Divide and Conquer Algorithms

Last Updated: September 30th, 2023

Introduction

A **divide-and-conquer algorithm** $\mathcal{A}$ follows the following general steps.

Base Case If the problem instance is O(1) in size, then use a brute-force procedure that requires O(1) steps.

Divide Divide the problem instance into one or more subproblem instances, each having a size that is smaller than the original instance.

Conquer Each subproblem instance is solved by making a recursive call to $\mathcal{A}$.

Combine Combine the subproblem-instance solutions into a final solution to the original problem instance.

The following are some problems that can be solved using a divide-and-conquer algorithm.

Binary Search locating an integer in a sorted array of integers

Quicksort and Mergesort sorting an array of integers

Order Statistics finding the k th least or greatest integer of an array

Convex Hulls finding the convex hull of a set of points in $\mathcal{R}^n$

Minimum Distance Pair finding two points from a set of points in $\mathcal{R}^2$ that are closest

Matrix Operations matrix inversion, matrix multiplication, finding the largest submatrix of 1's in a Boolean matrix.

Fast Fourier Transform finding the product of two polynomials

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 determinng 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.

1 Hoare's Quicksort

Before introducing Hoare's 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 1.1. Determine the median of $7, 5, 7, 3, 4, 8, 2, 3, 7, 8, 2$, and the medians of $4, 5, 10, 12, 6, 3$.

Solution.

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_{left} and a_{right}. However, where as Mergesort simply divides a into two equal halves, Quicksort performs the **Partitioning Algorithm** on a which is described below.

1.1 Partitioning Algorithm

Calculate Pivot The pivot M is an element of a which is used to divide a into two subarrays a_{left} and a_{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 with the last member of a located at index $n-1$ (M is now in a safe place).

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_{left} consists of the first i elements of a, while a_{right} consists of the last $n-i-1$ elements of a.Thus, $a[i] = M$ is to the right of a_{left} and to the left of a_{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 partitiioned a into a_{left} and a_{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_{left} occupies locations 0 through 10 of a, then only those locations will be affected when Quicksort is called on input a_{left}. It is this in-place property that gives Quicksort an advantage over Mergesort.

Example 1.2. Demonstrate the quicksort algorithm using the array $5, 8, 6, 2, 7, 1, 0, 9, 3, 4, 6$.

Solution.

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_{left} always has a length equal to one. In this case the lengths of each a_{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 (assuming n is odd)

$$T(n) = O(n + (n-2) + (n-4) + \cdots + 1) = O(\sum_{i=1}^{(n+1)/2} (2i-1)) = O(n^2),$$

and so `Quicksort` has a worst-case quadratic running time.

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, `lower` and `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).

```
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

```
find_statistic(a,k,0,n-1)
```

Below is an implementation of `find_statistic`

```
//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 `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 $T(n) = \text{O}(n)$.

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

```
oracle(a,lower,upper)
```

with

```
find_statistic(a,n/2,lower,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 `lower` value been increased, nor has the `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_{medians}. This array has a length of $\lceil n/5 \rceil$ elements. Now replace the function call

```
oracle(a,lower,upper)
```

with

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

In other words, the new pivot is equal to the median of the medians of each group of five.

Example 2.1. Demonstrate how the pivot is selected for the (median-of-five) `find_statistic` algorithm using the array

$$5, 18, 36, 42, 27, 11, 70, 49, 33, 84, 66, 37, 83, 15, 5, 42, 54, 97, 68, 43, 92, 77.$$

Theorem 1. The `find_statistic` algorithm has a worst-case running time of $T(n) = \mathrm{O}(n)$.

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_{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 $\lfloor bn \rfloor$ represents the worst-case length of either a_{left} or a_{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_{medians}.

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

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

elements of a, assuming $n \geq 180$.

Proof of Claim. Since M is the median of a_{medians} it must be greater than or equal to at least $L = \lfloor \frac{1}{2} \lceil \frac{n}{5} \rceil \rfloor$ elements of a_{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_{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

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

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

Furthermore, using the inequalities, $\lfloor x \rfloor \geq x - 1$ and $\lceil x \rceil \geq x$, we arrive at $3(L-2) \geq \frac{3n}{10} - 9$. Finally, basic algebra shows that the inequality

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

is true provided $n \geq 180$. 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_{left} and a_{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 $T(n) = \mathrm{O}(n)$.

3 Randomized Versions of Quicksort and Find-Statistic

Although the Median-of-Five Find-Statistic algorithm seems both clever in its design and valuable for establishing a worst-case linear bound on computing an array statistic (including the median), on average it can be significantly outperformed by a randomized algorithm that randomly selects the pivot in each round rather than use the "median of medians-of-five" heuristic.

We may also randomize Hoare's Quicksort algorithm, although in this case the improvement is not as significant given that the "median-of-three" heuristic only requires O(1) steps and tends to induce good array splits on average.

Definition 3.1. A **finite random variable** X is one whose domain is a finite set of real numbers $\{x_1, \ldots, x_n\}$, and associated with each domain member x_i is a probability value p_i, with the property that $p_1 + \cdots + p_n = 1$. The collection of probabilities, denoted as $\vec{p}$ is called the **probability distribution** of X.

A similar definition may be given for a **discrete random variable**, with the only difference being that its domain is now countably infinite.

Definition 3.2. Let X be a finite random variable whose domain is $\{x_1, \ldots, x_n\}$ and whose distribution is $\vec{p} = (p_1, \ldots, p_n)$. Then the **expectation** of X, denoted $E[X]$, is defined as

$$E[X] = \sum_{i=1}^{n} x_i \cdot p_i.$$

A similar definition may be given for a discrete random variable.

In the case of both randomized algorithms, the running time now becomes a random variable, and we content ourselves with determining an upper bound for the expectation of this random variable. We call this value the **average running time**. Furthermore, we let $T(n)$ denote the expectation of the randomized algorithm on some input array a which we assume is a permutation of n distinct integers. Because our algorithms, on average, behave the same on any such input array, we do not concern ourselves with its members and their relative order in the array.

Proposition 3.3. Let X, Y, and Z be random variables, with $Z = X + Y$. Then

$$E[X+Y] = E[X] + E[Y].$$

Proof. Without loss of generality, we assume X has domain $\{x_1, \ldots, x_m\}$ and distribution $\vec{p}$, Y has domain $\{y_1, \ldots, y_n\}$ and distribution $\vec{q}$, while Z has domain $\{x_i + y_j | 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$ and distribution r_{ij}. Then we have

$$E[Z] = \sum_{i=1}^{m} \sum_{j=1}^{n} (x_i + y_j) r_{ij} = [(x_1+y_1)r_{11} + \cdots + (x_1+y_n)r_{1n}] + \cdots + [(x_m+y_1)r_{m1} + \cdots + (x_m+y_n)r_{mn}] =$$

$$[x_1(r_{11} + \cdots + r_{1n}) + \cdots + x_m(r_{m1} + \cdots + r_{mn})] + y_1(r_{11} + \cdots + r_{m1}) + \cdots + y_n(r_{1n} + \cdots + r_{mn}) =$$

$$(x_1 \cdot p_1 + \cdots + x_m \cdot p_m) + (y_1 \cdot q_1 + \cdots + y_n \cdot q_n) = E[X] + E[Y],$$

where we have used the fact that, for all $i = 1, \ldots, m$

$$p_i = r_{i1} + \cdots + r_{in},$$

and, for all $j = 1, \ldots, n$,

$$q_j = r_{1j} + \cdots + r_{mj}.$$

$\square$

3.1 Analysis of Randomized Find-Statistic Algorithm

We first analyze the expected runnning time of the randomized Find-Statistic algorithm and make use of the fact that, if a random variable X satisfies $X = Y + Z$, where Y and Z are both random variables, then

$$E[X] = E[Y] + E[Z],$$

where $E[X]$ refers to the **expectation** or **average** of X. In words, the expectation of a sum is the sum of the expectations for each of the variables in the sum. Moreover, we let $E[Y]$ denote the expected time it takes to narrow the search to an array whose size is at most $\frac{3n}{4}$, and $E[Z]$ the expected time it takes to locate the desired statistic in an array of size at most $\frac{3n}{4}$. By definition, the latter expectation is at most $T(\frac{3n}{4})$.

