CHAPTER

18

Algorithm Efficiency and Sorting

Objectives

• To estimate algorithm efficiency using the Big O notation (§18.2).

• To understand growth rates and why constants and smaller terms can be ignored in the estimation (§18.2).

• To know the examples of algorithms with constant time, logarithmic time, linear time, log-linear time, quadratic time, and exponential time (§18.2).

• To analyze linear search, binary search, selection sort, and insertion sort (§18.2).

• To design, implement, and analyze bubble sort (§18.3).

• To design, implement, and analyze merge sort (§18.4).

• To design, implement, and analyze quick sort (§18.5).

• To design, implement, and analyze heap sort (§18.6).

• To sort large data in a file (§18.7).
18.1 Introduction

<side remark: why study sorting?>
Sorting is a classic subject in computer science. There are three reasons for studying sorting algorithms. First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems. Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, functions, and arrays. Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

<side remark: what data to sort?>
The data to be sorted might be integers, doubles, characters, or any generic element type. §6.8, “Sorting Arrays,” presented selection sort and insertion sort for numeric values. The selection sort algorithm was extended to sort an array of the generic element type in §15.3, “Example: A Generic Sort.” For simplicity, this section assumes:

1. data to be sorted are integers,
2. data are sorted in ascending order, and
3. data are stored in an array.

The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in a vector or a linked list.

There are many algorithms on sorting. In order to analyze and compare the complexities of these algorithms, this chapter first introduces the Big O notation for estimating algorithm efficiency. The whole chapter is optional. No chapter in the book is dependent on this chapter.

18.2 Estimating Algorithm Efficiency
<side remark: execution time>
Suppose two algorithms perform the same task such as search (linear search vs. binary search) or sort (selection sort vs. insertion sort). Which one is better? One possible approach to answer this question is to implement these algorithms and run the programs to get execution time. But there are two problems for this approach:

1. First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
2. Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be
the first in the list, linear search will find the element quicker than binary search.

<side remark: growth rate>
It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm’s execution time increases as the input size increases, so you can compare two algorithms by examining their growth rates.

18.2.1 Big O Notation
<side remark: Big O notation>
Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires \( n \) comparisons for an array of size \( n \). If the key is in the array, it requires \( n/2 \) comparisons on average. The algorithm’s execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of \( n \). Computer scientists use the Big O notation to abbreviate for “order of magnitude.” Using this notation, the complexity of the linear search algorithm is \( O(n) \), pronounced as “order of \( n \).”

<side remark: best-case>
<side remark: worst-case>
<side remark: average-case>
For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

<side remark: ignoring multiplicative constants>
The linear search algorithm requires \( n \) comparisons in the worst-case and \( n/2 \) comparisons in the average-case. Using the Big \( O \) notation, both cases require \( O(n) \) time. The multiplicative constant (1/2) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for \( n/2 \) or \( 100n \) is the same as \( n \), i.e. \( O(n) = O(n/2) = O(100n) \).

<side remark: ignoring non-dominating terms>
<side remark: large input size>
Consider the algorithm for finding the maximum number in an array of \( n \) elements. If \( n \) is 2, it takes one comparison to find the maximum number. If \( n \) is 3, it takes two comparisons to find the maximum number. In general, it takes \( n-1 \) times of comparisons to find maximum number in a list of \( n \) elements. Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm’s efficiency. As \( n \) grows larger, the \( n \) part in the expression \( n-1 \) dominates the complexity. The Big \( O \) notation allows you to ignore the non-dominating part (e.g., -1 in the expression \( n-1 \)) and highlight the important part (e.g., \( n \) in the expression \( n-1 \)). So, the complexity of this algorithm is \( O(n) \).

<side remark: constant time>
The Big \( O \) notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation \( O(1) \). For example, a function that retrieves an element at a given index in an array takes constant time, because the time does not grow as the size of the array increases.

18.2.2 Analyzing Binary Search
The binary search algorithm presented in Listing 6.8, BinarySearch.cpp, searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by \( c \). Let \( T(n) \) denote the time complexity for a binary search on a list of \( n \) elements. Without loss of generality, assume \( n \) is a power of 2 and \( k = \log_2 n \). Since binary search eliminates half of the input after two comparisons,

\[
T(n) = T\left( \frac{n}{2} \right) + c = T\left( \frac{n}{4} \right) + c + c = \ldots = T\left( \frac{n}{2^k} \right) + ck = T(1) + c \log_2 n = 1 + c \log_2 n
\]

<side remark: logarithmic time>
Ignoring constants and smaller terms, the complexity of the binary search algorithm is \( O(\log n) \). An algorithm with the
an $O(\log n)$ time complexity is called a logarithmic algorithm. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.

18.2.3 Analyzing Selection Sort
The selection sort algorithm presented in Listing 6.9, SelectionSort.cpp, finds the largest number in the list and places it last. It then finds the largest number remaining and places it next to last, and so on until the list contains only a single number. The number of comparisons is $n-1$ for the first iteration, $n-2$ for the second iteration, and so on. Let $T(n)$ denote the complexity for selection sort and $c$ denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c + \ldots + 2 + c + 1 + c = n^2 - \frac{n}{2} + cn$$

Ignoring constants and smaller terms, the complexity of the selection sort algorithm is $O(n^2)$.

<side remark: quadratic time>
An algorithm with the $O(n^2)$ time complexity is called a quadratic algorithm. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with two nested loops are often quadratic.

18.2.4 Analyzing Insertion Sort
The insertion sort algorithm presented in Listing 6.10, InsertionSort.cpp, sorts a list of values by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the $k$th iteration, to insert an element to a array of size $k$, it may take $k$ comparisons to find the insertion position, and $k$ moves to insert the element. Let $T(n)$ denote the complexity for insertion sort and $c$ denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = 2 + c + 2 \times 2 + c + \ldots + 2 \times (n-1) + c = n^2 - n + cn$$

Ignoring constants and smaller terms, the complexity of the insertion sort algorithm is $O(n^2)$.

18.2.5 Analyzing Towers of Hanoi
The Towers of Hanoi problem presented in Listing 8.7, TowersOfHanoi.cpp, moves $n$ disks from tower A to tower B with the assistance of tower C recursively as follows:

1. Move the first $n - 1$ disks from A to C with the assistance of tower B.
2. Move disk $n$ from A to B.
3. Move $n - 1$ disks from C to B with the assistance of tower A.

