie metody natychmiast się kończy. A zatem próba posortowania tablicy jednoelementowej stanowi przypadek bazowy.

Jeśli sortowana tablica składa się z dwóch lub większej liczby elementów, to algorytm dzieli ją wykorzystując w tym celu metodę `partitionIt()` opisaną we wcześniejszej części rozdziału. Metoda `partitionIt()` zwraca indeks komórki stanowiącej granicę podziału — skrajnej lewej komórki należącej do prawej podtablicy (zawierającej większe klucze). A zatem podział wyznacza granicę pomiędzy dwiema podtablicami. Sytuacja ta została zilustrowana na rysunku 7.8.



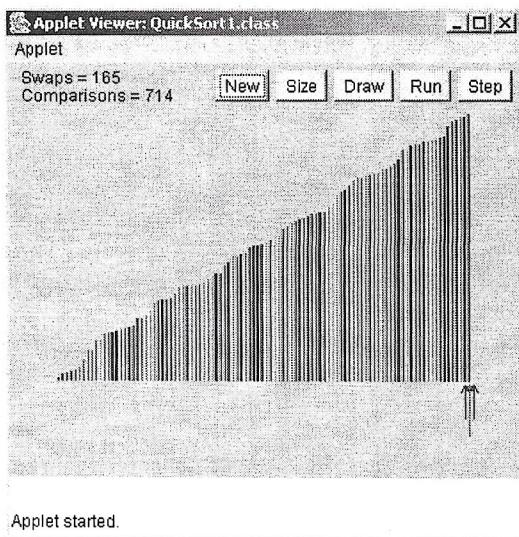
RYSUNEK 7.8. Rekurencyjne wywoływanie metody umożliwiają posortowanie podtablic

Po dokonaniu podziału metoda `recQuickSort()` dwukrotnie, rekurencyjnie wywołuje samą siebie: za pierwszym razem w celu posortowania lewej części tablicy wyznaczonej przez indeksy `left` oraz `partition-1`, a za drugim — w celu posortowania prawej części tablicy wyznaczonej przez indeksy `partition+1` oraz `right`. Warto zwrócić uwagę, że element znajdujący się w komórce o indeksie `partition` nie jest uwzględniany w żadnym z rekurencyjnych wywołań metody `recQuickSort()`. Ale dlaczego? Czy ten element nie musi być posortowany? Odpowiedź na to pytanie tkwi w sposobie dobrania wartości osiowej

Wybór wartości osiowej

Jaka wartość ma być użyta przez metodę `partitionIt()` jako wartość osiowa? Oto kilka ciekawych pomysłów:

- Wartość osiowa powinna być wartością kluczową jednego z elementów danych; element `tem` jest określony jako *os podziału*.
- Element stanowiący os podziału można wybrać w sposób mniej lub bardziej losowy. Dla uproszczenia problemu można założyć, że zawsze wybierany będzie skrajny prawy element *dzielonej* podtablicy.



