Week 14
Sorting and Searching
(Chapter 14)

Chapter Goals

© Volkan Ersoy/iStockphoto.

- 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

© Zone Creative/iStockphoto.

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.

- Insertion sort is an $O(n^2)$ algorithm.

section_1/SelectionSorter.java

```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
      */
       public static void sort(int[] a)
         {
            for (int i = 0; i < a.length - 1; i++)
            {
               int minPos = minimumPosition(a, i);
               ArrayUtil.swap(a, minPos, i);
               System.out.println(Arrays.toString(a));
            }
         }
```

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)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));

      SelectionSorter.sort(a);

      System.out.println(Arrays.toString(a));
   }
}
```

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]
```

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

```
/**
    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

```
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

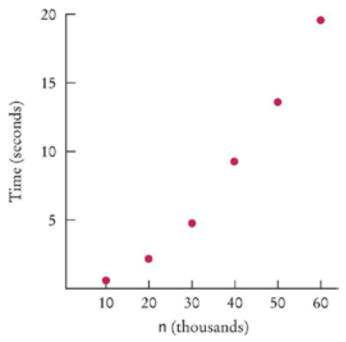


Figure 1 Time Taken by Selection Sort

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.

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:
  - $n + 2 + (n - 1) + 2 + (n - 2) + 2 + \ldots + 2 + 2$
  - 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.

Common Big-Oh Growth Rates

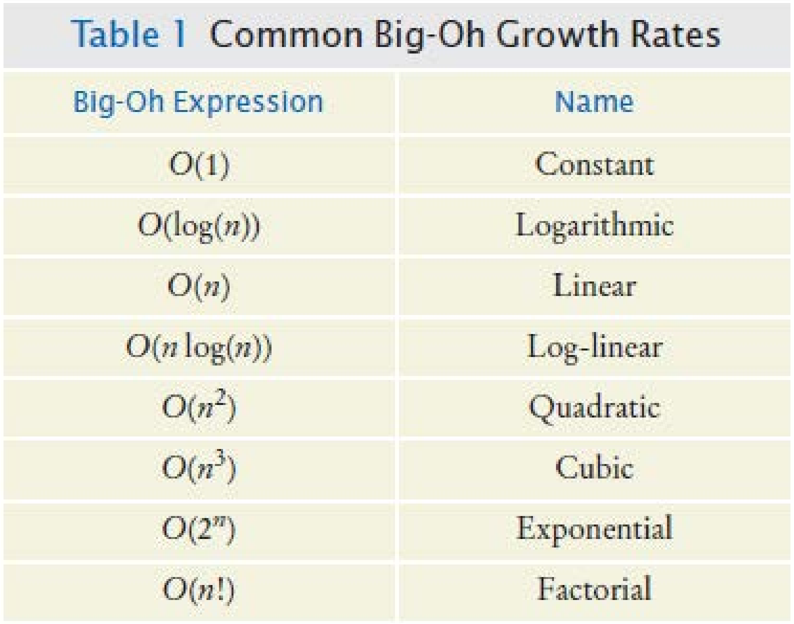
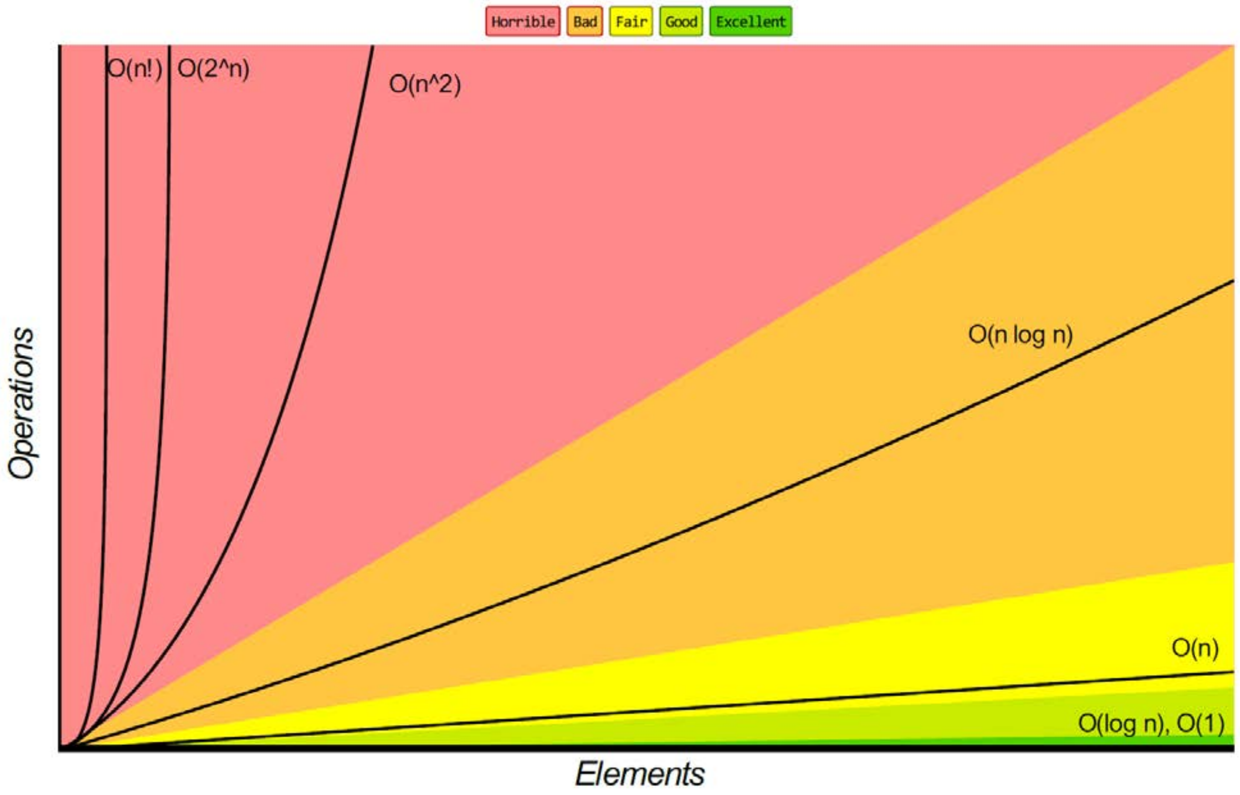


