Divide and Conquer*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

X033533-Algorithm: Analysis and Theory

*Special thanks is given to Prof. Yijia Chen for sharing his teaching materials.

Outline

1. Divide-and-Conquer
   - Basic Technique
   - An Introductory Example: Multiplication
   - Recurrence Relations
2. Applications
   - Search and Sort
   - Median
   - Matrix Multiplication
3. Sorting Networks
   - Comparison Networks
   - Zero-One Principle
   - Construction of a Sorting Network

Divide-and-Conquer Strategy

The divide-and-conquer strategy solves a problem P by:

(1) Breaking P into subproblems that are themselves smaller instances of the same type of problem.
(2) Recursively solving these subproblems.
(3) Appropriately combining their answers.

Key Works

The real work to implement Divide-and-Conquer strategy is done piecemeal, where the key works lay in three different places:

(1) How to partition problem into subproblems.
(2) At the very tail end of the recursion, how to solve the smallest subproblems outright.
(3) How to glue together the partial answers.

Johann C.F. Gauss

Johann Carl Friedrich Gauss

1777 - 1855

$$1 + 2 + \cdots + 100 = \frac{100 \cdot (1 + 100)}{2} = 5050.$$

Multiplication for Complex Numbers

Gauss once noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a+b)(c+d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd.$$

In our big-O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. However, this modest improvement becomes *very significant when applied recursively*.

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is a power of 2.

Lemma: $\forall n \in \mathbb{N}, \exists n'$ *with* $n \leq n' \leq 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The additions take linear time, as do the multiplications by powers of 2 (merely left-shifts). The significant operations are the four $n/2$**-bit multiplications**; these we can handle by *four recursive calls*.

Our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers, and then evaluates the preceding expression in $O(n)$ time.

Writing $T(n)$ for the overall running time on n-bit inputs, we get **the recurrence relation**:

$$T(n) = 4T(n/2) + O(n)$$

Solution: $O(n^2)$.

By **Gauss**'s trick, three multiplications, $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, suffice, as

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

A divide-and-conquer algorithm for integer multiplication

```
MULTIPLY(x,y)
    Input: positive integers x and y, in binary
    Output: their product
 1: n = max(size of x, size of y) rounded as a power of 2.
 2: if n = 1 then
 3:     return xy.
 4: end if
 5: x_L, x_R = leftmost n/2, rightmost n/2 bits of x
 6: y_L, y_R = leftmost n/2, rightmost n/2 bits of y
 7: P_1 = MULTIPLY(x_L, y_L)
 8: P_2 = MULTIPLY(x_R, y_R)
 9: P_3 = MULTIPLY(x_L + x_R, y_L + y_R)
10: return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^{n/2} + P_2.
```


The time analysis

The recurrence relation: $T(n) = 3T(n/2) + O(n)$

- The algorithm's recursive calls form a **tree structure**.
- At each successive level the subproblems get halved in size.
- At the $(\log_2 n)^{\text{th}}$ level, the subproblems get down to size 1, and so the recursion ends.
- The **height** of the tree is $\log_2 n$.
- The **branching factor** is 3: each problem recursively produces three smaller ones, with the result that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

For each subproblem, a linear amount of work is done in identifying further subproblems and combining their answers. Therefore the total time spent at depth k in the tree is $3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n)$.

The time analysis (cont'd)

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

At the very top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is

$$O\left(3^{\log_2 n}\right) = O\left(n^{\log_2 3}\right)$$

Between these two endpoints, the work done increases *geometrically* from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

The sum of any increasing geometric series is, within a constant factor, simply the last term of the series. Therefore the overall running time is

$$O(n^{\log_2 3}) \approx O(n^{1.59}).$$

We can do even better!