RYSUNEK 7.11.
Aplet demonstracyjny QuickSort1 po posortowaniu 100 słupków

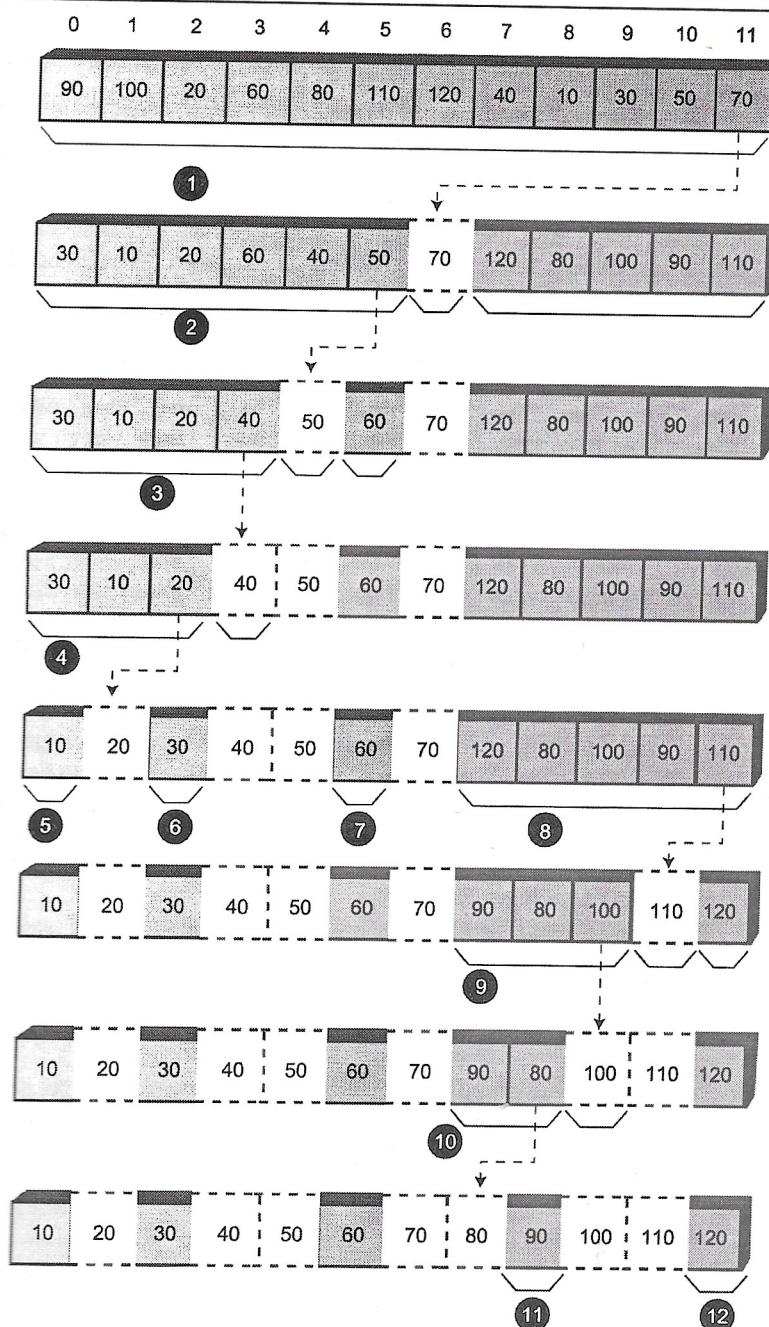
Gdyby wyświetlić wszystkie te linie jednocześnie, to poniżej i powyżej każdej z nich (z wyjątkiem tych najkrótszych) powinny być wyświetcone krótsze linie (zapewne oddzielone kolejnymi liniemi), a suma ich długości powinna odpowiadać długości oryginalnej linii (pomniejszonej o jeden słopek). Linie te odpowiadają dwóm częścioom, na które została podzielona każda podtablica.

Szczegółowa analiza

Aby bardziej szczegółowo przyjrzeć się sposobowi działania algorytmu quicksort, warto zażądać sortowania 12 słupków i wykonać cały proces krok po kroku. Dopiero w tym przypadku można się przekonać, że wartość osiowa odpowiada wysokości elementu osiowego znajdującego się na prawym krańcu i zobaczyć, w jaki sposób algorytm dzieli tablicę na dwie części, umieszcza pomiędzy nimi element stanowiący oś podziału, a następnie sortuje obie grupy powstałe w wyniku podziału (wykorzystując w tym celu wiele rekurencyjnych wywołań tej samej metody).

Rysunek 7.12 przedstawia wszystkie etapy sortowania 12 słupków. Poziome klamry widoczne poniżej tablicami pokazują, która podtablica jest dzielona w danym etapie, a liczby podane w kółkach określają kolejność wykonywanych operacji podziału. Element stanowiący oś podziału wstawiany w odpowiednie miejsce jest oznaczany poprzez strzałkę rysowaną linią przerywaną. Ostateczne położenie tego elementu jest przedstawiane jako komórka tablicy, której krawędzie są narysowane linią przerywaną, co dodatkowo pokazuje, które komórki tablicy zostały już posortowane i nie będą modyfikowane w dalszej części działania algorytmu. Poziome klamry wyświetlane poniżej pojedynczych komórek (5, 6, 7, 11 oraz 12) reprezentują przypadki bazowe metody recQuickSort(), które sprawiają, że jej działanie zostaje natychmiast zakończone.

W niektórych przypadkach (na rysunku 7.12 są one oznaczone liczbami 4 oraz 10) elementy stanowiące oś podziału zostają ponownie umieszczone w swym początkowym położeniu — na prawym krańcu sortowanej tablicy. W takim przypadku lewa podtablica nie istnieje, a algorytm musi posortować tylko jedną podtablicę — tę która znajduje się z lewej strony elementu stanowiącego oś podziału.



