Data Structures and Algorithms

Final Project

Sorting Algorithms

Asim Adnan Eijaz

2012077

Sorting

Putting an unsorted list of data elements into order- sorting is a very common useful operation entire books have been written about various sorting algorithms. The goal is to come up with a better, more efficient sorts. Because sorting a large number of elements can be extremely time-consuming, a good sorting algorithm is very desirable. This is one area in which programmers are sometimes encouraged to sacrifice clarity in favor of speed of execution.

How do we describe efficiency? We pick an operation central to most sorting algorithms: the operation that compares two values to see which is smaller. In our study of sorting algorithms, we relate the number of comparisons to the number of elements in the list (*N*) as a rough measure of the efficiency of each algorithm.

The number of swaps made is another measure of sorting efficiency.

Another efficiency consideration is the amount of memory space required. In general, memory space is not a very important factor in choosing a sorting algorithm. The usual time versus space tradeoff applies to sorts-more space often means less time, and vice versa.

Straight Selection Sort

If you were handed a list of names and asked to put them in alphabetical order, you might use this general approach:

1. Find the name that comes first in the alphabet, and write it on a second sheet of paper.
2. Cross the name out on the original list.
3. Continue this cycle until all names on the original list have been crossed out and written onto the second list, at which point the second list is sorted.

This algorithm is simple to translate into a computer program, but it has one drawback: It requires space in memory to store two complete lists. A slight adjustment to this manual approach does away with the need to duplicate space, however. As you cross a name off the original list, a free space opens up. Instead of writing the minimum value on a second list, you can exchange it with the value currently in the position where the crossed-off item should go. Our "by-hand list" is represented in an array.

Analyzing the Selection Sort

Measuring the amount of "work" required by this algorithm. We describe the number of comparisons as a function of the number of items in the array. To be concise, in this discussion we refer to numValues as *N*.

The comparison operation occurs in the function MinIndex. We know from the loop condition in the Selection Sort function that MinIndex is called *N* - 1 times. Within MinIndex, the number of comparisons varies, depending on the values of startIndex and endIndex:

In the first call to MinIndex, startIndex is 0 and endIndex is numValues - 1, so *N* - 1 comparisons occur; in the next call, *N* - 2 comparisons occur; and so on. In the last call, only one comparison takes place. The total number of comparisons is

*(N - 1) + (N - 2) + (N - 3) + ...+ 1 = N(N - 1)/2*

To accomplish our goal of sorting an array of *N* elements, the straight selection sort requires *N* (*N* - 1)/2 comparisons. Note that the particular arrangement of values in the array does not affect the amount of work done. Even if the array is in sorted order *before* the call to Selection Sort, the function still makes *N* (*N* - 1)/2 comparisons.

Bubble Sort

The bubble sort is a sort that finds the minimum (or maximum) value. Each iteration puts the smallest unsorted element into its correct place, but it also changes the locations of the other elements in the array.

The first iteration puts the smallest element in the array into the first array position. Starting with the last array element we compare successive pairs of elements, swapping them whenever the bottom element of the pair is smaller than the one above it. In this way, the smallest element "bubbles up" to the top of the array. The next iteration puts the smallest element in the unsorted part of the array into the second array position, using the same technique.

Analyzing the Bubble Sort

The comparisons occur in inner for loop, which is called ***N* - 1 times**. There are *N* - 1 comparisons the first time, ***N* - 2** comparisons the second time, and so on.

What purpose do these intermediate data swaps serve? By reversing out-of-order pairs of data as they are noticed, the function might get the array in order before *N* - 1 calls to Inner for loop. However, this version of the bubble sort makes no provision for stopping when the array is completely sorted. Even if the array is already in sorted order when Bubble Sort is called, this function continues to call inner for loop (which changes nothing) *N* - 1 times.

We could quit before the maximum number of iterations if inner for loop returns a Boolean flag, sorted, to tell us when the array is sorted. Within inner for loop, we initially set sorted to true; then in the loop, if any swaps occur, we reset sorted to false. If no elements have been swapped, we know that the array is already in order. Now the bubble sort needs to make only *one* extra call to inner for loop when the array is in order.

