CS61B Summer 2006
Instructor: Erin Korber
Lectures 13,14,15: 18,19,20 July

Sorting

The need to sort numbers, strings, and other records arises frequently in computer applications. The entries in any modern phone book were sorted by a computer. Databases have features that sort the records returned by a query, ordered according to any field the user desires.

Sorting is perhaps the simplest fundamental problem that offers a large variety of algorithms, each with its own inherent advantages and disadvantages. We'll study and compare several sorting algorithms, using the algorithm analysis tools that we've learned.

Most of the sorting algorithms that we discuss (the first 5) will be "comparison-based" - that is, they compare pairs of items to sort the set. A feature of the Java library that will be relevant in this context is the `Comparable` interface, which has one method, `compareTo`. This, as you might guess, compares two objects. So things that you want to sort using comparison-based methods should implement this interface.

1 Insertion Sort

Insertion sort is very simple and runs in $O(n^2)$ time. We employ a list S, and maintain the invariant that S is sorted.

Pseudocode:

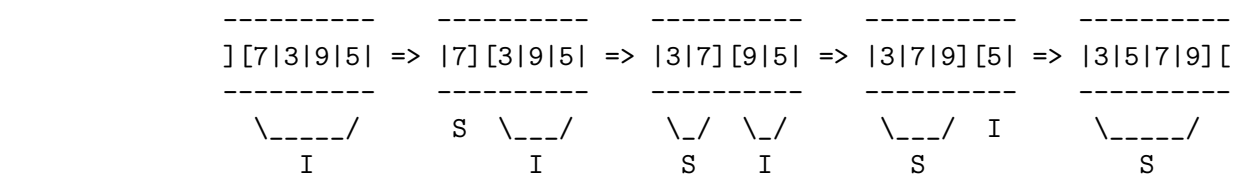
```
Start with an empty list S and the unsorted list I of n input items.
for (each item x in I) {
  insert x into the list S, positioned so that S remains in sorted order.
}
```

S may be an array or a linked list. If S is a linked list, then it takes $\Theta(n)$ worst-case time to find the right position to insert each item (since we have to walk down the list to the right position to insert into). If S is an array, we can find the right position to insert into in O(log n) time by binary search, but it takes $\Theta(n)$ worst-case time to shift the larger items over to make room for the new item. In either case, insertion sort is an $O(n^2)$ algorithm–but for a different reason in each case.

What are the memory requirements for this sort? Can you think of a way to do it with lists without creating a lot of new objects?

If S is an array, one of the nice things about insertion sort is that it's an in-place sort. An "in-place" sort leaves the sorted items in the same array that initially held the input items, and uses only O(1) or perhaps O(log n) additional memory (in addition to the input array).

To do an in-place insertion sort, we partition the array into two pieces: the left portion (initially empty) holds S, and the right portion holds I. With each iteration, the dividing line between S and I moves one step to the right.



If the input list I is "almost" sorted, insertion sort can be as fast as $\Theta(n)$, if the algorithm starts its search for the right place to insert from the *end* of S. In this case, the running time is proportional to n plus the number of "inversions". An inversion is a pair of values $j > k$ such that j appears before k in I (j and k are in the wrong order - "inverted"). I could have anywhere from zero to n (n - 1) / 2 inversions.

2 Selection sort

Selection sort is equally simple, and also runs in quadratic time. Again we employ a list S, and maintain the invariant that S is sorted. Now, however, we walk through I and pick out the largest item, which we insert at the front of S.

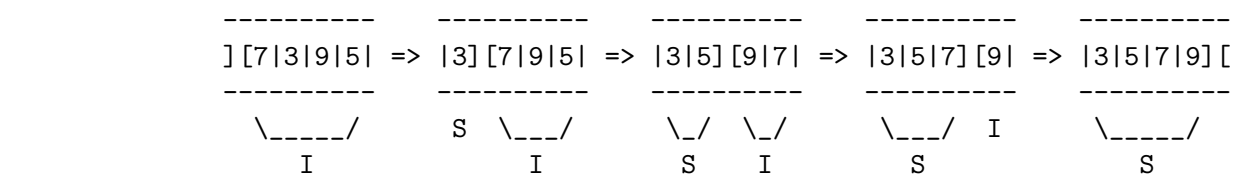
Pseudocode:

```
Start with an empty list S and the unsorted list I of n input items.
for (i = 0; i < n; i++) {
  Let x be the largest-valued item in I
  Remove x from I.
  Insert x at the front of S.
}
```

Whether S is an array or linked list, finding the largest item takes $\Theta(n)$ time, so selection sort takes $\Theta(n^2)$ time, even in the best case! Hence, it's even worse than insertion sort - they have the same performance in the worst

case, but insertion sort is sometimes better - selection sort will always be this bad.

If S is an array, we can do an in-place selection sort to save memory. After finding the smallest item in I, swap it with the first item in I, as shown here. We just need to have one extra temporary variable to hold one of the items while the swap is taking place. (Note that it doesn't matter if we search for the smallest and store the sorted portion to the left, or search for the largest and store the sorted portion to the write - they do essentially the same thing.)



3 Bubble sort

Another simple sorting algorithm is bubble sort, in which we make passes over the list, and in each pass, each element is compared to the one after it; if the two are found to be in the wrong order, they are switched. Bubble sort is also $\Theta(n^2)$, even in the best case. Bubble sort is generally a bad choice of algorithm - in addition to being just as slow asymptotically as selection sort, the constant factors will be larger (since we have to go through the entire list each pass, instead of only the part that is "still unsorted", since the whole thing is still unsorted until the very end).

