LACROSSE UNIVERSITY

SORTING ALGORITHMS

A PAPER SUBMITTED TO
THE FACULTY OF THE DIVISON OF SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

By
Ayesha Asghar

May 10, 2006

To Khalida

If thy theorems fail deduction reasoning thou shalt apply induction reasoning with the most meticulous care.

Rashid

CONTENTS

ILLUSTRATIONS

Figure Page

PREFACE

This paper is a collection of algorithms for sorting. The descriptions given are brief and intuitive, with just enough theory thrown in to elucidate the sorting concepts. I assume one knows a high-level language, such as C++, and that are familiar with programming concepts including arrays and pointers.

The first section gives an introduction to algorithms and various basic concepts associated to it. The next section presents most commonly used *sorting* algorithms. Source code, examples and analysis for each algorithm, in MS Visual C++, is included.

The last section critically evaluates the sorting algorithms, throws light on the importance and application of sorting algorithms and concludes on a note explaining the need for understanding the algorithms.

ACKNOWLEDGEMENTS

I would like to thank the auspicious instructors of Lacrosse University under whose guidance I've been able to come up with this report. I thank them for their cooperation through out and also for resolving all problems and every issue that I faced.

Next, I would like to thank my family who provided me with all the sources and material required for making this possible. Lastly; I would like to thank God Almighty who gave me the strength and will power to complete this piece of work.

1. Introduction

1.1. Purpose

The purpose of this paper is to describe a well-formed and a thorough analysis of the sorting algorithms used in practice by programmers. This paper also gives an insight into algorithms and their analysis plus performance.

It focuses on the most commonly used sorting algorithms, chalks out their pros and cons. An empirical analysis has been given to elucidate the efficiency concerns. Further it has been explained in depth using general algorithms and examples written in MS Visual C++.

This paper provides a deep understanding of the need and benefits of sorting algorithms. This paper will assist the novice programmers to have a better understanding of sorting algorithms and their application. While, the expert programmers will also agree upon the comparison and concept details provided.

1.2. Scope

The paper provides answers to what, when, how and who questions related to algorithms, such as:

- What are algorithms?
- What is algorithm analysis and performance?
- Commonly used sorting algorithms.
- Source Codes and examples for sorting algorithms.
- Benefits and need for sorting algorithms.

2. Algorithms

2.1. Overview

An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibn Musu Al-Khowarizmi is a set of rules for carrying out calculation either by hand or on a machine. It is a finite step-by-step procedure to achieve a required result.

In its most general sense, an algorithm is any set of detailed instructions which results in a predictable end-state from a known beginning. Algorithms are only as good as the instructions given, however, and the result will be incorrect if the algorithm is not properly defined.

A common example of an algorithm would be instructions for assembling a model airplane. Given the starting set of a number of marked pieces, one can follow the instructions given to result in a predictable end-state: the completed airplane. Misprints in the instructions, or a failure to properly follow a step will result in a faulty end product.

A computer program is another pervasive example of an algorithm. Every computer program is simply a series of instructions (of varying degrees of complexity) in a specific order, designed to perform a specific task.

The most famous algorithm in history dates well before the time of the ancient Greeks: this is Euclid's algorithm for calculating the greatest common divisor of two integers.

2.1.1. The Classic Multiplication Algorithm

Multiplication, the American way

Multiply the multiplicand one after another by each digit of the multiplier taken from right to left.

```
          9 8 1
        1 2 3 4
------------------
        3 9 2 4
      2 9 4 3
    1 9 6 2
  9 8 1
------------------
1 2 1 0 5 5 4
```

Fig. 1. Multiplication, the American way

Multiplication, the English way

Multiply the multiplicand one after another by each digit of the multiplier taken from left to right.

```
          9 8 1
        1 2 3 4
------------------
  9 8 1
1 9 6 2
    2 9 4 3
      3 9 2 4
------------------
1 2 1 0 5 5 4
```

Fig. 2. Multiplication, the English way

Algorithmic is a branch of computer science that consists of designing and analyzing computer algorithms. Where, the “design” pertain to the description of algorithm at an abstract level by means of a pseudo language, and proof of correctness that is, the algorithm

solves the given problem in all cases. And the "analysis" deals with performance evaluation (complexity analysis).

2.2. Performance

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. The time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the **O** (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are **Θ** (Big Theta) notation and **Ω** (Big Omega) notation.

The performance evaluation of an algorithm is obtained by totalling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

Θ-Notation (Same order)

This notation bounds a function to within constant factors. It says $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

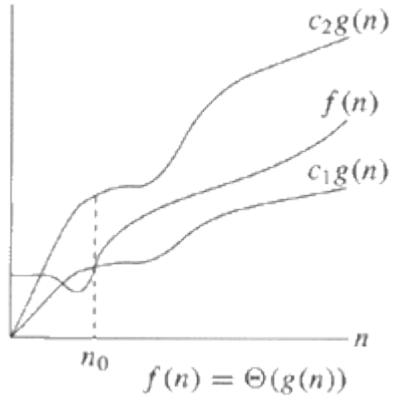

Fig. 3. Algorithm performance; Θ-Notation

O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. One writes $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.

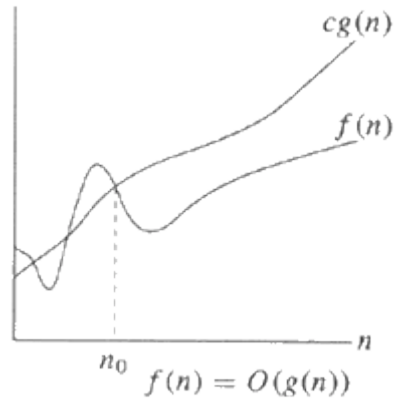

Fig. 4. Algorithm performance; O-Notation

Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. One writes $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

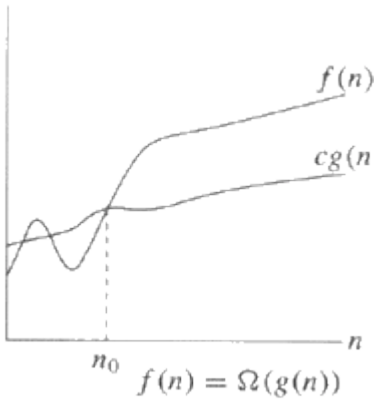

Fig. 5. Algorithm Performance; Ω-Notation

2.3. Analysis

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

There are two interpretations of upper bound.

Worst-case Complexity

The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.

Average-case Complexity

The running time for any given size input will be the average number of operations over all problem instances for a given size.

Because, it is quite difficult to estimate the statistical behaviour of the input, most of the time one contents oneself to a worst case behaviour. Most of the time, the complexity of $g(n)$ is approximated by its family $o(f(n))$ where $f(n)$ is one of the following functions. n (linear complexity), log n (logarithmic complexity), n^a where $a \geq 2$ (polynomial complexity), a^n (exponential complexity).

2.4. Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem. For example, any algorithm solving "the intersection of n segments" problem will execute at least n^2 operations in the worst case even if it does nothing but print the output. This is abbreviated by saying that the problem has $\Omega(n^2)$ complexity. If one finds an $O(n^2)$ algorithm that solve this problem, it will be optimal and of complexity $\Theta(n^2)$.

2.5. Reduction

Another technique for estimating the complexity of a problem is the transformation of problems, also called problem reduction. As an example, suppose one knows a lower bound for a problem A, and that would like to estimate a lower bound for a problem B. If one can transform A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

The Convex hull problem nicely illustrates "reduction" technique. A lower bound of Convex-hull problem established by reducing the sorting problem (complexity: $\Theta(nlogn)$) to the Convex hull problem.

