- Note that we swap keys j and $j-1$ if they are out of order. We then continue to move the smaller element down, until it is either first or the item below it is smaller than it. By the inductive hypothesis, the initial segment of the list is sorted. By the current argument, the new item is in place, and by the inductive hypothesis, the final segment of the list is sorted.

2.6 Step 3: Efficiency

An easy upper bound is obtained by bounding the two loops by n. This gives a running time of $O(n^2)$.
Is the worst-case running time $\Theta(n^2)$? That is, are there inputs for which running time is quadratic?
$\sum_{i=1}^{n-1} i = 1/2(n-1)n = \Theta(n^2)$.
Is the running time $\Theta(n^2)$? That is, is it quadratic for *all* inputs? What is a best-case input for insertion sort?
$\sum_{i=1}^{n-1} 1 = n-1 = \Theta(n)$.

2.7 Measures of Sortedness

Often it useful to talk about measures of list sortedness. Such a *distance* measure should assign a score of zero to a sorted list, and any unsorted list should score a positive amount. Some measures:

- Unsortedp: Is the list completely sorted? Score 1 if no.
- Items out of place: Score 1 for each position that contains an element other than what would be there in a sorted list. Maximum distance is n.
- Inversions: Score 1 for every pair that is out of order. Maximum distance is $n(n-1)/2$.
- Adjacent inversions: Score 1 for every adjacent pair that is out of order. Maximum distance is $n-1$.
- Insertion index: Find the longest increasing subsequence (LIS). Score n minus the length of the LIS. The same as asking, what's the smallest number of keys I can remove and the reinsert to get a sorted list? Maximum distance is $n-1$.

Example:

- **2 1 6 3 5 4 7**
- unsortedp: 1
- items out of place: 5.
- inversions: 1+0+3+0+1=5.
- adjacent inversions: 3.
- insertion index: 7-4=3.

2.8 Sorting with Small Insertion Indices

It is very common to apply sorting algorithms to lists with small insertion indices. Example? Insertion sort is nice because it runs fast for lists with small insertion indices.

- Given a list with insertion index k, how many times will insertion sort have to bubble an item into place?
- What's the maximum work that it takes to bubble a single item into place?
- This gives a $O(kn)$ bound. In the homework, you'll show that this bound is not tight.

2.9 An Assortment of Sorts

Several other sorts have similar asymptotic profiles to insertion sort, with different advantages and disadvantages.

- insertion sort: To sort $A[1] \ldots A[n]$, recursively sort $A[1] \ldots A[n-1]$. Then, bubble $A[n]$ downward until it stops. $T(n) = n + T(n-1)$. Worst case $\Theta(n^2)$, best case $\Theta(n)$ on sorted list.
- selection sort: To sort $A[1] \ldots A[n]$, pick out the max in $\Theta(n)$. Move max to the end in one swap. Recurse on $A[1] \ldots A[n-1]$. $T(n) = n + T(n-1)$. Running time is always $\Theta(n^2)$. Swaps $\Theta(n)$, which is asymptotically optimal.
- Bubble sort: Scan through the list over and over again swapping adjacent elements that are out of order. Like insertion sort only much slower.

3 DIVIDE-AND-CONQUER SORTING

3.1 Recurrences for Earlier Sorts

We can write recurrences for the previous sorts as follows:

- insertion sort: $T(n) = T(n-1) + n$
- Bubblesort, selection sort: $T(n) = n + T(n-1)$.

All basically focus on sorting a single element per iteration.

3.2 Divide and Conquer

Two choices:

- We can blindly break the problem into smaller lists, sort them, then do some work to patch things up after the fact (like insertion sort).

- Or we can do some work up front so that when the subproblems are solved, we are left with a sorted list (like selection sort).

But we need to break our lists in half: $T(n) = n + 2T(n/2)$.
By balancing, we get $T(n) = n \log n$.

3.3 Two Algorithms

Based on these insights, we can create the skeleton of two distinct algorithms.

- Break the list into two lists of size $n/2$. Sort each separately. Now, put the two sorted lists together in linear time.

- In linear time, split the list into two lists such that all the small elements are in one list and all the big elements are in the other. Once the two sublists are sorted recursively, the final result is sorted.

Recognize these algorithms?

3.4 Merge Sort

Structure: Break the list into two equal-size sublists. Recursively sort the two sublists. Merge them together to form a single sorted list.
Tricky part: How do you merge?
Insertion sort is a special case in which we merge a sorted list of size $n-1$ and one of size 1.
Idea: Look at the first element in each list. Remove the smaller one (by definition, an empty list is never the smaller one). Repeat. Animated example.
This involves constant work per element of the final list, i.e., linear time. (To be analyzed later.)
Awkwardness: Not "in place."

