How Fast Can We Sort?

It is hard to do something we would call sorting without looking at the data, so a lower bound for sorting n data elements is $\Omega(n)$.

We can even achieve this lower bound. Suppose we have a list of n numbers (where n is very large) that are all between 0 and 9. Take an array Counts of 10 entries, all initialized to 0 and iterate through the numbers. Each time you see a 6 increment Counts[6]; each time you see a 9 increment Counts[9], and so forth. Then replace the original data by Counts[0] 0's, Counts[1] 1's and so forth in that order:

```
// This assumes every element of A is between 0 and 9.
public void Sort(int[] A) {
        int[] Counts = new int[10];

        for (int i = 0; i < A.length; i++)
                Counts[ A[i] ] += 1;
        p = 0;
        for (int j = 0; j <= 9; j++ ) {
                for (int k = 0; k < Counts[j] ) {
                        A[p] = j;
                        p += 1;
                }
        }
}
```

This is certainly order n -- it consists of two passes that each look at each data item once.

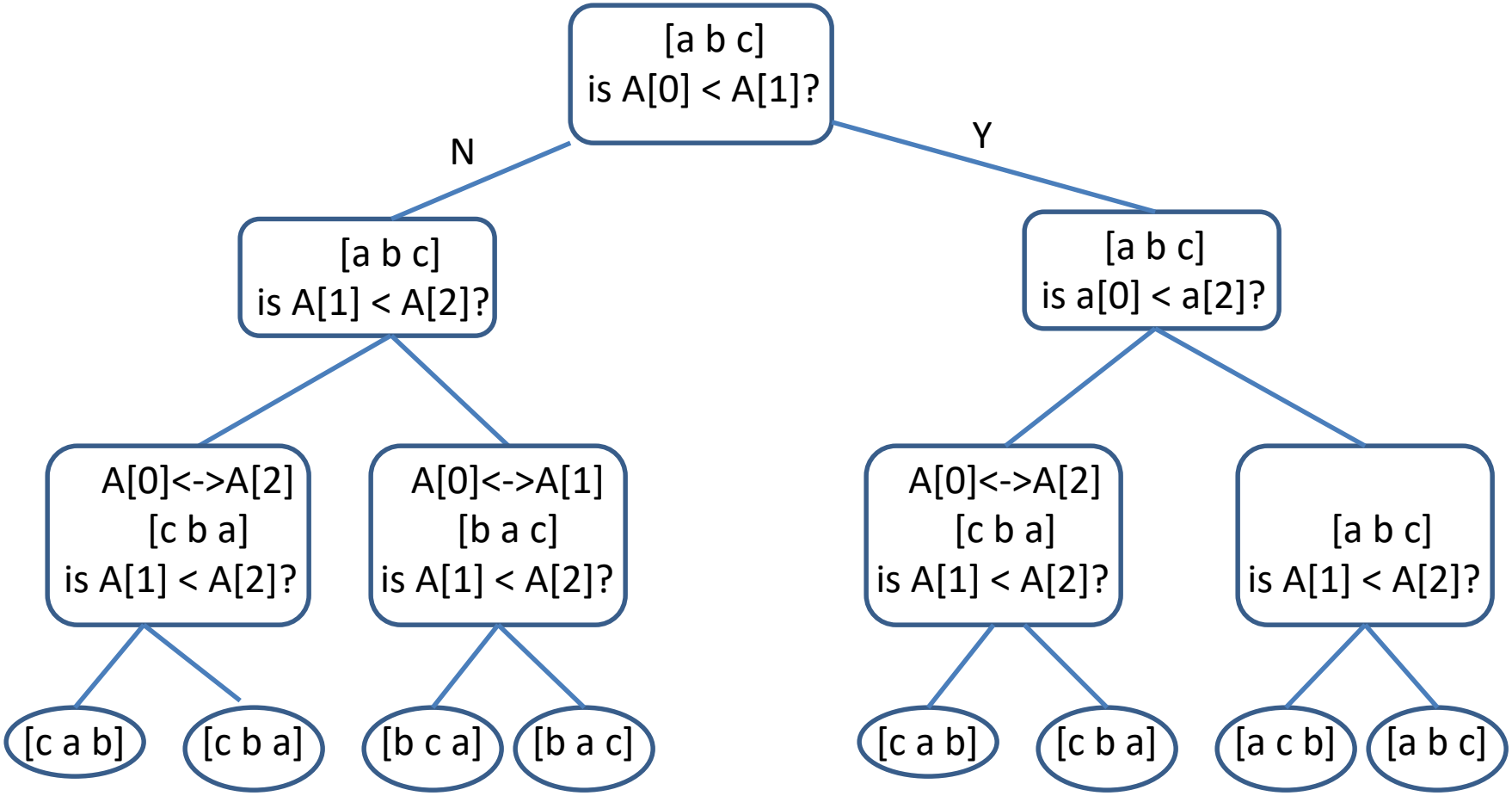
This algorithm goes by various names; many people call it BucketSort. But somehow it seems like cheating; this isn't what we usually mean by sorting.

More typically, we are interested in sorting algorithms that work by comparing the data elements; if you don't know anything in advance about the data this is your only option.

We will show that a lower bound for how many comparisons such algorithms make is $\Omega(n*\log(n))$.

Although we wouldn't code it this way, we can think of any algorithm that sorts by comparing data elements as asking a sequence of questions that compare different elements, sometimes making assignments that interchange elements. Which elements get compared depends on the answers to previous questions, so this forms a *Decision Tree*.

For example, here is a decision tree for SelectionSort for sorting a list of 3 items. Branches to the left reflect NO decisions on the previous questions; branches to the right reflect YES.



Here A[0]<->A[1] means interchange A[0] and A[1].

Note that the decision tree must have at least n! leaves since there are n! different orderings of n elements and there must be at least one leaf for each possible ordering.

It is easy to see that

Theorem: A binary tree of height h can have no more than 2^h leaves.

If you stand that on its head it says

Theorem: A binary tree with n leaves must have height at least log(n).

Since our decision trees have n! leaves and their heights are the maximum number of comparisons needed to sort any particular ordering of the data we can say that the sorting algorithms all do at least log(n!) comparison.

So how big is log(n!)??

log(n!) = log(n) + log(n-1) + log(n-2) + ... + log(1)

The first n/2 terms: log(n), log(n-1) etc. are all >= log(n/2).

So

log(n!) >= (n/2)*log(n/2)

= (n/2)*[log(n) - 1]

= (n/2)*log(n) - n/2

= Θ(n*log(n))

Altogether, we can conclude that:

Any algorithm that sorts by comparing data elements has to do at least $\Omega(n*\log(n))$ comparisons and that a lower bound for its running time is $\Omega(n*\log(n))$.

This semester we have talked about BubbleSort, SelectionSort, and InsertionSort, which are all $O(n^2)$, and MergeSort, QuickSort and HeapSort, which are all $O(n*\log(n))$. We know there is no algorithm with a smaller order of growth.

There are a few other sorting algorithms. A famous one is called ShellSort after its inventor, Donald Shell; it runs slightly faster than QuickSort on real-world data. Most people feel that QuickSort is good enough for almost any situation.