Chapter 9-1, Lecture notes

Measuring the Efficiency of Algorithms

1. Comparing how efficient algorithms are is central to computer science.
   1.1 Getting the right algorithm for an application.
   1.2 Having responsive word processors, grocery checkout systems, ATMs, video games and life support systems would depend on efficient algorithms.

2. We have emphasized the human cost in software development and the importance of style and readability.
   2.1 However, we do the "analysis of algorithms" to figure out the efficiency of different algorithms – not programs.
   2.2 It is the primarily the "significant differences" of algorithms not the specific implementation of the algorithm.
   2.3 Typically it is the superior algorithm and not clever coding tricks that make the most difference.
   2.4 You should concentrate on those gross differences to prevent selecting an algorithm that would be only a few seconds more efficient but take too much time implementing.

3 Now you could code the algorithms in C++ and run them to get a time.
   3.1 However, there are three main difficulties doing that:
      3.1.1 How are the algorithms coded?
         3.1.1.1 If the first algorithm was coded better than the second algorithm then you could be measuring how well the implementation was done and NOT the efficiency of the algorithms.
      3.1.2 What computer should you use?
         3.1.2.1 If you run the programs on different computers, then one computer can be faster than the other.
         3.1.2.2 There can be other factors that can cause one algorithm to run faster on one computer and on another computer can actually slow down another algorithm.
      3.1.3 What data should the programs use?
         3.1.3.1 The most important part is to select the correct data.
         3.1.3.2 One set of data could favor one algorithm over the other.
         3.1.3.3 Like with comparing sequential search and binary search with sorted data.
         3.1.3.4 What if the search would find the smallest item?
         3.1.3.5 The sequential search would find the smallest item immediately while the binary search would take some time.
         3.1.3.6 However, the binary search is actually much more efficient over all.

   3.2 We have to count the number of operation it requires (can ignore some of the set-up operations).

3.3 A few examples would be

3.3.1 Traversal of a linked list like so:

```
Node *cur = head;                   // 1, assignment
while (cur != NULL)                 // n + 1 comparisons
{       cout << cur->item << endl;  // n writes
        cur = cur->next;            // n assignments
 } // while
```

3.3.1.1 So it needs n+1 assignments, n+1 comparisons, and n write operations.
3.3.1.2 If it takes, respectively, a, c, and w time units, the time would be roughly (n + 1) * (a + c) + n * w time units.

3.3.2 The Towers of Hanoi. It takes with n disks, it takes $2^n - 1$ moves – so if each move takes time m you would have: $(2^n - 1) * m$ time units.

3.3.3 Nested loops.
3.3.3.1 Look at the algorithm of:

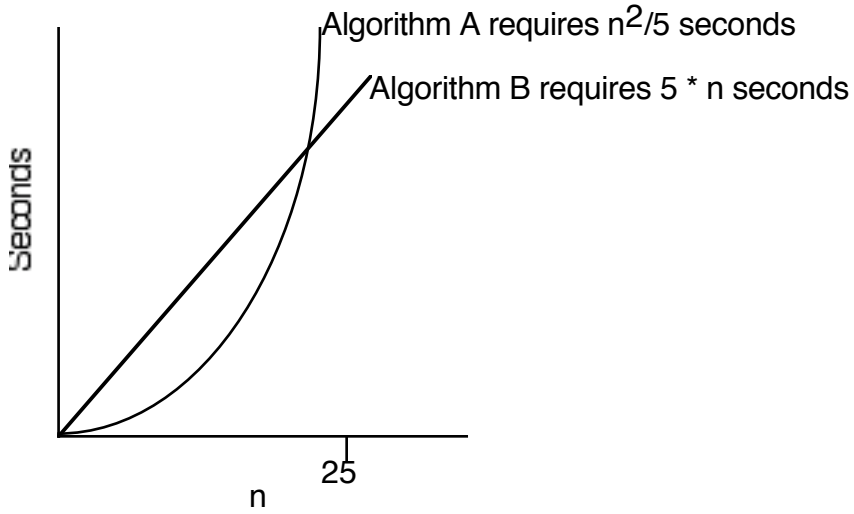
```
for (i = 1 through n)
        for (j = 1 through i)
                for (k = 1 through 5)
                        Task T
```

3.3.4 If task T requires t time units then the innermost loop on k takes 5 * t time units.
3.3.4.1 The loop on j requires 5 * t * I time units and the outermost loop on i requires:

$$\sum_{i=1}^{n} (5 * t * i) = 5 * t * (1 + 2 + \ldots + n) = 5 * t * n * (n + 1) / 2$$

4 The algorithm's time requirement is a function of the problem size.
4.1 For instance, if:
4.1.1 Algorithm A requires $n^2/5$ time units to solve a problem of size n
4.1.2 Algorithm B requires 5 * n time units to solve a problem of size n

4.2 An algorithm can do well with a small n but not with a larger n.
4.3 Also, an algorithm could do poorly with a small n but well with a larger n.
4.4 For instance, figure 9-1 would have:

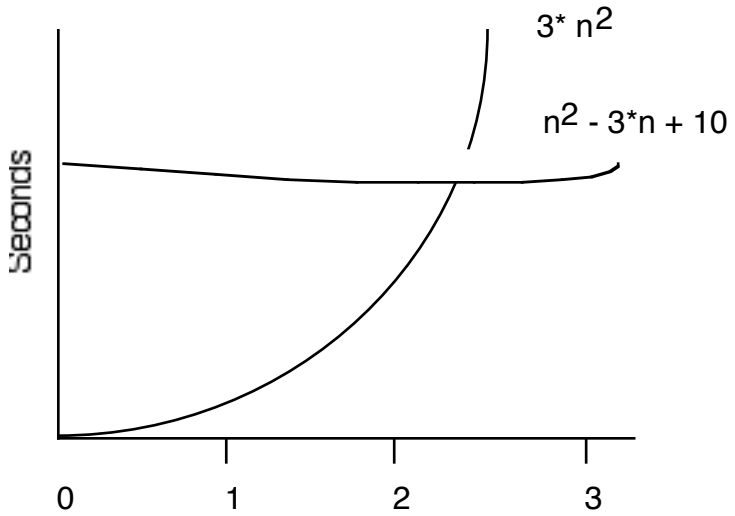


5 There is a simplification to figuring an algorithm's efficiency.
5.1 Any constants or even the lower order variables are not needed If:
Algorithm A requires time proportional to f(n)

5.2 Algorithm A has the "order f(n)" which is denoted as "O(f(n))".
5.2.1 The f(n) would be the algorithm's "growth-rate function".
5.2.2 And because the notation uses a capital "O" it is called "Big O notation".
5.2.3 If, for instance, the growth rate is n^2 then the problem is $O(n^2)$.