3.5 Quicksort

Structure: Separate list into the "lows" and "highs." Put the "lows" in the left section of the array and the "highs" in the right. Sort each recursively. Resulting list is definitely sorted.
Tricky part: How separate list?
Selection sort is a special case in which we define the "highs" to be a set of size 1 and the lows all the rest.
Idea: We choose an element in the list to be the "pivot." Easy to create two sublists from this, depending on which side of the pivot we're on. Animated example.
This involves constant work per element, because we simply check each one and then stow it away according to whether it's bigger or smaller than the pivot.
The partitioning procedure is especially elegant because it can be done in place.
Awkwardness: Hard to pick a good pivot fast.

3.6 Partitioning In Place

```
int partition( vector<int> & data, int start, int end, int pivot_position )
{
  int x = data[pivot_position];
  int i = start - 1;
  int j = end + 1;

  while( true )
  {
    do j--;
    while( data[j] > x );

    do i++;
    while( data[i] < x );

    if( i < j )
      swap( data, i, j );
    else
      return j;
  }
}
```

3.7 Explanation

Start at opposites sides of the segment of array we'd like to partition.
Move left pointer inwards as long as elements are less than pivot.
Move right pointer inwards as long as elements are greater than pivot.
Swap elements under left and right pointers and repeat.

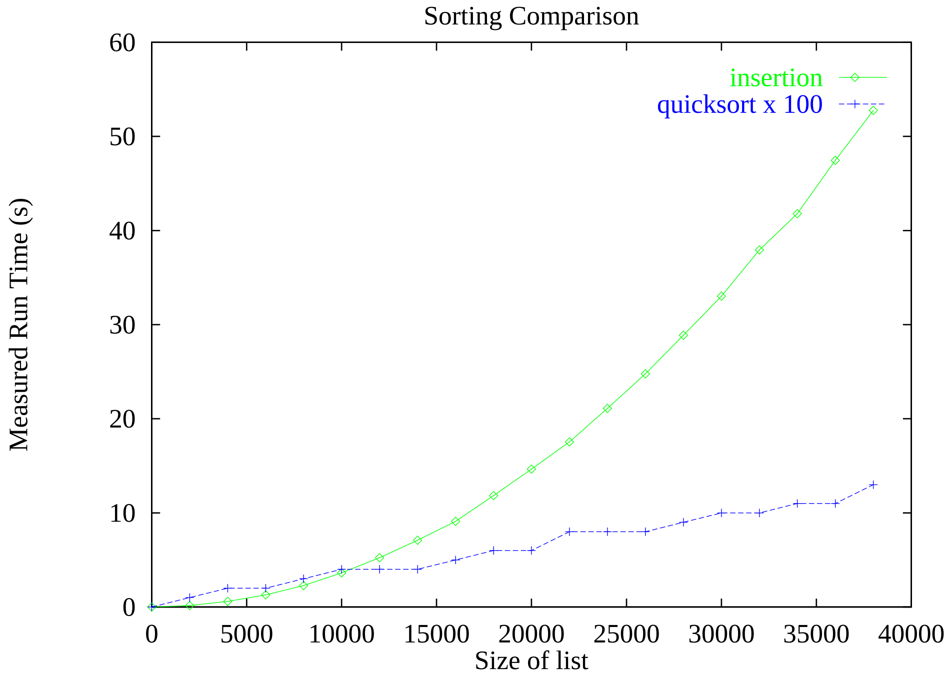
3.8 Picking A Pivot

What kind of pivot do we want to pick?

- Max or min? No, then we get selection sort because one list will have $n-1$ elements and the other 0.
- First element in list? Not good if list sorted or reverse sorted (get max or min)!
- Median of list? That would be perfect... $T(n) = n + 2T(n/2)$. Hard to find quickly. (Can be done in linear time, however.)
- Random element in list? Good chance of breaking the list into substantial chunks. Fast. But random... more complex analysis. (Next time.)
- Median of 3 randoms? More work, but more stable.

3.9 Empirical Comparison

Quicksort has better scaling properties *and* a better constant!



Note that the quicksort data is scaled up to make it visible on this plot!

3.10 A Consortium

There are some really nice demos of sorting algorithms on the web. Here are my favorites:

- High-level view of several sorting algorithms. Boy, quicksort is cool. Smaller set of algorithms.
- Nicer animations of how the algorithms work. Also provides mechanisms for collecting data to compare the variety of sorts provided.

4 SORTING IN LINEAR TIME

4.1 A Lower Bound

Several sorts are known that take $n \log n$ time (mergesort, quicksort, heapsort, and several others).
Is that as fast as one can go?
If we insist that sorting by done by comparison only, the answer is yes!

4.2 The Decision Tree

No matter how we sort, if we plan to put n items in order, we need to compare a sufficient number of items to uniquely identify the permutation.

- So, how many different permutations are there?
- Each comparison has the effect of ruling out some of the permutations. So, we can imagine that we start out with all permutations being possible, then making a comparison and splitting the set of possibilities in two.
- The entire sequence of comparisons for any given sorting algorithm can be thought of as a big tree, where each leaf of the tree contains no more than one permutation.
- The deepest path in this tree is the maximum number of comparisons we need to do to sort any list. We'll show that this is $\Omega(n \log n)$.