Table 1 Common Big-Oh Growth Rates

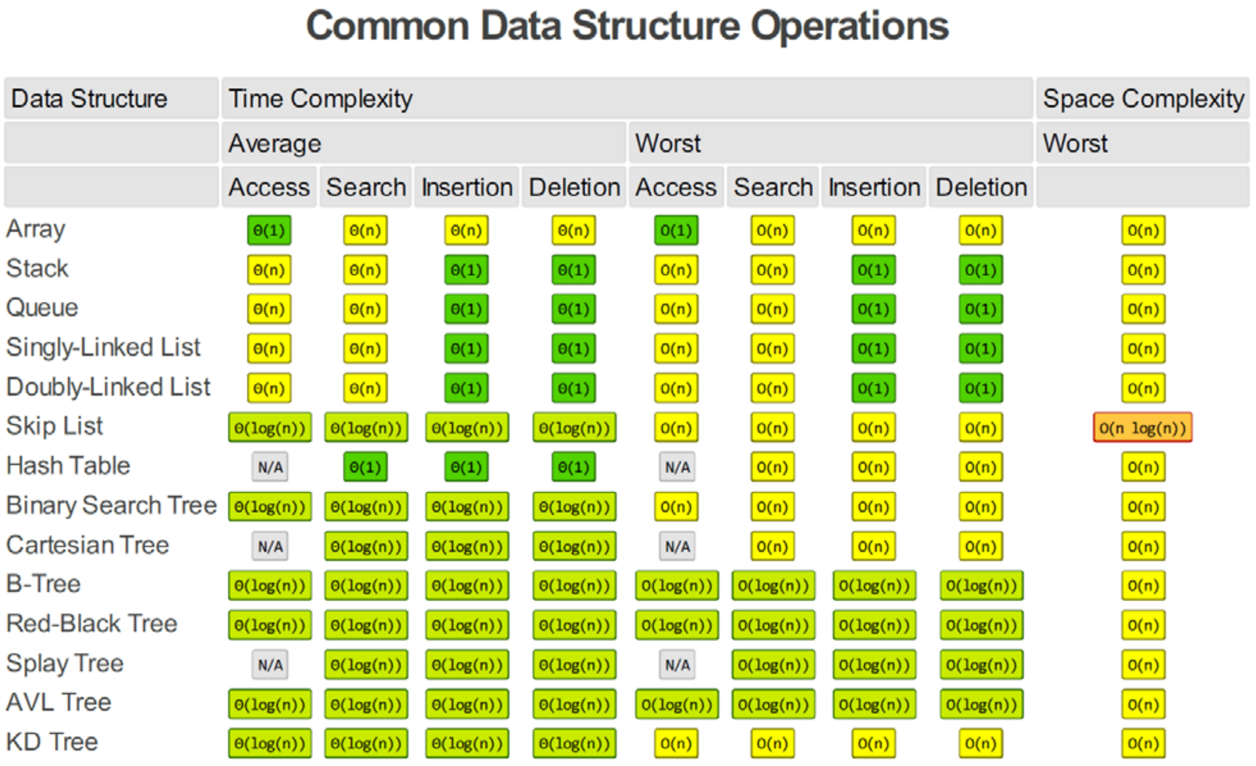
Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Big-Oh Complexity Chart



