Why do we study sorting?
- Because we have to
- Because sorting is beautiful
- Example of algorithm analysis in a simple, useful setting

There are $n$ sorting algorithms, how many should we study?
- $O(n)$, $O(\log n)$, ...
- Why do we study more than one algorithm?
  - Some are good, some are bad, some are very, very sad
  - Paradigms of trade-offs and algorithmic design
- Which sorting algorithm is best?
- Which sort should you call from code you write?
Sorting out sorts

- **Simple, quadratic sorts --- for sorting n elements**
  - Selection sort --- $n^2$ comparisons, $n$ swaps, easy to code
  - Insertion sort --- $n^2$ comparisons, $n^2$ moves, stable, fast
  - Bubble sort --- $n^2$ everything, slow, slower, and ugly

- **Divide and conquer faster sorts: $n \log n$ work for $n$ elements**
  - Quick sort: fast in practice, quadratic worst case
  - Merge sort: good worst case, uses extra storage for arrays

- **Other sorts:**
  - Heap sort, priority queue sorting – $n \log n$
  - Shell sort: quasi-insertion, fast in practice: $n^{1.5}$ comparisons
Selection sort: summary

- Simple to code $n^2$ sort: $n^2$ comparisons, $n$ swaps

```java
void selectSort(String[] a) {
    int len = a.length;
    for(int k=0; k < len; k++){
        int mindex = getMinIndex(a,k,len);
        swap(a,k,mindex);
    }
}
```

- # comparisons: $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = O(n^2)$
  - Swaps?
  - Invariant:

```
| Sorted, won’t move final position | ????
```

CompSci 6
Insertion Sort: summary

- **Stable sort, \(O(n^2)\), good on nearly sorted vectors**
  - Stable sorts maintain order of equal keys
  - Good for sorting on two criteria: name, then age

```java
void insertSort(String[] a){
    int k, loc; String elt;
    for(k=1; k < a.length; ++k) {
        elt = a[k];
        loc = k;
        // shift until spot for elt is found
        while (0 < loc && elt.compareTo(a[loc-1]) < 0) {
            a[loc] = a[loc-1];  // shift right
            loc=loc-1;
        }
        a[loc] = elt;
    }
}
```

| Sorted relative to each other | ?????
Bubble sort: summary of a dog

- For completeness you should know about this sort
  - Really, really slow (to run), really really fast (to code)
  - Can code to recognize already sorted vector (see insertion)
    - Not worth it for bubble sort, much slower than insertion

```java
void bubbleSort(String[] a) {
    for(int j=a.length-1; j >= 0; j--) {
        for(int k=0; k < j; k++) {
            if (a[k].compareTo(a[k+1]) > 0)
                swap(a,k,k+1);
        }
    }
}
```

- “bubble” elements down the vector/array
Summary of simple sorts

- **Selection sort** has n swaps, good for "heavy" data
  - moving objects with lots of state, e.g., …
    - In C or C++ this is an issue
    - In Java everything is a pointer/reference, so swapping is fast since it's pointer assignment

- **Insertion sort** is good on nearly sorted data, it’s stable, it’s fast
  - Also foundation for Shell sort, very fast non-recursive
  - More complicated to code, but relatively simple, and fast

- **Bubble sort** is a travesty? But it's fast to code if you know it!
  - Can be parallelized, but on one machine don’t go near it
    (see quotes at end of slides)
Quicksort: fast in practice

- Invented in 1962 by C.A.R. Hoare, didn’t understand recursion
  - Worst case is $O(n^2)$, but avoidable in nearly all cases
  - In 1997 Introsort published (Musser, introspective sort)
    - Like quicksort in practice, but recognizes when it will be bad and changes to heapsort

```java
void quick(String[], int left, int right)
{
    if (left < right) {
        int pivot = partition(a,left,right);
        quick(a,left,pivot-1);
        quick(a,pivot+1, right);
    }
}
```