4.3 The Bound

Since a binary tree of height h has no more than 2^h leaves, we have: $2^h \geq n! \geq (n/2)^{n/2}$. The latter inequality comes from breaking up the factorial function:

$$n! = (n)(n-1)\cdots(n/2)(n/2-1)\cdots(2)(1) \geq (n/2)(n/2)\cdots(n/2)(1)\dots(1)(1) = (n/2)^{n/2}.$$

Taking logs and using their monotonicity property, we have $h \geq \log((n/2)^{n/2}) = n/2 \log(n) - n/2 = \Omega(n \log n)$.
So, the height of the tree is at least $\Omega(n \log n)$... with any fewer comparisons, we won't be able to sort.

4.4 Faster Sorts Via Bit Operations

The only way to beat the $n \log n$ bound is to use something other than comparisons in the sort.
Some well known sorts use various properties to achieve $O(n)$ running times: counting sort, radix sort, bucket sort.
I'll describe one that is an in-place sort that runs in time $O(nb)$ time sorting integers with b bits.

4.5 Forward Radix Sort

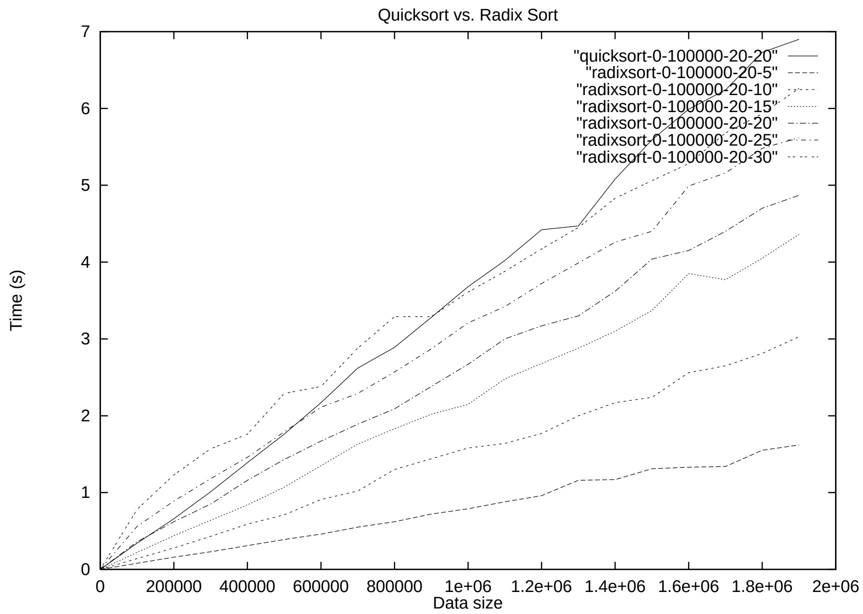
```
int partition_radix( vector<int> & data, int start, int end, int bit )
{
  int i = start - 1;
  int j = end + 1;

  while( true )
  {
    do j--;
    while( get_bit( data[j], bit ) == 1 && j >= start );
```

```
    do i++;
    while( get_bit( data[i], bit ) == 0 && i <= end );

    if( i < j )
      swap( data, i, j );
    else
      return j;
  }
}

void radixsort( vector<int> & data, int start, int end, int bit )
{
  if( start >= end ) return;
  if( bit == -1 ) return;

  int middle = partition_radix( data, start, end, bit );
  radixsort( data, start, middle, bit - 1 );
  radixsort( data, middle + 1, end, bit - 1 );
}
```

4.6 Discussion

This looks at the most significant bit, partitions the list into those with a zero there and those with a one. (Like quicksort.)
At this point, we can sort the two sublists separately.
Recurrence: $T(n, 0) = 0$, $T(n, b) = n + T(a, b-1) + T(n-a, n-1)$ (for some $1 \leq a \leq n$).
Can prove by induction that this gives $T(n) = bn$.

4.7 Empirical Results on Random Lists



5 SPARSE MATRICES

5.1 Matrices

An m by n matrix A contains nm elements in it. For $1 \leq i \leq m$, $1 \leq j \leq n$, $A[i,j]$ is the (i,j)th element of matrix A.
Matrices can be used to represent all kinds of things. $A[i,j]$ might be:

- the grade for student i on exam j;
- the driving distance from city i to city j;
- the number of times word i appears in encyclopedia article j;
- the probability of an elevator going from floor i to floor j.

5.2 Vectors and Matrix-Vector Product

A vector is just a $n \times 1$ matrix.
The *product* of an $m \times n$ matrix A and an $n \times 1$ vector x is an $m \times 1$ vector y where $y_i = \sum_{j=1}^{n} A[i,j]x[j]$. It is the "dot product" between the rows of A and x.
Matrix-vector products come up all over the place. For example:

- If A is the matrix of student exam grades and x is a vector of "weights" (what fraction of the final grade is assigned to each exam), then Ax is a vector of final scores for each student.