Divide and Conquer Algorithms

Introduction

There exist many problems that can be solved using a divide-and-conquer algorithm. A divide-and-conquer algorithm $A$ follows these general guidelines.

**Divide** Algorithm $A$ divides original problem into one or more subproblems of a smaller size.

**Conquer** Each subproblem is solved by making a recursive call to $A$.

**Combine** Finally, $A$ combines the subproblem solutions into a final solution to the original problem.

Some problems that can be solved using a divide-and-conquer algorithm:

**Binary Search** locating an element in a sorted array

**Quicksort and Mergesort** sorting an array

**Order Statistics** finding the $k$th least or greatest element of an array

**Geometric Algorithms** finding the convex hull of a set of points; finding two points that are closest.

**Matrix Operations** matrix inversion, Fast-Fourier Transform, matrix multiplication, finding the largest submatrix of 1’s in a Boolean matrix.

**Maximum Subsequence Sum** finding the maximum sum of any subsequence in a sequence of integers.

**Minimum Positive Subsequence Sum** finding the minimum positive sum of any subsequence in a sequence of integers.
Multiplication of Binary Numbers finding the product of two binary numbers

From an analysis-of-algorithms perspective, the more interesting part of the analysis is often found in establishing the algorithm’s running time. Usually this involves determining the big-O growth of a function \( T(n) \) that satisfies a divide-and-conquer recurrence. Hence, the techniques from the previous lecture prove quite useful. Some algorithms require a degree of mathematical proof, but the proofs usually seem more palpable than those required for, say, greedy algorithms. Quite often the correctness of the algorithm seems clear from its description. As for implementation, most divide-and-conquer algorithms act on arrays of numbers, matrices, or points in space, and do not require any special data structures.

Mergesort

The Mergesort algorithm is a divide-and-conquer algorithm for sorting an array of comparable elements. The algorithm begins by checking if input array \( a \) has two or fewer elements. If so, then \( a \) is sorted in place by making at most one swap. Otherwise, \( a \) is divided into two (almost) equal halves \( a_{\text{left}} \) and \( a_{\text{right}} \). Both of these subarrays are sorted by making recursive calls to Mergesort. Once sorted, a merge operation merges the elements of \( a_{\text{left}} \) and \( a_{\text{right}} \) into an auxiliary array. This sorted auxiliary array is then copied over to the original array.
Example 1. Demonstrate the Mergesort algorithm using the array 5, 8, 6, 2, 7, 1, 0, 9, 3, 4, 6.
Letting $T(n)$ denote the running time of Mergesort on an array of size $n$, we see that $T(n)$ satisfies the uniform recurrence

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

The first term is due to the fact that the algorithm makes two recursive calls on subproblems of size $n/2$, while the second term $n$ is a (big-O) bound on the time it takes to merge $a_{\text{left}}$ and $a_{\text{right}}$. Therefore, by Case 2 of the Master theorem, $T(n) = \Theta(n \log n)$.

**Quicksort**

Hoare’s Quicksort

Before introducing the Quicksort algorithm, recall that the median of an array $a$ of $n$ numbers $a[0], \ldots, a[n-1]$ is the $(n+1)/2$ least element of $a$, if $n$ is odd, and is equal to either the $n/2$ or $n/2 + 1$ least element of $a$ if $n$ is even (even-length arrays have two medians).
Example 2. Determine the median of 7, 5, 7, 3, 4, 8, 2, 3, 7, 8, 2, and the medians of 4, 5, 10, 12, 6, 3.

\[
\text{median} \ (a) = 5 \\
\text{median} \ (b) = \frac{5}{2}, 6
\]

**Quicksort** is considered in practice to be the most efficient sorting algorithm for arrays of data stored in local memory. Quicksort is similar to Mergesort in that the first (non base case) step is to divide the input array \( a \) into two arrays \( a_{\text{left}} \) and \( a_{\text{right}} \). However, where as Mergesort simply divides \( a \) into two equal halves, Quicksort performs the **Partitioning Algorithm** on \( a \) which is described below.
Partitioning Algorithm

**Calculate Pivot** The pivot $M$ is an element of $a$ which is used to divide $a$ into $a_{\text{left}}$ and $a_{\text{right}}$. Namely all elements $x \in a_{\text{left}}$ satisfy $x \leq M$, while all elements $x \in a_{\text{right}}$ satisfy $x \geq M$. A common heuristic for computing $M$ is called **median-of-three**, where $M$ is chosen as the median of the first, last, and middle elements of $a$; i.e. $\text{median}(a[0], a[(n-1)/2], a[n-1])$.

**Swap Pivot** Swap the pivot element with the last element of $a$.

**Initialize Markers** Initialize a left marker to point to $a[0]$. Initialize a right marker to point to $a[n-2]$. Let $i = 0$ denote the current index location of the left marker, and $j = n-2$ denote the current index location of the right marker.

**Examine Markers** Execute one of the following cases.

- If $i \geq j$, then swap $a[i]$ with $M = a[n-1]$. In this case $a_{\text{left}}$ consists of the first $i$ elements of $a$, while $a_{\text{right}}$ consists of the last $n-i-1$ elements of $a$. Thus, $a[i] = M$ is to the right of $a_{\text{left}}$ and to the left of $a_{\text{right}}$.
- Else if $a[i] \geq M$ and $a[j] \leq M$, then swap $a[i]$ with $a[j]$, increment $i$, and decrement $j$.
- Else increment $i$ if $a[i] < M$ and/or decrement $j$ if $a[j] > M$

**Repeat** Re-examine markers until $i \geq j$.

Once the Partitioning algorithm has partitioned $a$ into $a_{\text{left}}$ and $a_{\text{right}}$, then Quicksort is recursively called on both these arrays, and the algorithm is complete.

Notice how Quicksort and Mergesort differ, in that Mergesort performs $O(1)$ steps in partitioning $a$, but $\Theta(n)$ steps to combine the sorted subarrays, while Quicksort performs $\Theta(n)$ steps to partition $a$, and requires no work to combine the sorted arrays. Moreover, Quicksort has the advantage of sorting “in place”, meaning that no additional memory is required outside of the input array. Indeed, the Partitioning algorithm only requires swapping elements in the original array, and, the sorting of each subarray only uses that part of $a$ where the elements of the subarray are located. For example, if $a_{\text{left}}$ occupies locations 0 through 10 of $a$, then only those locations will be affected when Quicksort is called on input $a_{\text{left}}$. It is this in-place property that gives Quicksort an advantage over Mergesort.
Example 3. Demonstrate the quicksort algorithm using the array 5, 8, 6, 2, 7, 1, 0, 9, 3, 4, 6.

\[ \text{Pivot} = \text{median}(A[6], A[8], A[5]) = \text{median}(5, 6, 1) = 5 \]

\[ \begin{array}{ccccccccccc}
4 & 6 & 3 & 8 & 2 & 7 & 1 & 5 & 6 & 9 & 3 & 8 & 4 & 6 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\text{l1} & \text{l2} & \text{l3} & \text{r1} & \text{r2} & \text{r3} & \text{r4} & \text{r5} & \text{r6} & \text{aright} \\
\end{array} \]

\[ a_{\text{left}} = \begin{array}{ccccccc}
4 & 3 & 0 & 2 & 1 & 5 & 6 & 9 & 8 & 6 & 7 \\
\end{array} \]

\[ M = \text{median}(4, 0) = 0 \]

\[ a_{\text{left}} = \begin{array}{ccccccc}
\text{0} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\text{l1} & \text{l2} & \text{l3} & \text{r1} & \text{r2} & \text{r3} & \text{r4} & \text{r5} & \text{r6} & \text{aright} \\
\end{array} \]

\[ a_{\text{left}} = \begin{array}{ccccccc}
\text{0} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\text{l1} & \text{l2} & \text{l3} & \text{r1} & \text{r2} & \text{r3} & \text{r4} & \text{r5} & \text{r6} & \text{aright} \\
\end{array} \]
Running time of Quicksort. The running time (i.e. number of steps $T(n)$ for an array size of $n$ comparables) of quicksort depends on how the pivot is chosen. Later in this lecture we demonstrate how to find an exact median in $O(n)$ steps. This algorithm could in theory be applied to finding the Quicksort pivot. Using this approach quicksort has a running time of $\Theta(n \log n)$, since $T(n)$ satisfies the recurrence