Let $T(n)$ denote the complexity for the algorithm that moves $n$ disks and $c$ denote the constant time to move one disk, i.e., $T(1)$ is $c$. So,

$$T(n) = T(n-1) + c + T(n-1) = 2T(n-1) + c$$
$$= 2(2T(n-2) + c) + c = 2^nT(1) + c2^{n-1} + ... + c2 + c =$$
$$= c2^n + c2^{n-1} + ... + c2 + c = c(2^{n+1} - 1) = O(2^n)$$

<side remark: exponential time>

<side remark: $O(c^2) \Rightarrow O(c^n)$>

An algorithm with the $O(c^n)$ time complexity is called an exponential algorithm. As the input size increases, the time for the exponential algorithm grows exponentially. The exponential algorithms are not practical for large input size.

18.2.6 Comparing Common Growth Functions

The preceding sections analyzed the complexity of several algorithms. Table 18.1 lists some common growth functions. These functions are ordered as follows:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Table 18.1

Common Growth Functions

<table>
<thead>
<tr>
<th>Big-O Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant time</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic time</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear time</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>log-linear time</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic time</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic time</td>
</tr>
</tbody>
</table>
Table 18.2 shows how growth rates change as the input size doubles from $n = 25$ to $n = 50$. 

Table 18.2

<table>
<thead>
<tr>
<th>Function</th>
<th>$n = 25$</th>
<th>$n = 50$</th>
<th>$f(50)/f(25)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>4.64</td>
<td>5.64</td>
<td>1.21</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>25</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>116</td>
<td>282</td>
<td>2.431</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>625</td>
<td>2500</td>
<td>4</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>15625</td>
<td>125000</td>
<td>8</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>$3.36 \times 10^7$</td>
<td>$1.27 \times 10^{15}$</td>
<td>$3.35 \times 10^7$</td>
</tr>
</tbody>
</table>

**18.3 Bubble Sort**

The bubble sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually "bubble" their way to the top and the larger values sink to the bottom. After first pass, the last element becomes the largest in the array. After the second pass, the second last element becomes the second largest in the array. Continue the process until all elements are sorted.

*Side Remark: bubble sort illustration*

Figure 18.1(a) shows the first pass of a bubble sort of an array of six elements (2 9 5 4 8 1). Compare the elements in the first pair (2 and 9), and no swap is needed because they are already in order. Compare the elements in the second pair (9 and 5), and swap 9 with 5 because 9 is greater than 5. Compare the elements in the third pair (9 and 4), and swap 9 with 4. Compare the elements in the fourth pair (9 and 8), and swap 9 with 8. Compare the elements in the fifth
pair (9 and 1), swap 9 with 1. The pairs being compared are highlighted and the numbers that are already sorted are italicized.

<table>
<thead>
<tr>
<th>2 5 4 8 1</th>
<th>2 5 4 8 1 9</th>
<th>2 4 5 1 8 9</th>
<th>2 4 5 1 8 9</th>
<th>2 4 5 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 4 8 1 9</td>
<td>2 4 5 1 8 9</td>
<td>2 4 1 5 8 9</td>
<td>1 2 4 5 8 9</td>
<td></td>
</tr>
<tr>
<td>2 5 4 8 1 9</td>
<td>2 4 5 1 8 9</td>
<td>2 4 1 5 8 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 5 4 8 1 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) 1st pass (b) 2nd pass (c) 3rd pass (d) 4th pass (e) 5th pass

**Figure 18.1**
Each pass compares and orders the pairs of elements sequentially.

The first pass places the largest number (9) as the last in the array. In the second pass, as shown in Figure 18.1(b), you compare and order pairs of elements sequentially. There is no need to consider the last pair, because the last element in the array is already the largest. In the third pass, as shown in Figure 18.1(c), you compare and order pairs of elements sequentially except the last two elements, because they are already ordered. So in the kth pass, you don’t need to consider the last k-1 elements, because they are already ordered.

**<Side Remark: algorithm>**
The algorithm for bubble sort can be described as follows:

**Listing 18.1 Bubble Sort Algorithm**

```java
for (int k = 1; k < arraySize; k++)
    // Perform the kth pass
    for (int i = 0; i < arraySize - k; i++)
        if (list[i] > list[i + 1])
            swap list[i] with list[i + 1];
```

Note that if no swap takes place in a pass, there is no need to perform the next pass, because all the elements are already sorted. You may improve the preceding algorithm by utilizing this property as in Listing 18.2.

**Listing 18.2 Improved Bubble Sort Algorithm**

```java
bool needNextPass = true;
for (int k = 1; k < arraySize && needNextPass; k++)
    // Array may be sorted and next pass not needed
    needNextPass = false;
    // Perform the kth pass
    for (int i = 0; i < arraySize - k; i++)
```
```cpp
if (list[i] > list[i + 1])
    swap list[i] with list[i + 1];
    needNextPass = true; // Next pass still needed
}
}

The algorithm can be implemented as follows:

### Listing 18.3 BubbleSort.h (Bubble Sort Function Header)

```cpp
/* The function for sorting the numbers */
void bubbleSort(int list[], int arraySize);

bool needNextPass = true;

for (int k = 1; k < arraySize; k++)
    // Array may be sorted and next pass not needed
    needNextPass = false;
for (int i = 0; i < arraySize - k; i++)
    if (list[i] > list[i + 1])
        // Swap list[i] with list[i + 1]
        int temp = list[i];
        list[i] = list[i + 1];
        list[i + 1] = temp;
        needNextPass = true; // Next pass still needed
}
```

### 18.3.1 Bubble Sort Time

In the best-case, the bubble sort algorithm needs just the first pass to find out that the array is already sorted. No next pass is needed. Since the number of comparisons is \(n-1\) in the first pass, the best-case time for bubble sort is \(O(n)\).

In the worst case, the bubble sort algorithm requires \(n-1\) passes. The first pass takes \(n-1\) comparisons; the second pass takes \(n-2\) comparisons; and so on; the last pass takes 1 comparison. So, the total number of comparisons is:

\[
(n-1)+(n-2)+...+2+1 = \frac{n^2}{2} - \frac{n}{2}
\]

Therefore, the worst-case time for bubble sort is \(O(n^2)\).

### 18.4 Merge Sort

The merge sort algorithm can be described recursively as follows: The algorithm divides the array into two halves and applies merge sort on each half recursively. After the two
halves are sorted, merge them. The algorithm is described in Listing 18.4.

Listing 18.4 Merge Sort Algorithm