2.6. Classes

While there is no universally accepted breakdown for the various types of algorithms, there are common classes that algorithms are frequently agreed to belong to. Among these are:

- **Dynamic Programming Algorithms:** This class remembers older results and attempts to use this to speed the process of finding new results.
- **Greedy Algorithms:** Greedy algorithms attempt not only to find a solution, but to find the ideal solution to any given problem.
- **Brute Force Algorithms:** The brute force approach starts at some random point and iterates through every possibility until it finds the solution.
- **Randomized Algorithms:** This class includes any algorithm that uses a random number at any point during its process.
- **Branch and Bound Algorithms:** Branch and bound algorithms form a tree of sub-problems to the primary problem, following each branch until it is either solved or lumped in with another branch.
- **Simple Recursive Algorithms:** This type of algorithm goes for a direct solution immediately, and then backtracks to find a simpler solution.
- **Backtracking Algorithms:** Backtracking algorithms test for a solution, if one is found the algorithm has solved, if not it recurs once and tests again, continuing until a solution is found.

- **Divide and Conquer Algorithms:** A divide and conquer algorithm is similar to a branch and bound algorithm, except it uses the backtracking method of recurring in tandem with dividing a problem into sub-problems.

In addition to these general classes, algorithms may also be divided into two primary groups: *serial algorithms*, which are designed for serial execution, wherein each operation is enacted in a linear order; and *parallel algorithms*, used with computers running parallel processors (as well as existing in the natural world in the case of, for example, genetic mutation over a species), wherein a number of operations are run in parallel.

3. Sorting Algorithms

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.

Sorting is one of the most common tasks performed by computers today. It has been estimated that as much as one quarter of the running time of all the world's computers is spent performing various types of sorting. Sorting is used to arrange names and numbers in meaningful ways. For example, it is relatively easy to look up the phone number of a friend because the names in the phone book have been sorted into alphabetical order. This example clearly illustrates one of the main reasons that sorting large quantities of information is desirable. That is, sorting greatly improves the efficiency of *searching*. If one was to open a phone book, and find that the names were not presented in any logical order, it would take an incredibly long time to look up someone's phone number!

Sorting can be performed in many ways. Over time, several methods have been devised to sort information following specific algorithms. Some examples of these algorithms are the Selection Sort, the Binary Sort, the Bubble Sort, the Merge Sort, the Heap Sort, the Quick Sort, and the Radix Sort. These will be explored in depth in this paper.

Some obvious questions that arise from the above discussion are: "Which sort method should one use? Is one sort algorithm clearly the best?" The answer to the second question is quite simple. There is no sort algorithm that is clearly better than all others in all circumstances. The answer to the first question, however, is difficult. The particular situations faced by people who need to sort information will invariably dictate what type of sort algorithm should be used. Some factors that will play in the decision-making process are the complexity of the

algorithm, the size of the data structure (an array, for example) to be sorted, and the time it takes for programmers to understand and implement various algorithms. For example, a small business that manages a list of customer names could easily use an algorithm such as the bubble sort. This algorithm is easy to understand and implement. Because of the fact the data set to be sorted is relatively small; a simple algorithm such as the bubble sort will perform satisfactorily. However, a large corporation with millions of customers would experience horrible delays and massive lag time trying to sort their records with a bubble sort algorithm. They would be well advised to spend the extra time learning a more efficient algorithm, like the Quick Sort, to sort their data more efficiently. In actual fact, even the Quick Sort may not be perfectly suited for this example.

Below are listed the seven of the most common sorting algorithms and are discussed in-detail with the help of analysis, source codes and examples.

3.1. Bubble Sort

3.1.1. Algorithm Analysis

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 an abysmal $O(n^2)$.

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.

3.1.2. Empirical Analysis

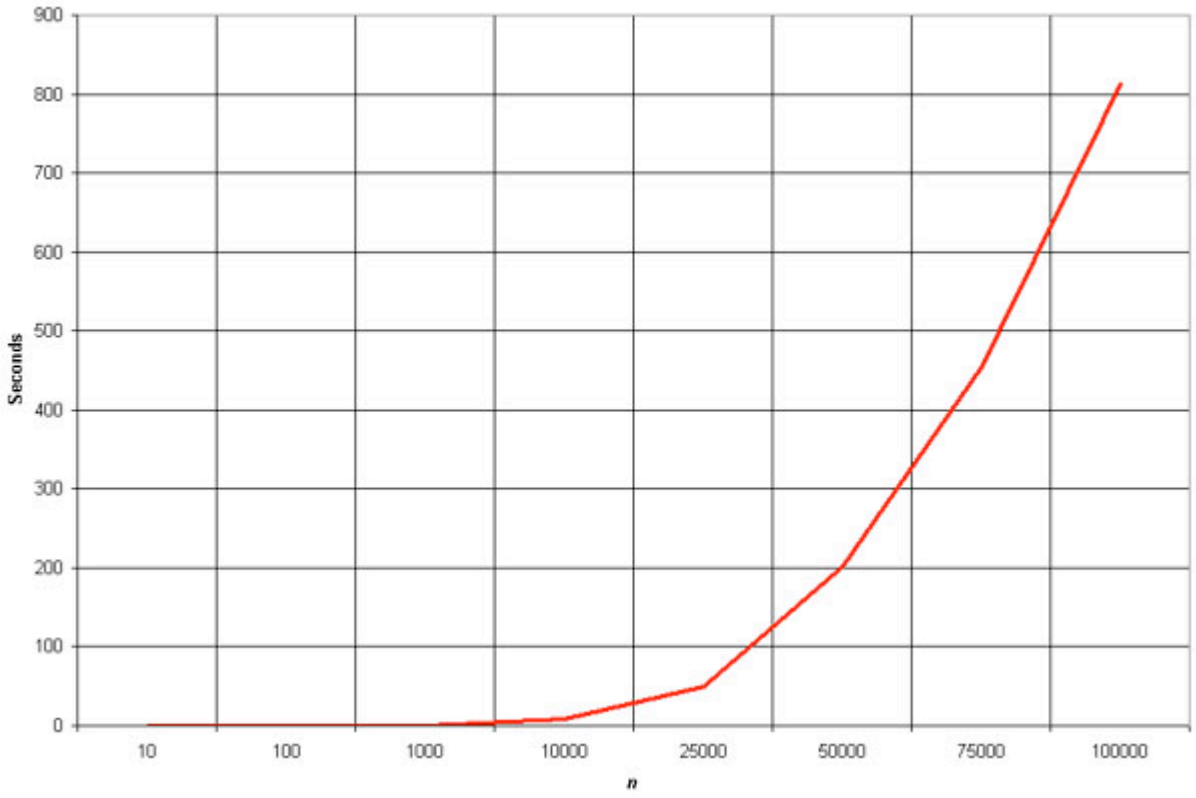

Fig. 6. Bubble Sort Efficiency

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

A fair number of algorithm purists (which means they've probably never written software for a living) 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 couple hundred items.

3.1.3. Source Code

Below is the basic bubble sort algorithm.

