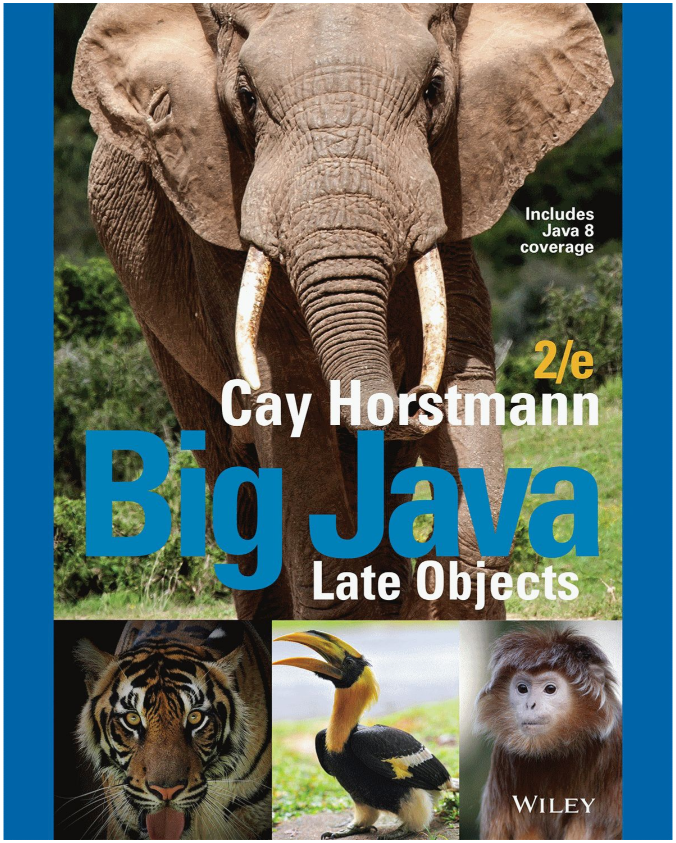
Chapter 14 - Sorting and Searching



Chapter Goals

- To study several sorting and searching algorithms

- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To estimate and compare the performance of algorithms
- To write code to measure the running time of a program

Selection Sort

- A sorting algorithm rearranges the elements of a collection so that they are stored in sorted order.
- Selection sort sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.
- Slow when run on large data sets.
- Example: sorting an array of integers

11 9 17 5 12

Sorting an Array of Integers

1. Find the smallest and swap it with the first element

   5 9 17 11 12

2. Find the next smallest. It is already in the correct place

   5 9 17 11 12

3. Find the next smallest and swap it with first element of unsorted portion

   5 9 11 17 12

4. Repeat

   5 9 11 12 17

5. When the unsorted portion is of length 1, we are done

5 9 11 12 17

Selection Sort

In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.

section_1/SelectionSorter.java

```
/**
      The sort method of this class sorts an array, using the selection
      sort algorithm.
*/
public class SelectionSorter
{
/**
         Sorts an array, using selection sort.
         @param a the array to sort
```

section_1/SelectionSortDemo.java

```java
import java.util.Arrays;

/**
    This program demonstrates the selection sort algorithm by
    sorting an array that is filled with random numbers.
*/
public class SelectionSortDemo
{
public static void main(String[] args)
```

section_1/ArrayUtil.java

```java
import java.util.Random;

/**
This class contains utility methods for array manipulation.
*/
public class ArrayUtil
{
private static Random generator = new Random();

```

Typical Program Run:

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

Self Check 14.1

Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?

Answer: Dropping the `temp` variable would not work. Then `a[i]` and `a[j]` would end up being the same value.

Self Check 14.2

What steps does the selection sort algorithm go through to sort the sequence `6 5 4 3 2 1`?

Answer:

```
1   5   4   3   2   6

1   2   4   3   5   6

1   2   3   4   5   6
```

Self Check 14.3

How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?

Answer: In each step, find the maximum of the remaining elements and swap it with the current element (or see Self Check 4).

Self Check 14.4

Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?

> **Answer:** The modified algorithm sorts the array in descending order.

Profiling the Selection Sort Algorithm

- We want to measure the time the algorithm takes to execute:
  - Exclude the time the program takes to load
    Exclude output time
- To measure the running time of a method, get the current time immediately before and after the method call.
- We will create a `StopWatch` class to measure execution time of an algorithm:
  - It can start, stop and give elapsed time
  - Use `System.currentTimeMillis` method
- Create a `StopWatch` object:
  - Start the stopwatch just before the sort
  - Stop the stopwatch just after the sort
  - Read the elapsed time

section_2/StopWatch.java

```java
/**
      A stopwatch accumulates time when it is running. You can
      repeatedly start and stop the stopwatch. You can use a
      stopwatch to measure the running time of a program.
*/
public class StopWatch
{
   private long elapsedTime;
   private long startTime;
```

section_2/SelectionSortTimer.java

```java
import java.util.Scanner;

/**
      This program measures how long it takes to sort an
      array of a user-specified size with the selection
      sort algorithm.
*/
public class SelectionSortTimer
{
```

Program Run:

```
Enter array size: 50000
Elapsed time: 13321 milliseconds
```

Selection Sort on Various Size Arrays

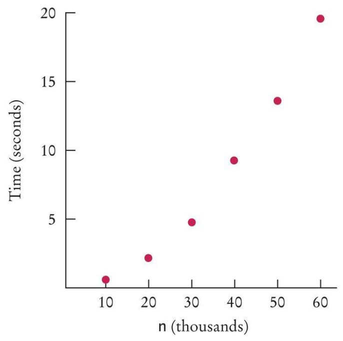


Figure 1 Time Taken by Selection Sort

n	Milliseconds
10,000	786
20,000	2,148
30,000	4,796
40,000	9,192
50,000	13,321
60,000	19,299

Doubling the size of the array more than doubles the time needed to sort it.

