- Insertion Sort

Insertion Sort works by continually adding one element to an already sorted sublist until the length of the sorted sublist is equal to N. The new element to add starts at A[1] and iterates through A until A[N-1]:

for (i from 1 to N):

spot = i-1

while (spot >= 0 && A[spot+1] < A[spot]):

//perform swap

temp = A[spot]

A[spot] = A[spot+1]

A[spot+1] = temp

spot--

The outer loop grabs the next element to insert into the sorted sublist, and the inner loop finds that next elements appropriate position in the sublist. The inner while loop determines the runtime function’s behavior, which runs (N-1)\*ti times:

T(N) = (N-1) \* ti

At worst, such as in the case that the array is already in reverse order prior to running the sort, the inner loop will run **i** times:

T(N)worst = sum(1,N,i) = ½(N2 + N)

O(N) = N2

At best, such as in the case that the array is already in order prior to running the sort, the inner loop runs 1 time:

T(N)best = (N-1)\*1

Ω(N) = N

On average, one could assume that the inner loop would run halfway between **i** and 1, or roughly i/2 times:

T(N)avg = sum(1,N,1/2i) = ½\*½\*(N2 + N)

Ø(N) = N2

So the theoretical runtime of Insertion Sort on an array of length N should be on the order of N2. Looking at graph <IS graph>, one can see that the ratio test does confirm this theoretical runtime function, as the ratios follow a converging pattern to a constant as increasing N increases from 1,000 to 1,000,000.

Observing the runtimes across computers, two general trends can be noted: the difference between newer and older machines is not noticeable until problem sizes of around 10,000. In fact, in the case of the iMac G5 vs the ThinkPad L560, an older machine outperforms a newer one.

- Merge Sort

This sort is an educational favorite, at least because of it’s commonly recursive “divide-and-conquer” implementation and easiness to understand. It’s broken into two functions: the sorter and the merger. The sorter splits the array into two semiarrays and recursively calls itself until the array size is 1 (trivially sorted), and assigns the brunt of the sorting to the merger. The merger’s job is to combine to sorted subarrays into one; eventually, it is given two sorted halves of the original array and returns the whole original, sorted.

merge(start,end):

length = end-start

if (length > 1):

middle = floor((start+end) / 2)

for (i=start to end):

if (l < middle):

if (r >= end || A[l] <= A[r]):

B[i-start] = A[l]

l++

else if (r < end):

B[i-start] = A[r]

r++

else if (r < end):

B[i-start] = A[r]

r++;

for (i=start to end):

A[i] = B[i-start]

mergeSort(start,end):

if ((end-start) > 1):

middle = ceiling(avg(start,end))

if (end-start > 2):

mergeSort(start,middle)

mergeSort(middle,end)

merge(start,end)

Our implementation of merge sort differed slightly from the one we were taught in class, but its effect on the sort’s theoretical runtime was negligible. The first **for** loop in the merger runs through the section of the original array that defines the left and right sorted sublists and puts the output in a separate array. The second one simply puts the merged sublist back into that section of the original array. Since they are not nested, the runtime of the merger is:

Tmerge(N) = 2N = Ø(N)

Since the sorter divides the problem continually in half until the subarrays are each length 1, the number of subproblems is lg(n):

T(N) = 2T(N/2) + Tmerge(N)

T(N) = 4T(N/4) + 2(2N/2) + 2n = 4T(N/4) + 2N + 2N

T(N) = sum(i=1,lg(N),2N) = 2N \* lg(N)

Ø(N) = (N lg(N))

The best and worst cases for Merge Sort’s runtime function are on the same order as N lg N, since the inner loops run unconditionally according to their counter variables. Our tests show that the theoretical model of runtime for Merge Sort is valid: the ratio comparing measured and theoretical times for each problem size converges to a constant.

As was the case for both recursive N lg N sorts, the performance of the newer computers vs. older computers crossed twice. The newer ones ran faster with the lowest and highest problem sizes, while the older ones were able to keep pace in the N= 10,000 range.

- Hash Sort

Hash Sort ws an attempt to make a sort that could run faster than the others we’d been given in class. The basic concept was a modification of Insertion Sort that could “guess” where the next element should end up, thus diminishing the average runtime. To have this estimation capability, it first passes through the whole array to find its smallest and largest values and compute the range (this first pass could be exploited to find other statistics about the array too):

Tstats(N)= N

The second step is to calculate where the given element will likely end up in the sorted array, using a hash function:

dest = (element-minimum) / range \* N

Finally, the element is is either swapped with the one at **dest** (if that element wasn’t hashed already), or placed next to the one at **dest**. The latter case, being a collision, was the most difficult part of the algorithm to write.

while (i<n):

if (!hashed[i]):

dest = (a[i]-min) / range \* (n-1)

if (dest == i):

hashed[i] = true

i++

else if (!hashed[dest]):

swap (a[i],a[dest])

else:

temp = a[i]

if (i<dest):

//shift everything to the left and insert

j=i+1

while (j<dest && (!hashed[j] || a[j]<a[i])):

//perform shift

a[j-1] = a[j]

hashed[j-1] = hashed[j]

j++

if (a[i] <= a[j]):

//insert left

a[j-1] = temp

hashed[j-1] = true

else:

//insert right

a[j-1] = a[j]

hashed[j-1] = true

a[j] = temp

else:

//shift everything to the right and insert

j=i-1

while (j>dest && (!hashed[j] || a[j]>a[i])):

…shifts

…insertion

The problem which still hasn’t been fully solved, though I plan to solve it before submitting the final version of the paper, is that even with the current collision handling algorithm the output array still will contain a small number of errors. The algorithm’s most obvious weakness is strongly skewed data, where the hash function’s estimate is far from where each element should be in sorted order. Other potential edge cases account for the intermittent errors that the algorithm generates.

The loop invariant for Hash Sort is that the elements that have already been hashed in the array **are in sorted order relative to each other**. This condition was key for understanding how to handle hashing collisions. As of now, however, due to the flaws of the algorithm as it stands, we have Insertion Sort as a final step in the algorithm to walk through Hash Sort’s inaccurate output and fix misplaced elements.

THashSort(N) = Tstats(N) + Thash(N) + TInsertionSort(N)

THashSort(N) = N + Thash(N) + Ø(N2)

The runtime of the hashing portion is difficult to estimate theoretically. Essentially, at best the elements are already sorted or are evenly spaced in value and do not collide with each other when hashed, meaning the nested while loops are skipped:

Thash(N) = Ω(N)

At worst, the algorithm continually encounters collisions and has to shift the elements right or left to find where they should be in relation to the other, already hashed elements (square brackets indicate hashed elements):

6 [1] [7] [7] [7] 8 8 8 : 6 hashes but is blocked and has to shift left 3 spaces

We assume that collisions could at worst happen about 50% of the time, having to shift increasing numbers of times from 1 to N as the number of hashed elements increases:

Thash(N) ~ O(sum(i=n/2,n,i))

THashSort(N) = Ø(N) + Ω(N) + Ø(N2)

… which still makes HashSort approximately **Ø(N2),** removing lower order terms and coefficients.

Looking at the test results, this estimate of Hash Sort as having a runtime on the order of N2 seems accurate: the ratios between each measured time and associated theoretical time, with problem sizes between 1,000 and 1,000,000, do visibly converge to a constant.

Comparison of runtime between computers