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.

Below are links to algorithms, analysis, and source code for seven of the most common sorting algorithms.

Sorting Algorithms
Bubble sort
Heap sort
Insertion sort
Merge sort
Quick sort
Selection sort
Shell sort

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is a complex subject (imagine that!) 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.

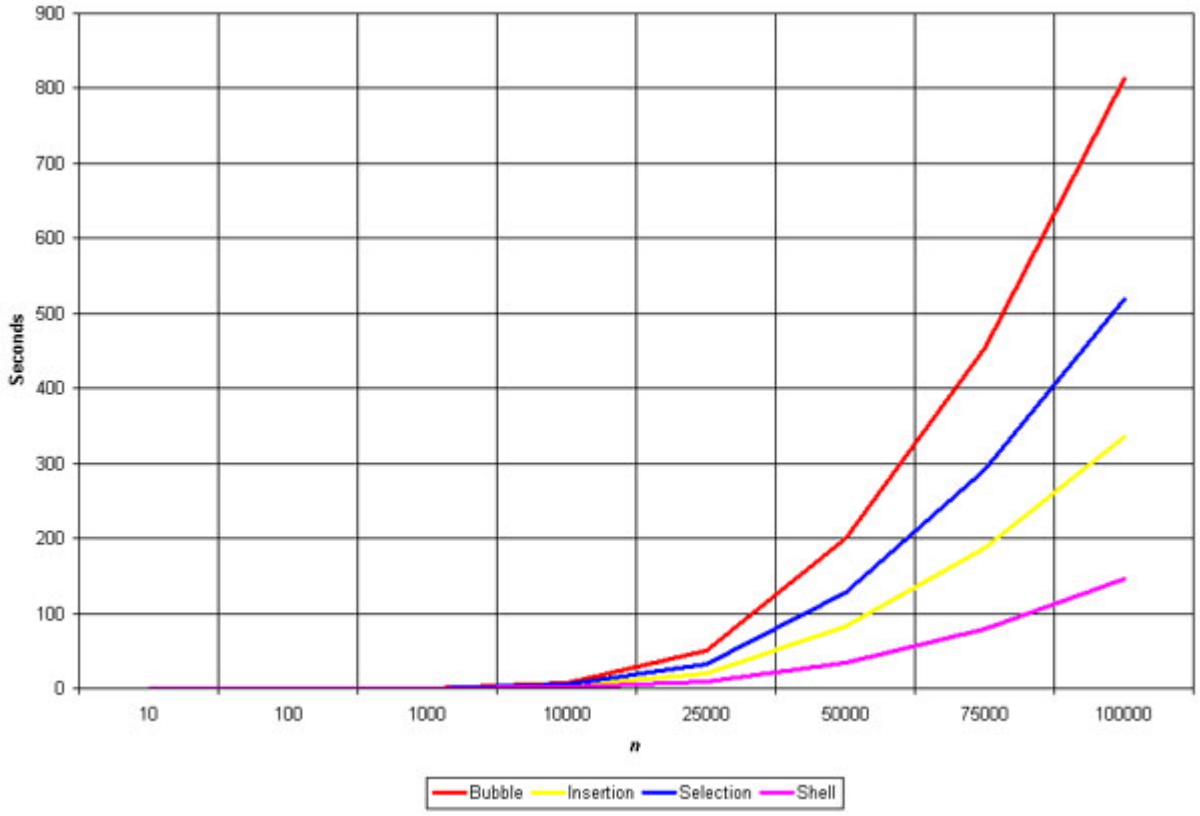
The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, and shell sorts; and $O(n \log n)$ which includes the heap, merge, and quick sorts.

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 to ether. The empirical data on this site is the averageof a hundred runs against random data sets

on a single-user 250MHz UltraSPARC II. The run times on your system will almost certainly vary from these results, but the relative speeds should be the same - the selection sort runs in roughly half the time of the bubble sort on the UltraSPARC II, and it should run in roughly half the time on whatever system you use as well.