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

However, in practice the pivot is chosen at random, or by using a heuristic such as median-of-three. Although both options offer a worst case running time of $O(n^2)$, in practice they outperform the approach that computes the median as the pivot. The worst case is $O(n^2)$ because, e.g., either approach could result in a sequence of pivots for which $a_{\text{left}}$ always has a length equal to one. In this case the lengths of each $a_{\text{right}}$ subarray are respectively, $n - 2, n - 4, n - 6, \ldots$ down to either 1 or 2. And since the Partition algorithm must be performed on each of these arrays, it yields a total running time of at least (assuming $n$ is odd)

$$T(n) = \Omega(n + (n - 2) + (n - 4) + \cdots + 1) = \Omega(\sum_{i=1}^{(n+1)/2} 2i) = \Omega(n^2).$$
Finding Order Statistics

The \( k \) th order statistic of an array \( a \) of \( n \) elements is the \( k \) th least element in the array, \( k = 0, \ldots, n - 1 \). Moreover, finding the \( k \) th order statistic of \( a \) can be accomplished by sorting \( a \) and returning the \( k \) th element in the sorted array. Using Mergesort, this will take \( \Theta(n \log n) \) steps. We now describe an algorithm that is similar to Quicksort and reduces the running time for finding the \( k \) th statistic down to \( O(n) \) steps.

For the moment assume that we have access to an oracle that can provide the median of an array \( a \) at a computational cost of one step. Also, assume that, in addition to \( a \) and \( k \), our algorithm has two additional inputs, \texttt{lower} and \texttt{upper}, that respectively give lower and upper bounds on the index of \( a \) where the \( k \) th statistic is located. Thus, our algorithm/function has the following signature (here we assume an array of integers).

\[
\text{int find_statistic(int a[], int k, int lower, int upper)}
\]

For example, if \( a \) has a length of \( n \), then the initial call would be

\[
\text{find_statistic(a,k,0,n-1)}
\]
Below is an implementation of `find_statistic`.

```c
// Returns the kth statistic of a which is located at an index i for which i >= lower and i <= upper
int find_statistic(int a[], int k, int lower, int upper)
{
    // Assume base case of 5 or fewer elements
    if(upper-lower <= 4)
        return find_statistic_base_case(a,k,lower,upper)

    // The oracle returns the index of where a's median is located
    int index = oracle(a,lower,upper)
    int M = a[index]

    // partition_algorithm returns the index of the final pivot location
    index = partition_algorithm(a,lower,upper,index)

    if(k == index) // kth least element equals the pivot
        return M

    if(k < index)
        return find_statistic(a,k,lower,index-1)

    // Must have k > index
    return find_statistic(a,k,index+1,upper)
}
```
Letting $T(n)$ denote the running time of $\text{find\_statistic}$, we see that $T(n)$ satisfies

$$T(n) \leq T(n/2) + n.$$ 

Indeed, the oracle call counts for one step, the Partition algorithm counts for $n$ steps (assuming $n = \text{upper} - \text{lower} + 1$), and the recursive call (assuming $k \neq \text{index}$) contributes another $T(n/2)$ steps. Thus, by Case 3 of the Master theorem, we see that $T(n) = O(n)$.

Now all that remains is to replace the oracle function. As a first attempt, since the $k = n/2$ statistic is a median of $a$, we could replace the function call

$$\text{oracle}(a, \text{lower}, \text{upper})$$

with

$$\text{find\_statistic}(a, n/2, \text{lower}, \text{upper})$$

The problem here is that the input to this recursive call does not represent a smaller subproblem. In other words, if $n = \text{upper} - \text{lower} + 1$ is the size of the original problem, then $n$ is also the size of the subproblem, since neither has the $\text{lower}$ value been increased, nor has the $\text{upper}$ value been decreased. As a result, if $n > 5$, then the base case will never be attained, and the function’s execution will result in an infinite loop.

Thus, when computing the median, we must reduce the problem size by finding the median of only some of the elements of $a$, and yet hope that the answer still provides for a good enough partition. To this end, the elements that will be used are determined as follows. Divide $a$ into $\lceil n/5 \rceil$ groups of five elements (the last group may contain fewer than five). Calculate the median of each group, and place it in the array $a_{\text{medians}}$. This array has a length of $\lceil n/5 \rceil$ elements. Now replace the function call

$$\text{oracle}(a, \text{lower}, \text{upper})$$

with

$$\text{find\_statistic}(a_{\text{medians}}, \frac{1}{2} \lceil n/5 \rceil, 0, \lceil n/5 \rceil - 1).$$

In other words, the new pivot is equal to the median of the medians of each group of five.
Example 4. Demonstrate the (median-of-five) \texttt{find statistic} algorithm using the array

\[
5, 8, 16, 2, 7, 11, 0, 9, 3, 4, 6, 7, 3, 15, 5, 12, 4, 7.
\]

and \(k = 3\).

\[
\begin{array}{cccccccccccccccc}
22 & -23 & 36 & 58 & 72 & 99 & 39 \\
121 & 65 & 91 & 39 & 26 & 97 & 86 & 66 \\
36 & 29 & 85 & 72 & 14 & 83 & 29 \\
\end{array}
\]

Medians: 36, 91, 66, 36, 83.

\texttt{Pivot} = \text{median of medians} = 66.

66 > 36, 36

66 < 91, 83.
Theorem 1. The find statistic algorithm has a worst-case running time of $T(n) = O(n)$.

Proof of Theorem 1. $T(n)$ satisfies the recurrence

$$T(n) \leq T(n/5) + T(bn) + n,$$

where $T(n/5)$ is the cost of finding the median of $a_{\text{medians}}$, $n$ is the cost of the Partitioning algorithm, and $T(bn)$ is the cost of the final recursive call (if necessary). Here, $b$ is a fraction for which $|bn|$ represents the worst-case length of either $a_{\text{left}}$ or $a_{\text{right}}$. Using the oracle, we had $b = 1/2$, since the oracle returned the exact median of $a$ which was used as the partitioning pivot. But now the pivot is determined by the median of $a_{\text{medians}}$.

Claim. The median $M$ of $a_{\text{medians}}$ is greater than or equal to (respectively, less than or equal to) at least

$$3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2\right) \geq 3\left(\frac{1}{2} \cdot \frac{n}{5} - 3\right) = \frac{3n}{10} - 9 \geq \frac{n}{4}$$

elements of $a$, assuming $n \geq 200$.

Proof of Claim. Since $M$ is the median of $a_{\text{medians}}$ it must be greater than or equal to at least

$L = \left\lfloor \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rfloor$ elements of $a_{\text{medians}}$. Moreover, we subtract 2 from $L$ to account for the median $M$ itself, and also the median of the last group, which might not have five elements. Thus, $L - 2$ is the number of elements of $a_{\text{medians}}$ that are distinct from $M$, and come from a group that has five elements, and which are less than or equal to $M$. But, since each of these elements is the median of its group, there must be two additional elements in its group that are also less than or equal to $M$. Hence, there are 3 elements in the group that are less than or equal to $M$, giving a total of

$$3(L - 2) = 3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2\right)$$

elements of $a$ that are less than or equal to $M$.

Furthermore, using the inequalities, $|x| \geq x - 1$ and $[x] \geq x$, we arrive at

$$3(L - 2) \geq 3\left(\frac{1}{2} \cdot \frac{n}{5} - 3\right) = \frac{3n}{10} - 9.$$

Finally, basic algebra shows that the inequality

$$\frac{3n}{10} - 9 \geq \frac{n}{4}$$

is true provided $n \geq 200$. A symmetrical argument may be given for establishing that $M$ is also less than or equal to at least $n/4$ elements of $a$.