5.3 The formal definition is:
Definition of the Order of an Algorithm
Algorithm A is order f(n) – denoted O(f(n)) – if constants k and n_0 exist such that A requires no more than k * f(n) time units to solve a problems of size $n \geq n_0$.

6 You do need a sufficiently large problem for Big O to be valid.
6.1 The following examples are:
6.1.1 If an algorithm requires $n^2 - 3 * n + 10$ seconds to solve a problem of size n. And if constants of k and n_0 exist such that:
$k * n^2 > n^2 - 3 * n + 10$ for all $n \geq n_0$
6.1.1.1 The algorithm is order n^2. If fact, if k is 3 and n_0 is 2,
$3 * n^2 > n^2 - 3 * n + 10$ for all $n \geq 2$
6.1.1.2 as Figure 9-2 shows. Therefore the algorithm requires no more than k * n^2 time units for $n \geq n_0$ so is $O(n^2)$.



6.1.2 Previously we found that displaying a linked list's first n items requires (n + 1) * (a + c) + n * w time units. If $2 * n \geq n + 1$ for $n \geq 1$,
$(2 * a + 2 * c + w) * n \geq (n + 1) * (a + c) + n * w$ for $n \geq 1$

6.1.2.1 You get O(n). The k is 2 * a + 2 * c + w and n_0 is 1.

6.1.3 The Towers of Hanoi problem requires $(2^n - 1) * m$. But since:
$m * 2^n > (2^n - 1) * m$ for $n \geq 1$

6.1.3.1 the solution is $O(2^n)$.

7 The time estimate would be too small for a finite amount of problem sizes.

7.1 We can typically ignore the problem size of n is 1 since it take a program a measurable finite amount of time to do just 1 (and do it rather quickly).

7.2 When the problem gets sufficiently large the time estimate becomes valid.

7.3 In general, the Big O would be:

$O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

7.4 The growth rate functions have the following intuitive interpretations:

Big O	Interpretation
1	The time is constant and does not depend on n
$\log_2 n$	The time requirement for a logarithmic algorithm increases slowly as the problem size increases. So if you square the problem size you only double its time requirement.
n	The time requirement for a linear algorithm increases directly with the size of the problem. Square the problem, square the time requirement.
$n^* \log_2 n$	The time requirement for an $n^* \log_2 n$ algorithm increases more rapidly than a linear algorithm. Such algorithms usually divide a problem into smaller problems that are each solved separately.
n^2	The time requirement for a quadratic algorithm increases rapidly with the size of the problem. Two nested loops are often quadratic.
n^3	The time requirement for a cubic algorithm increases more rapidly with the size of the problem than the time requirements for a quadratic algorithm. Three nested loops are often cubic.
2^n	As the size of the problem increases, the time requirement for an exponential algorithm usually increases too rapidly to be practical.

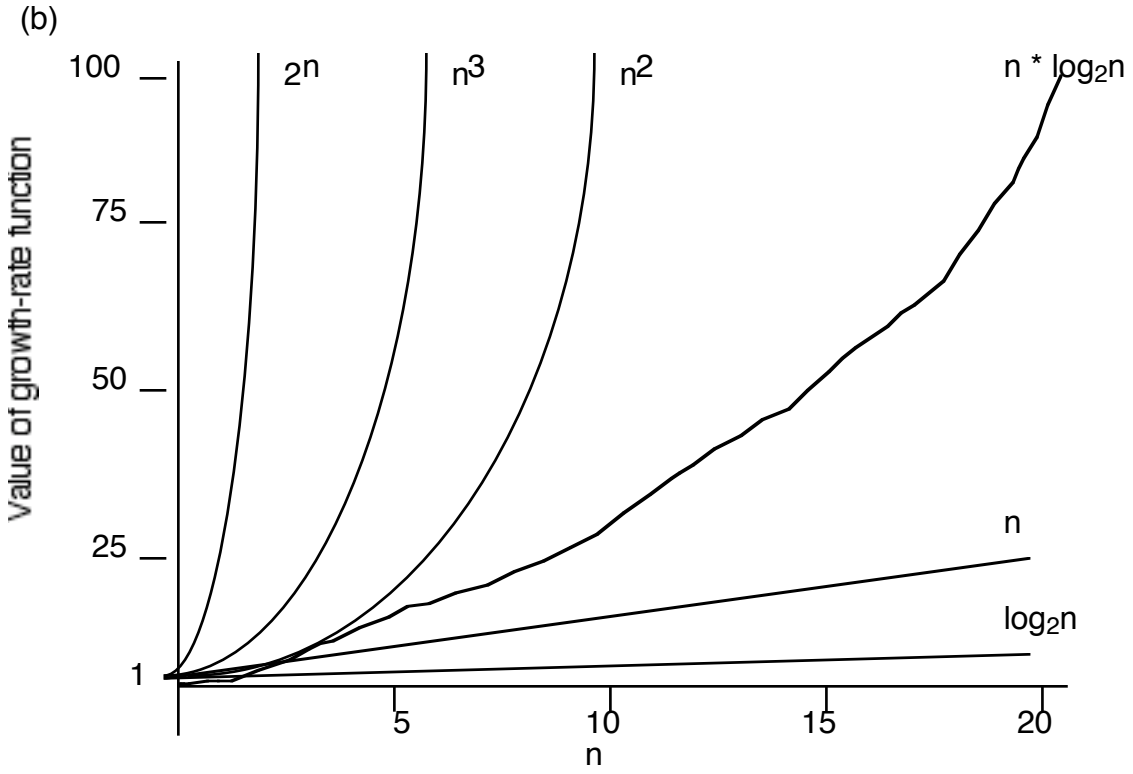
7.5 So if you have algorithm A that uses a particular Big O and algorithm B uses a slower growing Big O then algorithm B will always be more efficient than algorithm A.

