## Recurrence Formula for Selection Sort:

**C(n) = # times compare two values in the array for problem of size n**. Start with two definitions

We know C(1) is 0 because when called on an array of 1 single element, there is no call to compare two values in the array. Let’s telescope the equation two times. By definition,

And substitute this into first equation, and you get:

One more time, expand C(N-2) = C(N-3) + (N-3) and insert into the original equation, and you have:

Rearrange terms to get:

If we do this telescoping replacement k times, this gives us:

Where is the triangular numbers I’ve mentioned in class. Note that .

How many times can you telescope? Well, you will be able to go k=N-1 times, until you get to C(1), which we have already defined above to be 0. So, when k=N-1, this equation is:

Or finally,

This is both the lower bound and the upper bound.

## Recurrence Formula for Binary Array Search:

**C(n) = # times compare target with an element from array.** Start with two definitions

We know C(1) is 1 because when rank is called on a problem of size 1, there is a single comparison. Now look at expanding first sub-case

Inserting back into the original equation, you get:

And again, one more time to get:

If we do this telescoping replacement k times, this gives us:

Now, how many times can this be subdivided? Exactly times. In which case you get:

And since C(1) is 1, this gives us, in the worst case:

To understand the best case, observe that with the very first invocation, you could stop (after being exceptionally lucky) in finding the element at the first pass. Thus in the best case, C(N) = 1.

## Recurrence Formula for Merge Search:

Start with the following two definitions for the number of comparisons of two elements in the array

in the best case

in the worst case

We know C(1) is 1 because when sort is called on a[lo..hi] and lo=hi, no comparisons occur. We start using worst case analysis, and then move on to best case.

Inserting back into the original equation for worst case, you get:

And again, one more time to get:

If we do this telescoping replacement k times, this gives us:

In the general case, you subdivide k times, and this results in:

And since C(1) is 0, this gives us, in the worst case:

How many times can N be subdivided by 2? k = log N times. So the resulting answer is:

In best case, we redesign the original formula to handle the Best Case scenarios:

The first sub-division would lead to formulae of the following:

Substituting in you have:

This expands to:

Or in the general *k* case, this is:

Once again, we can subdivide k=log N times, until you get to C(1) = 0