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

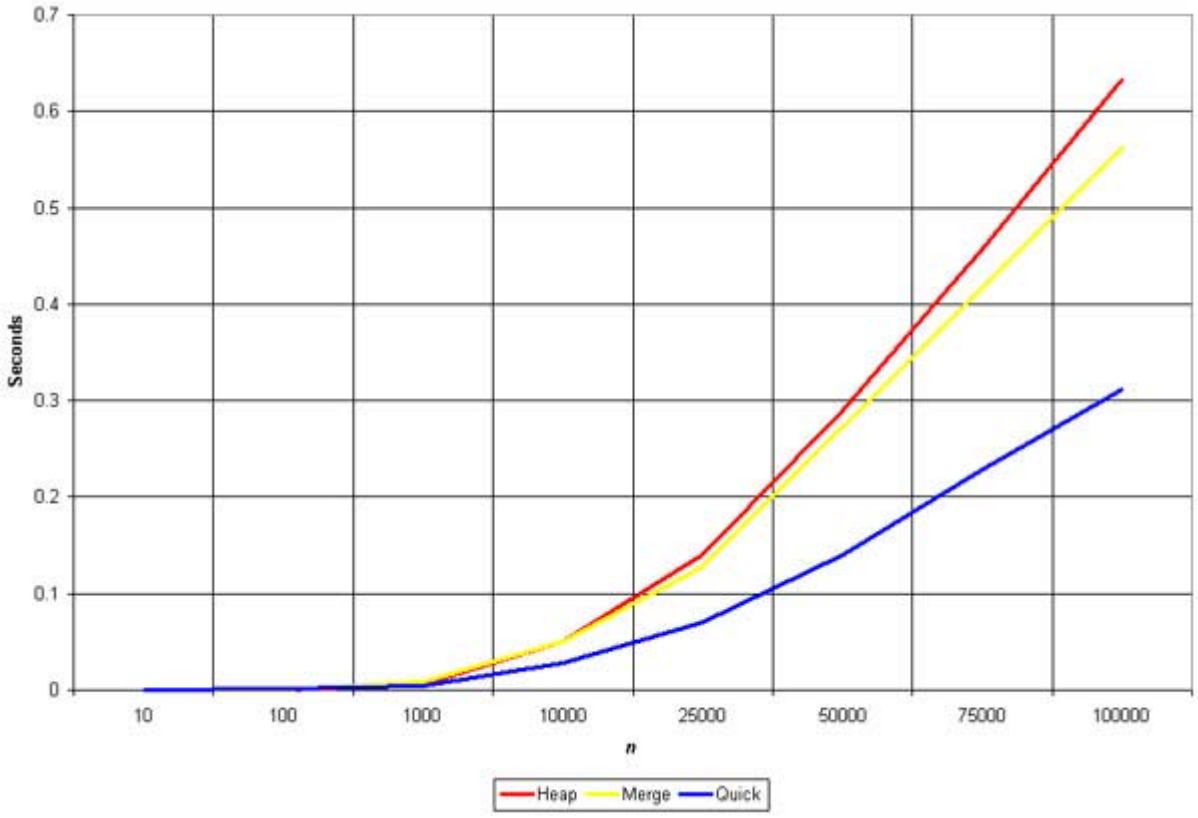
O(n^2) Sorts



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 you want to use for huge amounts of data in an interactive application. Even using the shell sort, users are going to be twiddling their thumbs if you try to sort much more than 10,000 data items.

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 you really think you need split-second efficiency.

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 you think is appropriate for the task at hand. You should be able to use the source code on this site as a "black box" if you need to - you can just use it, without understanding how it works. Obviously taking the time to understand how the algorithm you choose works is preferable, but time constraints are a fact of life.

Bubble Sort

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.