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

                for (i = (array_size - 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;
                                }
                        }
                }
        }
```

3.1.4. Example

A sample MS Visual C++ program that demonstrates the use of the bubble sort can be seen in Appendix A.

3.2. Heap Sort

3.2.1. Algorithm Analysis

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.

3.2.2. Empirical Analysis

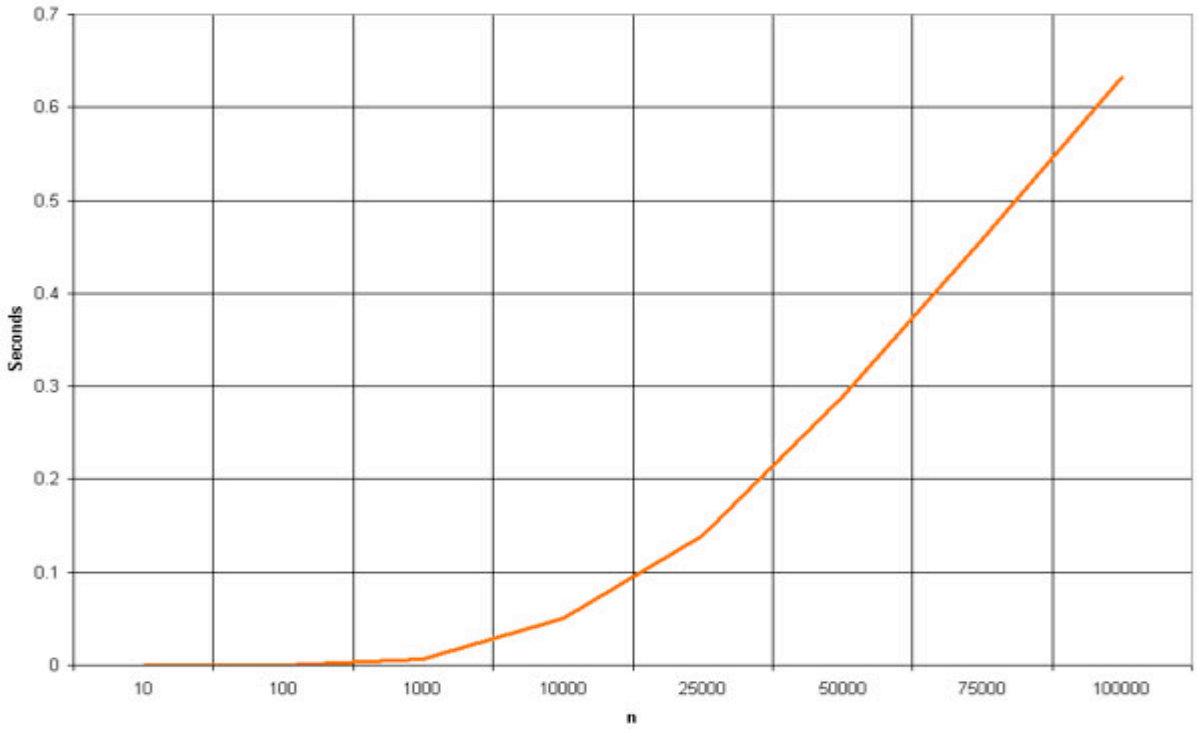

Fig. 7. Heap Sort Efficiency

As mentioned above, 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. But if one is working on a common user-level application, there's always going to be some yahoo who tries to run it on junk machine older than the programmer who wrote it, so better safe than sorry.

3.2.3. Source Code

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

```
void heapSort(int numbers[], int array_size)
        {
        int i, temp;

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

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

void siftDown(int numbers[], int root, int bottom)
        {
        int done, maxChild, temp;

        done = 0;

        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 = 1;
                }
        }
```

3.2.4. Example

A sample Microsoft Visual C++ program that demonstrates the use of the heap sort can be referred in Appendix B.

3.3. Insertion Sort

3.3.1. Algorithm Analysis

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.

3.3.2. Empirical Analysis

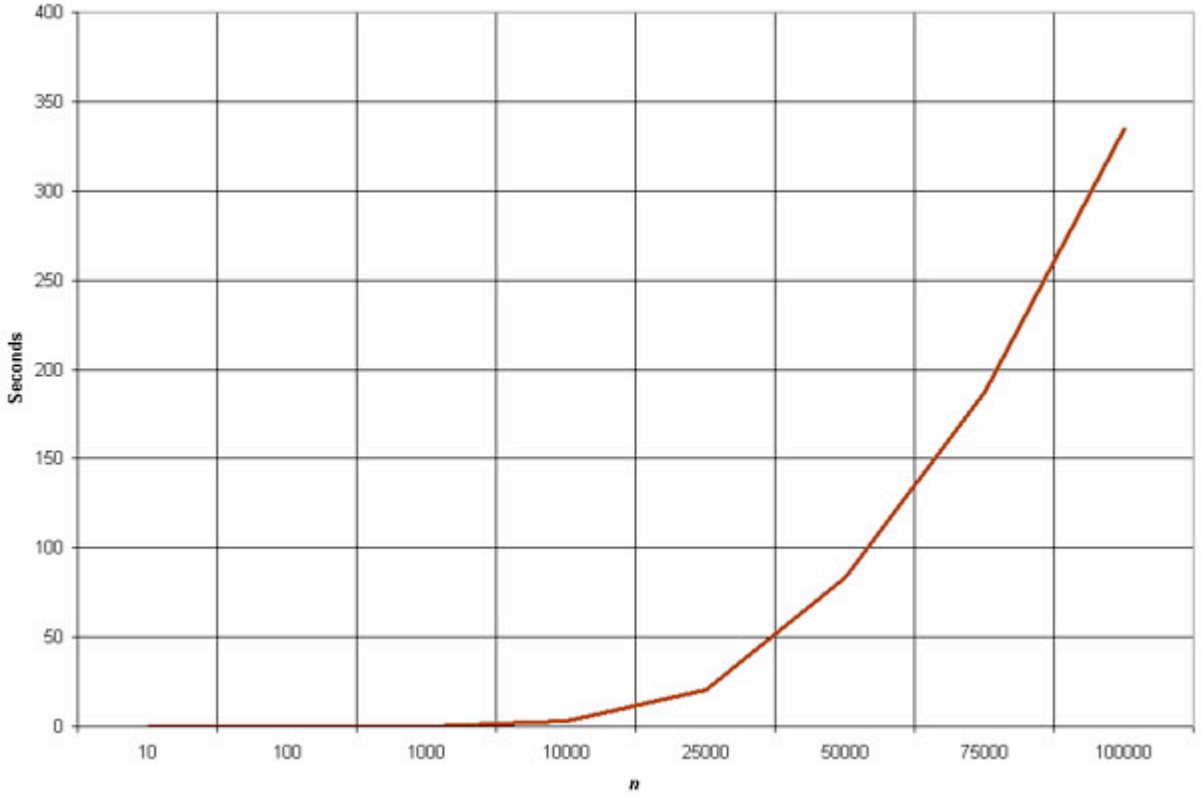

Fig. 8. Insertion Sort Efficiency

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.

3.3.3. Source Code

Below is the basic insertion sort algorithm.

