DATA STRUCTURES AND ALGORITHMS

An *algorithm* is any technique used to solve a given problem. The problem could be that of rearranging a given sequence of numbers, solving a system of linear equations, finding the shortest path between two nodes in a graph, and so on. An algorithm consists of a sequence of basic operations, such as addition, multiplication, comparison, and so on and is typically described in a machine-independent manner. When an algorithm is coded in a specified programming language, such as C, C++, or Java, it becomes a *program* that can be executed on a computer.

For any given problem, there could be many different techniques that solve it. Thus it becomes necessary to define performance measures to judge different algorithms. Two popular measures are *time complexity* and *space complexity.*

The *time complexity* or the *run time* of an algorithm is the total number of basic operations performed in the algorithm. As an example, consider the problem of finding the minimum of n given numbers. This is accomplished by using $(n - 1)$ comparisons. Of the two measures perhaps time complexity is more important. This measure is useful for the following reasons: (1) We can use the time complexity of an algorithm to predict its actual run time when it is coded in a programming language and run on a specific machine. (2) Given several different algorithms for solving the same problem, we can use their run times to identify the best one.

The *space complexity* of an algorithm is defined as the amount of space (i.e., the number of memory cells) used by the algorithm. This measure is critical especially when the input data are huge.

We define the *input size* of a problem instance as the amount of space needed to specify the instance. For the problem of finding the minimum of n numbers, the input size is n because we need n memory cells, one for each number, to specify the problem instance. For the problem of multiplying two $(n \times n)$ matrices, the input size is $2n^2$ because that many elements are in the input. Both the run time and the space complexity of an algorithm are expressed as functions of the input size.

For any given problem instance, its input size alone may not be enough to decide its time complexity. To illustrate this point, consider the problem of checking if an element x is in an array $a[1:n]$. This is called the *searching problem.* One way of solving this problem is to check if $x = a[1]$. If not check if $x = a[2]$, and so on. This algorithm may terminate after the first comparison, after the second comparison, . . ., or after comparing x with every element in $a[\ ]$. Thus it is necessary to qualify the time complexity as the *best case,* the *worst case,* the *average case,* etc. The *average-case* run time of an algorithm is the average run time taken over all possible inputs (of a given size).

Analysis of an algorithm is simplified using asymptotic functions, such as $O(.)$, $\Omega(.)$, and so on. Let $f(n)$ and $g(n)$ be nonnegative integral functions of n. We say $f(n)$ is $O[g(n)]$ if $f(n) \leq c\, g(n)$ for all $n \geq n_0$, where c and n_0 are some constants. Also, $f(n) = \Omega[g(n)]$ if $f(n) \geq c\, g(n)$ for all $n \geq n_0$, for some constants c and n_0. If $f(n) = O[g(n)]$ and $f(n) = \Omega[g(n)]$, then $f(n) = \Theta[g(n)]$. Usually we express the run times (or the space complexities) of algorithms using $\Theta(\ )$. The algorithm for finding the minimum of n given numbers takes $\Theta(n)$ time.

An algorithm designer is faced with the task of developing the best possible algorithm (typically an algorithm whose run time is the best possible) for any given problem. Unfortunately, there is no standard recipe for doing this. Algorithm researchers have identified a number of useful techniques, such as the divide-and-conquer, dynamic programming, greedy, backtracking, and branch-and-bound. Application of any one or a combination of these techniques by itself may not guarantee the best possible run time. Some innovations (small and large) may have to be discovered and incorporated.

Note that all logarithms used in this article are to the base 2, unless otherwise mentioned.

DATA STRUCTURES

An algorithm can be thought of as a mapping from the input data to the output data. A data structure refers to the way the data are organized. Often the choice of the data structure determines the efficiency of the algorithm using it. Thus the study of data structures plays an essential part in algorithmic design.

Examples of basic data structures include queues, stacks, etc. More advanced data structures are based on *trees.* Any data structure supports certain operations on the data. We can classify data structures depending on the operations supported. A *dictionary* supports `Insert`, `Delete`, and `Search` operations. On the other hand a *priority queue* supports `Insert`, `Delete-Min`, and `Find-Min` operations. The operation *Insert* is to insert an arbitrary element into the data structure. `Delete` is the operation of deleting a specified element. `Search` takes an element x as input and decides if x is in the data structure. `Delete-Min` deletes and returns the minimum element from the data structure. `Find-Min` returns the minimum element from the data structure.

Queues and Stacks

In a *queue,* two operations are supported, namely, `insert` and `delete`. The operation `insert` is supposed to insert a given element into the data structure. On the other hand, `delete` deletes the first element inserted into the data structure. Thus a queue employs the first in, first out policy. A *stack* also supports `insert` and `delete` operations but uses the last in, first out policy.

A queue or a stack is implemented easily by using an array of size n, where n is the maximum number of elements that is ever stored in the data structure. In this case an `insert` or a `delete` is performed in $O(1)$ time. We can also implement stacks and queues by using linked lists. Even then the operations take only $O(1)$ time.

We can also implement a dictionary or a priority queue using an array or a linked list. For example consider the implementation of a dictionary using an array. At any given time, if there are n elements in the data structure, these ele-

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering.*

ments are stored in $a[1:n]$. If x is a given element to be `Insert`ed, it is stored in $a[n + 1]$. To `Search` for a given x, we scan through the elements of $a[\ ]$ until we either find a match or realize the absence of x. In the worst case this operation takes $O(n)$ time. To `Delete` the element x, we first `Search` for it in $a[\ ]$. If x is not in $a[\ ]$, we report so and quit. On the other hand, if $a[i] = x$, we move the elements $a[i + 1]$, $a[i + 2]$, . . ., $a[n]$ one position to the left. Thus the `Delete` operation takes $O(n)$ time.

It is also easy to see that a priority queue is realized by using an array such that each of the three operations takes $O(n)$ time. The same is also done by using a linked list.

Binary Search Trees

We can implement a dictionary or a priority queue in time better than that offered by queues and stacks with the help of *binary trees* that have certain properties.

A *binary tree* is a set of nodes that is either empty or has a node called the *root* and two disjoint binary trees. These trees are called the left and right subtrees, respectively. The root of the left subtree is called the left child of the root. The right child of the root is also defined similarly. We store some data at each node of a binary tree. Figure 1 shows examples of binary trees.

Each node has a label associated with it. We might use the data stored at any node itself as its label. For example, in Fig. 1(a), 5 is the root. Eight is the right child of 5 and so on. In Fig. 1(b), 11 is the root. Five is the left child of 11. The subtree containing the nodes 5, 12, and 8 is the left subtree of 11, etc. We can also define *parent* relationship in the usual manner. For example, in the tree of Fig. 1(a), 5 is the parent of 8, 8 is the parent of 3, and so on. A tree node is called a *leaf* if it does not have any children. Nine is a leaf in the tree of Fig. 1(a). The nodes 8 and 9 are leaves in the tree of Fig. 1(b).

The *level* of the root is defined as 1. The level of any other node is defined as $(\ell + 1)$, where ℓ is the level of its parent. In the tree of Fig. 1(b), the level of 3 and 5 is 2, the level of 12 and 1 is 3, and the level of 8 and 9 is 4. The *height* of a tree is defined as the maximum level of any node in the tree. The trees of Fig. 1 have a height of 4.

A *binary search tree* is a binary tree such that the data (or key) stored at any node are greater than any key in its left subtree and smaller than any key in its right subtree. Trees in Fig. 1 are not binary search trees because, for example, in the tree of Fig. 1(a), the right subtree of node 8 has a key 3 that is smaller than 8. Figure 2 shows an example of a binary search tree.

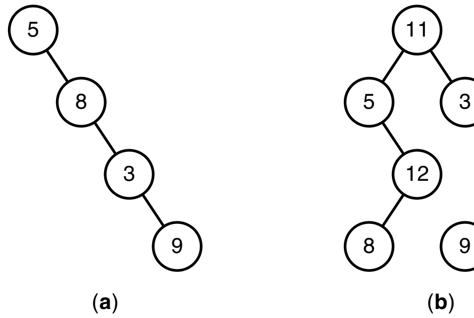

Figure 1. Examples of binary trees.

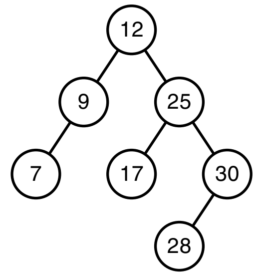

Figure 2. Examples of a binary search tree.

We can verify that the tree of Fig. 2 is a binary search tree by considering each node of the tree and its subtrees. For the node 12, the keys in its left subtree are 9 and 7 which are smaller. Keys in its right subtree are 25, 17, 30, and 28 which are all greater than 12. Node 25 has 17 in its left subtree and 30 and 28 in its right subtree, and so on.

We can implement both a dictionary and a priority queue using binary search trees. Now we illustrate how to perform the following operations on a binary search tree: `Insert`, `Delete`, `Search`, `Find-Min`, and `Delete-Min`.

To `Search` for a given element x, we compare x with the key at the root y. If $x = y$, we are done. If $x < y$, then if x is in the tree at all, it has to be in the left subtree. On the other hand, if $x > y$, x can only be in the right subtree, if at all. Thus after making one comparison, the searching problem reduces to searching either the left or the right subtree, i.e., the search space reduces to a tree of height one less. Thus the total time taken by this search algorithm is $O(h)$, where h is the height of the tree.

To `Insert` a given element x into a binary search tree, we first search for x in the tree. If x is already in the tree, we can quit. If not, the search terminates in a leaf y such that x can be inserted as a child of y. Look at the binary search tree of Fig. 2. Say we want to insert 19. The `Search` algorithm begins by comparing 19 with 12 realizing that it should proceed to the right subtree. Next 19 and 25 are compared to note that the search should proceed to the left subtree. Next 17 and 19 are compared to realize that the search should move to the right subtree. But the right subtree is empty. This is where the `Search` algorithm terminates. The node 17 is y. We can insert 19 as the right child of 17. Thus we see that we can also process the `Insert` operation in $O(h)$ time.

A `Delete` operation can also be processed in $O(h)$ time. Let the element to be deleted be x. First we `Search` for x. If x is not in the tree, we quit. If not, the `Search` algorithm returns the node in which x is stored. There are three cases to consider. (1) The node x is a leaf. This is an easy case. We just delete x and quit. (2) The node x has only one child y. Let z be the parent of x. We make z the parent of y and delete x. In Fig. 2, if we want to delete 9, we can make 12 the parent of 7 and delete 9. (3) The node x has two children. There are two ways to handle this case. The first is to find the largest key y from the left subtree. Replace the contents of node x with y, and delete node y. Note that the node y can have one child at most. In the tree of Fig. 2, say, we desire to delete 25. The largest key in the left subtree is 17 (there is only one node in the left subtree). We replace 25 with 17 and delete node 17 which happens to be a leaf. The second way to handle this case is to identify the smallest key z in the right subtree of x, replace x with z, and delete node z. In either case, the algorithm takes time $O(h)$.

The operation `Find-Min` can be performed as follows. We start from the root and always go to the left child until we cannot go any further. The key of the last visited node is the minimum. In the tree of Fig. 2, we start from 12, go to 9, and then go to 7. We realize that 7 is the minimum. This operation also takes $O(h)$ time.

We can process `Delete-Min` using `Find-Min` and `Delete`, and hence this operation also takes $O(h)$ time.

If we have a binary search tree with n nodes in it, how large can h become? The value of h can be as large as n. Consider a tree whose root has the value 1, its right child has a value 2, the right child of 2 is 3, and so on. This tree has a height n. Thus we realize that in the worst case even the binary search tree may not be better than an array or a linked list. But fortunately, it has been shown that the expected height of a binary search tree with n nodes is only $O(\log n)$. This is based on the assumption that each permutation of the n elements is equally likely to be the order in which the elements are inserted into the tree. Thus we arrive at the following theorem.

Theorem 1. Both the dictionary and the priority queue can be implemented by using a binary search tree so that each of the underlying operations takes only an expected $O(\log n)$ time. In the worst case, the operations might take $O(n)$ time each.

It is easy to see that any binary tree with n nodes has to have a height of $\Omega(\log n)$. There are a number of other schemes based on binary trees which ensure that the height of the tree does not become very large. These schemes maintain a tree height of $O(\log n)$ at any time and are called *balanced tree schemes.* Examples include red–black trees, AVL trees, 2–3 trees, etc. These schemes achieve a worst case run time of $O(\log n)$ for each of the operations of our interest. We state the following theorem without proof.

Theorem 2. A dictionary and a priority queue can be implemented so that each of the underlying operations takes only $O(\log n)$ time in the worst case.

Theorem 2 has been used to derive several efficient algorithms for differing problems. We illustrate just one example. Consider the problem of sorting. Given a sequence of n numbers, the problem of sorting is to rearrange this sequence in nondecreasing order. This comparison problem has attracted the attention of numerous algorithm designers because of its applicability in many walks of life. We can use a priority queue to sort. Let the priority queue be empty to begin with. We insert the input keys one at a time into the priority queue. This involves n invocations of the `Insert` operation and hence takes a total of $O(n \log n)$ time (see Theorem 2). Followed by this we apply `Delete-Min` n times to read out the keys in sorted order. This also takes another $O(n \log n)$ time. Thus we have an $O(n \log n)$-time sorting algorithm.

ALGORITHMS FOR SOME BASIC PROBLEMS

In this section we deal with some basic problems such as matrix multiplication, binary search, etc.

Matrix Multiplication

Matrix multiplication plays a vital role in many areas of science and engineering. Given two $(n \times n)$ matrices A and B, the problem is to compute $C = AB$. By definition, $C[i, j] = \sum_{k=1}^{n} A[i, k] * B[k, j]$. Using this definition, each element of C can be computed in $\Theta(n)$ time and because there are n^2 elements to compute, C can be computed in $\Theta(n^3)$ time. This algorithm can be specified as follows:

for i := **to** n **do**
  for j := **to** n **do**
    $C[i, j] := 0;$
    for k := 1 **to** n **do**
      $C[i, j] := C[i, j] + A[i, k] * B[k, j];$

One of the most popular techniques for developing (both sequential and parallel) algorithms is *divide and conquer.* The idea is to partition the given problem into k (for some $k \geq 1$) subproblems, solve each subproblem, and combine these partial solutions to arrive at a solution to the original problem. It is natural to describe any algorithm based on divide and conquer as a *recursive* algorithm (i.e., an algorithm that calls itself). The run time of the algorithm is expressed as a *recurrence relationship* which upon solution indicates the run time as a function of the input size.

Strassen has developed an elegant algorithm based on the divide-and-conquer technique that multiplies two $(n \times n)$ matrices in $\Theta(n^{\log_2 7})$ time. This algorithm is based on the critical observation that two (2×2) scalar matrices can be multiplied using only seven scalar multiplications (and 18 additions—the asymptotic run time of the algorithm is oblivious to this number). Partition A and B into submatrices of size $(n/2 \times n/2)$ each as shown:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$
$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Now use the formulas developed by Strassen to multiply two (2×2) scalar matrices. Here there are also seven multiplications, but each multiplication involves two $(n/2 \times n/2)$ submatrices. These multiplications are performed recursively. There are also 18 additions [of $(n/2 \times n/2)$ submatrices]. Because two $(m \times m)$ matrices can be added in $\Theta(m^2)$ time, all of these 18 additions need only $\Theta(n^2)$ time.

If $T(n)$ is the time taken by this divide-and-conquer algorithm to multiply two $(n \times n)$ matrices, then $T(n)$ satisfies

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

whose solution is $T(n) = \Theta(n^{\log_2 7})$.

Coppersmith and Winograd proposed an algorithm that takes only $O(n^{2.376})$ time. This is a complex algorithm details of which can be found in the references at the end of this article.

Binary Search

Let $a[1:n]$ be a given array whose elements are in nondecreasing order, and let x be another element. The problem is to check if x is a member of $a[\ ]$. A simple divide-and-conquer algorithm can also be designed for this problem.

The idea is first to check if $x = a[n/2]$. If so, the problem has been solved. If not, the search space reduces by a factor of 2 because if $x > a[n/2]$, then x can be only in the second half of the array, if at all. Likewise, if $x < a[n/2]$, then x can be only in the first half of the array, if at all. If $T(n)$ is the number of comparisons made by this algorithm on any input of size n, then $T(n)$ satisfies $T(n) = T(n/2) + 1$, which reduces to $T(n) = \Theta(\log n)$.

SORTING

Several optimal algorithms have been developed for sorting. We have already seen one such algorithm in the section on Binary Search Trees that employs priority queues. We assume that the elements to be sorted are from a *linear order.* If no other assumptions are made about the keys to be sorted, the sorting problem is called *general sorting* or *comparison sorting.* In this section we consider general sorting and sorting with additional assumptions.

General Sorting

We look at two general sorting algorithms. The first algorithm is called the *selection sort.* Let the input numbers be in the array $a[1:n]$. First we find the minimum of these n numbers by scanning through them. This takes $(n - 1)$ comparisons. Let this minimum be in $a[i]$. We exchange $a[1]$ and $a[i]$. Next we find the minimum of $a[2:n]$ by using $(n - 2)$ comparisons, and so on.

The total number of comparisons made in the algorithm is $(n - 1) + (n - 2) + \cdots + 2 + 1 = \Theta(n^2)$.

An asymptotically better algorithm is obtained using divide and conquer. This algorithm is called the *merge sort.* If the input numbers are in $a[1:n]$, we divide the input into two halves, namely, $a[1:n/2]$ and $a[n/2 + 1:n]$. Sort each half recursively, and finally *merge* the two sorted subsequences. The problem of *merging* is to take two sorted sequences as input and produce a sorted sequence of all the elements of the two sequences. We can show that two sorted sequences of length l and m, respectively, can be merged in $\Theta(l + m)$ time. Therefore, the two sorted halves of the array $a[\ ]$ can be merged in $\Theta(n)$ time.

If $T(n)$ is the time taken by the merge sort on any input of size n, then $T(n) = 2T(n/2) + \Theta(n)$, which reduces to $T(n) = \Theta(n \log n)$.

Now we show how to merge two given sorted sequences with l and m elements, respectively. Let $X = q_1, q_2, \ldots, q_l$ and $Y = r_1, r_2, \ldots, r_m$ be the sorted (in nondecreasing order) sequences to be merged. Compare q_1 and r_1. Clearly, the minimum of q_1 and r_1 is also the minimum of X and Y combined. Output this minimum, and delete it from the sequence from which it came. Generally, at any given time, compare the current minimum element of X with the current minimum of Y, output the minimum of these two, and delete the output element from its sequence. Proceed this way until one of the sequences becomes empty. At this time, output all the elements of the remaining sequence (in order).

Whenever the above algorithm makes a comparison, it outputs one element (either from X or from Y). Thus it follows that the algorithm cannot make more than $(l + m - 1)$ comparisons.

Theorem 3. We can sort n elements in $\Theta(n \log n)$ time.

It is easy to show that any general sorting algorithm has to make $\Omega(n \log n)$ comparisons, and hence the merge sort is asymptotically optimal.

Integer Sorting

We can perform sorting in time better than $\Omega(n \log n)$ by making additional assumptions about the keys to be sorted. In particular, we assume that the keys are integers in the range $[1, n^c]$, for any constant c. This version of sorting is called *integer sorting.* In this case, sorting can be done in $\Theta(n)$ time.

We begin by showing that n integers in the range $[1, m]$ can be sorted in time $\Theta(n + m)$ for any integer m. We use an array $a[1:m]$ of m lists, one for each possible value that a key can have. These lists are empty to begin with. Let $X = k_1, k_2, \ldots, k_n$ be the input sequence. We look at each input key and put it in an appropriate list of $a[\ ]$. In particular, we append key k_i to the end of list $a[k_i]$ for $i = 1, 2, \ldots, n$. This takes $\Theta(n)$ time. Basically we have grouped the keys according to their values.

Next, we output the keys of list $a[1]$, the keys of list $a[2]$, and so on. This takes $\Theta(m + n)$ time. Thus the whole algorithm runs in time $\Theta(m + n)$.

If one uses this algorithm (called the *bucket sort*) to sort n integers in the range $[1, n^c]$ for $c > 1$, the run time is $\Theta(n^c)$. This may not be acceptable because we can do better using the merge sort.

We can sort n integers in the range $[1, n^c]$ in $\Theta(n)$ time by using the bucket sort and the notion of *radix sorting.* Say we are interested in sorting n two-digit numbers. One way of doing this is to sort the numbers with respect to their least significant digits and then to sort with respect to their most significant digits. This approach works provided the algorithm used to sort the numbers with respect to a digit is *stable.* We say a sorting algorithm is *stable* if equal keys remain in the same relative order in the output as they were in the input. Note that the bucket sort previously described is stable.

If the input integers are in the range $[1, n^c]$, we can think of each key as a $c \log n$-bit binary number. We can conceive of an algorithm where there are $\lceil c \rceil$ *stages.* In stage i, the numbers are sorted with respect to their ith most significant $\log n$ bits. This means that in each stage we have to sort n $\log n$-bit numbers, that is, we have to sort n integers in the range $[1, n]$. If we use the bucket sort in every stage, the stage takes $\Theta(n)$ time. Because there are only a constant number of stages, the total run time of the algorithm is $\Theta(n)$. We get the following theorem.

Theorem 4. We can sort n integers in the range $[1, n^c]$ in $\Theta(n)$ time for any constant c.

SELECTION

In this section we consider the problem of selection. We are given a sequence of n numbers, and we are supposed to identify the ith smallest number from these for a specified i, $1 \leq i \leq n$. For example, if $i = 1$, we are interested in finding the smallest number. If $i = n$, we are interested in finding the largest element.

A simple algorithm for this problem could pick any input element k, partition the input into two—the first part is those input elements less than x and the second part consists of input elements greater than x—identify the part that contains the element to be selected, and finally recursively perform an appropriate selection in the part containing the element of interest. This algorithm has an expected (i.e., average-case) run time of $O(n)$. Generally the run time of any divide-and-conquer algorithm is the best if the sizes of the subproblems are as even as possible. In this simple selection algorithm, it may happen that one of the two parts is empty at each level of recursion. The second part may have $(n - 1)$ elements. If $T(n)$ is the run time corresponding to this input, then $T(n) = T(n - 1) + \Omega(n)$. This reduces to $T(n) = \Omega(n^2)$. In fact if the input elements are already in sorted order and we always pick the first element of the array as the partitioning element, then the run time is $\Omega(n^2)$.

So, even though this simple algorithm has a good average-case run time, in the worst case it can be bad. We are better off using the merge sort. It is possible to design an algorithm that selects in $\Theta(n)$ time in the worst case, as has been shown by Blum, Floyd, Pratt, Rivest, and Tarjan.

Their algorithm employs a primitive form of "deterministic sampling." Say we are given n numbers. We group these numbers so that there are five numbers in each group. Find the median of each group. Find also the median M of these group medians. We can expect M to be an "approximate median" of the n numbers.

For simplicity assume that the input numbers are distinct. The median of each group is found in $\Theta(1)$ time, and hence all the medians (except M) are found in $\Theta(n)$ time. Having found M, we partition the input into two parts X_1 and X_2. X_1 consists of all the input elements less than M, and X_2 contains all the elements greater than M. This partitioning can also be done in $\Theta(n)$ time. We can also count the number of elements in X_1 and X_2 within the same time. If $|X_1| = i - 1$, then clearly M is the element to be selected. If $|X_1| \geq i$, then the element to be selected belongs to X_1. On the other hand, if $|X_1| < i - 1$, then the ith smallest element of the input belongs to X_2.

It is easy to see that the size of X_2 can be at most $\frac{7}{10}n$. This can be argued as follows: Let the input be partitioned into the groups $G_1, G_2, \ldots, G_{n/5}$ with five elements in each part. Assume without loss of generality that every group has exactly five elements. There are $n/10$ groups such that their medians are less than M. In each such group there are at least three elements that are less than M. Therefore, there are at least $\frac{3}{10}n$ input elements that are less than M. In turn, this means that the size of X_2 is at most $\frac{7}{10}n$. Similarly, we can also show that the size of X_1 is no more than $\frac{7}{10}n$.

Thus we can complete the selection algorithm by performing an appropriate selection in either X_1 or X_2, recursively, depending on whether the element to be selected is in X_1 or X_2, respectively.

Let $T(n)$ be the run time of this algorithm on any input of size n and for any i. Then it takes $T(n/5)$ time to identify the median of medians M. Recursive selection on X_1 or X_2 takes no more than $T(7/10n)$ time. The rest of the computations account for $\Theta(n)$ time. Thus $T(n)$ satisfies

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + \Theta(n)$$

which reduces to $T(n) = \Theta(n)$. This can be proved by induction.

Theorem 5. Selection from out of n elements can be performed in $\Theta(n)$ time.

RANDOMIZED ALGORITHMS

The performance of an algorithm may not be completely specified even when the input size is known, as has been pointed out before. Three different measures can be conceived of: the best case, the worst case, and the average case. Typically, the average-case run time of an algorithm is much smaller than the worst case. For example, Hoare's quicksort has a worst case run time of $O(n^2)$, whereas its average-case run time is only $O(n \log n)$. While computing the average-case run time, one assumes a distribution (e.g., uniform distribution) on the set of possible inputs. If this distribution assumption does not hold, then the average-case analysis may not be valid.

Is it possible to achieve the average-case run time without making any assumptions about the input space? Randomized algorithms answer this question in the affirmative. They make no assumptions on the inputs. The analysis of randomized algorithms is valid for all possible inputs. Randomized algorithms obtain such performance by introducing randomness into the algorithms themselves.

Coin flips are made for certain decisions in randomized algorithms. A randomized algorithm with one possible sequence of outcomes for coin flips can be thought of as different from the same algorithm with a different sequence of outcomes for coin flips. Thus a randomized algorithm can be viewed as a family of algorithms. Some of the algorithms in this family might have 'poor performance' with a given input. It should be ensured that, for any input, the number of algorithms in the family that performs poorly with this input is only a small fraction of the total number of algorithms. If we can find at least a $(1 - \epsilon)$ (ϵ is very close to 0) portion of algorithms in the family that have 'good performance' with any given input, then clearly, a random algorithm in the family will have 'good performance' with any input with probability $\geq (1 - \epsilon)$. In this case, we say that this family of algorithms (or this randomized algorithm) has 'good performance' with probability $\geq (1 - \epsilon)$. ϵ is called the *error probability* which is independent of the input distribution.

We can interpret 'good performance' in many different ways. Good performance could mean that the algorithm outputs the correct answer or that its run time is small, and so on. Different types of randomized algorithms can be conceived of depending on the interpretation. A *Las Vegas* algorithm is a randomized algorithm that always outputs the correct answer but whose run time is a random variable (possibly with a small mean). A *Monte Carlo* algorithm is a randomized algorithm that has a predetermined run time but whose output may be incorrect occasionally.

We can modify asymptotic functions such as $O(.)$ and $\Omega(.)$ in the context of randomized algorithms as follows: A randomized algorithm is said to use $\tilde{O}[f(n)]$ amount of resources (like time, space, etc.) if a constant c exists such that the amount of resources used is no more than $c\alpha f(n)$ with probability $\geq 1 - n^{-\alpha}$ on any input of size n and for any positive $\alpha \geq 1$. Similarly, we can also define $\tilde{\Omega}[f(n)]$ and $\tilde{\Theta}[f(n)]$. If n is the input size of the problem under consideration, then, by *high probability* we mean a probability of $\geq 1 - n^{-\alpha}$ for any fixed $\alpha \geq 1$.

Illustrative Examples

We provide two examples of randomized algorithms. The first is a Las Vegas algorithm, and the second is a Monte Carlo algorithm.

Example 1 [Repeated Element Identification]**.** The input is an array a[ ] of n elements wherein there are $(n - \epsilon n)$ distinct elements and ϵn copies of another element, where ϵ is a constant >0 and <1. The problem is to identify the repeated element. Assume without loss of generality that ϵn is an integer.

Any deterministic algorithm to solve this problem must take at least $(\epsilon n + 2)$ time in the worst case. This fact can be proven as follows: Let the input be chosen by an adversary who has perfect knowledge about the algorithm used. The adversary can make sure that the first $(\epsilon n + 1)$ elements examined by the algorithm are all distinct. Therefore, the algorithm may not be in a position to output the repeated element even after having examined $(\epsilon n + 1)$ elements. In other words, the algorithm must examine at least one more element, and hence the claim follows.

We can design a simple $O(n)$ time deterministic algorithm for this problem. Partition the elements such that each part (except possibly one part) has $(\lceil 1/\epsilon \rceil + 1)$ elements. Then search the individual parts for the repeated element. Clearly, at least one of the parts will have at least two copies of the repeated element. This algorithm runs in time $\Theta(n)$.

Now we present a simple and elegant Las Vegas algorithm that takes only $\tilde{O}(\log n)$ time. This algorithm is comprised of *stages.* Two random numbers i and j are picked from the range $[1, n]$ in any stage. These numbers are picked independently with replacement. As a result, there is a chance that these two are the same. After picking i and j, we check if $i \neq j$ and $a[i] = a[j]$. If so, the repeated element has been found. If not, the next stage is entered. We repeat the stages as many times as it takes to arrive at the correct answer.

Lemma 6. The previous algorithm runs in time $\tilde{O}(\log n)$.

Proof. The probability of finding the repeated element in any given stage is given by $P = \epsilon n(\epsilon n - 1)/n^2 \approx \epsilon^2$. Thus the probability that the algorithm does not find the repeated element in the first $c\alpha \log_e n$ (c is a constant to be fixed) stages is expressed as

$$< (1 - \epsilon^2)^{c\alpha \log_e n} \leq n^{-\epsilon^2 c\alpha}$$

using the fact that $(1 - x)^{1/x} \leq 1/e$ for any $0 < x < 1$. This probability is $< n^{-\alpha}$ if we pick $c \geq 1/\epsilon^2$, that is, the algorithm takes no more than $1/\epsilon^2\alpha \log_e n$ stages with probability $\geq 1 - n^{-\alpha}$. Because each stage takes $O(1)$ time, the run time of the algorithm is $\tilde{O}(\log n)$.

Example 2 [Large Element Selection]**.** Here also the input is an array $a[\ ]$ of n numbers. The problem is to find an element of the array that is greater than the median. We can assume, without loss of generality, that the array numbers are distinct and that n is even.

Lemma 7. The preceding problem can be solved in $O(\log n)$ time by using a Monte Carlo algorithm.

Proof. Let the input be $X = k_1, k_2, \ldots, k_n$. We pick a random sample S of size $c\alpha \log n$ from X. This sample is picked with replacement. Find and output the maximum element of S. The claim is that the output of this algorithm is correct with high probability.

The algorithm gives an incorrect answer only if all the elements in S have a value $\leq M$, where M is the median. The probability that any element in S is $\leq M$ is 1/2. Therefore, the probability that all the elements of S are $\leq M$ is given by $P = (1/2)^{c\alpha \log n} = n^{-c\alpha}$. $P \leq n^{-\alpha}$ if c is picked to be ≥ 1.

In other words, if the sample S has $\geq \alpha \log n$ elements, then the maximum of S is a correct answer with probability $\geq (1 - n^{-\alpha})$.

PARALLEL COMPUTING

One of the ways of solving a given problem quickly is to employ more than one processor. The basic idea of parallel computing is to partition the given problem into several subproblems, assign a subproblem to each processor, and combine the partial solutions obtained by the individual processors.

If P processors are used to solve a problem, then there is a potential of reducing the run time by a factor of up to P. If S is the best known *sequential run time* (i.e., the run time using a single processor), and if T is the parallel run time using P processors, then $PT \geq S$. If not, we can simulate the parallel algorithm by using a single processor and get a run time better than S (which is a contradiction). PT is called the *work done* by the parallel algorithm. A parallel algorithm is said to be *work-optimal* if $PT = O(S)$. We provide a brief introduction to parallel algorithms in the next section.

Parallel Models

The random access machine (RAM) model has been widely accepted as a reasonable sequential model of computing. In the RAM model, we assume that each of the basic, scalar, binary operations, such as addition, multiplication, etc. takes one unit of time. We have assumed this model in our discussion thus far. In contrast, many well-accepted parallel models of computing exist. In any such parallel model an individual processor can still be thought of as a RAM. Variations among different architectures arise in the ways they implement interprocessor communications. In this article we categorize parallel models into *shared-memory models* and *fixed-connection machines.*

A shared-memory model [also called the parallel random access machine (PRAM)] is a collection of RAMs working in synchrony which communicate with the help of a common block of global memory. If processor i has to communicate with processor j, it can do so by writing a message in memory cell j which then is read by processor j.

Conflicts for global memory access can arise. Depending on how these conflicts are resolved, a PRAM can further be classified into three categories. An exclusive read and exclusive write (EREW) PRAM does not permit concurrent reads or concurrent writes. A concurrent read and exclusive write (CREW) PRAM allows concurrent reads but not concurrent writes. A concurrent read and concurrent write (CRCW) PRAM permits both concurrent reads and concurrent writes. For a CRCW PRAM, we need an additional mechanism for handling write conflicts because the processors trying to write at the same time in the same cell may have different data to write and a decision has to be made as to which data are written. Concurrent reads do not pose such problems because the data read by different processors are the same. In a common-CRCW PRAM, concurrent writes are allowed only if the processors that try to access the same cell have the same data to write. In an arbitrary-CRCW PRAM, if more than one processor tries to write in the same cell at the same time, arbitrarily, one of them succeeds. In a priority-CRCW PRAM, write conflicts are resolved by using priorities assigned to the processors.

A fixed-connection machine can be represented as a directed graph whose nodes represent processors and whose edges represent communication links. If there is an edge connecting two processors, they communicate in one unit of time. If two processors not connected by an edge want to communicate, they do so by sending a message along a path that connects the two processors. We can think of each processor in a fixed-connection machine as a RAM. Examples of fixed-connection machines are the mesh, the hypercube, the star graph, etc. Our discussion on parallel algorithms is confined to PRAMs because of their simplicity.

Boolean Operations

The first problem considered is that of computing the Boolean OR of n given bits. With n common-CRCW PRAM processors, we compute the Boolean OR in $O(1)$ time as follows. The input bits are stored in common memory (one bit per cell). Every processor is assigned an input bit. We employ a common memory cell M that is initialized to zero. All the processors that have ones try to write a one in M in one parallel write step. The result is ready in M after this write step. Using a similar algorithm, we can also compute the Boolean AND of n bits in $O(1)$ time.

Lemma 8. The Boolean OR or Boolean AND of n given bits can be computed in $O(1)$ time using n Common-CRCW PRAM processors.

The different versions of the PRAM form a hierarchy in terms of their computing power. EREW PRAM, CREW PRAM, common-CRCW PRAM, arbitrary-CRCW PRAM, priority-CRCW PRAM is an ordering of some of the PRAM versions. Any model in the sequence is strictly less powerful than any to its right and strictly more powerful than any to its left. As a result, for example, any algorithm that runs on the EREW PRAM runs on the common-CRCW PRAM and preserves the processor and time bounds, but the converse may not be true.

Finding the Maximum

Now we consider the problem of finding the maximum of n given numbers. We describe an algorithm that solves this problem in $O(1)$ time using n^2 common-CRCW PRAM processors.

Partition the processors so that there are n processors in each group. Let the input be $k_1, k_2, \ldots, k_n$, and let the groups be $G_1, G_2, \ldots, G_n$. Group i is assigned the key k_i. G_i is in charge of checking if k_i is the maximum. In one parallel step, processors of group G_i compare k_i with every input key. In particular, processor j of group G_i computes the bit $b_{ij} = k_i \geq k_j$. The bits $b_{i1}, b_{i2}, \ldots, b_{in}$ are ANDed using the algorithm of Lemma 8. This is done in $O(1)$ time. If G_i computes a one in this step, then one of the processors in G_i outputs k_i as the answer.

Lemma 9. The maximum (or minimum) of n given numbers can be computed in $O(1)$ time using n^2 common-CRCW PRAM processors.

Prefix Computation

Prefix computation plays a vital role in designing parallel algorithms. This is as basic as any arithmetic operation in sequential computing. Let $\oplus$ be any associative unit-time computable binary operator defined in some domain Σ. Given a sequence of n elements $k_1, k_2, \ldots, k_n$ from Σ, the problem of *prefix computation* is to compute $k_1, k_1 \oplus k_2, k_1 \oplus k_2 \oplus k_3, \ldots, k_1 \oplus k_2 \oplus \cdots \oplus k_n$. Examples of $\oplus$ are addition, multiplication, and min. Example of Σ are the set of integers, the set of reals, etc. The *prefix sums computation* refers to the special case when $\oplus$ is addition. The results themselves are called *prefix sums*.

Lemma 10. We can perform prefix computation on a sequence of n elements in $O(\log n)$ time using n CREW PRAM processors.

Proof. We can use the following algorithm. If $n = 1$, the problem is solved easily. If not, the input elements are partitioned into two halves. Solve the prefix computation problem on each half recursively assigning $n/2$ processors to each half. Let $y_1, y_2, \ldots, y_{n/2}$ and $y_{n/2+1}, y_{n/2+2}, \ldots, y_n$ be the prefix values of the two halves.

There is no need to modify the values $y_1, y_2, \ldots$, and $y_{n/2}$, and hence they can be output as such. Prefix values from the second half can be modified as $y_{n/2} \oplus y_{n/2+1}, y_{n/2} \oplus y_{n/2+2}, \ldots, y_{n/2} \oplus y_n$. This modification is done in $O(1)$ time by using $n/2$ processors. These $n/2$ processors first read $y_{n/2}$ concurrently and then update the second half (one element per processor).

Let $T(n)$ be the time needed to perform prefix computation on n elements by using n processors. $T(n)$ satisfies $T(n) = T(n/2) + O(1)$, which reduces to $T(n) = O(\log n)$.

The processor bound of the preceding algorithm is reduced to $n/\log n$ as follows: Each processor is assigned $\log n$ input elements. (1) Each processor computes the prefix values of its $\log n$ elements in $O(\log n)$ time. Let $x_1^i, x_2^i, \ldots, x_{\log n}^i$ be the elements assigned to processor i. Also let $X_i = x_1^i \oplus x_2^i \oplus \cdots$

$\oplus\, x^i_{\log n}$. (2) Now the $n/\log n$ processors perform a prefix computation on $X_1, X_2, \ldots, X_{n/\log n}$, using the algorithm of Lemma 10. This takes $O(\log n)$ time. (3) Each processor modifies the $\log n$ prefixes that it computed in step (1) using the result of step (2). This also takes $O(\log n)$ time.

Lemma 11. Prefix computation on a sequence of length n can be performed in $O(\log n)$ time by using $n/\log n$ CREW PRAM processors.

Realize that the preceding algorithm is work-optimal. In all of the parallel algorithms we have seen so far, we have assumed that the number of processors is a function of the input size. But the machines available in the market may not have these many processors. Fortunately, we can simulate these algorithms on a parallel machine with a fewer number of processors and preserve the asymptotic work done.

Let $\mathcal{A}$ be an algorithm that solves a given problem in time T by using P processors. We can simulate every step of $\mathcal{A}$ on a P'-processor (with $P' \leq P$) machine in time $\leq \lceil P/P' \rceil$. Therefore, the simulation of $\mathcal{A}$ on the P'-processor machine takes a total time of $\leq T\lceil P/P' \rceil$. The total work done by the P'-processor machine is $\leq P'T\lceil P/P' \rceil \leq PT + P'T = O(PT)$.

Lemma 12 [The Slow-Down Lemma]**.** We can simulate any PRAM algorithm that runs in time T by using P processors on a P'-processor machine in time $O(PT/P')$ for any $P' \leq P$.

BIBLIOGRAPHIC NOTES

There are several excellent texts on data structures. A few of these are by Horowitz, Sahni, and Mehta (1); Kingston (2); Weiss (3); and Wood (4). A discussion on standard data structures such as red-black trees can be found in algorithm texts also. For example, see the text by Cormen, Leiserson, and Rivest (5).

There are also numerous wonderful texts on algorithms. Here we list only a small group: Aho, Hopcroft, and Ullman (6); Horowitz, Sahni, and Rajasekaran (7,8); Cormen, Leiserson, and Rivest (5); Sedgewick (9); Manber (10); Baase (11); Brassard and Bratley (12); Moret and Shapiro (13); Rawlins (14); Smith (15); Nievergelt and Hinrichs (16); and Berman and Paul (17).

The technique of randomization was popularized by Rabin (18). One of the problems considered in Ref. 18 was primality testing. In an independent work at around the same time, Solovay and Strassen (19) presented a randomized algorithm for primality testing. The idea of randomization itself had been employed in Monte Carlo simulations a long time before. The sorting algorithm of Frazer and McKellar (20) is also one of the early works on randomization.

Randomization has been employed in the sequential and parallel solution of numerous fundamental problems of computing. Several texts cover randomized algorithms at length. A partial list is Horowitz, Sahni, and Rajasekaran (7,8), Já Já (21); Leighton (22); Motwani and Raghavan (23); Mulmuley (24); and Reif (25).

The texts of Refs. 7, 8, 21, 22, and 25 cover parallel algorithms. For a survey of sorting and selection algorithms over a variety of parallel models, see Ref. 26.

ACKNOWLEDGMENTS

This work is supported in part by an NSF Award CCR-95-03-007 and an EPA Grant R-825-293-01.0.

BIBLIOGRAPHY

1. E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++,* San Francisco: Freeman, 1995.
2. J. H. Kingston, *Algorithms and Data Structures,* Reading, MA: Addison-Wesley, 1990.
3. M. A. Weiss, *Data Structures and Algorithm Analysis,* Menlo Park, CA: Benjamin/Cummings, 1992.
4. D. Wood, *Data Structures, Algorithms, and Performance,* Reading, MA: Addison-Wesley, 1993.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms,* Cambridge, MA: MIT Press, 1990.
6. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Reading, MA: Addison-Wesley, 1974.
7. E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms,* San Francisco: Freeman, 1998.
8. E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms / C++,* San Francisco: Freeman, 1997.
9. R. Sedgewick, *Algorithms,* Reading, MA: Addison-Wesley, 1988.
10. U. Manber, *Introduction to Algorithms: A Creative Approach,* Reading, MA: Addison-Wesley, 1989.
11. S. Baase, *Computer Algorithms,* Reading, MA: Addison-Wesley, 1988.
12. G. Brassard and P. Bratley, *Fundamentals of Algorithms,* Upper Saddle River, NJ: Prentice-Hall, 1996.
13. B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP,* Menlo Park, CA: Benjamin/Cummings, 1991.
14. G. J. E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms,* San Francisco: Freeman, 1992.
15. J. D. Smith, *Design and Analysis of Algorithms,* PWS-KENT, 1989.
16. J. Nievergelt and K. H. Hinrichs, *Algorithms and Data Structures,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
17. K. A. Berman and J. L. Paul, *Fundamentals of Sequential and Parallel Algorithms,* Boston: PWS, 1997.
18. M. O. Rabin, Probabilistic Algorithms, in J. F. Traub (ed.), *Algorithms and Complexity,* New York: Academic Press, 1976, pp. 21–36.
19. R. Solovay and V. Strassen, A Fast Monte-Carlo Test for Primality, *SIAM J. Comput.,* **6**: 84–85, 1977.
20. W. D. Frazer and A. C. McKellar, Samplesort: A Sampling Approach to Minimal Storage Tree Sorting, *J. ACM,* **17** (3): 496–502, 1977.
21. J. Já Já, *Parallel Algorithms: Design and Analysis,* Reading, MA: Addison-Wesley, 1992.
22. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes,* San Mateo, CA: Morgan-Kaufmann, 1992.
23. R. Motwani and P. Raghavan, *Randomized Algorithms,* Cambridge, UK: Cambridge Univ. Press, 1995.
24. K. Mulmuley, *Computational Geometry: An Introduction Through Randomized Algorithms,* Englewood Cliffs, NJ: Prentice-Hall, 1994.
25. J. H. Reif (ed.), *Synthesis of Parallel Algorithms,* San Mateo, CA: Morgan-Kaufmann, 1992.

26. S. Rajasekaran, Sorting and Selection on Interconnection Networks, *DIMACS Series Discrete Math. Theoretical Comput. Sci.,* **21**: 275–296, 1995.

27. D. E. Knuth, *The Art of Computer Programming,* vol. 3, *Sorting and Searching,* Reading, MA: Addison-Wesley, 1973.

PANOS M. PARDALOS
University of Florida
SANGUTHEVAR RAJASEKARAN
University of Florida

DATA TRANSMISSION CODES. See INFORMATION THEORY OF DATA TRANSMISSION CODES.