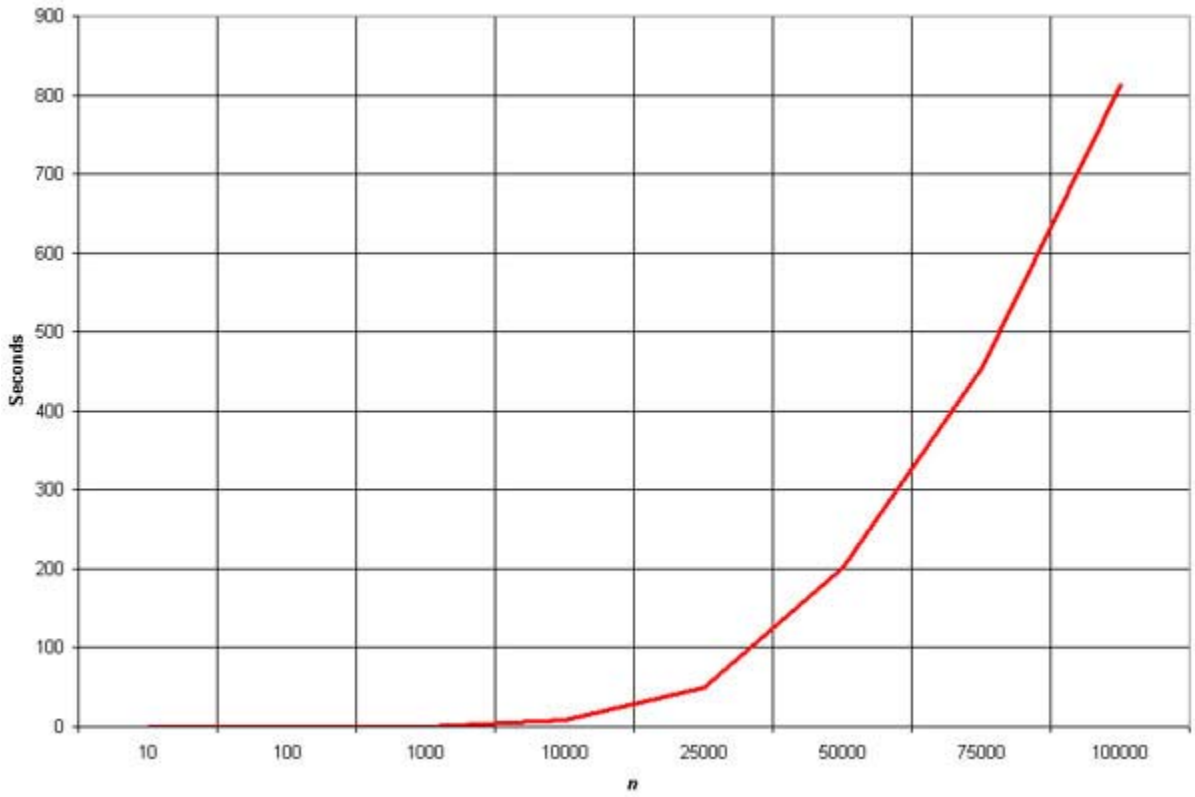
The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. General-case is an abysmal $O(n^2)$.

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.

Empirical Analysis

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.

