CHAPTER 8

Sorting Theory

Introduction

The problem of developing and implementing good sorting algorithms has been extensively studied. If you've taken a programming course, you have probably seen code for specific sorting algorithms. You may have programmed various sorting algorithms. Our focus will be different, emphasizing the general framework over the specific implementations. We'll also look at "sorting networks" which are a type of hardware implementation of certain sorting algorithms.

In the last section, we'll explore the "divide and conquer" technique. The major aspect of this technique is the recursive approach to problems.

Before discussing sorting methods, we'll need a general framework for thinking about the subject. Thus we'll look briefly at how sorting algorithms can be classified. All of them involve making comparisons, require some sort of storage medium for the list and must be physically implemented in some fashion. We can partially classify algorithms according to how these things are handled.

- Type of comparison:
  - Relative: An item is compared to another item. The result simply says which of the two is larger. Almost all sorts are of this type.
  - Absolute: An item is ranked using an absolute standard. The result says which of several items it equals (or lies between). These are the *bucket* sorts.
- Data access needed:
  - Random Access: These types of sorts need to be able to look at almost any item in the (partially sorted) list at almost any time.
  - Sequential Access: These types of sorts make a rather limited number of sequential passes through the data. Consequently, the data could be stored on *magnetic tape*.[1] These are usually *merge* sorts.
- Implementation method:
  - Software: Most sorts are implemented as a program on a general purpose computer.

[1] Magnetic tape is a medium that was used before large disc drives were available at reasonable prices. The data on the tape can be thought of as one long list. It was very time consuming to move to a position in the list that was far away from the current position. Hence it was desirable to read the list without skipping around. This is the origin of the term *sequential access*. One tape provided a single sequential access medium. Two tapes provided two sequential access media, and so forth.

- Hardware: Some sorts are implemented by hardware. Depending on how versatile the hardware is, it could be close to a software implementation. The least versatile (and also most common) hardware implementations are (a) the card sorters of a largely bygone era which are used to bucket sort punched cards and (b) *sorting networks*, which we'll study in Section 8.3.

8.1 Limits on Speed

Suppose someone comes to you with two sorting algorithms and asks you which is faster. How do you answer him? Unless he has actual code for a particular machine, you couldn't obtain actual times. However, since comparisons are the heart of sorting, we could ask: "How many comparisons does this algorithm make in the process of sorting?" We could then suggest that the algorithm that required less comparisons was the faster of the two. There are some factors that make this only a rough estimate:

- We did not include relocation overhead—somehow items must be repositioned to obtain the sorted list.
- We did not include miscellaneous overhead such as initialization and subroutine calls.

We will ignore such problems and just look at the number of comparisons. Even so, there are problems:

- Using the parallel processing capabilities of supercomputers or special purpose devices will throw time estimates off because more than one comparison can be done at a time. The amount of parallelism that is possible can vary from algorithm to algorithm.
- The number of comparisons needed may vary greatly, depending on the order of the items in the unsorted list.

We'll ignore these factors in the discussion, except for parallelism in sorting networks, where it is of major importance.

Besides all these problems with estimating running time, there is another problem: Running time is not the only standard that can be used to decide how good an algorithm is. Other important questions include

- How long will it take to get an error free program running?
- How much storage space will the algorithm require?

We'll ignore these issues and focus on running time.

Let $C(n)$ be the number of comparisons that an algorithm requires to sort n items. "Foul!" cries a careful reader. "You pointed out that the time a sort takes depends on the original order of the list and now you're talking about this number $C(n)$ as if that weren't the case." True. We should specify $C(n)$ more carefully. It's reasonable to consider two measures of speed:

- Worst case: $C(n) = \mathrm{WC}(n)$, the greatest number of comparisons the algorithm requires to sort n items. This is important where the results of a sort are needed quickly.
- Average: $C(n) = \mathrm{AC}(n)$, the average number of comparisons the algorithm requires to sort n items. This is important when we are doing many sorts and want to minimize overall computer usage.

The average referred to here is the average over all $n!$ possible orderings of the list. Obviously $\mathrm{WC}(n) \geq \mathrm{AC}(n)$. Our goal in this section is to motivate and prove

Theorem 8.1 Lower Bound on Comparisons AC(n), *the average number of comparisons required by any sorting algorithm that correctly sorts all possible lists of n items by comparing pairs of elements is at least* $\log_2(n!)$.

By Stirling's formula (Theorem 1.5 (p. 12)), this bound is close to $n\log_2 n$ when n is large. In view of this, a sorting algorithm with running time $\Theta(n \ln n)$ is usually considered to be a reasonably fast algorithm. Most commonly used sorting algorithms are reasonably fast. We'll look at an old friend now.

Example 8.1 Merge sorting is reasonably fast In Example 7.13 (p. 211) we saw that a simple merge sort takes at most about $n\log_2 n$ comparisons when n is a power of 2. (Actually, this result is true for all large n.) Thus, a merge sort is reasonably fast. As indicated in Example 7.13, there are some storage space problems with merge sorting. $\square$

Motivation and Proof of the Theorem

Our proof of Theorem 8.1 will be by induction. Induction requires knowing the result (in this case $\text{AC}(n) \geq \log_2(n!)$) beforehand. How would one ever come up with the result beforehand? A result like the Four Color Theorem (p. 158) might be conjectured after some experimentation, but one is unlikely to stumble upon Theorem 8.1 experimentally. We will "discover" it by relating sorting algorithms to decision trees, which lead easily to the inequality $\text{WC}(n) \geq \log_2(n!)$. One might then test $\text{AC}(n) \geq \log_2(n!)$ for small values of n, thus motivating Theorem 8.1. Someone versed in information theory might motivate the theorem as follows: "A comparison gives us one bit of information. Given k bits of information, we can distinguish among at most 2^k different things. Since we must distinguish among $n!$ different arrangements, we require that $2^k \geq n!$ and so $k \geq \log_2(n!)$." (This is motivation, not a proof—it's not even clear if it's referring to worst case or average case behavior.) Let's get on with things.

Suppose we are given a comparison based sorting algorithm. Since it is assumed to correctly sort n-lists, it must correctly sort lists in which all items are different. By simply renaming our items 1, 2, ..., n, we can suppose that we are sorting lists which are permutations of $\underline{n}$.

Our proof will make use of the decision tree associated with this sorting algorithm. We construct the tree as follows. Whenever we make a comparison in the course of the algorithm, our subsequent action depends on whether an inequality holds or fails to hold. Thus there are two possible decisions at each comparison and so each vertex in our decision tree has at most two sons.

Label each leaf of the tree with the permutations that led to that leaf and throw away any leaves that are not associated with any permutation. To do this, we start at the root with a permutation f of $\underline{n}$ and at each vertex in the tree we go left if the inequality we are checking holds and go right if it fails to hold. At the same time, we carry out whatever manipulations on the data in f that the algorithm requires. When we arrive at a leaf, the data in f will be sorted. Label the leaf with the f we started out with at the root, written in one line form. Do this for all $n!$ permutations of $\underline{n}$.

