- 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. A similar concept to which we were introduced recently is the idea of “counting sorts” as opposed to “comparison sorts”. 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**. However, in both cases, the position then has to be adjusted for skew of data values and collisions.

checkLeft(int start, int startValue) {

j=start

if (a[start] > startValue):

newDest = start

else:

newDest = -1

confirmed = false

while (!confirmed):

while (j>0 && !hashed[j]):

j--

if (a[j] > startValue):

if (hashed[j]):

newDest = j

if (j==0):

confirmed = true

else:

j--

else:

confirmed = true

checkRight(int start, int startValue) …

shiftLeft(int start, int end):

j=start+1

while (j <= end):

a[j-1] = a[j]

j++

shiftRight(int start, int end) …

hashSort():

getMinRange()

i = 0

while (i < n):

if (!hashed[i]):

dest = (int)(((double) (a[i] - min)) / range \* nMinus1)

if (dest != i && hashed[dest]):

if (a[i] < a[dest]):

checkLeft(dest,a[i])

if (i < newDest):

shiftLeft(i,newDest-1)

a[newDest-1] = a[i]

else:

shiftRight(i,newDest)

a[newDest] = a[i]

else if (a[i] > a[dest]):

checkRight(dest,a[i])

if (i > newDest):

shiftRight(i,newDest+1)

a[newDest+1] = a[i]

else:

shiftLeft(i,newDest)

a[newDest] = a[i]

else:

if (i < dest):

shiftLeft(i,dest-1)

a[dest-1] = a[i]

else:

shiftRight(i,dest+1)

a[dest+1] = a[i]

else:

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

checkLeft(dest, a[dest])

if (newDest != -1):

temp = a[dest]

for (j=dest-1 to newDest):

a[j+1] = a[j]

a[newDest] = temp

else:

checkRight(dest, a[dest])

if (newDest != -1):

temp = a[dest]

for (j=dest+1 to newDest):

a[j-1] = a[j]

a[newDest] = temp

else:

i++

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. Unfortunately, with the current Hash Sort algorithm, the use of many potentially O(N) procedures is inevitable to account for these collisions. Firstly, swapping and shifting in place costs extra time. Secondly, since hashed elements can move around in the array, though not relative to each other, linear search is required, which is inherently a O(N) algorithm.

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

THashSort(N) = N + N\*Tsearch(i)\*Tshift(i)

The runtime of the shifting 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:

Tshift(N) = Ω(1)

However, the current algorithm doesn’t allow for this to be exploited, because it checks for surrounding hashed elements regardless of collisions. Below illustrates the shifting process, where [ and ] indicate a hashed element.

6 [1] [7] 9 8 [7] 8 8… : 6 hashes and swaps but has to shift left 3 spaces

8 [1] [7] 9 6 [7] 8 8…

8 [1] [7] 6 9 [7] 8 8…

8 [1] 6 [7] 9 [7] 8 8…

We assume that at worst the elements would have to be shifted about 50% of the time, having to shift on average about N/2 positions each time. Any worse is likely impossible given that the hashing should do a pretty good job of placing the elements, even in skewed data. We also assume that the search would iterate through the list about N/2 positions, going either right or left of the hashed element:

THashSort(N) = Ø(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.

Looking at the graph of the ratio test for runtime across machines, the trends are similar to those of the other N2 sorts; the differences are steadily more apparent as the problem size increases, though it is worth noting that the runtimes for Hash Sort were closer together across machines than those of Insertion and Selection.