7.6 Remember that O(f(n)) means "is order f(n)" or "has order f(n)". O is not a function.

7.7 The following figure 9-3 on page 453, has part a show the Big O with sample values and part b is Big O graphed.

(a)

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n^* \log_2 n$	30	644	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$



8 There are several properties of Big O that can simplify the analysis of the algorithm.

8.1 You can ignore low-order terms in an algorithm's growth-rate function.
- 8.1.1 If you have $O(n^3 + 4 * n^2 + 3 * n)$, it is also $O(n^3)$.
- 8.1.2 In figure 9-3, you can see that n^3 grows at a significantly greater rate than the other terms such as n^2 and n.

8.2 You can ignore a multiplicative constant in the high-order term of an algorithm's growth-rate functions. For example if $O(5 * n^3)$ then it is also $O(n^3)$.

8.3 $O(f(n)) + O(g(n)) = O(f(n) + g(n))$. So if you combine grow-rate functions. $O(n^2) + O(n)$ would be $O(n^2 + n)$. You can simplify to $O(n^2)$.

8.4 You need an estimate, not an exact time requirements of the algorithms.

9 The Big O would deal with the "worst-case analysis" or values that would cause the algorithm to take the longest time to complete.

9.1 It is pessimistic but you are assured that the algorithm will not run any worse than the worst-case scenario.

9.2 Most of the time the algorithm will run much faster.

9.3 The "average-case analysis" deals with data that the algorithm would most likely encounter and the average-time it would take to finish.

9.4 It is much more difficult to determine what data would be used.

9.5 You need to keep your perspective.

9.5.1 For instance, an array based sorted list would retrieve items quickly which a pointer based list would take n times to get at a node.
9.5.2 The array-based implementation would be significantly faster as n grows.
9.5.3 However, if you needed to insert and delete nodes frequently and don't need to retrieve that frequently then the pointer-based implementation would be better.

9.6 Also, picking an algorithm is never completely straightforward.
9.6.1 Many times memory can be used to gain speed.
9.6.2 An algorithm that used little memory could be slow.
9.6.3 However, an algorithm that used a lot memory can be much faster. It would depends on how much memory is available.

9.7 The sequential search would have the O(n) since it takes n/2 average time to search a list of n. The binary search would divide the n list in half each time and so would have a $O(\log_2 n)$. The binary search is much faster than the sequential search.

Chapter 9-2, Sorting Algorithms and Their Efficiency

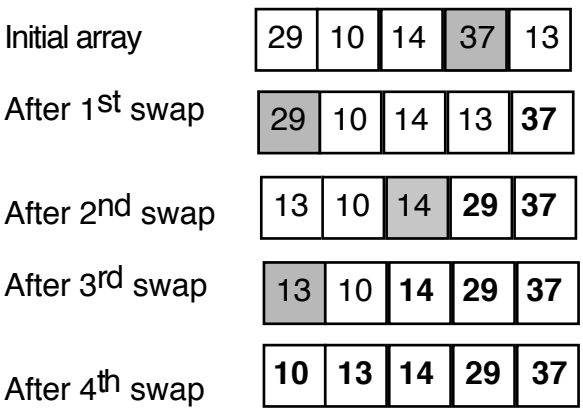
1 There are many times when you need to sort data into either ascending or descending order (or, for allowing duplicate values, you can have ascending as nondecreasing and descending as nonincreasing).
1.1 For the binary search to work efficiently, you need the data sorted.
1.2 There are other algorithms that need ordered data so a good sorting algorithm is valuable.

2 You can organize the sorting algorithms into two categories: "internal sort" that sorts the data that fits entirely into the computer's memory and "external sort" that sorts data that would not fit entirely into the computer's memory.
2.1 We will deal with internal sort immediately and leave the external sort to chapter 14.
2.2 For the sorts, we'll assume that the data is either numbers or characters, data is stored in an array and that the algorithm will sort into ascending order (it is simple to change it to descending order).

3 Imagine data that you can examine all at once.
3.1 To sort it, you select the largest item and put it in its place, select the next largest and put it in its place, and so on.
3.1.1 Selection sort does that (sort of).
3.1.2 It will search for the largest item and place it in the correct slot.
3.1.3 Then look for the next largest item and so on.
3.1.4 Actually, the selection sort will swap values (the largest value swaps places with the last spot of the array and so on).

3.2 Figure 9-4 shows how the selection sort works.

3.2.1 It does not need to swap 14 with itself but it does – easier to perform an unnecessary swap than to check whether or not to swap.



3.2.2 The Selection Sort code is:

```
typedef type-of-array-item DataType;

/** Sorts the items in an array into ascending order.
 *  @pre   theArray is an array of n items.
 *  @post  The array theArray is sorted into ascending order; n is unchanged.
 *  @parm theArray   The array to sort.
 *  @parm n    The size of theArray.  */
void selectionSort(DataType theArray[], int n)
{
        // last = index of the last item in the subarray of items
        //        yet to be sorted,
        // largest = index of the largest item found

        for (int last = n-1; last >= 1; --last)
        {       // Invariant: theArray[last+1 .. n-1] is sorted and > theArray[0..last]
                // select largest item in theArray[0..last]
                int largest = indexOfLargest(theArray, last+1);

                // swap largest item theArray[largest] with theArray[last]
                swap(theArray[largest], theArray[last]);
        } // end for
} // end selectionSort
```

3.3 The function selectionSort calls the following two functions:

```
/** Find the largest item in an array.
 * @pre      theArray is an array of size items, size >= 1.
 * @post     None.
 * @param theArray    The given array.
 * @param size  The last of element in theArray.
 * @return The index of the largest item in the array.
 *             The arguments are unchanged. */
int indexOfLargest(const DataType theArray[], int size)
{
        int indexSoFar = 0;     // index of largest item found so far

        for (int currentIndex = 1; currentItem <= size; ++currentIndex)
        {       // Invariant: theArray[indexSoFar] >= theArray[0..currentIndex-1]
                if (theArray[currentIndex] > theArray[indexSoFar])
                        indexSoFar = currentIndex;
        } // end for
        return indexSoFar;      // index of largest item
} // end IndexOfLargest

/** Swaps two items.
 * @pre      x and y are the items to be swapped.
 * @post    Contents of actual locations that x and y represent are swapped
 * @param x   Given data item.
 * @param y   Given data item. */
void swap(DataType& x, DataType& y)
{
        DataType temp = x;
        x = y;
        y = temp;
} // end swap
```