```
void insertionSort(int numbers[], int array_size)
        {
        int i, j, index;
        for (i=1; i < array_size; i++)
                {
                index = numbers[i];
                j = i;

                while ((j > 0) && (numbers[j-1] > index))
                        {
                        numbers[j] = numbers[j-1];
                        j = j - 1;
                        }

                numbers[j] = index;
                }
        }
```

3.3.4. Example

A sample MS Visual C++ program that demonstrates the use of the insertion sort is given in Appendix C.

3.4. Merge Sort

3.4.1. Algorithm Analysis

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.

3.4.2. Empirical Analysis

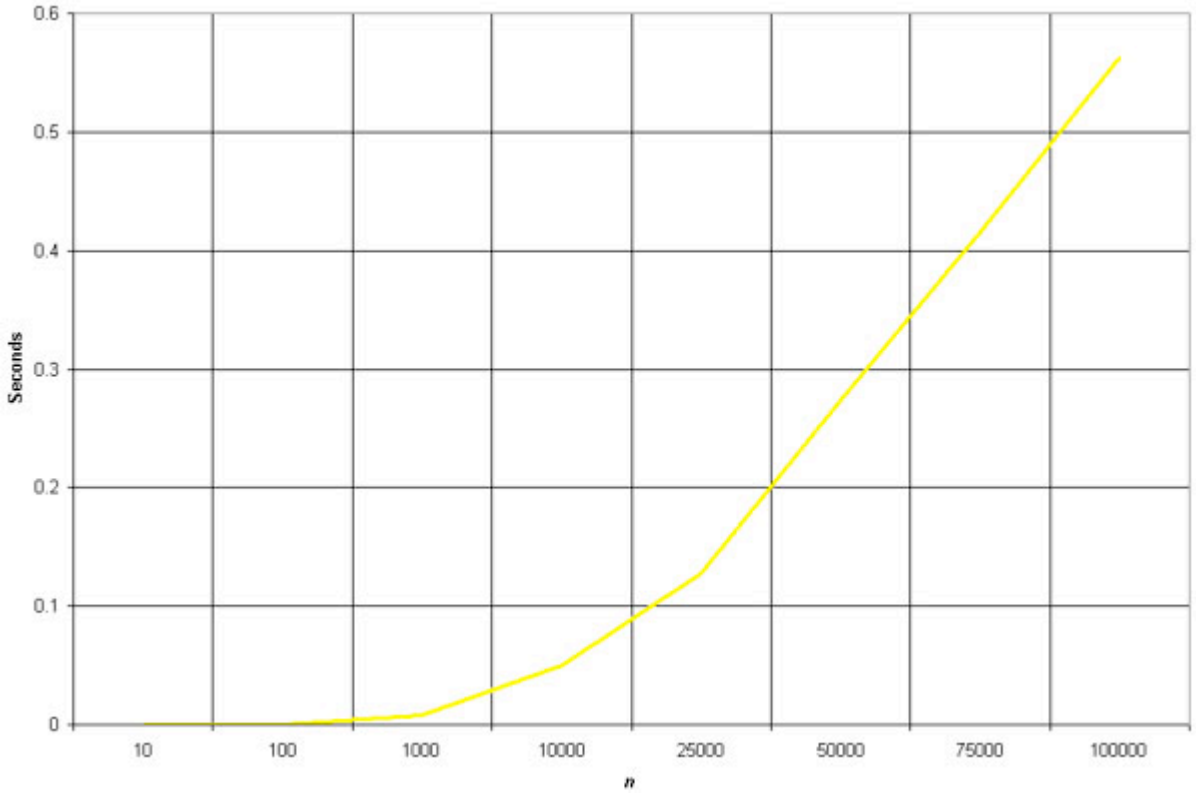

Fig. 9. Merge Sort Efficiency

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.

3.4.3. Source Code

Below is the basic merge sort algorithm.

```
void mergeSort(int numbers[], int temp[], int array_size)
{
  m_sort(numbers, temp, 0, array_size - 1);
}


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);
       }
}

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;
         }
}
```

3.4.4. Example

A sample C program that demonstrates the use of the merge sort can be seen in Appendix D.

3.5. Quick Sort

3.5.1. Algorithm Analysis

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.

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

- If there is/are one or less elements in the array to be sorted, return immediately.

- Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
- Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
- Recursively repeat the algorithm for both halves of the original array.

The efficiency of the algorithm is greatly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort, $O(n^2)$, 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.

3.5.2. Empirical Analysis

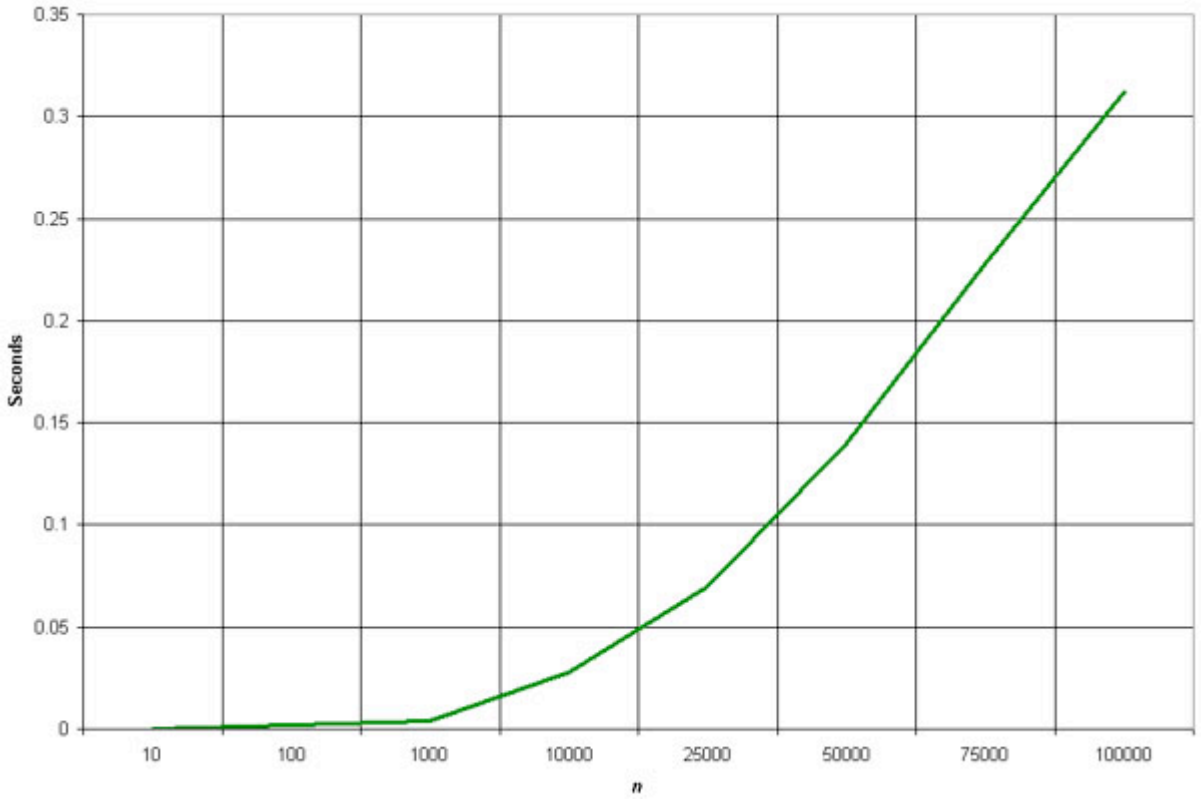

Fig. 10. Quick Sort Efficiency

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 soon as students figure this out, their immediate impulse is to use the quick sort for everything - after all, faster is better, right? It's important to resist this urge - the quick sort isn't always the best choice. As mentioned earlier, it's massively recursive (which means that for very large sorts, one 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.

With that said, in most cases the quick sort is the best choice if speed is important (and it almost always is). Use it for repetitive sorting, sorting of medium to large lists, and as a default choice when one is 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.

3.5.3. Source Code

Below is the basic quick sort algorithm.

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


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);
}
```

3.5.4. Example

A sample C program that demonstrates the use of the quick sort may be seen in Appendix E.

3.6. Selection Sort

3.6.1. Algorithm Analysis

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.