http://bigocheatsheet.com/

Big-Oh Complexity Chart



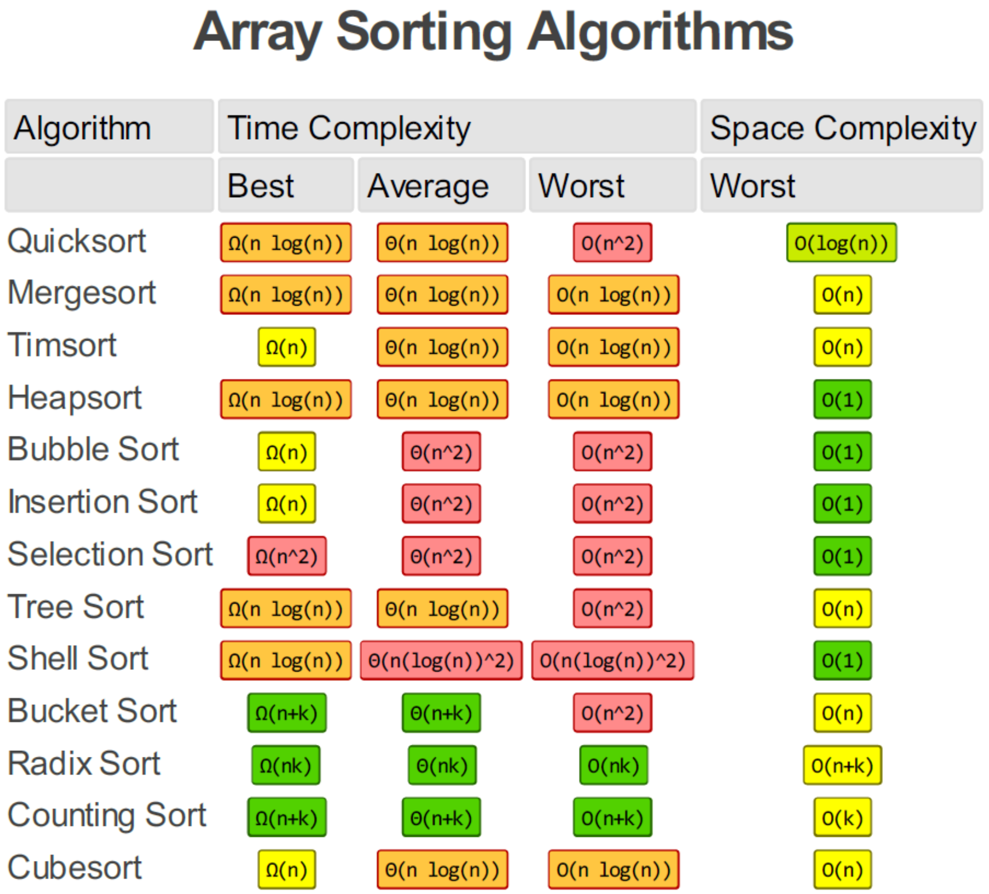
Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

