on a priority queue that never has more than C elements. A simple implementation of the priority queue, using a list, would give an $O(C^2)$ algorithm. The choice of priority queue implementation depends on how large C is. In the typical case of an ASCII character set, C is small enough that the quadratic running time is acceptable. In such an application, virtually all the running time will be spent on the disk I/O required to read the input file and write out the compressed version.

There are two details that must be considered. First, the encoding information must be transmitted at the start of the compressed file, since otherwise it will be impossible to decode. There are several ways of doing this; see Exercise 10.4. For small files, the cost of transmitting this table will override any possible savings in compression, and the result will probably be file expansion. Of course, this can be detected and the original left intact. For large files, the size of the table is not significant.

The second problem is that, as described, this is a two-pass algorithm. The first pass collects the frequency data, and the second pass does the encoding. This is obviously not a desirable property for a program dealing with large files. Some alternatives are described in the references.

10.1.3 Approximate Bin Packing

In this section, we will consider some algorithms to solve the **bin-packing problem**. These algorithms will run quickly but will not necessarily produce optimal solutions. We will prove, however, that the solutions that are produced are not too far from optimal.

We are given N items of sizes $s_1, s_2, \ldots, s_N$. All sizes satisfy $0 < s_i \leq 1$. The problem is to pack these items in the fewest number of bins, given that each bin has unit capacity. As an example, Figure 10.20 shows an optimal packing for an item list with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

There are two versions of the bin packing problem. The first version is **online bin packing**. In this version, each item must be placed in a bin before the next item can be processed. The second version is the **offline bin packing problem**. In an offline algorithm, we do not need to do anything until all the input has been read. The distinction between online and offline algorithms was discussed in Section 8.2.

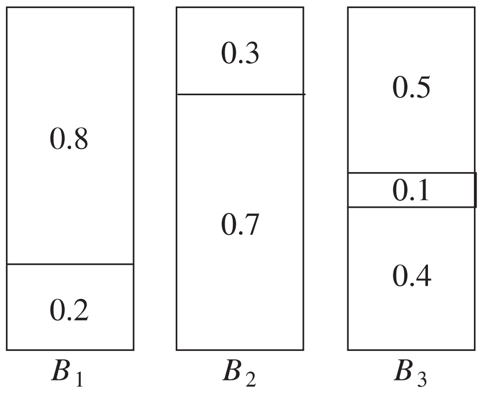

Figure 10.20 Optimal packing for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Online Algorithms

The first issue to consider is whether or not an online algorithm can actually always give an optimal answer, even if it is allowed unlimited computation. Remember that even though unlimited computation is allowed, an online algorithm must place an item before processing the next item and cannot change its decision.

To show that an online algorithm cannot always give an optimal solution, we will give it particularly difficult data to work on. Consider an input sequence, I_1, of M small items of weight $\frac{1}{2} - \epsilon$ followed by M large items of weight $\frac{1}{2} + \epsilon$, $0 < \epsilon < 0.01$. It is clear that these items can be packed in M bins if we place one small item and one large item in each bin. Suppose there were an optimal online algorithm, A, that could perform this packing. Consider the operation of algorithm A on the sequence I_2, consisting of only M small items of weight $\frac{1}{2} - \epsilon$. I_2 can be packed in $\lceil M/2 \rceil$ bins. However, A will place each item in a separate bin, since A must yield the same results on I_2 as it does for the first half of I_1, and the first half of I_1 is exactly the same input as I_2. This means that A will use twice as many bins as is optimal for I_2. What we have proved is that there is no optimal algorithm for online bin packing.

What the argument above shows is that an online algorithm never knows when the input might end, so any performance guarantees it provides must hold at every instant throughout the algorithm. If we follow the foregoing strategy, we can prove the following.

Theorem 10.1

There are inputs that force any online bin packing algorithm to use at least $\frac{4}{3}$ the optimal number of bins.

Proof

Suppose otherwise, and suppose for simplicity, that M is even. Consider any online algorithm A running on the input sequence I_1, above. Recall that this sequence consists of M small items followed by M large items. Let us consider what the algorithm A has done after processing the Mth item. Suppose A has already used b bins. At this point in the algorithm, the optimal number of bins is $M/2$, because we can place two elements in each bin. Thus we know that $2b/M < \frac{4}{3}$, by our assumption of a better-than-$\frac{4}{3}$ performance guarantee.