3.4 The selection sort has a loop in the main part that calls another loop in the indexOfLargest.

3.4.1 Also, going through the data, the main loop would call indexOfLargest with another 1 subtracted from n.

3.4.2 You would get an equation like so:

$$(n - 1) + (n - 2) + \ldots + 1 = n*(n - 1)/2$$

3.4.3 You would also have n – 1 calls to swap to result in n – 1 exchanges.

3.4.4 Each exchange needs three assignments or data moves. So:

$$3 * (n - 1)$$

3.4.5 All together you would get:

$$n * (n - 1)/2 + 3 * (n - 1) = n^2/2 + 5 * n/2 - 3$$

3.5 However, if you ignore the lower terms, you would have $O(n^2/2)$ and ignoring factors you would get $O(n^2)$ overall – which is the Big O for selection sort.

3.6 However, the data moves are only O(n) which makes it a good choice where data moves are costly but comparisons are not.

4 The Bubble Sort is a classic that many have seen.

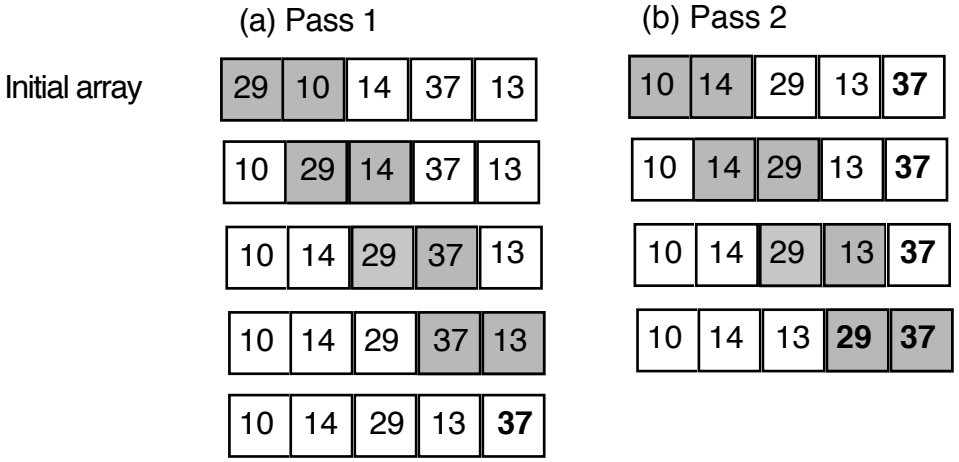
4.1 It has about the same Big O as selection sort and is simple – some claim too simple.

4.1.1 The bubble sort compares values next to each other and will swap them if need be.

4.1.2 The sort requires several passes over the data.

4.1.3 Fortunately, it, like the selection sort, tends to place the larger values toward the end of the array (i.e. it "bubbled" to the end) so we can speed it up a bit by decreasing n for each pass.

4.2 The bubble sort would operate on data like in figure 9-5:



4.2.1 The above is not sorted, of course.

4.2.2 The bubble will keep going until either the number of remaining items becomes 0 or until there is a pass through the array that does not swap values (i.e. it is sorted).

4.2.3 Below is the bubble sort code and it uses the previous swap function.

```
/** Sorts the items in an array into ascending order.
 * @pre    theArray is an array of n items.
 * @post  theArray is sorted into ascending order: n is unchanged
 * @param theArray     The given array
 * @param n    The size of theArray */
void bubbleSort(DataType theArray[], int n)
{
        bool sorted = false;  // false when swaps occur

        for (int pass = 1; (pass < n) && (!sorted); ++pass)
        {       // Invariant: theArray(n+1-pass .. n-1] is sorted and > theArray[0..n-pass]
                sorted = true; // assume sorted
                for (int index = 0; index < n-pass; ++index)
                {       // Invariant: theArray[0..index – 1] <= theArray[index]
                        int nextIndex = index + 1;
                        if (theArray[index] > theArray[nextIndex])
                        {       // exchange items
                                swap(theArray[index], theArray[nextIndex]);
                                sorted = false; // signal exchange
                        } // end if
                } // end for

                // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
        } // end for
} // end bubbleSort
```

4.3 Another version of it would be:

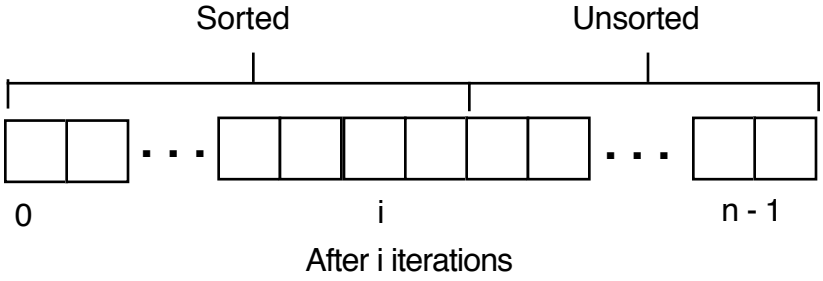
```
/** Sorts the items in an array into ascending order.
 * @pre    theArray is an array of n items.
 * @post  theArray is sorted into ascending order: n is unchanged
 * @param theArray     The given array
 * @param len    The size of theArray */
void bubbleSort(DataType theArray[], int len)
{
   bool sorted;
   int n = len-1;
   do
   {  sorted = true;
      for (int i = 0; i < n; i++)
         if (theArray[i] > theArray[i+1])
         {       swap(theArray[i], theArray[i+1]);
                 sorted = false;
          }
       n--;
   } while (!sorted);
}
```