For example, consider the following algorithm for sorting a permutation of $\underline{3}$.

1. If the entry in the first position exceeds the entry in the third position, switch them.
2. If the entry in the first position exceeds the entry in the second position, switch them.
3. If the entry in the second position exceeds the entry in the third position, switch them.

Figure 8.1 shows the labeled decision tree. Two positions where you would ordinarily expect leaves have none because they are never reached. Consider the permuted sequence 231. The "if" in Step 1 is true and results in a switch to give 132. The second "if" is false and results in no switch. The third

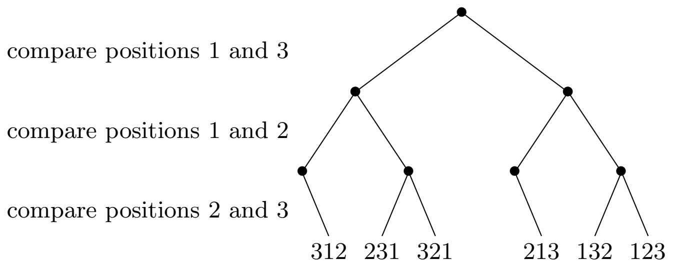

compare positions 1 and 3
compare positions 1 and 2
compare positions 2 and 3
312
231
321
213
132
123

Figure 8.1 A sorting algorithm decision tree. Leftward branches correspond to decisions to switch. Leaves are labeled with starting sequences.

is true and results in a switch to give 123. Thus, the sequence of decisions associated with sorting 231 is switch, no switch and switch, respectively.

We now show that a decision tree for a correct sorting algorithm has exactly $n!$ leaves. This will follow from the fact that each leaf is labeled with exactly one permutation. Why can't we have two permutations at the same leaf? Since each leaf in any such decision tree is associated with a particular rearrangement of the data, both permutations of $\underline{n}$ would be rearranged in the same fashion. Since they differed originally, their rearrangements would differ. Thus at least one of the rearrangements would not be $1, 2, \ldots, n$ and so would be incorrectly sorted.

These decision trees are binary RP-trees. (An RP-tree was defined in Definition 5.12 (p. 139) and it is binary if each node has at most two sons.) The set B of all binary RP-trees with $n!$ leaves includes the set S of those decision trees that come from sorting algorithms. Since S is a subset of B, a lower bound for any function on B is also a lower bound for that function on S. Hence a lower bound on worst case or average values for the set of all binary RP-trees with $n!$ leaves will also be a lower bound on $\mathrm{WC}(n)$ or $\mathrm{AC}(n)$ for any algorithm that correctly sorts n items by using pairwise comparisons.

In our decision tree approach, a comparison translates into a decision. If we only wanted to study $\mathrm{WC}(n)$, we could finish quickly as follows. By the definition of $\mathrm{WC}(n)$, the longest possible sequence of decisions contains $\mathrm{WC}(n)$ decisions. This is called the *height* of the tree. To get a tree with as many leaves as possible, we should let the tree branch as much as possible. Since the number of nodes doubles for each level of decisions in such a tree, you should be able to see that a binary RP-tree of height k has at most 2^k leaves. Since there are $n!$ leaves, we must have $2^{\mathrm{WC}(n)} \geq n!$. Taking $\log_2$ of both sides gives us $\mathrm{WC}(n) \geq \log_2(n!)$.

What about $\mathrm{AC}(n)$? It's not too difficult to compute the average number of comparisons for various binary RP-trees when they are not too large. If you were to do that for a while, you would probably begin to believe that the lower bound we just derived for $\mathrm{WC}(n)$ is also a lower bound for $\mathrm{AC}(n)$, as claimed in the theorem. This completes the motivation for believing the theorem.

How might we prove the theorem? Since the first decision (the one at the root) divides the tree into two smaller trees, it seems reasonable to try induction. Unfortunately, a tree with $(n+1)!$ leaves is a lot bigger than one with only $n!$ leaves. This can cause problems with induction. Let's sever our ties with sorting algorithms and consider decision trees with any number of leaves, not just those where the number of leaves is $n!$ for some n.

From now on, n will now indicate the total number of leaves in the tree, not the number of things being permuted.

For a decision tree T, let $\mathrm{TC}(T)$ be the sum, over all leaves ℓ of T, of the number of decisions needed to reach ℓ. The average cost is then $\mathrm{TC}(T)/n$. To prove the theorem, it suffices to prove

$$\mathrm{TC}(T) \geq n \log_2 n \text{ for all binary RP-trees } T \text{ with } n \text{ leaves.} \qquad 8.1$$

Call this $\mathcal{A}(n)$. Clearly $\mathcal{A}(1)$ is true since $\log_2 1 = 0$.

We now proceed by induction: Given $\mathcal{A}(k)$ for all $k < n$, we will prove $\mathcal{A}(n)$. Let T be a binary RP-tree with n leaves. Consider the root of T. If it has only one son, removing the root gives a binary RP-tree T' with n leaves and $\mathrm{TC}(T) = \mathrm{TC}(T') + n$. (The "$+n$" arises because each of the n leaves is one vertex further from the root in T than in T'.) If $\mathcal{A}(n)$ were false for T, it would also be false for T'. Thus we don't need to consider T at all if T has only one son. Thus it suffices to prove (8.1) when the root has degree 2. Let v_L and v_R be the two children of the root and let T_L and T_R be the trees rooted at v_L and v_R. Let k be the number of leaves in T_L. Then T_R has $n-k$ leaves. We have

$$\mathrm{TC}(T) \;=\; \Big(\mathrm{TC}(T_L)+k\Big)+\Big(\mathrm{TC}(T_R)+(n-k)\Big) \;\ge\; k\log_2 k+(n-k)\log_2(n-k)+n, \qquad 8.2$$

where the last part follows from $\mathcal{A}(k)$ and $\mathcal{A}(n-k)$ since $k<n$ and $n-k<n$. Clearly

$$k\log_2 k+(n-k)\log_2(n-k) \;\ge\; \min f(x), \qquad 8.3$$

where $f(x) \;=\; x\log_2 x+(n-x)\log_2(n-x)$ and the minimum is over all real x with $1 \le x \le n-1$.