3.6.2. Empirical Analysis

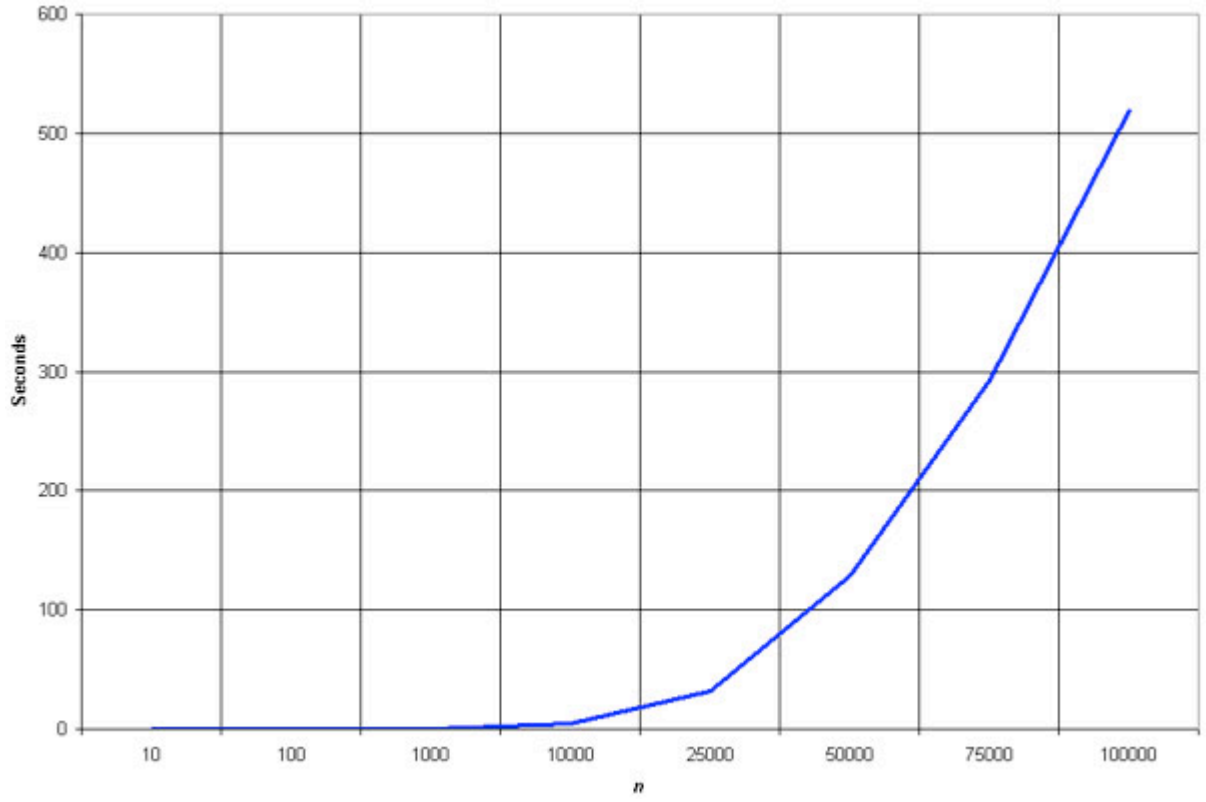

Fig. 11. Selection Sort Efficiency

The selection sort is the unwanted stepchild of the n^2 sorts. It 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. In short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If one really wants 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.

3.6.3. Source Code

Below is the basic selection sort algorithm.

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

         for (i = 0; i < array_size-1; i++)
         {
                min = i;
                   for (j = i+1; j < array_size; j++)
                   {
                     if (numbers[j] < numbers[min])
                       min = j;
                   }

                temp = numbers[i];
                numbers[i] = numbers[min];
                numbers[min] = temp;
         }
}
```

3.6.4. Example

A sample C program that demonstrates the use of the selection sort is given in Appendix F.

3.7. Shell Sort

3.7.1. Algorithm Analysis

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.

The size of the sets used, for each iteration; has a major impact on the efficiency of the sort. Several Heroes Of Computer Science™, including Donald Knuth and Robert Sedgewick, have come up with more complicated versions of the shell sort that improve efficiency by carefully calculating the best sized sets to use for a given list.

Pros

- Efficient for medium-size lists.

Cons

- Somewhat complex algorithm, not nearly as efficient as the merge, heap, and quick sorts.

3.7.2. Empirical Analysis

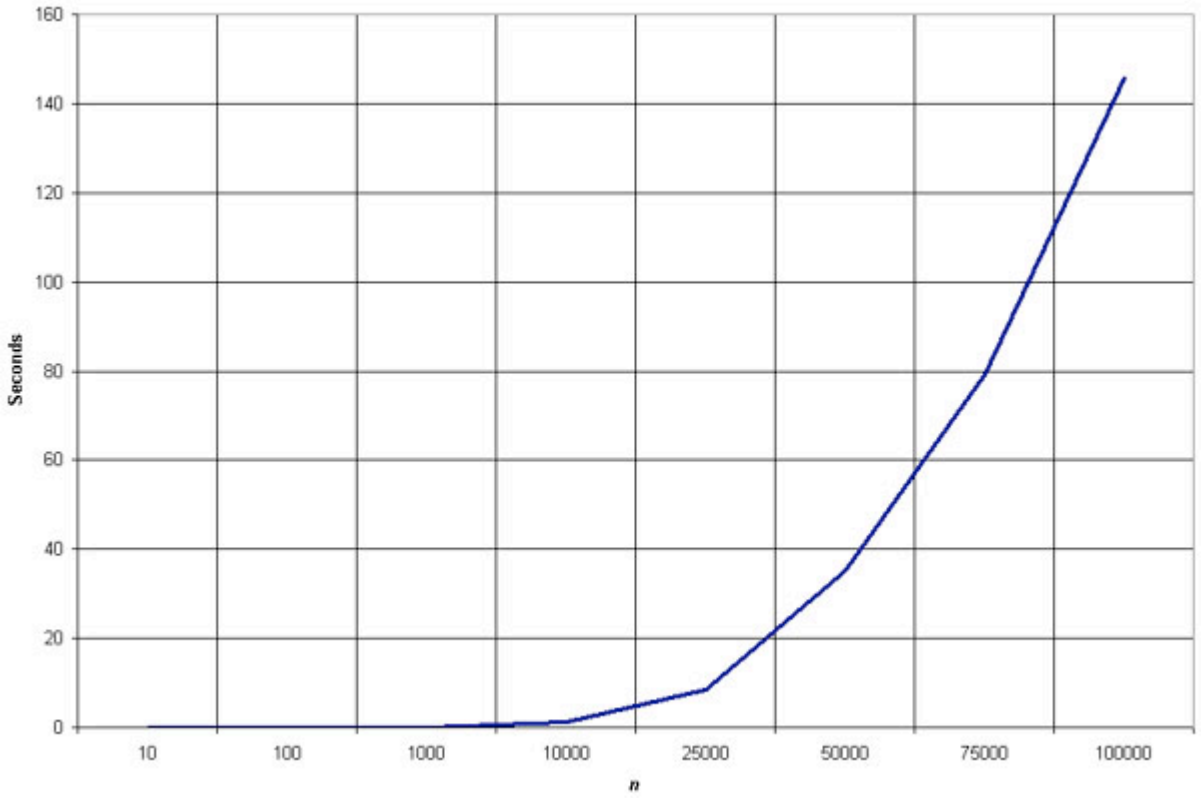

Fig. 12. Shell Sort Efficiency

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.

3.7.3. Source Code

Below is the basic shell sort algorithm.

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

  increment = 3;
  while (increment > 0)
  {
    for (i=0; i < array_size; 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;
  }
}
```

3.7.4. Example

A sample C program that demonstrates the use of the shell sort can be reviewed in Appendix G.