Insertion Sort

The principle of the insertion sort is quite simple: Each successive element in the array to be sorted is inserted into its proper place with respect to the other, already-sorted elements. As with the previous sorts, we divide our array into a sorted part and an unsorted part. Initially, the sorted portion contains only one element: the first element in the array. Now we take the second element in the array and put it into its correct place in the sorted part; that is, values [0] and values [1] are in order with respect to each other. Now the value in values [2] is put into its proper place, so values [0]. . Values [2] are in order with respect to each other. This process continues until all the elements have been sorted.

Analyzing the Insertion Sort

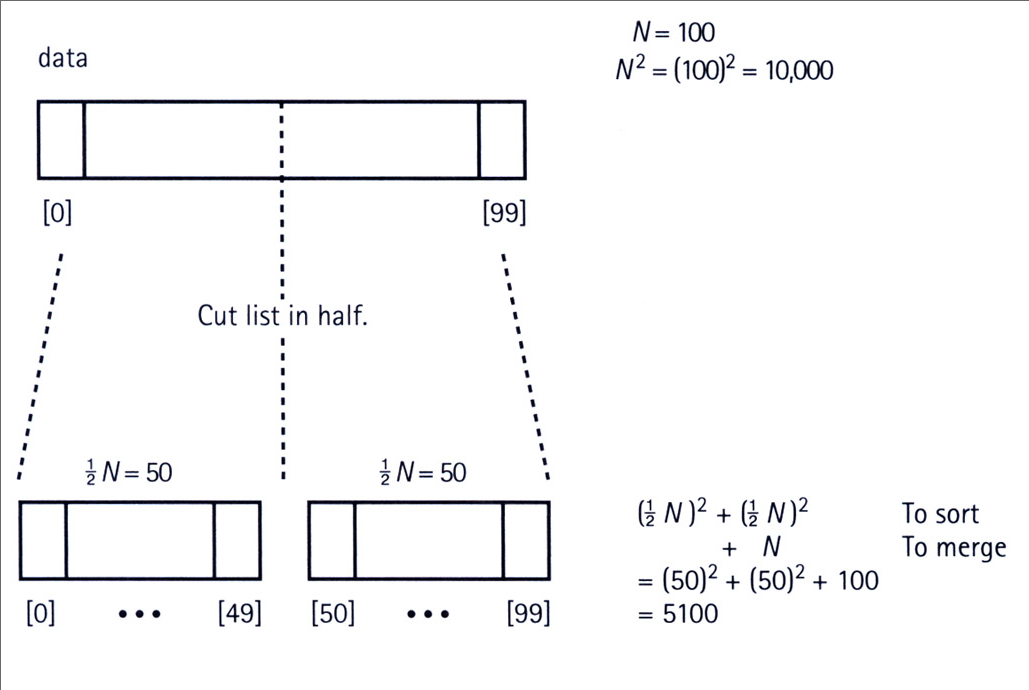
Insertion Sort has a best case: the data are already sorted in ascending order. When the data are in ascending order, InsertItem is called *N* times, but only one comparison is made each time and no swaps occur. The maximum number of comparisons is made only when the elements in the array are in reverse order.

If we know nothing about the original order in the data to be sorted, Selection Sort, Bubble Sort, and Insertion Sort are all O (*N*2) sorts and are too time-consuming for sorting large arrays. Thus we need sorting methods that work better when *N* is large.

**If we know nothing about the original order in the data to be sorted, Selection Sort, Bubble Sort, and Insertion Sort are all O (*N*2) sorts and are too time-consuming for sorting large arrays. Thus we need sorting methods that work better when *N* is large.**

O (N log2N) Sorts

Considering how rapidly *N*2 grows as the size of the array increases, can't we do better? We note that *N*2 is a lot larger than (1/2)2 + (1/2)2. If we could cut the array into two pieces, sort each segment, and then merge the two pieces again, we should end up sorting the entire array with a lot less work.



Merge Sort

Summarizing the Merge sort in simple words.

* Cut the array in half
* Sort the left half
* Sort the right half
* Merge the two sorted halves into one sorted array

Merging the two halves together is an O(*N*) task: We merely go through the sorted halves, comparing successive pairs of values (one in each half) and putting the smaller value into the next slot in the final solution. Even if the sorting algorithm used for each half is O (*N*2), we should see some improvement over sorting the whole array at once.

