Sorting Algorithms

And

Sorting in Java

Algorithm Defined

In mathematics and computing, an algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

Informally, the concept of an algorithm is often illustrated by the example of a recipe, although many algorithms are much more complex; algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison). Algorithms can be composed to create more complex algorithms.

The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers. The concept was formalized in 1936 through Alan Turing's Turing machines and Alonzo Church's lambda calculus, which in turn formed the foundation of computer science.

Computer programs can directly implement most algorithms; computer programs can at least in theory simulate any other algorithms. In many programming languages, algorithms are implemented as functions or procedures.

Sorting

One of the fundamental problems of computer science is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightening-fast results.

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is a complex subject that would take too much time to explain here, but suffice it to say that there's a direct correlation between the complexity of an algorithm and its relative efficiency.

Algorithmic complexity is generally written in a form known as Big-O notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

For example, O(n) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity were $O(n^2)$ (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

Sorting (cont.)

The two classes of sorting algorithms are O(n2), which includes the bubble, insertion, selection, and shell sorts; and O(*n log n*), which includes the heap, merge, and quick sorts.

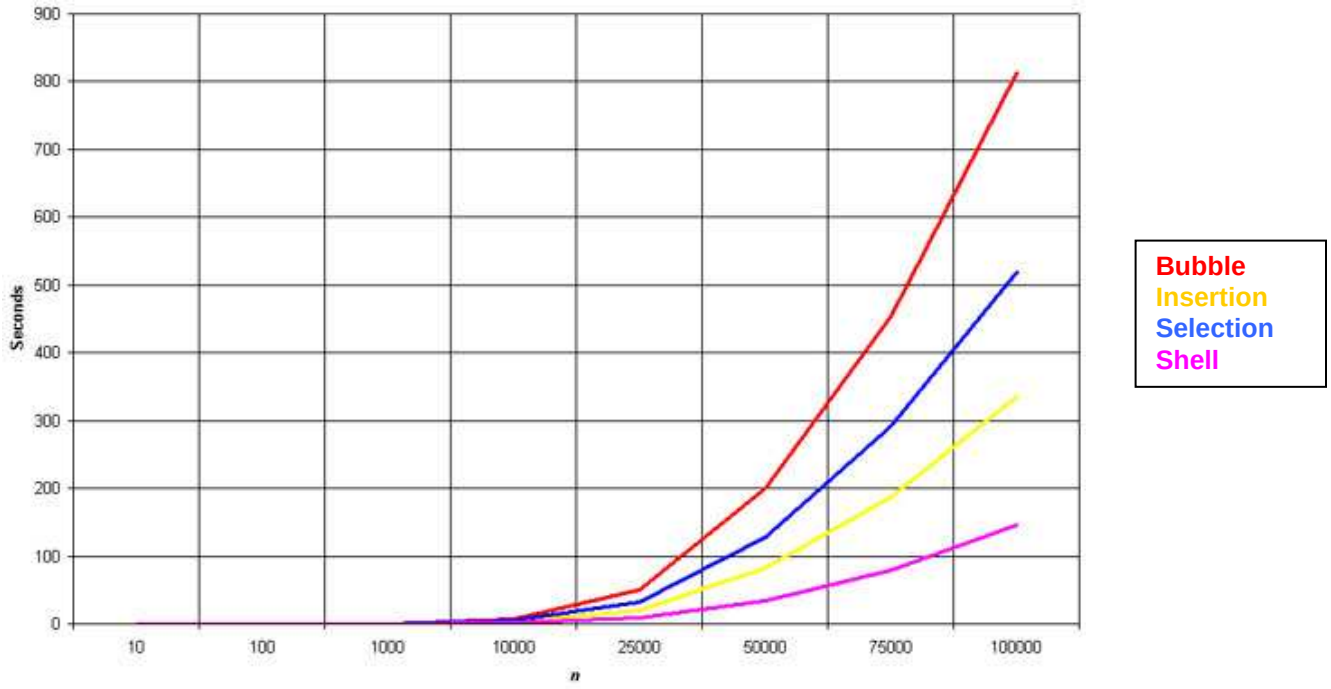
In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. The empirical data on this site is the average of a hundred runs against random data sets.

The run times on your system will almost certainly vary from these results, but the relative speeds should be the same - the selection sort runs in roughly half the time of the bubble sort, and it should run in roughly half the time on whatever system you use as well.

These empirical efficiency graphs are kind of like golf - the lowest line is the "best". Keep in mind that "best" depends on your situation - the quick sort may look like the fastest sort, but using it to sort a list of 20 items is kind of like going after a fly with a sledgehammer.

Sorting Algorithms

$O(n^2)$ Sorts

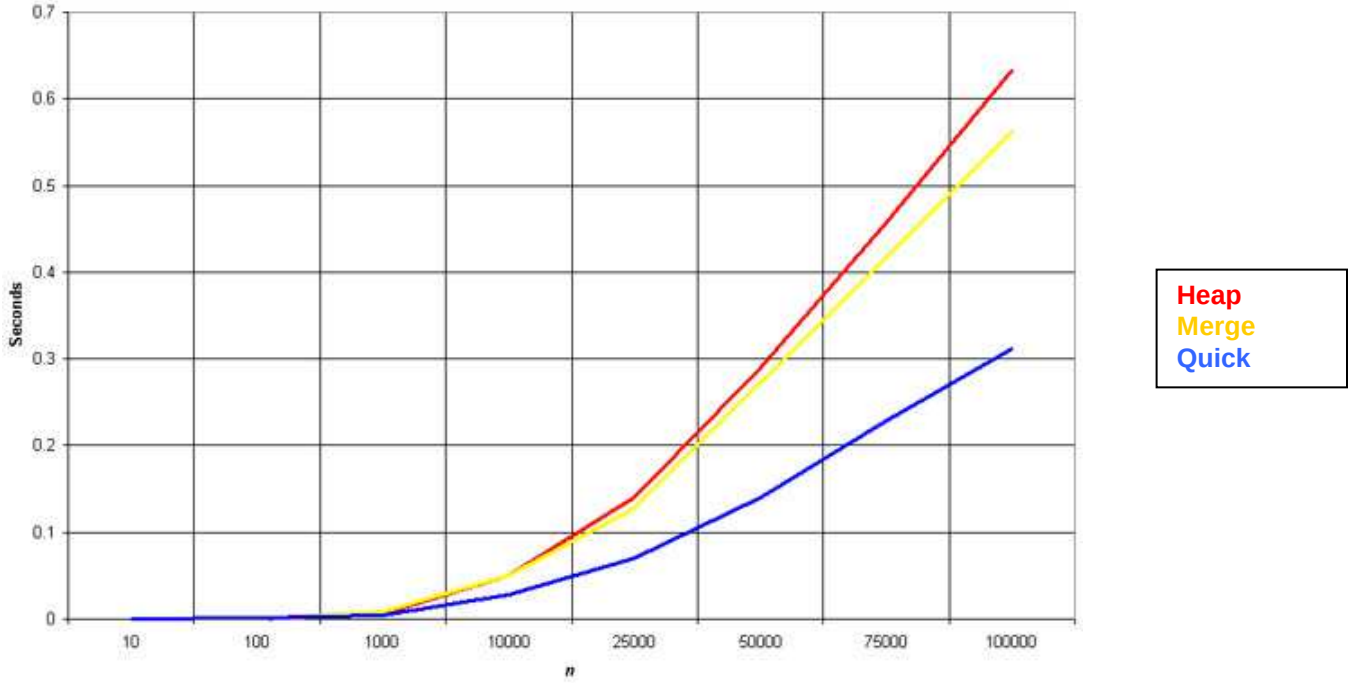


As the graph pretty plainly shows, the bubble sort is grossly inefficient, and the shell sort is the most efficient. Notice that the first horizontal line in the plot area is 100 seconds - these aren't sorts that you want to use for huge amounts of data in an interactive application. Even using the shell sort, users are going to be twiddling their thumbs if you try to sort much more than 10,000 data items.