```c
void mergeSort(int list[], int arraySize)
{
    if (arraySize > 1)
    {
        mergeSort list[0 ... arraySize / 2];
        mergeSort list[arraySize / 2 + 1 ... arraySize];
        merge list[0 ... arraySize / 2] with
        list[arraySize / 2 + 1 ... arraySize];
    }
}
```

Figure 18.2 illustrates a merge sort of an array of eight elements (2 9 5 4 8 1 6 7). The original array is split into (2 9 5 4) and (8 1 6 7). Apply merge sort on this two subarrays recursively to split (1 9 5 4) into (1 9) and (5 4) and (8 1 6 7) into (8 1) and (6 7). This process continues until the subarray contains only one element. For example, array (2 9) is split into subarrays (2) and (9). Since array (2) contains a single element, it cannot be further split. Now merge (2) with (9) into a new sorted array (2 9), merge (5) with (4) into a new sorted array (4 5). Merge (2 9) with (4 5) into a new sorted array (2 4 5 9), and finally merge (2 4 5 9) with (1 6 7 8) into a new sorted array (1 2 4 5 6 7 8 9).
Merge sort employs a divide-and-conquer approach to sort the array.

The recursive call continues dividing the array into subarrays until each subarray contains only one element. The algorithm then merges these small subarrays into larger sorted subarrays until one sorted array results. The function for merging two sorted arrays is given in Listing 18.5.

Listing 18.5 Function for Merging Two Arrays

```c
void merge(int list1[], int list1Size,
           int list2[], int list2Size, int temp[])
{
    int current1 = 0; // Index in list1
    int current2 = 0; // Index in list2
    int current3 = 0; // Index in temp
    while (current1 < list1Size && current2 < list2Size)
    {
        if (list1[current1] < list2[current2])
            temp[current3++] = list1[current1++];
        else
            temp[current3++] = list2[current2++];
    }
    while (current1 < list1Size)
        temp[current3++] = list1[current1++];
    while (current2 < list2Size)
        temp[current3++] = list2[current2++];
}
```

This function merges arrays list1 and list2 into a new array temp. So, the size of temp should be list1Size + list2Size. current1 and current2 point to the current element to be considered in list1 and list2 (lines 4-5). The function repeatedly compares the current elements from list1 and list2 and moves the smaller one to temp. current1 is increased by 1 (line 11) if the smaller one is in list1 and current2 is increased by 1 (line 13) if the smaller one is in list2. Finally, all the elements in one of the lists are moved to temp. If there are still unmoved elements in list1, copy them to temp (lines 16-17). If there are still unmoved elements in list2, copy them to temp (lines 19-20).

Figure 18.3 illustrates how to merge two arrays list1 (2 4 5 9) and list2 (1 6 7 8). Initially the current elements to be considered in the arrays are 2 and 1. Compare them and move the smaller element 1 to temp, as shown in Figure 18.3(a). current2 and current3 are increased by 1. Continue to compare the current elements in the two arrays and move the smaller one to temp until one of the arrays is completely
moved. As shown in Figure 18.3(b), all the elements in list2 are moved to temp and current1 points to element 9 in list1. Copy 9 to temp, as shown in Figure 18.3(c).

Figure 18.3

Two sorted arrays are merged into one sorted array.

The merge sort algorithm is implemented in Listing 18.6.

Listing 18.6 MergeSort.h (Merge Sort)

```c
***PD: Please add line numbers in the following code***
<Side Remark line 9: mergeSort>
<Side Remark line 14: create firstHalf>
<Side Remark line 16: sort firstHalf>
<Side Remark line 20: create secondHalf>
<Side Remark line 22: sort secondHalf>
<Side Remark line 26: merge two halves>
<Side Remark: line 28: copy to original array>
<Side Remark line 29: delete>

// Function prototype
void arraycopy(int source[], int sourceStartIndex, int target[], int targetStartIndex, int length);
void merge(int list1[], int list1Size, int list2[], int list2Size, int temp[]);

/* The function for sorting the numbers */
void mergeSort(int list[], int arraySize)
{
    if (arraySize > 1)
    {
        // Merge sort the first half
        int *firstHalf = new int[arraySize / 2];
        arraycopy(list, 0, firstHalf, 0, arraySize / 2);
        mergeSort(firstHalf, arraySize / 2);

        // Merge sort the second half
        int secondHalfLength = arraySize - arraySize / 2;
        int *secondHalf = new int[secondHalfLength];
        arraycopy(list, arraySize / 2, secondHalf, 0, secondHalfLength);
        mergeSort(secondHalf, secondHalfLength);

        // Merge firstHalf with secondHalf
        int *temp = new int[arraySize];
        merge(firstHalf, arraySize / 2, secondHalf, secondHalfLength, temp);
        arraycopy(temp, 0, list, 0, arraySize);
    }
}
```
delete [] temp;
delete [] firstHalf;
delete [] secondHalf;

void merge(int list1[], int list1Size, int list2[], int list2Size, int temp[])
{
   // Same as in Listing 18.5, so omitted
}

void arraycopy(int source[], int sourceStartIndex, int target[], int targetStartIndex, int length)
{
   for (int i = 0; i < length; i++)
   {
      target[i + targetStartIndex] = source[i + sourceStartIndex];
   }
}

The algorithm creates a new array firstHalf, which is a copy of the first half of list (line 14). The algorithm invokes mergeSort recursively on firstHalf (line 16). The length of the firstHalf is arraySize / 2 and the length of the secondHalf is arraySize - arraySize / 2. The new array secondHalf was created to contain the second part of the original array list. The algorithm invokes mergeSort recursively on secondHalf (line 22). After firstHalf and secondHalf are sorted, they are merged to become a new sorted array in temp (line 26). Finally, temp is copied to the original array list (line 26). So, array list is now sorted.

18.4.1 Merge Sort Time

<Side Remark: time analysis>

Let \( T(n) \) denote the time required for sorting an array of \( n \) elements using merge sort. Without loss of generality, assume \( n \) is a power of 2. The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays. So,

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{mergetime}
\]

The first \( T\left(\frac{n}{2}\right) \) is the time for sorting the first half of the array and the second \( T\left(\frac{n}{2}\right) \) is the time for sorting the second half. To merge two subarrays, it takes at most \( n-1 \) comparisons to compare the elements from the two subarrays and \( n \) moves to move elements to the temporary array. So, the total time is \( 2n-1 \). Therefore,
\[ T(n) = 2T\left(\frac{n}{2}\right) + 2n - 1 = 2\left(2T\left(\frac{n}{4}\right) + 2\frac{n}{2} - 1\right) + 2n - 1 = 2^2 T\left(\frac{n}{2^2}\right) + 2n - 2 + 2n - 1 \]
\[ = 2^k T\left(\frac{n}{2^k}\right) + 2n - 2^{k-1} + \ldots + 2n - 2 + 2n - 1 \]
\[ = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + 2n - 2^{\log_2 n-1} + \ldots + 2n - 2 + 2n - 1 \]
\[ = n + 2n \log n - 2^{\log_2 n} + 1 = 2n \log n + 1 = O(n \log n) \]

**<Side Remark: \(O(n \log n)\) merge sort>**
The complexity of merge sort is \(O(n \log n)\). This algorithm is better than selection sort, insertion sort, and bubble sort.

### 18.5 Quick Sort

Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the **pivot**, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part. The algorithm is described in Listing 18.7.

#### Listing 18.7 Quick Sort Algorithm

***PD: Please add line numbers in the following code***

**<Side Remark line 3: base condition>**
**<Side Remark line 5: select the pivot>**
**<Side Remark line 6: partition the list>**
**<Side Remark line 9: sort first part>**
**<Side Remark line 10: sort second part>**