4.4 Either version of the bubble sort would have worst case of $O(n^2)$.

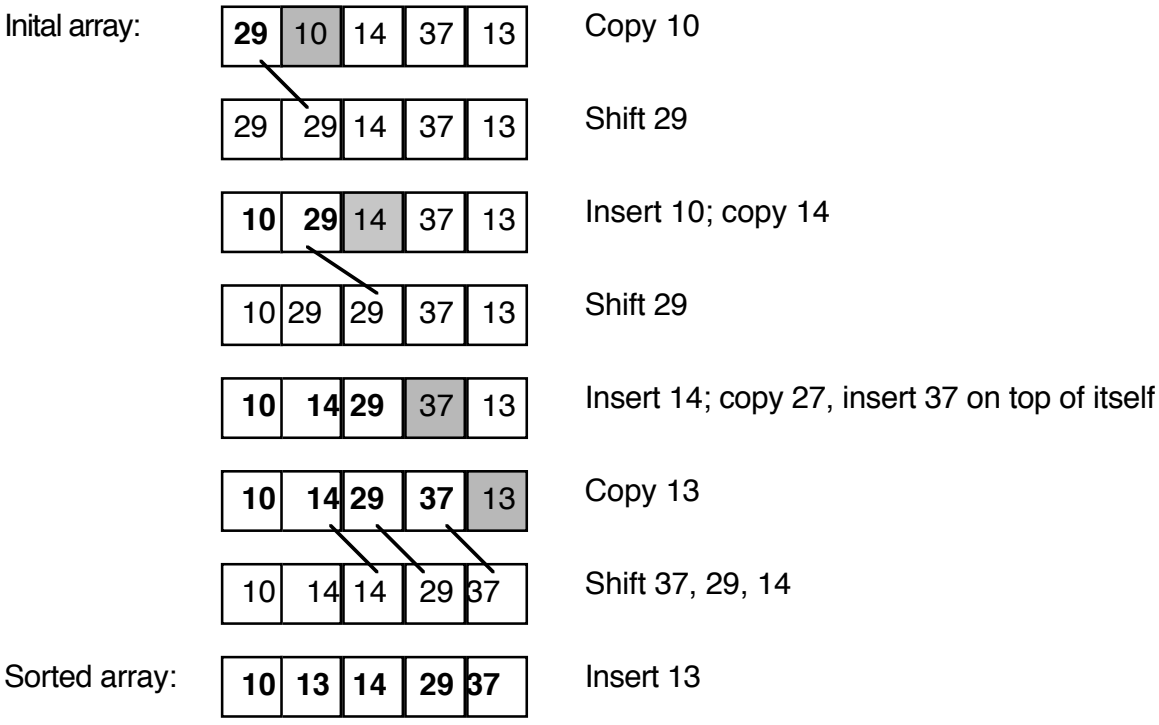
4.4.1 Best case (for a nearly sorted array) would be more like O(n).
4.4.2 Paradoxically, the bubble sort can be either among the slowest or among the fastest depending on the data.

5 Now imagine that you are arranging a had of cards but you pick up one card at a time and insert it into its proper position – that would be the idea of the Insertion Sort.

5.1 The insertion sort would divide the array into to parts: sorted and unsorted like shown in figure 9-6 on page 464:



5.2 You take each item, find the place it is supposed to go, shift the values down (if necessary), then copy the item to the correct position.
5.3 Figure 9-7 shows that:



5.4 The code for the insertion sort is:

```
/** Sorts the items in an array into ascending order.
 * @pre      theArray is an array of n items
 * @post    theArray is sorted into ascending order; n is unchanged
 * @param theArray   The given array
 * @param n   The size of theArray */
void insertionSort(DataType theArray[], int n)
{
        // unsorted = first index of the unsorted region,
        // loc = index of insertion in the sorted region,
        // nextItem = next item in the unsorted region

        // initially, sorted region is theArray[0],
        //                  unsorted region is theArray[1..n – 1];
        // in general, sorted region is theArray[0..unsorted – 1],
        // unsorted region is theArray[unsorted..n – 1]

        for (int unsorted = 1; unsorted < n; ++unsorted)
        {       // Invariant: theArray[0..unsorted – 1] is sorted
                // find the right position (loc) in theArray[0..unsorted]
                // for theArray[unsorted], which is the first item in the
                // unsorted region, shift, if necessary, to make room
                DataType nextItem = theArray[unsorted];
                int loc = unsorted;

                for(; (loc > 0) && (theArray[loc – 1] > nextItem); --loc)
                        // shift theArray[loc-1] to the right
                        theArray[loc] = theArray[loc – 1];
                // Assertion: theArray[loc] is where nextItem belongs

                // insert nextItem into Sorted region
                theArray[loc] = nextItem;
        } // end for
} // end insertionSort
```

5.5 The equation for insertion sort would be: $n^* (n – 1) + 2 * (n – 1) = n^2 + n – 2$.

5.6 The Big O for insertion sort is $O(n^2)$. For small arrays, insertion sort works well. Large arrays, it is inefficient.

6 There are two important divide-and-conquer sorting algorithms: mergesort and quicksort.

6.1 They have elegant recursive formulations and are highly efficient.

6.2 Mergesort will generalize with external sorting.

6.2.1 Mergesort also gives the same performance regardless how the array is initially set up.

6.2.2 The basic idea is to divide the array into halves, sort each half, then merge the sorted halves into one sorted array as figure 9-8.