Master Theorem

If

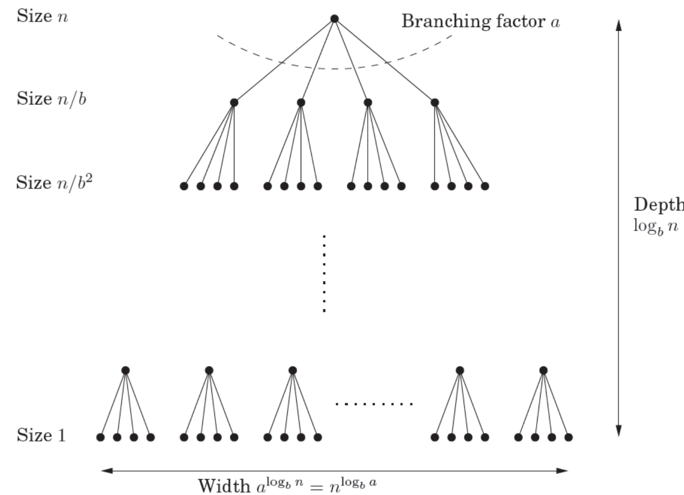
$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

Proof of Master Theorem

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

Next, notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. *This is the height of the recursion tree.*

The branching factor of the recursion tree is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k.

The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

Proof of Master Theorem

The total work done is

$$\sum_{k=0}^{\log_b n} \left(a^k \times O\left(\frac{n}{b^k}\right)^d\right) = \sum_{k=0}^{\log_b^n} \left(O(n^d) \times \left(\frac{a}{b^d}\right)^k\right).$$

It's the sum of a geometric series with ratio a/b^d.

- ▷ The ratio is less than 1. Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
- ▷ The ratio is greater than 1. The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

- ▷ The ratio is exactly 1. In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

BinarySearch Algorithm

BINARYSEARCH($A[1 \dots n]$)

Input: An array $A[1..n]$ in nondecreasing order and an x.
Output: j if $x = A[j]$, $1 \le j \le n$, and 0 otherwise.

1: $low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$;
2: **while** $low \le high$ **and** $j = 0$
3: $\quad mid \leftarrow \lfloor (low + high)/2 \rfloor$;
4: $\quad$ **if** $x = A[mid]$ **then** $j \leftarrow mid$ **break**;
5: $\quad$ **else if** $x < A[mid]$ **then** $high \leftarrow mid - 1$;
6: $\quad$ **else** $low \leftarrow mid + 1$;
7: **end while**
8: **return** j

The Time Analysis

To find a key x in $A[1, \cdots, n]$ in sorted order, we first compare x with $A[n/2]$, and depending on the result we recurse either on the first half of the array $A[1, \cdots, n/2 - 1]$, or on the second half $A[n/2, \cdots, n]$.

The recurrence function is

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(1),$$

By Master Theorem, $a = 1$, $b = 2$, $d = 0$, and thus the running time should be $O(\log n)$.

MergeSort Algorithm

MERGESORT($a[1 \ldots n]$)

Input: an array of numbers $a[1 \ldots n]$
Output: A sorted version of this array

1: **if** $n > 1$ **then**
2: return MERGE(MERGESORT($a[1 \ldots \lfloor n/2 \rfloor]$),
3: MERGESORT($a[\lfloor n/2 \rfloor + 1 \ldots n]$)),
4: **else** return a.

MergeSort Algorithm (Cont.)

MERGE($x[1 \ldots k], y[1 \ldots \ell]$)

Input: two sorted arrays x and y
Output: A sorted version of the union of x and y

1: **if** $k = 0$ **then** return $y[1 \ldots \ell]$
2: **if** $\ell = 0$ **then** return $x[1 \ldots k]$
3: **if** $x[1] \leq y[1]$ **then**
4: return $x[1] \circ$ MERGE($x[2 \ldots k], y[1 \ldots \ell]$)
5: **else** return $y[1] \circ$ MERGE($x[1 \ldots k], y[2 \ldots \ell]$).

The Time Analysis

The recurrence relation:

$$T(n) = 2T(n/2) + O(n);$$

By Master Theorem

$$T(n) = O(n \log n).$$

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

every permutation must appear as the label of a leaf.

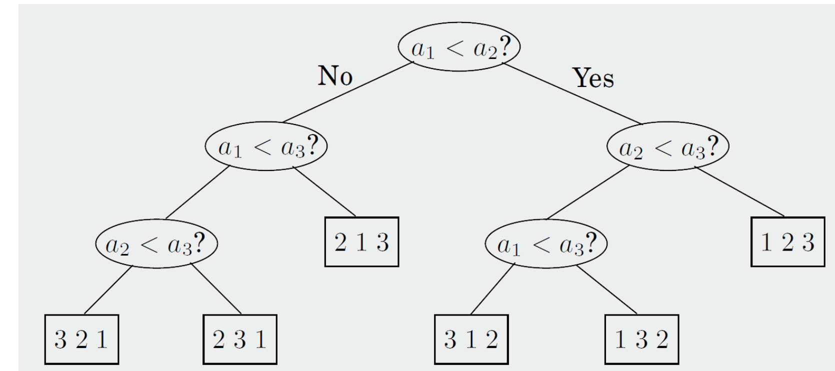
This is a binary tree with $n!$ leaves. Thus, the depth of our tree – and the complexity of our algorithm – must be at least

$$\log(n!) \approx \log\left(\sqrt{\pi\,(2n + 1/3)} \cdot n^n \cdot e^{-n}\right) = \Omega(n \log n),$$

where we use Stirling's formula.

A Sorting Permutation Tree

An example sorts for $\{a_1, a_2, a_3\}$:



Median

The **median** of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller.

If the list has *even length*, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value.

Computing the median of n numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something *linear*.

We have reason to be hopeful, because sorting is doing far more work than we really need – we just want the middle element and don't care about the relative ordering of the rest of them.

Selection

Input: A list of numbers S; an integer K.
Output: The kth smallest element of S.

A randomized divide-and-conquer algorithm for selection

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R respectively.

$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

How to choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$|S_L|, |S_R| \approx \frac{|S|}{2}.$$

If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n) = O(n).$$

But this requires picking v to be the median, which is our ultimate goal!

Instead, we follow a much simpler alternative: we pick v randomly from S.

How to choose v? (cont'd)

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + \frac{n}{2} = \Theta(n^2)$$

Best-case scenario: $O(n)$.

Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the average running time lie? Fortunately, it lies very close to the best-case time.

The efficiency analysis

v is good if it lies within the 25th to 75th percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good,

Lemma: On average a fair coin needs to be tossed two times before a "heads" is seen.

Proof: $E :=$ expected number of tosses before head is seen.
We need at least one toss, and it's heads, we're done.
If it's tail (with probability $1/2$), we need to repeat. Hence

$$E = 1 + \frac{1}{2}E,$$

whose solution is $E = 2$

The efficiency analysis (cont'd)

Let $T(n)$ be the **expected running time** on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n) = O(n).$$

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i,j)th entry

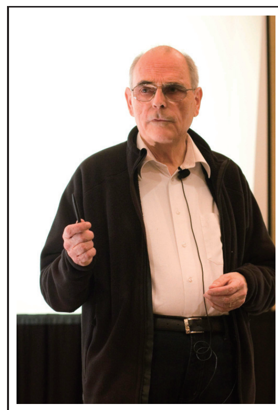
$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

That is, Z_{ij} is the dot product of the ith row of X with the jth column of Y.

In general, XY is not the same as YX; *matrix multiplication is not commutative*.

The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication.

Volker Strassen



Volker Strassen (1936 –)

In 1969, the German mathematician **Volker Strassen** announced a surprising $O(n^{2.81})$ algorithm.

Divide and conquer

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Then

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

To compute the size-n product XY, recursively compute eight size-$n/2$ products $AE, BG, AF, BH, CE, DG, CF, DH$ and then do some $O(n^2)$-time addition.

The recurrence is

$$T(n) = 8T(n/2) + O(n^2)$$

with solution $O(n^3)$.

Strassen's trick

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 = P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F-H) & P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) & & \end{aligned}$$

The recurrence is

$$T(n) = 7T(n/2) + O(n^2)$$

with solution $O(n^{\log_2 7}) \approx O(n^{2.81})$.

Introduction

Previously, we examined sorting algorithms for serial computers (random-access machines, RAM's) that allow only one operation to be executed at a time.

In this section, we investigate sorting algorithms based on a comparison-network model of computation, in which many comparison operations can be performed simultaneously.

Comparison Network **VS** RAM's

- Comparison network can only perform comparisons. (Cannot deal with Counting Sort etc)
- Comparison network run parallel operations.

Definition

A comparison network is composed solely of wires and comparators.

A comparator is a device with two inputs, x and y, and two outputs, x' and y', that performs the following function:

$$x' = \min\{x, y\}, \quad y' = \max\{x, y\}.$$

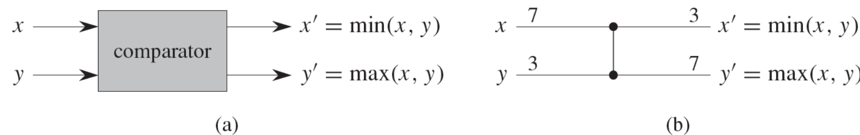


Figure 27.1 **(a)** A comparator with inputs x and y and outputs x' and y'. **(b)** The same comparator, drawn as a single vertical line. Inputs $x = 7$, $y = 3$ and outputs $x' = 3$, $y' = 7$ are shown.

Each comparator operates in $O(1)$ time.

Wire

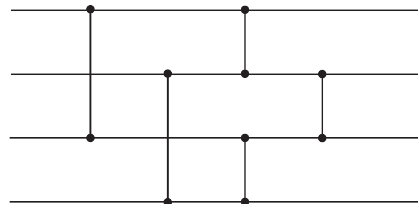
A wire transmits a value from place to place.

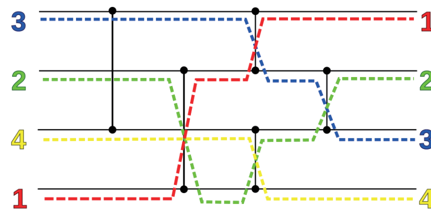
- Connect the output of one comparator to the input of another;
- The network input wires or output wires.

Assume a comparison networks contains n input wires $< a_1, a_2, \cdots, a_n >$, through which the values to be sorted enter the network, and n output wires $< b_1, b_2, \cdots, b_n >$, which produce the results computed by the network.

Draw a comparison network on n inputs as a collection of n horizontal lines with comparators stretched vertically.

An Example





Data move from left to right.

Interconnections must be acyclic.

If a comparator has two input wires with depths d_x and d_y, then its output wire have depth $\max\{d_x, d_y\} + 1$. (Initially is 0)

Sorting Network

A sorting network is a comparison network for which the output sequence is monotonically increasing ($b_1 \leq b_2 \leq \cdots \leq b_n$) for **every** input sequence.

We are discussing a family of comparison networks according to the input size.

Zero-One Principle

The zero-one principle says that if a sorting network works correctly when each input is drawn from the set $\{0,1\}$, then it works correctly on arbitrary input numbers (e.g., integers, reals, or any linearly ordered set).

This principle allow us to focus on the operations for input sequences consisting solely of 0's and 1's.

Once we have constructed a sorting network and proved that it can sort all zero-one sequence, we shall appeal to 0-1 principle to prove its correctness on arbitrary values.

Note: the proof of 0-1 principle relies on the notion of monotonically increasing function.

Lemma

Lemma: If a comparison network transforms the input sequence $\mathbf{a} = < a_1, a_2, \cdots, a_n >$ into the output sequence $\mathbf{b} = < b_1, b_2, \cdots, b_n >$, then for any monotonically increasing function f, the network transforms the input sequence $f(\mathbf{a}) = < f(a_1), f(a_2), \cdots, f(a_n) >$ into the output sequence $f(\mathbf{b}) = < f(b_1), f(b_2), \cdots, f(b_n) >$.

Proof (by Induction)

Consider a comparator whose input values are x and y. The upper output is $\min\{x,y\}$ while the lower output is $\max\{x,y\}$.

If we apply $f(x)$ and $f(y)$ as the inputs, the operation of the comparator yields the value of upper $\min\{f(x),f(y)\}$ and lower $\max\{f(x),f(y)\}$.

Since f is monotonically increasing, $x \leq y$ implies $f(x) \leq f(y)$. Thus we have

$$\min\{f(x),f(y)\} = f(\min\{x,y\}),$$

$$\max\{f(x),f(y)\} = f(\max\{x,y\}),$$

which completes the proof of the claim as the base case.

Proof (Continued)

We use induction on the depth of each wire in a general comparison network to prove a stronger result than the statement of the lemma:

If a wire assumes the value a_i when the input sequence is $\mathbf{a}$, then it assumes the value $f(a_i)$ when the input sequence is $f(\mathbf{a})$.

Since the output wires are included in this statement, proving it will prove the lemma.