4 Mergesort

Mergesort is based on the observation that it's possible to merge two sorted lists into one sorted list in linear time.

Pseudocode:

```
Let L1 and L2 be two sorted lists, and L be an empty list
while (neither L1 nor L2 is empty) {
  item1 = L1.head;
  item2 = L2.head;
  remove the smaller of item1 and item2 from its
     present list and put it into L.
}
append the remaining non-empty list (L1 or L2) to the end of L.
```

At each iteration, the merge routine chooses the item having smallest key from the two input lists, and appends it to the output list. Since the two input lists are sorted, there are only two items to test, so each iteration takes constant time. Hence, merging takes O(n) time.

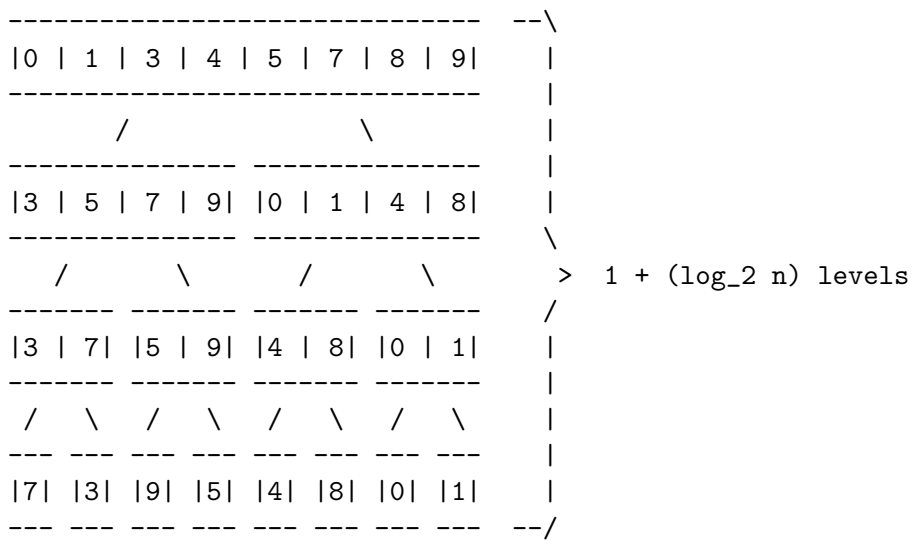
Mergesort is a *recursive divide-and-conquer algorithm*, in which the merge routine is what allows us to reunite what we divided. The procedure:

- Start with the unsorted list I of n input items.
- Break I into two halves I1 and I2, each having n/2 items
- Sort I1 recursively, yielding the sorted list S1.
- Sort I2 recursively, yielding the sorted list S2.
- Merge S1 and S2 into a sorted list S.

The recursion bottoms out at one-item lists. How long does mergesort take? The answer is made apparent by examining its recursion tree (next page):

Each level of the tree involves O(n) operations (to do the merging), and there are O(log n) levels. Hence, mergesort runs in O(n log n) time.

What makes mergesort a memory-efficient algorithm for sorting linked lists makes it a memory-inefficient algorithm for sorting arrays. Unlike the other sorting algorithms we've considered, mergesort is not an in-place algorithm. There is no reasonably efficient way to merge two arrays in place. Instead, we need to use an extra array of O(n) size to temporarily hold the result of a merge.