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;
      }
    }
  }
}
```

Heap Sort

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.

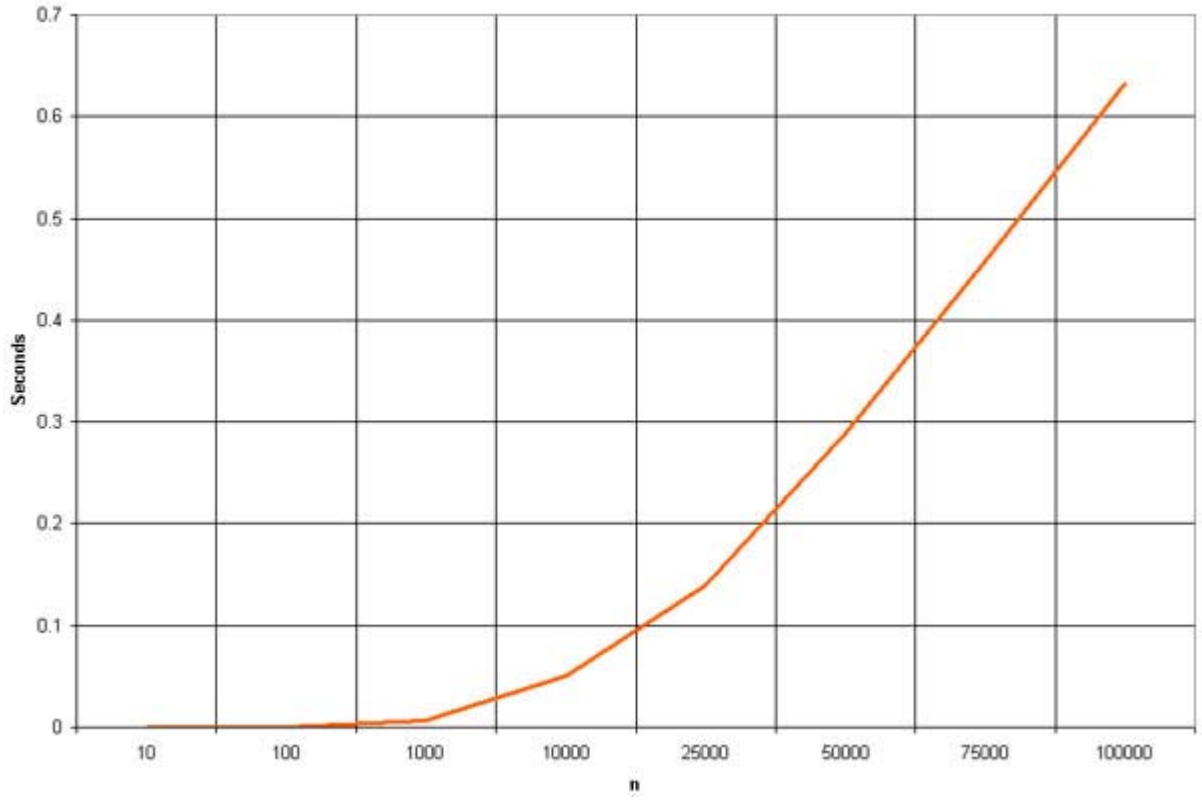
The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open 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.

Empirical Analysis

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 you're 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.

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;
  }
}
```

