

Motivation

What is the importance of this problem in the world?

For an increasing problem size, the running time may take anywhere from hours to days, depending on an algorithm's efficiency with certain problem sizes and data compositions. For businesses, a larger amount of time spent usually means a larger amount of money spent, so it is in our best interest to minimize time with the goal of minimizing cost, and getting accurate sorting results in the least amount of time. Therefore it is good to be able to calculate the efficiency of each algorithm on a certain set of data, so the fastest algorithm can be chosen. The best sorting algorithm does as few moves and compares as possible. Algorithms perform with different efficiencies, depending on the problem size, so it is important that we are able to determine which algorithm performs most quickly and correctly on a particular data set without actually having to test it programmatically.

The Solution

The description of the solution is to take an unsorted array of data, such as integers, and put them into correct sorted order, ascending or descending, using an algorithm. Insertion sort is an incremental algorithm, sorting the sequence one number at a time. It works by repeatedly taking the next item in the array and inserting it into its proper place in the array, according to the items that have already been sorted. Merge sort is a divide-and-conquer algorithm invented by John von Neumann in 1945. It works by recursively dividing the array into halves down to the smallest subarray possible, then the two sorted subarrays are merged back into one sorted array.

Background Research

In the past, algorithms were written on punched cards, and the cards were fed to machine which read, processed, and executed the algorithm's instructions. As times became more modern, the machines processing the data became much faster, and human error was reduced since a human was no longer required to feed cards in the machine, that they could possibly feed in the wrong order. Now, sorting is done with computers containing microprocessors. Radix sort is among one of the fastest data sorts that exist. Radix sort's advantage is that sorting time as a function of the number of data is linear, while most other commonly used sorts are exponential, which is much slower for a large number of data.

Efficiency of Sorts

Insertion Sort:

–best case — $\theta(n)$

An array already sorted in ascending order

–worst case — $\theta(n^2)$

An array sorted in reverse (descending) order

-average case — $\theta(n^2)$

A randomly shuffled array

Merge Sort:

–best case — $\theta(n \lg n)$

–worst case — $\theta(n \lg n)$

-average case — $\theta(n \lg n)$

Pseudo code

Insertion Sort

//Precondition: A contains the values to be sorted.

//Postcondition: If A is empty no work is done. Otherwise, the elements of

//the array are sorted in ascending order.

```
InsertionSort(A)
//Invariant: At the start of each iteration of the for loop, the subarray
//A[1..j - 1] contains the elements originally in A[1..j - 1] in sorted
//order.
assert(length[A]>0);
for(c=1; c<j-1; c++)
    assert(A[c]<A[c+1]);
for j←2 to length[A]
    do key←A[j]
    i ←j-1
    //Invariant: A[i .. j] are each ≥ key
    for(b=i; b<=j; b++)
        assert(A[b]≥key);
    while i>0 and A[i]>key
        do A[i+1]←A[i]
        i←i-1
    A[i+1]←key
```

Merge Sort

```
// Precondition: A begins at p and ends at r
// Postcondition: A[p..r] is sorted
```

```
MergeSort(A,p,r)
if(p<r)
    then q←lower_stair_fuction[(p+r)/2] //Divide
        MergeSort(A,p,q) //Conquer
        MergeSort(A,q+1,r) //Conquer
        Merge(A,p,q,r) //See Merge code (Combine)
```

```
// Precondition: A[p..q-1]and A[q..r] are sorted; indices p, q, r such that
//p ≤ q < r
// Postcondition: A[p..r] is the merge of A[p..q-1] and A[q..r]
Merge(A,p,q,r)
n1←-- q -p +1
n2←-- q -p +1
create array L[1.... n1 +1] and R[1... n2 +1]
```

```
for i ← 1 to n1
    do L[i] ← A[p+i-1]
for j ← 1 to n2
    do R[j] ← A[q+j]
L[n1+1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
//Invariant: A[p...k-1] contains the k-p elements smallest in value of
//L[1 .. n1 + 1]
// and R[1 .. n2 + 1] in ascending order, L[i] and R[j] are the smallest
//values that haven't been copied back to A yet.
for(b=1; b < length[L]; b++)
    assert(A[b] ≤ A[b+1]);
for(c=1; c < length[R]; c++)
    assert(A[c] < A[c+1]);
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
            i ← i+1
        else A[k] ← R[j]
            j ← j+1
```