To finish the proof of Theorem 1, since there are at least $n/4$ elements to the left and right of $M$, we know that both $a_{\text{left}}$ and $a_{\text{right}}$ cannot have lengths that exceed $n - n/4 = 3n/4$. Thus, $b = 3/4$, and we have

$$T(n) \leq T(n/5) + T(3n/4) + n.$$

Finally, by Exercise 16 from the Recurrence Relations lecture with $a = 1/5$ and $b = 3/4$, we have

$a + b = 1/5 + 3/4 = 19/20 < 1$ which implies that $T(n) = O(n)$.
Minimum-Distance-Pair Problem \[
\binom{n}{2} = \frac{n(n-1)}{2}
\]

Given a set of points \(p_1, \ldots, p_n \in \mathcal{R}^2\), find two distinct points \(p_i\) and \(p_j\) whose distance \(d(p_i, p_j)\) is minimum among all pairs of distinct points. This problem may be easily solved in \(\Theta(n^2)\) steps by finding the distance between all \(\binom{n}{2}\) pairs, and keeping track of the least calculated distance. However, the divide-and-conquer algorithm we present has a running time of \(\Theta(n \log n)\).

The following notation will prove useful. Let \(d(p_1, p_2)\) denote the Euclidean distance between points \(p_1\) and \(p_2\). Also, \(d(P)\) will denote the minimum distance between any two points in a set \(P\) of points. For convenience, we define \(d(\emptyset) = \infty\).

The Minimum-Distance-Pair (MDP) algorithm takes as input an array of points \(p_1, \ldots, p_n \in \mathcal{R}^2\). We assume that these points are presented in two sorted arrays \(X\) and \(Y\), where the \(X\) (respectively, \(Y\)) array is a sorting of the points based on their \(x\)-coordinates (respectively, \(y\)-coordinates). The algorithm begins by dividing the arrays into two equal halves \(X_{\text{left}}\) and \(X_{\text{right}}\), and \(Y_{\text{left}}\) and \(Y_{\text{right}}\). For example, the points in \(X_{\text{left}}\) are those points that fall to the left of the median of the \(x\)-coordinate values, while \(Y_{\text{left}}\) has the same points as \(X_{\text{left}}\), but sorted by the \(y\)-coordinate values. The algorithm then makes two recursive calls, one on \((X_{\text{left}}, Y_{\text{left}})\), the other on \((X_{\text{right}}, Y_{\text{right}})\), to produce the two respective values \(\delta_L = d(X_{\text{left}})\) and \(\delta_R = d(X_{\text{right}})\). Then letting \(\delta = \min(\delta_L, \delta_R)\), we see that any two points that either both lie to the left or to the right of the median line, are at least \(\delta\) apart in distance.

All that remains is to check if there are two points \(p_1\) and \(p_2\) for which \(d(p_1, p_2) < \delta\), where, e.g., \(p_1 = (x_1, y_2)\) lies to the left of the median line, and \(p_2 = (x_2, y_2)\) lies to the right of the line. Let \(x = x_m\) be the equation of the median line. If two such points did exist, then it must be the case that \(x_m - \delta \leq x_1 \leq x_m\) and \(x_m \leq x_2 \leq x_m + \delta\). In other words, both points must lie within the vertical \(\delta\)-strip whose left boundary is \(x = x_m - \delta\), and whose right boundary is \(x = x_m + \delta\). Hence, the next step is to remove all points from \(X\) and \(Y\) that do not lie in this strip. This can be done in linear time.

Finally, the algorithm then considers each point \(p = (x_0, y_0)\) in the \(\delta\)-strip by increasing value of \(y\)-coordinate. In other words, the algorithm now iterates through the \(Y\) array. For such \(p \in Y\), consider the rectangle \(R\) in the \(\delta\)-strip whose bottom side has equation \(y = y_0\), top side has equation \(y = y_0 + \delta\), and whose sides have respective equations \(x = x_m - \delta\) and \(x = x_m + \delta\). Then any point that lies above \(p\) and is within \(\delta\) of \(p\) must lie in \(R\). Moreover \(R\) has dimensions \(2\delta \times \delta\), and thus consists of two \(\delta \times \delta\) squares, \(S_L\) and \(S_R\).

**Claim.** Both \(S_L\) and \(S_R\) can fit at most four points from \(Y\).

**Proof of Claim.** Consider \(S_L\). It is a \(\delta \times \delta\) square that lies to the left of the median line. Therefore, since \(\delta \leq \delta_L\), no two \(Y\) points in \(S_L\) can be less than \(\delta\) in distance from each other. Moreover, the most points that one could place in \(S_L\) under this constraint is four, with one point being placed at each of the corners of \(S_L\). Two see why five points could not be placed, we can apply the pigeon-hole
principle, where the pigeons are points and the holes are each of the four $\delta/2 \times \delta/2$ quadrants that comprise $S_L$. If five $Y$ points were in $S_L$, then, by the pigeon-hole principle, two of points must lie in one of the quadrants. But the diameter of each quadrant (i.e. the length of a quadrant diagonal) equals

$$\sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \delta/\sqrt{2} < \delta,$$

which implies the two points would have to be less than $\delta$ apart, a contradiction. Similarly, $S_R$ can have at most four points from $Y$.

To finish the algorithm, it follows that, since $p$ is in $R$, then at most $4 + 3 = 7$ other $Y$ points can lie in $R$, and have a chance of being less than $\delta$ from $p$. Hence, the algorithm checks the distance between $p$ and the next seven points that follow $p$ in the $Y$ array.

The running time of the MDP algorithm satisfies

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

The first term comes from the two recursive calls on $(X_{\text{left}}, Y_{\text{left}})$ and $(X_{\text{right}}, Y_{\text{right}})$, with each input having size $n/2$. The second term $n$ comes from the fact that both the filtering of the $Y$ array to include only $\delta$-strip points, and the comparing of each point in the $\delta$-strip with the next seven points that follow it can both be performed in linear time. Therefore, by Case 2 of the Master theorem, $T(n) = \Theta(n \log n)$.  

\[ \delta = \min \{\delta_L, \delta_R\} \]

\[ \max \text{ load} = 8 \]
Strassen’s Algorithm for Matrix Multiplication

Review of matrix multiplication

A matrix is a two-dimensional array of numbers, called the entries of the matrix. A matrix with \( m \) rows and \( n \) columns is called an \( m \times n \) matrix. The values \( m \) and \( n \) are referred to as the dimensions of the matrix. A square matrix is one in which the number of rows and columns are equal (i.e. \( m = n \)).

In this lecture we use an uppercase letter, such as \( A, B, C \), etc., to denote a matrix, and a corresponding lowercase (subscripted) letter to denote its entries. For example, if

\[
A = \begin{pmatrix} 1 & 3 & 0 \\ -2 & 2 & 5 \end{pmatrix},
\]

then \( a_{11} = 1, a_{21} = -2, \) and \( a_{23} = 5 \). In the case that a matrix consists of a single row or column, then we may also use a single subscript index. For example, if

\[
B = \begin{pmatrix} 8 \\ -5 \end{pmatrix},
\]

then we may write \( b_1 = 8 \) and \( b_2 = -5 \). Note however, that, e.g., \( b_{11} = 8 \) is also correct. Also, the notation \( a_i \) refers to the \( i \) th row of \( A \), while \( a_{\cdot j} \) refers to the \( j \) th column. For example,

\[
a_{2,\cdot} = \begin{pmatrix} -2 & 2 & 5 \end{pmatrix},
\]

while

\[
a_{\cdot 3} = \begin{pmatrix} 0 \\ 5 \end{pmatrix},
\]

To add/subtract two matrices having the same dimensions, one simply adds/subtracts the corresponding entries of each matrix. Matrix multiplication is defined differently, in that corresponding matrix entries are not necessarily multiplied. It turns out that there is a more useful multiplication operation that is based on a vector operation called the dot product. A vector is a one-dimensional array of \( n \) numbers, called its components. Here \( n \) is referred to as the dimension of the vector. In this lecture a lowercase letter, such as \( u, v, w, x, y \) etc., is used to represent a vector, while a subscripted lowercase letter denotes a component of the vector. For example,

\[
v = \begin{pmatrix} -3 & 7 & -1 & 8 \end{pmatrix}
\]

is a four-dimensional vector, while \( v_1 = -3 \), and \( v_4 = 8 \).

Notice that a vector is just a special case of a matrix. For example, vector \( v \) above can be viewed as
a 1 × 4 matrix, or, if we write it as a column, such as

\[ v = \begin{pmatrix} -3 \\ 7 \\ -1 \\ 8 \end{pmatrix}, \]

then we may view it as an 4 × 1 matrix. For this reason vector addition and subtraction is defined in the same manner as matrix addition and subtraction.

Now let \( u \) and \( v \) be \( n \)-dimensional vectors. Then the **dot product** \( u \cdot v \) is defined as

\[ u \cdot v = u_1v_1 + \cdots + u_nv_n. \]

The dot product between two vectors is also referred to as the **inner product**.

**Example 5.** Given vectors \( u = ( -3 \ 7 \ -1 \ 8 ) \) and

\[ v = ( 5 \ -2 \ -3 \ 2 ), \]

determine \( u + v, u - v, \) and \( u \cdot v. \)
Just as a vector may be viewed as a matrix, so can the rows and columns of a matrix be viewed as vectors. For example, given
\[
A = \begin{pmatrix}
1 & 3 & 0 \\
-2 & 2 & 5
\end{pmatrix}
\]
Then the row vectors of \( A \) are
\[
a_1 = \begin{pmatrix}
1 \\
-2
\end{pmatrix}, \quad a_2 = \begin{pmatrix}
3 \\
2
\end{pmatrix}
\]
while the column vectors of \( A \) are
\[
a_{11} = \begin{pmatrix}
1
\end{pmatrix}, \quad a_{12} = \begin{pmatrix}
-2
\end{pmatrix}, \quad a_{21} = \begin{pmatrix}
3
\end{pmatrix}, \quad a_{22} = \begin{pmatrix}
0
\end{pmatrix}.
\]

We are now in the position to define matrix multiplication. Let \( A \) be an \( m \times p \) matrix, and \( B \) be a \( p \times n \) matrix. Then the matrix product of \( A \) with \( B \), denoted \( AB \), is the \( m \times n \) matrix \( C = AB \) for which \( c_{ij} = a_{i} \cdot b_{j} \). In words, entry \( c_{ij} \) is obtained by taking the dot product of row \( i \) of \( A \) with column \( j \) of \( B \). Notice that \( c_{ij} = a_{i} \cdot b_{j} \) is well defined, since both \( a_{i} \) and \( b_{j} \) are \( p \)-dimensional vectors. Here are two more ways of writing \( c_{ij} \).

\[
c_{ij} = a_{i1}b_{1j} + \cdots + a_{ip}b_{pj},
\]

and, using summation notation,
\[
c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}.
\]
Example 6. Given matrices

\[ A = \begin{pmatrix} 1 & 3 & 0 \\ -2 & 2 & 5 \\ -3 & -1 & 4 \\ 2 & 0 & -1 \end{pmatrix} \]

and

\[ B = \begin{pmatrix} -2 & 1 \\ 0 & -3 \\ 2 & 0 \end{pmatrix} , \]

Then \( C = AB \) has the following entries.

\[ c_{11} = a_1 \cdot b_1 = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix} = (1)(-2) + (3)(0) + (0)(2) = -2. \]

\[ c_{12} = a_1 \cdot b_2 = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -3 \\ 0 \end{pmatrix} = (1)(1) + (3)(-3) + (0)(0) = -8. \]

\[ c_{21} = a_2 \cdot b_1 = \begin{pmatrix} -2 \\ 2 \\ 5 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix} = (-2)(-2) + (2)(0) + (5)(2) = 14. \]

\[ c_{22} = a_2 \cdot b_2 = \begin{pmatrix} -2 \\ 2 \\ 5 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -3 \\ 0 \end{pmatrix} = (-2)(1) + (2)(-3) + (5)(0) = -8. \]

\[ c_{31} = a_3 \cdot b_1 = \begin{pmatrix} -3 \\ -1 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix} = (-3)(-2) + (-1)(0) + (4)(2) = 14. \]

\[ c_{32} = a_3 \cdot b_2 = \begin{pmatrix} -3 \\ -1 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -3 \\ 0 \end{pmatrix} = (-3)(1) + (-1)(-3) + (4)(0) = 0. \]

\[ c_{41} = a_4 \cdot b_1 = \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix} = (2)(-2) + (0)(0) + (-1)(2) = -6. \]
\[ c_{42} = a_4, \cdot b_{,2} = \begin{pmatrix} 2 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -3 \\ 0 \end{pmatrix} = (2)(1) + (0)(-3) + (-1)(0) = 2. \]

Therefore,

\[ AB = C = \begin{pmatrix} -2 & -9 \\ 14 & -8 \\ 14 & 0 \\ -6 & 2 \end{pmatrix} \]

Notice that the number of steps needed to compute \( AB \) is \( \Theta(mnp) \), where \( A \) is an \( m \times p \) matrix, and \( B \) is an \( p \times n \) matrix. For example, if \( m = p = n \), then multiplying two square \( n \times n \) matrices requires \( \Theta(n^3) \) steps.
A Divide-and-Conquer algorithm for matrix multiplication

One interesting property of matrices is that their entries do not necessarily have to be real numbers. They can be any kind of element for which addition, subtraction, and multiplication have been defined. Therefore, the entries of a matrix can be matrices! For example, below is a $2 \times 2$ matrix whose entries are themselves $2 \times 2$ matrices.

\[
\begin{pmatrix}
(2 & -1) & (-1 & 3) \\
(4 & -2) & (5 & 0) \\
(4 & 1) & (2 & 6) \\
(-3 & 0) & (-2 & 3)
\end{pmatrix}
\]

**Theorem 2.** Let $A$ and $B$ be two square $n \times n$ matrices, where $n$ is even. Let $A_{11}$, $A_{12}$, $A_{21}$, and $A_{22}$ represent the four $\frac{n}{2} \times \frac{n}{2}$ submatrices of $A$ that correspond to its four quadrants. For example, $A_{11}$ consists of rows 1 through $\frac{n}{2}$ of $A$ whose entries are restricted to columns 1 through $\frac{n}{2}$. Similarly, $A_{12}$ consists of rows 1 through $\frac{n}{2}$ of $A$ whose entries are restricted to columns $\frac{n}{2} + 1$ through $n$. Finally, $A_{21}$ and $A_{22}$ represent the bottom half of $A$. Thus, $A$ can be written as

\[
\hat{A} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\]

Submatrices $B_{11}$, $B_{12}$, $B_{21}$, and $B_{22}$ are defined similarly. Finally, let $\hat{A}$ and $\hat{B}$ be the $2 \times 2$ matrices whose entries are the four quadrants of $A$ and $B$ respectively. Then the entries of $\hat{C} = \hat{A}\hat{B}$ are the four quadrants of $C = AB$.

**Proof of Theorem 2.** Consider the $(i, j)$ entry of $C = AB$. For simplicity of notation, assume that $(i, j)$ lies in the upper left quadrant of $C$. Then we have

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.
\]

We must show that $c_{ij}$ is equal to entry $(i, j)$ of $\hat{c}_{11}$, where $\hat{c}_{11}$ is the $\frac{n}{2} \times \frac{n}{2}$ matrix that is entry $(1, 1)$ of $\hat{C}$. To simplify the indexing notation, let

\[
\hat{A} = \begin{pmatrix}
p & q \\
r & s
\end{pmatrix}
\quad \text{and} \quad
\hat{B} = \begin{pmatrix}
t & u \\
v & w
\end{pmatrix}
\]

be the respective quadrant matrices of $A$ and $B$. Then matrices $p$ through $w$ are all $\frac{n}{2} \times \frac{n}{2}$ matrices. Now,

\[
\hat{c}_{11} = A_{11}B_{11} + A_{12}B_{21} = pt + qv
\]

is the sum of two matrix products. Thus, the $(i, j)$ entry of $\hat{c}_{11}$ is equal to

\[
\sum_{k=1}^{n/2} p_{ik} t_{kj} + \sum_{k=1}^{n/2} q_{ik} v_{kj} =
\]
\[
\sum_{k=1}^{n/2} a_{ik} b_{kj} + \sum_{k=1}^{n/2} a_{i(k+n/2)} b_{(k+n/2)j} =
\]

\[
\sum_{k=1}^{n/2} a_{ik} b_{kj} + \sum_{k=n/2}^{n} a_{ik} b_{kj} =
\]

\[
\sum_{k=1}^{n} a_{ik} b_{kj} = c_{ij},
\]

and the proof is complete.
Example 7. Given the matrices

\[
A = \begin{pmatrix}
-2 & -4 & 1 & 0 \\
4 & -2 & -3 & 1 \\
1 & 0 & -4 & -2 \\
-3 & 2 & 1 & -4
\end{pmatrix}, \quad
B = \begin{pmatrix}
-1 & 4 & -1 & 2 \\
3 & -3 & 4 & -1 \\
-1 & 2 & -2 & -1 \\
-2 & -2 & -1 & 4
\end{pmatrix}
\]