This paragraph deals with the technicality of showing that the minimum of $f(x)$ is $n\log_2(n/2)$ and that it occurs at $x = n/2$. According to calculus we can find the minimum of $f(x)$ over an interval by looking at the values of $f(x)$ at the endpoints of the interval and when $f'(x)=0$. The present endpoints are awkward (but they could be dealt with). It would be nicer to increase the interval to $0 \le x \le n$. To do this we must assign a value to $0\log_2 0$ so that $f(x)$ is continuous at 0. Thus we define $0\log_2 0$ to be

$$\begin{aligned}\lim_{x\to0^+} x\log_2 x &= \lim_{x\to0^+}\frac{\log_2 x}{1/x}\\ &= \lim_{x\to0^+}\frac{1/x\ln 2}{-1/x^2} && \text{by l'Hôpital's Rule from calculus}\\ &= \lim_{x\to0^+}\frac{-x}{\ln 2}=0.\end{aligned}$$

Hence $f(0)=f(n)=n\log_2 n$. Since

$$f'(x) \;=\; \log_2 x+\frac{x}{x\ln 2}-\log_2(n-x)-\frac{n-x}{(n-x)\ln 2} \;=\; \log_2\left(\frac{x}{n-x}\right)$$

and the logarithm is zero only at 1, it follows that $f'(x)=0$ if and only if $\frac{x}{n-x}=1$; that is, $x=n/2$. Since

$$f(n/2)=n\log_2(n/2)<n\log_2 n=f(0)=f(n),$$

the minimum of $f(x)$ occurs at $x=n/2$.

We have now shown that the right side of (8.3) is $n\log_2(n/2)$ and so, from (8.2),

$$\mathrm{TC}(n) \;\ge\; n\log_2(n/2)+n \;=\; n\log_2 n-n\log_2 2+n \;=\; n\log_2 n.$$

This completes the induction step and hence the proof of the theorem. □

Exercises

8.1.1. In some data storage algorithms, information is stored at the leaves of a binary RP-tree and one reaches a leaf by answering a question at each vertex on the path from the root to a leaf, including at the leaf. (The question at the leaf is to allow for the possibility that the data may not be in the tree.) Suppose there are n items of data stored in such a tree. Show that the average number of questions required to recover stored data is at least $1+\log_2 n$.

8.1.2. The average case result in Theorem 8.1 depends heavily on the fact that we expect to visit each leaf equally often. This may not be the case. For example, in some situations the list to be sorted is often nearly in order already. To see the importance of such frequency of visiting the leaves, consider a decision tree in which the relative frequency of visitation is 0.1, 0.2, 0.3, and 0.4. Find the tree for which $\mathrm{AC}(T)$ is a minimum.

8.1.3. Let T be a binary decision tree with n leaves. Suppose that $\mathrm{TC}(T)$ is the least it can be for a binary RP-tree with n leaves. Since the minimum in (8.3) occurred at $x = n/2$, one might expect the two principal subtrees of T to have a nearly equal number of leaves. But this is not the case. In this exercise, we explore what can be said.

(a) Prove that T is a full binary tree; i.e., there is no vertex with just one child.

(b) For any vertex in T, let $h(v)$ be the length of the path from v to the root of T. Let l_1 and l_2 be two leaves of T. Prove that $|h(l_1) - h(l_2)| \le 1$.
Hint. Suppose $h(l_1) - h(l_2) \ge 2$. Let v be the parent of l_1 and consider what happens when l_2 and the subtree rooted at v are interchanged.

*(c) If $2^{m-1} < n \le 2^m$, prove that the height of T is m. (The height of a tree is the maximum of $h(v)$ over all vertices v of T.)

*(d) If $2^{m-1} < n \le 2^m$, prove that the maximum number of leaves that can be in a principal subtree of T is 2^{m-1}. Explain how to achieve this maximum.

8.1.4. In some sorts (e.g., a bucket sort described in the next section) a question may have more possible answers than just "yes" or "no." Suppose that each question has k possible answers. Show that the average number of questions required to sort n objects is at least $\log_k(n!)$. You may use the following fact without proof:
The minimum of $x_1 \log_k x_1 + \cdots + x_d \log_k x_d$ over all positive x_i that sum to n is obtained when all the x_i equal n/d and the value of the minimum is $n \log_k(n/d)$.

8.1.5. In some data storage and retrieval algorithms, a key and data are stored at each vertex of a binary RP-tree and is retrieved by means of a key. Suppose κ is the key whose data one wants. At each vertex v, one chooses the vertex (and stops) or one of the principal subtrees at v, depending on whether κ equals, exceeds, or is less than the key at v. Let $\mathrm{TC}^*(T)$ be the sum over all vertices v of T of the length of the path from v to the root.

(a) Show that a binary tree with no path to the root longer than n can store at most $2^{n+1} - 1$ keys.
Hint. Obtain a recursion for the number and use induction.

(b) Show that a tree T as in (a) storing the maximum number of possible keys has
$\mathrm{TC}^*(T) = (n-1)2^{n+1} + 2.$

8.2 Software Sorts

A merge sort algorithm was studied in Example 7.13 (p. 211). You should review it now.

All reasonably fast software sorts use a *divide and conquer* method for attacking the problem. As you may recall, divide and conquer means splitting the problem into a few smaller problems which are easier either because they are smaller or because they are simpler. In problems where divide and conquer is most successful, it is often the case that the smaller problems are simply instances of the same type of problem and they are handled by applying the algorithm recursively. To give you a bit better idea of what divide and conquer means, here is how the algorithms we'll discuss use it. Some of this may not mean much to you until you've finished this section, so you may want to reread this list later. This is by no means an exhaustive list of the different types of software sorts.

- *Quicksort* and *merge sorts* split the data and spend most of their time sorting the separate pieces. Thus they divide and conquer by producing two smaller sorting problems which are handled in a recursive manner.

Quicksort spends a little time dividing the data in such a way that recombining the two pieces after they are sorted is immediate. It divides the items into two collections so that all the items in the first collection should precede all the items in the second. The division is done "in place" by interchanging items that are in the wrong lists. Unless it is extremely unlucky, the two collections will have roughly the same number of elements. The two collections are then sorted separately.

Merge sorts reverse this: dividing is immediate and recombination takes a little time. In both cases, the "little time" required is proportional to the number of items being sorted because it requires a number of comparisons that is nearly equal to the number of items being sorted.

- An *insertion sort* builds up the sorted list by taking the items on the unsorted list one at a time and inserting them in a sorted list it is building. Divide and conquer can be used in the insertion process: To do a *binary insertion sort*, split the list into two nearly equal parts, decide which sublist should contain the new item, and iterate the process, using the sublist as the list.
- Suppose we are sorting a list of words (or numbers). *Bucket sort* focuses on one position in the words at a time. This is not usually a good divide and conquer approach because the task is not divided into just a few subproblems of roughly equal difficulty: On an n-long list with k characters per word, we focus in turn on each of the k positions. When n is large, k will be large, too.

  It is easy to get a time estimate for the algorithm. The amount of time it takes to process one character position for all n words is proportional to n. Thus, the time to sort is proportional to nk. How fast a bucket sort is depends on how large k is compared to n.
- *Heapsort* divides the sorting task into two simpler tasks.
  - First, the items are arranged in a structure, called a "heap," which is a rooted tree such that the smallest item in the tree is at the root and each of the sons of the root is also the root of a heap.
  - Second, the items are removed from the heap one by one, starting with the top and preserving the heap structure.

