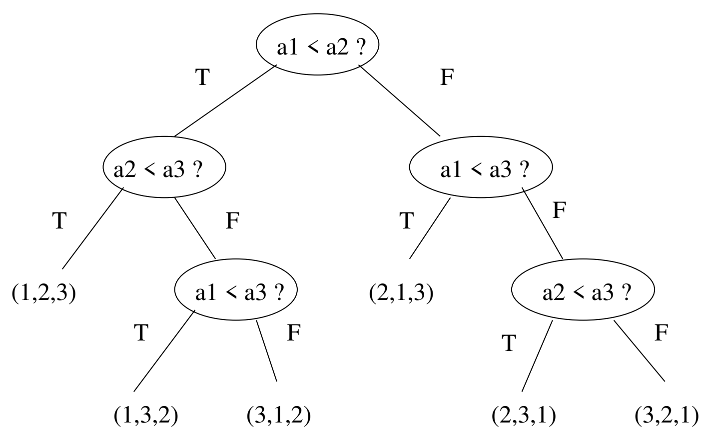
Can we sort in better than $n \lg n$?

Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.

Running the same program twice on the same permutation causes it to do exactly the same thing, but running it on different permutations of the same data causes a different sequence of comparisons to be made on each.



Claim: the height of this decision tree is the worst-case complexity of sorting.

Once you believe this, a lower bound on the time complexity of sorting follows easily.

Since any two different permutations of n elements requires a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree, ie. at least $n!$ different leaves in the tree.

Since only binary comparisons (less than or greater than) are used, the decision tree is a binary tree.

Since a binary tree of height h has at most 2^h leaves, we know $n! \leq 2^h$, or $h \geq \lg(n!)$.

By inspection $n! > (n/2)^{n/2}$, since the last $n/2$ terms of the product are each greater than $n/2$. By Sterling's approximation, a better bound is $n! > (n/e)^n$ where $e = 2.718$.

$$h \geq \lg(n/e)^n = n \lg n - n \lg e = \Omega(n \lg n)$$

Non-Comparison-Based Sorting

All the sorting algorithms we have seen assume binary comparisons as the basic primative, questions of the form “is x before y?”.

Suppose you were given a deck of playing cards to sort. Most likely you would set up 13 piles and put all cards with the same number in one pile.

```
A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K
A 2 3 4 5 6 7 8 9 10 J Q K
```

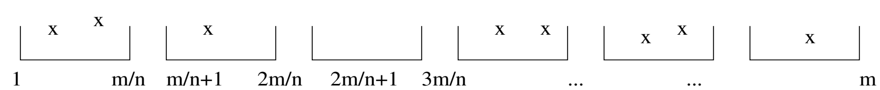
With only a constant number of cards left in each pile, you can use insertion sort to order by suite and concatenate everything together.

If we could find the correct pile for each card in constant time, and each pile gets $O(1)$ cards, this algorithm takes $O(n)$ time.

Bucketsort

Suppose we are sorting n numbers from 1 to m, where we know the numbers are approximately uniformly distributed.

We can set up n buckets, each responsible for an interval of m/n numbers from 1 to m



Given an input number x, it belongs in bucket number $\lceil xn/m \rceil$.

If we use an array of buckets, each item gets mapped to the right bucket in $O(1)$ time.

With uniformly distributed keys, the expected number of items per bucket is 1. Thus sorting each bucket takes $O(1)$ time!

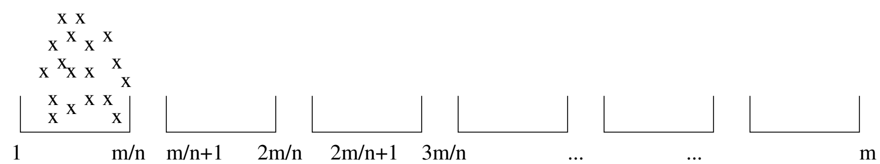
The total effort of bucketing, sorting buckets, and concatenating the sorted buckets together is $O(n)$.

What happened to our $\Omega(n \lg n)$ lower bound!

We can use bucketsort effectively whenever we understand the distribution of the data.

However, bad things happen when we assume the wrong distribution.

Suppose in the previous example all the keys happened to be 1. After the bucketing phase, we have:



We spent linear time distributing our items into buckets and learned *nothing*. Perhaps we could split the big bucket recursively, but it is not certain that we will ever win unless we understand the distribution.

Problems like this are why we worry about the worst-case performance of algorithms!

Such distribution techniques can be used on strings instead of just numbers. The buckets will correspond to letter ranges instead of just number ranges.

The worst case "shouldn't" happen if we understand the distribution of our data.

Real World Distributions

Consider the distribution of names in a telephone book.

- Will there be a lot of Skiena's?
- Will there be a lot of Smith's?
- Will there be a lot of Shifflitt's?

Either make *sure* you understand your data, or use a good worst-case or randomized algorithm!

The Shifflett's of Charlottesville

For comparison, note that there are seven Shifflett's (of various spellings) in the 1000 page Manhattan telephone directory.