- *Partial-sum(k)* – Return the sum of all the elements currently in the set whose key is less than y, i.e. $\sum_{x_j<y} x_i$.

The worst case running time should still be $O(n \log n)$ for any sequence of $O(n)$ operations.

3-15. *[8]* Design a data structure that allows one to search, insert, and delete an integer X in $O(1)$ time (i.e. , constant time, independent of the total number of integers stored). Assume that $1 \leq X \leq n$ and that there are $m + n$ units of space available, where m is the maximum number of integers that can be in the table at any one time. (Hint: use two arrays $A[1..n]$ and $B[1..m]$.) You are not allowed to initialize either A or B, as that would take $O(m)$ or $O(n)$ operations. This means the arrays are full of random garbage to begin with, so you must be very careful.

Implementation Projects

3-16. *[5]* Implement versions of several different dictionary data structures, such as linked lists, binary trees, balanced binary search trees, and hash tables. Conduct experiments to assess the relative performance of these data structures in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. This application can be efficiently implemented by maintaining a dictionary of all distinct words that have appeared thus far in the text and inserting/reporting each word that is not found. Write a brief report with your conclusions.

3-17. *[5]* A Caesar shift (see Section 18.6 (page 641)) is a very simple class of ciphers for secret messages. Unfortunately, they can be broken using statistical properties of English. Develop a program capable of decrypting Caesar shifts of sufficiently long texts.

Interview Problems

3-18. *[3]* What method would you use to look up a word in a dictionary?

3-19. *[3]* Imagine you have a closet full of shirts. What can you do to organize your shirts for easy retrieval?

3-20. *[4]* Write a function to find the middle node of a singly-linked list.

3-21. *[4]* Write a function to compare whether two binary trees are identical. Identical trees have the same key value at each position and the same structure.

3-22. *[4]* Write a program to convert a binary search tree into a linked list.

3-23. *[4]* Implement an algorithm to reverse a linked list. Now do it without recursion.

3-24. *[5]* What is the best data structure for maintaining URLs that have been visited by a Web crawler? Give an algorithm to test whether a given URL has already been visited, optimizing both space and time.

3-25. *[4]* You are given a search string and a magazine. You seek to generate all the characters in search string by cutting them out from the magazine. Give an algorithm to efficiently determine whether the magazine contains all the letters in the search string.

3-26. *[4]* Reverse the words in a sentence—i.e., “My name is Chris” becomes “Chris is name My.” Optimize for time and space.

3-27. *[5]* Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.

3-28. *[5]* You have an unordered array X of n integers. Find the array M containing n elements where M_i is the product of all integers in X except for X_i. You may not use division. You can use extra memory. (Hint: There are solutions faster than $O(n^2)$.)

3-29. *[6]* Give an algorithm for finding an ordered word pair (e.g., “New York”) occurring with the greatest frequency in a given webpage. Which data structures would you use? Optimize both time and space.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

3-1. “Jolly Jumpers” – Programming Challenges 110201, UVA Judge 10038.

3-2. “Crypt Kicker” – Programming Challenges 110204, UVA Judge 843.

3-3. “Where’s Waldorf?” – Programming Challenges 110302, UVA Judge 10010.

3-4. “Crypt Kicker II” – Programming Challenges 110304, UVA Judge 850.

4 Sorting and Searching

Typical computer science students study the basic sorting algorithms at least three times before they graduate: first in introductory programming, then in data structures, and finally in their algorithms course. Why is sorting worth so much attention? There are several reasons:

- Sorting is the basic building block that many other algorithms are built around. By understanding sorting, we obtain an amazing amount of power to solve other problems.

- Most of the interesting ideas used in the design of algorithms appear in the context of sorting, such as divide-and-conquer, data structures, and randomized algorithms.

- Computers have historically spent more time sorting than doing anything else. A quarter of all mainframe cycles were spent sorting data [Knu98]. Sorting remains the most ubiquitous combinatorial algorithm problem in practice.

- Sorting is the most thoroughly studied problem in computer science. Literally dozens of different algorithms are known, most of which possess some particular advantage over all other algorithms in certain situations.

In this chapter, we will discuss sorting, stressing how sorting can be applied to solving other problems. In this sense, sorting behaves more like a data structure than a problem in its own right. We then give detailed presentations of several fundamental algorithms: heapsort, mergesort, quicksort, and distribution sort as examples of important algorithm design paradigms. Sorting is also represented by Section 14.1 (page 436) in the problem catalog.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_4,
© Springer-Verlag London Limited 2008

4.1 Applications of Sorting

We will review several sorting algorithms and their complexities over the course of this chapter. But the punch-line is this: clever sorting algorithms exist that run in $O(n \log n)$. This is a *big* improvement over naive $O(n^2)$ sorting algorithms for large values of n. Consider the following table:

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*

You might still get away with using a quadratic-time algorithm even if $n = 10,000$, but quadratic-time sorting is clearly ridiculous once $n \geq 100,000$.

Many important problems can be reduced to sorting, so we can use our clever $O(n \log n)$ algorithms to do work that might otherwise seem to require a quadratic algorithm. An important algorithm design technique is to use sorting as a basic building block, because many other problems become easy once a set of items is sorted.

Consider the following applications:

- *Searching* – Binary search tests whether an item is in a dictionary in $O(\log n)$ time, provided the keys are all sorted. Search preprocessing is perhaps the single most important application of sorting.

- *Closest pair* – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them? Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job, for a total of $O(n \log n)$ time including the sorting.

- *Element uniqueness* – Are there any duplicates in a given set of n items? This is a special case of the closest-pair problem above, where we ask if there is a pair separated by a gap of zero. The most efficient algorithm sorts the numbers and then does a linear scan though checking all adjacent pairs.

- *Frequency distribution* – Given a set of n items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting.

  To find out how often an arbitrary element k occurs, look up k using binary search in a sorted array of keys. By walking to the left of this point until the first the element is not k and then doing the same to the right, we can find

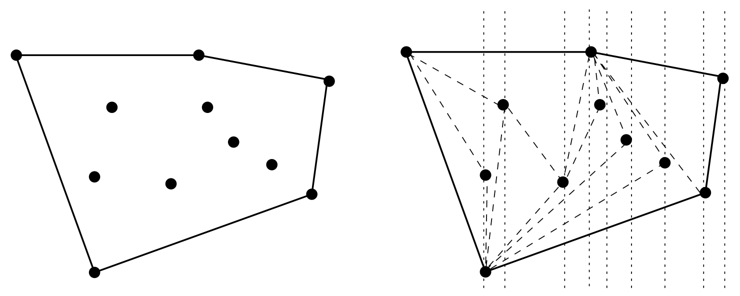


Figure 4.1: The convex hull of a set of points (l), constructed by left-to-right insertion.

this count in $O(\log n + c)$ time, where c is the number of occurrences of k. Even better, the number of instances of k can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$ (where ϵ is arbitrarily small) and then taking the difference of these positions.

- *Selection* – What is the kth largest item in an array? If the keys are placed in sorted order, the kth largest can be found in constant time by simply looking at the kth position of the array. In particular, the median element (see Section 14.3 (page 445)) appears in the $(n/2)$nd position in sorted order.

- *Convex hulls* – What is the polygon of smallest area that contains a given set of n points in two dimensions? The convex hull is like a rubber band stretched over the points in the plane and then released. It compresses to just cover the points, as shown in Figure 4.1(l). The convex hull gives a nice representation of the shape of the points and is an important building block for more sophisticated geometric algorithms, as discussed in the catalog in Section 17.2 (page 568).

  But how can we use sorting to construct the convex hull? Once you have the points sorted by x-coordinate, the points can be inserted from left to right into the hull. Since the right-most point is always on the boundary, we know that it will appear in the hull. Adding this new right-most point may cause others to be deleted, but we can quickly identify these points because they lie inside the polygon formed by adding the new point. See the example in Figure 4.1(r). These points will be neighbors of the previous point we inserted, so they will be easy to find and delete. The total time is linear after the sorting has been done.

While a few of these problems (namely median and selection) can be solved in linear time using more sophisticated algorithms, sorting provides quick and easy solutions to all of these problems. It is a rare application where the running time of sorting proves to be the bottleneck, especially a bottleneck that could have otherwise been removed using more clever algorithmics. Never be afraid to spend time sorting, provided you use an efficient sorting routine.

Take-Home Lesson: Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.

Stop and Think: Finding the Intersection

Problem: Give an efficient algorithm to determine whether two sets (of size m and n, respectively) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

Solution: At least three algorithms come to mind, all of which are variants of sorting and searching:

- *First sort the big set* – The big set can be sorted in $O(n \log n)$ time. We can now do a binary search with each of the m elements in the second, looking to see if it exists in the big set. The total time will be $O((n+m)\log n)$.