  Each of these two tasks requires about the same amount of time. Adding an item to the heap is done in a recursive manner, as is removing an item from the heap. The fact that a heap is defined recursively makes it easy to design recursive algorithms for manipulating heaps.

Binary Insertion Sort

Let $u_1, u_2 \ldots, u_n$ be the unsorted list. At the end of step t, the sorted list will be $s_1, s_2 \ldots, s_t$. At step t, an *insertion sort* determines where u_t belongs in the sorted list, opens up space and inserts it. For $t = 1$, this is trivial since the sorted list is empty. In general, we have a list $s_1, \ldots, s_{t-1}$. We must first find an index j such that $s_i \le u_t$ for $i \le j$ and $s_i \ge u_t$ for $i > j$, and then define a new sorted list by

$$\text{new } s_i = \begin{cases} \text{old } s_i & \text{if } 1 \le i < j; \\ u_t & \text{if } i = j; \\ \text{old } s_{i-1} & \text{if } j < i \le t. \end{cases}$$

How can we insure that a small number of comparisons is required for determining j? Simply searching the list from the start to find the place would, on average, take an amount of time proportional to k. A *binary insertion sort* uses divide and conquer to produce a much quicker insertion. It looks at the middle of the sorted list to decide which half should contain u_t and then iterates on that half until we are reduced to comparing u_t to a single item. This dividing of the list makes insertion in a k long list take one more comparison than insertion into a $k/2$ long list. Calculating the number of comparisons is left to Exercise 8.2.2.

Programming Note *In order to avoid moving a lot of items to insert u_t, insertion sorts are implemented by using a more complex data structure than a simple array. These data structures require more storage and somewhat more comparisons than binary insertion requires, but they require less movement of items. We will not discuss them. If you want further information, consult a good text on data structures and algorithms.*

Bucket Sort

As the name may suggest, a *bucket sort* is like throwing things into buckets. If we have a collection of buckets, we can put each item into the bucket where it belongs. If the buckets are in order and each bucket contains at most one item, then the items are sorted.

Since it is not practical to have this many buckets, a recursive method is used. This is better thought of in terms of piles instead of buckets. Suppose that you want to bucket sort the two digit numbers

$$22\ 31\ 12\ 23\ 13\ 33\ 11\ 21.$$

Here's how to do it.

1. Construct piles of numbers, one pile for each unit's digit, making sure that the order within a pile is the same as the order in the list. Here's the result.

   $$\mathsf{1}\text{: } 31\ 11\ 21 \quad \mathsf{2}\text{: } 22\ 12 \quad \mathsf{3}\text{: } 23\ 13\ 33.$$

2. Make a list from the piles, preserving the order. Here's the result.

   $$31\ 11\ 21\ 22\ 12\ 23\ 13\ 33.$$

3. Repeat the first two steps using the new list and using the ten's digit instead of the unit's digit. Here are the results.

   $$\mathsf{1}\text{: } 11\ 12\ 13 \quad \mathsf{2}\text{: } 21\ 22\ 23 \quad \mathsf{3}\text{: } 31\ 33$$

   gives

   $$11\ 12\ 13\ 21\ 22\ 23\ 31\ 33.$$

This method can be generalized to k digit numbers and k letter words. It is left as an exercise. In this case, each item is examined k times in order to place it in a bucket. Thus we place items in buckets kn times. If we think of digits as "letters," then a k digit number is simply a k letter word. Bucket sorting puts the words in lexicographic order. This sorting method can only be used on strings that are thought of as "words" made up of "letters" because we must look at items one "letter" at a time. The result is always a lexicographic ordering.

Suppose that we are sorting a large number n of k letter words where k is fairly small. In this case, kn, the number of times we have to place something in a bucket is less than the lower bound, $\log_2(n!)$, for the number of comparisons that we proved in the previous section. How can this be?

To begin with, deciding which bucket to place an item in is not a simple comparison unless there are only two buckets. If we want to convert that into a process that involves making decisions between pairs of items, we must do something like a binary insertion, comparing the item with the labels on the buckets. That will require about $\log_2 A$ comparisons, where A is the number of letters in the alphabet. Taking the number of comparisons into account, we obtain an estimate of $kn \log_2 A$ for the number of comparisons needed to bucket sort a list of n items.

It seems that we could simply keep k and A small and let n get large. This would still violate the lower bound of $\log_2(n!)$, which is about $n \log_2 n$. What has been ignored here is the fact that the lower bound was derived under the assumption that there can be $n!$ different orderings of a list. This can only happen if the list contains no duplicate words. Thus we must be able to create at least n

distinct words. Since there are A^k possible k letter words using an A letter alphabet, we must have $A^k \geq n$. Thus $k \log_2 A \geq \log_2 n$ and so $kn \log_2 A > n \log_2 n$, which agrees with our bound.

Programming Note *Hardware implementations of a bucket sort were in use for quite some time before inexpensive computers largely replaced them. They sorted "IBM cards," thin rectangular pieces of cardboard with holes punched in them by a "keypunch." The operator could set the sorting machine to drop the cards into bins according to the* j*th "letter" punched on the card. To sort cards by the word in columns* i *through* j*, the operator sorted on column* j*, restacked the cards, sorted on column* $j-1$*, restacked the cards, …, sorted on column* i *and restacked the cards. In other words, a bucket sort was used. These machines were developed about a hundred years ago for dealing with United States census data. The company that manufactured them became IBM.*

In software bucket sorts, one usually avoids comparisons entirely by maintaining an A *long table of pointers, one pointer for each bucket. The letter is then used as an index into the table to find the correct bucket.*

Merge Sorts

We've discussed a simple merge sort already in Example 7.13 (p. 211). Recall that the list is divided arbitrarily into two pieces, each piece is sorted (by using the merge sort recursively) and, finally, the two sorted pieces are merged. The naïve method for merging two lists is to repeatedly move the smaller of the top items in the two lists to the end of the merged list we are constructing, but this cannot be implemented in a sorting network. The Batcher sort uses a more complex merging process that can be implemented in a sorting network. We'll study it in the next section.

Programming Note *For simplicity, assume that* n *is a power of 2. You can imagine the original list as consisting of* n *1-long lists one after the other. These can be merged two at a time to produce* $(n/2)$ *2-long lists one after the other. In general, we have* $(n/2^k)$ 2^k*-long lists which can be merged two at a time to produce* $(n/2^{k+1})$ 2^{k+1}*-long lists one after the other. If the data is on tape (see footnote page 227), one starts with two sets of* 2^k*-long lists and produces two sets of* 2^{k+1}*-long lists by merging the top two lists in each set and placing the merged lists alternately in the output sets. Since only sequential access is required, this simple merge sort is ideally suited to data on tape if four tape drives are available. There are variations of this idea which are faster because they require less tape movement.*

Quicksort

In Quicksort, an item is selected and the list is divided into two pieces: those items that should be before the selected item and those that should be after it. This is done in place so that one sublist precedes the other. If the two sublists are then sorted recursively using Quicksort, the entire original list will be sorted. About n comparisons are needed to divide the list.