6.2.3 Mergesort uses an auxiliary temporary array.

```
Const in MAX_SIZE = maximum-number-of-items-in-array;

/** Merges two sorted array segments theArray[first..mid] and theArray[mid+1..last]
 *  into one sorted array.
 *  @pre  first <= mid <= last. The subarrays theArray[first..mid] and
 *           theArray[mid+1..last] are each sorted in increasing order.
 *  @post  theArray[first..mid] is sorted.
 *  @param theArray  The given array.
 *  @param first   The beginning of the first segment in theArray.
 *  @param mid   The end of the first segment in theArray.
 *              mid+1 marks the beginning of the second segment.
 *  @param last    The last element in the second segment in theArray.
 *  @note  This function merges the two subarrays into a temporary array
 *            and copies the result into the original array theArray. */
void merge(DataType theArray[], int first, int mid, int last)
{
        DataType tempArray[MAX_SIZE];  // temporary array

        // initialize the local indexes to indicate the subarrays
        int first1 = first;         // beginning of first subarray
        int last1 = mid;            // end of first subarray
        int first2 = mid + 1;       // beginning of second substring
        int last2 = last;           // end of second subarray

        // while both subarrays are not empty, copy the smaller item
        // into the temporary array
        int index = first1;         // next available location in tempArray
        for (; (first1 <= last1) && (first2 <= last2); ++index)
        {       // Invariant: tempArray[first..index-1] is in order
                if (theArray[first1] < theArray[first2])
                {       tempArray[index] = theArray[first1];
                        ++first1;
                }
                else
                {       tempArray[index] = theArray[first2];
                        ++first2;
                } // end if
        } // end for

        // finish off the nonempty subarray

        // finish off the first subarray, if necessary
```

```
        for (; first1 <= last1; ++ first1, ++ index)
                // Invariant: tempArray[first..index-1] is in order
                tempArray[index] = theArray[first1];

        // finish off the second subarray, if necessary
        for (; first2 <= last2; ++first2, ++index)
                // Invariant: tempArray[first..index-1] is in order
                tempArray[index] = theArray[first2];

        // copy the result back into the original array
        for (index = first; index <= last; ++index)
                theArray[index] = tempArray[index];
} // end merge

/** Sorts the items in an array into ascending order.
 *  @pre     theArray[first..last] is an array.
 *  @post    theArray[first..last] is sorted in ascending order.
 *  @param theArray   The given array.
 *  @param first    The first element to consider in theArray.
 *  @param last     The last element to consider in theArray. */
void mergesort(DataType theArray[], int first, int last)
{
        if (first < last)
        {       // sort each half
                int mid = (first + last)/2;         // index of midpoint
                // sort left half theArray[first..mid]
                mergesort(theArray, first, mid);
                // sort right half theArray[mid+1..last]
                mergesort(theArray, mid+1, last);

                // merge the two halves
                merge(theArray, first, mid, last);
        } // end if
} // end mergesort
```

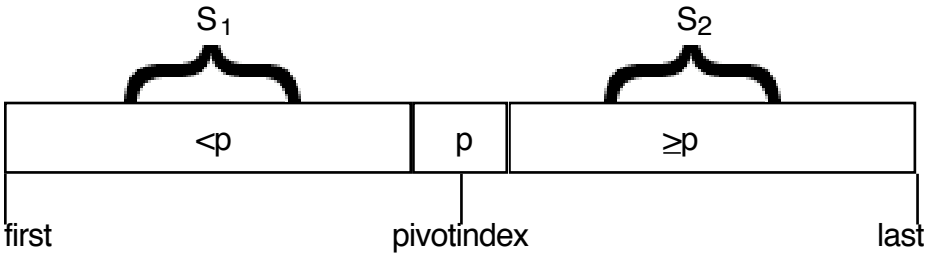
6.2.4 Mergesort has O(n * log n) in both worst and average cases.

6.3 The basic idea of Quicksort is to pick a value from the array and use it as a pivot: those values less than the pivot value go to the left and those greater go to the right.

6.3.1 Or with pseudocode:

```
Choose a pivot item p from theArray[first..last]
Partition the items of theArray[first..last] about p
```

6.3.2 Interesting properties of quicksort is that those values less than pivot will always be to the left of the pivot and those greater than will always be to the right as shown in Figure 9-12:



6.3.3 This lends itself nicely to recursion where you quicksort the left partition of S1 and then sort the right partition of S2.
6.3.4 You can keep calling the quicksort routine until it is sorted.
6.3.5 The pseudocode for the quicksort is:

```
Quicksort(inout theArray:ItemArray, in first:integer, in last:integer)
// Sorts theArray[first..last]
        if (first < last)
        {       Choose a pivot item p from theArray[first..last]
                Partition the items of theArray[first..last] about p
                // the partition is theArray[first..pivotIndex..last]

                // sort S1
                quicksort(theArray, first, pivotIndex-1)
                // sort S2
                quicksort(theArray, pivotIndex+1, last)
        }
        // if first >= last, there is nothing to do
```

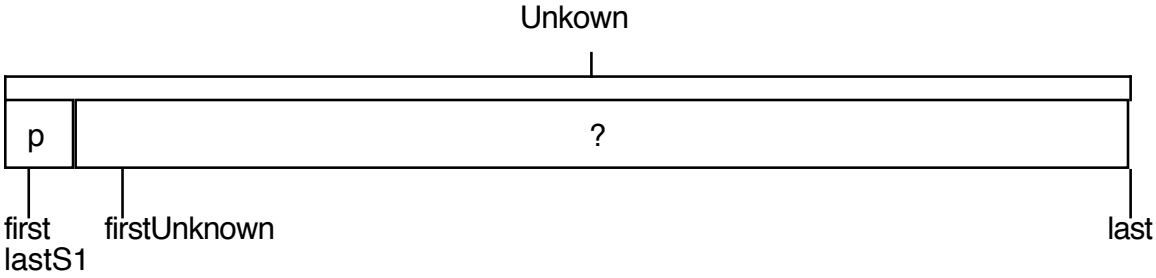
6.3.6 Trying to figure out which pivot to use can be a bit of a problem.
6.3.6.1 Many times, using the first position as the pivot works well.
6.3.6.2 Now the invariant that can be used is:

The items in the region S1 are all less than the pivot, and those in S2 are all greater than or equal to the pivot.

6.3.7 You need to have a partition algorithm to do the partitioning.
6.3.7.1 It would start as:

```
lastS1 = first
firstUnknown = first + 1
```

