Why don't CS profs ever stop talking about sorting?!

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

2. Sorting is the best studied problem in computer science, with a variety of different algorithms known.

3. Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms - we will concentrate on the analysis.

Applications of Sorting

One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.

Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.

Speeding up searching is perhaps the most important application of sorting.

Closest pair

Given n numbers, find the pair which are closest to each other.

Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Element uniqueness

Given a set of n items, are they all unique or are there any duplicates?

Sort them and do a linear scan to check all adjacent pairs.

This is a special case of closest pair above.

Frequency distribution – Mode

Given a set of n items, which element occurs the largest number of times?

Sort them and do a linear scan to measure the length of all adjacent runs.

Median and Selection

What is the kth largest item in the set?

Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.

Binary Heaps

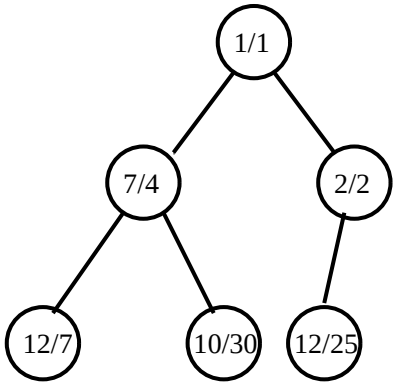
A *binary heap* is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.

2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

3. The key in root is $\geq$ all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.



Constructing Heaps

Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.

If the new element is greater than its parent, swap their positions and recur.

Since at each step, we replace the root of a subtree by a larger one, we preserve the heap order.

Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.

Doing n such insertions takes $\Theta(n \log n)$, since the last $n/2$ insertions require $O(\log n)$ time each.

Heapify

The bottom up insertion algorithm gives a good way to build a heap, but Robert Floyd found a better way, using a *merge* procedure called *heapify*.

Given two heaps and a fresh element, they can be merged into one by making the new one the root and trickling down.

```
Build-heap(A)
        n = |A|
        For i = ⌊n/2⌋ to 1 do
                Heapify(A,i)


Heapify(A,i)
        left = 2i
        right = 2i + 1
        if (left ≤ n) and (A[left] > A[i]) then
                max = left
                else max = i
        if (right ≤ n) and (A(right] > A[max]) then
                max = right
        if (max ≠ i) then
                swap(A[i],A[max])
                Heapify(A,max)
```

Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

```
Heapsort(A)
        Build-heap(A)
        for i = n to 1 do
                swap(A[1],A[i])
                n = n - 1
                Heapify(A,1)
```

If we construct our heap from bottom to top using Heapify, we do not have to do anything with the last $n/2$ elements.

With the implicit tree defined by array positions, (i.e. the ith position is the parent of the $2i$th and $(2i+1)$st positions) the leaves start out as heaps.

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm, named *Heapsort*.

The Lessons of Heapsort, II

Always ask yourself, "Can we use a different data structure?"

Selection sort scans throught the entire array, repeatedly finding the smallest remaining element.

For $i = 1$ to n
A: Find the smallest of the first $n - i + 1$ items.
B: Pull it out of the array and put it first.

Using arrays or unsorted linked lists as the data structure, operation A takes $O(n)$ time and operation B takes $O(1)$.

Using heaps, both of these operations can be done within $O(\lg n)$ time, balancing the work and achieving a better tradeoff.

Priority Queues

A *priority queue* is a data structure on sets of keys supporting the following operations:

- *Insert(S, x)* - insert x into set S
- *Maximum(S)* - return the largest key in S
- *ExtractMax(S)* - return and remove the largest key in S

These operations can be easily supported using a heap.

- *Insert* - use the trickle up insertion in $O(\log n)$.
- *Maximum* - read the first element in the array in $O(1)$.
- *Extract-Max* - delete first element, replace it with the last, decrement the element counter, then heapify in $O(\log n)$.

Greedy Algorithms

In greedy algorithms, we always pick the next thing which locally maximizes our score. By placing all the things in a priority queue and pulling them off in order, we can improve performance over linear search or sorting, particularly if the weights change.

Example: Sequential strips in triangulations.