4. Applications of Sorting

An important key to algorithm design is to use sorting as a basic building block, because once a set of items is sorted, many other problems become easy. Consider the following applications:

- *Searching* - Binary search enables one to test whether an item is in a dictionary in $O(\lg n)$ time, once the keys are all sorted. Search preprocessing is perhaps the single most important application of sorting.

- *Closest pair* - Given a set of *n* numbers, how does one finds the pair of numbers that have the smallest difference between them? After the numbers are sorted, the closest pair of numbers will lie next to each other somewhere in sorted order. Thus a linear-time scan through them completes the job, for a total of $O(n \lg n)$ time including the sorting.

- *Element uniqueness* - Are there any duplicates in a given a set of *n* items? The most efficient algorithm is to sort them and then do a linear scan though them checking all adjacent pairs. This is a special case of the closest-pair problem above, where one might ask if there is a pair separated by a gap of zero.

- *Frequency distribution* - Given a set of *n* items, which element's occurrence has the largest number of times in the set? If the items are sorted, one can sweep from left to right and count them, since all identical items will be lumped together during sorting. To find out how often an arbitrary element *k* occurs, start by looking up *k* using binary search in a sorted array of keys. By walking to the left of this point until the element is not *k* and then walking to the right, one can find this count in $O(\lg n + c)$ time, where *c* is the number of occurrences of *k*. The number of instances of *k* can be found in $O(\lg n)$ time by using binary

search to look for the positions of both $k-\varepsilon$ and $k+\varepsilon$, where ε is arbitrarily small, and then taking the difference of these positions.

- *Selection* - What is the *k*th largest item in the set? If the keys are placed in sorted order in an array, the *k*th largest can be found in constant time by simply looking at the *k*th position of the array. In particular, the median element appears in the (*n*/2)nd position in sorted order.

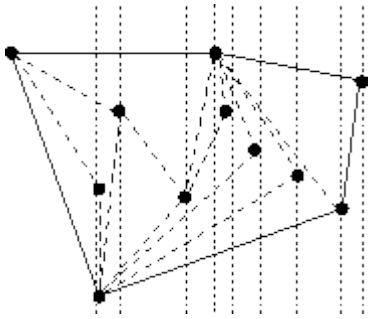

Fig. 13. Constructing the convex hull of points in the plane

- *Convex hulls* - Given *n* points in two dimensions, what is the polygon of smallest area that contains them all? The convex hull is like a rubber band stretched over the points in the plane and then released. It compresses to just cover the points, as shown in the figure above. The convex hull gives a nice representation of the shape of the points and is the most important building block for more sophisticated geometric algorithms.

  But how can one use sorting to construct the convex hull? Once one has the points sorted by *x*-coordinate, the points can be inserted from left to right into the hull. Since the rightmost point is always on the boundary, it is known that it will be inserted into the hull. Adding this new rightmost point might cause others to be deleted, but one can quickly identify these points because they lie inside the polygon formed by adding the new point.

These points to delete would be neighbors of the previous point that are inserted, so they will be easy to find. The total time is linear after the sorting has been done.

While some of these problems (particularly median and selection) can be solved in linear time using more sophisticated algorithms, sorting provides quick and easy solutions to all of these problems. It is a rare application whose time complexity is such that sorting proves to be the bottleneck, especially a bottleneck that could have otherwise been removed using more clever algorithmics. Don't ever be afraid to spend time sorting whenever one uses an efficient sorting routine.

5. Importance of Sorting

By the time one graduates, computer science students are likely to have studied the basic sorting algorithms in their introductory programming class, then in their data structures class, and finally in their algorithms class. Why is sorting worth so much attention? There are several reasons:

- Sorting is the basic building block around which many other algorithms are built. By understanding sorting, one obtains an amazing amount of power to solve other problems.

- Historically, computers have spent more time sorting than doing anything else. A quarter of all mainframe cycles are spent sorting data [Knuth, 1973]. Although it is unclear whether this remains true on smaller computers, sorting remains the most ubiquitous combinatorial algorithm problem in practice.

- Sorting is the most thoroughly studied problem in computer science. Literally dozens of different algorithms are known, most of which possess some advantage over all other algorithms in certain situations. To become convinced of this, the reader is encouraged to browse through [Knuth, 1973], with hundreds of pages of interesting sorting algorithms and analysis.

- Most of the interesting ideas used in the design of algorithms appear in the context of sorting, such as divide-and-conquer, data structures, and randomized algorithms.

6. Critical Evaluation

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 was $O(n^2)$ (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.

The two classes of sorting algorithms are $O(n^2)$, 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 run times from one computer to another on 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 one uses.

These empirical efficiency graphs are kind of like golf - the lowest line is the "best". Keep in mind that "best" depends on one’s 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.

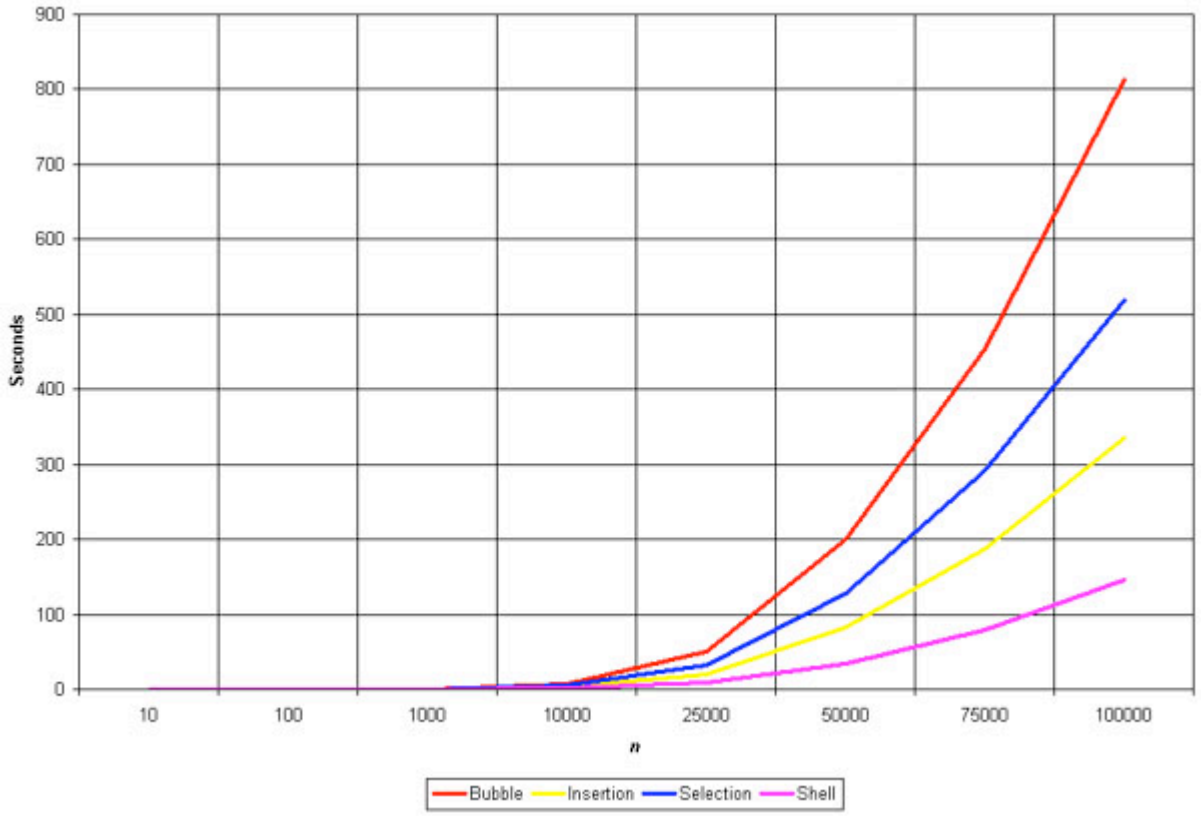

Fig. 14. O(n2) Sorts

As the graph pretty plainly shows, the bubble sort is grossly inefficient, and the shell sort blows it out of the water. Notice that the first horizontal line in the plot area is 100 seconds - these aren't sorts that one wants 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 one tries to sort much more than 10,000 data items.