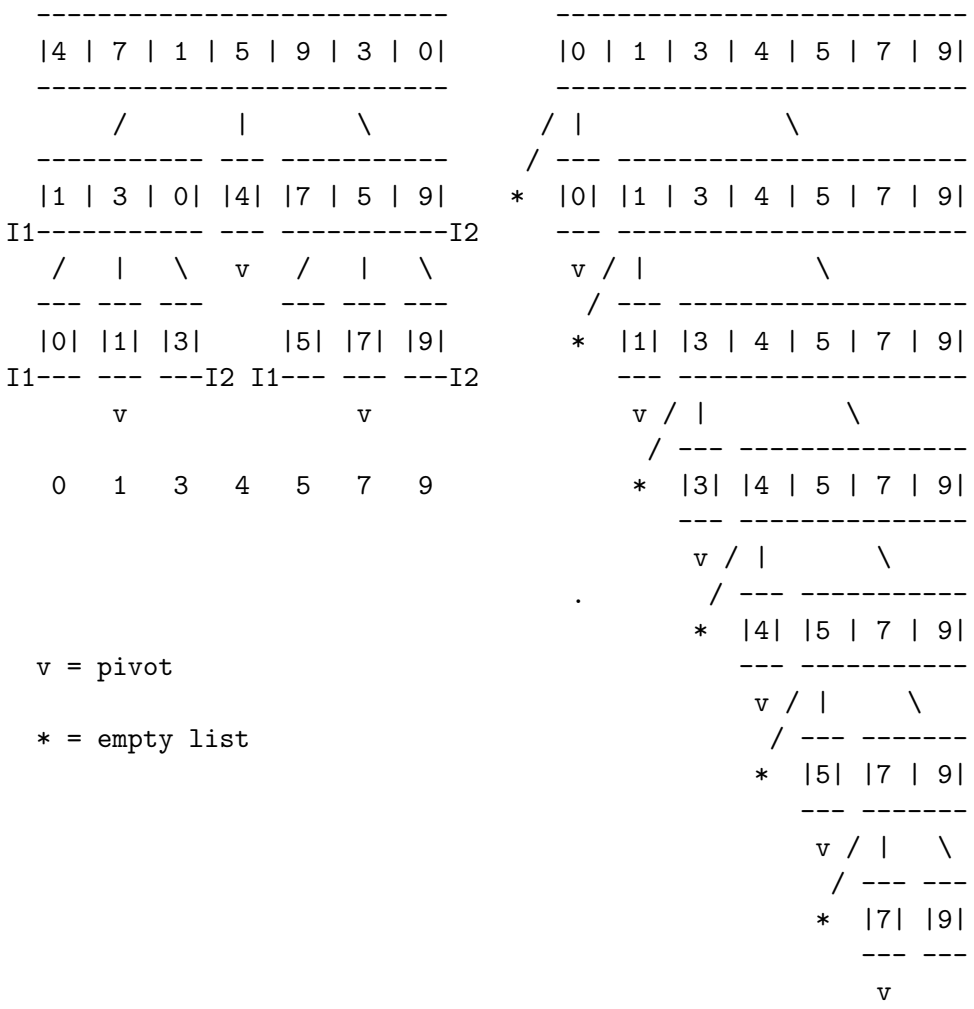
5 Quicksort

Quicksort is a recursive divide-and-conquer algorithm, like mergesort. Quicksort is in practice the fastest known comparison-based sort for arrays, even though it has a $\Theta(n^2)$ worst-case running time. If properly designed, however, it virtually always runs in O(n log n) time. On arrays, this asymptotic bound hides a constant smaller than mergesort's, but mergesort is probably slightly faster for sorting linked lists.

Given an unsorted list I of items, quicksort chooses a "pivot" item v from I, then puts each item of I into one of two unsorted lists, depending on whether its value is less or greater than v's. (Items whose values are equal to v's can go into either list; we'll discuss this issue later.)

- Start with the unsorted list I of n input items.
- Choose a pivot item v from I.
- Partition I into two unsorted lists I1 and I2.
  - I1 contains all items whose keys are smaller than v's key.
  - I2 contains all items whose keys are larger than v's.
  - Items with the same key as v can go into either list.
  - The pivot v, however, does not go into either list.
- Sort I1 recursively, yielding the sorted list S1.
- Sort I2 recursively, yielding the sorted list S2.
- Append S1, v, and S2 together, yielding a sorted list S.

The recursion bottoms out at one-item and zero-item lists. (Zero-item lists can arise when the pivot is the smallest or largest item in its list.) How long does quicksort take? The answer is made apparent by examining several possible recursion trees. In the illustrations below, the pivot v is always chosen to be the first item in the list.



In the example at left, we get lucky, and the pivot always turns out to be the item having the median key. Hence, each unsorted list is partitioned into two pieces of equal size, and we have a well-balanced recursion tree. Just like in mergesort, the tree has O(log n) levels. Assuming that partitioning the list is a linear-time operation, the total running time is O(n log n).

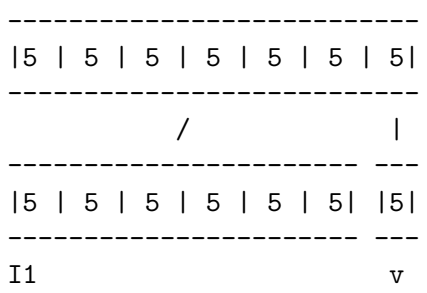
The example at right, on the other hand, shows the dreaded worst-case $\Theta(n^2)$ performance that results if the pivot always proves to have the smallest or largest key in the list. (You can see it takes $\Omega(n^2)$ time because the first n/2 levels each process a list of length n/2 or greater.) The recursion tree is as unbalanced as it can be. This example shows that when the input list I happens to be already sorted, choosing the pivot to be the first item of the list is a disastrous policy.

5.1 Choosing a Pivot

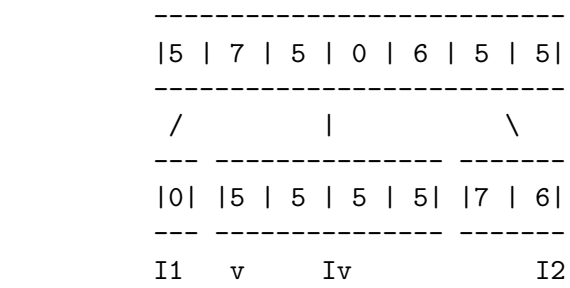
We need a better way to choose a pivot. A respected, time-tested method is to randomly select an item from I to serve as pivot. With a random pivot, we can expect "on average" to obtain a 1/4 - 3/4 split; half the time we'll obtain a worse split, half the time better. A little math shows that the average running time of quicksort with random pivots is in O(n log n) - we'll examine how to determine this more carefully later in the semester when we talk about "randomized analysis".

5.2 Quicksort on Linked Lists

We left unresolved the question of what to do with items that have the same key as the pivot. Suppose we put all the items having the same key as v into the list I1. If we try to sort a list in which every single item has the same key, then *every* item will go into list I1, and quicksort will have quadratic running time! (See illustration below - you can see how all the items will always be in I1)



When sorting a linked list, a far better solution is to partition I into *three* unsorted lists I1, I2, and Iv. Iv contains the pivot v and all other items with the same key. (See diagram below - next page). We sort I1 and I2 recursively, yielding S1 and S2. Iv, of course, does not need to be sorted. Finally, we append S1, Iv, and S2 to yield S.



This strategy is quite fast if there are a large number of duplicate keys, because the lists called "Iv" (at each level of the recursion tree) require no further sorting or manipulation.