```c
void quickSort(int list[], int arraySize)
{
  if (arraySize > 1)
  {
    select a pivot;
    partition list into list1 and list2 such that
    all elements in list1 <= pivot and all elements
    in list2 > pivot;
    quickSort on list1;
    quickSort on list2;
  }
}
```

**<Side Remark: how to partition>**
Each partition places the pivot in the right place. The selection of the pivot affects the performance of the algorithm. Ideally, you should choose the pivot that divides the two parts evenly. For simplicity, assume the first element in the array is chosen as the pivot. Exercise 18.a proposes an alternative strategy for selecting the pivot.

**<Side Remark: quick sort illustration>**
Figure 18.4 illustrates how to sort an array (5 2 9 3 8 4 0 1 6 7) using quick sort. Choose the first element 5 as the pivot. The array is partitioned into two parts, as shown in Figure 18.4(b). The highlighted pivot is placed in the right place in the array. Apply quick sort on two partial arrays (4 2 1 3 0) and then (8 9 6 7). The pivot 4 partitions (4 2 1 3 0) into just one partial array (0 2 1 3), as shown in Figure 18.4(c). Apply quick sort on (0 2 1 3). The pivot 0 partitions it to just one partial array (2 1 3), as shown in Figure 18.4(d). Apply quick sort on (2 1 3). The pivot 2 partitions it to (1) and (3), as shown in Figure 18.4(e). Apply quick sort on (1). Since the array contains just one element, no further partition is needed.

The quick sort algorithm is recursively applied to partial arrays.

Now turn attention to partition. To partition an array or a partial array, search for the first element from left forward in the array that is greater than the pivot, then search for the first element from right backward in the array that is less than or equal to the pivot. Swap these two elements. Repeat the same search and swap operations until all the elements are searched. Listing 18.8 gives a function that partitions a partial array list[first..last]. The first element in the partial array is chosen as the pivot.
pivot (line 3). Initially low points to the second element in the partial array and high points to the last element in the partial array. The function returns the new index for the pivot that divides the partial array into two parts.

Listing 18.8 Partition Function

```c
/* Partition the array list[first..last] */
int partition(int list[], int first, int last)
{
    int pivot = list[first]; // Choose the first element as the pivot
    int low = first + 1; // Index for forward search
    int high = last; // Index for backward search

    while (high > low)
    {
        // Search forward from left
        while (low <= high && list[low] <= pivot)
            low++;

        // Search backward from right
        while (low <= high && list[high] > pivot)
            high--;

        // Swap two elements in the list
        if (high > low)
        {
            int temp = list[high];
            list[high] = list[low];
            list[low] = temp;
        }
    }

    while (high > first && list[high] >= pivot)
        high--;

    // Swap pivot with list[high]
    if (pivot > list[high])
    {
        list[first] = list[high];
        list[high] = pivot;
        return high;
    }
    else
        return first;
}
```

Figure 18.5 illustrates how to partition an array (5 2 9 3 8 4 0 1 6 7). Choose the first element 5 as the pivot. Initially low is the index that points to element 2 and high points to element 7, as shown in Figure 18.5(a). Advance index low forward to search for the first element (9) that is greater than the pivot and move index high backward to search for the first element (1) that is less than or equal to the pivot, as shown in Figure 18.5(b). Swap 9 with 1, as
shown in Figure 18.5(c). Continue the search and move low to point to element 8 and high to point to element 0, as shown in Figure 18.5(d). Swap element 8 with 0, as shown in Figure 18.5(e). Continue to move low until it passes high, as shown in Figure 18.5(f). Now all the elements are examined. Swap the pivot with element 4 at index high. The final partition is shown in Figure 18.5(g). The index of the pivot is returned when the function is finished.

(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

The index of the pivot is returned

**Figure 18.5**

The partition function returns the index of the pivot after it is put in the right place.

The quick sort algorithm is implemented in Listing 18.9. There are two overloaded quickSort functions in the class. The first function (line 2) is used to sort an array. The
second is a helper function (line 6) that sorts a partial array with a specified range.

**PD: Please add line numbers in the following code**

```c
Listing 18.9 QuickSort.h (Quick Sort)
***PD: Please add line numbers in the following code***

// Function prototypes
void quickSort(int list[], int arraySize);
void quickSort(int list[], int first, int last);
int partition(int list[], int first, int last);

void quickSort(int list[], int arraySize)
{
    quickSort(list, 0, arraySize - 1);
}

void quickSort(int list[], int first, int last)
{
    if (last > first)
    {
        int pivotIndex = partition(list, first, last);
        quickSort(list, first, pivotIndex - 1);
        quickSort(list, pivotIndex + 1, last);
    }
}