On the bright side, 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 one really thinks one needs split-second efficiency.

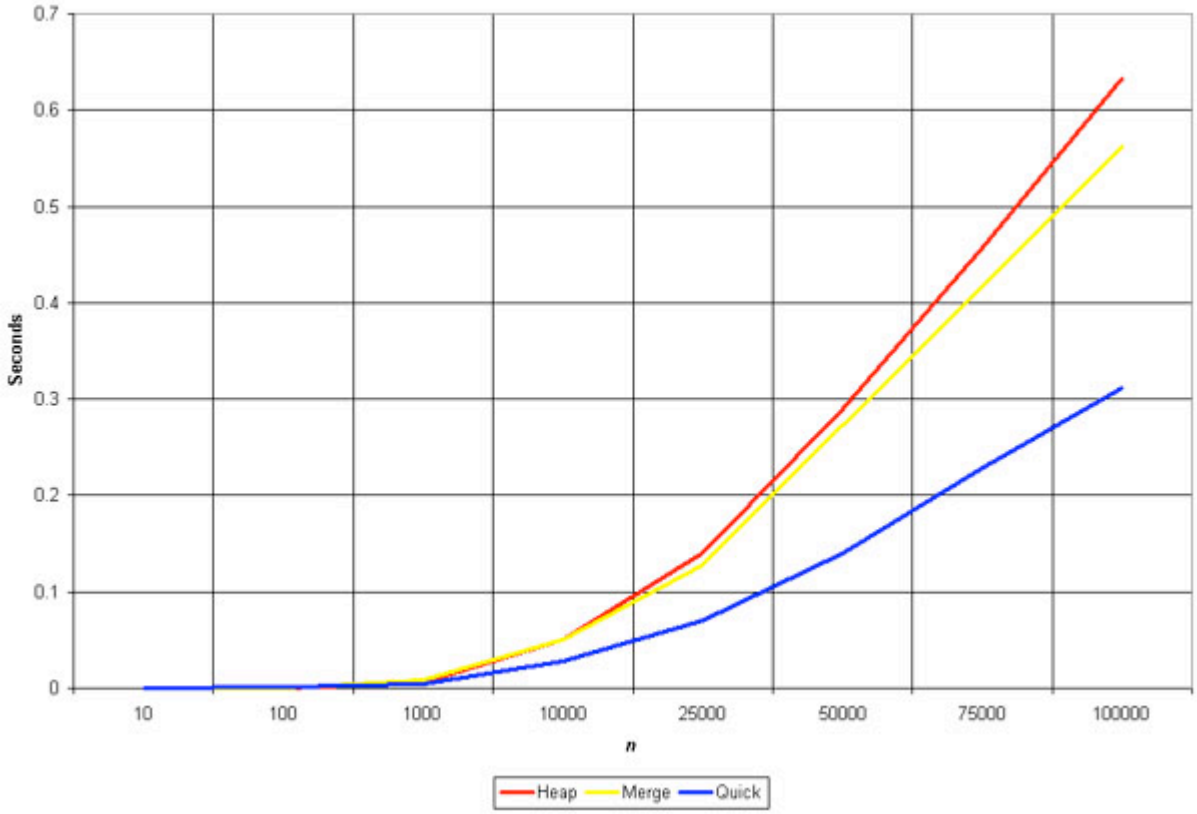

Fig. 15. O(n log n) Sorts

Speaking of split-second efficiency, the $O(n \log n)$ sorts are where it's at. Notice that the time on this graph is measured in tenths of seconds, instead hundreds of seconds like the $O(n^2)$ graph.

But as with everything else in the real world, there are trade-offs. These algorithms are blazingly fast, but that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays - these algorithms make extensive use of those nasty things.

In the end, the important thing is to pick the sorting algorithm that one thinks is appropriate for the task at hand.

7. Conclusion

As a computer scientist, it is important to understand all of these types of algorithms so that one can use them properly. If one is working on an important piece of software, one will likely need to be able to estimate how fast it is going to run. Such an estimate will be less accurate without an understanding of runtime analysis. Furthermore, one needs to understand the details of the algorithms involved so that one will be able to predict if there are special cases in which the software won't work quickly, or if it will produce unacceptable results.

Of course, there are often times when one will run across a problem that has not been previously studied. In these cases, one has to come up with a new algorithm, or apply an old algorithm in a new way. The more one knows about algorithms in this case, the better one's chances are of finding a good way to solve the problem. In many cases, a new problem can be reduced to an old problem without too much effort, but one will need to have a fundamental understanding of the old problem in order to do this.

As an example of this, let's consider what a switch does on the Internet. A switch has N cables plugged into it, and receives packets of data coming in from the cables. The switch has to first analyze the packets, and then send them back out on the correct cables. A switch, like a computer, is run by a clock with discrete steps - the packets are send out at discrete intervals, rather than continuously. In a fast switch, one wants to send out as many packets as possible during each interval so they don't stack up and get dropped. The goal of the algorithm to develop here is to send out as many packets as possible during each interval, and also to send them out so that the ones that arrived earlier get sent out earlier. In this case it turns out that an algorithm for a problem that is known as "stable matching" is directly applicable to this problem, though at first glance this relationship seems unlikely. Only through pre-existing algorithmic knowledge and understanding can such a relationship be discovered.

The different algorithms that people study are as varied as the problems that they solve. However, chances are good that the problem one is trying to solve is similar to another problem in some respects.

In short; by developing a good understanding of a large range of algorithms, one will be able to choose the right one for a problem and apply it properly.

Appendix A: Bubble Sort

This program demonstrates the use of the bubble sort algorithm

```
#include <iostream>
#define NUM_ITEMS 10

using std::cout;
using std::endl;

void bubbleSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];

void main( )
	{
	int i;
	srand(0);				//seed random number generator

	for (i = 0; i < NUM_ITEMS; i++)
		numbers[i] = rand();		//fill array with random integers

	bubbleSort(numbers, NUM_ITEMS);		//perform bubble sort on array

	cout<<"Done with sort.\n";

	for (i = 0; i < NUM_ITEMS; i++)
		cout<<numbers[i]<<endl;
	}

void bubbleSort(int numbers[], int array_size)
{
	int i, j, temp;

	for (i = (array_size - 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;
			}
		}
	}
}
```