How is the division accomplished? Here's one method. Memorize a list element, say x. Start a pointer at the left end of the list and move it rightward until something larger than x is encountered. Similarly, start a pointer moving leftward from the right end until something smaller than x is encountered. Switch the two items and start the pointers moving again. When the pointers reach the same item, everything to the left of the item is at most equal to x and everything to right of it is at least equal to x.

How long Quicksort takes depends on how evenly the list is divided. In the worst case, one sublist has one item and the remaining items are in the other. If this continues through each division, the number of comparisons needed to sort the original list is about $n^2/2$. In the best case, the two sublists are as close to equal as possible. If this continues through each division, the number of comparisons needed to sort the original list is about $n \log_2 n$. The average number of sorts required is fairly close to $n \log_2 n$, but it's a bit tricky to prove it. We'll discuss it in Example 10.12 (p. 289).

Heapsort

We give a rough idea of the nature of Heapsort. A full discussion of Heapsort is beyond the scope of this text.

To explain Heapsort, we must define a *heap*. This data structure was described in a rough form above. Here is a complete, but terse, definition. A heap is a rooted binary tree with a bijection between the vertices and the items that are being sorted. The tree and bijection f satisfy the following.

- The smallest item in the image of f is associated with the root.
- The heights of the sons of the root differ by at most one.
- For each son v of the root, the subtree rooted at v and the bijection restricted to that subtree form a heap.

Thus the smallest case is a tree consisting of one vertex. This corresponds to a list with one element.

It turns out that it is fairly easy to add an item so that the heap structure is preserved and to remove the least item from the heap, in a way that preserves the heap structure. We will not discuss how a heap is implemented or how these operations are carried out. If you are interested in such details, see a text on data structures and algorithms.

Heapsort creates a heap from the unsorted list and then creates a sorted list from the heap by removing items one by one so that the heap structure is preserved. Thus the divide and conquer method in Heapsort involves the dividing of sorting into two phases: (a) creating the heap and (b) using the heap. Inserting or removing an item involves traversing a path between the root and a leaf. Since the greatest distance to a leaf in a tree with k nodes is about $\log_2 k$, the creation of the heap from an unsorted list takes about $\sum_{k=1}^{n} \log_2 k \approx n \log_2 n$ comparisons, as does the dismantling of the heap to form the sorted list. Thus Heapsort is a reasonably fast sort.

Note that a heap is an intermediate data structure which is quickly constructed from an unsorted list and which quickly leads to a sorted list. This observation is important. If we are in a situation where we need to continually add to a list and remove the smallest item, a heap is a good data structure to use. This is illustrated in the following example.

Programming Note *One way a heap can be implemented is with the classical form of a heap—a binary RP-tree in which*

- *each nonleaf node has a left son and, possibly, a right son;*
- *the longest and shortest distances from the root to leaves differ by at most one and*
- *an item is stored at each node in such a way that an item at a node precedes each of its sons in the sorted order.*

Note that the last condition implies that the smallest item is at the top of the heap. This tree can be stored as an array indexed from 1 to n. The sons of the item in position k are in positions $2k$ and $2k+1$, if these numbers do not exceed n. With clever programming, the unsorted list, the heap and the sorted list can all occupy this space, so no extra storage is needed.

Exercises

8.2.1. (Binary Insertion Sort) Show the steps in a binary insertion sort of the list

9 15 6 12 3 7 11 5 14 1 10 4 2 13 16 8.

8.2.2. (Binary Insertion Sort) Show that the number of comparisons needed to find the location where u_t belongs is about $\log_2 t$. Use this to show that about $n \log_2 n$ comparisons are needed to sort an n long list by this method.
Hint. Using calculus, one can show that

$$\sum_{t=2}^{n} \log_2 t \;\approx\; \int_1^n x \log_2 x \, dx \;\approx\; n \log_2 n.$$

8.2.3. (Binary Insertion Sort) You are to construct the decision tree for the binary insertion sort. Label each leaf with the unsorted list written in one line form, as was done in Figure 8.1. If we are comparing an item with a sorted list of even length, there is no middle item. In this case, use the item just before the middle. This problem is a bit tricky. Unlike the tree in Figure 8.1, not all vertices at the same distance from the root involve the same comparison. As a help, you may want to label the vertices of the decision tree with the comparisons that are being made; for example, $i : j$ could mean that u_i is being compared to s_j.

(a) Construct the decision tree for the binary insertion sort on permutations of $\underline{2}$.

(b) Construct the decision tree for the binary insertion sort on permutations of $\underline{3}$.

8.2.4. (Bucket Sort) Show the steps in bucket sorting 33, 41, 23, 14, 21, 24.

8.2.5. (Bucket Sort)

(a) Carefully state an algorithm for bucket sorting words which have *exactly* k letters each.

(b) Carefully state an algorithm for bucket sorting words which have *at most* k letters each.

8.2.6. (Bucket Sort) Use induction on k to prove that the algorithms in the previous problem work.
Hint. By induction, after $k - 1$ steps we have a list in which the items are in order if the first letter is ignored. What happens to this order within each of the piles we create when we look at the first letter?

8.2.7. (Merge Sort) Using the programming suggestion in the text, show on paper how merge sorting with 4 tapes works for the list

9 15 6 12 3 7 11 5 14 1 10 4 2 13 16 8.

(See the footnote on page 227 for an explanation of "tape.")

8.2.8. (Merge Sort) Suppose we have n items stored on a single tape and that we can sort k items at one time in memory. Explain how to reduce the number of times tapes need to be read for merge sort by first sorting sets of k items in memory.

8.2.9. (Quicksort) Prove that the number of comparisons required when the list is split as unevenly as possible is about $n^2/2$ as claimed. Prove that it is about $n \log_2 n$ when the splits are about as even as possible.

8.2.10. (Quicksort) Suppose that the each time a list of k items is divided in Quicksort the smaller piece contains rk items and the larger contains $(1-r)k$. Let $Q(n)$ be the number of comparisons needed to sort a list in this case.

(a) Show that $Q(n)$ is about $n + Q(rn) + Q((1-r)n)$.

(b) Present a reasonable argument (i.e., it need not be quite a proof) that $Q(n)$ is about $an \ln n$ where

$$1 + a(r \ln r + (1-r)\ln(1-r)) \;=\; 0.$$

(c) Verify that this gives the correct answer when the list is divided evenly each time ($r = 1/2$).

(d) It can be shown that $r = 1/3$ is a reasonable approximation for average behavior. What is $Q(n)$ in this case?