Proof (Continued)

Basis: A wire at depth 0 is an input wire a_i. When $f(\mathbf{a})$ is applied to the network, the input wire carries $f(a_i)$.

Induction: A wire at depth $d \geq 1$ is the output of a comparator at depth d, and the input wires to this comparator are at a depth strictly less than d. By inductive hypothesis, if the input wires carry values a_i and a_j with input sequence $\mathbf{a}$, then they carry $f(a_i)$ and $f(a_j)$ with input sequence $f(\mathbf{a})$.

By previous claim, the output wires of this comparator then carry $f(\min\{a_i, a_j\})$ and $f(\max\{a_i, a_j\})$. Since the carry $\min\{a_i, a_j\}$ and $\max\{a_i, a_j\}$ when the input sequence is $\mathbf{a}$, the lemma is proved.

An Example

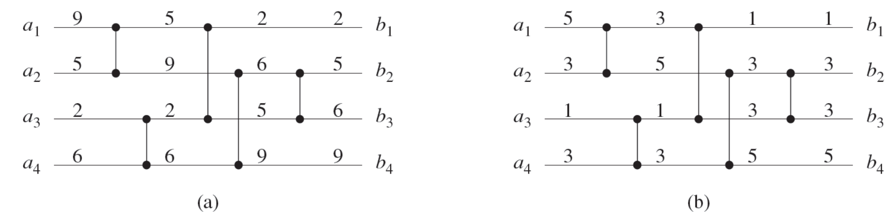


Figure 27.5 **(a)** The sorting network from Figure 27.2 with input sequence $\langle 9, 5, 2, 6\rangle$. **(b)** The same sorting network with the monotonically increasing function $f(x) = \lceil x/2 \rceil$ applied to the inputs. Each wire in this network has the value of f applied to the value on the corresponding wire in (a).

Zero-One Principle

Theorem: If a comparison network with n inputs sorts all 2^n possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof: (Contradiction) Suppose there exists a sequence of arbitrary numbers that the network does not correctly sort. That is , there exists an input sequence $< a_1, a_2, \cdots, a_n >$ containing elements a_i and a_j, such that $a_i < a_j$, but the network places a_j before a_i in the output sequence.

Proof (Continued)

Define a monotonically increasing function f as

$$f(x) = \begin{cases} 0 & \text{if } x \le a_i, \\ 1 & \text{if } x > a_i. \end{cases}$$

Since the network places a_j before a_i, by previous lemma, it will place $f(a_j)$ before $f(a_i)$ in the output sequence when $< f(a_1), f(a_2), \cdots, f(a_n) >$ is input.

However, since $f(a_j) = 1$ and $f(a_i) = 0$, the network fails to sort the zero-one sequence $< f(a_1), f(a_2), \cdots, f(a_n) >$ correctly.

A contradiction!

Construction of a Sorting Network

To construct a sorting network, we need three steps:

Step 1: Construct a Bitonic Sorter $\Rightarrow$ to sort bitonic sequence.

Step 2: Construct a Merger $\Rightarrow$ to merge two sorted sequence.

Step 3: Construct a Sorter $\Rightarrow$ to sort an arbitrary sequence.

Step 1: Construct a Bitonic Sorter

We start from Bitonic sequence.

A Bitonic Sequence is a sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Examples: <1,4,6,8,3,2>, <6,9,4,2,3,5>, <9,8,3,2,4,6>

Half-Cleaner

A half-cleaner is a comparison network of depth 1, in which input line i is compared with line $i + \frac{n}{2}$ for $i = 1, 2, \cdots, \frac{n}{2}$ (assume n is even).

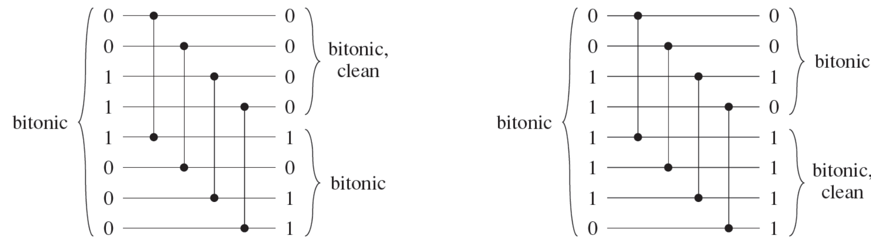


Figure 27.7 The comparison network HALF-CLEANER[8]. Two different sample zero-one input and output values are shown. The input is assumed to be bitonic. A half-cleaner ensures that every output element of the top half is at least as small as every output element of the bottom half. Moreover, both halves are bitonic, and at least one half is clean.

Half-Cleaner (2)

When a bitonic sequence of 0's and 1's is applied as input to a half-cleaner, it produces an output sequence with smaller values in the top-half, and larger values in the bottom-half. Both halves are bitonic.

In fact, at least one of the halves is clean, say, consists of either all 0's or all 1's.

Half-Cleaner Lemma

Lemma: If the input to a half-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the following properties:

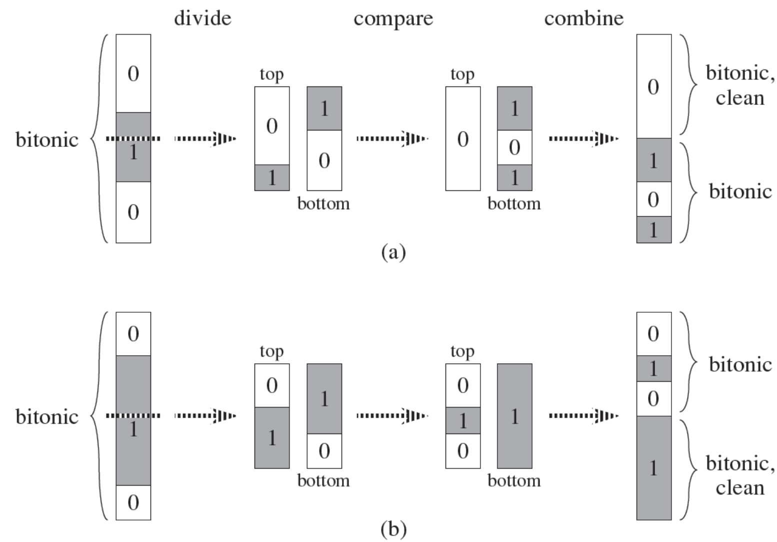
- both the top half and the bottom half are bitonic;
- every element in the top half is at least as small as every element of the bottom half, and at least one half is clean.

Proof

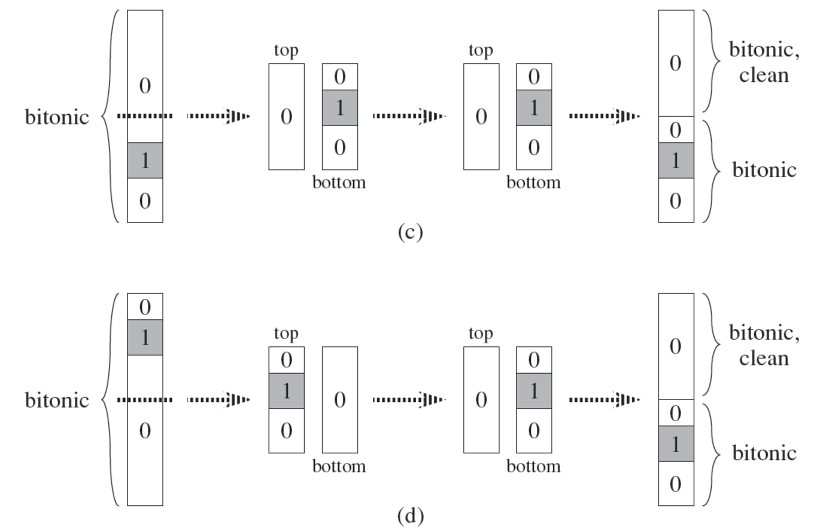
The comparison network HALF-CLEANER$[n]$ compares inputs i and $i + n/2$ for $i = 1, 2, \cdots, n/2$. Without loss of generality, suppose that the input is of the form $00\cdots011\cdots100\cdots0$ (the situation of $11\cdots100\cdots011\cdots1$ are symmetric).

There are three possible cases depending upon the block of consecutive 0's and 1's in which the midpoint $n/2$ falls, and one of these cases is further split into two cases. In each of the cases, the lemma holds.