- *First sort the small set* – The small set can be sorted in $O(m \log m)$ time. We can now do a binary search with each of the n elements in the big set, looking to see if it exists in the small one. The total time will be $O((n+m)\log m)$.

- *Sort both sets* – Observe that once the two sets are sorted, we no longer have to do binary search to detect a common element. We can compare the smallest elements of the two sorted sets, and discard the smaller one if they are not identical. By repeating this idea recursively on the now smaller sets, we can test for duplication in linear time after sorting. The total cost is $O(n \log n + m \log m + n + m)$.

So, which of these is the fastest method? Clearly small-set sorting trumps big-set sorting, since $\log m < \log n$ when $m < n$. Similarly, $(n+m)\log m$ must be asymptotically less than $n \log n$, since $n+m < 2n$ when $m < n$. Thus, sorting the small set is the best of these options. Note that this is linear when m is constant in size.

Note that *expected* linear time can be achieved by hashing. Build a hash table containing the elements of both sets, and verify that collisions in the same bucket are in fact identical elements. In practice, this may be the best solution. ∎

4.2 Pragmatics of Sorting

We have seen many algorithmic applications of sorting, and we will see several efficient sorting algorithms. One issue stands between them: in what order do we want our items sorted?

The answers to this basic question are application-specific. Consider the following considerations:

- *Increasing or decreasing order?* – A set of keys S are sorted in *ascending* order when $S_i \leq S_{i+1}$ for all $1 \leq i < n$. They are in *descending order* when $S_i \geq S_{i+1}$ for all $1 \leq i < n$. Different applications call for different orders.

- *Sorting just the key or an entire record?* – Sorting a data set involves maintaining the integrity of complex data records. A mailing list of names, addresses, and phone numbers may be sorted by names as the key field, but it had better retain the linkage between names and addresses. Thus, we need to specify which field is the key field in any complex record, and understand the full extent of each record.

- *What should we do with equal keys?* Elements with equal key values will all bunch together in any total order, but sometimes the relative order among these keys matters. Suppose an encyclopedia contains both Michael Jordan (the basketball player) and Michael Jordan (the statistician). Which entry should appear first? You may need to resort to secondary keys, such as article size, to resolve ties in a meaningful way.

  Sometimes it is required to leave the items in the same relative order as in the original permutation. Sorting algorithms that automatically enforce this requirement are called *stable*. Unfortunately few fast algorithms are stable. Stability can be achieved for any sorting algorithm by adding the initial position as a secondary key.

  Of course we could make no decision about equal key order and let the ties fall where they may. But beware, certain efficient sort algorithms (such as quicksort) can run into quadratic performance trouble unless explicitly engineered to deal with large numbers of ties.

- *What about non-numerical data?* – Alphabetizing is the sorting of text strings. Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation. Is *Skiena* the same key as *skiena*? Is *Brown-Williams* before or after *Brown America*, and before or after *Brown, John*?

The right way to specify such matters to your sorting algorithm is with an application-specific pairwise-element *comparison function*. Such a comparison function takes pointers to record items a and b and returns "<" if $a < b$, ">" if $a > b$, or "=" if $a = b$.

By abstracting the pairwise ordering decision to such a comparison function, we can implement sorting algorithms independently of such criteria. We simply pass the comparison function in as an argument to the sort procedure. Any reasonable programming language has a built-in sort routine as a library function. You are almost always better off using this than writing your own routine. For example, the standard library for C contains the `qsort` function for sorting:

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
          int (*compare) (const void *, const void *));
```

The key to using `qsort` is realizing what its arguments do. It sorts the first `nel` elements of an array (pointed to by `base`), where each element is `width`-bytes long. Thus we can sort arrays of 1-byte characters, 4-byte integers, or 100-byte records, all by changing the value of `width`.

The ultimate desired order is determined by the `compare` function. It takes as arguments pointers to two `width`-byte elements, and returns a negative number if the first belongs before the second in sorted order, a positive number if the second belongs before the first, or zero if they are the same. Here is a comparison function to sort integers in increasing order:

```
int intcompare(int *i, int *j)
{
    if (*i > *j) return (1);
    if (*i < *j) return (-1);
    return (0);
}
```

This comparison function can be used to sort an array `a`, of which the first `n` elements are occupied, as follows:

```
    qsort(a, n, sizeof(int), intcompare);
```

`qsort` suggests that quicksort is the algorithm implemented in this library function, although this is usually irrelevant to the user.

4.3 Heapsort: Fast Sorting via Data Structures

Sorting is a natural laboratory for studying algorithm design paradigms, since many useful techniques lead to interesting sorting algorithms. The next several sections will introduce algorithmic design techniques motivated by particular sorting algorithms.

The alert reader should ask why we review the standard sorting when you are better off *not* implementing them and using built-in library functions instead. The answer is that the design techniques are very important for other algorithmic problems you are likely to encounter.

We start with data structure design, because one of the most dramatic algorithmic improvements via appropriate data structures occurs in sorting. Selection sort is a simple-to-code algorithm that repeatedly extracts the smallest remaining element from the unsorted part of the set:

```
SelectionSort($A$)
    For $i = 1$ to $n$ do
        $Sort[i]$ = Find-Minimum from $A$
        Delete-Minimum from $A$
    Return($Sort$)
```

A C language implementation of selection sort appeared back in Section 2.5.1 (page 41). There we partitioned the input array into sorted and unsorted regions. To find the smallest item, we performed a linear sweep through the unsorted portion of the array. The smallest item is then swapped with the ith item in the array before moving on to the next iteration. Selection sort performs n iterations, where the average iteration takes $n/2$ steps, for a total of $O(n^2)$ time.

But what if we improve the data structure? It takes $O(1)$ time to remove a particular item from an unsorted array once it has been located, but $O(n)$ time to find the smallest item. These are exactly the operations supported by priority queues. So what happens if we replace the data structure with a better priority queue implementation, either a heap or a balanced binary tree? Operations within the loop now take $O(\log n)$ time each, instead of $O(n)$. Using such a priority queue implementation speeds up selection sort from $O(n^2)$ to $O(n \log n)$.

The name typically given to this algorithm, *heapsort*, obscures the relationship between them, but heapsort is nothing but an implementation of selection sort using the right data structure.

4.3.1 Heaps

Heaps are a simple and elegant data structure for efficiently supporting the priority queue operations insert and extract-min. They work by maintaining a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the minimum element can be quickly identified).

Power in any hierarchically-structured organization is reflected by a tree, where each node in the tree represents a person, and edge (x, y) implies that x directly supervises (or dominates) y. The fellow at the root sits at the "top of the heap."

In this spirit, a *heap-labeled tree* is defined to be a binary tree such that the key labeling of each node *dominates* the key labeling of each of its children. In a

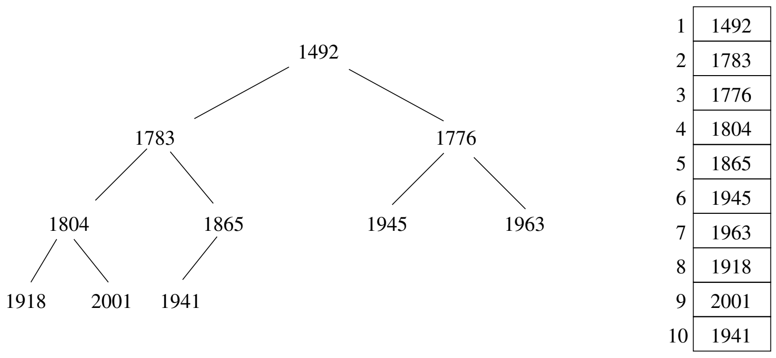


Figure 4.2: A heap-labeled tree of important years from American history (l), with the corresponding implicit heap representation (r)

min-heap, a node dominates its children by containing a smaller key than they do, while in a *max-heap* parent nodes dominate by being bigger. Figure 4.2(l) presents a min-heap ordered tree of red-letter years in American history (kudos to you if you can recall what happened each year).

The most natural implementation of this binary tree would store each key in a node with pointers to its two children. As with binary search trees, the memory used by the pointers can easily outweigh the size of keys, which is the data we are really interested in.

The heap is a slick data structure that enables us to represent binary trees without using any pointers. We will store data as an array of keys, and use the position of the keys to *implicitly* satisfy the role of the pointers.

We will store the root of the tree in the first position of the array, and its left and right children in the second and third positions, respectively. In general, we will store the 2^l keys of the lth level of a complete binary tree from left-to-right in positions 2^{l-1} to $2^l - 1$, as shown in Figure 4.2(r). We assume that the array starts with index 1 to simplify matters.

```
typedef struct {
        item_type q[PQ_SIZE+1];       /* body of queue */
        int n;                        /* number of queue elements */
} priority_queue;
```

What is especially nice about this representation is that the positions of the parent and children of the key at position k are readily determined. The *left* child of k sits in position $2k$ and the right child in $2k + 1$, while the parent of k holds court in position $\lfloor n/2 \rfloor$. Thus we can move around the tree without any pointers.