- Recurrence?
Partition code for quicksort

what we want

<table>
<thead>
<tr>
<th>&lt;= pivot</th>
<th>&gt; pivot</th>
</tr>
</thead>
</table>

left  right

pIndex

what we have

<table>
<thead>
<tr>
<th>???</th>
<th>???</th>
</tr>
</thead>
</table>

left  right

pIndex  k

invariant

- Easy to develop partition

```java
int partition(String[] a, int left, int right)
{
    String pivot = a[left];
    int k, pIndex = left;
    for(k=left+1; k <= right; k++) {
        if (a[k].compareTo(pivot) <= 0){
            pIndex++;
            swap(a,k,pIndex);
        }
    }
    swap(a,left,pIndex);
}
```

- loop invariant:
  - statement true each time loop test is evaluated, used to verify correctness of loop
  - Can swap into a[left] before loop
    - Nearly sorted data still ok
Merge sort

- Divide and conquer --- recursive sort
  - Divide list/vector into two halves
    - Sort each half
    - Merge sorted halves together

```java
void mergesort(String[] a, int left, int right)
{
    if (left < right) {
        int mid = (right+left)/2;
        mergesort(a, left, mid);
        mergesort(a, mid+1, right);
        merge(a, left, mid, right);
    }
}
```

- How does merge work? How long does it take?
Merge for arrays

- Array code for merge isn’t pretty, but it’s not hard
  - Mergesort itself is elegant

```java
void merge(String[] a,
           int left, int middle, int right)
// pre:  left <= middle <= right,
//       a[left] <= … <= a[middle],
//       a[middle+1] <= … <= a[right]
// post: a[left] <= … <= a[right]
```

- Need extra storage, can't easily merge in place
  - Can alternate between arrays: one merged into, then swap
Heap Sort

- Uses a heap (java.util.PriorityQueue) successively find minimum element.

- Operations logarithmic in terms of number of elements in heap
  - Insert element into heap
  - Remove minimum element

- Fast, not stable, may use extra data structure

```java
void heapSort(String[] a){
    PriorityQueue<String> pq = new PriorityQueue<String>();
    // Add elements to priority queue
    for (String elem: a)
        pq.add(elem);
    // Take elements from priority queue in order
    for (int k=0; k < a.length; k++)
        a[k] = pq.poll();
}
```
Summary of $O(n \log n)$ sorts

- **Quicksort is relatively straight-forward to code, very fast**
  - Worst case is very unlikely, but possible, therefore ...
  - But, if lots of elements are equal, performance will be bad
    - One million integers from range 0 to 10,000
    - How can we change partition to handle this?

- **Merge sort is stable, it's fast, good for linked lists, harder to code?**
  - Worst case performance is $O(n \log n)$, compare quicksort
  - Extra storage for array/vector

- **Heapsort, more complex to code, good worst case, not stable**
  - Basically heap-based priority queue in a vector
Shell Sort: improving insertion

- Insertion sort only exchanges adjacent items
- When will insertion sort be slowest?
- Shell’s sort sorts every $h$th element and then changes the value of $h$

```java
void shellSort(String[] a){
    for (int h=1; h <= (a.length-1)/9; h = 3*h+1);
    for (; h > 0; h /= 3)
        for (int i=h; i < a.length; i++) {
            int j = i; String v = a[i];
            while (j >h && v.compareTo(a[j-h]) < 0) {
                a[j] = a[j-h];
                j -= h;
            }
            a[j] = v;
        }
}
```
Sorting in practice

- **Use Collections.sort and Arrays.sort!**
- **Rarely will you need to roll your own sort, but when you do** ...
  - What are key issues?

- **If you use a library sort, you need to understand the interface**
  - In C++ we have STL
    - STL has sort, and stable_sort
  - In C generic sort is complex to use because arrays are ugly
  - In Java guarantees and worst-case are important
    - Why won’t quicksort be used?

- **Comparators permit sorting criteria to change simply**