Now consider the performance of algorithm A after all items have been packed. All bins created after the bth bin must contain exactly one item, since all small items are placed in the first b bins, and two large items will not fit in a bin. Since the first b bins can have at most two items each, and the remaining bins have one item each, we see that packing $2M$ items will require at least $2M - b$ bins. Since the $2M$ items can be optimally packed using M bins, our performance guarantee assures us that $(2M - b)/M < \frac{4}{3}$.

The first inequality implies that $b/M < \frac{2}{3}$, and the second inequality implies that $b/M > \frac{2}{3}$, which is a contradiction. Thus, no online algorithm can guarantee that it will produce a packing with less than $\frac{4}{3}$ the optimal number of bins.

There are three simple algorithms that guarantee that the number of bins used is no more than twice optimal. There are also quite a few more complicated algorithms with better guarantees.

Next Fit

Probably the simplest algorithm is **next fit.** When processing any item, we check to see whether it fits in the same bin as the last item. If it does, it is placed there; otherwise, a new bin is created. This algorithm is incredibly simple to implement and runs in linear time. Figure 10.21 shows the packing produced for the same input as Figure 10.20.

Not only is next fit simple to program, its worst-case behavior is also easy to analyze.

Theorem 10.2

Let M be the optimal number of bins required to pack a list I of items. Then next fit never uses more than $2M$ bins. There exist sequences such that next fit uses $2M - 2$ bins.

Proof

Consider any adjacent bins B_j and B_{j+1}. The sum of the sizes of all items in B_j and B_{j+1} must be larger than 1, since otherwise all of these items would have been placed in B_j. If we apply this result to all pairs of adjacent bins, we see that at most half of the space is wasted. Thus next fit uses at most twice the optimal number of bins.

To see that this ratio, 2, is tight, suppose that the N items have size $s_i = 0.5$ if i is odd and $s_i = 2/N$ if i is even. Assume N is divisible by 4. The optimal packing, shown in Figure 10.22, consists of $N/4$ bins, each containing 2 elements of size 0.5, and one bin containing the $N/2$ elements of size $2/N$, for a total of $(N/4) + 1$. Figure 10.23 shows that next fit uses $N/2$ bins. Thus, next fit can be forced to use almost twice as many bins as optimal.

First Fit

Although next fit has a reasonable performance guarantee, it performs poorly in practice, because it creates new bins when it does not need to. In the sample run, it could have placed the item of size 0.3 in either B_1 or B_2, rather than create a new bin.

The **first fit** strategy is to scan the bins in order and place the new item in the first bin that is large enough to hold it. Thus, a new bin is created only when the results of previous placements have left no other alternative. Figure 10.24 shows the packing that results from first fit on our standard input.

