```
} // koniec klasy MergeApp
////////////////////////////////////////////////////////////////
```

W metodzie `main()` tworzone są trzy tablice — `arrayA`, `arrayB` oraz `arrayC`, następnie wywoływana jest metoda `merge()` scalająca tablice `arrayA` i `arrayB` i zapisująca ich zawartość w tablicy `arrayC`, a w końcu zawartość tablicy `arrayC` jest wyświetlana. Oto wyniki wykonania programu:

```
7 14 23 39 47 55 62 74 81 95
```

Wewnątrz metody `merge()` umieszczone są trzy pętle `while`. Pierwsza z nich analizuje zawartość tablicy `arrayA` oraz `arrayB`, porównując ich elementy i zapisując mniejszy z nich do tablicy `arrayC`.

Druga pętla obsługuje sytuację, w której cała zawartość tablicy `arrayB` została zapisana w `arrayC`, natomiast w tablicy `arrayA` wciąż pozostały elementy do przeniesienia (właśnie tak dzieje się w naszym przykładzie, w którym w tabeli `arrayA` pozostają liczby 81 i 95). Pętla ta kopiuje pozostałe elementy tablicy `arrayA` i zapisuje je w tablicy `arrayC`.

Trzecia pętla while obsługuje podobną sytuację, gdy wszystkie elementy z tablicy `arrayA` zostały zapisane w `arrayC`, lecz w tablicy `arrayB` wciąż pozostają elementy, które należy przenieść. Pętla kopiuje te elementy do tablicy `arrayC`.

Sortowanie przez scalanie

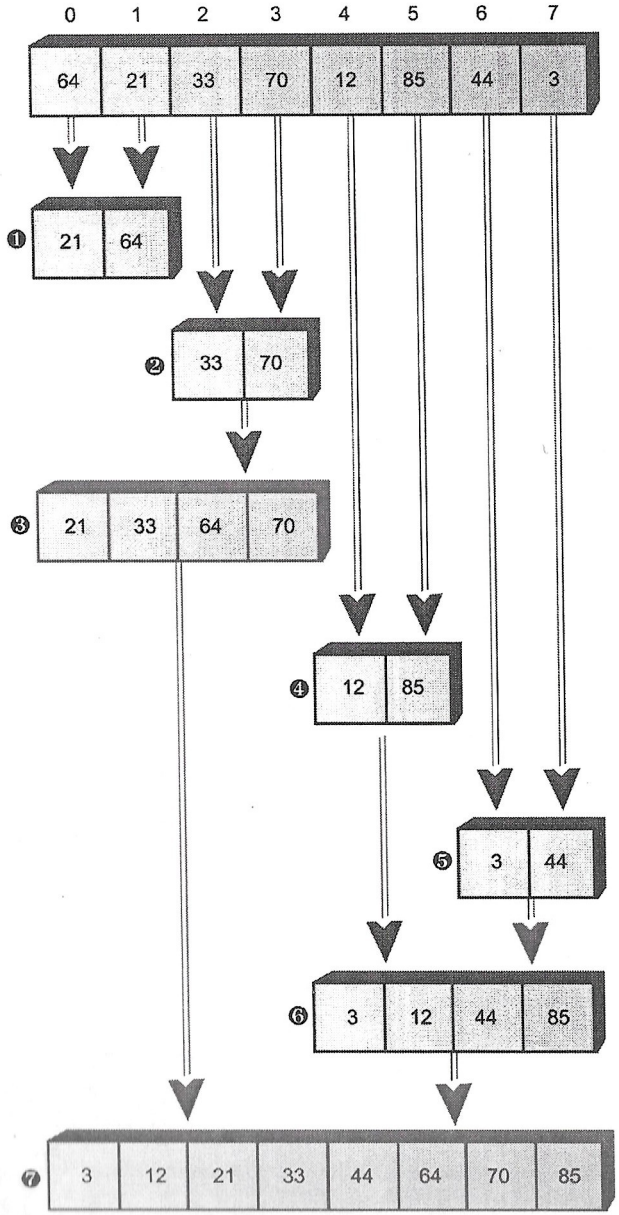
Idea działania algorytmu sortowania przez scalanie polega na podzieleniu tablicy na dwie części, posortowaniu każdej z nich, a następnie scaleniu obu posortowanych połówek z wykorzystaniem metody `merge()` z powrotem w jedną tablicę. A w jaki sposób są sortowane te połówki oryginalnej tablicy? Niniejszy rozdział przedstawia zagadnienia rekurencji, a zatem Czytelnik zapewne już zna odpowiedź — te połówki dzieli się na połowy, które następnie są sortowane i ponownie scalane.