http://bigocheatsheet.com/

Big-Oh Complexity Chart

http://bigocheatsheet.com/



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

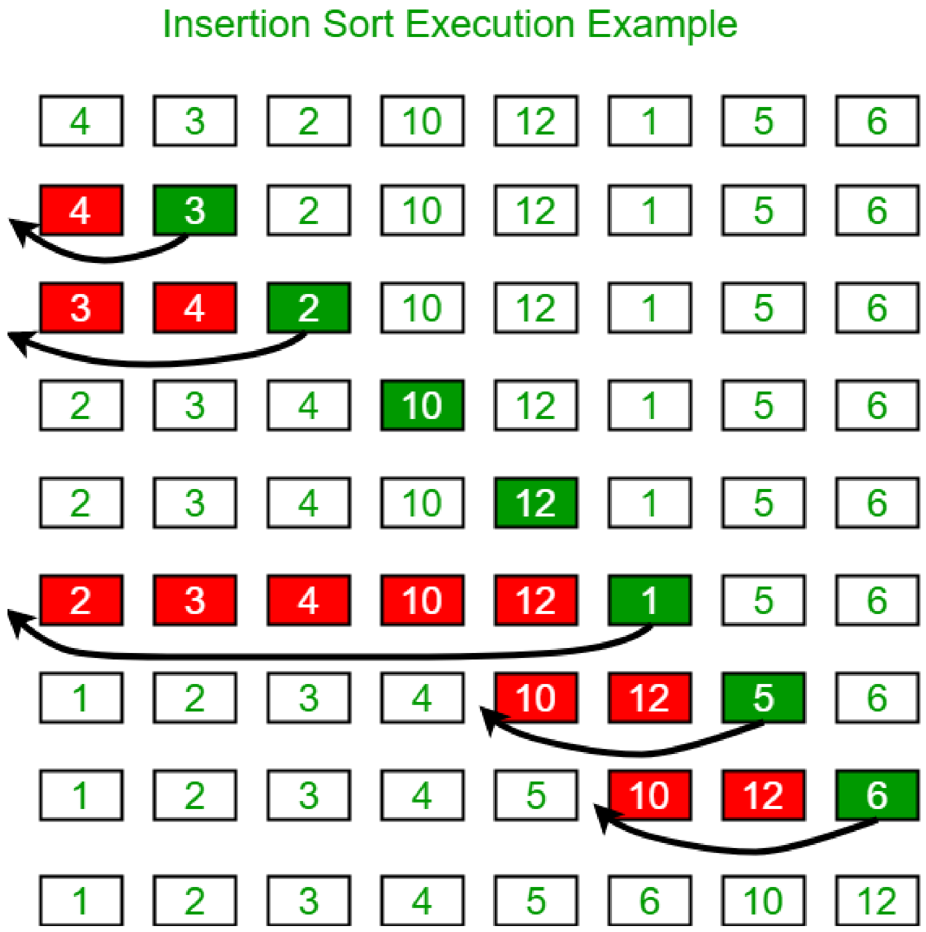
Insertion Sort

- Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.

© Kirby Hamilton/iStockphoto.

- Insertion sort is an $O(n^2)$ algorithm.

Insertion Sort



Insertion Sort

- Assume initial sequence `a[0] . . . a[k]` is sorted (`k = 0`):

  11 9 16 5 7

- Add `a[1]`; element needs to be inserted before `11`

  9 11 16 5 7

- Add `a[2]`

  9 11 16 5 7

- Add `a[3]`

  5 9 11 16 7

- Finally, add `a[4]`

  5 7 9 11 16

Insertion Sort

```
public class InsertionSorter
{
   /**
      Sorts an array, using insertion sort.
      @param a the array to sort
   */
   public static void sort(int[] a)
   {
      for (int i = 1; i < a.length; i++)
      {
         int next = a[i];
         // Move all larger elements up
         int j = i;
         while (j > 0 && a[j - 1] > next)
         {
            a[j] = a[j - 1];
            j--;
         }
         // Insert the element
         a[j] = next;
      }
   }
}
```

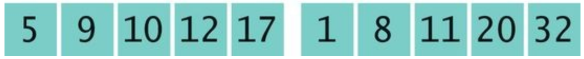
- Insertion sort is an $O(n^2)$ algorithm.