All of these algorithms are incredibly simple (with the possible exception of the shell sort). For quick test programs, rapid prototypes, or internal-use software they're not bad choices unless you really think you need split-second efficiency.

Sorting Algorithms

O(*n* log n) Sorts



The O(*n* log n) sorts offer improved performance. Notice that the time on this graph is measured in *tenths* of seconds, instead *hundreds* of seconds like the $O(n^2)$ graph. These algorithms are fast, but that speed comes at the cost of complexity. These algorithms make extensive use of recursion, advanced data structures, and multiple arrays.

Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

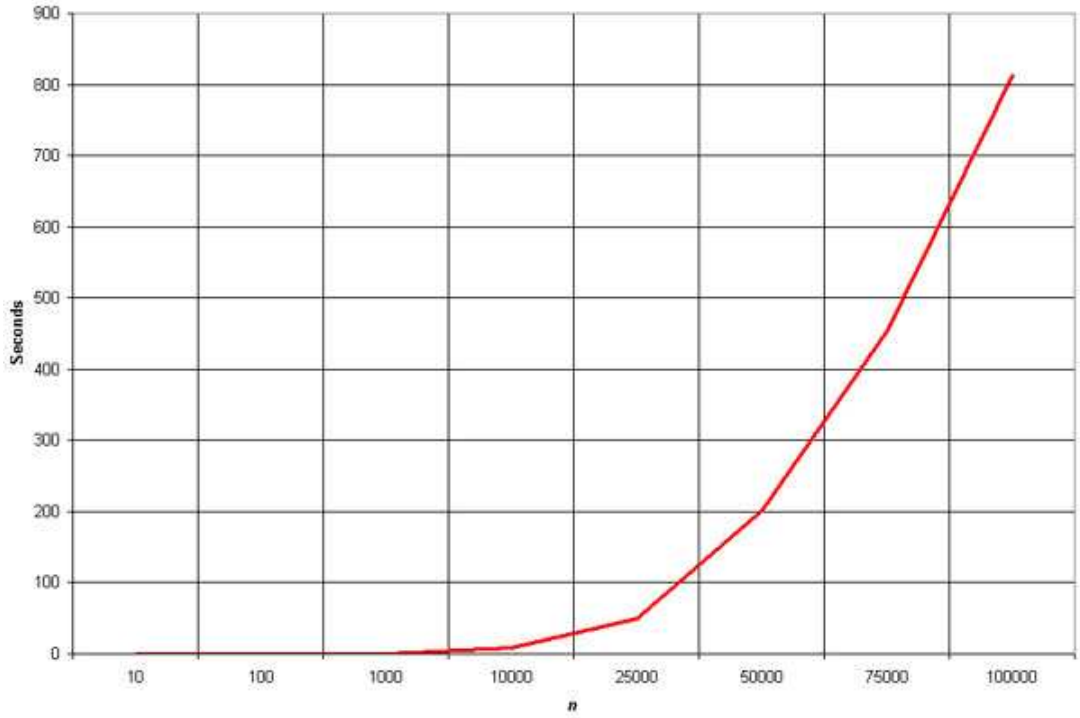
The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is $O(n^2)$.

While the insertion, selection, and shell sorts also have $O(n^2)$ complexities, they are significantly more efficient than the bubble sort.

Pros: Simplicity and ease of implementation.
Cons: Horribly inefficient.

Bubble Sort (cont.)

Bubble Sort Efficiency



The graph clearly shows the n^2 nature of the bubble sort.

Algorithm purists claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a few hundred items.

Bubble Sort - Source Code Example

Below is the basic bubble sort algorithm.

```
private static void bubbleSort(int numbers[])
{
    int i, j, temp;

    for (i = (numbers.length - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (numbers[j - 1] > numbers[j])
            {
                temp = numbers[j - 1];
                numbers[j - 1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
```

Performance:

Time to sort 1000 elements: 10 (ms)
Time to sort 10000 elements: 552 (ms)
Time to sort 100000 elements: 45890 (ms)

Heap Sort

The heap sort is the slowest of the O(*n* log n) sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.

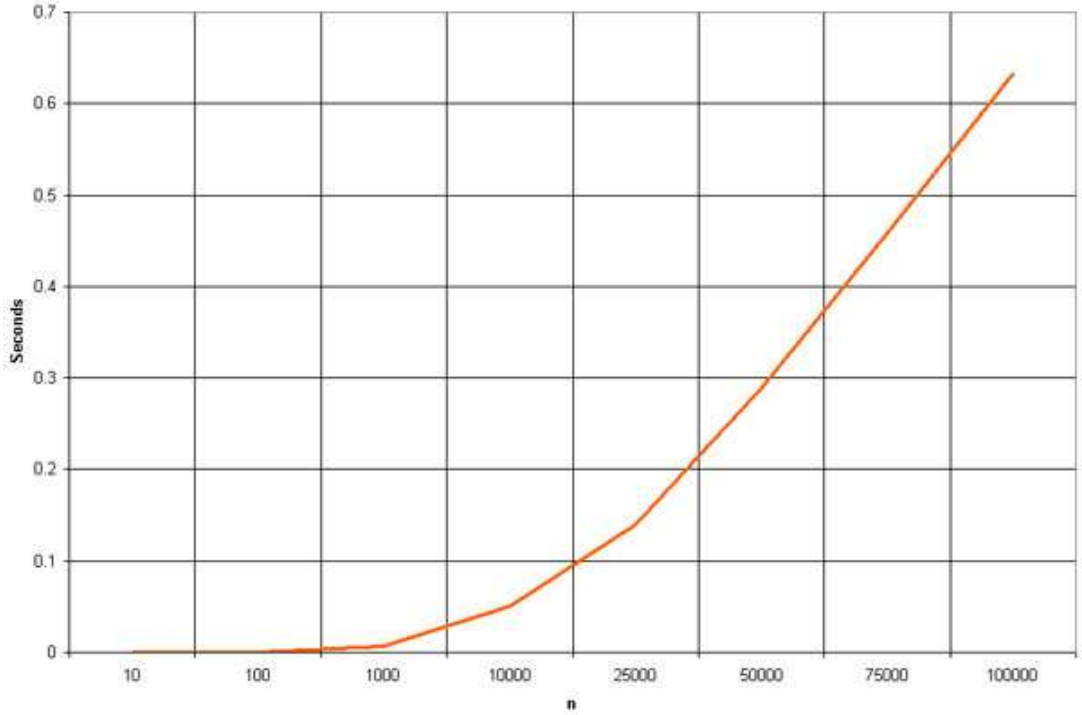
The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

To do an in-place sort and save the space the second array would require, the algorithm below “cheats” by using the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

Pros: In-place and non-recursive, making it a good choice for extremely large data sets.
Cons: Slower than the merge and quick sorts.