```
pq_parent(int n)
{
        if (n == 1) return(-1);
        else return((int) n/2);     /* implicitly take floor(n/2) */
}

pq_young_child(int n)
{
        return(2 * n);
}
```

So, we can store any binary tree in an array without pointers. What is the catch? Suppose our height h tree was sparse, meaning that the number of nodes $n < 2^h$. All missing internal nodes still take up space in our structure, since we must represent a full binary tree to maintain the positional mapping between parents and children.

Space efficiency thus demands that we not allow holes in our tree—i.e., that each level be packed as much as it can be. If so, only the last level may be incomplete. By packing the elements of the last level as far to the left as possible, we can represent an n-key tree using exactly n elements of the array. If we did not enforce these structural constraints, we might need an array of size 2^n to store the same elements. Since all but the last level is always filled, the height h of an n element heap is logarithmic because:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geq n$$

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

This implicit representation of binary trees saves memory, but is less flexible than using pointers. We cannot store arbitrary tree topologies without wasting large amounts of space. We cannot move subtrees around by just changing a single pointer, only by explicitly moving each of the elements in the subtree. This loss of flexibility explains why we cannot use this idea to represent binary search trees, but it works just fine for heaps.

Stop and Think: Who's where in the heap?

Problem: How can we efficiently search for a particular key in a heap?

Solution: We can't. Binary search does not work because a heap is not a binary search tree. We know almost nothing about the relative order of the $n/2$ leaf elements in a heap—certainly nothing that lets us avoid doing linear search through them. ∎

4.3.2 Constructing Heaps

Heaps can be constructed incrementally, by inserting each new element into the left-most open spot in the array, namely the $(n+1)$st position of a previously n-element heap. This ensures the desired balanced shape of the heap-labeled tree, but does not necessarily maintain the dominance ordering of the keys. The new key might be less than its parent in a min-heap, or greater than its parent in a max-heap.

The solution is to swap any such dissatisfied element with its parent. The old parent is now happy, because it is properly dominated. The other child of the old parent is still happy, because it is now dominated by an element even more extreme than its previous parent. The new element is now happier, but may still dominate its new parent. We now recur at a higher level, *bubbling up* the new key to its proper position in the hierarchy. Since we replace the root of a subtree by a larger one at each step, we preserve the heap order elsewhere.

```
pq_insert(priority_queue *q, item_type x)
{
  if (q->n >= PQ_SIZE)
         printf("Warning: priority queue overflow insert x=%d\n",x);
  else {
         q->n = (q->n) + 1;
         q->q[ q->n ] = x;
         bubble_up(q, q->n);
  }
}

bubble_up(priority_queue *q, int p)
{
    if (pq_parent(p) == -1) return; /* at root of heap, no parent */

    if (q->q[pq_parent(p)] > q->q[p]) {
            pq_swap(q,p,pq_parent(p));
            bubble_up(q, pq_parent(p));
    }
}
```

This swap process takes constant time at each level. Since the height of an n-element heap is $\lfloor \lg n \rfloor$, each insertion takes at most $O(\log n)$ time. Thus an initial heap of n elements can be constructed in $O(n \log n)$ time through n such insertions:

```
pq_init(priority_queue *q)
{
        q->n = 0;
}
```

```
make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        pq_init(q);
        for (i=0; i<n; i++)
                pq_insert(q, s[i]);
}
```

4.3.3 Extracting the Minimum

The remaining priority queue operations are identifying and deleting the dominant element. Identification is easy, since the top of the heap sits in the first position of the array.

Removing the top element leaves a hole in the array. This can be filled by moving the element from the *right-most* leaf (sitting in the nth position of the array) into the first position.

The shape of the tree has been restored but (as after insertion) the labeling of the root may no longer satisfy the heap property. Indeed, this new root may be dominated by both of its children. The root of this min-heap should be the smallest of three elements, namely the current root and its two children. If the current root is dominant, the heap order has been restored. If not, the dominant child should be swapped with the root and the problem pushed down to the next level.

This dissatisfied element *bubbles down* the heap until it dominates all its children, perhaps by becoming a leaf node and ceasing to have any. This percolate-down operation is also called *heapify*, because it merges two heaps (the subtrees below the original root) with a new key.

```
item_type extract_min(priority_queue *q)
{
        int min = -1;                   /* minimum value */

        if (q->n <= 0) printf("Warning: empty priority queue.\n");
        else {
                min = q->q[1];

                q->q[1] = q->q[ q->n ];
                q->n = q->n - 1;
                bubble_down(q,1);
        }

        return(min);
}
```

```
bubble_down(priority_queue *q, int p)
{
        int c;                          /* child index */
        int i;                          /* counter */
        int min_index;                  /* index of lightest child */

        c = pq_young_child(p);
        min_index = p;

        for (i=0; i<=1; i++)
                if ((c+i) <= q->n) {
                   if (q->q[min_index] > q->q[c+i]) min_index = c+i;
                }

        if (min_index != p) {
                pq_swap(q,p,min_index);
                bubble_down(q, min_index);
        }
}
```

We will reach a leaf after $\lfloor \lg n \rfloor$ `bubble_down` steps, each constant time. Thus root deletion is completed in $O(\log n)$ time.

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

```
heapsort(item_type s[], int n)
{
        int i;                  /* counters */
        priority_queue q;       /* heap for heapsort */

        make_heap(&q,s,n);

        for (i=0; i<n; i++)
                s[i] = extract_min(&q);
}
```

Heapsort is a great sorting algorithm. It is simple to program; indeed, the complete implementation has been presented above. It runs in worst-case $O(n \log n)$ time, which is the best that can be expected from any sorting algorithm. It is an *in-place* sort, meaning it uses no extra memory over the array containing the elements to be sorted. Although other algorithms prove slightly faster in practice, you won't go wrong using heapsort for sorting data that sits in the computer's main memory.

Priority queues are very useful data structures. Recall they were the hero of the war story described in Section 3.6 (page 85). A complete set of priority queue implementations is presented in catalog Section 12.2 (page 373).

4.3.4 Faster Heap Construction (*)

As we have seen, a heap can be constructed on n elements by incremental insertion in $O(n \log n)$ time. Surprisingly, heaps can be constructed even faster by using our `bubble_down` procedure and some clever analysis.

Suppose we pack the n keys destined for our heap into the first n elements of our priority-queue array. The shape of our heap will be right, but the dominance order will be all messed up. How can we restore it?

Consider the array in reverse order, starting from the last (nth) position. It represents a leaf of the tree and so dominates its nonexistent children. The same is the case for the last $n/2$ positions in the array, because all are leaves. If we continue to walk backwards through the array we will finally encounter an internal node with children. This element may not dominate its children, but its children represent well-formed (if small) heaps.

This is exactly the situation the `bubble_down` procedure was designed to handle, restoring the heap order of arbitrary root element sitting on top of two sub-heaps. Thus we can create a heap by performing $n/2$ non-trivial calls to the `bubble_down` procedure:

```
make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        q->n = n;
        for (i=0; i<n; i++) q->q[i+1] = s[i];

        for (i=q->n; i>=1; i--) bubble_down(q,i);
}
```

Multiplying the number of calls to `bubble_down` (n) times an upper bound on the cost of each operation ($O(\log n)$) gives us a running time analysis of $O(n \log n)$. This would make it no faster than the incremental insertion algorithm described above.

But note that it is indeed an *upper bound*, because only the last insertion will actually take $\lfloor \lg n \rfloor$ steps. Recall that `bubble_down` takes time proportional to the height of the heaps it is merging. Most of these heaps are extremely small. In a full binary tree on n nodes, there are $n/2$ nodes that are leaves (i.e. , height 0), $n/4$

nodes that are height 1, $n/8$ nodes that are height 2, and so on. In general, there are at most $\lceil n/2^{h+1}\rceil$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n\rfloor} \lceil n/2^{h+1}\rceil h \le n \sum_{h=0}^{\lfloor \lg n\rfloor} h/2^h \le 2n$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum, but rest assured that the puny contribution of the numerator (h) is crushed by the denominator (2^h). The series quickly converges to linear.

Does it matter that we can construct heaps in linear time instead of $O(n \log n)$? Usually not. The construction time did not dominate the complexity of heapsort, so improving the construction time does not improve its worst-case performance. Still, it is an impressive display of the power of careful analysis, and the free lunch that geometric series convergence can sometimes provide.

Stop and Think: Where in the Heap?

Problem: Given an array-based heap on n elements and a real number x, efficiently determine whether the kth smallest element in the heap is greater than or equal to x. Your algorithm should be $O(k)$ in the worst-case, independent of the size of the heap. Hint: you do not have to find the kth smallest element; you need only determine its relationship to x.

Solution: There are at least two different ideas that lead to correct but inefficient algorithms for this problem:

1. Call extract-min k times, and test whether all of these are less than x. This explicitly sorts the first k elements and so gives us more information than the desired answer, but it takes $O(k \log n)$ time to do so.

2. The kth smallest element cannot be deeper than the kth level of the heap, since the path from it to the root must go through elements of decreasing value. Thus we can look at all the elements on the first k levels of the heap, and count how many of them are less than x, stopping when we either find k of them or run out of elements. This is correct, but takes $O(\min(n, 2^k))$ time, since the top k elements have 2^k elements.

An $O(k)$ solution can look at only k elements smaller than x, plus at most $O(k)$ elements greater than x. Consider the following recursive procedure, called at the root with $i = 1$ with *count* $= k$:

```
int heap_compare(priority_queue *q, int i, int count, int x)
{
        if ((count <= 0) || (i > q->n) return(count);

        if (q->q[i] < x) {
            count = heap_compare(q, pq_young_child(i), count-1, x);
            count = heap_compare(q, pq_young_child(i)+1, count, x);
        }

        return(count);
}
```

If the root of the min-heap is $\geq x$, then no elements in the heap can be less than x, as by definition the root must be the smallest element. This procedure searches the children of all nodes of weight smaller than x until either (a) we have found k of them, when it returns 0, or (b) they are exhausted, when it returns a value greater than zero. Thus it will find enough small elements if they exist.

But how long does it take? The only nodes whose children we look at are those $< x$, and at most k of these in total. Each have at most visited two children, so we visit at most $3k$ nodes, for a total time of $O(k)$. ∎

4.3.5 Sorting by Incremental Insertion

Now consider a different approach to sorting via efficient data structures. Select an arbitrary element from the unsorted set, and put it in the proper position in the sorted set.

```
InsertionSort($A$)
    $A[0] = -\infty$
    for $i = 2$ to $n$ do
        $j = i$
        while $(A[j] < A[j-1])$ do
            swap$(A[j], A[j-1])$
            $j = j - 1$
```

A C language implementation of insertion sort appeared in Section 2.5.2 (page 43). Although insertion sort takes $O(n^2)$ in the worst case, it performs considerably better if the data is almost sorted, since few iterations of the inner loop suffice to sift it into the proper position.

Insertion sort is perhaps the simplest example of the *incremental insertion* technique, where we build up a complicated structure on n items by first building it on $n-1$ items and then making the necessary changes to add the last item. Incremental insertion proves a particularly useful technique in geometric algorithms.

Note that faster sorting algorithms based on incremental insertion follow from more efficient data structures. Insertion into a balanced search tree takes $O(\log n)$ per operation, or a total of $O(n \log n)$ to construct the tree. An in-order traversal reads through the elements in sorted order to complete the job in linear time.

4.4 War Story: Give me a Ticket on an Airplane

I came into this particular job seeking justice. I'd been retained by an air travel company to help design an algorithm to find the cheapest available airfare from city x to city y. Like most of you, I suspect, I'd been baffled at the crazy price fluctuations of ticket prices under modern "yield management." The price of flights seems to soar far more efficiently than the planes themselves. The problem, it seemed to me, was that airlines never wanted to show the true cheapest price. By doing my job right, I could make damned sure they would show it to me next time.

"Look," I said at the start of the first meeting. "This can't be so hard. Construct a graph with vertices corresponding to airports, and add an edge between each airport pair (u, v) which shows a direct flight from u to v. Set the weight of this edge equal to the cost of the cheapest available ticket from u to v. Now the cheapest fair from x to y is given by the shortest x-y path in this graph. This path/fare can be found using Dijkstra's shortest path algorithm. Problem solved!" I announced, waiving my hand with a flourish.

The assembled cast of the meeting nodded thoughtfully, then burst out laughing. It was I who needed to learn something about the overwhelming complexity of air travel pricing. There are literally millions of different fares available at any time, with prices changing several times daily. Restrictions on the availability of a particular fare in a particular context is enforced by a complicated set of pricing rules. These rules are an industry-wide kludge—a complicated structure with little in the way of consistent logical principles, which is exactly what we would need to search efficiently for the minimum fare. My favorite rule exceptions applied only to the country of Malawi. With a population of only 12 million and per-capita income of \$596 (179th in the world), they prove to be an unexpected powerhouse shaping world aviation price policy. Accurately pricing any air itinerary requires at least implicit checks to ensure the trip doesn't take us through Malawi.

Part of the real problem is that there can easily be 100 different fares for the first flight leg, say from Los Angeles (LAX) to Chicago (ORD), and a similar number for each subsequent leg, say from Chicago to New York (JFK). The cheapest possible LAX-ORD fare (maybe an AARP children's price) might not be combinable with the cheapest ORD-JFK fare (perhaps a pre-Ramadan special that can only be used with subsequent connections to Mecca).

After being properly chastised for oversimplifying the problem, I got down to work. I started by reducing the problem to the simplest interesting case. "So, you

X	Y
\$100	\$50
\$110	\$75
\$130	\$125

X+Y	
\$150	(1,1)
\$160	(2,1)
\$175	(1,2)
\$180	(3,1)
\$185	(2,2)
\$205	(2,3)
\$225	(1,3)
\$235	(2,3)
\$255	(3,3)

Figure 4.3: Sorting the pairwise sums of lists X and Y.

need to find the cheapest two-hop fare that passes your rule tests. Is there a way to decide in advance which pairs will pass without explicitly testing them?"

"No, there is no way to tell," they assured me. "We can only consult a black box routine to decide whether a particular price is available for the given itinerary/travelers."

"So our goal is to call this black box on the fewest number of combinations. This means evaluating all possible fare combinations in order from cheapest to most expensive, and stopping as soon as we encounter the first legal combination."

"Right."

"Why not construct the $m \times n$ possible price pairs, sort them in terms of cost, and evaluate them in sorted order? Clearly this can be done in $O(nm \log(nm))$ time."[1]

"That is basically what we do now, but it is quite expensive to construct the full set of $m \times n$ pairs, since the first one might be all we need."

I caught a whiff of an interesting problem. "So what you really want is an efficient data structure to repeatedly return the *next* most expensive pair without constructing all the pairs in advance."

This was indeed an interesting problem. Finding the largest element in a set under insertion and deletion is *exactly* what priority queues are good for. The catch here is that we could not seed the priority queue with all values in advance. We had to insert new pairs into the queue after each evaluation.

I constructed some examples, like the one in Figure 4.3. We could represent each fare by the list indexes of its two components. The cheapest single fare will certainly be constructed by adding up the cheapest component from both lists,

[1]The question of whether all such sums can be sorted faster than nm arbitrary integers is a notorious open problem in algorithm theory. See [Fre76, Lam92] for more on $X + Y$ sorting, as the problem is known.

described $(1,1)$. The second cheapest fare would be made from the head of one list and the second element of another, and hence would be either $(1,2)$ or $(2,1)$. Then it gets more complicated. The third cheapest could either be the unused pair above or $(1,3)$ or $(3,1)$. Indeed it would have been $(3,1)$ in the example above if the third fare of X had been \$120.

"Tell me," I asked. "Do we have time to sort the two respective lists of fares in increasing order?"

"Don't have to." the leader replied. "They come out in sorted order from the database."

Good news. That meant there was a natural order to the pair values. We never need to evaluate the pairs $(i+1,j)$ or $(i,j+1)$ before (i,j), because they clearly define more expensive fares.

"Got it!," I said. "We will keep track of index pairs in a priority queue, with the sum of the fare costs as the key for the pair. Initially we put only pair $(1,1)$ on the queue. If it proves it is not feasible, we put its two successors on—namely $(1,2)$ and $(2,1)$. In general, we enqueue pairs $(i+1,j)$ and $(i,j+1)$ after evaluating/rejecting pair (i,j). We will get through all the pairs in the right order if we do so."

The gang caught on quickly. "Sure. But what about duplicates? We will construct pair (x,y) two different ways, both when expanding $(x-1,y)$ and $(x,y-1)$."

"You are right. We need an extra data structure to guard against duplicates. The simplest might be a hash table to tell us whether a given pair exists in the priority queue before we insert a duplicate. In fact, we will never have more than n active pairs in our data structure, since there can only be one pair for each distinct value of the first coordinate."

And so it went. Our approach naturally generalizes to itineraries with more than two legs, (a complexity which grows with the number of legs). The best-first evaluation inherent in our priority queue enabled the system to stop as soon as it found the provably cheapest fare. This proved to be fast enough to provide interactive response to the user. That said, I haven't noticed my travel tickets getting any cheaper.

4.5 Mergesort: Sorting by Divide-and-Conquer

Recursive algorithms reduce large problems into smaller ones. A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements. This algorithm is called *mergesort*, recognizing the importance of the interleaving operation:

```
Mergesort($A[1,n]$)
    Merge( MergeSort($A[1,\lfloor n/2\rfloor]$), MergeSort($A[\lfloor n/2\rfloor+1,n]$) )
```

The basis case of the recursion occurs when the subarray to be sorted consists of a single element, so no rearrangement is possible. A trace of the execution of

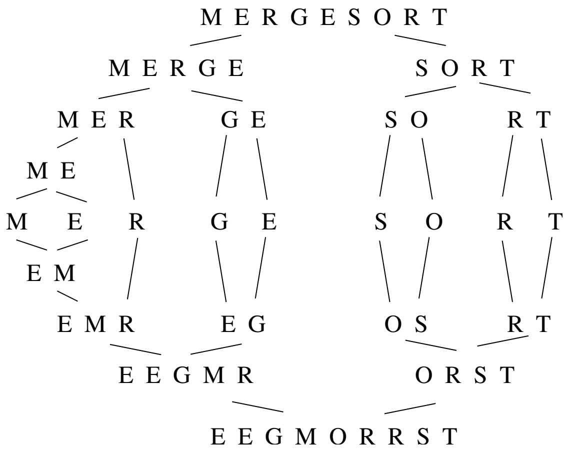


Figure 4.4: Animation of mergesort in action

mergesort is given in Figure 4.4. Picture the action as it happens during an in-order traversal of the top tree, with the array-state transformations reported in the bottom, reflected tree.

The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list. We could concatenate them into one list and call heapsort or some other sorting algorithm to do it, but that would just destroy all the work spent sorting our component lists.

Instead we can *merge* the two lists together. Observe that the smallest overall item in two lists sorted in increasing order (as above) must sit at the top of one of the two lists. This smallest element can be removed, leaving two sorted lists behind—one slightly shorter than before. The second smallest item overall must be atop one of these lists. Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n - 1$ comparisons or $O(n)$ total work.

What is the total running time of mergesort? It helps to think about how much work is done at each level of the execution tree. If we assume for simplicity that n is a power of two, the kth level consists of all the 2^k calls to `mergesort` processing subranges of $n/2^k$ elements.

The work done on the $(k = 0)$th level involves merging two sorted lists, each of size $n/2$, for a total of at most $n-1$ comparisons. The work done on the $(k = 1)$th level involves merging two pairs of sorted lists, each of size $n/4$, for a total of at most $n-2$ comparisons. In general, the work done on the kth level involves merging 2^k pairs sorted list, each of size $n/2^{k+1}$, for a total of at most $n - 2^k$ comparisons. *Linear work is done merging all the elements on each level.* Each of the n elements appears in exactly one subproblem on each level. The most expensive case (in terms of comparsions) is actually the top level.

The number of elements in a subproblem gets halved at each level. Thus the number of times we can halve n until we get to 1 is $\lceil \lg_2 n \rceil$. Because the recursion goes $\lg n$ levels deep, and a linear amount of work is done per level, mergesort takes $O(n \log n)$ time in the worst case.

Mergesort is a great algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort. Its primary disadvantage is the need for an auxilliary buffer when sorting arrays. It is easy to merge two sorted linked lists without using any extra space, by just rearranging the pointers. However, to merge two sorted arrays (or portions of an array), we need use a third array to store the result of the merge to avoid stepping on the component arrays. Consider merging $\{4, 5, 6\}$ with $\{1, 2, 3\}$, packed from left to right in a single array. Without a buffer, we would overwrite the elements of the top half during merging and lose them.

Mergesort is a classic divide-and-conquer algorithm. We are ahead of the game whenever we can break one large problem into two smaller problems, because the smaller problems are easier to solve. The trick is taking advantage of the two partial solutions to construct a solution of the full problem, as we did with the merge operation.

Implementation

The divide-and-conquer `mergesort` routine follows naturally from the pseudocode:

```
mergesort(item_type s[], int low, int high)
{
        int i;                  /* counter */
        int middle;             /* index of middle element */

        if (low < high) {
                middle = (low+high)/2;
                mergesort(s,low,middle);
                mergesort(s,middle+1,high);
                merge(s, low, middle, high);
        }
}
```

More challenging turns out to be the details of how the merging is done. The problem is that we must put our merged array somewhere. To avoid losing an element by overwriting it in the course of the merge, we first copy each subarray to a separate queue and merge these elements back into the array. In particular:

```
merge(item_type s[], int low, int middle, int high)
{
  int i;                  /* counter */
  queue buffer1, buffer2; /* buffers to hold elements for merging */

  init_queue(&buffer1);
  init_queue(&buffer2);

  for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
  for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

  i = low;
  while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
          if (headq(&buffer1) <= headq(&buffer2))
                  s[i++] = dequeue(&buffer1);
          else
                  s[i++] = dequeue(&buffer2);
  }

  while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
  while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

4.6 Quicksort: Sorting by Randomization

Suppose we select a random item p from the n items we seek to sort. *Quicksort* (shown in action in Figure 4.5) separates the $n - 1$ other items into two piles: a low pile containing all the elements that appear before p in sorted order and a high pile containing all the elements that appear after p in sorted order. Low and high denote the array positions we place the respective piles, leaving a single slot between them for p.

Such partitioning buys us two things. First, the pivot element p ends up in the exact array position it will reside in the the final sorted order. Second, after partitioning no element flops to the other side in the final sorted order. *Thus we can now sort the elements to the left and the right of the pivot independently!* This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem. The algorithm must be correct since each element ultimately ends up in the proper position:

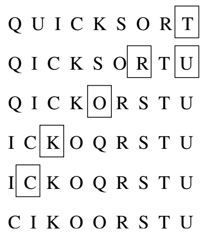


Figure 4.5: Animation of quicksort in action

```
quicksort(item_type s[], int l, int h)
{
        int p;                  /* index of partition */

        if ((h-l)>0) {
                p = partition(s,l,h);
                quicksort(s,l,p-1);
                quicksort(s,p+1,h);
        }
}
```

We can partition the array in one linear scan for a particular pivot element by maintaining three sections of the array: less than the pivot (to the left of `firsthigh`), greater than or equal to the pivot (between `firsthigh` and `i`), and unexplored (to the right of `i`), as implemented below:

```
int partition(item_type s[], int l, int h)
{
        int i;                  /* counter */
        int p;                  /* pivot element index */
        int firsthigh;          /* divider position for pivot element */

        p = h;
        firsthigh = l;
        for (i=l; i<h; i++)
                if (s[i] < s[p]) {
                        swap(&s[i],&s[firsthigh]);
                        firsthigh ++;
                }
        swap(&s[p],&s[firsthigh]);

        return(firsthigh);
}
```

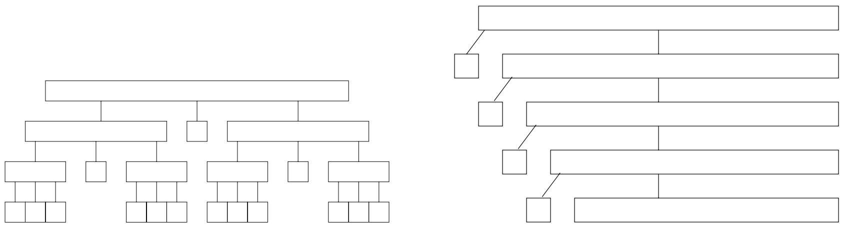


Figure 4.6: The best-case (l) and worst-case (r) recursion trees for quicksort

Since the partitioning step consists of at most n swaps, it takes linear time in the number of keys. But how long does the entire quicksort take? As with mergesort, quicksort builds a recursion tree of nested subranges of the n-element array. As with mergesort, quicksort spends linear time processing (now `partition`ing instead of `merge`ing) the elements in each subarray on each level. As with mergesort, quicksort runs in $O(n \cdot h)$ time, where h is the height of the recursion tree.

The difficulty is that the height of the tree depends upon where the pivot element ends up in each partition. If we get very lucky and *happen* to repeatedly pick the median element as our pivot, the subproblems are always half the size of the previous level. The height represents the number of times we can halve n until we get down to 1, or at most $\lceil \lg_2 n \rceil$. This happy situation is shown in Figure 4.6(l), and corresponds to the best case of quicksort.

Now suppose we consistently get unlucky, and our pivot element always splits the array as unequally as possible. This implies that the pivot element is always the biggest or smallest element in the sub-array. After this pivot settles into its position, we are left with one subproblem of size $n - 1$. We spent linear work and reduced the size of our problem by one measly element, as shown in Figure 4.6(r). It takes a tree of height $n - 1$ to chop our array down to one element per level, for a worst case time of $\Theta(n^2)$.

Thus, the worst case for quicksort is worse than heapsort or mergesort. To justify its name, quicksort had better be good in the average case. Understanding why requires some intuition about random sampling.