W podobny sposób wszystkie pary ósmych części tablicy są scalane, tworząc posortowane ćwiartki, wszystkie pary szesnastych części tablicy są scalane, tworząc posortowane ósme części tablicy, i tak dalej. Tablica jest dzielona, aż do momentu uzyskania poddrzewa zawierającego tylko jeden element. To jest przypadek bazowy — zakładamy bowiem, że tablica zawierająca jeden element jest posortowana.

Jak można się było przekonać, każde rekurencyjne wywołanie metody powoduje zmniejszenie danych, na jakich operujemy, jednak przed zakończeniem wywołania dane te są ponownie łączone. W przypadku algorytmu sortowania przez scalanie każde rekurencyjne wywołanie metody powoduje podzielenie zakresu danych, na jakich operujemy na połowę i ponowne scalenie mniejszych zakresów przez zakończeniem wywołania.

Gdy metoda `mergeSort()` kończy działanie po dotarciu do dwóch jednoelementowych tablic, elementy te są scalane do postaci posortowanej tablicy dwuelementowej. Wszystkie pary tablicy dwuelementowych są następnie scalane do postaci tablicy czteroelementowych. Proces ten jest kontynuowany, aż do momentu gdy cała tablica zostanie posortowana. Sposób działania algorytmu najłatwiej można przedstawić, w przypadku gdy ilość elementów w początkowej tablicy jest potęgą liczby 2, jak pokazano na rysunku 6.15.

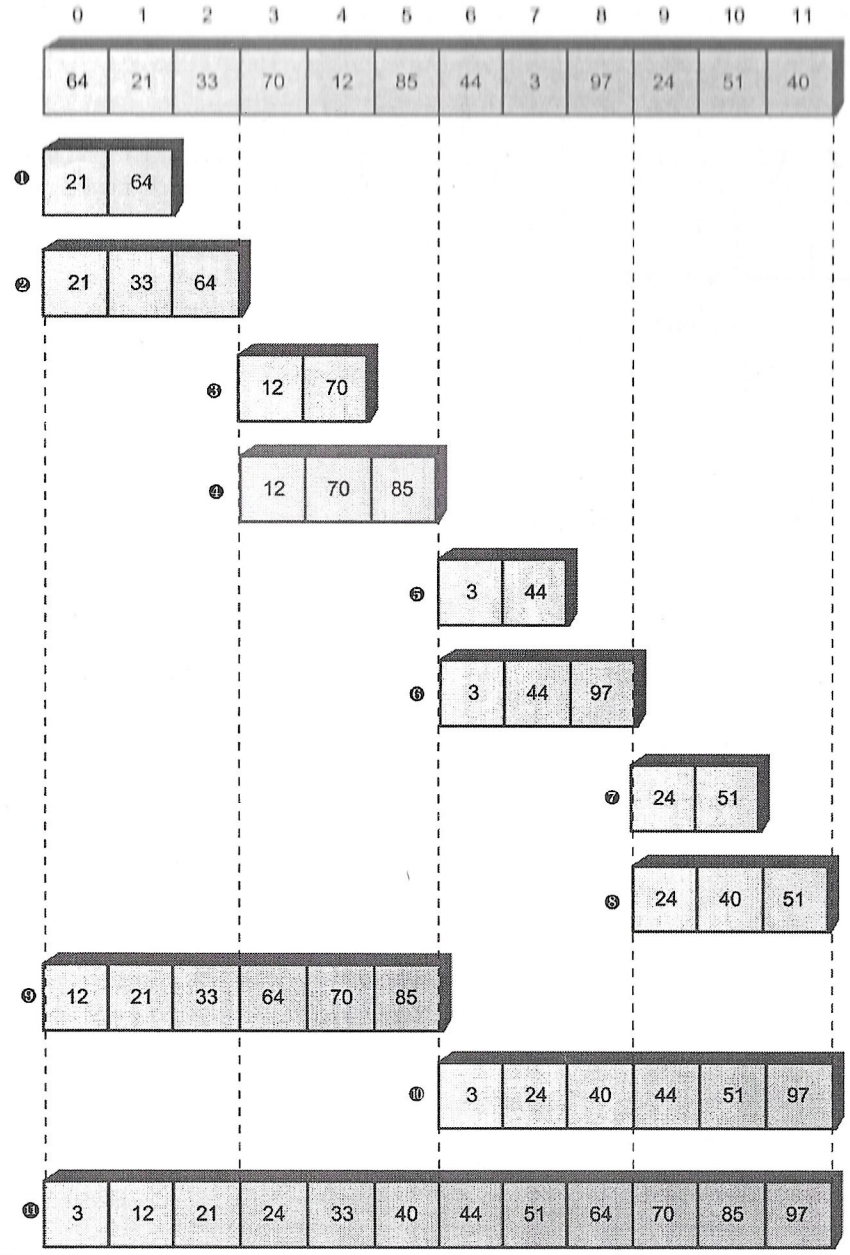
W pierwszej kolejności na samym dole tablicy zakresy 0-0 oraz 0-1 są scalane do zakresu 0-1. Oczywiście w rzeczywistości zakresy 0-0 oraz 1-1 nie są prawdziwymi zakresami — są to pojedyncze elementy, spełniające warunek bazowy. Podobnie zakresy 2-2 oraz 3-3 są scalane do zakresu 2-3. Następnie zakresy 0-1 oraz 2-3 są scalane do zakresu 0-3.



RYSUNEK 6.15. Scalanie coraz większych tablic

W górnej połówce tablicy zakresy 4-4 i 5-5 są scalane do zakresu 4-5, zakresy 6-6 i 7- 7 do zakresu 6-7, a zakresy 4-5 i 6-7 do zakresu 4-7. W końcu dolna połówka tablicy (zakres 0-3) oraz jej górna połówka (zakres 4-7) są łączone w pełną, posortowaną tablicę (zakres 0-7).

Jeśli natomiast ilość elementów tablicy nie jest potęgą liczby 2, zachodzi konieczność scalenia tablic o różnych wielkościach. Na przykład rysunek 6.16 przedstawia sytuację, w której sortowana tablica zawiera 12 elementów. W tym przypadku należy scalić tablice zawierającą 2 elementy z ta-



RYSUNEK 6.16. Scalanie tablic, których wielkość nie jest potęgą liczby 2

W pierwszej kolejności jednoelementowy zakres 0-0 jest scalany z jednoelementowym zakresem 1-1, tworząc w ten sposób dwuelementowy zakres 0-1. Następnie zakres 0-1 jest scalany z jednoelementowym zakresem 2-2. W ten sposób tworzony jest trójelementowy zakres 0-2. Ten zakres jest z kolei scalany z trójelementowym zakresem 3-5. Proces ten trwa aż do posortowania całej tablicy.

```
      int maxSize = 100;            // wielkość tablicy
      DArray arr;                   // odwołanie do tablicy
      arr = new DArray(maxSize);    // tworzymy tablicę

      arr.insert(64);               // wstawiamy elementy
      arr.insert(21);
      arr.insert(33);
      arr.insert(70);
      arr.insert(12);
      arr.insert(85);
      arr.insert(44);
      arr.insert(3);
      arr.insert(99);
      arr.insert(0);
      arr.insert(108);
      arr.insert(36);

      arr.display();                // wyświetlamy elementy

      arr.mergeSort();              // sortujemy tablicę

      arr.display();                // ponownie wyświetlamy elementy
      } // koniec metody main()
   } // koniec class MergeSortApp
////////////////////////////////////////////////////////////////
```

Wyniki wykonania tego programu przedstawiają jedynie nieposortowaną oraz posortowaną zawartość tablicy:

```
64 21 33 70 12 85 44 3 99 0 108 36
0 3 12 21 33 36 44 64 70 85 99 108
```

Umieszczając w kodzie metody `recMergeSort()` dodatkowe instrukcje, można by generować komunikaty informujące o czynnościach, jakie program wykonuje podczas sortowania. Poniższy przykład pokazuje, jak mogłyby wyglądać takie informacje, generowane podczas sortowania tablicy zawierającej 4 liczby {64, 21, 33, 70} (tablicę tę można sobie wyobrazić jako dolną połówkę tablicy z rysunku 6.15).

```
Wywołanie 0-3
   Sortowana będzie dolna połówka 0-3
   Wywołanie 0-1
      Sortowana będzie dolna połówka 0-1
         Wywołanie 0-0
         Przypadek bazowy - zwracane 0-0
      Sortowana będzie górna połówka 0-1
         Wywołanie 1-1
         Przypadek bazowy - zwracane 1-1
      Scalanie połówek w zakres 0-1
   Zwracane 0-1                    theArray=21 64 33 70
```

```
   Wywołanie 2-3
      Sortowana będzie dolna połówka 2-3
         Wywołanie 2-2
         Przypadek bazowy - zwracane 2-2
      Sortowana będzie górna połówka 2-3
         Wywołanie 3-3
         Przypadek bazowy - zwracane 3-3
      Scalanie połówek w zakres 2-3
   Zwracane 2-3                    theArray=21 64 33 70
   Scalanie połówek w zakres 0-3
Zwracane 0-3                    theArray=21 33 64 70
```

Mniej więcej te same informacje zwróciłby aplet demostracyjny *MergeSort*, gdyby był w stanie sortować jedynie cztery elementy. Analiza powyższych wyników oraz porównanie ich z kodem metody `recMergeSort()` i rysunkiem 6.15 umożliwi dokładne zrozumienie zasady działania omawianego algorytmu sortowania.

Efektywność działania algorytmu sortowania przez scalanie

Zgodnie z podanymi wcześniej informacjami, efektywność działania algorytmu sortowania przez scalanie wynosi $O(N * \log N)$. Ale skąd to wiadomo? Przekonajmy się, w jaki sposób można określić, ile razy podczas działania algorytmu dane będą musiały zostać skopiowane oraz ile razy trzeba je będzie porównywać. Zakładamy, że właśnie kopiowanie danych oraz ich porównywanie są operacjami zajmującymi najwięcej czasu oraz że rekurencyjne wywołania metod i zwracanie wartości przez metodę nie powoduje znaczących narzutów czasowych.

Ilość kopiowań

Przeanalizujmy rysunek 6.15. Każda komórka umieszczona poniżej górnego wiersza liczb reprezentuje daną skopiowaną z tablicy do obszaru roboczego.

Dodając do siebie ilość wszystkich komórek widocznych na rysunku 6.15 (w siedmiu ponumerowanych etapach), można się przekonać, że posortowanie 8 elementów wymaga wykonania 24 operacji kopiowania. $\text{Log}_2 8$ ma wartość 3, a zatem $8 * \log_2 8$ daje 24. Wynika stąd, że dla tablicy zawierającej 8 elementów, ilość operacji kopiowania jest proporcjonalna do wartości $N * \log_2 N$.

Na analizowany proces sortowania można jednak spojrzeć w inny sposób. Otóż posortowanie 8 liczb wymaga 3 *poziomów*, a każdy z nich składa się z 8 operacji kopiowania. Poziom oznacza w tym przypadku wszystkie operacje kopiowania do podtablicy o takiej samej wielkości. Na pierwszym poziomie wykorzystywane są cztery podtablice 2-elementowe, na drugim poziomie — dwie podtablice 4-elementowe, a na trzecim poziomie — jednak podtablica 8-elementowa. Każdy poziom zawiera 8 elementów. A zatem także przy takim podejściu wykonywanych jest $3 * 8$, czyli 24 operacje kopiowania.

Analizując fragmenty rysunku 6.15, można się przekonać, że do posortowania 4 elementów koniecznych jest 8 operacji kopiowania (etapy 1, 2 oraz 3), a do posortowania 2 elementów — dwie operacje kopiowania. Identyczne obliczenia pozwalają określić ilość operacji kopiowania, jakie należy wykonać w celu posortowania większych tablic. Podsumowanie tych informacji zostało przedstawione w tabeli 6.4.