Heap Sort (cont.)

Heap Sort Efficiency



As mentioned, the heap sort is slower than the merge and quick sorts but doesn't use multiple arrays or massive recursion like they do. This makes it a good choice for really large sets, but most modern computers have enough memory and processing power to handle the faster sorts unless over a million items are being sorted.

The "million item rule" is just a rule of thumb for common applications - high-end servers and workstations can probably safely handle sorting tens of millions of items with the quick or merge sorts.

Heap Sort - Source Code Example

Below is the basic heap sort algorithm. The *siftDown()* function builds and reconstructs the heap.

```
static void heapSort(int numbers[])
{
  int i, temp;
  int arraySize = numbers.length-1;

  for (i = (arraySize / 2)-1; i >= 0; i--)
    siftDown(numbers, i, arraySize);

  for (i = arraySize-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}
```

Performance:

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 0 (ms)
Time to sort 100000 elements: 60 (ms)
Time to sort 1000000 elements: 512 (ms)
Time to sort 10000000 elements: 9778 (ms)

```
static void siftDown(int numbers[], int root, int bottom)
{
  int maxChild, temp;
  boolean done = false;

  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = true;
  }
}
```

Insertion Sort

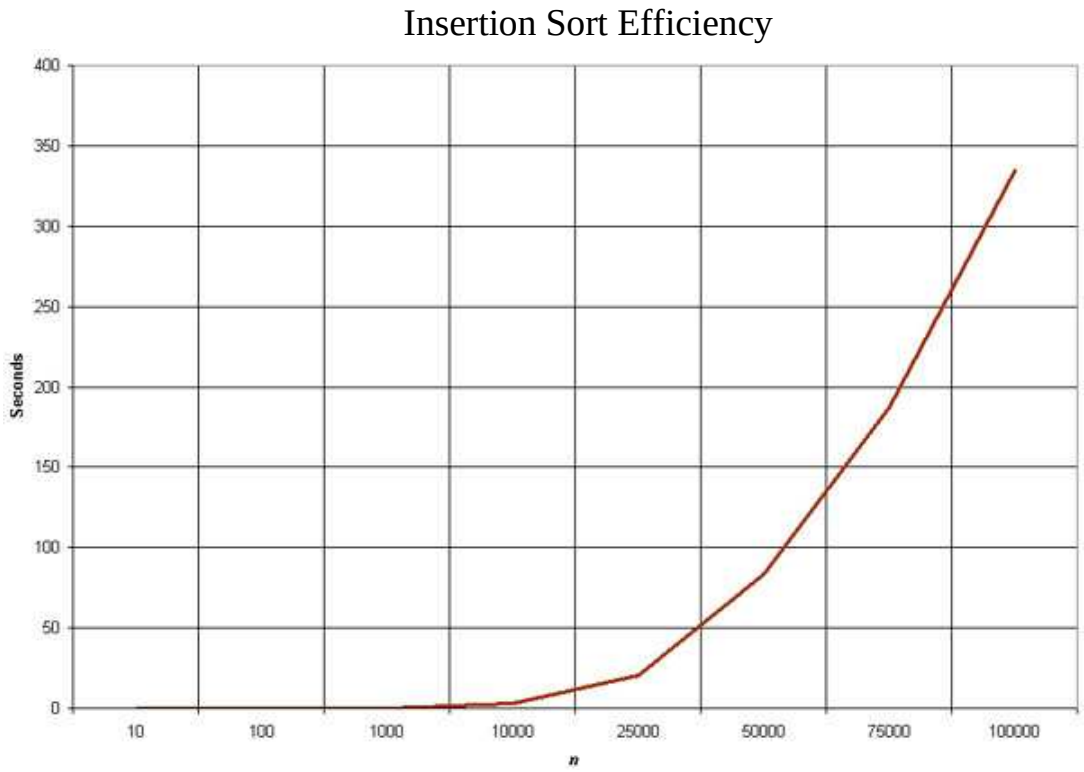
The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list.

The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of $O(n^2)$. Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.

Pros: Relatively simple and easy to implement.
Cons: Inefficient for large lists.

Insertion Sort (cont.)



The graph demonstrates the n^2 complexity of the insertion sort.

The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Insertion Sort - Source Code Example

Below is the basic insertion sort algorithm.

```
static void insertionSort(int numbers[])
{
  int i, j, index;

  for (i=1; i < numbers.length; i++)
  {
    index = numbers[i];
    j = i;
    while ((j > 0) && (numbers[j-1] > index))
    {
      numbers[j] = numbers[j-1];
      j = j - 1;
    }
    numbers[j] = index;
  }
}
```

Performance:

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 191 (ms)
Time to sort 100000 elements: 12790 (ms)

Merge Sort

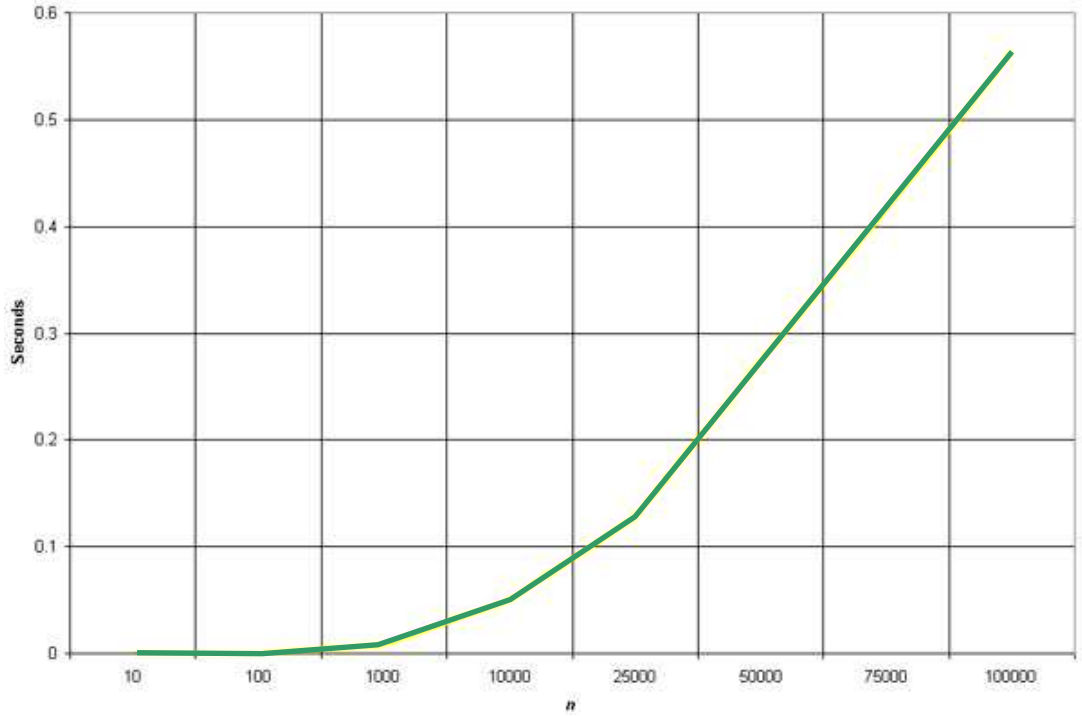
The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of O(*n* log n).