/* Partition the array list[first..last] */
int partition(int list[], int first, int last)
{
    // Same as in Listing 18.7, so omitted
}
```

18.5.1 Quick Sort Time

**O(n) partition time**

To partition an array of \( n \) elements, it takes \( n \) comparisons and \( n \) moves in the worst case. So, the time required for partition is \( O(n) \).

**O(n^2) worst-case time**

In the worst case, each time the pivot divides the array into one big subarray with the other empty. The size of the big subarray is one less than the one before divided. The algorithm requires \((n-1)+(n-2)+\ldots+2+1=O(n^2)\) time.

**O(n log n) best-case time**

In the best case, each time the pivot divides the array into two parts of about the same size. Let \( T(n) \) denote the time
required for sorting an array of \( n \) elements using quick sort. So,

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n.
\]

Similar to the merge sort analysis, \( T(n) = O(n\log n) \).

<Side Remark: \( O(n\log n) \) average-case time>
On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part. Statistically, the sizes of the two parts are very close. So the average time is \( O(n\log n) \). The exact average-case analysis is beyond the scope of this book.

<Side Remark: quick sort vs. merge sort>
Both merge sort and quick sort employ the divide-and-conquer approach. For merge sort, the bulk of work is to merge two sublists, which takes place after the sublists are sorted. For quick sort, the bulk of work is to partition the list into two sublists, which takes place before the sublists are sorted. Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case. Merge sort requires a temporary array for merging two subarrays. Quick sort does not need additional array space. So, quick sort is more space efficient than merge sort.

18.6 Heap Sort

Heap sort uses a binary heap to sort an array. The binary heap, introduced in §20.5, “Heaps and Priority Queues,” can be visualized as a complete binary tree. Each node in the tree is greater than or equal to its descendants, as shown in Figure 18.6(a). A binary heap can be implemented using an array, as shown in Figure 18.6(b). The root is at position 0, and its two children are at positions 1 and 2. For a node at position \( i \), its left child is at position \( 2i+1 \) and its right child is at position \( 2i+2 \), and its parent is \( (i-1)/2 \). For example, the node for element 39 is at position 4, so its left child (element 14) is at position \( 9 \) \( (2\times4+1) \), its right child (element 33) is at position \( 10 \) \( (2\times4+2) \), and its parent (element 42) is at position \( 1 \) \( ((4-1)/2) \).
A binary heap can be implemented using an array.

18.6.1 Sorting an Array from a Heap

Once you have such a heap, it is easy to obtain a sorted array as follows: Swap the root (i.e., the first element in the array) with the last leaf in the binary heap (i.e., the last element in the array). Remove the last element from the heap and rebuild heap. Repeat the swap and rebuild operations until the heap is empty. Figure 18.7(a) shows the tree after swapping the root with the last leaf. After this swap, the tree is no longer a heap. To rebuild a heap, use the following algorithm:

Listing 18.10 Algorithm for Rebuilding a Heap

```
Let the root be the current node;
while (the current node has children and the current node is smaller than one of its children)
    Swap the current node with the larger of its children;
    Now the current node is one level down;
```

Figure 18.7 shows the process of rebuilding a heap by swapping 9 with 59 in (b), swapping 9 with 44 in (c), and swapping 9 with 30 in (d).
Figure 18.7

Heap sort performs swap and rebuild operations.

The algorithm is implemented in Listing 18.11.

Listing 18.11 Function for Rebuilding a Heap

```java
void rebuildHeap(int list[], int last) {
    int currentIndex = 0;
    bool isHeap = false;
    while (!isHeap) {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;
        int maxIndex = currentIndex;
        if (leftChildIndex <= last &&
            list[maxIndex] < list[leftChildIndex]) {
            maxIndex = leftChildIndex;
        }
        if (rightChildIndex <= last &&
            list[maxIndex] < list[rightChildIndex]) {
            maxIndex = rightChildIndex;
        }
        // Record max value
        list[maxIndex] = list[currentIndex];
        list[currentIndex] = list[maxIndex];
        currentIndex = maxIndex;
    }
}
```

(a) After swapping 62 with 9, 62 is removed from the tree
(b) After swapping 9 with 59
(c) After swapping 9 with 44
(d) After swapping 9 with 30
### 18.6.2 Creating an Initial Heap

You know how to produce a sorted array from a heap. Now the question is how to create a heap from an arbitrary array list initially. You may create a heap by adding a node to the tree one at a time. The heap initially contains list[0] as the root. Add list[1] to the heap. Swap list[0] with list[1] if list[0] < list[1]. Suppose that list[0..k-1] is already a heap. To add list[k] into the heap, consider list[k] as the last node in the existing heap. Compare it with its parent and swap them if list[k] is greater than its parent. Continue the compare and swap operations until list[k] is put in the right place in the heap. Figure 18.8 illustrates the process of adding element 88 to an existing heap (59 42 44 32 39 30 13 22 29 14 33 17).
A new element is added to the heap.

Listing 18.12 presents a function for making list[0..k] a heap, assume that list[0..k-1] is already a heap.

Listing 18.12 Function for Making a Heap

```c
/* Assume list[0..k-1] is a heap, add list[k] to the heap */
void makeHeap(int list[], int k)
{
    int currentIndex = k;

    while (currentIndex > 0 && list[currentIndex] >
        list[(currentIndex - 1) / 2])
    {
        // Swap list[currentIndex] with list[(currentIndex - 1) / 2]
        int temp = list[currentIndex];
        list[currentIndex] = list[(currentIndex - 1) / 2];
        list[(currentIndex - 1) / 2] = temp;

        currentIndex = (currentIndex - 1) / 2;
    }
}
```

The function constructs a heap in array list[0..k]. Before invoking the function, all the nodes in the tree represented in array list[0..k-1] already form a heap. The function
starts from list[k], compares it with its parent, and swaps it with its parent if list[k] is greater than its parent (lines 8-10). Continue to move the new element up to the root or it is less than its parent in the while loop.

18.6.3 Heap Sort Implementation

The complete heap sort algorithm can be implemented in Listing 18.13. The algorithm first creates a heap by adding one element from an array at a time (lines 4-6), and then sorts the array by repeatedly removing the root from the heap (line 9-15).

Listing 18.13 HeapSort.cpp: Heap Sort

```cpp
void heapSort(int list[], int arraySize);  // Function prototype
void makeHeap(int list[], int k);          
void rebuildHeap(int list[], int last);    
void heapSort(int list[], int arraySize)
{
  // Create a heap from the list
  for (int i = 1; i < arraySize; i++)
    makeHeap(list, i);

  // Produce a sorted array from the heap
  for (int last = arraySize - 1; last > 0; )
  {
    // Swap list[0] with list[last]
    int temp = list[last];
    list[last] = list[0];
    list[0] = temp;
    rebuildHeap(list, --last);
  }
}