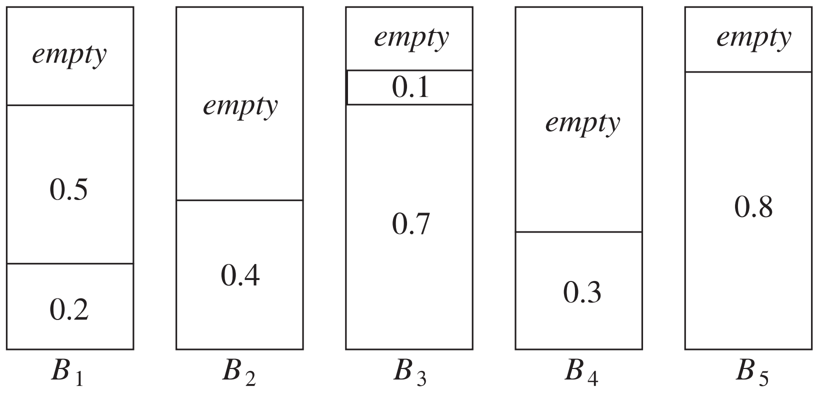

Figure 10.21 Next fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

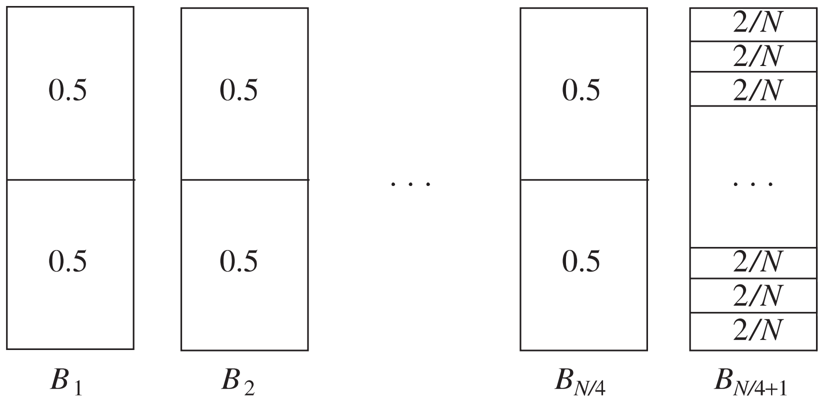

Figure 10.22 Optimal packing for 0.5, 2/*N*, 0.5, 2/*N*, 0.5, 2/*N*, . . .

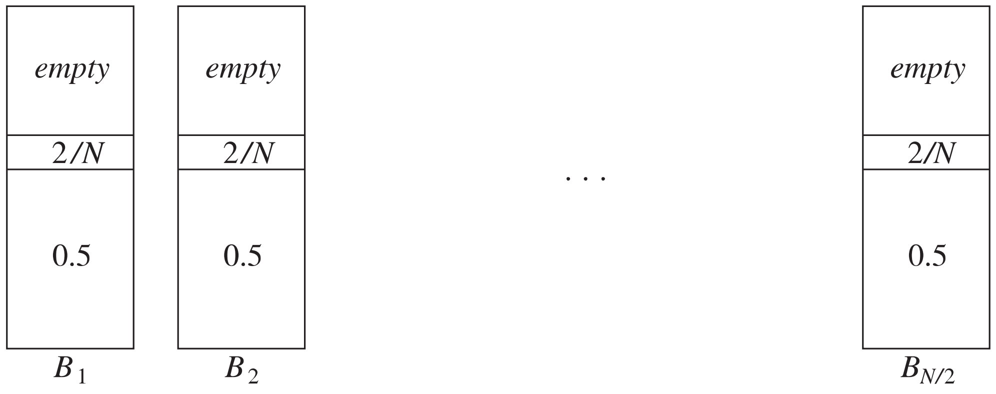

Figure 10.23 Next fit packing for 0.5, 2/*N*, 0.5, 2/*N*, 0.5, 2/*N*, . . .

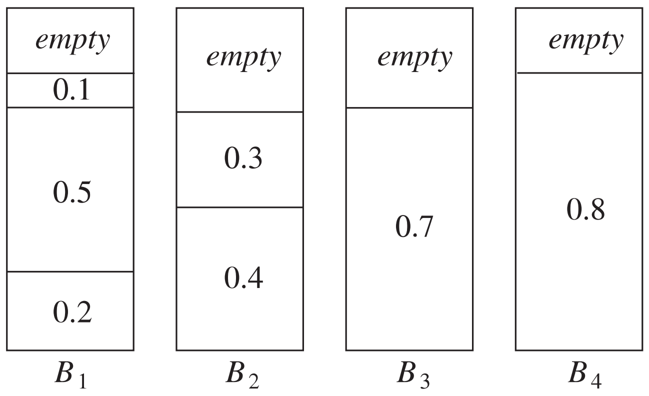

Figure 10.24 First fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

A simple method of implementing first fit would process each item by scanning down the list of bins sequentially. This would take $O(N^2)$. It is possible to implement first fit to run in $O(N \log N)$; we leave this as an exercise.

A moment's thought will convince you that at any point, at most one bin can be more than half empty, since if a second bin were also half empty, its contents would fit into the first bin. Thus, we can immediately conclude that first fit guarantees a solution with at most twice the optimal number of bins.

On the other hand, the bad case that we used in the proof of next fit's performance bound does not apply for first fit. Thus, one might wonder if a better bound can be proven. The answer is yes, but the proof is complicated.

Theorem 10.3

Let M be the optimal number of bins required to pack a list I of items. Then first fit never uses more than $\frac{17}{10}M + \frac{7}{10}$ bins. There exist sequences such that first fit uses $\frac{17}{10}(M-1)$ bins.

Proof

See the references at the end of the chapter.

An example where first fit does almost as poorly as the previous theorem would indicate is shown in Figure 10.25. The input consists of $6M$ items of size $\frac{1}{7}+\epsilon$, followed by $6M$ items of size $\frac{1}{3}+\epsilon$, followed by $6M$ items of size $\frac{1}{2}+\epsilon$. One simple packing places one item of each size in a bin and requires $6M$ bins. First fit requires $10M$ bins.