Elementary implementations of the merge sort make use of three arrays - one for each half of the data set and one to store the sorted list in. The below algorithm merges the arrays in-place, so only two arrays are required. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

Pros: Marginally faster than the heap sort for larger sets.
Cons: At least twice the memory requirements of the other sorts; recursive.

Merge Sort (cont.)

Merge Sort Efficiency



The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. This additional memory requirement makes it unattractive for most purposes - the quick sort is a better choice most of the time and the heap sort is a better choice for very large sets.

Like the quick sort, the merge sort is recursive, which can make it a bad choice for applications that run on machines with limited memory.

Merge Sort - Source Code Example

Below is the basic merge sort algorithm.

```
static void mergeSort(int numbers[])
{
  int[] temp = new int[numbers.length];
  m_sort(numbers, temp, 0, numbers.length - 1);
}

static void m_sort(int numbers[], int temp[], int left, int right)
{
  int mid;

  if (right > left)
  {
    mid = (right + left) / 2;
    m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, mid+1, right);

    merge(numbers, temp, left, mid+1, right);
  }
}

static void merge(int numbers[], int temp[], int left,
                                          int mid, int right)
{
  int i, left_end, num_elements, tmp_pos;

  left_end = mid - 1;
  tmp_pos = left;
  num_elements = right - left + 1;

  while ((left <= left_end) && (mid <= right))
  {
    if (numbers[left] <= numbers[mid])
    {
      temp[tmp_pos] = numbers[left];
      tmp_pos = tmp_pos + 1;
      left = left +1;
    }
    else
    {
      temp[tmp_pos] = numbers[mid];
      tmp_pos = tmp_pos + 1;
      mid = mid + 1;
    }
  }

  while (left <= left_end)
  {
    temp[tmp_pos] = numbers[left];
    left = left + 1;
    tmp_pos = tmp_pos + 1;
  }
  while (mid <= right)
  {
    temp[tmp_pos] = numbers[mid];
    mid = mid + 1;
    tmp_pos = tmp_pos + 1;
  }

  for (i=0; i < num_elements; i++)
  {
    numbers[right] = temp[right];
    right = right - 1;
  }
}
```

Performance

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 0 (ms)
Time to sort 100000 elements: 50 (ms)
Time to sort 1000000 elements: 411 (ms)
Time to sort 10000000 elements: 4618 (ms)

Quick Sort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps (which closely resemble the merge sort):

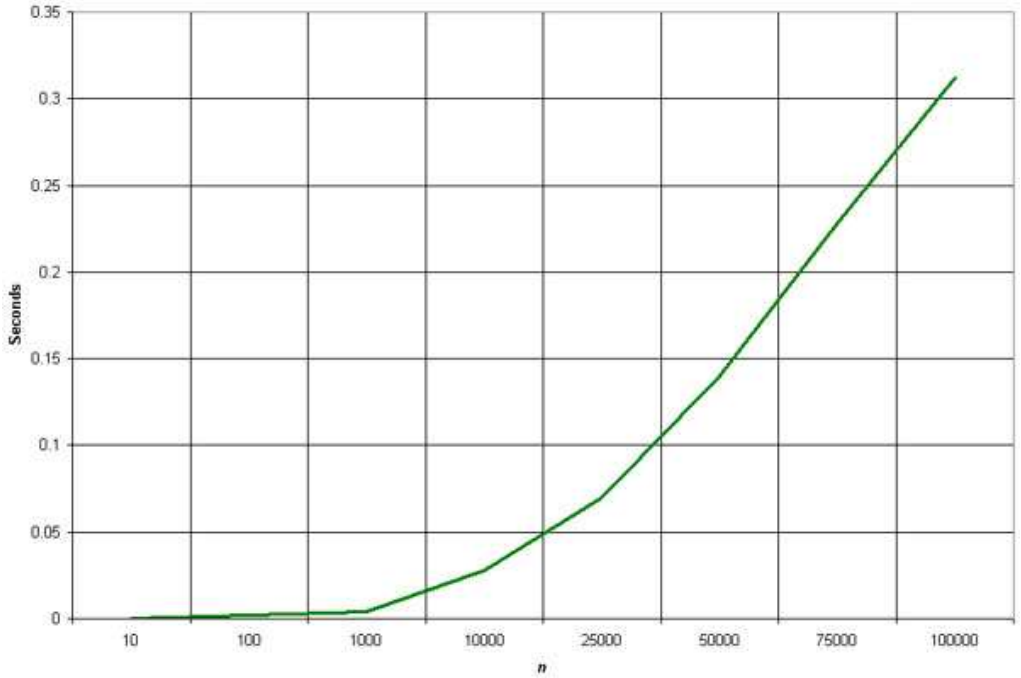
1. If there are one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

The efficiency of the algorithm is majorly impacted by which element is choosen as the pivot point. The worst-case efficiency of the quick sort, O(n2), occurs when the list is sorted and the left-most element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O(n log n).

Pros: Extremely fast.
Cons: Very complex algorithm, massively recursive.

Quick Sort (cont.)

Quick Sort Efficiency



The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster.

As mentioned earlier, it is massively recursive (which means that for very large sorts, you can run the system out of stack space pretty easily). It's also a complex algorithm - a little too complex to make it practical for a one-time sort of 25 items, for example.

In most cases the quick sort is the best choice if speed is important. Use it for repetitive sorting, sorting of medium to large lists, and as a default choice when you're not really sure which sorting algorithm to use. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order - avoid it in those situations.