8.3 Sorting Networks

In sorting, items are compared and actions taken as a result of the comparison. Since items must be rearranged, the simplest kind of action one might visualize is to either do nothing or interchange the two items that were compared. We can imagine a hardware device for doing this, which we will call a *comparator*: It has two inputs, say x_1 and x_2, and two outputs, say y_1 and y_2, where y_1 is whichever one of x_1 and x_2 should appear earlier in the sorted list and y_2 is the other. A simple hardware device for sorting consists of a network of interconnected comparators. This is called a *sorting network*. For example, consider the following algorithm for sorting a permutation of $\underline{3}$.

1. If the entry in the second position exceeds the entry in the third position, switch them.
2. If the entry in the first position exceeds the entry in the second position, switch them.
3. If the entry in the second position exceeds the entry in the third position, switch them.

Figure 8.2 shows a sorting network to accomplish this. The data enters at the left end of the network and moves along the lines. The rearranged data emerges at the right end. A vertical connection between two lines represents a comparator—the two inputs emerge in sorted order.

Some of the questions we'd like to ask about sorting networks are

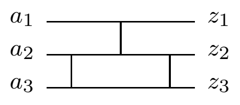
- How fast can sorting networks be?
- How many comparators are needed?
- How can we tell if a network sorts correctly?

8.3.1 Speed and Cost

In this section we'll focus on the first two questions that we raised above.

Sorting networks achieve speed by three methods: fast comparators, parallelism and pipelining. The first method is a subject for a course in hardware design, so we'll ignore it. Parallelism is built into any sorting network; we just haven't realized that in our discussion yet. Pipelining is a common design technique for speeding up computers.

Two things that will make a network cheaper to manufacture is decreasing the number of comparators it contains and increasing the regularity of the layout. Thus we can ask how many comparators are needed and if they can be arranged in a regular pattern.



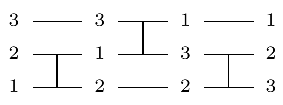
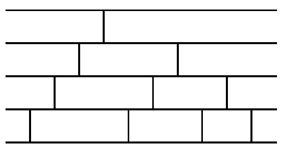


Figure 8.2 Left: A sorting network with inputs a_i and outputs z_i. Vertical lines are comparators Right: What the network does to the inputs 3,2,1.



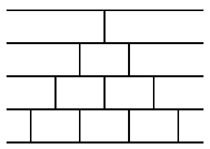
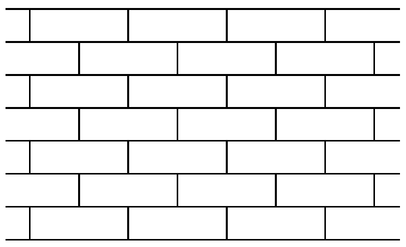


Figure 8.3 Left: A sorting network for the *Bubble Sort.* Right: The same network, but faster.



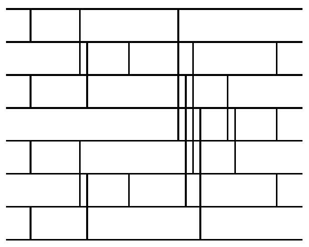


Figure 8.4 Left: A Brick Wall network for eight items. Right: The "Batcher" network for eight items is faster.

Parallelism

All the algorithms we've discussed so far have been carried out sequentially; that is, one thing is done at a time. This may not be realistic for algorithms on a supercomputer. It is certainly not realistic for sorting networks because parallelism is implicit in their design. Compare the two networks in Figure 8.3. They both perform the same sorting, but, as is evident from the second picture, some comparators can work at the same time.

In the expanded form on the left side, it is easy to see that the network sorts correctly. The leftmost set of comparators rising to the right finds the first item; the next set, the second item; the next, the third. The rightmost comparator puts the last two items in order. This idea can be extended to obtain a network that will sort $n > 1$ items in $2n - 3$ "time units," where a time unit is the length of time it takes a comparator to operate. Can we improve on this?

A natural idea is to fill in as many comparators as possible, thereby obtaining a *brick wall* type of pattern as shown in Figure 8.4. How long does the brick wall have to be? It turns out that, for n items, it must have length n. Can we do better than a brick wall if the comparators connect two lines which are not adjacent? Yes, we can. Figure 8.4 shows a "Batcher sort," which is faster than the brick wall. We'll explain Batcher sorts later. A vertical line spanning several horizontal lines represents a comparator connecting the topmost with the bottommost. To avoid overlapping lines in diagrams, comparators that are separated by a small horizontal distances in a diagram are understood to all be operating at the same time. The brick wall in Figure 8.4 takes 8 time units and the Batcher sort takes 6.

How Fast Can a Network Be?

Since a network must have at least $\log_2(n!)$ comparators and since at most $n/2$ comparators can operate at the same time, a network must take at least $\log_2(n!)/(n/2) \approx 2\log_2 n$ time units. It is known that for some C and for all large n, there exist networks that take no more than $C\log_2 n$ time units. This is too complicated for us to pursue here.

Pipelining is an important method for speeding up a network if many sets of items must be sorted. Suppose that we have a delay unit that delays an item by the length of time it takes a comparator to operate. Insert delay units in the network where there are no comparators so that all the items move through the network together. Once a set of items has passed a position, a new set can enter. Thus we can feed a new set of items into the network each time unit. The first set will emerge from the end some time later and then each successive set will take just one additional time unit to emerge. For example, a brick wall network can sort one set of 1,000 items in 1,000 time units, but 50 sets take only 1,049 time units instead of $1,000 \times 50$. This technique is known as pipelining because the network is thought of as a pipeline through which things are moving.

Pipelining is used extensively in computer design. It is obvious in the "vector processing" hardware of supercomputers, but it appears in some form in many central processing units. For example, the INTEL-8088 microprocessor can roughly be thought of as a two step pipeline: (i) the bus interface unit fetches instructions and (ii) the execution unit executes them. (It's actually a bit more complicated since the bus interface unit handles all memory read/write.)

How Cheap Can a Network Be?

Our previous suggestion for using a brick wall network to sort 1,000 items could be expensive—it requires 500,000 comparators. This number could be reduced by using a more efficient network; however, we'd probably have to sacrifice our nice regular pattern. There's another way we can achieve a dramatic saving if we are willing to abandon pipelining.

The brick wall network is simply the first two columns of comparators repeated about $n/2$ times. We could make a network that consists of just these two columns with the output feeding back in as input. Start the network by feeding in the desired values. After about n time units the sorted items will simply be circulating around in the network and can be read whenever we wish.

If we insist on pipelining, how many comparators are needed? From the exercises in the next section, you will see that a Batcher sorting network requires considerably less time and considerably less comparators than a brick wall network; however, it is not best possible. Unlike software sorting, there is a large gap between theory and practice in sorting nets: Theory provides a lower bound on the number of comparators that are required and specific networks provide an upper bound. There is a large gap between these two numbers. In contrast, the upper and lower bounds for pairwise comparisons in software sorts differ from $n\ln n$ by constant factors of reasonable size.