/* Assume list[0..k-1] is a heap, add list[k] to the heap */
void makeHeap(int list[], int k)
{
  // Same as in Listing 18.12, so omitted
}

void rebuildHeap(int list[], int last)
{
  // Same as in Listing 18.11, so omitted
}
```

18.6.4 Heap Sort Analysis

Let \( h \) denote the height for a heap of \( n \) elements. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the \( k \)th level has \( 2^{k-1} \) nodes, the \( h-1 \)th level has \( 2^{h-2} \) nodes, and the \( h \)th level has at least one node and at most \( 2^{h-1} \) nodes. Therefore,
\[ 1 + 2 + \ldots + 2^{h-2} < n \leq 1 + 2 + \ldots + 2^{h-2} + 2^h \]
i.e.,
\[ 2^{h-1} - 1 < n \leq 2^h - 1 \]

So, \( \log(n+1) \leq h < \log(n+1) + 1 \). Hence, the height of the heap is \( O(\log n) \).

**Side Remark: \( O(n \log n) \) worst-case time**
Since the `makeHeap` function traces a path from a leaf to a root, it takes at most \( h \) steps to add a new element to the heap. Since the `makeHeap` function is invoked \( n \) times, the total time for constructing an initial heap is \( O(n \log n) \).

Since the `rebuildHeap` function traces a path from a root to a leaf, it takes at most \( h \) steps to rebuild a heap after removing the root from the heap. Since the `rebuildHeap` function is invoked \( n \) times, the total time for producing a sorted array from a heap is \( O(n \log n) \).

**Side Remark: heap sort vs. merge sort**
Both merge sort and heap sort requires \( O(n \log n) \) time. Merge sort requires a temporary array for merging two subarrays. Heap sort does not need additional array space. So, heap sort is more space efficient than merge sort.

### 18.7 External Sort

All the sort algorithms discussed in the preceding sections assume that all data to be sorted is available at one time in internal memory such as an array. To sort data stored in an external file, you may first bring data to the memory, then sort it internally. However, if the file is too large, all data in the file cannot be brought to memory at one time. This section discusses how to sort data in a large external file.

For simplicity, assume that two million int values are stored in a binary file named `largedata.dat`. This file was created using the following program:

### Listing 18.14 CreateLargeFile.cpp (Creating a Large Binary File)

***PD: Please add line numbers in the following code***

**Side Remark line 9: a binary output stream**

**Side Remark line 13: random value**

**Side Remark line 14: output an int value**

**Side Remark line 17: close output**

**Side Remark line 21: a binary output stream**

**Side Remark line 27: read input**
A variation of merge sort can be used to sort this file in two phases:

**Phase I:** Repeatedly bring data from the file to an array, sort the array using an internal sorting algorithm, and output the data from the array to a temporary file. This process is shown in Figure 18.9. Ideally, you want to create a large array, but the maximum size of the array is limited. Assume that the maximum array size is of 100000 int values. In the temporary file, every 100000 int values are sorted. They are denoted as $S_1, S_2, \ldots, S_k$, where $S_k$ may contain less than 100000 values.
The original file is sorted by pieces.

**Phase II:** Merge a pair of sorted segments (e.g., $S_1$ with $S_2$, $S_3$ with $S_4$, ..., and so on) into a larger sorted segment and save the new segment into a new temporary file. Continue the same process until one sorted segment results. Figure 18.10 shows how to merge eight segments.

**Figure 18.10**

*Sorted segments are merged iteratively.*

Note: It is not necessarily to merge two successive segments. For example, you may merge $S_1$ with $S_5$, $S_2$ with $S_6$, $S_3$ with $S_7$, and $S_4$ with $S_8$, in the first merge step.

18.7.1 Implementing Phase I

Assume MAX ARRAY SIZE is declared as a constant 100000. Listing 18.15 gives the function that sorts every 100000 in largedata.dat and stores the sorted segments into a new file named f1.dat. The function returns the number of segments.

**Listing 18.15 Creating Initial Sorted Segments**
The function declares an array with the max size in line 4, declares a data input stream for the original file in line 7, and declares a data output stream for a temporary file in line 9.

Lines 14-17 read a segment of data from the file into the array. Line 26 sorts the array. Lines 29-32 write the data in the array to the temporary file.

The number of the segments is returned in line 38. Note that every segment has MAX_ARRAY_SIZE number of elements except the last segment that may have a smaller number of elements.

18.7.2 Implementing Phase II
Each merge step merges two sorted segments to form a new segment. The new segment doubles the number elements. So the number of segments is reduced by half after each merge step. A segment is too large to be brought to an array in memory. To implement a merge step, copy half number of segments from file f1.dat to a temporary file f2.dat. Then merge the first remaining segment in f1.dat with the first segment in f2.dat into a temporary file named f3.dat, as shown in Figure 18.11.

```
   S1  S2  S3  S4  S5  S6  S7  S8  
   f1.dat

   S1  S3 merged  S2  S4 merged  
   f2.dat

   S1, S3 merged  S2, S4 merged  S3, S7 merged  S4, S8 merged  
   f3.dat
```

**Figure 18.11**

Sorted segments are merged iteratively.

Note: f1.dat may have one segment more than f2.dat. If so, move the last segment into f3.dat after the merge.

Listing 18.16 gives a function that copies the first half of the segments in f1.dat to f2.dat. Listing 18.17 gives a function that merges a pair of segments in f1.dat and f2.dat. Listing 18.18 gives a function that merges two segments.

**Listing 18.16 Copying First Half Segments**

```cpp
/* Copy first half number of segments from f1.dat to f2.dat */
void copyHalfToF2(int numberOfSegments, int segmentSize,
      fstream &f1, fstream &f2) {
    for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
        int value;
        f1.read(reinterpret_cast<char *> (& value), sizeof(value));
        f2.write(reinterpret_cast<char *> (& value), sizeof(value));
    }
}
```

**Listing 18.17 Merging All Segments**