Verify that quadrant $C_{11}$ of $C = AB$ is equal to entry $(1, 1)$ of $\hat{C} = \hat{A} \hat{B}$.

\[
\begin{align*}
C_{11} & \quad \text{via standard multiplication:} \\
C_{11} & \quad \text{via Quadrant-matrix multiplication:}
\end{align*}
\]

\[
C_{11} = \begin{pmatrix}
-2 & -4 \\
4 & -2
\end{pmatrix} \begin{pmatrix}
-1 & 4 \\
3 & -3
\end{pmatrix} + \begin{pmatrix}
1 & 0 \\
-3 & 1
\end{pmatrix} \begin{pmatrix}
-1 & 2 \\
-2 & -2
\end{pmatrix} = \\
= \begin{pmatrix}
-10 & 4 \\
-10 & 22
\end{pmatrix} + \begin{pmatrix}
-1 & 2 \\
1 & -8
\end{pmatrix} = \begin{pmatrix}
-11 & 6 \\
-9 & 14
\end{pmatrix}
\]
Theorem 2 leads to the following divide-and-conquer algorithm for multiplying two $n \times n$ matrices $A$ and $B$, where $n$ is a power of two. Let

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

be two $n \times n$ matrices, where $n$ is a power of two, and, e.g. $a$ represents the upper left quadrant of $A$, $b$ the upper right quadrant of $A$, etc.. The goal is to compute $C = AB$, where

$$C = \begin{pmatrix} r & s \\ t & u \end{pmatrix}.$$ 

The algorithm divides $A$ and $B$ into their four quadrants and proceeds to make 8 recursive calls to obtain products $ae$, $bg$, $af$, $bh$, $ce$, $dg$, $cf$, and $dh$. Finally, these products are added to obtain

$$r = ae + bg,$$
$$s = af + bh,$$
$$t = ce + dg,$$
$$u = cf + dh.$$ 

Letting $T(n)$ denote the running time of the algorithm, then $T(n)$ satisfies

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

where the first term is due to the 8 recursive calls on matrices whose dimensions are $n/2$, and the second term $n^2$ represents the big-O number of steps needed to divide $A$ and $B$ into their quadrants, and to add the eight products that form the quadrants of $C$. Therefore, by Case 1 of the Master Theorem, $T(n) = \Theta(n^3)$, and the algorithm’s running time is equivalent to the running time when using the standard matrix-multiplication procedure.

$$f(n) = n^2 = O(n^{1.8928}) = \Theta(n^3) \Rightarrow T(n) = \Theta(n^3)$$
Strassen’s insight was to first take linear combinations of the quadrants of $A$ and $B$, and then multiply these combinations. By doing this, he demonstrated that only 7 products are needed. These products are then added to obtain the quadrants of $C$. Moreover, since computing a linear combination of $A$ and $B$ quadrants takes $\Theta(n^2)$ steps (since we are just adding and subtracting a constant number of $n/2 \times n/2$ matrices), the recurrence produced by Strassen is

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

which improves the running time to $n^{\log_7 7}$, where $\log 7 \approx 2.8$. 
Strassen’s Seven Matrix Products

1. \( A_1 = a, \ B_1 = f - h, \ P_1 = A_1B_1 = a(f - h) = af - ah \)
2. \( A_2 = a + b, \ B_2 = h, \ P_2 = A_2B_2 = (a + b)h = ah + bh \)
3. \( A_3 = c + d, \ B_3 = e, \ P_3 = A_3B_3 = (c + d)e = ce + de \)
4. \( A_4 = d, \ B_4 = g - e, \ P_4 = A_4B_4 = d(g - e) = dg - de \)
5. \( A_5 = a + d, \ B_5 = e + h, \ P_5 = A_5B_5 = (a + d)(e + h) = ae + ah + de + dh \)
6. \( A_6 = b - d, \ B_6 = g + h, \ P_6 = A_6B_6 = (b - d)(g + h) = bg + bh - dg - dh \)
7. \( A_7 = a - c, \ B_6 = e + f, \ P_7 = A_7B_7 = (a - c)(e + f) = ae - ce - cf + af \)

Example 8. Write \( r, s, t, u \) as linear combinations of \( P_1, \ldots, P_7 \).

\[
\begin{align*}
  r &= ae + bg = P_5 + P_6 + P_4 - P_2 \\
  s &= af + bh = af + bh = P_1 + P_2 \\
  t &= ce + dg = P_3 + P_4 \\
  u &= cf + dh = -P_7 + P_5 + P_1 - P_3
\end{align*}
\]
Exercises

1. Draw the recursion tree that results when applying Mergesort to the array 5, −2, 0, 7, 3, 11, 2, 9, 5, 6. Label each node with the sub-problem to be solved at that point of the recursion. Assume arrays of size 1 and 2 are base cases. Assume that odd-sized arrays are split so that the left subproblem has one more integer than the right. Next to each node, write the solution to its associated subproblem.

2. Write an algorithm for the function Merge that takes as input two sorted integer arrays $a$ and $b$ of sizes $m$ and $n$ respectively, and returns a sorted integer array of size $m + n$ whose elements are the union of elements from $a$ and $b$.

3. Perform the partitioning step of Quicksort on the array 5, −2, 0, 7, 3, 11, 2, 9, 5, 6, where the pivot is chosen using the median-of-three heuristic.

4. Suppose the average-case running time of Quicksort is to be computed when applied to a random permutation of the array $a = 1, 2, \ldots, n$. Moreover, assume that Quicksort chooses its pivot randomly and uniformly from the available array members. Empirically, Quicksort may be run against several random-permutation test cases, and we keep track of the following statistics:

   $$ T(0), T(1), \ldots, T(n - 1), T_1(n), \ldots, T_n(n). $$

   These are defined as follows. At the top level of recursion, if $i \in [1, n]$ is the selected random pivot, then the two subproblems have sizes $i - 1$ and $n - i$, respectively. For simplicity, assume the partition step requires $n$ steps (i.e. we ignore the multiplicative constant). Furthermore, add to $T(i - 1)$ the time required to sort the first array subproblem, and add to $T(n - i)$ the time required to sort the second. Finally, add to $T_i(n)$ the quantity

   $$ T(i - 1) + T(n - i) + n. $$

   After all test cases have been run, we define $T(n)$ as

   $$ T(n) = \frac{1}{n} (T_1(n) + \cdots + T_n(n)), $$

   i.e. the uniform average of each $T_i(n)$. This makes sense since all pivot values have equal probability. Use this definition to show that

   $$ T(n) = n + \frac{2}{n} \sum_{i=1}^{n} T(i - 1). $$

   where we assume that the partitioning algorithm takes exactly $n$ steps.

5. Use the substitution method to prove that, for the recurrence for $T(n)$ derived in the previous exercise, it implies that $T(n) = O(n \log n)$. Hint: you’ll also need the Integral theorem with careful attention paid to the choice of constant $C > 0$.

6. Provide a permutation of the numbers 1-9 so that, when sorted by Quicksort using median-of-three heuristic, the left subarray always has one element in rounds 1, 2, and 3. Note: in general, when using the median-of-three heuristic, Quicksort is susceptible to $\Theta(n^2)$ worst case performance.
7. For the Minimum-Distance-Pair algorithm, if the input points are
\[ X = (-1, 2), (0, 2), (1, 4), (3, 1), (3, -1), (4, 4), \]
then provide \( \delta \) and the equation of the median line.

8. Given \( n \) distinct integers, prove that the greatest element of \( a \) can be found using \( n - 1 \) comparisons, and that one can do no better than \( n - 1 \).

9. Given \( n \) distinct integers, show that the second greatest element can be found with \( n + \lceil \log n \rceil - 2 \) comparisons in the worst case.

10. Given \( n \) distinct integers, prove the lower bound of \([3n/2] - 2\) comparisons in the worst case to determine both the least and greatest element.

11. If
\[ a = 2, 4, 1, 3, 8, 9, 3, 5, 7, 6, 5, 8, 5 \]
serves as input to the Median-of-Five Find Statistic algorithm, then what pivot is used for the algorithm’s partitioning step?

12. For the Median-of-Five Find Statistic algorithm, does the algorithm still run in linear time if groups of seven are used? Explain and show work. How about groups of 3?

13. For the Median-of-Five Find Statistic algorithm, show that if \( n \geq 200 \), then at least \( \lceil n/4 \rceil \) elements of \( a \) are greater than (and respectively less than) or equal to the median of medians.

14. Explain how the Median-of-Five Find Statistic Algorithm could be used to make Quicksort run in time \( O(n \log n) \) in the worst-case.

15. Suppose you have a “black box” worst-case linear-time algorithm that can find the median of an array of integers. Using this algorithm, describe a simple linear-time algorithm that solves the Find \( k \) th Statistic problem. Prove that your algorithm runs in linear time.

16. The \( q \) th quantiles of an \( n \)-element array are the \( q - 1 \) order statistics that divide the sorted array into \( q \) equal-sized subarrays (to within 1). In other words, the \( q \) th quantiles of an \( n \)-element array are the \( q - 1 \) \( k \) th least elements of \( a \), for
\[ k = \lfloor n/q \rfloor, \lfloor 2n/q \rfloor, \ldots, \lfloor (q - 1)n/q \rfloor. \]
Provide the 3rd quantiles for the array of integers
\[ 5, 8, 16, 2, 7, 11, 0, 9, 3, 4, 6, 7, 3, 15, 5, 12, 4, 7. \]

17. Provide an \( O(n \log q) \)-time algorithm that finds the \( q \) th quantiles of an array. Hint: modify the Find-Statistic algorithm so that multiple statistics (i.e. the \( q \) th quantiles) can be simultaneously found. At what level of recursion will the algorithm reduce to the original algorithm for just one statistic? Notice that from this level down the algorithm will then run in linear time in the size of the array at that level.

18. Describe an \( O(n) \)-time algorithm that, given an array of \( n \) distinct numbers, and a positive integer \( k \leq n \), determines the \( k \) elements in the array that are closest to the median of the array. Hint: first find the median and form a new array that is capable of giving the answer.
19. Let $a$ and $b$ be two odd-lengthed $n$-element arrays already in sorted order. Give an $O(\log n)$-time algorithm to find the two medians of all the $2n$ elements in arrays $a$ and $b$ combined, denoted $a \cup b$.

20. Given the matrices

$$A = \begin{pmatrix} 1 & 0 & -2 \\ -1 & 2 & 3 \\ 3 & 1 & -4 \end{pmatrix},$$

and

$$B = \begin{pmatrix} 3 & -1 \\ 2 & 1 \\ -1 & 2 \end{pmatrix},$$

compute entries $c_{11}$, $c_{21}$, and $c_{32}$ of matrix $C = AB$.

21. Given the matrices

$$A = \begin{pmatrix} 3 & 0 \\ -1 & 2 \\ 1 & 1 \end{pmatrix},$$

and

$$B = \begin{pmatrix} 4 & -1 \\ 0 & 2 \end{pmatrix},$$

Compute $AB$.

22. Given the matrices

$$A = \begin{pmatrix} 3 & -2 & 7 \\ 6 & 5 & 4 \\ 0 & 4 & 9 \end{pmatrix},$$

and

$$B = \begin{pmatrix} 6 & -2 & 4 \\ 0 & 1 & 3 \\ 7 & 7 & 5 \end{pmatrix},$$

Compute $AB$ and $BA$.

23. For the matrices $A$ and $B$ in Example 7, compute the remaining quadrants $C_{12}$, $C_{21}$, and $C_{22}$ of $C = AB$ and verify that they are the entries of matrix $\hat{C} = \hat{A}\hat{B}$, where, e.g. $\hat{A}$ is the matrix whose entries are the quadrants of $A$.

24. Prove the other four cases of Theorem 2, i.e. the cases where entry $(i,j)$ of $C$ lies in the upper right, lower left, and lower right quadrant.

25. Use Strassen’s algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

26. Suppose you want to apply Strassen’s algorithm to square matrices whose number of rows are not powers of 2. To do this you, add more columns and rows of zeros to each matrix until the number of rows (and columns) of each matrix reaches a power of 2. The perform the algorithm. If $m$ is the original dimension, and $n$ is the dimension after adding more rows and columns, is the running time still $\Theta(m^{\log 7})$? Explain and show work.
27. What is the largest $k$ such that you can multiply $3 \times 3$ matrices using $k$ multiplications, then you can multiply matrices in time $o(n^{\log 7})$? Explain and show work.

28. Professor Jones has discovered a way to multiply $68 \times 68$ matrices using 132,464 multiplications, and a way to $70 \times 70$ matrices using 143,640 multiplications. Which method yields the better asymptotic running time? How do these methods compare with Strassen’s algorithm?

29. Using Strassen’s algorithm, describe an efficient way to multiply a $kn \times n$ matrix with an $n \times kn$ matrix. You may assume $n$ is a power of 2.

30. Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take $a$, $b$, $c$, and $d$ as input, and produce the real component $ac – bd$ and imaginary component $ad + bc$. Note that the straightforward approach requires four multiplications. We seek a more clever approach.

31. Consider the following algorithm called multiply for multiplying two $n$-bit binary numbers $x$ and $y$. Let $x_L$ and $x_R$ be the leftmost $\lceil n/2 \rceil$ and rightmost $\lfloor n/2 \rfloor$ bits of $x$ respectively. Define $y_L$ and $y_R$ similarly. Let $P_1$ be the result of calling multiply on inputs $x_L$ and $y_L$, $P_2$ be the result of calling multiply on inputs $x_R$ and $y_R$, and $P_3$ the result of calling multiply on inputs $x_L + x_R$ and $y_L + y_R$. Then return the value $P_1 \times 2^{\lceil \frac{n}{2} \rceil} + (P_3 – P_1 – P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$. Provide the divide-and-conquer recurrence for this algorithm’s running time $T(n)$, and use it to determine the running time.

32. For the two binary integers $x = 1101111$ and $y = 1011101$, determine the top-level values of $P_1$, $P_2$, and $P_3$, and verify that $xy = P_1 \times 2^{\lceil \frac{n}{2} \rceil} + (P_3 – P_1 – P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$.

33. Verify that the algorithm always works by proving in general that $xy = P_1 \times 2^n + (P_3 – P_1 – P_2) \times 2^{n/2} + P_2$ for arbitrary $x$ and $y$. Hint: you may assume that $x$ and $y$ both have even lengths as binary words.

34. Given an array $a[]$ of integers, a subsequence of the array is a sequence of the form $a[i], a[i + 1], a[i + 2], \ldots, a[j]$, where $i \leq j$. Moreover, the sum of the subsequence is defined as $a[i] + a[i + 1] + a[i + 2] + \cdots + a[j]$. Describe in words a divide-and-conquer algorithm for finding the maximum sum that is associated with any subsequence of the array. Make sure your description has enough detail so that someone could read it and understand how to program it.

35. Provide a divide-and-conquer recurrence relation that describes the running time $T(n)$ of the algorithm from the previous problem, and use the Master Theorem to provide an asymptotic solution for the running time.

36. Repeat the previous two problems, but now your algorithm should find the minimum positive subsequence sum. In other words, of all subsequences whose sum adds to a positive number, you want to determine the minimum of such sums.
Exercise Hints and Answers

1. We use $lr$-strings for the addresses of each node. For example, $\lambda$ denotes the root, while $lrr$ denotes the right child of the right child of the left child of the root. Then

\[
\begin{align*}
\lambda &: 5, -2, 0, 7, 3, 11, 2, 9, 5, 6 \\
l &: 5, -2, 0, 7, 3 \\
r &: 11, 2, 9, 5, 6 \\
ll &: 5, -2 \quad llr &: 0 \\
rl &: 11, 2 \quad rll &: 11, 2 \quad rlr &: 9
\end{align*}
\]

2. //Merges a and b into merged. n= |a|, m = |b|
void merge(int[] a, int n, int[] b, int m, int[] merged)
{
    int i=0;
    int j=0;
    int value_a = a[0];
    int value_b = b[0];
    int count = 0;

    while(true)
    {
        if(value_a <= value_b)
        {
            merged[count++] = value_a;
            i++;

            if(i < n)
                value_a = a[i];
            else
                break;
        }
        else
        {
            merged[count++] = value_b;
            j++;

            if(j < m)
                value_b = b[j];
            else
                break;
        }
    }

    //copy remaining values
    if(j < m)
        for(; j < m; j++)
merged[count++] = b[j];
else
    for(; i < n; i++)
        merged[count++] = a[i];
}

3. Pivot = 5. \(a_{\text{left}} = 5, -2, 0, 2, 3\), \(a_{\text{right}} = 7, 9, 6, 11\)

4. Consider each of the equations
\[
\frac{1}{n}(T_i(n) = n + T(i - 1) + T(n - i)),
\]
for \(i = 1, 2, \ldots, n\). Add the left sides of each equation to get \(T(n)\), and add the right sides to get the right side of the desired recurrence.