4.6.1 Intuition: The Expected Case for Quicksort

The expected performance of quicksort depends upon the height of the partition tree constructed by random pivot elements at each step. Mergesort ran in $O(n \log n)$ time because we split the keys into two equal halves, sorted them recursively, and then merged the halves in linear time. Thus, whenever our pivot element is near the center of the sorted array (i.e. , the pivot is close to the median element), we get a good split and realize the same performance as mergesort.

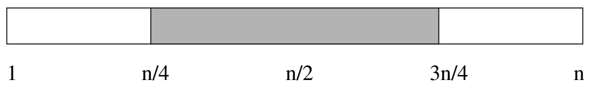


Figure 4.7: Half the time, the pivot is close to the median element

I will give an intuitive explanation of why quicksort is $O(n \log n)$ in the average case. How likely is it that a randomly selected pivot is a good one? The best possible selection for the pivot would be the median key, because exactly half of elements would end up left, and half the elements right, of the pivot. Unfortunately, we only have a probability of $1/n$ of randomly selecting the median as pivot, which is quite small.

Suppose a key is a *good enough* pivot if it lies is in the center half of the sorted space of keys—i.e. , those ranked from $n/4$ to $3n/4$ in the space of all keys to be sorted. Such *good enough* pivot elements are quite plentiful, since half the elements lie closer to the middle than one of the two ends (see Figure 4.7). Thus, on each selection we will pick a *good enough* pivot with probability of 1/2.

Can you flip a coin so it comes up tails each time? Not without cheating. If you flip a fair coin n times, it will come out heads about half the time. Let heads denote the chance of picking a *good enough* pivot.

The worst possible *good enough* pivot leaves the bigger half of the space partition with $3n/4$ items. What is the height h_g of a quicksort partition tree constructed repeatedly from the worst-possible *good enough* pivot? The deepest path through this tree passes through partitions of size $n, (3/4)n, (3/4)^2 n, \ldots$, down to 1. How many times can we multiply n by 3/4 until it gets down to 1?

$$(3/4)^{h_g} n = 1 \Rightarrow n = (4/3)^{h_g}$$

so $h_g = \log_{4/3} n$.

But only half of all randomly selected pivots will be *good enough*. The rest we classify as *bad*. The worst of these bad pivots will do essentially nothing to reduce the partition size along the deepest path. The deepest path from the root through a typical randomly-constructed quicksort partition tree will pass through roughly equal numbers of good-enough and bad pivots. Since the expected number of good splits and bad splits is the same, the bad splits can only double the height of the tree, so $h \approx 2h_g = 2 \log_{4/3} n$, which is clearly $\Theta(\log n)$.

On average, random quicksort partition trees (and by analogy, binary search trees under random insertion) are very good. More careful analysis shows the average height after n insertions is approximately $2 \ln n$. Since $2 \ln n \approx 1.386 \lg_2 n$, this is only 39% taller than a perfectly balanced binary tree. Since quicksort does $O(n)$ work partitioning on each level, the average time is $O(n \log n)$. If we are *extremely* unlucky and our randomly selected elements always are among the largest or smallest element in the array, quicksort turns into selection sort and runs in $O(n^2)$. However, the odds against this are vanishingly small.

4.6.2 Randomized Algorithms

There is an important subtlety about the expected case $O(n \log n)$ running time for quicksort. Our quicksort implementation above selected the last element in each sub-array as the pivot. Suppose this program were given a sorted array as input. If so, at each step it would pick the worst possible pivot and run in quadratic time.

For any deterministic method of pivot selection, there exists a worst-case input instance which will doom us to quadratic time. The analysis presented above made no claim stronger than:

> "Quicksort runs in $\Theta(n \log n)$ time, with high probability, *if* you give me randomly ordered data to sort."

But now suppose we add an initial step to our algorithm where we randomly permute the order of the n elements before we try to sort them. Such a permutation can be constructed in $O(n)$ time (see Section 13.7 for details). This might seem like wasted effort, but it provides the guarantee that we can expect $\Theta(n \log n)$ running time *whatever* the initial input was. The worst case performance still can happen, but it depends only upon how unlucky we are. There is no longer a well-defined "worst case" input. We now can say

> "Randomized quicksort runs in $\Theta(n \log n)$ time on *any* input, with high probability."

Alternately, we could get the same guarantee by selecting a random element to be the pivot at each step.

Randomization is a powerful tool to improve algorithms with bad worst-case but good average-case complexity. It can be used to make algorithms more robust to boundary cases and more efficient on highly structured input instances that confound heuristic decisions (such as sorted input to quicksort). It often lends itself to simple algorithms that provide randomized performance guarantees which are otherwise obtainable only using complicated deterministic algorithms.

Proper analysis of randomized algorithms requires some knowledge of probability theory, and is beyond the scope of this book. However, some of the approaches to designing efficient randomized algorithms are readily explainable:

- *Random sampling* – Want to get an idea of the median value of n things but don't have either the time or space to look at them all? Select a small random sample of the input and study those, for the results should be representative.

  This is the idea behind opinion polling. Biases creep in unless you take a truly *random* sample, as opposed to the first x people you happen to see. To avoid bias, actual polling agencies typically dial random phone numbers and hope someone answers.

- *Randomized hashing* – We have claimed that hashing can be used to implement dictionary operations in $O(1)$ "expected-time." However, for any hash function there is a given worst-case set of keys that all get hashed to the same bucket. But now suppose we randomly select our hash function from a large family of good ones as the first step of our algorithm. We get the same type of improved guarantee that we did with randomized quicksort.

- *Randomized search* – Randomization can also be used to drive search techniques such as simulated annealing, as will be discussed in detail in Section 7.5.3 (page 254).

Stop and Think: Nuts and Bolts

Problem: The *nuts and bolts* problem is defined as follows. You are given a collection of n bolts of different widths, and n corresponding nuts. You can test whether a given nut and bolt fit together, from which you learn whether the nut is too large, too small, or an exact match for the bolt. The differences in size between pairs of nuts or bolts are too small to see by eye, so you cannot compare the sizes of two nuts or two bolts directly. You are to match each bolt to each nut.

Give an $O(n^2)$ algorithm to solve the nuts and bolts problem. Then give a randomized $O(n \log n)$ expected time algorithm for the same problem.

Solution: The brute force algorithm for matching nuts and bolts starts with the first bolt and compares it to each nut until we find a match. In the worst case, this will require n comparisons. Repeating this for each successive bolt on all remaining nuts yields a quadratic-comparison algorithm.

What if we pick a random bolt and try it? On average, we would expect to get about halfway through the set of nuts before we found the match, so this randomized algorithm would do half the work as the worst case. That counts as some kind of improvement, although not an asymptotic one.

Randomized quicksort achieves the desired expected-case running time, so a natural idea is to emulate it on the nuts and bolts problem. Indeed, sorting both the nuts and bolts by size would yield a matching, since the ith largest nut must match the ith largest bolt.

The fundamental step in quicksort is partitioning elements around a pivot. Can we partition nuts and bolts around a randomly selected bolt b? Certainly we can partition the nuts into those of size less than b and greater than b. But decomposing the problem into two halves requires partitioning the bolts as well, and we cannot compare bolt against bolt. But once we find the matching nut to b we can use it to partition the bolts accordingly. In $2n - 2$ comparisons, we partition the nuts and bolts, and the remaining analysis follows directly from randomized quicksort.

What is interesting about this problem is that no simple deterministic algorithm for nut and bolt sorting is known. It illustrates how randomization makes the bad case go away, leaving behind a simple and beautiful algorithm. ∎

4.6.3 Is Quicksort Really Quick?

There is a clear, asymptotic difference between an $\Theta(n \log n)$ algorithm and one that runs in $\Theta(n^2)$. Thus, only the most obstinate reader would doubt my claim that mergesort, heapsort, and quicksort should all outperform insertion sort or selection sort on large enough instances.

But how can we compare two $\Theta(n \log n)$ algorithms to decide which is faster? How can we prove that quicksort is really quick? Unfortunately, the RAM model and Big Oh analysis provide too coarse a set of tools to make that type of distinction. When faced with algorithms of the same asymptotic complexity, implementation details and system quirks such as cache performance and memory size may well prove to be the decisive factor.

What we can say is that experiments show that where a properly implemented quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort. The primary reason is that the operations in the innermost loop are simpler. But I can't argue with you if you don't believe me when I say quicksort is faster. It is a question whose solution lies outside the analytical tools we are using. The best way to tell is to implement both algorithms and experiment.