When first fit is run on a large number of items with sizes uniformly distributed between 0 and 1, empirical results show that first fit uses roughly 2 percent more bins than optimal. In many cases, this is quite acceptable.

Best Fit

The third online strategy we will examine is **best fit.** Instead of placing a new item in the first spot that is found, it is placed in the tightest spot among all bins. A typical packing is shown in Figure 10.26.

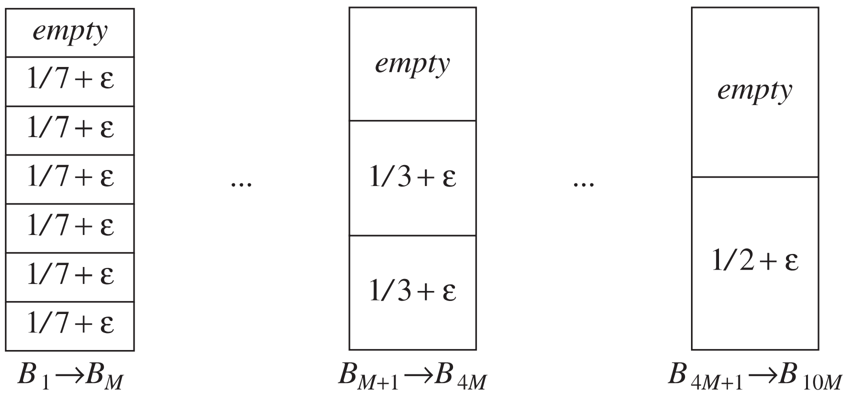

Figure 10.25 A case where first fit uses $10M$ bins instead of $6M$

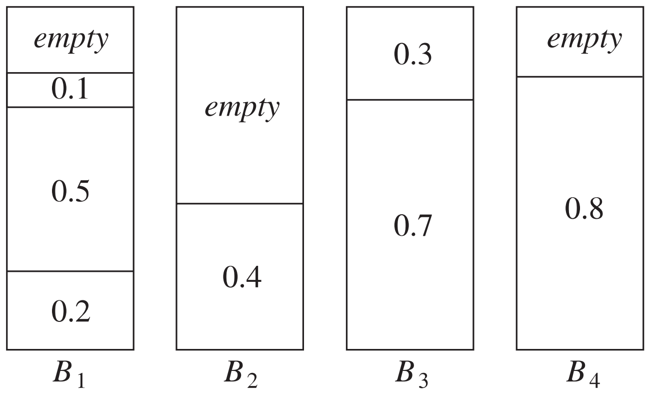

Figure 10.26 Best fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Notice that the item of size 0.3 is placed in B_3, where it fits perfectly, instead of B_2. One might expect that since we are now making a more educated choice of bins, the performance guarantee would improve. This is not the case, because the generic bad cases are the same. Best fit is never more than roughly 1.7 times as bad as optimal, and there are inputs for which it (nearly) achieves this bound. Nevertheless, best fit is also simple to code, especially if an $O(N \log N)$ algorithm is required, and it does perform better for random inputs.

Offline Algorithms

If we are allowed to view the entire item list before producing an answer, then we should expect to do better. Indeed, since we can eventually find the optimal packing by exhaustive search, we already have a theoretical improvement over the online case.

The major problem with all the online algorithms is that it is hard to pack the large items, especially when they occur late in the input. The natural way around this is to sort the items, placing the largest items first. We can then apply first fit or best fit, yielding the algorithms **first fit decreasing** and **best fit decreasing,** respectively. Figure 10.27

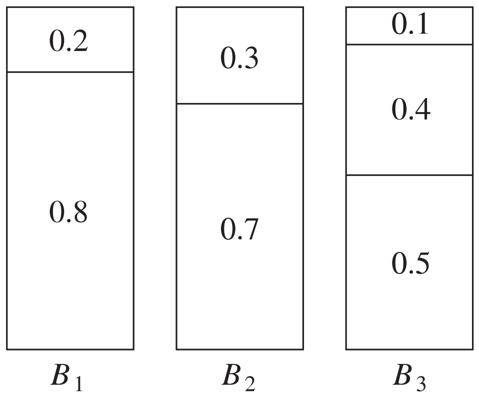

Figure 10.27 First fit for 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1

shows that in our case this yields an optimal solution (although, of course, this is not true in general).