Self Check 14.5

Approximately how many seconds would it take to sort a data set of 80,000 values?

Answer: Four times as long as 40,000 values, or about 37 seconds.

Self Check 14.6

Look at the graph in Figure 1. What mathematical shape does it resemble?

Answer: A parabola.

Analyzing the Performance of the Selection Sort Algorithm

- In an array of size *n*, count how many times an array element is visited:
    - To find the smallest, visit *n* elements + 2 visits for the swap
    - To find the next smallest, visit *(n* - 1) elements + 2 visits for the swap
    - The last term is 2 elements visited to find the smallest + 2 visits for the swap

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits:
  - $n + 2 + (n - 1) + 2 + (n - 2) + 2 + . . .+ 2 + 2$
  - This can be simplified to $n^2 /2 + 5n/2 - 3$
  - $5n/2 - 3$ is small compared to $n^2 /2$ – so let's ignore it
  - Also ignore the 1/2 – it cancels out when comparing ratios

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits is of the order n^2.
- Computer scientists use the big-Oh notation to describe the growth rate of a function.
- Using big-Oh notation: The number of visits is $O(n^2)$.
- Multiplying the number of elements in an array by **2** multiplies the processing time by **4.**
- To convert to big-Oh notation: locate fastest-growing term, and ignore constant coefficient.

Self Check 14.7

If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?

Answer: It takes about 100 times longer.

Self Check 14.8

How large does n need to be so that $1/2\ n^2$ is bigger than $5/2\ n - 3$?

Answer: If n is 4, then n^2 is 8 and $5/2\ n - 3$ is 7.

Self Check 14.9

Section 7.3.6 has two algorithms for removing an element from an array of length *n*. How many array visits does each algorithm require on average?

Answer: The first algorithm requires one visit, to store the new element. The second algorithm requires $T(p) = 2 \times (n - p - 1)$ visits, where p is the location at which the element is removed. We don't know where that element is, but if elements are removed at random locations, on average, half of the removals will be above the middle and half below, so we can assume an average p of $n / 2$ and $T(n) = 2 \times (n - n / 2 - 1) = n - 2$.

Self Check 14.10

Describe the number of array visits in Self Check 9 using the big-Oh notation.

Answer: The first algorithm is $O(1)$, the second $O(n)$.

Self Check 14.11

What is the big-Oh running time of checking whether an array is already sorted?

Answer: We need to check that `a[0] ≤ a[1]`, `a[1] ≤ a[2]`, and so on, visiting $2n - 2$ elements. Therefore, the running time is $O(n)$.

Self Check 14.12

Consider this algorithm for sorting an array. Set k to the length of the array. Find the maximum of the first k elements. Remove it, using the second algorithm of Section 7.3.6. Decrement k and place the

removed element into the k th position. Stop if k is 1. What is the algorithm's running time in big-Oh notation?

Answer: Let n be the length of the array. In the kth step, we need k visits to find the minimum. To remove it, we need an average of $k - 2$ visits (see Self Check 9). One additional visit is required to add it to the end. Thus, the kth step requires $2k - 1$ visits. Because k goes from n to 2, the total number of visits is

$$2n - 1 + 2(n - 1) - 1 + \ldots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 =$$

$$2(n + (n - 1) + \ldots + 3 + 2 + 1 - 1) - (n - 1) =$$

$$n(n + 1) - 2 - n + 1 = n^2 - 3$$

$$(\text{because } 1 + 2 + 3 + \ldots + (n - 1) + n = n\,(n + 1) / 2)$$

Therefore, the total number of visits is $O(n^2)$.

Common Big-Oh Growth Rates

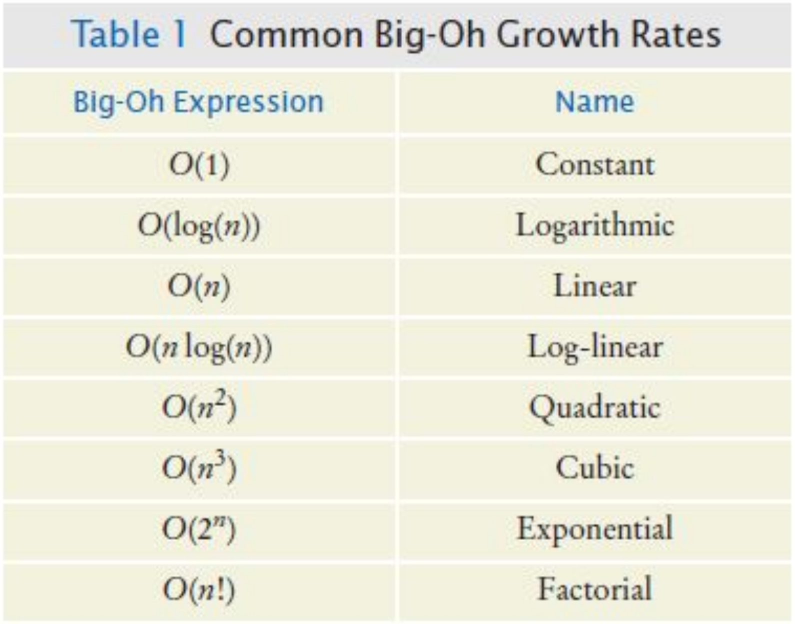


Table 1 Common Big-Oh Growth Rates

Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial