Chapter 6

Sorting Algorithms

6.1 Introduction

In this chapter we consider some basic ideas in designing algorithms which are crucially based on the notion of *order* on a set of elements of size n. We will assume that the order is total, i.e., for every two elements x and y either x is less than or equal to y or y is less than or equal to x in that order.

Before we discuss sorting algorithms, let us consider the problem of searching an element in a sorted array. Suppose that we are given a sorted array A of size n and an element x. We are interested in finding if there exists an i such that $A[i]$ equals x. On a single processor, we can accomplish this using binary search in $O(\log n)$ time. The idea is to compare x with the middle element of the array. If the middle element equals x, we are done otherwise the range of indices of A which can possibly have x is divided by a factor of 2.

It is easy to generalize this divide-and-conquer strategy for p processors by dividing the range of indices into contiguous $p+1$ ranges. In the second time step the range of indices is reduced by a factor of $p+1$. Therefore, we have $\log n/\log(p+1)$ parallel algorithm on p processors. In an unsorted array, search would require n/p time in the worst case. This difference in search time motivates sorting of arrays used in software systems.

This chapter is organized as follows. Section 6.2 discusses sorting algorithms that are based on swapping consecutive entries that are out of order. This is a general class of algorithms that are very natural. However, these algorithms take $O(n^2)$ sequential time and $O(n)$ parallel time in the worst case. Section 6.3 discusses sequential and parallel versions of *mergesort*. Section 6.4 discusses sequential and parallel versions of *quicksort*. Since Merge Sort and Quick Sort are based on divide-and-conquer paradigm, they are revisited in Chapter 12. Section 6.5 discusses bitonic sort. All these sorting algorithms are based on comparison of two entries in the array. Finally, Section 6.6 describes a sorting algorithm that is not based on comparison of two entries.

6.2 Sorting Algorithms based on Swapping Consecutive Entries

Let A be an array of distinct integers. Our goal is to sort the array. This problem at face does not appear to be searching for a satisfying element in a lattice. However, with a little effort it can be viewed from that angle. Observe that the problem of sorting the array is same as finding an permutation π such that on applying that permutation to the array, we get a sorted array. For example, if the input array is $[45, 12, 15]$, then we know that the permutation $[3, 1, 2]$ will sort the array once the entry i in array A goes to $\pi(i)$. There are many different ways we can represent a permutation. Let us define the *inversion* number of any entry i as the number of entries less than i that appear to the right of i in the permutation. Thus, the permutation $[3, 1, 2]$ can be equivalently written using the concept of inversions as $[2, 0, 0]$ because there are two numbers less than 3 that appear to the right of 3, zero numbers less than

1 that appear to the right of 2 and zero numbers less than 2 that appear to the right of 2. The reader can verify that there is a 1-1 correspondence between the set of inversions and the set of permutations.

We now consider the set of all inversions. Note that the last entry on an inversion table is always zero because there cannot be any inversion after the last entry in the array. The first entry can have $0 \dots n-1$ inversions, the second entry can have $0 \dots n-2$ inversions, and so on. Thus, the total number of inversion vectors possible are $n!$. We define an order between two inversion vectors base on component-wise order. The set of all inversions forms a finite distributive lattice under this order. The bottom element is the zero vector which corresponds to the identity permutation. When the zero inversion vector is applied to any array, we get back the same array. Our goal is to find the inversion vector which on its application gives us the sorted array. G is initialized to the zero inversion vector. Let us formalize the definition of a feasible inversion vector. Suppose that we are at an inversion vector G such that it is less than the unique inversion vector that sorts A. When G is applied to A, the array is not sorted. This means that there exists an index i such that still has nonzero inversions, i.e, there exists $j > i$ such that $A[j] < A[i]$. It is clear that any inversion vector in which the number of inversions for i is not increased cannot result in the sorted array. To check for forbidden indices, instead of checking for all inversions, we will only check for *immediate* inversion in our first algorithm. If $A[i] > A[i+1]$ on applying G, then clearly i is forbidden in G. Swapping $A[i]$ and $A[i+1]$ will advance G vector by incrementing $G[i]$.

Instead of first finding the inversion vector and then applying it, we will continue to apply inversions as we traverse the lattice searching for the optimal inversion vector. In fact, we will not even maintain the inversion vector; we will only keep the effect of applying the inversion vector to the original array.

Algorithm LLP-Sort1: A High-Level LLP Sorting algorithm based on immediate swaps

```
1 P_j: Code for thread j
2 var A: array[1..n] of int; // input array
3 G: array[1..n] of 0..n − 1 initially G[i] = 0 for all i// abstract variable: just for correctness
4     forbidden(j): A[j] > A[j + 1] // G[j] is missing at least one inversion
5         advance(j): swap(A[j], A[j + 1]) // increment G[j]
```

Observe that in this algorithm, in a parallel setting, it is important that the statement $swap(A[j], A[j+1])$ is executed atomically whenever $A[j] > A[j+1]$. Algorithm LLP-Sort1 is a non-deterministic algorithm since multiple j may be forbidden at any point. One can create many instances of deterministic algorithms with different order of evaluating forbidden indices.

Algorithm Bubble-Sort: Bubble Sort B

```
1 var A: array[1..n] of int; // input array
2 found: boolean;
3 repeat
4     found := false;
5     for j := 1 to n − 1 do
6         if A[j] > A[j + 1]
7             found := true;
8             swap(A[j], A[j + 1])
9 until (found = false);
```

Algorithm Bubble-Sort checks forbidden indices from 1 to $n-1$ in round robin order until no forbidden index is found. We leave it as an exercise to show that the repeat loop is executed at most n times giving us the sequential time complexity of $O(n^2)$.

Another schedule is to increase the size of the sorted array one at a time. Suppose that $A[1..i-1]$ is sorted. This means that the inversion vector is zero vector for $A[1..i-1]$. We now consider the array $A[1..i]$. Since there is at most one entry added at the end, the inversion number for any entry can increase by at most one. We simply have to check for inversion with respect to the new entry.

Algorithm Insertion-Sort: Insertion Sort B

```
1 var A: array[1..n] of int; // input array
2 for i := 1 to n do
3 // Assuming that A[1..i − 1] is sorted, ensure that A[1..i] is sorted
4     for j := i − 1 to 1 do
5         if A[j] > A[j + 1]
6             swap(A[j], A[j + 1])
```

The time complexity is clearly $O(n^2)$ due to nested *for* loops. We leave it for the reader to show how the second for loop can be cut short.

We now adapt Algorithm LLP-Sort1 for parallel execution. In particular, we saw that the algorithm required atomicity of checking for the forbidden predicate and swaps. We can avoid this expensive synchronization by scheduling the forbidden indices as follows. In the first phase only odd indices check for the forbidden predicate and swap with the right entry if necessary. In the second phase only even indices carry out this operation. These phases are executed alternatively until there is no forbidden index left. This algorithm, called Odd-Even Sort, is similar to the sequential bubble sort, in that it is based on swapping out of order entries. It is shown in Algorithm OddEven-Sort.

Algorithm OddEven-Sort: Odd-Even Sort B

```
1  var A: array[1..n] of int; // input array
2  repeat
3      found := false;
4      forall odd(j), j ∈ [1..n − 1] in parallel do
5          if (A[j] > A[j + 1])
6              found := true;
7              swap(A[j], A[j + 1])
8      forall even(j), j ∈ [1..n − 1 in parallel do
9          if (A[j] > A[j + 1])
10             found := true;
11             swap(A[j], A[j + 1])
12 until (¬found);
```

It is easy to see that the repeat loop is executed at most n times. Hence, the algorithm takes $O(n)$ time and $O(n^2)$ work. An example is shown in Fig. 6.1.

6.3 Merge Sort

Merge Sort is the classic example of divide-and-conquer method of designing algorithms. To sort an array, A, assume that the left-half and the right-half of the arrays are sorted. Then we are simply left with the task of merging these two sorted halves. How do we sort the left and the right halves? By using recursion with the base case when the

8 1 4 6 9 5 2 7; original unsorted array
1 8 4 6 5 9 2 7; all odd entries are swapped in parallel with next entries, if inverted
1 4 8 5 6 2 9 7; even entries are swapped, if inverted
1 4 5 8 2 6 7 9; odd entries are swapped
1 4 5 2 8 6 7 9; ; even entries are swapped, if inverted
1 4 2 5 6 8 7 9; odd entries are swapped, if inverted
1 2 4 5 6 7 8 9; even entries are swapped, if inverted

Figure 6.1: An Execution of Odd-Even Sort

halves have only single elements and are already sorted. So, it is sufficient to consider the following problem: we have two sorted arrays B and C, each of size n. We would like to merge them into another array D such that D is sorted.

Algorithm MergeSort: MergeSort

```
1 MergeSort(A, low, high)
2 if low < high then
3     mid = ⌊(lo + high)/2⌋ ;
4     B := MergeSort(A, low, mid) in parallel with C := MergeSort(A, mid + 1, high)
5     return MergeTwo(B, C);
```

It is easy to design a sequential algorithm that merges them two arrays B and C into D. We simply keep two indices i and j in arrays B and C, respectively. At any step of the algorithm, we compare $B[i]$ and $C[j]$. If $B[i]$ is smaller than $C[j]$, then we copy $B[i]$ into the next available slot in D and advance index i. If $C[j]$ is smaller than $B[i]$, then we copy $C[j]$ into the next available slot in D and advance index j. This algorithm takes $O(n)$ time.

Algorithm MergeTwo: Merging Two arrays

```
1  var
2      B: array[1..m] of int; // input array
3      C: array[1..n] of int; // input array
4      D: array[1..m + n] of int; // output array
5  int i, j, k := 0, 0, 0;
6  while (i < m) and (j < n) do
7      if (B[i] < C[j]) then {D[k] = B[i]; i + +; }
8      else {D[k] = C[i]; j + +;}
9      k + +;
10 while (i < m) { D[k] = B[i]; i + +; k + +; }
11 while (j < n) { D[k] = C[j]; j + +; k + +; }
```

We now devise a parallel algorithm to merge two sorted arrays. For simplicity, we assume that all elements are unique. This algorithm is based on finding the location in D where each element of B and C will appear. If every element in B and C determines its rank in parallel, it can write that value at the correct location in D.

Define the rank of any element x as the number of elements in B and C that are less than x, i.e., $rank(B[i], D)$ = the number of elements in B less than $B[i]$ + $rank(B[i], C)$. The first number is simply i (for arrays that start with index 0). The second number can be determined using binary search in $O(\log(n))$ time. So the overall time is:

$T(n) = O(1) + O(\log(n))$. This algorithm requires concurrent reads but no concurrent writes, so a *CREW* PRAM is sufficient.

6.4 Quicksort

Quicksort (due to C.A.R. Hoare) is one of the fastest sorting algorithms on sequential computers. It has $O(n^2)$ worst case time complexity but requires $O(n \log n)$ time on average. The algorithm is based on first partitioning the array into two parts based on a *pivot*. Once we have the property that the lower half of the array is less than or equal to pivot and the upper half of the array has elements greater than pivot, then we can recurse on each half. Since sorting on each half is independent, they can be sorted in parallel.

There are multiple methods to choose a pivot to partition the array A. Choosing an element at random is an easy method that will result in approximately equal sized partitions on average. This gives us the average case sequential time complexity of $O(n \log n)$. In the worst case, however, the recursion may reduce the range by only 1, resulting in the sequential time compexity of $O(n^2)$.

Algorithm QuickSort: QuickSort

```
1 procedure QuickSort(A, low, high)
2 if low < high then
3     pivot := choosePivot()
4     (p, q) := partition(A, pivot, low, high)
5     QuickSort(A, low, p) in parallel with QuickSort(A, q, high)
```

We describe a slight variant of the Quicksort algorithm in which we partition the array into three parts. Algorithm QuickSort takes array A, *low* and *high* as input parameters. The part of the array this method sorts is given by

$$\{A[i] \mid low \leq i < high\}$$

Note that when *low* equals *high*, the range is empty.

The method *partition* returns two indices p and q. All elements in the range $[lo \ldots p)$ are strictly less than the pivot, in the range $[p \ldots q)$ are equal to the pivot and in the range $[q \ldots high)$ are greater than the pivot. We only need to recurse on the first and the third part. Depending upon the pivot, any (or both!) of the first and the third parts may be empty.

Once we have a pivot, how do we partition the array into three parts: the first partition with all elements less than the pivot, the second one with all elements equal to the pivot and the the third partition with elements strictly greater than the pivot. Algorithm Three-Way-Partition is due to Dijkstra who called this problem as the Dutch National Flag problem. The *while* loop maintains the invariant that entries $[low \ldots p)$ are less than pivot, $[p \ldots q)$ are equal to pivot and $[k \ldots high)$ are greater than pivot. The algorithm initializes p and q to *low*, therefore the first two ranges are empty initially and trivially satisfies the invariants. It also initializes k to *high* making the range $[k \ldots high)$ empty and thereby ensuring that the invariant holds initially. The range $[q..k)$ corresponds to the initial input and initially contains entries that may be less than, equal to, or greater than the pivot. In each iteration of the *while* loop, this range is reduced by one by either increasing q or decreasing k. Depending upon the comparison between $A[q]$ and the pivot, the entry $A[q]$ is placed in the appropriate range maintaining the invariants.

It is easy to verify that the algorithm takes $O(n)$ time when the range has n elements. A parallel version for *QuickSort* based on divide and conquer is discussed in Chapter 12.

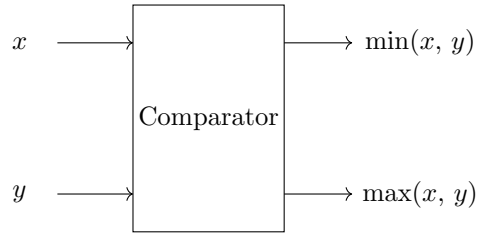
6.5 Bitonic Sort

Bitonic sort is a parallel sorting algorithm. One of its nice properties is that it is an *oblivious* sorting algorithm. A sorting algorithm is oblivious if its control flow is independent of the actual data. For example, an oblivious sorting

Algorithm Three-Way-Partition: Three-Way-Partition

```
1  function Partition(A, pivot, low, high) returns (int, int)
2      k := high;
3      if k > low then
4      p := low
5      q := low
6      while q < k do
7          if A[q] < pivot then
8              swap( A[p], A[q])
9              p := p + 1; q := q + 1;
10         else if (A[q] > pivot) then
11             k := k − 1
12             swap( A[q], A[k])
13         else
14             q := q + 1;
15     return (p, q)
```

algorithm will take the same steps if the input array is already sorted! Independence of the control flow from the actual data is nice because there is the execution of each thread is deterministic and hence it is easy to map the control flow onto a GPU (and to build hardware for sorting). Oblivious sorting algorithms can be built using a network of *comparators*. A comparator can be considered as a block with two inputs x and y and two outputs lo and hi such that lo is minimum of x and y and hi is the maximum of x and y. Let us denote the comparator as a function that returns a pair (lo, hi). For example, $comparator(3, 5)$ will return $(3, 5)$ and $comparator(5, 3)$ will also return $(3, 5)$. Pictorially, a comparator can be shown as



By combining comparators, we can create a *sorting network* that produces a sorted output from any input.

Odd-even sort that we studied earlier is also an oblivious sorting algorithm. The sorting network for odd-even sort for eight elements is shown in Fig. 6.2.

Before we give the details of bionic sort, we describe an important principle, called *zero-one principle*, that is quite useful in analysis of algorithms based on sorting networks. The zero-one principle says that if a sorting network gives correct output on all inputs that consist of only zeroes and ones, then that sorting network will give correct output on all inputs. If we can show zero-one principle, then analyzing sorting networks becomes much simpler; we only need to consider binary inputs. To prove the zero-one principle, we analyze how a comparator works when instead of presenting input $x = (x_1, x_2)$, it is presented with input $(f(x_1), f(x_2))$. Let f be any *monotonic* function, i.e., $x_1 \leq x_2$ implies that $f(x_1) \leq f(x_2)$. It is clear that f distributes over min and max, i.e.,
$f(min(x_1, x_2)) = min(f(x_1), f(x_2))$, and
$f(max(x_1, x_2)) = max(f(x_1), f(x_2))$.

Hence, when $(f(x_1), f(x_2))$ is presented to a comparator, it outputs $f(min(x_1, x_2))$ and $f(max(x_1, x_2))$.

By applying induction on the number of comparators, it can be shown that when the input sequence x is applied and any wire in the network has value x_i, then it has the value $f(x_i)$ when the input sequence $f(x)$ is applied.

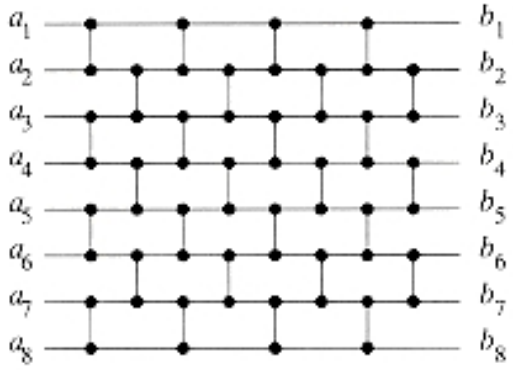


Figure 6.2: Sorting Network for Odd-Even Sort

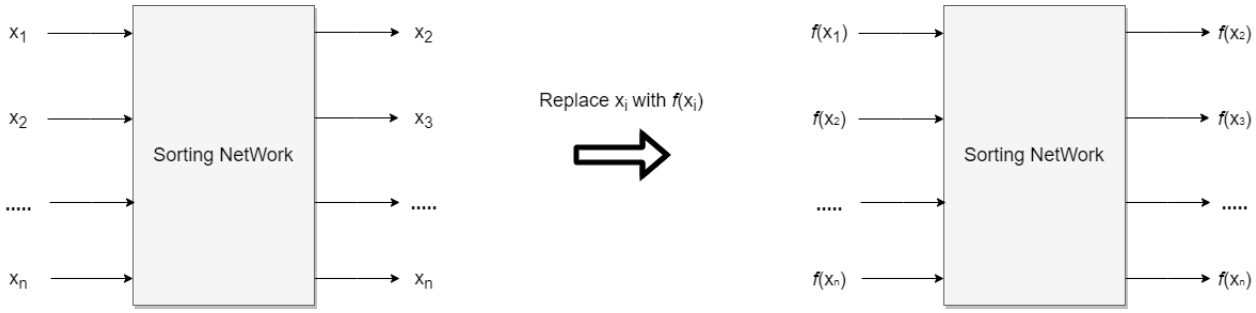


Figure 6.3: Replacing inputs of a sorting network

Armed with the above observation, we show the zero-one principle. Suppose that a network sorts all zero-one sequences correctly but does not sort a sequence of arbitrary numbers. Let that sequence be $(x_1, x_2, \ldots x_n)$ such that the network puts x_i higher than x_j even though x_i is smaller than x_j. We now show a zero-one sequence in which the network gives wrong answers. We use a monotonic function f that maps every element in x to 0 if it is less than or equal to x_i and 1 otherwise. Since x_i is placed higher than x_j, from our earlier observation $f(x_i)$ is placed higher than $f(x_j)$ which is incorrect because 0 is placed higher than 1. Hence, we conclude that there is no sequence of arbitrary elements that is sorted incorrectly by the network.

Bitonic sort is in some sense similar to Mergesort, in that it also merges two sorted sequences. The merge in bitonic sort, called bitonic-merge, has a very simple parallel implementation.

Let us begin with the definition of a bitonic sequence. A bitonic sequence is more conveniently viewed as a circular array. We define a range $A[l \ldots h]$ in a circular array A as follows. If l is less than or equal to h, then this range equals

$$[A[i]|l \leq i \leq h].$$

If l is greater than h, then this range equals $[A[i]|l \leq i \leq n-1]$ concatenated with $[A[i]|0 \leq i \leq h]$. In other words, we traverse all the indices going from l to h by incrementing the index by 1 modulo n. For example, if $A = [2, 4, 6, 8]$, then $A[2..3] = [6, 8]$, but $A[3..2] = [8, 2, 4, 6]$.

A sequence of numbers A of size n is called *ascending* if it is sorted in ascending order,i.e.,

$$\forall i : 0 \leq i < n-1 : A[i] \leq A[i+1]$$

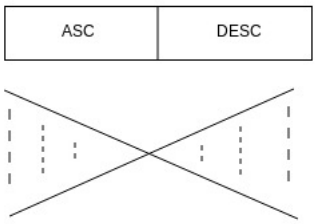


Figure 6.4: Bitonic Merge Lemma

It is *descending* if

$$\forall i : 0 \leq i < n-1 : A[i] \geq A[i+1]$$

Definition 6.1 (Bitonic Sequence) *A sequence* $A[0], A[1], \ldots A[n-1]$ *is* bitonic *if there exists two indices* l *and* h *such that* $A[l..h]$ *is an ascending sequence and* $A[h..l]$ *is a descending sequence.*

It can be easily verified that $A[l]$ corresponds to a minimum value in array A and $A[h]$ corresponds to a maximum value.

For example, $A = [4, 3, 1, 5, 8, 9, 7, 4]$ is a bitonic sequence because $A[2..5] = [1, 5, 8, 9]$ is ascending and $A[5..2] = [9, 7, 4, 4, 3, 1]$ is descending. Array $B = [5, 6, 7, 6, 5, 4, 3, 2]$ is also bitonic because $B[7..2] = [2, 5, 6, 7]$ is ascending and $B[2..7] = [7, 6, 5, 4, 3, 2]$ is descending. It can be verified that the sequence $[2, 4, 9, 3, 7]$ is not bitonic.

We now state a crucial property of bitonic sequences.

Lemma 6.2 (Bitonic Merge Lemma) *Let* A *be a bitonic sequence of length* $2n$*. Consider* B *and* C *defined as follows.* $B = [min(A[i], A[i+n] | 0 \leq i < n]$ $C = [max(A[i], A[i+n] | 0 \leq i < n]$ *Then* B *and* C *are bitonic and all values in* B *are less than or equal to all values in* C*.*

Proof: *(Sketch)* We apply the zero-one principle to prove this Lemma. Note that all bitonic sequences of zeroes and ones are either of the form $0^k1^l0^m$ or $1^k0^l1^m$ for some $k, l, m \geq 0$. By doing case analysis of such sequences the lemma can be shown.

■

Fig. 6.4 shows the process of splitting an array A of size $2n$ such that the first half is ascending and the second half is descending and then taking componentwise minimum B and maximum C. It is easy to see visually that B and C are bitonic and all values in B are less than or equal to all values in C.

We now give the algorithm for bitonic sort. As mentioned earlier, the structure of the algorithm is very similar to that of a Mergesort. In mergesort, two halves of the arrays are recursively sorted and then merged. In Bitonic sort, we use bitonic merge which is more easier to parallelize using a comparator network. In Bitonic sort, instead of sorting both halves in the same direction (ascending or descending), we sort the first half in the ascending order and the second half in the descending order. The resulting array is a bitonic sequence and we apply bitonic merge to it. Bitonic merge splits the array into two parts and does pairwise comparison according to Lemma 6.2. As a consequence of the lemma, all the elements in the lower half are less than the elements in the upper half. Furthermore, each of the halves are bitonic and we can call bitonic merge recursively on each half.

An example of bitonic sorting network for four inputs is shown in Fig. 6.5 and for eight inputs in Fig. 6.6.

BitonicSort algorithm recursively calls BitonicSort(L) and BitonicSort(R) on the left half L and right-half R. BitonicSort(L) and BitonicSort(R) can be done in parallel. BitonicMerge first does $n/2$ comparisons in parallel. It then recursively calls bitonicMerge on both halves. Hence, it takes $O(\log n)$ time. We can now calculate $T(n)$, parallel time, for bitonic sort. We have,

Algorithm BitonicSort: Bitonic Sort

```
1  procedure bitonicSort(int low, int n, boolean isAscending)
2  // Assume that n is a power of 2
3  begin
4      if (n > 1)
5          int m = n/2;
6          bitonicSort(lo, m, true) in parallel bitonicSort(lo + m, m, false);
7          bitonicMerge(lo, n, isAscending);
8  end

9  procedure bitonicMerge(int low, int n, boolean isAscending)
10 begin
11     if (n > 1)
12         int m = n/2;
13         for (int i = low; i < low + m; i++) in parallel do
14             int j = i + m;
15             if ((A[i] > A[j]) and isAscending) swap(A[i], A[j]);
16             else if ((A[i] < A[j]) and ¬isAscending) swap(A[i], A[j]);
17         bitonicMerge(lo, m, isAscending) in parallel bitonicMerge(lo + m, m, isAscending)
18 end
```

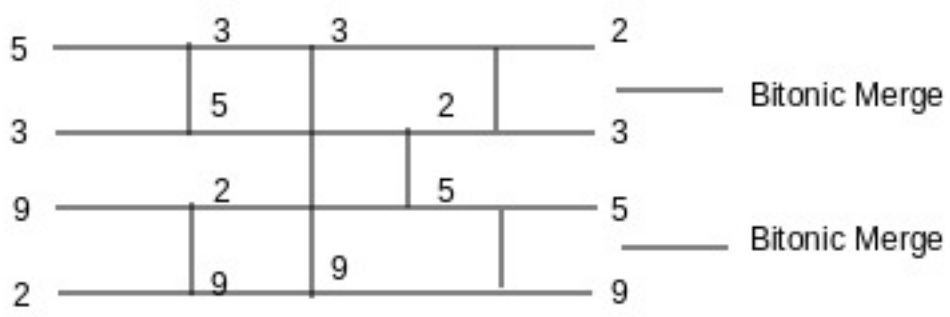


Figure 6.5: Example of a Bitonic Sorting network on four inputs.

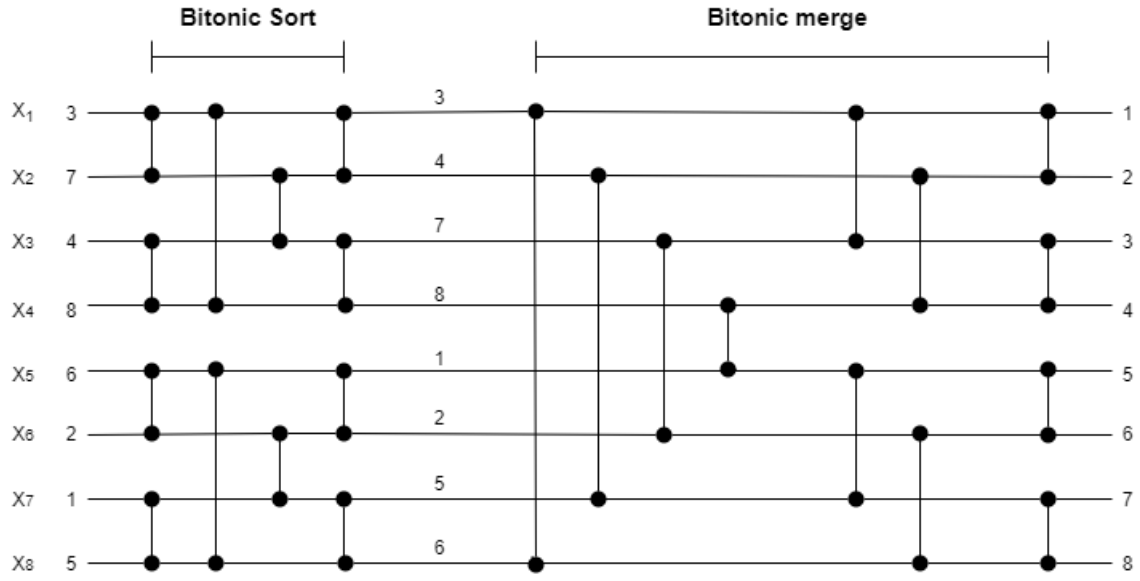


Figure 6.6: Example of a Bitonic Sort on eight inputs.

$T(n) = T(n/2) + O(log(n))$
$T(1) = O(1)$
Therefore, $T(n) = T(n/2) + log(n) = O(log^2 n)$.

The algorithm uses $O(n \log^2 n)$ comparators and therefore its work complexity is also $O(n \log^2 n)$.

6.6 Radix Sort

All our sorting algorithms, so far, have been based on comparison. Any comparison-based sorting algorithm must do at least $O(n \log n)$ work. We now show an algorithm that works when keys for sorting have a fixed number of digits (in any radix r, generally a power of 2). In our examples below, we simply use digits to the base 10, as they are easy for us humans. Suppose that we need to sort the following list. $[85, 72, 94, 45, 13, 12, 61, 81]$. Here, our keys have two digits. The idea of the radix sort is to sort numbers, one digit at a time. When sorting on a single digit, we use a simple linear time sort by exploiting that there are only r possibilities for each digit. On getting any number, we can simply add it to the pile associated with that value. The number of passes we will make on the array is equal to the number of digits.

It may appear that it is easier to sort starting from the most significant digit (msd) first. If we employed that strategy, we would get $[13, 12, 45, 61, 72, 85, 81, 94]$ after the first pass. The second pass would consist of sorting all subarrays with the same msd and we would get the array $[12, 13, 45, 61, 72, 81, 85, 94]$. The problem with this approach is that we are forced to maintain different subarrays - one for each digit after the first pass. With every subsequent pass, it gets even more cumbersome. Hence, somewhat counterintuitively, we will employ the least significant digit first strategy. After the first pass, $[61, 81, 72, 12, 13, 94, 85, 45]$. We do not need to remember any sublists that are created during the first pass. Now, we sort on the second least significant digit. We need to ensure that if two numbers have the same digit, then their order is preserved. In other words, we require our sorting algorithms at each pass to be *stable*. After the second pass, we get the sorted array $[12, 13, 45, 61, 72, 81, 85, 94]$. Since 12 appeared before 13 after the first pass, the order is preserved after the second pass.

Thus, the sequential algorithm is simply stated as follows:

for $i := 1$ to k do // the key has k digits
use stable sort for Array A on digit i

The time complexity of this algorithm is $O(kn)$ assuming that the stable sort is accomplished in $O(n)$ time.
Problem 8 asks you to design a parallel algorithm for Radix Sort.

6.7 Summary

The following table lists all the algorithms for comparison-based sorting discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Sorting	Odd-Even Sort	$O(n)$	$O(n^2)$
Sorting	Insert Sort	$O(n)$	$O(n^2)$
Sorting	Sequential Merge Sort	$O(n \log n)$	$O(n \log n)$
Sorting	Parallel Merge Sort	$O(\log^2 n)$	$O(n \log^2 n)$
Sorting	Sequential Quick Sort	$O(n^2)$	$O(n^2)$
Sorting	Bitonic Sort	$O(\log^2 n)$	$O(n \log^2 n)$

6.8 Problems

1. Implement the binary search algorithm discussed in Section 6.1.
2. Show that any algorithm that is based on comparison of consecutive entries must take $O(n^2)$ comparisons in the worst case.
3. Give a parallel Insertsort algorithm based on parallel-prefix.
4. Give an algorithm to merge k sorted arrays of size n into a single sorted array.
5. We have partitioned the array into three parts in the QuickSort algorithm. Give a version of the QuickSort algorithm in which the array is partitioned only in two parts: entries that are less than *pivot* and the entries that are greater than or equal to the pivot.
6. Give a randomized version of Quicksort in which the pivot is chosen at random. Give the expected running time of your algorithm.
7. Give a parallel Quicksort algorithm based on parallel-prefix that works as follows. The procedure takes an array and divides into three parts: all elements less than the pivot, all elements equal to the pivot and all elements greater than the pivot. It can then recurse on each of the subparts.
8. Give a parallel algorithm for RadixSort using parallel-prefix.

6.9 Bibliographic Remarks

Quicksort is a divide-and-conquer algorithm that was invented by Tony Hoare in 1962 [Hoa61] It is one of the most widely used sorting algorithms and has a time complexity of $O(n \log n)$ in the average case. Merge sort is another divide-and-conquer algorithm that was first described by John von Neumann in 1945. Radix sort is a non-comparison-based sorting algorithm that was first described by Herman Hollerith in 1887.