Unfortunately, with linked lists, selecting a pivot is annoying. With an array, we can read a randomly chosen pivot in a very quick constant time; with a linked list we must walk half-way through the list on average, significantly increasing the constant in our running time. However, if we restrict ourselves to pivots near the beginning of the linked list instead of truly choosing randomly, we risk quadratic running time (for instance, if I is already in sorted order, or nearly so).

5.3 Quicksort on Arrays

Quicksort shines for sorting arrays. In-place quicksort is very fast. But a fast in-place quicksort is tricky to code. It's easy to write a buggy or quadratic version by mistake - many textbooks even have this error.

Suppose we have an array a in which we want to sort the items starting at a[low] and ending at a[high]. We choose a pivot v and move it out of the way by swapping it with the last item, a[high].

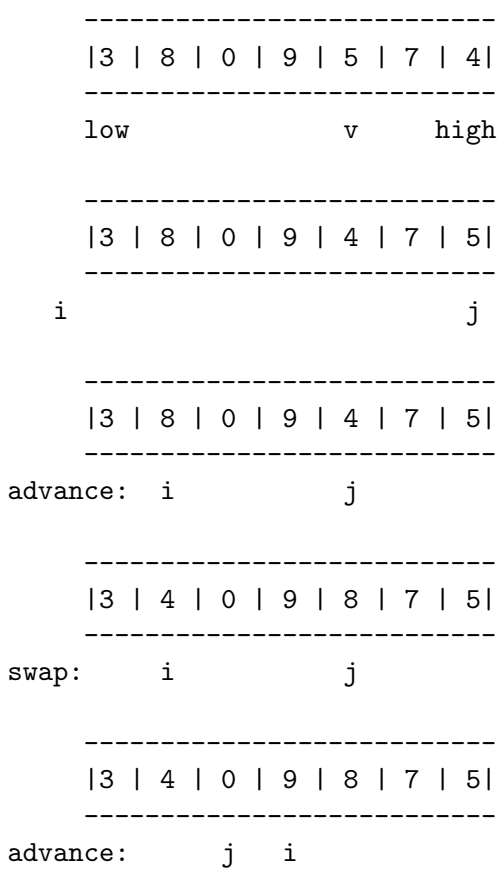
We employ two array indices, i and j. i is initially low-1, and j is initially high, so that i and j sandwich the items to be sorted (not including the pivot). We will enforce the following invariants:

- All items at or left of index i are $\leq$ the pivot.
- All items at or right of index j are $\geq$ the pivot.

To partition the array, we advance the index i until it encounters an item greater than or equal to the pivot; then we decrement the index j until it encounters an item less than or equal to the pivot. Then, we swap the items at i and j. We repeat this sequence until the indices i and j meet in the middle. Then, we move the pivot back into the middle (by swapping it with the item at index i).

An example is given below. The randomly selected pivot, whose value is 5, is moved to the end of the array by swapping it with the last item. The indices i and j are created.i advances until it reaches an item ≥ 5, and j retreats until it reaches an item ≤ 5. The two items are swapped, and i advances and j retreats again. After the second advance/retreat, i and j have crossed paths, so we do not swap their items. Instead, we swap the pivot with the item at index i, putting it between the lists I1 and I2 where it belongs.

What about items equal to the pivot? Handling these is particularly tricky. We'd like to put them on a separate list (as we did for linked lists), but doing that in place is too slow. As I noted previously, if we put all these items into the list I1, we'll have quadratic running time when all the items in the array are equal, so we don't want to do that either.



```
----------- --- -----------
|3 | 4 | 0| |5| |8 | 7 | 9|
----------- --- -----------
I1       j   i           I2
```

The solution is to make sure each index, i and j, stops whenever it reaches a key equal to the pivot. Every key equal to the pivot (except perhaps one, if we end with i=j) takes part in one swap. Swapping an item equal to the pivot may seem unnecessary, but it has an excellent side effect: if all the items in the array are equal, half these items will go into I1, and half into I2, giving us a well-balanced recursion tree. (To see why, try running the pseudocode below on paper with an array of equal values.)

```
public static void quicksort(Comparable[] a, int low, int high) {
  // If there's fewer than two items, do nothing.
  if (low < high) {
    int pivotIndex = random number between low and high
    Comparable pivot = a[pivotIndex];
    a[pivotIndex] = a[high];         // Swap pivot with last item
    a[high] = pivot;

    int i = low - 1;
    int j = high;
    do {
      do { i++; } while (a[i] is less than pivot);
      do { j--; } while ((a[j] is greater than pivot) && (j > low));
      if (i < j) { swap a[i] and a[j]; }
    } while (i < j);

    a[high] = a[i];
    a[i] = pivot;                    // Put pivot in the middle where it belongs
    quicksort(a, low, i - 1);        // Recursively sort left list
    quicksort(a, i + 1, high);       // Recursively sort right list
  }
}
```

Can the "do i++ " loop walk off the end of the array and generate an out-of-bounds exception? No, because a[high] contains the pivot, so i will stop advancing when i == high (if not sooner). There is no such assurance for j, though, so the "do j– " loop explicitly tests whether $j > low$ before retreating.

6 The selection problem - quickselect

Suppose that we want to find the kth smallest item in a list. In other words, we want to know which item has index j if the list is sorted (where j = k - 1). We could simply sort the list, then look up the item at index j. But if we don't actually need to sort the list, is there a faster way? This problem is called *selection*.

One example is finding the median of a set of values. If n items are numbered from 0 to n - 1 (where n is odd), we are looking for the item whose index is j = (n - 1) / 2 in the sorted list.

We can modify quicksort to perform selection for us. Observe that when we choose a pivot v and use it to partition the list into three lists I1, Iv, and I2, we know which of the three lists contains index j, because we know the lengths of I1 and I2. Therefore, we only need to search one of the three lists.

Here's the quickselect algorithm for finding the item at index j - that is, the (j + 1)th smallest item.

- Start with an unsorted list I of n input items.
- Choose a pivot item v from I.
- Partition I into three unsorted lists I1, Iv, and I2.
  - I1 contains all items whose keys are smaller than v's key.
  - I2 contains all items whose keys are larger than v's.
  - Iv contains the pivot v.
  - Items with the same key as v can go into any of the three lists. (In list-based quickselect, they go into Iv; in array-based quickselect, they go into I1 and I2, just like in array-based quicksort.)
- If (j < I1.length), recursively find the item with index j in I1; return it.
- Else, if (j < I1.length + Iv.length), return the pivot v.
- Otherwise, recursively find the item with index j - I1.length - Iv.length in I2; return it.

The advantage of quickselect over quicksort is that we only have to make one recursive call, instead of two. Since we make at most *one* recursive call at *every* level of the recursion tree, quickselect is much faster than quicksort. (It actually runs in $\Theta(n)$ average time if we select pivots randomly, but we won't analyze this here.)

We can easily modify the code for quicksort on arrays to do selection. The partitioning step is done exactly according to the pseudocode for array

quicksort. Recall that when the partition stage finishes, the pivot is stored at index "i" (see the variable "i" in the array quicksort pseudocode). In the quickselect pseudocode above, just replace I1.length with i and Iv.length with 1.

7 A Lower Bound On Running Time for Comparison-Based Sorts

Suppose we have a scrambled array of n numbers, with each number from 1...n occurring once. How many possible orders can the numbers be in?

The answer is n!, where n! = 1 * 2 * 3 * ... * (n-2) * (n-1) * n. Here's why: the first number in the array can be anything from 1...n, yielding n possibilities. Once the first number is chosen, the second number can be any one of the remaining n-1 numbers, so there are n * (n-1) possible choices of the first two numbers. The third number can be any one of the remaining n-2 numbers, yielding n * (n-1) * (n-2) possibilities for the first three numbers, and so on.

Each different order is called a *permutation* of the numbers, and there are n! possible permutations.

Observe that if $n > 0$,

$$n! = 1 * 2 * ... * (n-1) * n \leq n * n * n * ... * n * n * n = n^n$$

and (supposing n is even)

$$n! = 1 * 2 * ... * (n-1) * n \geq \frac{n}{2} * (\frac{n}{2} + 1) * ... * (n-1) * n \geq (\frac{n}{2})^{\frac{n}{2}}$$

so n! is between $(n/2)^{(n/2)}$ and n^n. Let's look at the logarithms of both these numbers: $\log((n/2)^{(n/2)}) = (n/2)\log(n/2)$, which is in $\Theta(n \log n)$, and $\log(n^n) = n \log n$. Hence, log(n!) is also in $\Theta(n \log n)$.

A comparison-based sort is one in which all decisions are based on comparing keys (generally using "if" statements). All actions taken by the sorting algorithm are based on the results of a sequence of true/false questions. All of the sorting algorithms we have studied are comparison-based.

Suppose we run a comparison-based sorting algorithm twice, on two different permutations of 1...n. Suppose that every execution of an "if" statement gives the same true/false answer during the second run as during the first run. Then both runs execute exactly the same operations (e.g. swapping numbers) in exactly the same order, so they both rearrange the

input in *exactly* the same way. One of the runs must have gotten the wrong answer!

A correct sorting algorithm must generate a *different* sequence of true/false answers for each different permutation of 1...n, because it takes a *different* sequence of computations to sort each permutation. There are n! different permutations, thus n! different sequences of true/false answers.

If a sorting algorithm asks d true/false questions for the worst-case permutation of 1...n, it generates $\leq 2^d$ different sequences of true/false answers. Therefore, $n! \leq 2^d$, so $\log_2(n!) \leq d$, and d is in $\Omega(n \log n)$. The algorithm spends $\Theta(d)$ time asking these questions. Therefore,

EVERY comparison-based sorting algorithm takes $\Omega(n \log n)$ worst-case time.

However, there are faster sorting algorithms that can make q-way decisions for large values of q, instead of true/false (2-way) decisions. Some of these algorithms run in linear time.

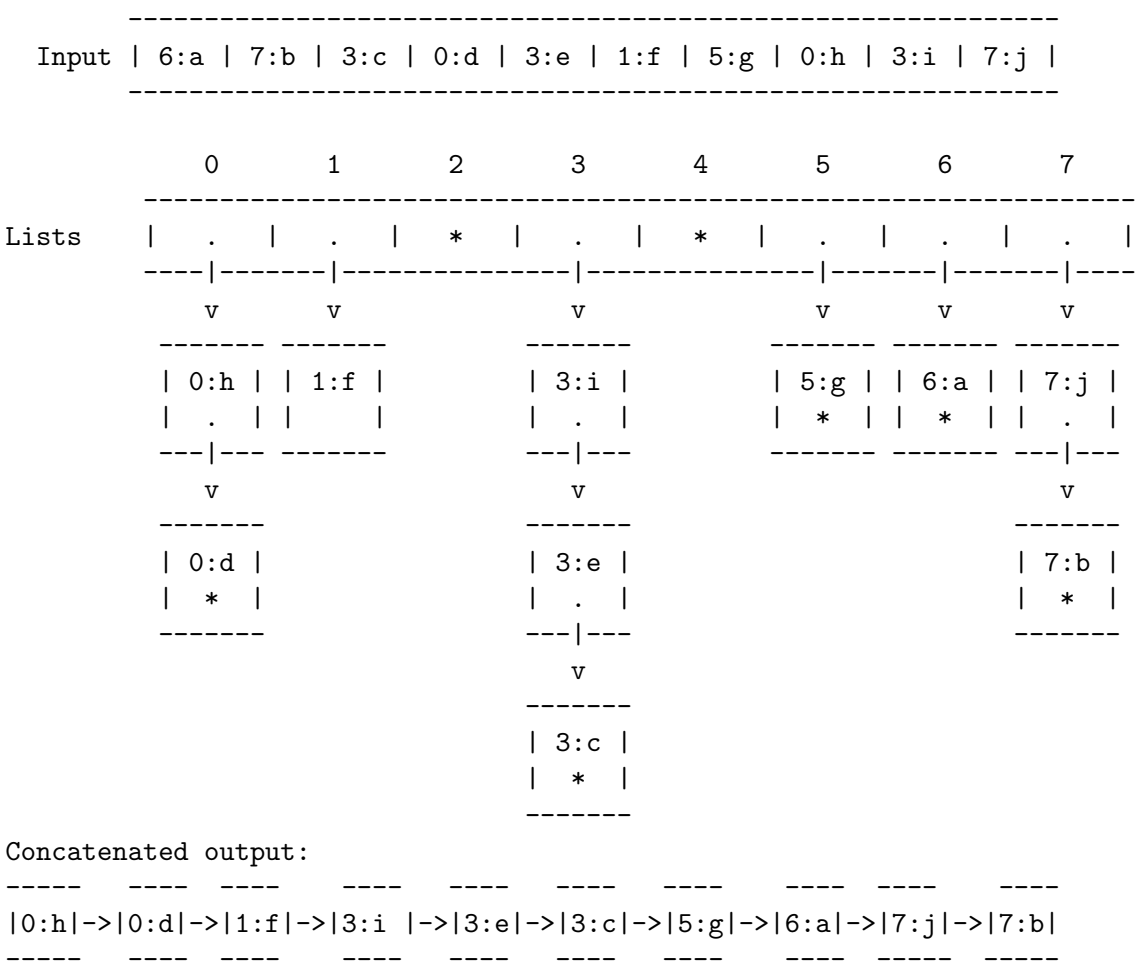
8 Linear-Time Sorting

8.1 Bucket Sort

Bucket sort works well when keys are distributed in a small range, e.g. from 0 to q-1, and the number of items n is larger than, or nearly as large as, q. In other words, when q is in O(n).

We allocate an array of q linked lists with tail references, numbered from 0 to q-1. The lists are called *buckets*. We walk through the list of input items, and put each item in the appropriate list: an item with key i goes into bucket i. When we're done, we append all the lists together.

Each item illustrated here has a numerical key and an associated value (here, a character).



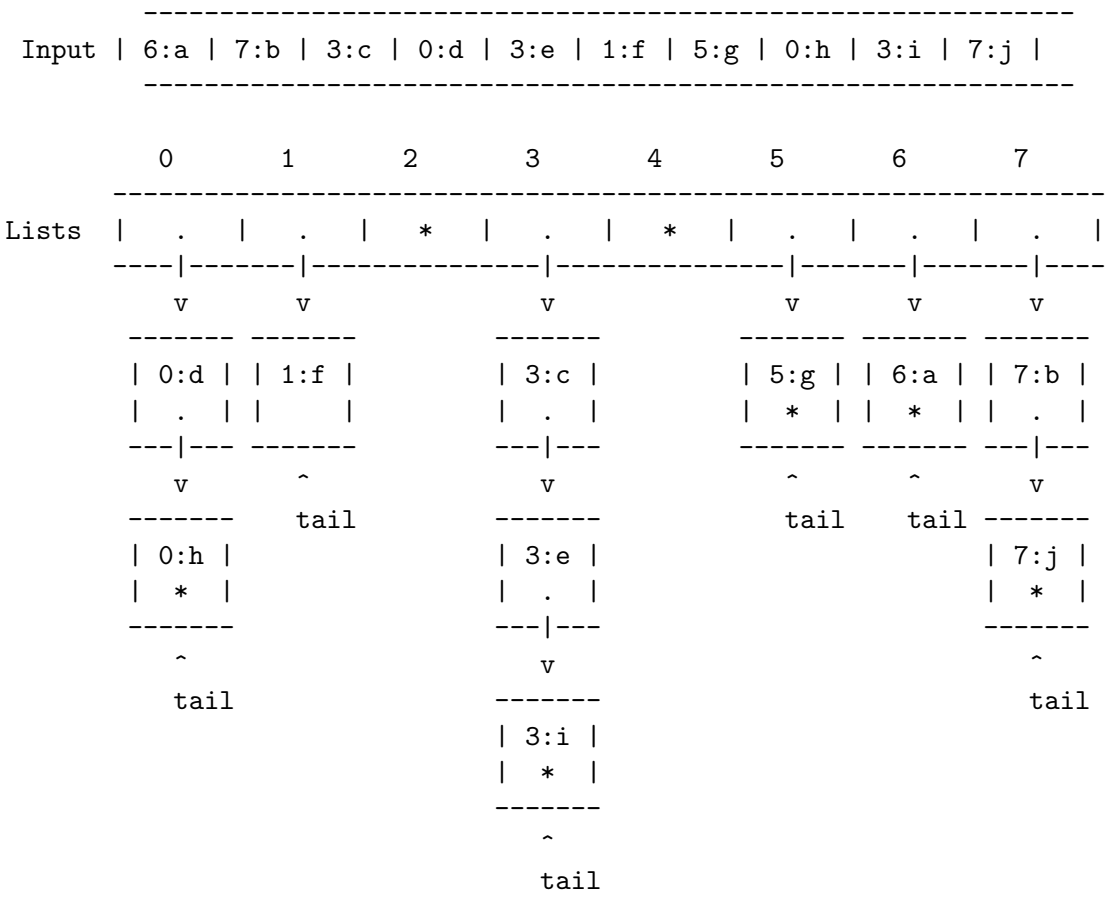
Bucket sort takes $\Theta(q+n)$ time–in the best case and in the worst case. It takes $\Theta(q)$ time to initialize the buckets in the beginning and to concatenate them together in the end. It takes $\Theta(n)$ time to put all the items in their buckets, and $\Theta(n)$ time to append all the lists together at the end. (Recall: $\Theta(q) + \Theta(n) + \Theta(n) = \Theta(q) + \Theta(2n) = \Theta(q + n)$)