5.

6. 173924685 is one possible permutation. Verify!

7. \(\delta_L = 1 = d((-1, 2), (0, 2))\), while \(\delta_R = 2 = d((3, 1), (3, -1))\). Hence, \(\delta = 1\). Also, vertical line \(x = 2\) serves as a median line, and is equidistant to the closest points on either side.

8. Let \(S_0\) denote the set of \(n\) integers. While there is more than one integer in \(S_i\), \(i \geq 0\), pair up the integers in \(S_j\). If \(|S_i|\) is odd, then add the last (unpaired) integer to \(S_{i+1}\). Perform a comparison on each pair and add the greater integer to \(S_{i+1}\). Thus, there is a one-to-one correspondence between integers that are left out of the next set \(S_{i+1}\) and comparisons performed (during iteration \(i\)). Moreover, since there will be some \(j\) for which \(|S_j| = 1\), \(S_j\) will contain the greatest integer after a total of \(n - 1\) comparisons.

9. Since the second greatest integer \(n_2\) does not appear in the final set \(S_j\) (see previous problem), there must exist an iteration \(i\) for which \(n_2\) is compared with \(n_1\), the greatest integer. This is true since \(n_1\) is the only integer that could prevent \(n_2\) from advancing. Thus, \(n_2\) can be found by examining the integers that were compared with \(n_1\). Since \(n_1\) is compared with at most \(\lceil \log n \rceil\) integers (why?), we can use the result of the previous problem to conclude that \(n_2\) can be found by first determining \(n_1\) using \(n - 1\) comparisons, and then using the same algorithm on the elements that were compared with \(n_1\) to find \(n_2\). This requires an additional \(\lceil \log n \rceil - 1\) comparisons. This gives a total of \(n + \lceil \log n \rceil - 2\) comparisons.

10. Pair up the integers and compare each pair. Place the greater integers in set \(G\), and the lesser integers in set \(L\). Now find the greatest element of \(G\), and the least element of \(L\).

11. The medians of groups \(G_1\), \(G_2\), and \(G_3\) are respectively, 3, 6, and 5. Therefore, the pivot is median(3, 6, 5) = 5.

12. True for groups of 7, since new recurrence is \(T(n) \leq T([n/7]) + T(5n/7 + 12) + O(n)\). Use the result that \(T(n) = T(an) + T(bn) + O(n)\), with \(a + b < 1\), implies \(T(n) = O(n)\). Not true for groups of 3, since, in the worst-case, the new recurrence is \(T(n) = T([n/3]) + T(2n/3) + O(n)\) which yields log-linear growth in the worst case. This can be verified using the substitution method.
13. \[ 3n/10 - 9 \geq n/4 + 1 \Leftrightarrow (6n - 180)/20 \geq (5n + 20)/20 \Leftrightarrow n \geq 200. \]

14. Use the Find-Statistic algorithm to determine the median \( M \) of the array, and use \( M \) as the pivot in the partitioning step. This ensures a Quicksort running time of \( T(n) = 2T(n/2) + O(n) \), since both subarrays are now guaranteed to have size \( n/2 \).

15. Similar to the previous problem, the black-box algorithm can be used to find the median \( M \) of the array, and use \( M \) as the pivot in the partitioning step. This ensures a running time of \( T(n) = T(n/2) + O(n) \).

16. The 3rd quantiles occur at index values \[ n/3 \] and \[ 2n/3 \] (of the sorted array). This corresponds with \( k = 6 \) and \( k = 12 \). Associated with these indices are elements 5 and 8, respectively.

17. If we modify Find-Statistic to simultaneously find each of the quantiles (there are \( q - 1 \) of them), then, since the quantiles are spread across the entire array, then, after the partitioning step, we will need to make recursive calls on both \( a_{\text{left}} \) and \( a_{\text{right}} \) (we may assume that we are using the exact median for the pivot during the partition step since the median can be found in linear time). The recurrence is thus \( T(n) = 2T(n/2) + O(n) \). Note however, that once the array sizes become sufficiently small during the recursion, there can be at most one quantile inside each array. Indeed, the quantiles are a guaranteed distance of \( n/q \) apart from each other. Moreover, the array sizes are being halved at each level of recursion, it will take a depth of \( \log q \) (verify!) before the array sizes are sufficiently small to only possess at most one quantile. When this happens, the normal Find-Statistic algorithm may be used, since now only a single \( k \) value is being sought. The running time is thus \( O(n \log q) \) for computational steps applied down to depth \( \log q \) of the recursion tree. The remainder of the tree consists of \( q \) problems of size \( n/q \), and each of these problems can be solved in linear time using the original Find-Statistic algorithm. This yields an additional \( qO(n/q) = O(n) \) running time. Therefore the total running time is \( O(n \log q) \).

18. Use Find-Statistic to find the median \( m \) of \( a \) in linear time. Then create the array \( b \), where \( b[i] = |a[i] - m| \). Then in linear time find the \( k \) th least element \( e \) of \( b \) (along with the subarray of elements of \( b \) that are all less than or equal to \( e \)). Translate these elements back to elements of \( a \).

19. For the base case, if \( n = 1 \), then \( a \cup b \) has two elements, each of which is a median. Now suppose \( n > 1 \) is odd. Let \( m_a \) be the median of \( a \), and \( m_b \) the median of \( b \) (both can be found in constant time since both arrays are sorted). If \( m_a = m_b \), then \( m_a = m_b \) are the two desired medians. of \( a \cup b \). Otherwise, assume WLOG (without loss of generality) that \( m_a < m_b \). If \( m \) is a median of \( a \cup b \), then we must have \( m_a \leq m \leq m_b \). Otherwise, suppose, e.g., that \( m < m_a \). Then there would be more elements of \( a \cup b \) that are to the right of \( m \) (why ?). Similarly, it is not possible for \( m > m_b \). Hence, the elements \( a_L \) of \( a \) to the left of \( m_a \) must be less than or equal to \( m \). Similarly, the elements \( b_R \) of \( b \) to the right of \( m_b \) must be greater than or equal to \( m \). Thus, if we remove \( a_L \) from \( a \) and \( b_R \) from \( b \), we obtain two odd-lenthed arrays that are now half the size, yet that still have the same medians as the previous arrays. Repeat the process until the base case is reached. Running time is \( O(\log n) \), since it satisfies the recurrence \( T(n) = T(n/2) + 1 \).

20. The entries are 

\[
c_{11} = a_{11} \cdot b_{11} = (1) (3) + (0) (2) + (-2) (-1) = 5,
\]

33
c_{21} = a_2 \cdot b_{11} = (-1)(3) + (2)(2) + (3)(-1) = -2,

and

c_{32} = a_3 \cdot b_{2} = (3)(-1) + (1)(1) + (4)(2) = 6.