Quick Sort - Source Code Example

Below is the basic quick sort algorithm.

```
static void quickSort(int numbers[])
{
  q_sort(numbers, 0, numbers.length - 1);
}


static void q_sort(int numbers[], int left, int right)
{
  int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];

  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;

    if (left != right)
    {
      numbers[left] = numbers[right];
      left++;
    }

    while ((numbers[left] <= pivot) && (left < right))
      left++;

    if (left != right)
    {
      numbers[right] = numbers[left];
      right--;
    }
  }

  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;

  if (left < pivot)
    q_sort(numbers, left, pivot-1);

  if (right > pivot)
    q_sort(numbers, pivot+1, right);

};
```

Performance

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 0 (ms)
Time to sort 100000 elements: 30 (ms)
Time to sort 1000000 elements: 271 (ms)
Time to sort 10000000 elements: 2379 (ms)

Selection Sort

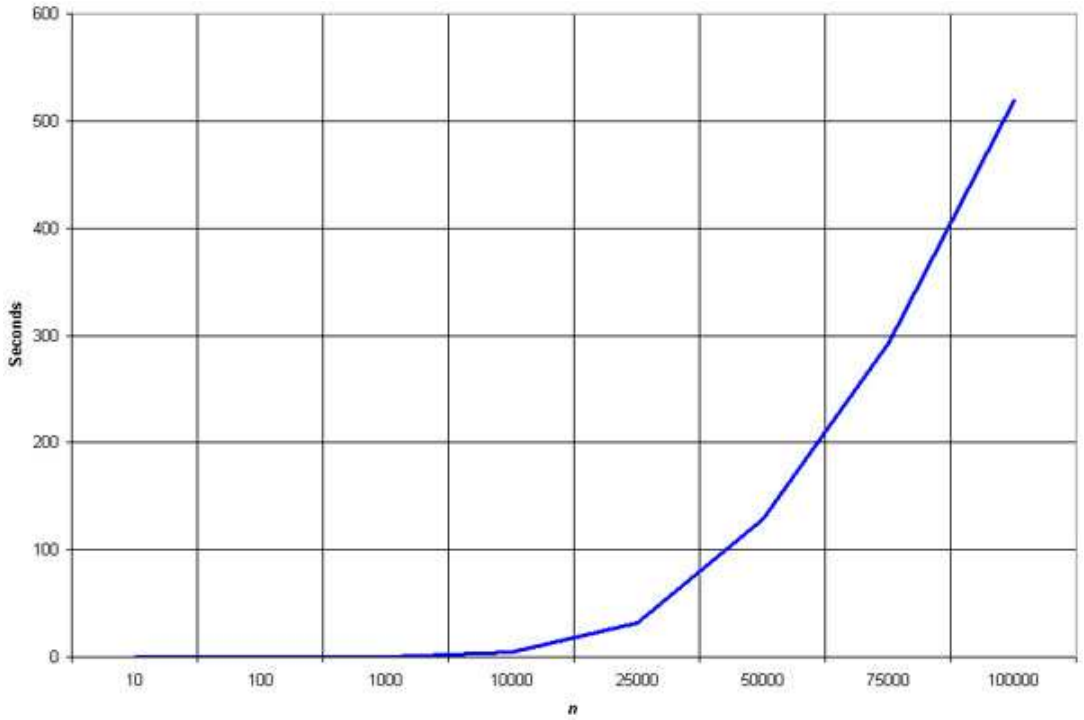
The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.

The selection sort has a complexity of $O(n^2)$.

Pros: Simple and easy to implement.
Cons: Inefficient for large lists, so similar to the more efficient insertion sort that the insertion sort should be used in its place.

Selection Sort (cont.)

Selection Sort Efficiency



Selection sort yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

Selection Sort - Source Code Example

Below is the basic selection sort algorithm.

```
static void selectionSort(int numbers[])
{
  int i, j;
  int min, temp;

  for (i = 0; i < numbers.length-1; i++)
  {
    min = i;
    for (j = i+1; j < numbers.length; j++)
    {
      if (numbers[j] < numbers[min])
        min = j;
    }
    temp = numbers[i];
    numbers[i] = numbers[min];
    numbers[min] = temp;
  }
}
```

Performance

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 271 (ms)
Time to sort 100000 elements: 21273 (ms)

Shell Sort

Invented by Donald Shell in 1959, the shell sort is the most efficient of the $O(n^2)$ class of sorting algorithms. Of course, the shell sort is also the most complex of the $O(n^2)$ algorithms.

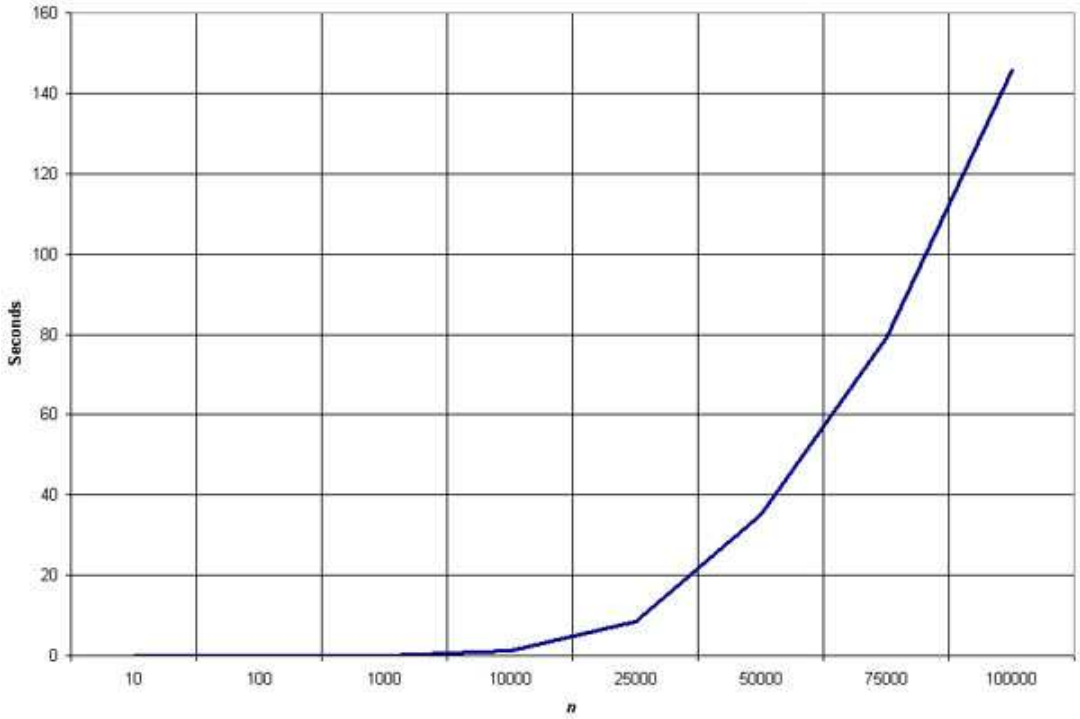
The shell sort is a "diminishing increment sort", better known as a "comb sort" to the unwashed programming masses. The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort. The size of the set to be sorted gets larger with each pass through the list, until the set consists of the entire list. (Note that as the size of the set increases, the number of sets to be sorted decreases.) This sets the insertion sort up for an almost-best case run each iteration with a complexity that approaches O(n).