Merge Sort

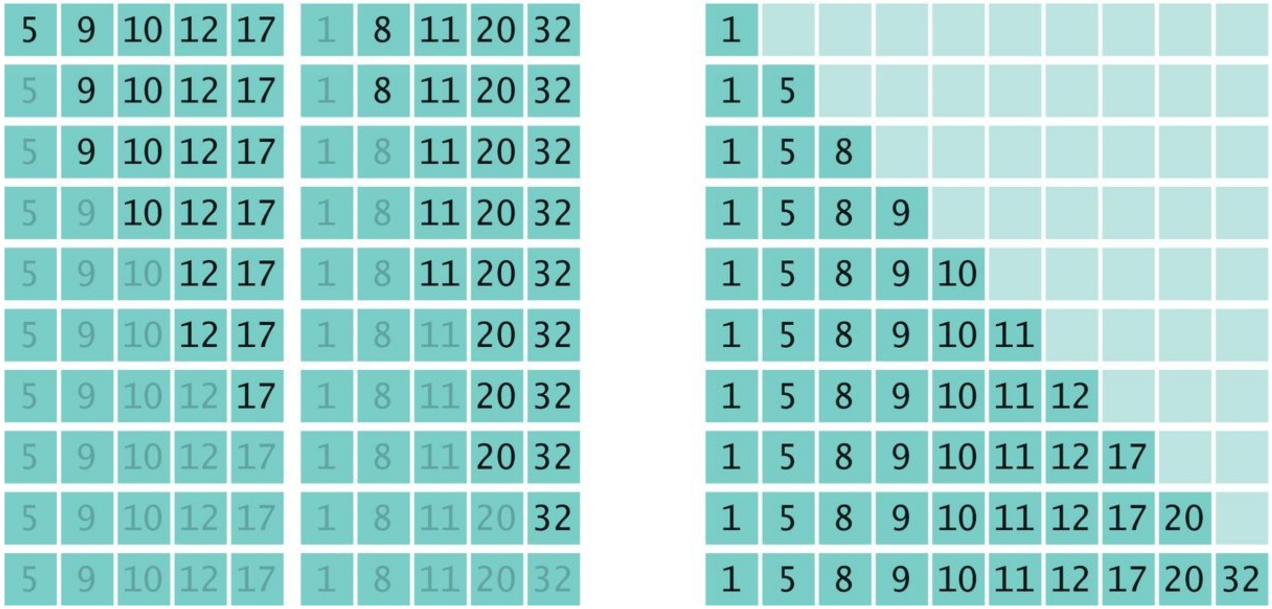
- Sorts an array by
  - Cutting the array in half
  - Recursively sorting each half
  - Merging the sorted halves
- Dramatically faster than the selection sort
- In merge sort, one sorts each half, then merges the sorted halves.

Merge Sort Example

- Divide an array in half and sort each half



- Merge the two sorted arrays into a single sorted array (*choose from unmerged elements*)



Merge Sort

```
public static void sort(int[] a)
{
  if (a.length <= 1) { return; }
  int[] first = new int[a.length / 2];
  int[] second = new int[a.length - first.length];
  // Copy the first half of a into first, the second half into second
  . . .
  sort(first);
  sort(second);
  merge(first, second, a);
}
```

section_4/MergeSorter.java

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

section_4/MergeSortDemo.java

```
import java.util.Arrays;

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

Typical Program Run:

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 76, 81, 89, 90, 98]
```

Analyzing the Merge Sort Algorithm

- In an array of size n, count how many times an array element is visited.
- Assume n is a power of 2: $n = 2^m$.
- Calculate the number of visits to create the two sub-arrays and then merge the two sorted arrays:

  3 visits to merge each element or $3n$ visits

  $2n$ visits to create the two sub-arrays

  total of $5n$ visits

Analyzing the Merge Sort Algorithm

- Let $T(n)$ denote the number of visits to sort an array of n elements then

  $\mathrm{T}(n) = T(n / 2) + T(n / 2) + 5n$ or

  $\mathrm{T}(n) = 2\mathrm{T}(n / 2) + 5n$

- The visits for an array of size $n / 2$ is: $T(n / 2) = 2T(n / 4) + 5\, n / 2$

  So $\mathrm{T}(n) = 2 \times 2\mathrm{T}(n /4) +5n + 5n$

- The visits for an array of size $n / 4$ is: $T(n / 4) = 2T(n / 8) + 5\, n / 4$

  So $\mathrm{T}(n) = 2 \times 2 \times 2\mathrm{T}(n / 8) + 5n + 5n + 5n$

Analyzing Merge Sort Algorithm

- Repeating the process *k* times: $T(n) = 2^{k}T(n / 2^{k})+5nk$
- Since $n = 2^{m}$, when $k = m$: $T(n) = 2^{m}T(n / 2^{m}) +5nm$
- $T(n) = 2^{m}T(2^{m} / 2^{m}) = nT(1)+5nm$
- $T(n) = n + 5n\log_{2}(n)$

Analyzing Merge Sort Algorithm

- To establish growth order:

  Drop the lower-order term *n*

  Drop the constant factor 5

  Drop the base of the logarithm since all logarithms are related by a constant factor

  We are left with *n* log(*n*)

- Using big-Oh notation: number of visits is *O*(*n* log(*n*)).