In this section, we will deal with first fit decreasing. The results for best fit decreasing are almost identical. Since it is possible that the item sizes are not distinct, some authors prefer to call the algorithm *first fit nonincreasing*. We will stay with the original name. We will also assume, without loss of generality, that input sizes are already sorted.

The first remark we can make is that the bad case, which showed first fit using $10M$ bins instead of $6M$ bins, does not apply when the items are sorted. We will show that if an optimal packing uses M bins, then first fit decreasing never uses more than $(4M + 1)/3$ bins.

The result depends on two observations. First, all the items with weight larger than $\frac{1}{3}$ will be placed in the first M bins. This implies that all the items in the extra bins have weight at most $\frac{1}{3}$. The second observation is that the number of items in the extra bins can be at most $M - 1$. Combining these two results, we find that at most $\lceil (M - 1)/3 \rceil$ extra bins can be required. We now prove these two observations.

Lemma 10.1

Let the N items have (sorted in decreasing order) input sizes $s_1, s_2, \ldots, s_N$, respectively, and suppose that the optimal packing is M bins. Then all items that *first fit decreasing* places in extra bins have size at most $\frac{1}{3}$.

Proof

Suppose the ith item is the first placed in bin $M + 1$. We need to show that $s_i \leq \frac{1}{3}$. We will prove this by contradiction. Assume $s_i > \frac{1}{3}$.

It follows that $s_1, s_2, \ldots, s_{i-1} > \frac{1}{3}$, since the sizes are arranged in sorted order. From this it follows that all bins $B_1, B_2, \ldots, B_M$ have at most two items each.

Consider the state of the system after the $(i-1)$st item is placed in a bin, but before the ith item is placed. We now want to show that (under the assumption that $s_i > \frac{1}{3}$) the first M bins are arranged as follows: First, there are some bins with exactly one element, and then the remaining bins have two elements.

Suppose there were two bins, B_x and B_y, such that $1 \leq x < y \leq M$, B_x has two items, and B_y has one item. Let x_1 and x_2 be the two items in B_x, and let y_1 be the item in B_y. $x_1 \geq y_1$, since x_1 was placed in the earlier bin. $x_2 \geq s_i$, by similar reasoning. Thus, $x_1 + x_2 \geq y_1 + s_i$. This implies that s_i could be placed in B_y. By our assumption this is not possible. Thus, if $s_i > \frac{1}{3}$, then, at the time that we try to process s_i, the first M bins are arranged such that the first j have one element and the next $M - j$ have two elements.

To prove the lemma we will show that there is no way to place all the items in M bins, which contradicts the premise of the lemma.

Clearly, no two items $s_1, s_2, \ldots, s_j$ can be placed in one bin, by any algorithm, since if they could, first fit would have done so too. We also know that first fit has not placed any of the items of size $s_{j+1}, s_{j+2}, \ldots, s_i$ into the first j bins, so none of them fit. Thus, in any packing, specifically the optimal packing, there must be j bins that do not contain these items. It follows that the items of size $s_{j+1}, s_{j+2}, \ldots, s_{i-1}$ must be contained in some set of $M - j$ bins, and from previous considerations, the total number of such items is $2(M - j)$.[1]

The proof is completed by noting that if $s_i > \frac{1}{3}$, there is no way for s_i to be placed in one of these M bins. Clearly, it cannot go in one of the j bins, since if it could, then first fit would have done so too. To place it in one of the remaining $M - j$ bins requires distributing $2(M - j) + 1$ items into the $M - j$ bins. Thus, some bin would have to have three items, each of which is larger than $\frac{1}{3}$, a clear impossibility.

This contradicts the fact that all the sizes can be placed in M bins, so the original assumption must be incorrect. Thus, $s_i \le \frac{1}{3}$.

Lemma 10.2
The number of objects placed in extra bins is at most $M - 1$.

Proof
Assume that there are at least M objects placed in extra bins. We know that $\sum_{i=1}^{N} s_i \le M$, since all the objects fit in M bins. Suppose that B_j is filled with W_j total weight for $1 \le j \le M$. Suppose the first M extra objects have sizes $x_1, x_2, \ldots, x_M$. Then, since the items in the first M bins plus the first M extra items are a subset of all the items, it follows that

$$\sum_{i=1}^{N} s_i \ge \sum_{j=1}^{M} W_j + \sum_{j=1}^{M} x_j \ge \sum_{j=1}^{M} (W_j + x_j)$$

Now $W_j + x_j > 1$, since otherwise the item corresponding to x_j would have been placed in B_j. Thus

$$\sum_{i=1}^{N} s_i > \sum_{j=1}^{M} 1 > M$$

But this is impossible if the N items can be packed in M bins. Thus, there can be at most $M - 1$ extra items.