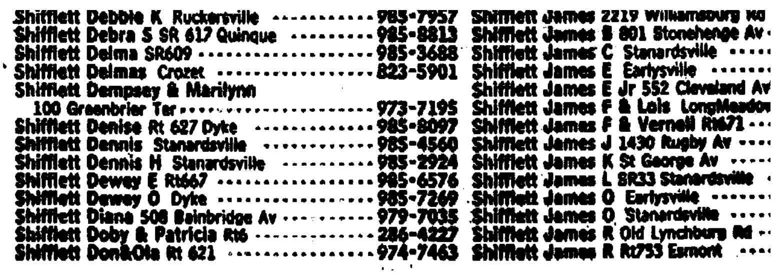
4.7 Distribution Sort: Sorting via Bucketing

We could sort sorting names for the telephone book by partitioning them according to the first letter of the last name. This will create 26 different piles, or buckets, of names. Observe that any name in the J pile must occur after every name in the I pile, but before any name in the K pile. Therefore, we can proceed to sort each pile individually and just concatenate the bunch of sorted piles together at the end.

If the names are distributed evenly among the buckets, the resulting 26 sorting problems should each be substantially smaller than the original problem. Further, by now partitioning each pile based on the second letter of each name, we generate smaller and smaller piles. The names will be sorted as soon as each bucket contains only a single name. The resulting algorithm is commonly called *bucketsort* or *distribution sort*.

Bucketing is a very effective idea whenever we are confident that the distribution of data will be roughly uniform. It is the idea that underlies hash tables, *kd*-trees, and a variety of other practical data structures. The downside of such techniques is that the performance can be terrible when the data distribution is not what we expected. Although data structures such as balanced binary trees offer guaranteed worst-case behavior for any input distribution, no such promise exists for heuristic data structures on unexpected input distributions.

Nonuniform distributions do occur in real life. Consider Americans with the uncommon last name of Shifflett. When last I looked, the Manhattan telephone directory (with over one million names) contained exactly five Shiffletts. So how many Shiffletts should there be in a small city of 50,000 people? Figure 4.8 shows



Shifflett Debbie K Ruckersville	985-7957	Shifflett James 2219 Williamsburg Rd
Shifflett Debra S SR 617 Quinque	985-8813	Shifflett James B 801 Stonehenge Av
Shifflett Delma SR609	985-3688	Shifflett James C Stanardsville
Shifflett Delmas Crozet	823-5901	Shifflett James E Earlysville
Shifflett Dempsey & Marilynn		Shifflett James E Jr 552 Cleveland Av
100 Greenbrier Ter	973-7195	Shifflett James F & Lois LongMeadow
Shifflett Denise Rt 627 Dyke	985-8097	Shifflett James F & Vernell Rt671
Shifflett Dennis Stanardsville	985-4560	Shifflett James J 1430 Rugby Av
Shifflett Dennis H Stanardsville	985-2924	Shifflett James K St George Av
Shifflett Dewey E Rt667	985-6576	Shifflett James L SR33 Stanardsville
Shifflett Dewey O Dyke	985-7269	Shifflett James O Earlysville
Shifflett Diana 508 Bainbridge Av	979-7035	Shifflett James O Stanardsville
Shifflett Doby & Patricia Rt6	286-4227	Shifflett James R Old Lynchburg Rd
Shifflett Don&Ola Rt 621	974-7463	Shifflett James R Rt753 Esmont

Figure 4.8: A small subset of Charlottesville Shiffletts

a small portion of the *two and a half pages* of Shiffletts in the Charlottesville, Virginia telephone book. The Shifflett clan is a fixture of the region, but it would play havoc with any distribution sort program, as refining buckets from *S* to *Sh* to *Shi* to *Shif* to ... to *Shifflett* results in no significant partitioning.

Take-Home Lesson: Sorting can be used to illustrate most algorithm design paradigms. Data structure techniques, divide-and-conquer, randomization, and incremental construction all lead to efficient sorting algorithms.

4.7.1 Lower Bounds for Sorting

One last issue on the complexity of sorting. We have seen several sorting algorithms that run in worst-case $O(n \log n)$ time, but none of which is linear. To sort n items certainly requires looking at all of them, so any sorting algorithm must be $\Omega(n)$ in the worst case. Can we close this remaining $\Theta(\log n)$ gap?

The answer is no. An $\Omega(n \log n)$ lower bound can be shown by observing that any sorting algorithm must behave differently during execution on each of the distinct $n!$ permutations of n keys. The outcome of each pairwise comparison governs the run-time behavior of any comparison-based sorting algorithm. We can think of the set of all possible executions of such an algorithm as a tree with $n!$ leaves. The minimum height tree corresponds to the fastest possible algorithm, and it happens that $\lg(n!) = \Theta(n \log n)$.

This lower bound is important for several reasons. First, the idea can be extended to give lower bounds for many applications of sorting, including element uniqueness, finding the mode, and constructing convex hulls. Sorting has one of the few nontrivial lower bounds among algorithmic problems. We will present an alternate approach to arguing that fast algorithms are unlikely to exist in Chapter 9.

4.8 War Story: Skiena for the Defense

I lead a quiet, reasonably honest life. One reward for this is that I don't often find myself on the business end of surprise calls from lawyers. Thus I was astonished to get a call from a lawyer who not only wanted to talk with me, but wanted to talk to me about sorting algorithms.

It turned out that her firm was working on a case involving high-performance programs for sorting, and needed an expert witness who could explain technical issues to the jury. From the first edition of this book, they could see I knew something about algorithms, but before taking me on they demanded to see my teaching evaluations to prove that I could explain things to people.[2] It proved to be a fascinating opportunity to learn about how *really* fast sorting programs work. I figured I could finally answer the question of which in-place sorting algorithm was fastest in practice. Was it heapsort or quicksort? What subtle, secret algorithmics made the difference to minimize the number of comparisons in practice?

The answer was quite humbling. *Nobody cared about in-place sorting.* The name of the game was sorting *huge* files, much bigger than could fit in main memory. All the important action was in getting the the data on and off a disk. Cute algorithms for doing internal (in-memory) sorting were not particularly important because the real problem lies in sorting gigabytes at a time.

Recall that disks have relatively long seek times, reflecting how long it takes the desired part of the disk to rotate under the read/write head. Once the head is in the right place, the data moves relatively quickly, and it costs about the same to read a large data block as it does to read a single byte. Thus, the goal is minimizing the number of blocks read/written, and coordinating these operations so the sorting algorithm is never waiting to get the data it needs.

The disk-intensive nature of sorting is best revealed by the annual *Minutesort* competition. The goal is to sort as much data in one minute as possible. The current champion is Jim Wyllie of IBM Research, who managed to sort 116 gigabytes of data in 58.7 seconds on his little old 40-node 80-Itanium cluster with a SAN array of 2,520 disks. Slightly more down-to-earth is the *Pennysort* division, where the goal is the maximized sorting performance per penny of hardware. The current champ here (*BSIS* from China) sorted 32 gigabytes in 1,679 seconds on a $760 PC containing four SATA drives. You can check out the current records at *http://research.microsoft.com/barc/SortBenchmark/*.

That said, which algorithm is best for external sorting? It basically turns out to be a multiway mergesort, employing a lot of engineering and special tricks. You build a heap with members of the top block from each of k sorted lists. By repeatedly plucking the top element off this heap, you build a sorted list merging these k lists. Because this heap is sitting in main memory, these operations are fast. When you have a large enough sorted run, you write it to disk and free up

[2]One of my more cynical faculty colleagues said this was the first time anyone, anywhere, had ever looked at university teaching evaluations.

memory for more data. Once you start to run out of elements from the top block of one of the k sorted lists you are merging, load the next block.

It proves very hard to benchmark sorting programs/algorithms at this level and decide which is *really* fastest. Is it fair to compare a commercial program designed to handle general files with a stripped-down code optimized for integers? The *Minutesort* competition employs randomly-generated 100-byte records. This is a different world than sorting names or integers. For example, one widely employed trick is to strip off a relatively short prefix of the key and initially sort just on that, to avoid lugging around all those extra bytes.

What lessons can be learned from this? The most important, by far, is to do everything you can to avoid being involved in a lawsuit as either a plaintiff or defendant.[3] Courts are not instruments for resolving disputes quickly. Legal battles have a lot in common with military battles: they escalate very quickly, become very expensive in time, money, and soul, and usually end only when both sides are exhausted and compromise. Wise are the parties who can work out their problems without going to court. Properly absorbing this lesson now could save you thousands of times the cost of this book.

On technical matters, it is important to worry about external memory performance whenever you combine very large datasets with low-complexity algorithms (say linear or $n \log n$). Constant factors of even 5 or 10 can make a big difference then between what is feasible and what is hopeless. Of course, quadratic-time algorithms are doomed to fail on large datasets regardless of data access times.