```cpp
/* Merge two segments */
void mergeSegments(fstream &f1, fstream &f2, fstream &f3) {
    int i = 0;
    while (i < segmentSize) {
        int value1, value2;
        f1.read(reinterpret_cast<char *> (& value1), sizeof(value1));
        f2.read(reinterpret_cast<char *> (& value2), sizeof(value2));
        f3.write(reinterpret_cast<char *> (& value1), sizeof(value1));
        f3.write(reinterpret_cast<char *> (& value2), sizeof(value2));
        i += 2;
    }
}
```
void mergeSegments(int numberOfSegments, int segmentSize,
    fstream &f1, fstream &f2, fstream &f3)
{
    for (int i = 0; i < numberOfSegments; i++)
    {
        mergeTwoSegments(segmentSize, f1, f2, f3);
    }
    // f1 may have one extra segment, copy it to f3
    while (!f1.eof())
    {
        int value;
        f1.read(reinterpret_cast<char *>(&value), sizeof(value));
        if (f1.eof()) break;
        f3.write(reinterpret_cast<char *>(&value), sizeof(value));
    }
}

Listing 18.18 Merging Two Segments

***PD: Please add line numbers in the following code***
<Side Remark line 2: input stream f1 and f2>
<Side Remark line 3: output stream f3>
<Side Remark line 6: read from f1>
<Side Remark line 8: read from f2>
<Side Remark line 14: write to f3>
<Side Remark line 18: segment in f1 finished>
<Side Remark line 21: write to f3>
<Side Remark line 35: segment in f2 finished>
<Side Remark line 49: remaining f1 segment>
<Side Remark line 58: remaining f2 segment>

void mergeTwoSegments(int segmentSize, fstream &f1, fstream &f2,
    fstream &f3)
{
    int intFromF1;
    f1.read(reinterpret_cast<char *>(&intFromF1), sizeof(intFromF1));
    int intFromF2;
    f2.read(reinterpret_cast<char *>(&intFromF2), sizeof(intFromF2));
    int f1Count = 1;
    int f2Count = 1;
    while (true)
    {
        if (intFromF1 < intFromF2)
        {
            f3.write(reinterpret_cast<char *>(
                &intFromF1), sizeof(intFromF1));
            if (f1.eof() || f1Count++ >= segmentSize)
            {
                if (f1.eof()) break;
                f3.write(reinterpret_cast<char *>(
                    &intFromF2), sizeof(intFromF2));
                break;
            }
            else
            {
                f1.read(reinterpret_cast<char *>(
                    &intFromF1), sizeof(intFromF1));
            }
        }
        else
        {
            f3.write(reinterpret_cast<char *>(
                &intFromF2), sizeof(intFromF2));
            if (f2.eof() || f2Count++ >= segmentSize)
            {
                if (f2.eof()) break;
                f3.write(reinterpret_cast<char *>(
                    &intFromF1), sizeof(intFromF1));
                break;
            }
            else
            {
                f2.read(reinterpret_cast<char *>(
                    &intFromF2), sizeof(intFromF2));
            }
        }
    }
}
while (!f1.eof() && f1Count++ < segmentSize) {
    int value;
    f1.read(reinterpret_cast<char *>(
        &value), sizeof(value));
    if (f1.eof()) break;
    f3.write(reinterpret_cast<char *>(
        &value), sizeof(value));
}

while (!f2.eof() && f2Count++ < segmentSize) {
    int value;
    f2.read(reinterpret_cast<char *>(
        &value), sizeof(value));
    if (f2.eof()) break;
    f3.write(reinterpret_cast<char *>(
        &value), sizeof(value));
}
}

18.7.3 Combining Two Phases

Listing 18.19 gives the complete program for sorting int values in largedata.dat and storing the sorted data in sortedlargedata.dat.

Listing 18.19 SortLargeFile.cpp (Sorting Large File)
***PD: Please add line numbers in the following code***

#include <iostream>
#include <fstream>
#include "QuickSort.h"
using namespace std;

// Function prototype
int initializeSegments(int segmentSize,
    char* originalFile, char* f1);
void mergeTwoSegments(int segmentSize, fstream &f1, fstream &f2,
    fstream &f3);
void merge(int numberOfSegments, int segmentSize,
    char* f1, char* f2, char* f3);
void copyHalfToF2(int numberOfSegments, int segmentSize,
    fstream &f1, fstream &f2);
void mergeOneStep(int numberOfSegments, int segmentSize,
    char* f1, char* f2, char* f3);
void mergeSegments(int numberOfSegments, int segmentSize,
    fstream &f1, fstream &f2, fstream &f3);
void copyFile(char * f1, char * target);

int main()
{
    const int MAX_ARRAY_SIZE = 100000;
    // Implement Phase 1: Create initial segments
    int numberOfSegments = initializeSegments(MAX_ARRAY_SIZE, "largedata.dat", "f1.dat");
    // Implement Phase 2: Merge segments recursively
    merge(numberOfSegments, MAX_ARRAY_SIZE,
        "f1.dat", "f2.dat", "sortedlargedata.dat");
}
"f1.dat", "f2.dat", "f3.dat")

/* Sort original file into sorted segments */
int initializeSegments(int segmentSize, char* originalFile, char* f1)
{
    int *list = new int[segmentSize];
    fstream input;
    input.open(originalFile, ios::in | ios::binary);
    fstream output;
    output.open(f1, ios::out | ios::binary);
    int numberOfSegments = 0;
    while (!input.eof()) {
        int i = 0;
        for (; !input.eof() && i < segmentSize; i++) {
            input.read(reinterpret_cast<char*>(list[i]), sizeof(list[i]));
        }
        if (input.eof()) i--;
        if (i <= 0)
            break;
        else
            numberOfSegments++;
        // Sort an array list[0..i-1]
        quickSort(list, i);
        // Write the array to f1.dat
        for (int j = 0; j < i; j++) {
            output.write(reinterpret_cast<char*>(&list[j]), sizeof(list[j]));
        }
    }
    input.close();
    output.close();
    delete[] list;
    return numberOfSegments;
}

/* Recursively merge sorted segments */
void merge(int numberOfSegments, int segmentSize,
           char* f1, char* f2, char* f3)
{
    if (numberOfSegments > 1)
        mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
    else
        // rename f1 as the final sorted file
        copyFile(f1, "sortedlargedata.dat");
        cout << "\nSorted into the file sortedlargedata.dat" << endl;
}

/* Copy file from f1 to target */
void copyFile(char * f1, char * target)
{
    fstream input;
    input.open(f1, ios::in | ios::binary);
    fstream output;
    output.open(target, ios::out | ios::binary);
    while (!input.eof()) // Continue if not end of file
    {
        int value;
        input.read(reinterpret_cast<char*>(value), sizeof(value));
        if (input.eof()) break;
        output.write(reinterpret_cast<char*>(value), sizeof(value));
    }
    input.close();
}
output.close();

/* Merge every pair of two segments */
void mergeOneStep(int numberOfSegments, int segmentSize, char* f1,