Merge Sort Vs SelectionSort

- Selection sort is an $O(n^2)$ algorithm.
- Insertion sort is an $O(n^2)$ algorithm.
- Merge sort is an $O(n \log(n))$ algorithm.
- The $n \log(n)$ function grows much more slowly than n^2.

Merge Sort Timing vs. Selection Sort

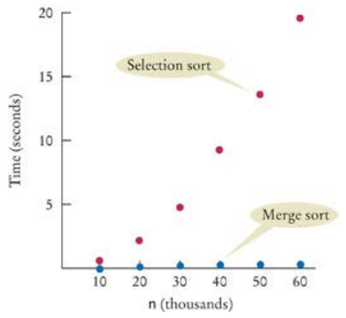


Figure 2 Time Taken by Merge vs. Selection Sort

n	Merge Sort (milliseconds)	Selection Sort (milliseconds)
10,000	40	786
20,000	73	2,148
30,000	134	4,796
40,000	170	9,192
50,000	192	13,321
60,000	205	19,299

The Quicksort Algorithm

- No temporary arrays are required. Divide and conquer

  1. Partition the range using “pivot”
  2. Sort each partition

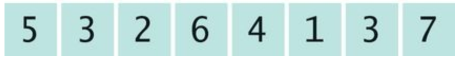
- In quicksort, one partitions the elements into two groups, holding all smaller elements on one partition and all larger elements in another. Then one sorts each group.

```
public void sort(int from, int to)
{
   if (from >= to) return;
   int p = partition(from, to);
   sort(from, p);
   sort(p + 1, to);
}
```

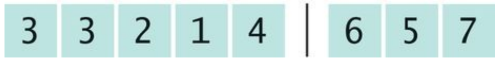
© Christopher Futcher/iStockphoto.

The Quicksort Algorithm

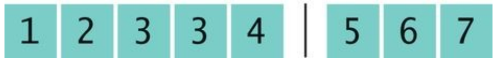
- Starting range



- A partition of the range so that no element in first section is larger than element in second section



- Recursively apply the algorithm until array is sorted



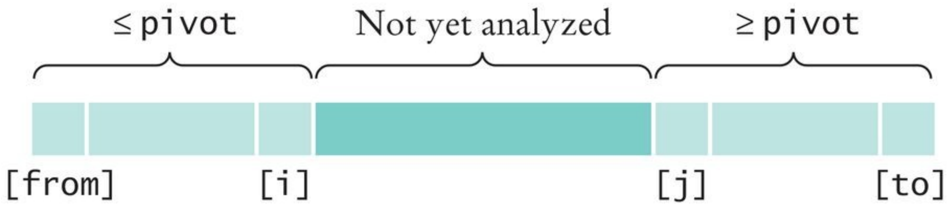
Demo

- http://me.dt.in.th/page/Quicksort/

The Quicksort Algorithm

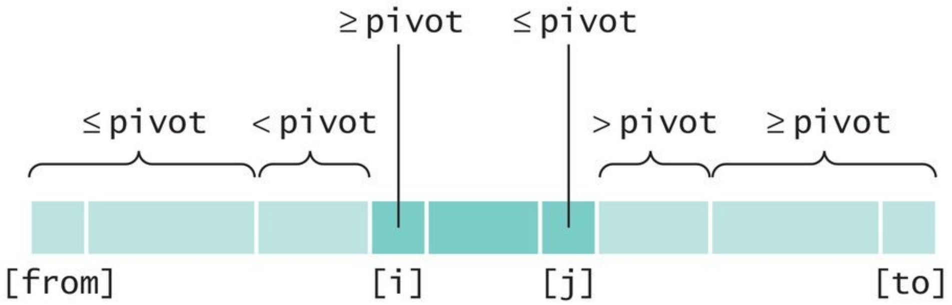
```
private static int partition(int[] a, int from, int to)
{
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i < j)
    {
        i++; while (a[i] < pivot) { i++; }
        j--; while (a[j] > pivot) { j--; }
        if (i < j) { ArrayUtil.swap(a, i, j); }
    }
    return j;
}
```

The Quicksort Algorithm - Partitioning



Partitioning a Range

Extending the Partitions



The Quicksort Algorithm

- On average, the quicksort algorithm is an $O(n \log(n))$ algorithm.
- Its worst-case run-time behavior is $O(n^2)$.
- If the pivot element is chosen as the first element of the region,
    - That worst-case behavior occurs when the input set is already sorted