6.3.7.2 Figure 9-15 shows the initial partitioning:



6.3.7.3 The following pseudocode is for the partitioning algorithm:

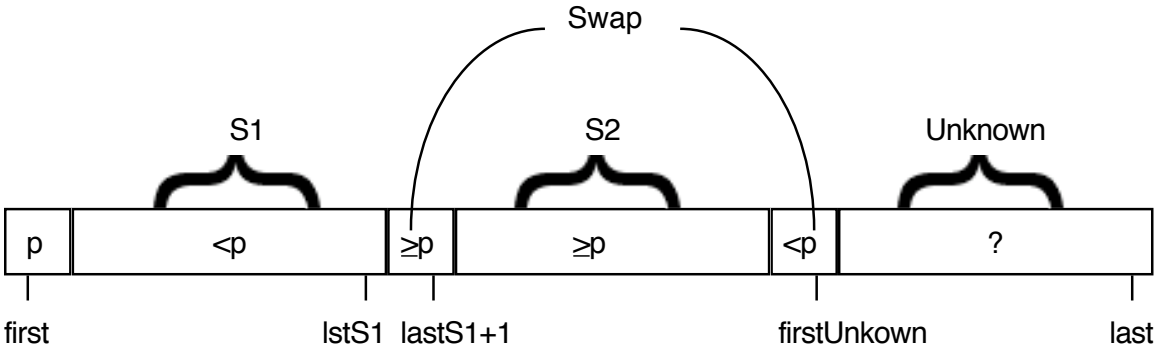
```
Partition(inout theArray:ItemArray, in first:integer, in last:integer,
          out pivotIndex:integer)
// Partitions theArray[first..last]

    // initialize
    Choose the pivot and swap it with theArray[first]
    p = theArray[first]      // p is the pivot

    // set S1 and S2 to empty, set unknown region to theArray[first+1..last]
    lastS1 = first
    firstUnknown = first + 1

    // determine the regions S1 and S2
    while (firstUnknown <= last)
    {       // consider the placement of the "leftmost" item
            //    in the unknown region
            if (theArray[firstUnknown] < p)
                    Move theArray[firstUnknown] into S1
            else
                    Move theArray[firstUnkown] into S2
    } // end while

    // place pivot in proper position between S1 and S2,
    //    and mark its new location
    swap theArray[first] with theArray[lastS1]
    pivotIndex = lastS1
```

6.4 A bit of a word about the Move into the partitions.

6.4.1 Moving theArray[firstUnknown] into S1 would involve comparing the value with the pivot then swapping it IF it is less than the pivot then incrementing the lastS1 index and incrementing the firstUnknown index.

6.4.2 What that does is move the right of the S1 partition over.

6.4.3 Figure 9-16 shows swapping from unknown partition into S1.

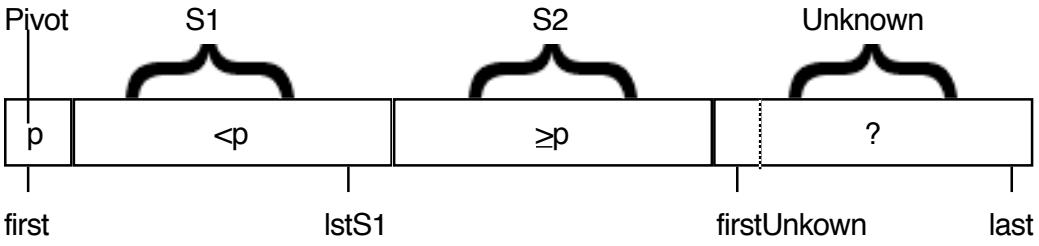


6.4.4 However, there is a S2 partition between S1 and the Unknown partition.

6.4.5 Do we move into that partition if the unknown value is greater than or equal?

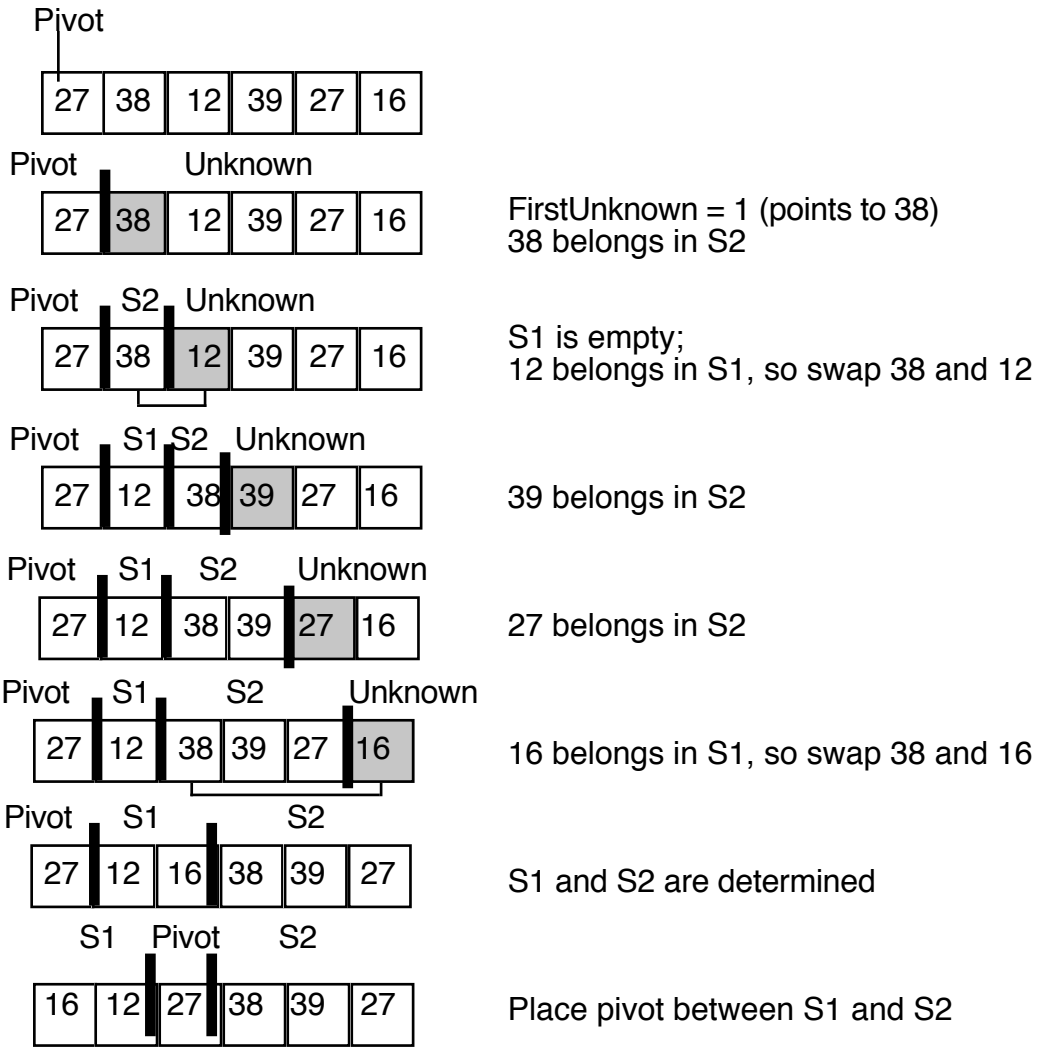
6.4.5.1 Actually, if we just increment the firstUnknown index that is the same as placing it in the S2 partition!