As for $E[Y]$, notice that the worst case occurs when the desired statistic lies in the middle at $k = \lfloor n/2 \rfloor$. Then to reduce the array length by at least 25%, the pivot's order (from 1 to n in terms of its relative size in the array) must lie in the interval $[n/4, 3n/4]$ which will happen with probability 1/2. Moreover, the expected number of random pivot selections that are needed before such a pivot is selected is equal to 2. This is because the probability distribution for the number of required selections follows a geometric distribution whose expectation is $1/p$, where p is the probability of reaching a 25% reduction of array size on a single attempt. In this case $p = 1/2$. Thus, the expected number of steps required to reduce the array size by 25% is $2\mathrm{O}(n) = \mathrm{O}(n)$, since each pivot selection is followed by the Partitioning algorithm which requires $\mathrm{O}(n)$ steps. Therefore,

$$T(n) \leq T(3n/4) + \mathrm{O}(n)$$

which by Case 1 of the Master Theorem implies that $T(n) = \mathrm{O}(n)$.

3.2 Analysis of Randomized Quicksort Algorithm

To analyze `Quicksort` we make use of the concept of a **conditional-expectation** random variable: given random variables X and Y, $E[X|Y]$ is a random variable that is observed once a value y for Y is observed, namely the value

$$E[X|Y=y] = \sum_{x \in X} x \cdot p(x|y),$$

which is the expectation of X, conditioned on having observed $Y = y$. We leave the proof of the following proposition as an exercise.

Proposition 3.4. Given random variables X and Y,

$$E[E[X|Y]] = E[X].$$

We now apply Proposition 3.4 to proving an upper bound on the expected running time of `Randomized Quicksort`. Again we assume an array input of size n that is an array a of distinct numbers. Let random variable X denote the running time of `Randomized Quicksort` on input a, and let random variable Y have domain $\{1, \ldots, n\}$, where a sample $i \in \{1, \ldots, n\}$ represents the order (in terms of increasing value) of pivot that is randomly selected at the top level of the recursion tree. Then, for all $i = 1, \ldots, n$,

$$E[X|Y=i] = T(i-1) + T(n-i) + \mathrm{O}(n).$$

In other words, after the i th least element in a is selected as pivot, the Partitioning algorithm requires $\mathrm{O}(n)$ steps, followed by two applications of `Randomized Quicksort`: once on a_{left} whose size equals $i-1$, and once on a_{right} whose size equals $n-i$, each having respective expected running times $T(i-1)$ and $T(n-i)$. Thus, by Proposition 3.4, the fact that $E[X]$ is denoted as $T(n)$, and the fact that each i has a probability of $1/n$ in being chosen, we have

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i) + \mathrm{O}(n)) = \frac{2}{n}\sum_{i=1}^{n} T(i-1) + \mathrm{O}(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + \mathrm{O}(n).$$

We leave it as an exercise to show that this recurrence induces $T(n)$ to grow as $\mathrm{O}(n \log n)$.

4 Minimum-Distance-Pair Problem

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 $\left(\begin{array}{c} n \\ 2 \end{array} \right)$ 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_{left} and X_{right}, and Y_{left} and Y_{right}. For example, the points in X_{left} are those points that fall to the left of the median of the x-coordinate values, while Y_{left} has the same points as X_{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 $\delta_{\text{L}} = d(X_{\text{left}})$ and $\delta_{\text{R}} = d(X_{\text{right}})$. Then letting $\delta = \min(\delta_{\text{L}}, \delta_{\text{R}})$, we see that any two points that either both lie to the left or to the right of the median line, are at least δ apart in distance.

All that remains is to check if their are two points p_1 and p_2 for which $d(p_1, p_2) < \delta$, where, e.g., $p_1 = (x_1, y_1)$ 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 δ-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 δ-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 δ-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 δ 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 δ 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 δ 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 δ 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 δ-strip points, and the comparing of each point in the δ-strip with the next seven points that follow it can both be performed in linear time. Therefore, by Case 2 of the Master theorem, $T(n) = \Theta(n \log n)$.

Strassen's Algorithm for Matrix Multiplication

Given two $n \times n$ matrices A and B, the standard way to compute their product $C = AB$ is to compute entry c_{ij} of C by taking the dot product of row i of A with column j of B. Furthermore, since a dot product requires $\Theta(n)$ operations and there are n^2 entries to compute, we see that the standard approach requires $\Theta(n^3)$ steps.

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 2×2 matrix whose entries are themselves 2×2 matrices.

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

