What is sorting?

- Input: sequence of n comparable values
- Reorder the input to be non-descending.
- Items we wish to sort are called "keys"
- Not here: retain associated information

Why discuss sorting?

- Standard library has sorting
- Why not use that and move on?

In this class, sorting is:

- a good intro for techniques
- a good intro to comparative algorithms

SelectionSort

Idea: Swap min into first spot, second-min to second, etc.

```
**for** $i \leftarrow 1$ to $n - 1$ **do**
  min $\leftarrow i$
  **for** $j \leftarrow i + 1$ to $n$ **do**
    **if** $A[j] < A[\text{min}]$ **then**
      min $\leftarrow j$
    **end if**
  **end for**
  Swap $A[i]$ and $A[\text{min}]$
**end for**
```

85	24	63	45	17	31	96	50

Let's talk about SelectionSort.

- Does it waste memory?
- Does it only work for numbers?
- What other info do we need?
- Are there inputs that are sorted faster?
- Is there a lot of data movement?

Bubble Sort

Idea: Think globally act locally

```
for $i \leftarrow 1$ to $n-1$ do
  for $j \leftarrow 1$ to $n-i$ do
    if $A[j+1] < A[j]$ then
      Swap $A[j]$ and $A[j+1]$
    end if
  end for
end for
```

85	24	63	45	17	31	96	50

InsertionSort

```
for $j \leftarrow 2$ to $n$ do
  key $\leftarrow A[j]$
  $i \leftarrow j-1$
  while $i > 0$ and $A[i]$ >key do
    $A[i+1] \leftarrow A[i]$
    $i = i-1$
  end while
  $A[i+1] \leftarrow$ key
end for
```

85	24	63	45	17	31	96	50

- What is the worst-case running time of InsertionSort?
- Why is InsertionSort correct?
- What is true *every time* we check the **for** loop?
  (including the time we find $j > n$ and stop)

HeapSort

Idea 1: Insert all n elements into an (initially empty) max heap. Then, repeatedly extract and place the maximum element from the heap into the last spot of the vector into which we have yet to place.

How long does this take?

Idea 2: Bottom-up heap construction. We know which locations will be leaf nodes.

4	1	3	2	16	9	10	14	8	7

Question: Once we have the array turned into a max-heap, what do we do? Where do you place the result of a `remove-max` operation?

MergeSort

Associated reading: §11.1, Goodrich/Tamassia text.

Suppose we have two sorted lists. How do we combine them into a single sorted list?

List 1:

41	48	51	66	68	84	87	89

List 2:

14	23	37	43	52	62	96	98

We don't always start with two sorted lists, though. How can we make use of the previous part to sort?

48	41	51	66	84	89	87	68	37	23	96	98	52	14	62	43

QuickSort

Associated reading: §11.2, Goodrich/Tamassia text.

MergeSort is a paradigm of algorithm called *Divide and Conquer*. You can imagine what two main things such an algorithm does. In MergeSort, the divide step happens first and the results of dividing are then combined (conquered).

QuickSort goes the other direction: first we're going to divide the array into two parts, hopefully halves. We then recursively sort each half. The key operation is `partition`, which is based on a *pivot*. Different flavors of QuickSort differ primarily on how they select the pivot.

48	41	51	66	84	89	87	68	37	23	96	98	52	14	62	43

Selecting pivots for QuickSort

Let's discuss some common ways to select a pivot for QuickSort:

- Deterministic, single-spot. For example, always select the first element in the array.
- Randomized. Select uniformly at random.
- Median-of-3.

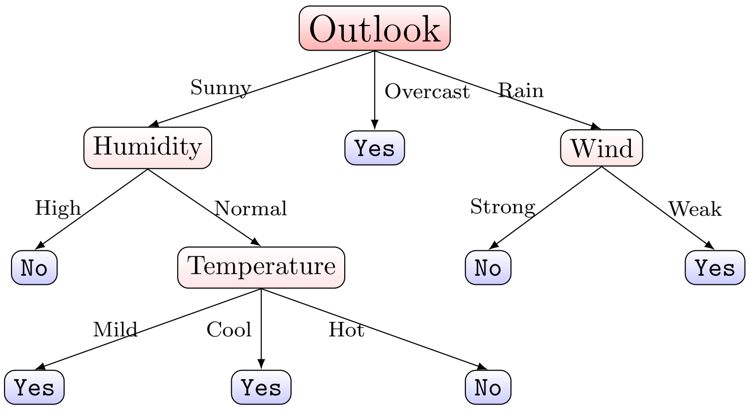
Question: Suppose I am going to sort a vector of n comparable objects by QuickSort. All $n!$ initial permutations are equally likely. What is the expected average case running time for the algorithm? For simplicity, assume all n keys are distinct.

Let $P_{i,j}$ be probability we compare S_i and S_j. What is $P_{i,j}$.

Let $X_{i,j}$ be an indicator random variable for whether or not i, j get compared. Because that is a binary outcome, $X_{i,j} = 1 \cdot P_{i,j} + 0 \cdot (1 - P_{i,j}) = P_{i,j}$

$$\begin{aligned}
E\big(\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{i,j}\big) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} E(X_{i,j}) \\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n}\sum_{k=2}^{n-i+1} \frac{2}{k} \\
&< \sum_{i=1}^{n}\sum_{k=1}^{n} \frac{2}{k}
\end{aligned}$$

Aside: Decision Trees



Draw a decision tree to determine the smallest from three distinct keys, x, y, and z.

Lower Bound for Sorting

We have seen algorithms that take $\mathcal{O}(n^2)$ time. We saw algorithms take $\mathcal{O}(n \log n)$ time.

Are there algorithms which are strictly better than $\mathcal{O}(n \log n)$ time for a general comparison-based sort?

To answer this question, we will build a **decision tree** that represents any comparison based sorting algorithm. Such an algorithm will ask questions of the form "is $x_i < x_j$?"

- What are leaf nodes of decision tree?
- What are internal nodes?
- What is height of the tree?
- What does this tell us about any such algorithm?

Many years ago, when he was a student, Professor Shindler worked for Middle Earth housing, here at UCI. One year, as we were preparing for move-in, a co-worker dropped a basket of key cards. Each was a small envelope with incoming residents' names listed on them, one per envelope. The students were going to be arriving in less than half an hour, and we needed to put the cards back in sorted order. Did we run QuickSort?