The items contained in each set are not contiguous - rather, if there are *i* sets then a set is composed of every *i-th* element. For example, if there are 3 sets then the first set would contain the elements located at positions 1, 4, 7 and so on. The second set would contain the elements located at positions 2, 5, 8, and so on; while the third set would contain the items located at positions 3, 6, 9, and so on.

Pros: Efficient for medium-size lists.
Cons: Somewhat complex algorithm, not nearly as efficient as the merge, heap, and quick sorts.

Shell Sort (cont.)

Shell Sort Efficiency



The shell sort is by far the fastest of the N^2 class of sorting algorithms. It's more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is hyper-critical. It's also an excellent choice for repetitive sorting of smaller lists.

Shell Sort - Source Code Example

Below is the basic shell sort algorithm.

```
static void shellSort(int numbers[])
{
  int i, j, increment, temp;

  increment = 3;
  while (increment > 0)
  {
    for (i=0; i < numbers.length; i++)
    {
      j = i;
      temp = numbers[i];
      while ((j >= increment) && (numbers[j-increment] > temp))
      {
        numbers[j] = numbers[j - increment];
        j = j - increment;
      }
      numbers[j] = temp;
    }
    if (increment/2 != 0)
      increment = increment/2;
    else if (increment == 1)
      increment = 0;
    else
      increment = 1;
  }
}
```

Performance

Time to sort 1000 elements: 10 (ms)
Time to sort 10000 elements: 50 (ms)
Time to sort 100000 elements: 3785 (ms)

Sorting Java Collections

The *Collections* class in Java provides a *sort()* method that allows a (modifiable) list to be sorted. The following code sample shows how a List can be created from an array, and then sorted using the *Collections.sort()* method:

```
String[] data = {"Kiwi","Banana","Mango","Aubergine","Strawberry"};

ArrayList<String> list = new ArrayList<String>(Arrays.asList(data));
Collections.sort(list);
System.out.println(list);
```

Displays: [Aubergine, Banana, Kiwi, Mango, Strawberry]

The first line creates an in-line array of Objects that can be used to sort strings. The Arrays utility class then converts the array into a List. This is an efficient way to create a List, which can then sort data.

Once we have the array, we then use the Collections.sort() method. It modifies the target list and arranges the elements in their natural order. The resulting list then displays to the console.

Sorting Java Collections (cont.)

So how do the generic sorting algorithms in Java compare elements in a type-independent way? The algorithms rely on the *Comparable* interface, which provides a single method, *compareTo()*:

```
public int compareTo(Object obj)
```

Which returns:

- A negative integer if this is less than obj
- Zero if this and obj are equivalent
- A positive integer if this is greater than obj

Classes that implement the Comparable interface can be compared with one another; this allows the sorting algorithms to arrange such elements in a list. As with the *equals()* method, you should only compare like-typed classes, otherwise a ClassCastException may be thrown.

The standard Java data types (e.g., String, Integer) all implement the Comparable interface to provide a natural sorting order.

Sorting Java Collections (cont.)

To enable sorting, the String class implements the Comparable interface. To compare Strings in a language-independent way, the String class implements the compareTo() method to provide a lexicographic ordering between strings. In other words, the strings are compared character by character, and the Unicode values determine whether the two strings are different:

```
"Aubergine".compareTo("Banana")    < 0
  "Banana".compareTo("Aubergine") > 0
"Aubergine".compareTo("Aubergine") == 0
```

But beware—you might not always get exactly what you expect using natural ordering. The character-by-character comparison implementation is case sensitive and behaves oddly for accented characters.

Note, using Java's build-in *Collections.sort()* sorting algorithm with the same integer data set used with the sorting algorithm examples earlier performed as follows when a:

Time to sort 1000 elements: 0 (ms)
Time to sort 10000 elements: 10 (ms)
Time to sort 100000 elements: 80 (ms)
Time to sort 1000000 elements: 1288 (ms)
Time to sort 10000000 elements: 18490 (ms)

"Comparable" Interface

The *Comparable* interface (java.util) provides a natural (i.e., default) sorting order for a class. We use an example *Date* class to demonstrate how sort order works:

```
public class Date implements Comparable {
  private int year;
  private int month;
  private int day;
  public Date(int yearIn, int monthIn, int dayIn) {
    year = yearIn;
    month = monthIn;
    day = dayIn;
  }
  public int getYear()  { return year; }
  public int getMonth() { return month; }
  public int getDay()   { return day; }
  public String toString() {
    return year + "-" + month + "-" + day;
  }
  public int compareTo(Object o) throws ClassCastException {
    Date d = (Date)o; // If this doesn't work, ClassCastException is thrown
    int yd = year  - d.year;
    int md = month - d.month;
    int dd = day   - d.day;
    if (yd != 0) return yd;
    else if (md != 0) return md;
    else return dd;
  }
}
```

This class defines a data structure that contains three integers: year, month, and day. In the *compareTo()* method, we calculate the difference in year, month, and day. If the years are not the same (yd!=0), then the method returns the difference in years. It will be negative if o < this, or positive otherwise. The same happens with the month. If the months are the same, then the method returns the difference in days.

Note: The return value does not have to be -1, 0, or 1. Only the sign is important, not the value. In this case, the code just returns the difference in years, without worrying about the magnitude.

Alternate Sort Ordering – The Comparator Interface

While the Comparable interface allows natural sorting order for a class, it is often desirable to sort data in a different order (such as reverse sorting or case-insensitive sorting). For example, sorting the list of Strings [Aubergine, banana, aubergine, Banana] results in [Aubergine, Banana, aubergine, banana], because the natural sorting order is a character-by-character comparison.

Although the String's natural sorting order can't change, you can define an external sort on an existing class using the *Comparator* interface (java.util).

The Comparator interface defines the `public int compare(Object o1, Object o2)` method that returns:

- A negative integer if o1 is less than o2
- Zero if o1 and o2 are considered equivalent
- A positive integer if o1 is greater than o2

To define a case-insensitive sorting operation for strings, we define the following class:

```
public static class CaseInsensitiveComparator implements Comparator<String>
{
  public int compare(String o1, String o2)
  {
    String s1 = o1.toString().toUpperCase();
    String s2 = o2.toString().toUpperCase();
    return s1.compareTo(s2);
  }
}
```

Alternate Sort Ordering – The Comparator Interface (cont.)

To use the comparator, we pass an instance as the second argument to the *Collections.sort()*:

```
String[] data = {"Aubergine","banana","aubergine","Banana"};

ArrayList<String> list =  new ArrayList<String>(Arrays.asList(data));
Collections.sort(list, new CaseInsensitiveComparator());
System.out.println(list);
```

Displays: [Aubergine, aubergine, banana, Banana]

The only difference between this code and the previous *Collections.sort()* code is the second argument to *Collections.sort()*.

In this case, an instance of *CaseInsensitiveComparator()* is passed, which allows the comparison of List's elements using CaseInsensitiveComparator() instead of the natural ordering provided by the String class's Comparable implementation.