Searching

- **Linear search:** also called **sequential search.**
- Examines all values in an array until it finds a match or reaches the end
- Number of visits for a linear search of an array of *n* elements:

  The average search visits *n*/2 elements

  The maximum visits is *n*
- A linear search locates a value in an array in *O*(*n*) steps

section_6_1/LinearSearcher.java

```
/**
    A class for executing linear searches in an array.
*/
public class LinearSearcher
{
   /**
      Finds a value in an array, using the linear search
      algorithm.
      @param a the array to search
```

section_6_1/ LinearSearchDemo.java

```java
import java.util.Arrays;
import java.util.Scanner;

/**
    This program demonstrates the linear search algorithm.
*/
public class LinearSearchDemo
{
    public static void main(String[] args)
```

Program Run:

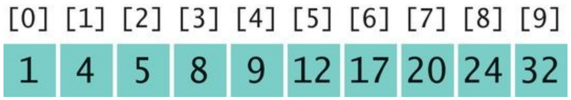
```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45,  88]
Enter number to search for, -1 to quit: 12
Found in position -1
Enter number to search for, -1 to quit: -1
```

Binary Search

- A binary search locates a value in a **sorted** array by:
  - Determining whether the value occurs in the first or second half
  - Then repeating the search in one of the halves
- The size of the search is cut in half with each step.

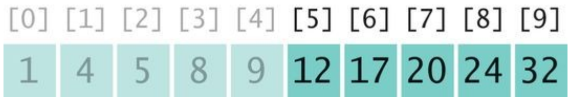
Binary Search

- Searching for 15 in this array



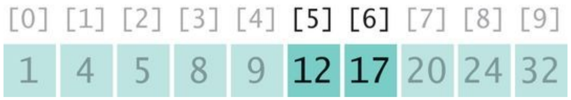
- The last value in the first half is 9

  So look in the second (darker colored) half



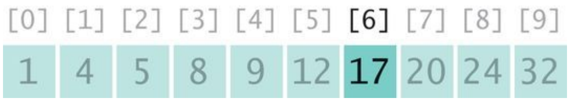
- The middle element of this sequence is 20, so the value must be in this sequence

  Look in the darker colored sequence



- The last value of the first half of this very short sequence is 12,

  This is smaller than the value that we are searching, so we must look in the second half



- 15 ≠ 17: we don't have a match

section_6_2/BinarySearcher.java

```
/**
    A class for executing binary searches in an array.
*/
public class BinarySearcher
{
   /**
      Finds a value in a range of a sorted array, using the binary
      search algorithm.
      @param a the array in which to search
```

Binary Search

- Count the number of visits to search a sorted array of size *n*
  - We visit one element (the middle element) then search either the left or right subarray
  - Thus: $T(n) = T(n/2) + 1$
- If *n* is $n / 2$, then $T(n / 2) = T(n / 4) + 1$
- Substituting into the original equation: $T(n) = T(n / 4) + 2$
- This generalizes to: $T(n) = T(n / 2^k) + k$

Binary Search

- Assume n is a power of 2, $n = 2^m$ where $m = \log_2(n)$
- Then: $T(n) = 1 + \log_2(n)$
- A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Binary Search

- Should we sort an array before searching?
  - Linear search - O(n)
  - Binary search - O(n log(n))
- If you search the array only once
  - Linear search is more efficient
- If you will make many searches
  - Worthwhile to sort and use binary search

Problem Solving: Estimating the Running Time of an Algorithm -Linear time

- Example: an algorithm that counts how many elements have a particular value

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
   if (a[i] == value) { count++; }
}
```

- Pattern of array element visits

Lightbulbs: © Kraska/iStockphoto.

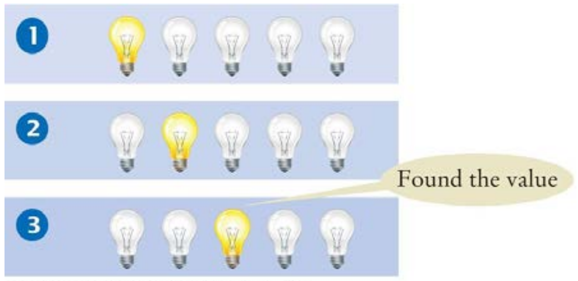
- There are a fixed number of actions in each visit independent of n.
- A loop with n iterations has O(n) running time if each step consists of a fixed number of actions.

Problem Solving: Estimating the Running Time of an Algorithm -Linear time

- Example: an algorithm to determine if a value occurs in the array