Whether or not we keep our pipelining capabilities, we face a variety of design tradeoffs in designing a VLSI chip to implement a sorting network. Among the issues are the number of comparators, the distance between the lines a comparator must connect, regularity of the design and delay problems. They are beyond the scope of our text.

Exercises

8.3.1. Find the fastest network you can for sorting 3 things and prove that there is no faster network.

8.3.2. Find the fastest network you can for sorting 4 things. If you've read the next section, prove that your network sorts correctly.

8.3.3. Find the fastest network you can for sorting 5 things. If you've read the next section, prove that your network sorts correctly.

8.3.4. Suppose we have a network for sorting n items and we wish to sort less than n. How can we use the network to do this?

8.3.5. In this exercise, comparators are only hooked to adjacent lines as in the brick wall. Find a network that will sort n inputs and that has as few comparators as you can manage.
Hint. Look at some small networks (e.g., $n = 3$ and $n = 4$) and try to find a simple pattern to generalize. If you read ahead a bit before working on this problem, you will encounter Theorem 8.3, which will help you prove that your network sorts.

8.3.6. Here's an idea for an even cheaper sorting network using the brick wall.

(a) Construct a two time unit network that consists of the first two time units of a brick wall. Feed the output of the network back as input. After some time $f(n)$, the output will be sorted regardless of the original input. Find the minimum value of $f(n)$ and explain why this behaves like a brick wall. (Note, that using the network k times means that $2k$ time units have elapsed.)

(b) What can be done if we have such an arrangement for sorting n items and we wish to sort less than n?

(c) When n is even, we can get by with even less comparators. Construct a network that compares the items $2i-1$ and $2i$ for $1 \le i \le n/2$. Call the inputs $x_1, \ldots, x_n$ and the outputs $y_1, \ldots, y_n$. Feed the output of the network back in as follows:

$$\text{new } x_k \;=\; \begin{cases} \text{old } y_n & \text{if } k = 1; \\ \text{old } y_{k-1} & \text{otherwise.} \end{cases}$$

At time $2k$ disable the comparator between lines $2k-1$ and $2k$. Show that this sorts after some number of steps. How many steps are needed in general?

8.3.2 Proving That a Network Sorts

Unlike many software sorts, it is frequently difficult to prove that a network actually sorts all inputs correctly. There is no panacea for this problem. The following two theorems are sometimes helpful.

Theorem 8.2 Zero-One Principle *If a network correctly sorts all inputs of zeroes and ones, then it correctly sorts all inputs.*

Theorem 8.3 Adjacent Comparisons *If the comparators in a network only connect adjacent lines and if the network correctly sorts the reversed sequence $n, \ldots, 2, 1$, then it correctly sorts all inputs.*

We will prove the Zero-One Principle shortly. The proof of the other theorem is more complicated and will not be given.

Since the Adjacent Comparisons Theorem requires that only one input be checked, it is quite useful. Unfortunately, the comparators must connect adjacent lines. To see that this is needed, consider a three input network in which the top and bottom lines are connected by a comparator and there are no other comparators. It correctly sorts the inputs 1,2,3 and 3,2,1, but it does not sort any other permutations of 1,2,3 correctly.

The Zero-One Principle may seem somewhat useless because it still requires that many inputs be considered. We will see that it is quite useful for proving that a Batcher sort works.

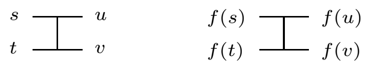


Figure 8.5 Left: A single comparator with arbitrary inputs. Right: Applying a nondecreasing function.

Proof: We now prove the Zero-One Principle. If the network fails to sort some sequence, we will show how to construct a sequence of zeroes and ones that it fails to sort.

The idea behind our proof is the following simple observation: Suppose that f is a nondecreasing function, then a comparator treats $f(s)$ and $f(t)$ the same as it does s and t. This is illustrated in Figure 8.5. It is easy to show by considering the three cases $s < t$, $s = t$ and $s > t$.

Suppose a network has N comparators, has inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_n$. Let f be a nondecreasing function. We will use induction on N to prove that if $f(x_1), \ldots, f(x_n)$ are fed into the network, then $f(y_1), \ldots, f(y_n)$ emerge at the other end.

If $N = 0$, the result is obviously true since there are no comparators present.

Now suppose that $N > 0$ and that the result is true for $N - 1$. We will prove it for N. Focus on one of the comparators that is used in the last time unit. Let network $\mathcal{N}_1$ be the original network with this comparator removed and let network $\mathcal{N}_2$ be the original network with all but this comparator removed. Let $z_1, \ldots, z_n$ be the output of $\mathcal{N}_1$ and use it as the input to $\mathcal{N}_2$. Clearly the output of $\mathcal{N}_2$ is $y_1, \ldots, y_n$.

Let $f(x_1), \ldots, f(x_n)$ be input to the network and let $u_1, \ldots, u_n$ be the output. We can break this into two pieces:

- Let $f(x_1), \ldots, f(x_n)$ be the input to $\mathcal{N}_1$.
  By the induction hypothesis, the output is $f(z_1), \ldots, f(z_n)$.
- Let $f(z_1), \ldots, f(z_n)$ be the input to $\mathcal{N}_2$.
  The output is then $u_1, \ldots, u_n$.

We must prove that $u_1, \ldots, u_n$ is $f(y_1), \ldots, f(y_n)$.

Recall that $\mathcal{N}_2$ consists of a single comparator. Let the input positions i and j go into the comparator. If $k \neq i, j$, then that input position does not go through the comparator and so $f(z_k)$ is treated the same as z_k. Our observation for a single comparator applies to the input positions for z_i and z_j since they feed into the comparator. This proves that $u_1, \ldots, u_n$ equals $f(y_1), \ldots, f(y_n)$ and so proves our claim about nondecreasing functions.

Now suppose that the network fails to sort $x_1, \ldots, x_n$. We will show how to construct a nondecreasing 0-1 valued function f such that the network fails to sort $f(x_1), \ldots, f(x_n)$. Since the sort fails, we must have $y_i > y_{i+1}$ for some $i < n$. Define

$$f(t) \;=\; \begin{cases} 0, & \text{if } t < y_i; \\ 1, & \text{if } t \geq y_i. \end{cases}$$

Since $y_{i+1} < y_i$, we have $f(y_{i+1}) = 0$. Because $f(y_i) = 1$, the network fails to sort $f(x_1), \ldots, f(x_n)$. □

The Batcher Sort

The Batcher sort is a merge sort that is suitable for implementation using a sorting network. Like all merge sorts, it is defined recursively. Let $k = \lceil n/2 \rceil$ be the result of rounding $n/2$ up; for example, $\lceil 3.5 \rceil = 4$. For a Batcher sort, the first k items are sorted and the last $n - k$ are sorted. Then the two sorted sublists are merged.