Actually, because Merge Sort is itself a sorting algorithm, we might as well use it to sort the two halves. We can make Merge Sort be a recursive function and let it call itself to sort each of the two sub arrays.

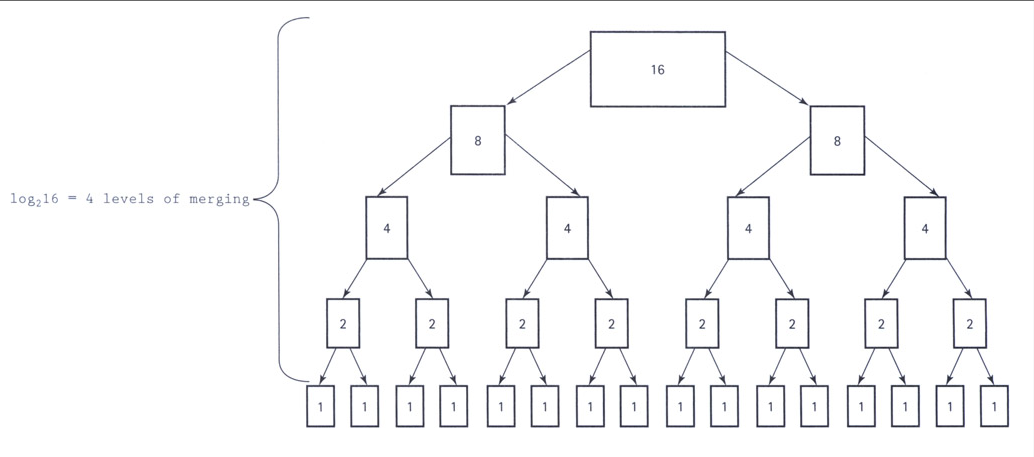
Recursive Merge Sort:

* Cut the array in half
* *Merge Sort* the left half
* *Merge Sort* the right half
* Merge the two sorted halves into one sorted array

Analyzing the Merge Sort

The Merge Sort function splits the original array into two halves. First it sorts the first half of the array, using the divide-and-conquer approach; then it sorts the second half of the array using the same approach; finally it merges the two halves. To sort the first half of the array it follows the same approach, splitting and merging. During the sorting process, the splitting and merging operations are intermingled. To simplify the analysis, however, we imagine that all of the splitting occurs first. We can view the process in this way without affecting the correctness of the algorithm.

We view the Merge Sort algorithm as continually dividing the original array (of size N) in two, until it has created N one-element sub arrays. Figure Below shows this point of view for an array with an original size of 16. The total work needed to divide the array in half, over and over again until we reach sub arrays of size 1, is O (N). After all, we end up with N sub arrays of size 1.



Each sub array of size 1 is obviously a sorted sub array. The real work of the algorithm comes in merging the smaller sorted sub arrays back into the larger sorted sub arrays. To merge two sorted sub arrays of size *X* and size *Y* into a single sorted sub array using the Merge operation requires O(*X* + *Y*) steps.

The original array of size *N* is eventually split into *N* sub arrays of size 1. Merging two of those sub arrays into a sub array of size 2 requires 0(1 + 1) = 0(2) steps based on the analysis given in the preceding paragraph. That is, we must perform this merge operation a total of 1/2 times (we have *N* one-element sub arrays and we are merging them two at a time). Thus the total number of steps to create all of the sorted two-element sub arrays is 0(*N*).

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **AverageCase** | **WorstCase** | **Advantage/Disadvantage** |
| Bubble Sort | O(n2) | O(n2) | * Straightforward, simple and slow. * Stable. * Inefficient on large tables. |
| Insertion Sort | O(n2) | O(n2) | * Efficient for small list and mostly sorted list. * Sort big array slowly. * Save memory. |
| Selection  Sort | O(n2) | O(n2) | * Improves the performance of bubble sort and also * slow. * Unstable but can be implemented as a stable sort. * Quite slow for large amount of data |
| Merge Sort | O(n log n) | O(n log n) | * Well for very large list, stable sort. * A fast recursive sorting. * Both useful for internal and external sorting. * It requires an auxiliary array that is as large as the * original array to be sorted. |