4.9 Binary Search and Related Algorithms

Binary search is a fast algorithm for searching in a sorted array of keys S. To search for key q, we compare q to the middle key $S[n/2]$. If q appears before $S[n/2]$, it must reside in the top half of S; if not, it must reside in the bottom half of S. By repeating this process recursively on the correct half, we locate the key in a total of $\lceil \lg n \rceil$ comparisons—a big win over the $n/2$ comparisons expect using sequential search:

```
int binary_search(item_type s[], item_type key, int low, int high)
{
        int middle;                     /* index of middle element */

        if (low > high) return (-1);    /* key not found */

        middle = (low+high)/2;
```

[3]It is actually quite interesting serving as an expert witness.

```
        if (s[middle] == key) return(middle);

        if (s[middle] > key)
                return( binary_search(s,key,low,middle-1) );
        else
                return(binary_search(s,key,middle+1,high) );
}
```

This much you probably know. What is important is to have a sense of just how fast binary search is. *Twenty questions* is a popular children's game where one player selects a word and the other repeatedly asks true/false questions in an attempt to guess it. If the word remains unidentified after 20 questions, the first party wins; otherwise, the second player takes the honors. In fact, the second player always has a winning strategy, based on binary search. Given a printed dictionary, the player opens it in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain 50,000 to 200,000 words, we can be certain that the process will terminate within twenty questions.

4.9.1 Counting Occurrences

Several interesting algorithms follow from simple variants of binary search. Suppose that we want to count the number of times a given key k (say "Skiena") occurs in a given sorted array. Because sorting groups all the copies of k into a contiguous block, the problem reduces to finding the right block and then measures its size.

The binary search routine presented above enables us to find the index of an element of the correct block (x) in $O(\lg n)$ time. The natural way to identify the boundaries of the block is to sequentially test elements to the left of x until we find the first one that differs from the search key, and then repeat this search to the right of x. The difference between the indices of the left and right boundaries (plus one) gives the count of the number of occurrences of k.

This algorithm runs in $O(\lg n + s)$, where s is the number of occurrences of the key. This can be as bad as linear if the entire array consists of identical keys. A faster algorithm results by modifying binary search to search for the *boundary* of the block containing k, instead of k itself. Suppose we delete the equality test

```
    if (s[middle] == key) return(middle);
```

from the implementation above and return the index `low` instead of -1 on each unsuccessful search. *All* searches will now be unsuccessful, since there is no equality test. The search will proceed to the right half whenever the key is compared to an identical array element, eventually terminating at the right boundary. Repeating the search after reversing the direction of the binary comparison will lead us to the left boundary. Each search takes $O(\lg n)$ time, so we can count the occurrences in logarithmic time regardless of the size of the block.

4.9.2 One-Sided Binary Search

Now suppose we have an array A consisting of a run of 0's, followed by an unbounded run of 1's, and would like to identify the exact point of transition between them. Binary search on the array would provide the transition point in $\lceil \lg n \rceil$ tests, if we had a bound n on the number of elements in the array. In the absence of such a bound, we can test repeatedly at larger intervals ($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, $\ldots$) until we find a first nonzero value. Now we have a window containing the target and can proceed with binary search. This *one-sided binary search* finds the transition point p using at most $2\lceil \lg p \rceil$ comparisons, regardless of how large the array actually is. One-sided binary search is most useful whenever we are looking for a key that lies close to our current position.

4.9.3 Square and Other Roots

The square root of n is the number r such that $r^2 = n$. Square root computations are performed inside every pocket calculator, but it is instructive to develop an efficient algorithm to compute them.

First, observe that the square root of $n \geq 1$ must be at least 1 and at most n. Let $l = 1$ and $r = n$. Consider the midpoint of this interval, $m = (l + r)/2$. How does m^2 compare to n? If $n \geq m^2$, then the square root must be greater than m, so the algorithm repeats with $l = m$. If $n < m^2$, then the square root must be less than m, so the algorithm repeats with $r = m$. Either way, we have halved the interval using only one comparison. Therefore, after $\lg n$ rounds we will have identified the square root to within ± 1.

This bisection method, as it is called in numerical analysis, can also be applied to the more general problem of finding the roots of an equation. We say that x is a *root* of the function f if $f(x) = 0$. Suppose that we start with values l and r such that $f(l) > 0$ and $f(r) < 0$. If f is a continuous function, there must exist a root between l and r. Depending upon the sign of $f(m)$, where $m = (l + r)/2$, we can cut this window containing the root in half with each test and stop soon as our estimate becomes sufficiently accurate.

Root-finding algorithms that converge faster than binary search are known for both of these problems. Instead of always testing the midpoint of the interval, these algorithms interpolate to find a test point closer to the actual root. Still, binary search is simple, robust, and works as well as possible without additional information on the nature of the function to be computed.

Take-Home Lesson: Binary search and its variants are the quintessential divide-and-conquer algorithms.

4.10 Divide-and-Conquer

One of the most powerful techniques for solving problems is to break them down into smaller, more easily solved pieces. Smaller problems are less overwhelming, and they permit us to focus on details that are lost when we are studying the entire problem. A recursive algorithm starts to become apparent when we can break the problem into smaller instances of the same type of problem. Effective parallel processing requires decomposing jobs into at least as many tasks as processors, and is becoming more important with the advent of cluster computing and multicore processors.

Two important algorithm design paradigms are based on breaking problems down into smaller problems. In Chapter 8, we will see dynamic programming, which typically removes one element from the problem, solves the smaller problem, and then uses the solution to this smaller problem to add back the element in the proper way. *Divide-and-conquer* instead splits the problem in (say) halves, solves each half, then stitches the pieces back together to form a full solution.

To use divide-and-conquer as an algorithm design technique, we must divide the problem into two smaller subproblems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two subproblems, we get an efficient algorithm. Mergesort, discussed in Section 4.5 (page 120), is the classic example of a divide-and-conquer algorithm. It takes only linear time to merge two sorted lists of $n/2$ elements, each of which was obtained in $O(n \lg n)$ time.

Divide-and-conquer is a design technique with many important algorithms to its credit, including mergesort, the fast Fourier transform, and Strassen's matrix multiplication algorithm. Beyond binary search and its many variants, however, I find it to be a difficult design technique to apply in practice. Our ability to analyze divide-and-conquer algorithms rests on our strength to solve the asymptotics of recurrence relations governing the cost of such recursive algorithms.

4.10.1 Recurrence Relations

Many divide-and-conquer algorithms have time complexities that are naturally modeled by recurrence relations. Evaluating such recurrences is important to understanding when divide-and-conquer algorithms perform well, and provide an important tool for analysis in general. The reader who balks at the very idea of analysis is free to skip this section, but there are important insights into design that come from an understanding of the behavior of recurrence relations.

What is a recurrence relation? It is an equation that is defined in terms of itself. The Fibonacci numbers are described by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ and discussed in Section 8.1.1. Many other natural functions are easily expressed as recurrences. Any polynomial can be represented by a recurrence, such as the linear function:

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$$

Any exponential can be represented by a recurrence:

$$a_n = 2a_{n-1}, a_1 = 1 \longrightarrow a_n = 2^{n-1}$$

Finally, lots of weird functions that cannot be described easily with conventional notation can be represented by a recurrence:

$$a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n!$$

This means that recurrence relations are a very versatile way to represent functions.

The self-reference property of recurrence relations is shared with recursive programs or algorithms, as the shared roots of both terms reflect. Essentially, recurrence relations provide a way to analyze recursive structures, such as algorithms.

4.10.2 Divide-and-Conquer Recurrences

Divide-and-conquer algorithms tend to break a given problem into some number of smaller pieces (say a), each of which is of size n/b. Further, they spend $f(n)$ time to combine these subproblem solutions into a complete result. Let $T(n)$ denote the worst-case time the algorithm takes to solve a problem of size n. Then $T(n)$ is given by the following recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

Consider the following examples:

- *Sorting* – The running time behavior of mergesort is governed by the recurrence $T(n) = 2T(n/2) + O(n)$, since the algorithm divides the data into equal-sized halves and then spends linear time merging the halves after they are sorted. In fact, this recurrence evaluates to $T(n) = O(n \lg n)$, just as we got by our previous analysis.

- *Binary Search* – The running time behavior of binary search is governed by the recurrence $T(n) = T(n/2) + O(1)$, since at each step we spend constant time to reduce the problem to an instance half its size. In fact, this recurrence evaluates to $T(n) = O(\lg n)$, just as we got by our previous analysis.

- *Fast Heap Construction* – The `bubble_down` method of heap construction (described in Section 4.3.4) built an n-element heap by constructing two $n/2$ element heaps and then merging them with the root in logarithmic time. This argument reduces to the recurrence relation $T(n) = 2T(n/2) + O(\lg n)$. In fact, this recurrence evaluates to $T(n) = O(n)$, just as we got by our previous analysis.

- *Matrix Multiplication* – As discussed in Section 2.5.4, the standard matrix multiplication algorithm for two $n \times n$ matrices takes $O(n^3)$, because we compute the dot product of n terms for each of the n^2 elements in the product matrix.