Note: Because “Aubergine” and “aubergine” are considered equivalent (as are “banana” and “Banana”), they remain in the same relative order as they were before the sort. Sorting algorithms that preserve the order for otherwise equal elements are stable, and the one implemented by the Collections class is stable. If you want capitalized words ahead of their lower-case counterparts, two sorting operations are required: a case-sensitive sort and a case-insensitive sort.

Also note: Sorting a collection of assorted classes might cause problems. Often, a Comparator may generate a ClassCastException when trying to compare two incompatible types.

Multiple Sort Orderings

Often, a set of objects may need to be sorted by varying fields (title, date, age, etc.). In this situation, multiple Comparator classes can be created (one to sort each of the "sortable" fields). To sort on a given field, one must simply pass the Comparator that is associated with the field. The following class contains multiple fields that can be sorted:

```
public class Movie2
{
    private String title;
    private int year;
    private String director;
    private int minutes;

    public Movie2(String titleIn, int yearIn, String directorIn, int minutesIn)
    {
        setTitle(titleIn);
        setYear(yearIn);
        setDirector(directorIn);
        setMinutes(minutesIn);
    }

    public String toString()
    {
        return String.format("%-30s %-10s %-20s %-10s",
           getTitle(), getYear(), getDirector(), getMinutes());
    }

      […]

}
```

Multiple Sort Orderings (cont.)

The following Comparator classes will address sorting by any of the available fields (title, year, director or minutes):

```
public class TitleComparator implements Comparator<Movie2>
{
    public int compare(Movie2 o, Movie2 o1)
    {
        return o.getTitle().compareTo(o1.getTitle());
    }
}


public class YearComparator implements Comparator<Movie2>
{
    public int compare(Movie2 m1, Movie2 m2)
    {
        return m1.getYear() - m2.getYear();
    }
}


public class DirectorComparator implements Comparator<Movie2>
{
    public int compare(Movie2 m1, Movie2 m2)
    {
        return m1.getDirector().compareTo(m2.getDirector());
    }
}


public class MinutesComparator implements Comparator<Movie2>
{
    public int compare(Movie2 m1, Movie2 m2)
    {
        return m1.getMinutes() - m2.getMinutes();
    }
}
```

Multiple Sort Orderings (cont.)

The following movies data will be used for sorting example purposes. Assume that the movies are initially stored in an ArrayList<Movie> called "movies" in the order shown:

	Title	Year	Director	Minutes
1.	Mummy, The	1999	Sommers, Stephen	124
2.	Run Lola Run	1998	Tykwer, Tom	80
3.	Big Fish	2003	Burton, Tim	125
4.	To Have and Have Not	1944	Hawks, Howard	100
5.	Boiler Room	2000	Younger, Ben	118
6.	Man Who Would Be King, The	1975	Huston, John	129
7.	Young Frankenstein	1974	Brooks, Mel	106
8.	Zombies on Broadway	1945	Douglas, Gordon	69
9.	Young and Innocent	1937	Hitchcock, Alfred	83
10.	2001: A Space Odyssey	1968	Kubrick, Stanley	160
11.	Big Sleep, The	1946	Hawks, Howard	114
12.	Big Sleep, The	1978	Winner, Michael	99

To sort this ArrayList of movies by "Title", the *TitleComparator* class should be used:

```
Collections.sort(movies, new TitleComparator());
```

Multiple Sort Orderings (cont.)

The resulting order after executing Collections.sort(movies, new TitleComparator()) is:

```
     Title                          Year       Director             Minutes
     -----                          ----       --------             -------
1.   2001: A Space Odyssey          1968       Kubrick, Stanley     160
2.   Big Fish                       2003       Burton, Tim          125
3.   Big Sleep, The                 1946       Hawks, Howard        114
4.   Big Sleep, The                 1978       Winner, Michael      99
5.   Boiler Room                    2000       Younger, Ben         118
6.   Man Who Would Be King, The     1975       Huston, John         129
7.   Mummy, The                     1999       Sommers, Stephen     124
8.   Run Lola Run                   1998       Tykwer, Tom          80
9.   To Have and Have Not           1944       Hawks, Howard        100
10.  Young Frankenstein             1974       Brooks, Mel          106
11.  Young and Innocent             1937       Hitchcock, Alfred    83
12.  Zombies on Broadway            1945       Douglas, Gordon      69
```

To sort this ArrayList of movies by “Year”, the *YearComparator* class should be used:

```
Collections.sort(movies, new YearComparator());
```

Multiple Sort Orderings (cont.)

The resulting order after executing Collections.sort(movies, new YearComparator()) is:

	Title	Year	Director	Minutes
1.	Young and Innocent	1937	Hitchcock, Alfred	83
2.	To Have and Have Not	1944	Hawks, Howard	100
3.	Zombies on Broadway	1945	Douglas, Gordon	69
4.	Big Sleep, The	1946	Hawks, Howard	114
5.	2001: A Space Odyssey	1968	Kubrick, Stanley	160
6.	Young Frankenstein	1974	Brooks, Mel	106
7.	Man Who Would Be King, The	1975	Huston, John	129
8.	Big Sleep, The	1978	Winner, Michael	99
9.	Run Lola Run	1998	Tykwer, Tom	80
10.	Mummy, The	1999	Sommers, Stephen	124
11.	Boiler Room	2000	Younger, Ben	118
12.	Big Fish	2003	Burton, Tim	125

To sort this ArrayList of movies by “Director”, the *DirectorComparator* class should be used:

```
Collections.sort(movies, new DirectorComparator());
```

Multiple Sort Orderings (cont.)

The resulting order after executing Collections.sort(movies, new DirectorComparator()) is:

	Title	Year	Director	Minutes
1.	Young Frankenstein	1974	Brooks, Mel	106
2.	Big Fish	2003	Burton, Tim	125
3.	Zombies on Broadway	1945	Douglas, Gordon	69
4.	To Have and Have Not	1944	Hawks, Howard	100
5.	Big Sleep, The	1946	Hawks, Howard	114
6.	Young and Innocent	1937	Hitchcock, Alfred	83
7.	Man Who Would Be King, The	1975	Huston, John	129
8.	2001: A Space Odyssey	1968	Kubrick, Stanley	160
9.	Mummy, The	1999	Sommers, Stephen	124
10.	Run Lola Run	1998	Tykwer, Tom	80
11.	Big Sleep, The	1978	Winner, Michael	99
12.	Boiler Room	2000	Younger, Ben	118

To sort this ArrayList of movies by “Minutes”, the *MinutesComparator* class should be used:

```
Collections.sort(movies, new MinutesComparator());
```

Multiple Sort Orderings (cont.)