Theorem 4.1. 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 $n/2$ of A whose entries are restricted to columns 1 through $n/2$. Similarly, A_{12} consists of rows 1 through $n/2$ of A whose entries are restricted to columns $n/2+1$ through n. Finally, A_{21} and A_{22} represent the bottom half of A. Thus, A can be written as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Submatrices B_{11}, B_{12}, B_{21}, and B_{22} are defined similarly. Finally, let $\hat{A}$ and $\hat{B}$ be the 2×2 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. Consider the (i,j) entry of $C = AB$. For simplicty 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

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

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 4.2. 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 $\hat{C} = \hat{A}\hat{B}$.

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

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

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.

4.1 Strassen's improvement

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 runnng time to $n^{\log 7}$, 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. $A_7 = a - c$, $B_7 = e + f$, $P_7 = A_7B_7 = (a-c)(e+f) = ae - ce - cf + af$

Example 4.3. Write r, s, t, u as linear combinations of $P_1, \ldots, P_7$.

$$r = ae + bg =$$

$$s = af + bh =$$

$$t = ce + dg =$$

$$u = cf + dh =$$

Exercises

1. Perform the partitioning step of Quicksort on the array $9, 6, 1, 9, 11, 10, 6, 9, 12, 2, 7$, where the pivot is chosen using the median-of-three heuristic.

2. Provide a permutation of the numbers 1-9 so that, when sorted by Quicksort using median-of-three heuristic, the a_{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.

3. 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 $n - 1$.

4. 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.

5. Given n distinct integers, prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to determine both the least and greatest element.

6. 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 at the root level of recursion?

7. 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?

8. For the Median-of-Five Find Statistic algorithm, show that if $n \geq 180$, then at least $n/4$ elements of a are greater than (and respectively less than) or equal to the pivot (i.e. the median of the medians of groups of 5).

9. 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.

10. 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.

11. 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
$$5, 8, 16, 2, 7, 11, 0, 9, 3, 4, 6, 7, 3, 15, 5, 12, 4, 7.$$

12. 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.

13. A **Geometric** random variable, denoted $G(s)$, $0 < s < 1$, is a discrete random variable having domain $\{1, 2, \ldots\}$ and for which $p_i = (1-s)^{i-1}s$. Using the fact that, for all $0 < x < 1$,

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$$

to prove that $E[G(s)] = \frac{1}{s}$. Conclude that when $s \geq 3/4$, as is the case in randomized Find-Statistic when we are attempting to reduce the array by at least 1/4 its size, then $E[G(s)] \leq 4/3$ and so the amount of steps needed to reduce the array size by 1/4 is expected as $4/3\mathrm{O}(n) = \mathrm{O}(n)$.

14. Use the substitution method to show that $T(n) = \mathrm{O}(n \log n)$, where $T(n)$ satisfies the recurrence.

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + C'n$$

where $C' > 0$ is a constant. Hint: use the following result from mathematical analysis. If $f(x)$ is an increasing real-valued function on the interval $[a, b]$, then there exists a real number $\gamma \in [0, 1]$ for which

$$\sum_{i=a}^{b} f(i) = \int_a^b f(x)\ \mathrm{dx} + \gamma(f(b) - f(a)).$$

Conclude that randomized Quicksort has a long-linear expected running time.

15. 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 δ and the equation of the median line.

16. For the matrices A and B in Example 4.2, 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.

17. 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.

18. Use Strassen's algorithm to compute the matrix product

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

19. 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.

20. What is the largest k such that you can multiply 3×3 matrices using k multiplications, then you can multiply matrices in time $o(n^{\log 7})$? Explain and show work.

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

22. 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.

23. 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.

24. 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 $P_1 \times 2^{2\lfloor \frac{n}{2} \rfloor} + (P_3 - P_1 - P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$. Provide the divide-and-conquer recurrence for this algorithm's running time $T(n)$, and use it to determine the running time.

25. 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 $xy = P_1 \times 2^{2\lfloor \frac{n}{2} \rfloor} + (P_3 - P_1 - P_2) \times 2^{\lfloor n/2 \rfloor} + P_2$.

26. 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.

27. 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.

28. 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 asympotic solution for the running time.

29. 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.