```
boolean found = false;
for (int i = 0; !found && i < a.length; i++)
{
   if (a[i] == value) { found = true; }
}
```

- Search may stop in the middle



Lightbulbs: © Kraska/iStockphoto.

- Still *O*(*n*) because we may have to traverse the whole array.

Problem Solving: Estimating the Running Time of an Algorithm -Quadratic time

- Problem: Find the most frequent element in an array.
- Try it with this array

8	7	5	7	7	5	4

- Count how often each element occurs.

  Put the counts in an array

a:	8	7	5	7	7	5	4
counts:	1	3	2	3	3	2	1

Find the maximum count

It is 3 and the corresponding value in original array is 7

Problem Solving: Estimating the Running Time of an Algorithm -Quadratic time

- Estimate how long it takes to compute the counts

```
for (int i = 0; i < a.length; i++)
{
     counts[i] = Count how often a[i] occurs in a
}
```

  - We visit each array element once - $O(n)$
  - Count the number of times that element occurs - $O(n)$
  - Total running time - $O(n^2)$

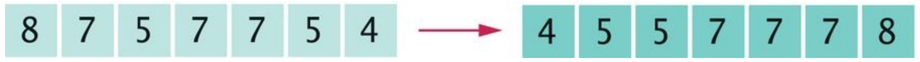
Problem Solving: Estimating the Running Time of an Algorithm -Quadratic time

- Three phases in the algorithm
  - Compute all counts. - O(n^2)
    Compute the maximum. O(n)
  - Find the maximum in the counts. O(n)
- A loop with *n* iterations has $O(n^2)$ running time if each step takes $O(n)$ time.
- The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.

Problem Solving: Estimating the Running Time of an Algorithm -Logarithmic time

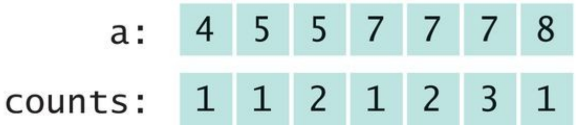
- Logarithmic time estimates arise from algorithms that cut work in half in each step.
- Another ideas for finding the most frequent element in an array:

  Sort the array first



  This is O(n log(n)) time

- Traverse the array and count how many times you have seen that element:



a:	4	5	5	7	7	7	8
counts:	1	1	2	1	2	3	1

Problem Solving: Estimating the Running Time of an Algorithm -Logarithmic time

- The code

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
   count++;
   if (i == a.length - 1 || a[i] != a[i + 1])
   {
      counts[i] = count;
      count = 0;
   }
}
```

Problem Solving: Estimating the Running Time of an Algorithm -Logarithmic time

- This takes the same amount of work per iteration:

  visits two elements 2n

  which is O(n)

Lightbulbs: © Kraska/iStockphoto.

- Running time of entire algorithm is *O*(*n* log(*n*)).
- An algorithm that cuts the size of work in half in each step runs in *O*(log(*n*)) time.

Sorting and Searching in the Java Library - Sorting

- You do not need to write sorting and searching algorithms.

  Use methods in the Arrays and Collections classes.

- The `Arrays` class contains static `sort` methods. To sort an array of integers:

```
int[] a = . . . ;
Arrays.sort(a);
```

  That `sort` method uses the Quicksort algorithm (see Special Topic 14.3).

- To sort an `ArrayList`, use `Collections.sort`

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

  Uses merge sort algorithm

Sorting and Searching in the Java Library - Binary Search

- `Arrays` and `Collections` classes contain static
- `binarySearch` methods.

  If the element is not found, returns -k - 1

  Where k is the position before which the element should be inserted

- For example

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// Returns –3; v should be inserted before position 2
```

Comparing Objects

- `Arrays.sort` sorts objects of classes that implement `Comparable` interface:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

  - The call `a.compareTo(b)` returns
    - A negative number if a should come before b
    - 0 if a and b are the same
    - A positive number otherwise

Comparing Objects

- Several classes in Java (e.g. `String` and `Date`) implement `Comparable`.
- You can implement `Comparable` interface for your own classes.
- The `Country` class could implement `Comparable`:

```
public class Country implements Comparable<Country>
{
   public int compareTo(Country other)
   {
       return Double.compare(area, other.area);
   }
}
```

- You could pass an array of countries to `Arrays.sort`

```
Country[] countries = new Country[n];
// Add countries

Arrays.sort(countries); // Sorts by increasing area
```