If q is in O(n)–that is, the number of possible keys isn't significantly larger than the number of items we're sorting–then $\Theta(q + n) = \Theta(n)$, so bucket sort takes $\Theta(n)$ time in this case. How did we get around the $\Omega(n \log n)$ lower bound on comparison-based sorting? Bucket sort is not comparison-based. We are making a q-way decision every time we put an item in a bucket, instead of the true/false decisions provided by comparisons and "if" statements.

If instead of just sticking the item in at the front of the bucket, we put it at the end (we could use a list with a tail pointer to make this efficient), then we get an additional nice property: bucket sort implemented this way is *stable*. A sort is *stable* if items with equal keys come out in the same order they went in.

Bucket sort is not the only stable sort we have seen; insertion sort, selection sort, and mergesort can all be implemented so that they are stable. The linked list version of quicksort we discussed can be stable, but the array version is decidedly not.

Here is the same example as above, but using the stable version of bucket sort. Notice how the original order is preserved for items having the same key. For example, observe that 3:c, 3:e, and 3:i appear in the same order in the output as they appeared in the input.




```
Concatenated output:
-----   ----  ----    ----   ----   ----   ----   ----  ----    ----
|0:d|->|0:h|->|1:f|->|3:c|->|3:e|->|3:i|->|5:g|->|6:a|->|7:b|->|7:j|
-----   ----  ----    ----   ----   ----   ----   ----  -----  -----
```

Take note that bucket sort is ONLY appropriate when keys are distributed in a small range; i.e. q is in O(n). What about when we have a larger set of keys? *Radix sort* can fix that problem. The stability of bucket sort will be important for radix sort.

8.2 Counting Sort

If the items we sort are naked keys, with no associated values, bucket sort can be simplified to become *counting sort*. In counting sort, we don't need lists at all; we need merely keep a count of how many copies of each key we have encountered. Suppose we sort 6 7 3 0 3 1 5 0 3 7:

	0	1	2	3	4	5	6	7
counts	2	1	0	3	0	1	1	2

When we are finished counting, it is straightforward to reconstruct the sorted keys from the counts: 0 0 1 3 3 3 5 6 7 7.

Now let's go back to the case where we have complete items (key plus associated value). We can use a more elaborate version of counting sort. The trick is to use the counts to find the right index to move each item to.

Let x be an input array of objects with keys. Begin by counting the keys in x.

```
for (i = 0; i < x.length; i++) {
  counts[x[i].key]++;
}
```

Next, do an accumulation of the "counts" array so that counts[i] contains the number of keys *less than* i (instead of equal to i).

	0	1	2	3	4	5	6	7
counts	0	2	3	3	6	6	7	8

```
total = 0;
for (j = 0; j < counts.length; j++) {
  c = counts[j];
  counts[j] = total;
  total = total + c;
}
```

Let y be the output array, where we will put the sorted objects. counts[i] tells us the first index of y where we should put items with key i. Walk through the array x and copy each item to its final position in y. When you copy an item with key k, you must increment counts[k] to make sure that the next item with key k goes into the next slot.

```
for (i = 0; i < x.length; i++) {
  y[counts[x[i].key]] = x[i];
  counts[x[i].key]++;
}
```

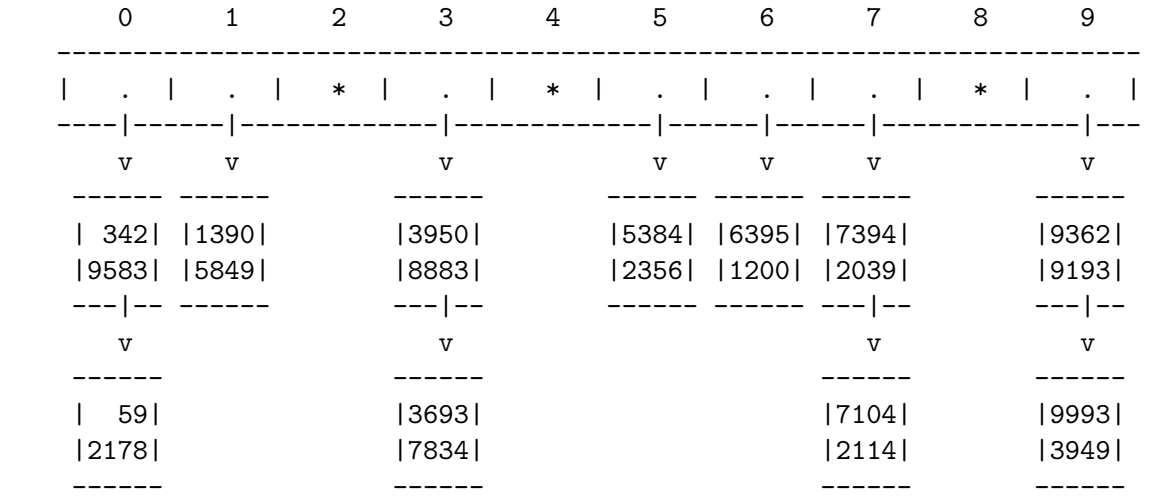
Bucket sort and counting sort both take O(q + n) time. If q is in O(n), then they take O(n) time. Counting sort is a little harder to understand than bucket sort, but if you're sorting an array, it takes less memory. (Bucket sort is more natural if you're sorting a linked list.)