30. 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.

31. 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 $a \cup b$.

Exercise Hints and Answers

1. Pivot $= 9$. $a_{\text{left}} = 7, 6, 1, 2, 9, 6$, $a_{\text{right}} = 11, 12, 9, 10$

2. 173924685 is one possible permutation. Verify!

3. 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_i. 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 $n - 1$ comparisions.

4. 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 $\lceil \log n \rceil - 1$ comparisions. This gives a total of $n + \lceil \log n \rceil - 2$ comparisons.

5. 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.

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

7. True for groups of 7, since new recurrence is $T(n) \leq T(\lceil n/7 \rceil) + T(5n/7 + 12) + O(n)$. 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 $T(n) = T(\lceil n/3 \rceil) + T(2n/3 + 6) + O(n)$ which yields log-linear growth in the worst case. This can be verified using the substitution method.

8. $$3n/10 - 9 \geq n/4 \Leftrightarrow (6n - 5n)/20 \geq 9 \Leftrightarrow n \geq 180.$$

9. 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 $n/2$.

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

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

12. 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_{left} and a_{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 $O(n \log q)$.

13. We have

$$E[G(s)] = \sum_{i=1}^{\infty} i \cdot s(1-s)^{i-1} = s\sum_{i=1}^{\infty} i \cdot (1-s)^{i-1}.$$

Then using the fact that, for all $0 < x < 1$,

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$$

we see that

$$E[G(s)] = \frac{s}{(1-(1-s))^2} = \frac{s}{s^2} = \frac{1}{s}.$$

14. Inductive assumption: $T(k) \le Ck \log k$ for all $k < n$ and some constant $C > 0$. Prove that $T(n) \le Cn \log n$.

From the provided hint and performing integration by parts on the integral $\int_0^{n-1} x \log x \text{ dx}$, we see that

$$T(n) = Cn(1-\frac{1}{n})\log(n-1) - \frac{C}{2\ln 2}n(1-\frac{1}{n}) + C'n + \gamma(1-\frac{1}{n})\log(n-1) \le Cn \log n$$

so long as $C \ge 2\ln 2(C' + \gamma)$, and n is sufficiently large.

15. $\delta_L = 1 = d((-1,2),(0,2))$, 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.

16. We have

$$C_{12} = \hat{c}_{12} = \hat{a}_{11}\hat{b}_{12} + \hat{a}_{12}\hat{b}_{22} = A_{11}B_{12} + A_{12}B_{22} =$$

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

$$\begin{pmatrix} -14 & 0 \\ -12 & 10 \end{pmatrix} + \begin{pmatrix} -2 & -1 \\ 5 & 7 \end{pmatrix} = \begin{pmatrix} -16 & -1 \\ -7 & 17 \end{pmatrix},$$

$$C_{21} = \hat{c}_{21} = \hat{a}_{21}\hat{b}_{11} + \hat{a}_{22}\hat{b}_{21} = A_{21}B_{11} + A_{22}B_{21} =$$

$$\begin{pmatrix} 1 & 0 \\ -3 & 2 \end{pmatrix}\begin{pmatrix} -1 & 4 \\ 3 & -3 \end{pmatrix} + \begin{pmatrix} -4 & -2 \\ 1 & -4 \end{pmatrix}\begin{pmatrix} -1 & 2 \\ -2 & -2 \end{pmatrix} =$$
$$\begin{pmatrix} -1 & 4 \\ 9 & -18 \end{pmatrix} + \begin{pmatrix} 8 & -4 \\ 16 & 10 \end{pmatrix} = \begin{pmatrix} 7 & 0 \\ 16 & -8 \end{pmatrix},$$

and

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

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

17. Consider the case where entry (i, j) lies in the upper right qudrant 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

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

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.

18.
$$\begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

19. 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 $\Theta(n^k)$.

20. 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 $k = \lfloor 3^{\log 7} \rfloor$.

21. $\log_{68}(132, 464) \approx 2.795$. 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 $\log 7 \approx 2.8$.

22. 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 $C_{ij} = A_i B_j$. 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 $\Theta(k^2 n^{\log 7})$.

23. ad,bc,$(a + b)(c - d)$

24. $T(n) = 3T(n/2) + O(n)$ yields $T(n) = \Theta(n^{\log 3})$.

25. $x = 111$, $y = 93$, $x_L = 13$, $x_R = 7$, $y_L = 11$, and $y_R = 5$. $P_1 = 143$, $P_2 = 35$, $P_3 = 320$. $(64)(143) + (8)(320 - 143 - 35) + 35 = 10323 = (111)(93)$.

26. 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.

27. Divide the array a into equal-length subarrays a_L and a_R. Let MSS_L denote the MSS of a_L (found by making a recursive call), MSS_R denote the MSS of a_R. Then calculate $\text{MSS}_{\text{middle}}$ 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 MSS_L, MSS_R, and $\text{MSS}_{\text{middle}}$.

28. The running time recurrence satisfies $T(n) = 2T(n/2) + O(n)$.

29. Same algorithm as for MSS of previous problem, but now it is not so easy to compute $\text{MPSS}_{\text{middle}}$, 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 $s = S_L(i) + S_R(j) \le 0$, 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 $\text{MPSS}_{\text{middle}} = s_{\min}$. 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 $T(n) = \Theta(n \log^2 n)$.

30. 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.

31. 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 $m_a \le m \le m_b$. 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 $\mathrm{O}(\log n)$, since it satisfies the recurrence $T(n) = T(n/2) + 1$.