Insertion Sort

Algorithm Analysis

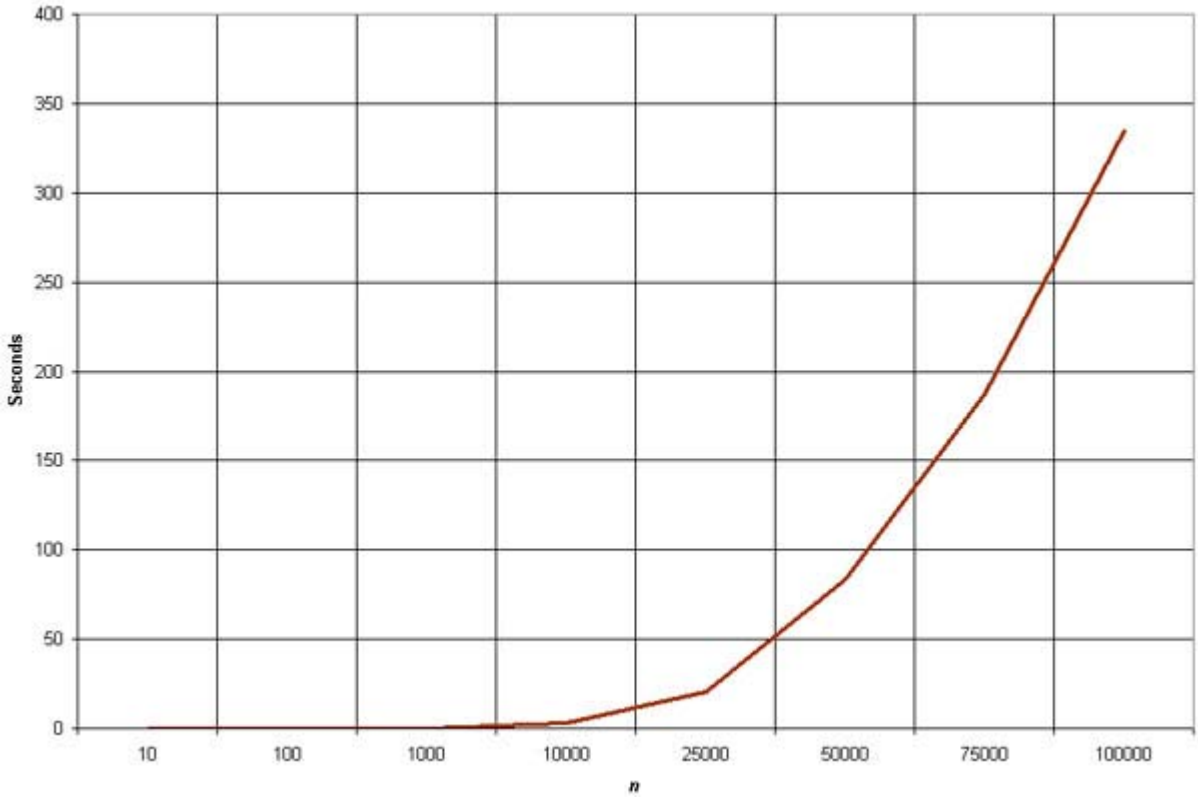
The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place 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.

Empirical Analysis

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.

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;
  }
}
```

Merge Sort

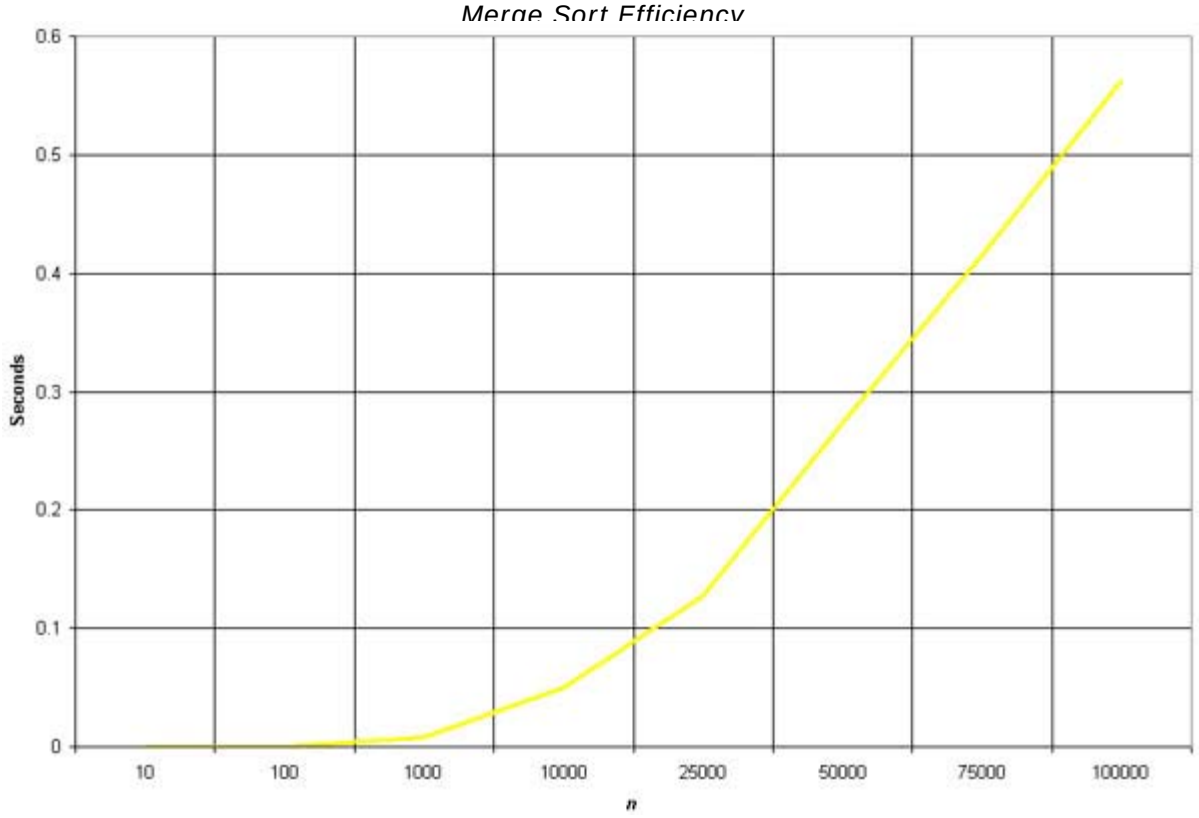
Algorithm Analysis

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.

Empirical Analysis



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.

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;
  }
}
```