However, if q is not in O(n) - there are many more possible values for keys than there are keys - we need a different method to get linear-time performance.

8.3 Radix Sort

Suppose we want to sort 1,000 items in the range from 0 to 99,999,999. If we use bucket sort, we'll spend so much time initializing and appending empty lists we'll wish we'd used selection sort instead.

Instead of providing 100 million lists, let's provide q = 10 lists and sort on the first digit only. (A number less than 10 million is said to have a first digit of zero.) We use bucket sort or counting sort, treating each item as if its key is the first digit of its true key.



Once we've dealt all 1,000 items into ten lists, we could sort each list recursively on the second digit; the lists that result would be sorted on the third digit, and so on. Unfortunately, this tends to break the set of input

items into smaller and smaller subsets, each of which will be sorted relatively inefficiently.

Instead, we use a clever but counterintuitive idea: we'll keep all the numbers together in one big pile throughout the sort; but we'll sort on the *last* digit first, then the next-to-last, and so on up to the most significant digit.

The reason this idea works is because bucket sort and counting sort are stable. Hence, once we've sorted on the last digit, the numbers 55,555,552 and 55,555,558 will remain ever after in sorted order, because their other digits will be sorted stably. Consider an example with three-digit numbers:

1s:	720	450	771	822	332	925	5	955	825	777	858	28	829
10s:	5	720	822	925	825	28	829	332	450	955	858	771	777
100s:	5	28	332	450	720	771	777	822	825	829	858	925	955

After we sort on the middle digit, observe that the numbers are sorted by their last two digits. After we sort on the most significant digit, the numbers are completely sorted.

Returning to our example with eight-digit numbers, we can do better than sorting on one decimal digit at a time. It would likely be faster if we sort on two digits at a time (using a radix of q = 100) or even three (using a radix of q = 1000). Furthermore, there's no need to use decimal digits at all; on computers, it's more natural to choose a power-of-two radix like q = 256. Base-256 digits are easier to extract from a key, because we can quickly pull out the eight bits that we need.

How many passes must we perform? Each pass inspects $\log_2 q$ bits of each key. If all the keys can be represented in b bits, the number of passes is $\lceil \frac{b}{\log_2 q} \rceil$. So the running time of radix sort is

$$O((n+q)\lceil \frac{b}{\log_2 q} \rceil)$$

How should we choose the number of lists q? Let's choose q to be in O(n), so each pass of bucket sort or counting sort takes O(n) time. However, we want q to be large enough to keep the number of passes small. Therefore, let's choose q to be approximately n. With this choice, the number of passes is $O(1 + \frac{b}{\log_2 n})$, and radix sort takes time in

$$O(n + n\frac{b}{\log n})$$

For many kinds of keys we might sort (like ints), b is a constant, and radix sort takes linear time. Even if the key length b tends to grow logarithmically

with n (a reasonable model in many applications), radix sort is still a linear-time algorithm.

In practice, an efficient choice is to make q equal to n rounded down to the next power of two. If we want to keep memory use low, however, we can make q equal to the square root of n, rounded to the nearest power of two. With this choice, the number of buckets is far smaller, but we only double the number of passes.

8.4 Postscript: Radix Sort in Real Life

Linear-time sorts tend to get less attention than comparison-based sorts in most computer science classes and textbooks. Perhaps this is because the theory behind linear-time sorts isn't as interesting as for other algorithms. Nevertheless, the library sort routines for machines like Crays all use radix sort, because its the best in the speed department.

Radix sort can be used not only with integers, but with almost any data that can be compared bitwise, like strings. The IEEE standard for floating-point numbers is designed to work with radix sort combined with a simple prepass and postpass (to flip the bits, except the sign bit, of each negative number).

Strings of different lengths can be sorted in time proportional to the total length of the strings. A first stage sorts the strings by their lengths. A second stage sorts the strings character by character (or several characters at a time), starting with the last character of the longest string and working backward to the first character of every string. We don't sort every string during every pass of the second stage; instead, a string is included in a pass only if it has a character in the appropriate place.

For instance, suppose we're sorting the strings CC, BA, CCAAA, ACCCD, and BAABA. The first three passes sort only the last three strings by their last three characters, yielding CCAAA BAABA ACCCD. The fourth pass is on the second character of each string, so we prepend the two-character strings to our list, yielding CC BA CCAAA BAABA ACCCD. After sorting on the second and first characters, we end with

ACCCD BA BAABA CC CCAAA.

Observe that BA precedes BAABA and CC precedes CCAAA because of the stability of the sort. That's why we put the two-character strings before the five- character strings when we began the fourth pass.