Theorem 10.4
Let M be the optimal number of bins required to pack a list I of items. Then first fit decreasing never uses more than $(4M + 1)/3$ bins.

Proof
There are at most $M - 1$ extra items, of size at most $\frac{1}{3}$. Thus, there can be at most $\lceil (M - 1)/3 \rceil$ extra bins. The total number of bins used by first fit decreasing is thus at most $\lceil (4M - 1)/3 \rceil \le (4M + 1)/3$.

It is possible to prove a much tighter bound for both first fit decreasing and next fit decreasing.

[1] Recall that first fit packed these elements into $M - j$ bins and placed two items in each bin. Thus, there are $2(M - j)$ items.

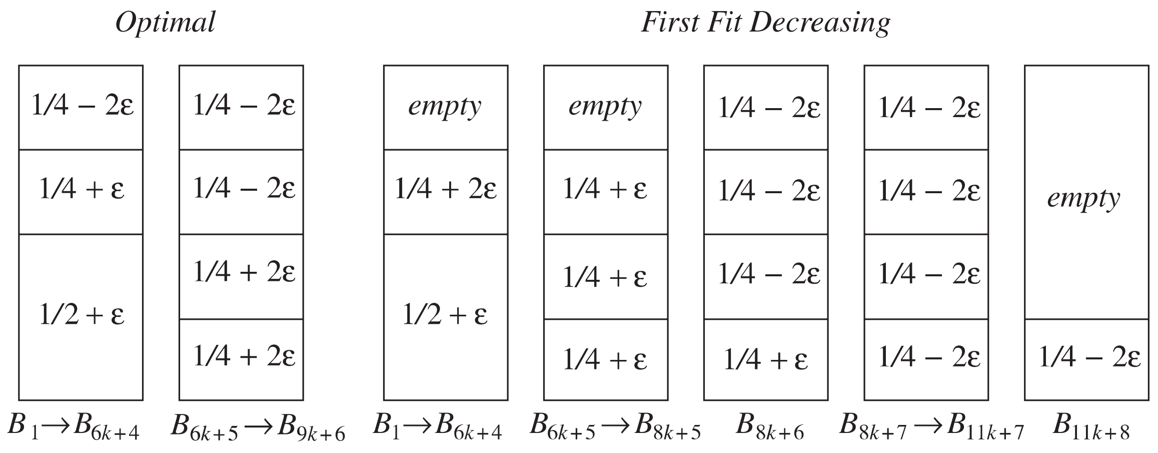

Figure 10.28 Example where first fit decreasing uses $11k + 8$ bins, but only $9k + 6$ bins are required

Theorem 10.5
Let M be the optimal number of bins required to pack a list I of items. Then first fit decreasing never uses more than $\frac{11}{9}M + \frac{6}{9}$ bins. There exist sequences such that first fit decreasing uses $\frac{11}{9}M + \frac{6}{9}$ bins.

Proof
The upper bound requires a very complicated analysis. The lower bound is exhibited by a sequence consisting of $6k+4$ elements of size $\frac{1}{2}+\epsilon$, followed by $6k+4$ elements of size $\frac{1}{4}+2\epsilon$, followed by $6k+4$ elements of size $\frac{1}{4}+\epsilon$, followed by $12k+8$ elements of size $\frac{1}{4}-2\epsilon$. Figure 10.28 shows that the optimal packing requires $9k+6$ bins, but first fit decreasing uses $11k+8$ bins. Set $M = 9k+6$, and the result follows.

In practice, first fit decreasing performs extremely well. If sizes are chosen uniformly over the unit interval, then the expected number of extra bins is $\Theta(\sqrt{M})$. Bin packing is a fine example of how simple greedy heuristics can give good results.

10.2 Divide and Conquer

Another common technique used to design algorithms is **divide and conquer.** Divide-and-conquer algorithms consist of two parts:

Divide: Smaller problems are solved recursively (except, of course, base cases).

Conquer: The solution to the original problem is then formed from the solutions to the subproblems.

Traditionally, routines in which the text contains at least two recursive calls are called divide-and-conquer algorithms, while routines whose text contains only one recursive call