6.4.5.2 Figure 9-17 shows "swapping" from unknown into S2 partition.



6.4.6 The final step is to move the pivot to the right most position of S1.

6.4.7 The figure 9-18 shows how partition would work:



6.5 The code for Quicksort follows:

```
/** Chooses a pivot for quicksort's partition algorithm and swaps it with the first item
 *    in an array
 *  @pre  theArray[first..last] is an array; first <= last
 *  @post  theArray[first] is the pivot
 *  @param  theArray      The given array
 *  @param  first     The first element to consider in theArray
 *  @param  last       The last element to consider in theArray */
void choosePivot(DataType theArray[], int first, int last);
// you figure it out (or just use the first element

/** Partitions an array for quicksort
 *  @pre theArray[first..last] is an array; first <= last
 *  @post Partitions   theArray[first..last] such that:
 *       S1 = theArray[first..pivotIndex-1]      < pivot
 *            theArray[pivotIndex]               == pivot
```

```
 *       S2 = theArray[pivotIndex+1..last]      > pivot
 *  @param first     The first element to consider in theArray
 *  @param last      The last element to consider in theArray
 *  @param pivotIndex   The index of the pivot after partitioning */
void partition(DataType theArray[], int first, int last, int& pivotIndex)
{       // place pivot in theArray[first]
        choosePivot(theArray, first, last);
        DataType pivot = theArray[first];   // copy pivot

        // initially, everything but pivot is in unknown
        int lastS1 = first;                 // index of last item in S1
        int firstUnknown = first + 1;  // index of first item in unknown

        // move one item at a time until unknown region is empty
        for (; firstUnknown <= last; ++firstUnknown)
        {       // Invariant: theArray[first+1..lastS1] < pivot
                //       theArray[lastS1+1..firstUnknown-1] >= pivot

                // move item from unknown to proper region
                if (theArray[firstUnknown] < pivot)
                {       // item from unknown belongs in S1
                        ++lastS1;
                        swap(theArray[firstUnknown], theArray[lastS1]);
                } // end if

                // else item from unknown belongs in S2
        } // end for

        // place pivot in proper position and mark its location
        swap[theArray[first], theArray[lastS1]);
        pivotIndex = lastS1;
} // end partition

/** Sorts the items in an array into ascending order
 *  @pre theArray[first..last] is an array
 *  @post theArray[first..last] is sorted
 *  @param theArray   The given array
 *  @param first    The first element to consider in theArray
 *  @param last     The last element to consider in theArray */
void quicksort(DataType theArray[], int first, int last)
{
        int pivotIndex;

        if (first < last)
        {       // create the partition: S1, pivot S2
                partition(theArray, first, last, pivotIndex);
```

```
            // sort regions S1 and S2
            quicksort(theArray, first, pivotIndex-1);
            quicksort(theArray, pivotIndex+1, last);
        } // end if
} // end quicksort
```

6.6 Then the quicksort calls itself to sort S1 and then call itself to sort S2. Eventually, you have a sorted array!

7 Mergesort and Quicksort are similar in spirit whereas Quicksort does its work before the recursive call and Mergesort does it after the recursive call.
- 7.1 The Quicksort has worst case of $O(n^2)$ compared to Mergesort's O(n*log n).
- 7.2 However, their average case is both O(n*log n).
- 7.3 So, depending on the data, Mergesort could be faster or Quicksort could be faster.
- 7.4 Therefore is it not cut and dry, which sort to use.

8 The fastest sort, hands down, is the Radix sort.
- 8.1 But it is not a general-purpose sort as you will see shortly.
- 8.2 Imagine you are sorting a hand of cards 2, 3, … 10, J, Q, K, A and you place them into 13 groups for each value.
- 8.3 Then combine them into one pile then sort them into 4 groups according to suit then combine again.
- 8.4 You will have a sorted deck of cards by suit!

9 Radix sort uses this idea of forming groups then combining them into one.
- 9.1 That sorts the values quickly. Let's use the idea of numbers – 4 digit numbers to be exact.
  - 9.1.1 Radix sort will have a loop according to the number of digits in the number.
    - 9.1.1.1 So for 4 digit numbers, we go from the least significant to the most significant or from right to left to place the number into different groups, combine them, and separate again according to which digit is being used at the time.

    - 9.1.1.2 Figure 9-21 has an example:

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by forth digit
1560, 2150, 1061, 0222, 0123 0283 2154 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined

9.1.2 The pseudocode for Radix sort is:

```
radixSort(inout theArray:ItemArray, in n:integer, in d:integer)
// Sorts n d-digit integers in the array theArray

    for (j = d down to 1)
    {       Initialize 10 groups to empty
            Initialize a counter for each group to 0
            for (i = 0 through n-1)
            {       k = j^th digit of theArra[i]
                    Place theArray[i] at the end of group k
                    Increase k^th counter by 1
            } // end for i

            Replace the items in theArray with all the items in group 0, followed
                    by all the items in group 1, and so on
    } // end for j
```

9.2 Big O for Radix is O(n), which is about as good as you can get!

9.2.1 But Radix sort trades speed for memory.

9.2.2 To sort an array of 1,000 numbers, you need at least 10 other arrays of 1,000 each, one array per digit.

9.2.3 If you tried to do the deck of cards, you'd need 13 arrays, upper case alphabet you'd need 26, upper and lower case you'd need 52, and so on.

9.2.4 It quickly becomes untenable to sort large arrays of text (or even numbers, for that matter).

9.3 Figure 9-22 on page 486 summarizes the worst-case and average case orders of magnitude.

9.4 We'll include the treesort and heapsort that will be covered in future chapters.

Sort	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix Sort	n	n
Treesort rt	n^2	n * log n
Heapsort	n * log n	n * log n

9.5 Of course, there are STL sorts that can be used.

9.6 You can study those on your own.