Quick Sort

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 (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

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

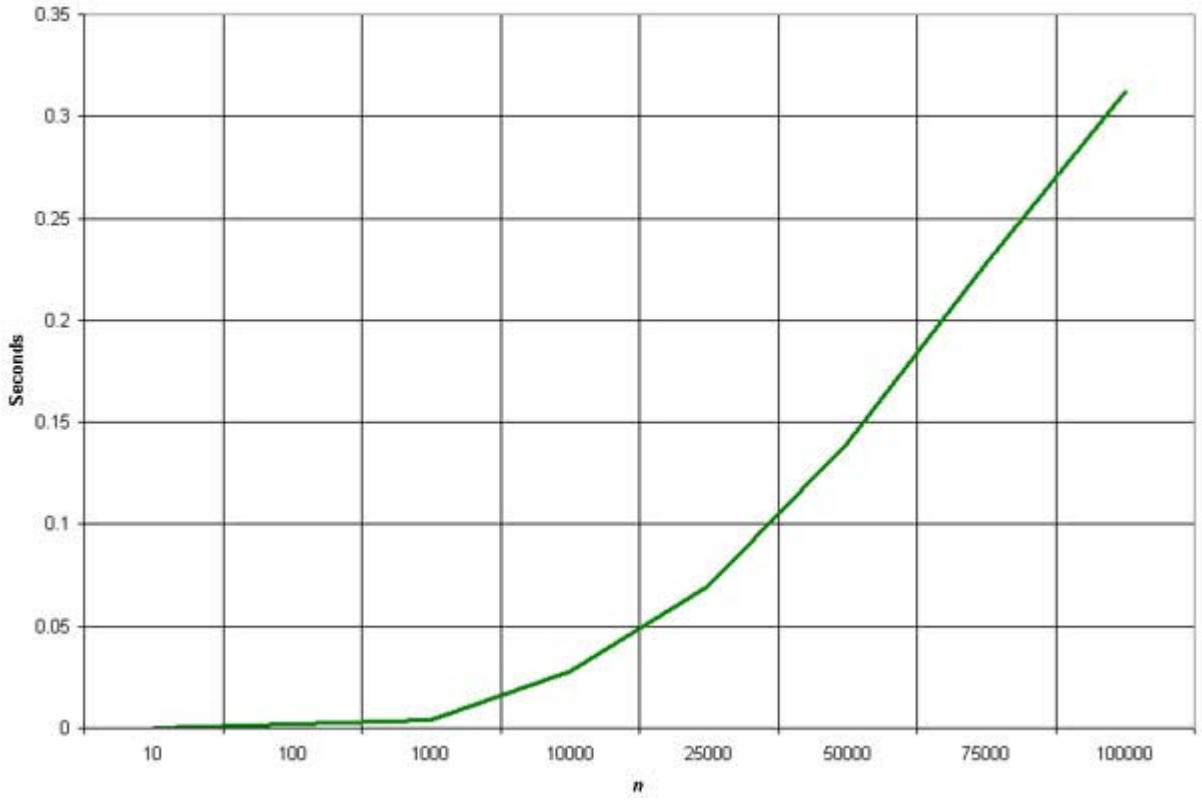
1. If there are one or less elements in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements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