21. The product of $A$ and $B$ is

$$AB = \begin{pmatrix} 12 & -3 \\ -4 & 5 \\ 4 & 1 \end{pmatrix}.$$  

22. The product of $A$ and $B$ is

$$AB = \begin{pmatrix} 67 & 41 & 41 \\ 64 & 21 & 59 \\ 63 & 67 & 57 \end{pmatrix},$$

while

$$BA = \begin{pmatrix} 6 & -6 & 70 \\ 6 & 17 & 31 \\ 63 & 41 & 122 \end{pmatrix}.$$  

23. We have

\begin{align*}
C_{12} &= \hat{c}_{12} = \hat{a}_{11} \hat{b}_{12} + \hat{a}_{12} \hat{b}_{22} = A_{11}B_{12} + A_{12}B_{22} = \\
&= \begin{pmatrix} -2 & -4 \\ 4 & -2 \end{pmatrix} \begin{pmatrix} -1 & 2 \\ 4 & -1 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -3 & 1 \end{pmatrix} \begin{pmatrix} -2 & -1 \\ -1 & 4 \end{pmatrix} = \\
&= \begin{pmatrix} -14 & 0 \\ -12 & 10 \end{pmatrix} + \begin{pmatrix} -2 & -1 \\ 5 & 7 \end{pmatrix} = \begin{pmatrix} -16 & -1 \\ -7 & 17 \end{pmatrix},
\end{align*}

\begin{align*}
C_{21} &= \hat{c}_{21} = \hat{a}_{21} \hat{b}_{11} + \hat{a}_{22} \hat{b}_{21} = A_{21}B_{11} + A_{22}B_{21} = \\
&= \begin{pmatrix} 1 & 0 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} -1 & 4 \\ 3 & -3 \end{pmatrix} + \begin{pmatrix} -4 & -2 \\ 1 & -4 \end{pmatrix} \begin{pmatrix} -1 & 2 \\ -2 & -2 \end{pmatrix} = \\
&= \begin{pmatrix} -1 & 4 \\ 9 & -18 \end{pmatrix} + \begin{pmatrix} 8 & -4 \\ 16 & 10 \end{pmatrix} = \begin{pmatrix} 7 & 0 \\ 16 & -8 \end{pmatrix},
\end{align*}

and

\begin{align*}
C_{22} &= \hat{c}_{22} = \hat{a}_{21} \hat{b}_{12} + \hat{a}_{22} \hat{b}_{22} = A_{21}B_{12} + A_{22}B_{22} = \\
&= \begin{pmatrix} 1 & 0 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} -1 & 2 \\ 4 & -1 \end{pmatrix} + \begin{pmatrix} -4 & -2 \\ 1 & -4 \end{pmatrix} \begin{pmatrix} -2 & -1 \\ -1 & 4 \end{pmatrix} = \\
&= \begin{pmatrix} -1 & 2 \\ 11 & -8 \end{pmatrix} + \begin{pmatrix} 10 & -4 \\ 2 & -17 \end{pmatrix} = \begin{pmatrix} 9 & -2 \\ 13 & -25 \end{pmatrix}.
\end{align*}

Verify by direct multiplication of $A$ with $B$ that these are the quadrants of $C = AB$.

24. Consider the case where entry $(i, j)$ lies in the upper right quadrant of $C$. Then we have

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$
We must show that \( c_{ij} \) is equal to entry \((i,j - n/2)\) of \( \hat{c}_{12} \), where \( \hat{c}_{12} \) is the \( \frac{n}{2} \times \frac{n}{2} \) matrix that is entry \((1,2)\) of \( \hat{C} \). To simplify the indexing notation, let

\[
\hat{A} = \begin{pmatrix} p & q \\ r & s \end{pmatrix} \quad \text{and} \quad \hat{B} = \begin{pmatrix} t & u \\ v & w \end{pmatrix}
\]

be the respective quadrant matrices of \( A \) and \( B \). Then matrices \( p \) through \( w \) are all \( \frac{n}{2} \times \frac{n}{2} \) matrices. Now,

\[
\hat{c}_{12} = A_{11}B_{12} + A_{12}B_{22} = pu + qw
\]

is the sum of two matrix products. Thus, the \((i,j - n/2)\) entry of \( \hat{c}_{12} \) is equal to

\[
\sum_{k=1}^{n/2} p_{ik}u_{k(j-n/2)} + \sum_{k=1}^{n/2} q_{ik}w_{k(j-n/2)} = 
\sum_{k=1}^{n/2} a_{ik}b_{kj} + \sum_{k=1}^{n/2} a_{i(k+n/2)}b_{(k+n/2)j} = 
\sum_{k=1}^{n/2} a_{ik}b_{kj} + \sum_{k=n/2+1}^{n} a_{ik}b_{kj} = 
\sum_{k=1}^{n} a_{ik}b_{kj} = c_{ij},
\]

and the proof is complete. The proof is similar for the cases when \((i,j)\) is in either the lower left or lower right quadrant.

25. \[
\begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}
\]

26. Padding the matrices with rows and columns of zeros to get a power of 2 number of rows will at most double the number of rows/columns of the matrix. But if \( T(n) = cn^k \), then \( T(2n) = c(2n)^k = 2^k cn^k \), and so the running time is still \( \Theta(n^k) \).

27. We need the largest \( k \) for which \( T(n) = kT(n/3) + O(n^2) \) yields a better running time than \( T(n) = 7T(n/2) + O(n^2) \). Thus we need \( \log_3 k < \log 7 \), or \( k = \lceil 3^{\log 7} \rceil \).

28. \( \log_{68}(132,464) \approx 2.795 \). Also, \( \log_{70}(143,640) \approx 2.795 \), so they are approximately the same in terms of running time. They are slightly better than Strassen’s algorithm, since \( \log 7 \approx 2.8 \).

29. We can think of the first matrix as a “column” of \( n \times n \) matrices \( A_1 \cdots A_k \), where the second matrix as a “row” of \( n \times n \) matrices \( B_1 \cdots B_k \). The product thus consists of \( k^2 \ n \times n \) blocks \( C_{ij} \), where \( C_{ij} = A_iB_j \). Thus, the product can be found via \( k^2 \) matrix multiplications, each of size \( n \times n \). Using Strassen’s algorithm yields a running time of \( \Theta(k^2 n^{\log 7}) \).

30. \( ad, bc, (a + b)(c - d) \)

31. \( T(n) = 3T(n/2) + O(n) \) yields \( T(n) = \Theta(n^{\log 3}) \).
32. \( x = 111, \ y = 93, \ x_L = 13, \ y_L = 11, \) and \( y_R = 5. \) \( P_1 = 143, \ P_2 = 35, \ P_3 = 320. \) 
\[
(64)(143) + (8)(320 - 143 - 35) + 35 = 10323 = (111)(93).
\]

33. We have \( x = (2^{n/2}x_L + x_R) \) and \( y = (2^{n/2}y_L + y_R). \) Multiply these together to derive the right side of the equation.

34. Divide the array \( a \) into equal-length subarrays \( a_L \) and \( a_R. \) Let \( \text{MSS}_L \) denote the MSS of \( a_L \) (found by making a recursive call), \( \text{MSS}_R \) denote the MSS of \( a_R. \) Then calculate \( \text{MSS}_{middle} \) in linear time by adding the MSS of \( a_L \) that ends with the last element of \( a_L \) to the MSS of \( a_R \) that begins with the first element of \( a_R. \) Return the maximum of \( \text{MSS}_L, \ \text{MSS}_R, \) and \( \text{MSS}_{middle}. \)

35. The running time recurrence satisfies \( T(n) = 2T(n/2) + O(n). \)

36. Same algorithm as for MSS of previous problem, but now it is not so easy to compute \( \text{MPSS}_{middle}, \) since it may be realized by any subsequence sum of \( a_L \) that ends with the last element of \( a_L \) being added to any other subsequence sum of \( a_R \) that begins with the first element of \( a_R. \) For both \( a_L \) and \( b_L \) there are \( n/2 \) such subsequence sums. Sort those of \( a_L \) in ascending order into a list \( S_L. \) Similarly sort those of \( a_R \) in descending order into a list \( S_R. \) Let \( i \) be an index marker of \( S_L, \) and \( j \) an index marker for \( S_R. \) Set \( s_{min} = \infty. \) If \( s = S_L(i) + S_R(j) \leq 0, \) then increment \( i. \) Else if \( s < s_{min}, \) then set \( s_{min} = s, \) and increment \( j. \) Otherwise, we have \( s > s_{min}, \) in which case we increment \( j. \) When either the elements of \( S_L \) or \( S_R \) have been exhausted, then set \( \text{MPSS}_{middle} = s_{min}. \) Running time \( T(n) \) satisfies \( T(n) = 2T(n/2) + an \ log \ n. \) Hence, by using the Master Equation and the substitution method, we can prove that \( T(n) = \Theta(n \ log^2 n). \)