Case Analysis

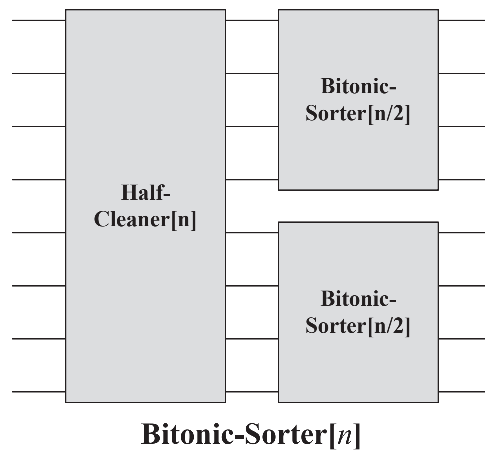


Case Analysis (Cont.)

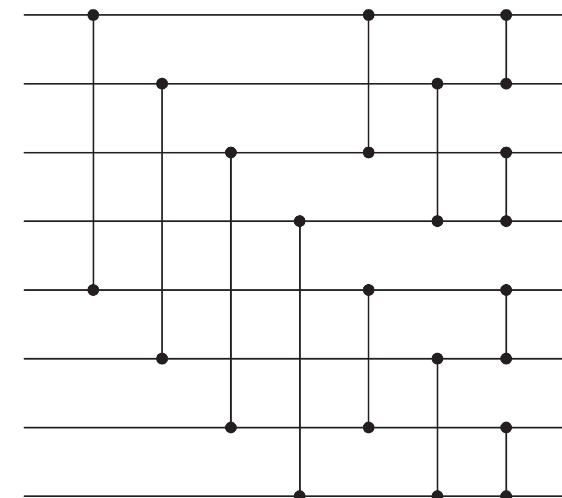


The Bitonic Sorter

By recursively combing half-cleaners, we can build a bitonic sorter, which is a network that sorts bitonic sequences.



An Example of $n = 8$



Depth $D(n)$

The depth $D(n)$ of BITONIC-SORTER$[n]$ is given by the recurrence

$$D(n) = \begin{cases} 0 & \text{if } n = 1; \\ D(n/2) + 1 & \text{if } n = 2^k \text{ and } k \geq 1, \end{cases}$$

Easy to see, $D(n) = \ln n$.

Thus, a zero-one bitonic sequence can be sorted by BITONIC-SORTER$[n]$, which has a depth of $\ln n$.

By zero-one principle, any bitonic sequence of arbitrary numbers can be sorted by this network.

Step 2: Construct a Merger

Merging Network can merge two sorted input sequences into one sorted output sequence.

Given two sorted sequences, if we reverse the order of the second sequence and then concatenate the two sequences, the resulting sequence is bitonic.

For instance:

$$\begin{aligned} X &= 00000111; \\ Y &= 00001111; \\ Y^R &= 11110000; \\ X \circ Y^R &= 0000011111110000. \end{aligned}$$

MERGER$[n]$

Given two sorted sequences $< a_1, a_2, \cdots, a_{n/2} >$ and $< a_{n/2+1}, a_{n/2+2}, \cdots, a_n >$, we want the effect of bitonically sorting the sequence $< a_1, a_2, \cdots, a_{n/2}, a_n, a_{n-1}, \cdots, a_{n/2+1} >$.

Since the first half-cleaner of BITONIC-SORTER$[n]$ compares inputs i with $n/2 + i$, for $i = 1, 2, \cdots, n/2$, we make the first stage of the merging network compare inputs i and $n - i + 1$.

The order of the outputs from the bottom of the first stage of MERGER$[n]$ are reversed compared with the order of outputs from an ordinary half-cleaner.

Comparison between MERGER$[n]$ and HALF-CLEANER$[n]$

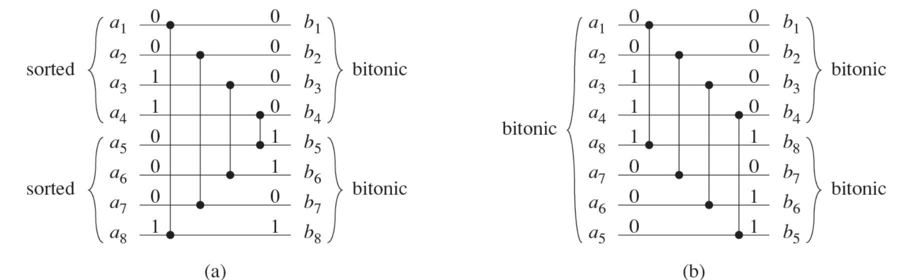
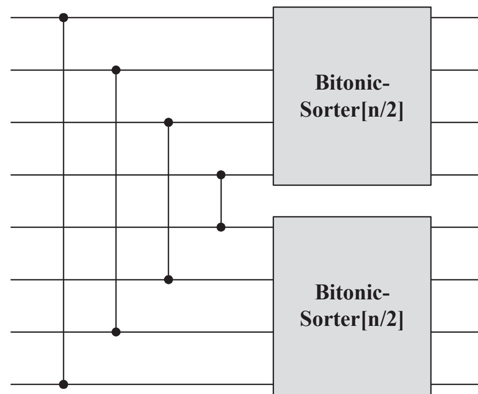


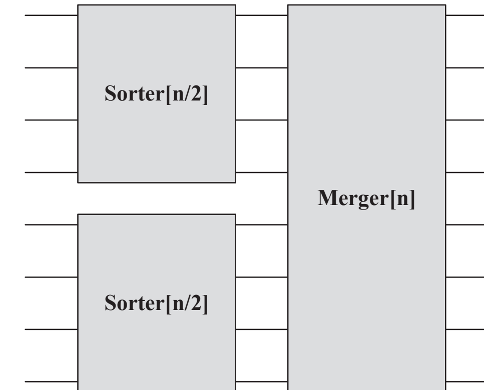
Figure 27.10 Comparing the first stage of MERGER$[n]$ with HALF-CLEANER$[n]$, for $n = 8$. **(a)** The first stage of MERGER$[n]$ transforms the two monotonic input sequences $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ and $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ into two bitonic sequences $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ and $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$. **(b)** The equivalent operation for HALF-CLEANER$[n]$. The bitonic input sequence $\langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1}\rangle$ is transformed into the two bitonic sequences $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ and $\langle b_n, b_{n-1}, \ldots, b_{n/2+1}\rangle$.

MERGER[n]

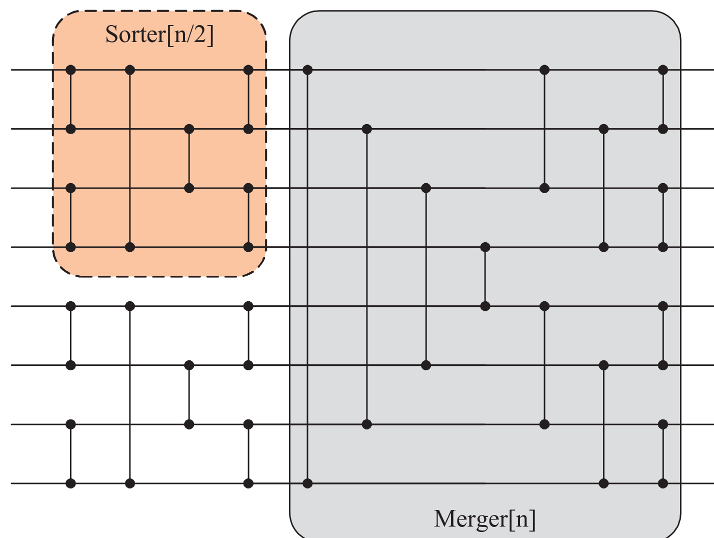


Step 3: Construct a Sorter

The sorting network SORTER[n] are composed by two copies of SORTER[$n/2$] and one MERGER[n] recursively.



An Example with $n = 8$



Performance Analysis

The depth $D(n)$ of SORTER[n] is the depth $D(n/2)$ of SORTER[$n/2$] plus the depth $\ln n$ of MERGER[n].

Consequently, the depth of SORTER[n] is given by the recurrence

$$D(n) = \begin{cases} 0 & \text{if } n = 1; \\ D(n/2) + \ln n & \text{if } k \geq 1. \end{cases}$$

By Master's Theorem, the solution is $D(n) = \Theta(\ln^2 n)$. Thus we can sort n numbers in parallel in $O(\ln^2 n)$ time.