                char* f2, char* f3)
{
  fstream f1Input;
  f1Input.open(f1, ios::in | ios::binary);

  fstream f2Output;
  f2Output.open(f2, ios::out | ios::binary);

  // Copy half number of segments from f1.dat to f2.dat
  copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
  f2Output.close();

  // Merge remaining segments in f1 with segments in f2 into f3
  fstream f2Input;
  f2Input.open(f2, ios::in | ios::binary);
  fstream f3Output;
  f3Output.open(f3, ios::out | ios::binary);
  mergeSegments(numberOfSegments / 2, segmentSize,
                 f1Input, f2Input, f3Output);
  f1Input.close();
  f2Input.close();
  f3Output.close();
}

/* Copy first half number of segments from f1.dat to f2.dat */
void copyHalfToF2(int numberOfSegments, int segmentSize,
                  fstream &f1, fstream &f2)
{
  // Same as Listing 18.15, so omitted
}

/* Merge all segments */
void mergeSegments(int numberOfSegments, int segmentSize,
                   fstream &f1, fstream &f2, fstream &f3)
{
  // Same as Listing 18.16, so omitted
}

/* Merge two segments */
void mergeTwoSegments(int segmentSize, fstream &f1, fstream &f2,
                    fstream &f3)
{
  // Same as Listing 18.17, so omitted
}

Line 27 creates initial segments from the original array and
stores the sorted segments in a new file f1.dat. Lines 30-31
produces a sorted file in sortedlargedata.dat. The merge
function

merge(int numberOfSegments,
      int segmentSize, String f1, String f2, String f3)

merges the segments in f1 into f3 using f2 to assist the
merge. The merge function is invoked recursively with many
merge steps. Each merge step reduces the numberOfSegments by
half and doubles the sorted segment size. After completing
one merge step, the next merge step merges the new segments
in f3 to f2 using f1 to assist the merge. So the statement
to invoke the new merge function is

merge((numberOfSegments + 1) / 2, segmentSize * 2, f3, f1, f2);
The `numberOfSegments` for the next merge step is 
\((numberOfSegments + 1) / 2\). For example, if `numberOfSegments` 
is 5, `numberOfSegments` is 3 for the next merge step, because 
every two segments are merged but there is one left 
unmerged.

The recursive `merge` function ends when `numberOfSegments` is 
1. In this case, `f1` contains sorted data. Copy `f1` to 
sortedlargedata.dat in line 85.

**Key Terms**

***PD: Please place terms in two columns same as in intro5e.***

- average-time analysis
- best-time analysis
- big O notation
- bubble sort
- constant time
- exponential time
- growth rate
- heap sort
- logarithmic time
- quadratic time
- merge sort
- quick sort
- worst-time analysis

**Chapter Summary**

- The Big O notation is a theoretical approach for analyzing the performance of the algorithm. It estimates how fast an algorithm’s execution time increases as the input size increases. so you can compare two algorithms by examining their growth rates.

- An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.

- An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems.
• If the time is not related to the input size, the algorithm is said to take constant time with the notation $O(1)$.

• Linear search takes $O(n)$ time. An algorithm with the $O(n)$ time complexity is called a linear algorithm. Binary search takes $O(\log n)$ time. An algorithm with the $O(\log n)$ time complexity is called a logarithmic algorithm.

• The worst time complexity for selection sort, insertion sort, bubble sort, and quick sort is $O(n^2)$. An algorithm with the $O(n^2)$ time complexity is called a quadratic algorithm.

• The average-time and worst-time complexity for merge sort and heap sort is $O(n \log n)$. The average time for quick sort is also $O(n \log n)$. An algorithm with the $O(n \log n)$ time complexity is called a log-linear time.

• A variation of merge sort can be applied to sort large data from external files.

**Review Questions**

*Sections 18.2 Estimating Algorithm Efficiency*

18.1

Put the following growth functions in order:

$$\frac{5n^3}{4032}, \ 44\log n, \ 10n \log n, \ 500, \ 2n^2, \ \frac{2^n}{45}, \ 3n$$

18.2

Use the Big O notation to estimate the time complexity of the following functions:
public static void mA(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(Math.random());
    }
}

public static void mB(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++)
            System.out.print(Math.random());
    }
}

public static void mC(int[] m) {
    for (int i = 0; i < m.length; i++) {
        System.out.print(m[i]);
    }
    for (int i = m.length - 1; i >= 0; )
    {
        System.out.print(m[i]);
        i--;
    }
}

public static void mD(int[] m) {
    for (int i = 0; i < m.length; i++) {
        for (int j = 0; j < i; j++)
            System.out.print(m[i] * m[j]);
    }
}

18.3

Estimate the time complexity for adding two $n \times m$ matrices, and for multiplying a $n \times m$ matrix with a $m \times k$ matrix.

Sections 18.3-18.7

18.4

Use Figure 18.1 as an example to show how to apply bubble sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

18.5

Use Figure 18.2 as an example to show how to apply merge sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

18.6

See Figure 18.4 as an example to show how to apply quick sort on {45, 11, 50, 59, 60, 2, 4, 7, 10}.

18.7

Show the steps of creating a heap using {45, 11, 50, 59, 60, 2, 4, 7, 10}.

18.8

Given the following heap, show the steps of removing all nodes from the heap.
There are 10 numbers \{2, 3, 4, 0, 5, 6, 7, 9, 8, 1\} stored in the external file largedata.dat. Trace the SortLargeFile program by hand with \texttt{MAX\_ARRAY\_SIZE 2}.

**Programming Exercises**

18.1
(Generic bubble sort) Write a generic function for bubble sort.

18.2
(Generic merge sort) Write a generic function for merge sort.

18.3
(Generic quick sort) Write a generic function for quick sort.

18.4
(Improving quick sort) The quick sort algorithm presented in the book selects the first element in the list as the pivot. Revise it by selecting the medium among the first, middle, and last elements in the list.

18.5
(Generic heap sort) Write a generic function for heap sort.

18.6
(Checking order) Write the following overloaded functions that check whether an array is ordered in ascending order, or descending order. By default, the function checks ascending order. To check descending order, pass \texttt{false} to the ascending argument in the function.

\begin{verbatim}
bool ordered(T list[]) // T is a generic type
bool ordered(T list[], bool ascending) // T is a generic type
\end{verbatim}