- 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)

Derivation of theoretical runtimes

Comparison between theoretical and measured times

Comparison of runtime between computers

- Hash Sort

Description

Pseudocode

Theoretical runtimes

Comparison between theoretical and measured times

Comparison of runtime between computers

- C++ Array Fill

See picture of paragraph draft

- Other topics…