RYSUNEK 7.12.
Proces sortowania
metodą quicksort

Różne etapy przedstawione na rysunku 7.12 są realizowane na różnych poziomach rekurencyjnych wywołań metody `recQuickSort()`. Etapy działania algorytmu wykonywane na poszczególnych poziomach zostały przedstawione w tabeli 7.3. Pierwszym poziomem jest początkowe wywołanie metody `recQuickSort()` wykonywane bezpośrednio w metodzie `main()`. Drugi poziom stanowią

TABELA 7.3.

Poziomy rekurencji przedstawione na rysunku 7.12

Etap	Poziom rekurencji
1	1
2, 8	2
3, 7, 9, 12	3
4, 10	4
5, 6, 11	5

dwa rekurencyjne wywołania metody `recQuickSort()` umieszczone wewnętrz niej samej. Każda z metod realizowanych na drugim poziomie ponownie wywołuje dwie metody `recQuickSort()`, stające trzeci poziom rekurencji.

Kolejność, w jakiej wyznaczane są kolejne podzielone grupy danych, odpowiadająca kolejnym etapom działania algorytmu, nie jest w żaden sposób związana z poziomami rekurencyjnych wywołań metody `recQuickSort()`. A zatem wszystkie grupy danych wyznaczone na pierwszym poziomie rekurencji nie są obsługiwane w pierwszej kolejności, a grupy wyznaczone na poziomie drugim, nie są obsługiwane w drugiej kolejności. Kolejność obsługiwanych grup określana jest na innej zasadzie — na każdym poziomie rekurencji lewa grupa jest obsługiwana przed prawą grupą.

Teoretycznie rzecz biorąc, na czwartym poziomie rekurencji powinno być wykonywanych 8 etapów, a na piątym — 16. Jednak w praktyce, operując na tablicy tak małej jak ta użyta w naszym przykładzie, ilość elementów zostanie wyczerpana, zanim pojawi się konieczność wykonania tak wielu etapów.

Ilość poziomów rekurencyjnych wywołań, koniecznych do posortowania 12 elementów pokazuje, iż stos maszynowy musi być na tyle duży, aby można na nim zmieścić 5 kompletów argumentów wywołania metody sortującej oraz wartości wynikowych — po jednym na każdy poziom rekurencji. Jak się wkrótce przekonamy, wartość ta jest nieco większa od logarytmu o podstawie 2 z liczby N — $\log_2 N$. Wielkość stosu maszynowego zależy od używanego systemu operacyjnego. Sortowanie bardzo dużych tablic przy użyciu algorytmów rekurencyjnych może zatem prowadzić do przepełnienia stosu i wystąpienia błędów w obsłudze pamięci.

Sprawy, o jakich należy pamiętać

Oto kilka rzeczy, które można zauważać podczas obserwowania działania apletu demonstracyjnego *QuickSort1*.

Można sądzić, że tak potężny algorytm jak quicksort nie będzie w stanie obsługiwać bardzo małych podtablic — dwu- lub trójelementowych. Niemniej jednak przedstawiona wersja algorytmu radzi sobie z takimi tablicami całkiem dobrze; po prostu, aby położenie wskaźników `leftScan` oraz `rightScan` pokryło się, nie trzeba będzie ich przesuwać na znaczne odległości. Dlatego też nie trzeba korzystać z innej metody sortowania w celu obsługi niewielkich tablic (choć informacje przedstawione w dalszej części rozdziału pokażą, że wykorzystanie takiego innego sposobu obsługi niewielkich tablic może dawać korzyści).

Po zakończeniu każdej analizy `leftScan` wskazuje na miejsce podziału, czyli na lewy element prawej podtablicy. Algorytm zamienia następnie element stanowiący oś podziału z elementem zapisanym w miejscu podziału, przez co pierwszy z nich zostaje umieszczony we właściwym miejscu. Zgodnie z podanymi informacjami, w etapach 3. i 9. przedstawionych na rysunku 7.12 wskaźnik `leftScan` pokazuje na element stanowiący oś podziału, przez co dokonywanie zamiany nie ma większego sensu. Można sądzić, że dokonywanie zamiany w takim przypadku to strata czasu i zdecydować się na zakończenie przesuwania wskaźnika `leftScan` o jeden słupek wcześniej. Niemniej