The resulting order after executing Collections.sort(movies, new DirectorComparator()) is:

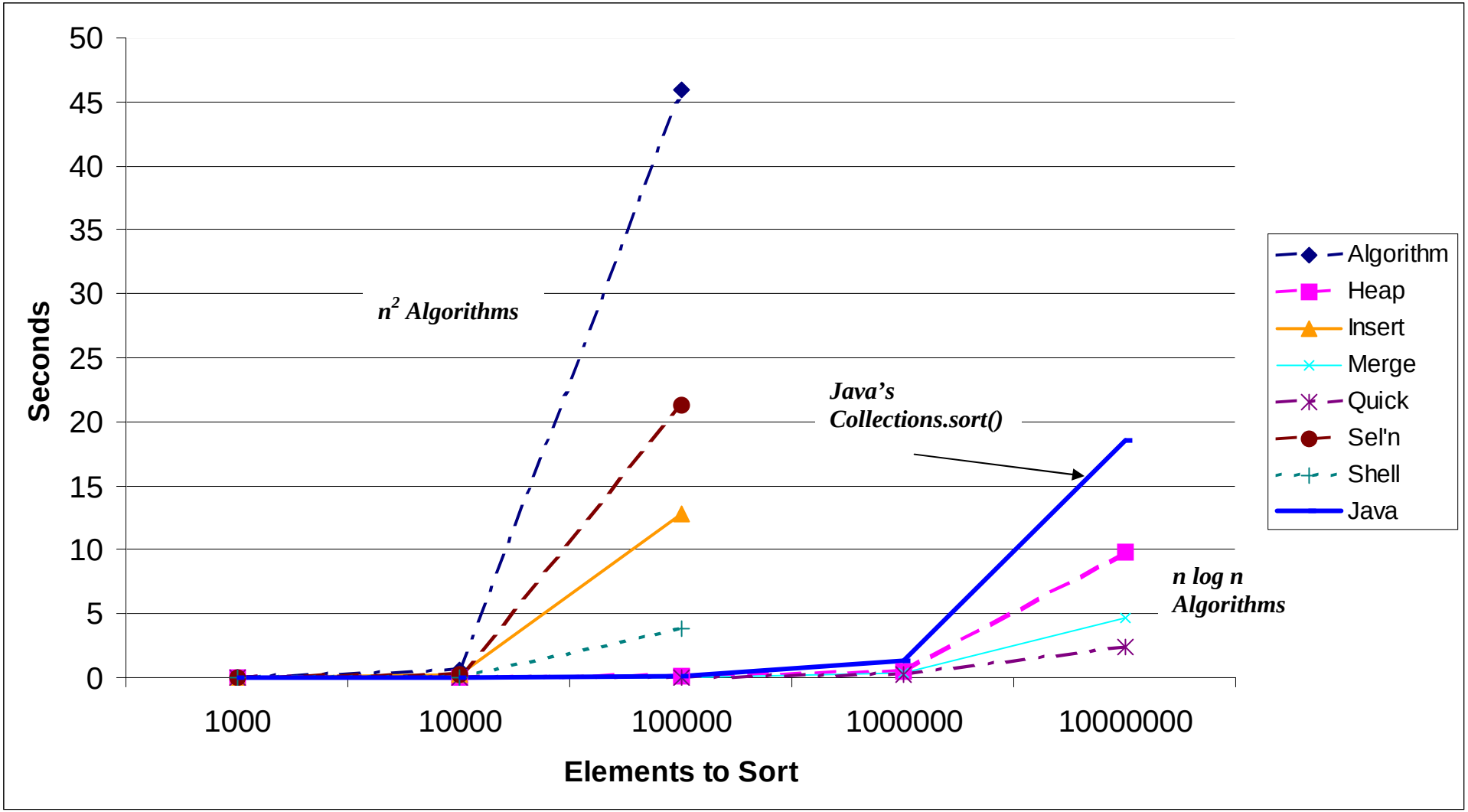
	Title	Year	Director	Minutes
1.	Zombies on Broadway	1945	Douglas, Gordon	69
2.	Run Lola Run	1998	Tykwer, Tom	80
3.	Young and Innocent	1937	Hitchcock, Alfred	83
4.	Big Sleep, The	1978	Winner, Michael	99
5.	To Have and Have Not	1944	Hawks, Howard	100
6.	Young Frankenstein	1974	Brooks, Mel	106
7.	Big Sleep, The	1946	Hawks, Howard	114
8.	Boiler Room	2000	Younger, Ben	118
9.	Mummy, The	1999	Sommers, Stephen	124
10.	Big Fish	2003	Burton, Tim	125
11.	Man Who Would Be King, The	1975	Huston, John	129
12.	2001: A Space Odyssey	1968	Kubrick, Stanley	160

Additional Comparators could be created to implement reverse sorting, joint-field sorting, etc.

Sort Overview

Name	Approximate Time Performance on a Data Set of Size *n*			Description
	Best Case	**Average Case**	**Worst Case**	
Insertion	*n*	n^2	n^2	A very intuitive yet somewhat slow algorithm. Each element in the collection is removed from its original place and inserted in the correct place. If two elements are equal, the original ordering will be maintained. Insertion sort is order n^2 in the worst and average cases, as *n-1* elements must be selected and re-inserted and each insertion requires up to *n-1* comparisons.
Shell	*n*	$n^{1.25}$	$n^{1.5}$	Although shell sort is order $n^{1.5}$, you may find it useful to know that its average time is $n^{1.25}$ and its optimal time is nearly *n*.
Quick Sort	*n log n*	*n log n*	n^2	Quick Sort is, on average, one of the fastest sorts. Its divide and conquer approach is extremely efficient for randomly sorted lists, but you may find it interesting that it exhibits very poor performance on sorted and nearly sorted lists.
Bubble	*n*	n^2	n^2	By far the slowest algorithm worth mentioning. Like insertion, selection, and shell sort, bubble sort is order n^2. Bubble Sort performs especially poorly on large data sets, while the complexity of sorted sets may be close to *n*.
Selection	n^2	n^2	n^2	Also a very slow n^2 algorithm, the Selection Sort is known to perform approximately 60% better than the bubble sort. However, the insertion sort still outperforms the selection sort by a notable margin. Expect the Selection Sort to perform especially poorly on large data sets.
Heap	*n log n*	*n log n*	*n log n*	Of the *n log n* algorithms listed here, heap sort is the slowest. The heap sort's performance diminishes in comparison to quick and merge sorts as the data set increases in size.
Merge	*n log n*	*n log n*	*n log n*	A popular algorithm for its performance. Detecting the comparative advantage of merge sort over heap sort is only discernable with large data sets.

Sorting Algorithm Performance & Java Collections Sorting



Summary

Why is sorting so important?
Having the items in order makes it much easier to find a given item. If we often have to look up items, it pays off to sort the whole collection first. Imagine using a dictionary or phone book where the entries don't appear in some (known) order.

The standard data structures provided in Java's collections classes allow data to be easily manipulated and sorted. If you need to define a natural ordering for your classes, you should implement the Comparable interface. However, if you want to define a new sort for an existing class (including ones for which you don't have source access), then you can implement a standalone Comparator.

The Java *Collections* sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a List and sorts it according to its elements' natural ordering.

The Java *Collections* sort operation uses a slightly optimized *merge* sort algorithm, which is fast and stable:

- Fast: It is guaranteed to run in n log(n) time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n log(n) performance.

- Stable: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.