Appendix B: Heap Sort

This program demonstrates the use of the heap sort algorithm.

```
#include <iostream>
#define NUM_ITEMS 10

using std::endl;
using std::cout;

void heapSort(int numbers[], int array_size);
void siftDown(int numbers[], int root, int bottom);

int numbers[NUM_ITEMS];

int main( )
{
        int i;
        srand(0);                   //seed random number generator

        for (i = 0; i < NUM_ITEMS; i++)
                numbers[i] = rand();                    //fill array with random integers

                heapSort(numbers, NUM_ITEMS);           //perform heap sort on array
                cout<<"Done with sort.\n";

                for (i = 0; i < NUM_ITEMS; i++)
                        cout<<numbers[i]<<endl;
}

void heapSort(int numbers[], int array_size)
{
        int i, temp;
        for (i = (array_size / 2)-1; i >= 0; i--)
                siftDown(numbers, i, array_size);

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

void siftDown(int numbers[], int root, int bottom)
{
        int done, maxChild, temp;
        done = 0;
```

```
        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 = 1;
                }
        }
}
```

Appendix C: Insertion Sort

This program demonstrates the use of the insertion sort algorithm.

```
#include<iostream>
#define NUM_ITEMS 100

void insertionSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];

int main()
{
	int i;
	srand(0);				//seed random number generator

	for (i = 0; i < NUM_ITEMS; i++)
		numbers[i] = rand();		//fill array with random integers

	insertionSort(numbers, NUM_ITEMS);	//perform insertion sort on array
		cout<<"Done with sort.\n";

	for (i = 0; i < NUM_ITEMS; i++)
		cout<<numbers[i]);
}

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

	for (i=1; i < array_size; i++)
	{
		index = numbers[i];
		j = i;

		while ((j > 0) && (numbers[j-1] > index))
		{
			numbers[j] = numbers[j-1]; j = j - 1;
		}
	numbers[j] = index;
	}
}
```

Appendix D: Merge Sort

This program demonstrates the use of the merge sort algorithm.

```
#include<iostream>
#define NUM_ITEMS 100

void mergeSort(int numbers[], int temp[], int array_size);
void m_sort(int numbers[], int temp[], int left, int right);
void merge(int numbers[], int temp[], int left, int mid, int right);
int numbers[NUM_ITEMS]; int temp[NUM_ITEMS];

int main( )
{
        int i;
        srand(0);                   //seed random number generator

        for (i = 0; i < NUM_ITEMS; i++)
                numbers[i] = rand();              //fill array with random integers

        mergeSort(numbers, temp, NUM_ITEMS);  //perform merge sort on array
                cout<<"Done with sort.\n";

        for (i = 0; i < NUM_ITEMS; i++)
                 cout<<numbers[i];
}

void mergeSort(int numbers[], int temp[], int array_size)
{
        m_sort(numbers, temp, 0, array_size - 1);
}

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);
        }
}

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;
                }
}
```

Appendix E: Quick Sort

This program demonstrates the use of the quick sort algorithm.

```
#include <iostream>
#define NUM_ITEMS 100

void quickSort(int numbers[], int array_size);
void q_sort(int numbers[], int left, int right);
int numbers[NUM_ITEMS];

int main( )
{
	int i;
	srand(0);		//seed random number generator

	for (i = 0; i < NUM_ITEMS; i++)
		numbers[i] = rand();		//fill array with random integers

	quickSort(numbers, NUM_ITEMS);		//perform quick sort on array
	cout<<"Done with sort.\n";

	for (i = 0; i < NUM_ITEMS; i++)
		cout<<numbers[i];
}

void quickSort(int numbers[], int array_size)
{
	q_sort(numbers, 0, array_size - 1);
}

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);
}
```

Appendix F: Selection Sort

This program demonstrates the use of the selection sort algorithm.

```
#include <iostteam>
#define NUM_ITEMS 100

void selectionSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];

int main()
{
	int i;
	srand(0);				//seed random number generator

	for (i = 0; i < NUM_ITEMS; i++)
		numbers[i] = rand();		//fill array with random integers

	selectionSort(numbers, NUM_ITEMS);		//perform selection sort on array
	cout<<"Done with sort.\n";

	 for (i = 0; i < NUM_ITEMS; i++)
		cout<<numbers[i]<<endl;
}

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

	for (i = 0; i < array_size-1; i++)
	{
		min = i;

		for (j = i+1; j < array_size; j++)
		{
			if (numbers[j] < numbers[min])
				min = j;
		}

		temp = numbers[i];
		numbers[i] = numbers[min];
		numbers[min] = temp;
	}
}
```

Appendix G: Shell Sort

This program demonstrates the use of the shell sort algorithm.

```
#include<iostream>
#define NUM_ITEMS 100

void shellSort(int numbers[], int array_size);
int numbers[NUM_ITEMS];

int main( )
{
        int i;
        srand(0);                   //seed random number generator

        for (i = 0; i < NUM_ITEMS; i++)
                numbers[i] = rand();            //fill array with random integers

        shellSort(numbers, NUM_ITEMS);              //perform shell sort on array
        cout<<"Done with sort.\n";

        for (i = 0; i < NUM_ITEMS; i++)
                cout<<numbers[i];
}

void shellSort(int numbers[], int array_size)
{
        int i, j, increment, temp;
        increment = 3;

        while (increment > 0)
        {
                for (i=0; i < array_size; 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;
        }
}
```

Bibliography

[Barker 1996; 1999]
Barker, Mark, 1996;1999, Discussion of Sorting Algorithms: Referencing, not plagiarism. Retrieved: April 20, 2006 from
http://atschool.eduweb.co.uk/mbaker/sorts.html

[Chalk, 2006]
Chalk , Peter D, (n.d.),Sorting Algorithms: Referencing, not plagiarism. Retrieved: April 22, 2006 from
http://homepages.north.londonmet.ac.uk/~chalkp/proj/algtutor/sortingal.html

[Cormen et al, 2001]
Cormen, T.H, Leiserson, C.E, Rivest, R.L, Stein, C. 2001, *Introduction to Algorithms.* 2nd Edition. McGraw-Hill. MIT Press.

[Horowitz et al, 1995]
Horowitz, E; Sahni, S; Mehta, D. 1995. *Fundamentals of Data Structures in C++*. Computer Science Press. England.

[Jason, 2006]
Jasson, Harrison. (n.d.) *Sorting Algorithms.* Retrieved: April 26, 2006 from
http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

[Katoen, 2002]
Katoen, J. Pieter, *Algorithms, Data Structure and Complexity*. Septemeber 10, 2002 from
www.cs.utwente.nl

[Knuth, 1973b]
Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading MA, 1973.

[Niemann, 2006]
Niemann, Thomas. [n.d.] *Sorting and Searching Algorithms.* Retrieved: April 20. 2006 from
http://www.epaperpress.com/

[OOPWeb, 2006]
OOPWeb, (n.d.), Sorting and Searching: Referencing: not plagiarism. Retrieved: April 24, 2006 from
http://oopweb.com/Algorithms/Documents/Sman/VolumeFrames.html

[Saskatchewan, 2006]
University of Saskatchewan, (n.d.), Introduction to Sorting: Referencing, not plagiarism. Retrieved: April 20, 2006 from
http://www.cs.usask.ca/resources/tutorials/csconcepts/1998_4/sorting.html

[Smith, 2006]
Smith, Maven. [n.d.] *Sorting Algorithms*. Retrieved: April 27, 2006 from http://maven.smith.edu/~thiebaut/java/sort/demo.html

[Toki, 2006]
Toki, (n.d.), Data Structures and Sorting: Referencing, not plagiarism. Retrieved: April 19, 2006 from http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK/NODE22.HTM

[Wikipedia, 2006]
Wikipedia- The Free Encyclopaedia. (n.d.) *Sorting Algorithm*. Retrieved: April 25, 2006 from http://en.wikipedia.org/wiki/Sort_algorithm#Graphical_representations