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

Empirical Analysis

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 implulse 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, you can run the system out of stack space pretty easily). It's also a complex algorithm - a little too complex to make it practical for a one-time sort of 25 items, for example.

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 you're not really sure which sorting algorithm to use. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order - avoid it in those situations.

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);
}
```

Selection Sort

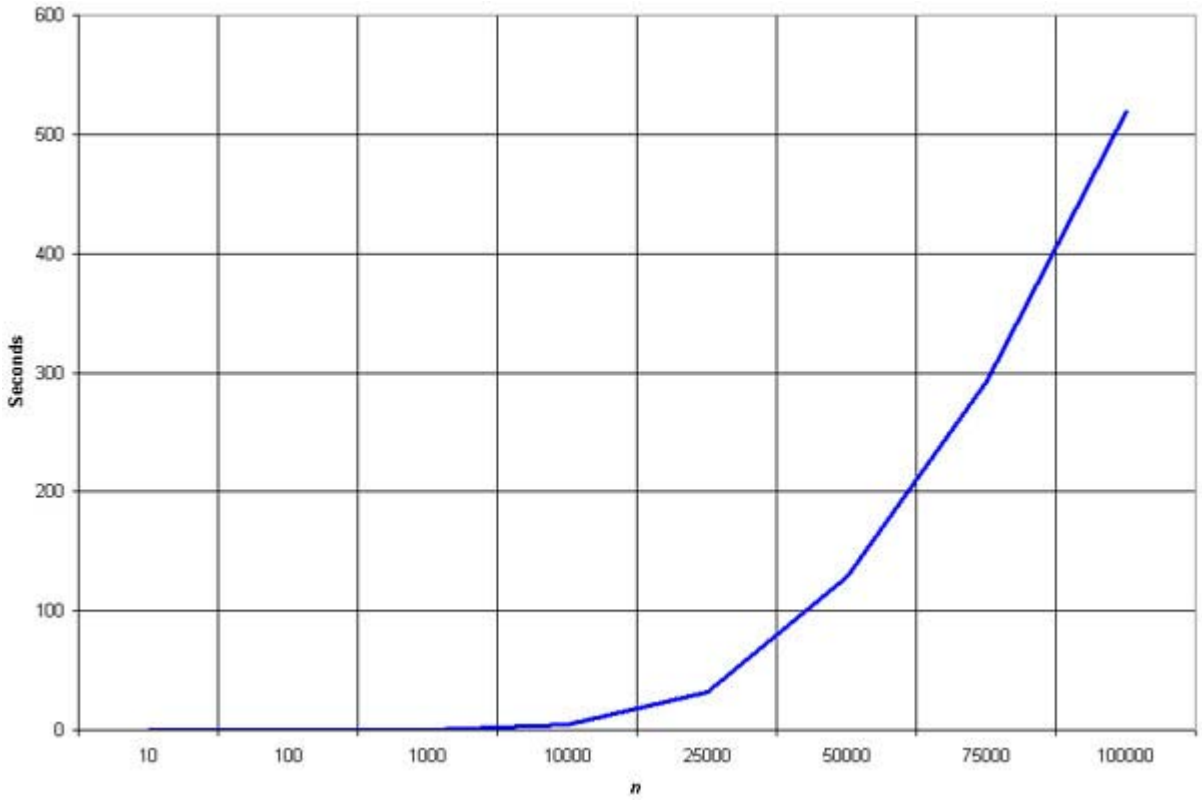
Algorithm Analysis

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.

Empirical Analysis

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

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;
  }
}
```

Shell Sort

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 $O(n)$.

