“Average-Case” Analysis

Expected Runtimes

Insertion Sort

Quicksort

Insertion Sort

Since insertion sort has a while loop inside, for the worst-case analysis of data comparisons we just assume the iterator is what makes it stop.

```
InsertionSort(L) {
  for pos = 2 to L.length {
      val = L[pos];
      iter = pos-1;
      while (iter<>0) and (L[iter]>val) {
          L[iter+1]=L[iter];
          iter--;
      }
      L[iter+1]=val;
  }
}
```

Insertion Sort

For best-case data comparison analysis we have the while loop terminate on its first data comparison.

For each outer loop (sum as i goes from 2 to n) the inner loop's total data comparisons done is 1…

Insertion Sort

For average-case comparison analysis we need to consider all possible ways the while loop might terminate.

For each outer loop (sum as i goes from 2 to n) the inner loop's total iterations can be between 1 and i-1 so we can determine the expected value of that to get an average-case runtime…

QuickSort Recap

This is another example of a pure "divide and conquer" algorithm.

<u>Step 1 (divide)</u>

Select a "pivot" value and logically partition the list into two sub-lists:

L1: values less than the pivot

L2: values greater than the pivot

Your list is now: L1,pivot,L2

<u>Step 2 (conquer)</u>

Sort L1 and L2

SORTED!

QuickSort Pseudocode

<u>Algorithm</u>

Let's assume that out list L is held in an array and that we want to use as little extra space as possible.

```
QuickSort(array L, int first, int last) {
   if (first<last) {
         pivotpos = Partition(L,first, last)
         QuickSort(L, first, pivotpos-1)
         QuickSort(L,pivotpos+1,last);
   }
}
```

NOTE: We would still need to write the **Partition** algorithm. The easiest thing to code would probably be to pick the last value in the list as the pivot and then partition based on that.

Partition's runtime…

There are many ways to implement the partition algorithm, but in terms of the number of data comparisons, it should be accomplished using n-1.

QuickSort's runtime…

Start with $\mathbf{T(0) = T(1) = 0}$

For the recurrence relation we want to consider three cases:

- With the worst case split.

  $$\mathbf{T(n) = (n-1) + T(0) + T(n-1)}$$

- With the best case split.

  $$\mathbf{T(n) = (n-1) + T(n/2-1) + T(n/2)}$$

- With the average/expected split….

Average Case Analysis

We return to the idea of expected values…

Let's assume that every "division situation" around the pivot is equally likely.

If we let ***i*** represent the position where L2 starts, then we could represent the expected runtime as being:

$$T(n) = (n-1) + \frac{\sum_{i=1}^{n} [T(i-1) + T(n-i)]}{n}$$

What about that worst case?

Recall that regardless of the "average" case, that if we expect mostly-sorted inputs, then the runtime will be bad.

How could we alter our approach to try to address (ie: decrease the likelihood of) the issue of sorted lists leading to n^2 runtime with the pivot/partitioning algorithm that I originally presented?