To write pseudocode for the Batcher sort, let Comparator(x, y) replace x with the smaller of x and y and replace y with the larger. The Batcher sort for array $x_1, \ldots, x_n$ is as follows.

```
BSORT($x_1, \ldots, x_n$)
    If $n = 1$
        Return
    End if
    $k = \lceil n/2 \rceil$
    BSORT($x_1, \ldots, x_k$)                      /* Do these in ... */
    BSORT($x_{k+1}, \ldots, x_n$)                  /* ... parallel. */
    BMERGE($x_1, \ldots, x_n$)
    Return
End

BMERGE($x_1, \ldots, x_n$)
    If $n = 1$
        Return
    End if
    If $n = 2$
        Comparator($x_1, x_2$)
        Return
    End if
    BMERGE2($x_1, \ldots, x_k$; $x_{k+1}, \ldots, x_n$)
    For $i = 1, 2, \ldots, \lceil n/2 \rceil - 1$
        Comparator($x_{2i}$,$x_{2i+1}$)            /* Do these in parallel. */
    End for
    Return
End

BMERGE2($x_1, \ldots, x_k$; $y_1, \ldots, y_j$)
    BMERGE($x_1, x_3, x_5, \ldots, y_1, y_3, y_5, \ldots$)    /* Do these in ... */
    BMERGE($x_2, x_4, x_6, \ldots, y_2, y_4, y_6, \ldots$)    /* ... parallel. */
    Return
End
```

The first procedure is the usual form for a merge sort. The other two procedures are the Batcher merge. They could be combined into one procedure, but keeping track of subscripts gets a bit messy.

The Batcher sort for eight items is shown in Figure 8.6 with some parts labeled. The part labeled S_2 is four copies of a two item Batcher sort. The parts labeled C_n are the comparators in BMERGE on a list of n items. From S_2 through C_4 inclusive is two copies of a four item Batcher sort, one for the top four inputs and one for the bottom four. The parts labeled O and E are the odd and even indexed entries being treated by BMERGE in the call of BMERGE2 with an eight long list.

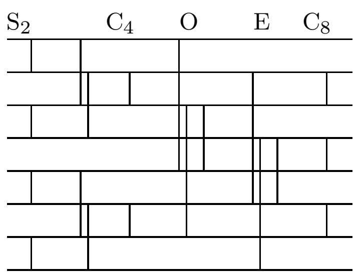


Figure 8.6 The Batcher network for eight items.

To prove that the Batcher sort works, it suffices to prove that the merge part works. Why is this? *Any* recursive sort with a merge will work: The sequence is split in two, each part is sorted and the two parts are merged.

A variation of the Zero-One Principle can be used to prove that the merge works. We need only prove the merge for all sequences of zeroes and ones for which both the first and second halves have been sorted. We remark that $j \le k \le j+1$ whenever `BMERGE2` is called. The idea of the proof is to use induction on n. A key observation is that the number of zeroes in the two sequences that `BMERGE2` passes to `BMERGE` are practically the same: The number of zeroes in the sequence made from the odd subscripted x's and y's less the number of zeroes in the other sequence is 0, 1 or 2. One can then consider the three possible cases separately. This is left as an exercise.

Exercises

8.3.7. Prove that if the comparators in a network only connect adjacent lines and if the network correctly sorts all sequences that consist of ones followed by zeroes, then it correctly sorts all inputs.

8.3.8. (Brick wall network correctness) This exercise is a preparation for the one to follow.

(a) Draw a diagram to show how the sequence 5,4,3,2,1 moves through the brick wall network. Show how to cut out all appearances of 5 and push the network together to obtain a diagram for 4,3,2,1.

(b) Draw diagrams to show how the two input sequences 1,1,0 and 1,0,0 move through the brick wall.

(c) Repeat the previous part for the sequences 1,1,1,0 and 1,1,0,0 and 1,0,0,0.

8.3.9. (Brick wall network correctness) Prove that the brick wall correctly sorts all inputs. You may find an idea for a proof in the previous exercise.

8.3.10. Draw networks for Batcher sorts for n up to 8.

8.3.11. (Batcher merge correctness) Fill in the details of the proof that `BMERGE` works when n is a power of 2; that is, if $x_1, \ldots, x_k$ are in order and $x_{k+1}, \ldots, x_n$ are sorted, then `BMERGE`$(x_1, \ldots, x_n)$ results in a sorted list.
Hint. In this case, division always comes out even and $k = j = n/2$.

8.3.12. (Batcher merge correctness) Fill in the details of the proof that `BMERGE` works for all n; that is, if $x_1, \ldots, x_k$ are in order and $x_{k+1}, \ldots, x_n$ are sorted, then `BMERGE`$(x_1, \ldots, x_n)$ results in a sorted list.

8.3.13. (Batcher network time) Let $S(N)$ be the number of time units a Batcher network takes to sort 2^N things and let $M(N)$ be the number of time units a Batcher network takes to execute `BMERGE` on 2^N things.

(a) Prove that $S(0) = 0$ and, for $N > 0$, $S(N) \le S(N-1) + M(N)$.

(b) Prove that $M(0) = 0$ and, for $N > 0$, $M(N) \le M(N-1) + 1$.

(c) Conclude that $S(N) \le N(N+1)/2$.

(d) What can you say about the time it takes a Batcher network to sort n things when n is not a power of two?

Notes and References

Most books on combinatorial algorithms discuss software algorithms for sorting and a few discuss hardware algorithms; i.e., sorting nets. You may wish to look at the references at the end of Chapter 6. The classic reference is Knuth [4] where you can find more details on the various software sorts that we've discussed. See also pp. 47–75 of Williamson [5].

"Expander graphs" are the basis of sorting networks that require only $\Theta(n \ln n)$ comparators. At present, they have not yet led to any practical networks. For a discussion of this topic see Chapter 5 of the book by Gibbons and Rytter [3]. It is possible to construct fairly cheap networks based on a modified Batcher sort. Like the brick wall, these networks consist of a pattern repeated many times. The brick wall is a 2-long pattern repeated $n/2$ times. The modified Batcher network is a $(\log_2 n)$-long pattern repeated $(\log_2 n)$ times. See [2] for details. See also [1]. A proof of Theorem 8.3 can be found on p. 63 of [5].

1. Edward A. Bender and S. Gill Williamson, Periodic sorting using minimum delay, recursively constructed sorting networks, *Electronic J. Combinatorics* **5** (1998), R5.
2. E. Rodney Canfield and S. Gill Williamson, A sequential sorting network analogous to the Batcher merge, *Linear and Multilinear Algebra* **29** (1991), 42–51.
3. Alan Gibbons and Wojciech Rytter, *Efficient Parallel Algorithms*, Cambridge Univ. Press (1988).
4. Donald E. Knuth, *The Art of Computer Programming. Vol. 3 Sorting and Searching*, 2nd ed., Addison-Wesley (1998).
5. S. Gill Williamson, *Combinatorics for Computer Science*, Dover (2002).