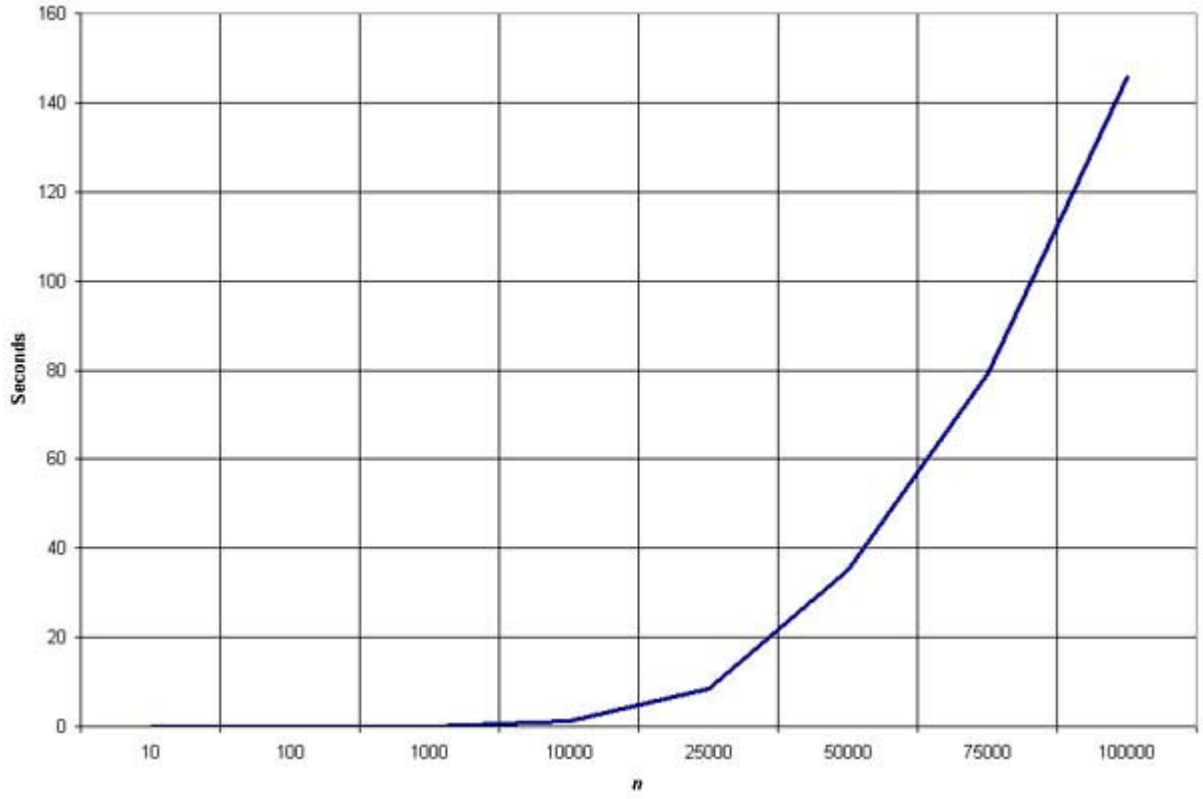
The items contained in each set are not contiguous - rather, if there are *i* sets then a set is composed of every *i*-th element. For example, if there are 3 sets then the first set would contain the elements located at positions 1, 4, 7 and so on. The second set would contain the elements located at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

Empirical Analysis

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.

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;
  }
}
```