Divide and Conquer Algorithms

Last Updated: January 29th, 2026

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, some divide-and-conquer algorithms have interesting proofs of correctness and for others bounding the number of required steps may seem interesting. In the latter case this involves determining the big-O growth of a function $T(n)$ that satisfies a divide-and-conquer recurrence. Hence, the techniques from the Recurrences 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 usually do not require special data structures for their implementation.

1 Hoare's Quicksort

Definition 1.1. The **median** of an array a of n numbers, denoted median(a), is defined as follows.

Odd n a number $x \in a$ for which $(n-1)/2$ members of a are less than or equal to x and $(n-1)/2$ members of a are greater than or equal to x

Even n two numbers $x, y \in a$ for which for which $(n-2)/2$ members of a are less than or equal to both x and y, and $(n-2)/2$ members of a are greater than or equal to both x and y.

Thus, when n is odd median(a) $= x$, while median(a) $= \{x, y\}$ when n is even.

Example 1.2. Determine the median of $7, 5, 7, 3, 4, 8, 2, 3, 7, 8, 2$, and the medians of $4, 5, 10, 12, 6, 3$.

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. 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 Do one of the following

- 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 if $a[i] < M$, then increment i.
- Else decrement j. //must have $a[j] > M$

While $i < j$.

Once the Partitioning algorithm has partitiioned a into a_{left} and a_{right}, then Quicksort is recursively called on both the left and right 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.

`Quicksort Pseudocode`

Name: `quicksort`

Inputs: a is an array of integers, lower and upper are integers.

Output: none

Side Effect: the subarray a[lower : upper] is sorted.

//Base Case:

//`Insertion Sort` is faster for sufficiently small arrays.

If (upper $-$ lower) $\leq$ SUFFICIENTLY_SMALL, then return `insertion_sort`$(a, \text{lower}, \text{upper})$.

//Recursive case:

index = `partition`$(a, \text{lower}, \text{upper})$.

`quicksort`$(a, \text{lower}, \text{index} - 1)$.

`quicksort`$(a, \text{index} + 1, \text{upper})$.

`Partition Pseudocode`

Name: `partition`

Inputs: a is an array of integers, lower and upper are integers.

Output: index of the final location of pivot M.

Side Effect: $a[i] \leq M$ for all lower $\leq i \leq$ index and $a[i] \geq M$ for all index $\leq i \leq$ upper.

$M = \text{median}(a[\text{lower}], a[(\text{lower} + \text{upper})/2)], a[\text{upper}])$.

//index equals the index of M in a

index = select from$(i = \text{lower}, (\text{lower} + \text{upper})/2, \text{upper})$ according to $a[i] == M$.

swap$(a, \text{index}, \text{upper})$ //swap M with the final element of a[lower : upper] for safe storage

$i = 0$

$j = \text{upper} - 1$

While $i < j$,

    While $a[i] < M$ or $a[j] > M$,
        If $a[i] < M$, then $i{+}{+}$.
        If $a[j] > M$, then $j{-}{-}$.
    swap(a, i, j).

swap(a, i, upper). //move M to its final (and correct) location

Return i. //Return the index of where the pivot is located in a[lower : upper]

Example 1.3. 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.

`Find Statistic Pseudocode`

Name: `find_statistic`

Inputs: a is an array of integers, lower and upper are integers, k is an index that lies between lower and upper.

Output: the index of the location of the kth least statistic of a.

Side Effect: the values stored in a may be rearranged.

//Base Case:

If $(\text{upper} - \text{lower}) \leq 4$, then return `easy_find_statistic`$(a, \text{lower}, \text{upper}, k)$.

//Recursive case:

index = `partition`$(a, \text{lower}, \text{upper})$.

If $k ==$ index, then return index

If $k <$ index, then return `find_statistic`$(a, \text{lower}, \text{index} - 1, k)$.

Return `find_statistic`$(a, \text{index} + 1, \text{upper}, k)$. //must have $k >$ index

Example 2.1. If initially (i.e. lower = 0 and upper = 10) $a = 105, 48, 27, 58, 13, 91, 39, 119, 79, 63, 85$ and $k = 6$, will the least statistic be found at the root level? If not, provide the updated values for lower and upper.

Solution.

As it currently stands, the `Find Statistic` algorithm described above does *not* have $O(n)$ worst-case running time. This is because the median-of-three pivot selection may lead to poor splitting of the data in the sense that the left (respectively, right) array may be much larger (or smaller) than the right (respectively, left) array. Similar to `Quicksort` if the median where selected as pivot each time, one would have the recurrence

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

which implies (by the Master Theorem) that $T(n) = \Theta(n)$. One seemingly clever idea is to compute the the pivot/median by making a recursive call to `Find Statistic` with input

$$k = \frac{\text{lower} + \text{upper}}{2},$$

since that statistic represents the median of a! The problem with this approach is that the recursive call

$$\text{find\_statistic}(a, \text{lower}, \text{upper}, \frac{\text{lower} + \text{upper}}{2})$$

does not reduce the search range (i.e., the values of lower and upper), and so the algorithm will go into an infinite loop!

Thus, when computing the median, we must either reduce the search range or, equivalently, use a different array of a smaller size. We choose the latter approach. To simplify matters, we assume that the search range is the entirety of a. The idea is to compute the median of only some of the elements of a, and yet hope that the answer still yields a sufficiently balanced splitting of a into left and right subarrays. 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, in the `Partition` algorithm, replace the lines

$$M = \text{median}(a[\text{lower}], a[(\text{lower} + \text{upper})/2)], a[\text{upper}])$$

$$\text{index} = \text{select from}(i = \text{lower}, (\text{lower} + \text{upper})/2, \text{upper}) \text{ according to } a[i] == M$$

with

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

$$M = a_{\text{medians}}[\text{index}]$$

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

Example 2.2. 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 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.

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

Theorem 3.1. 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

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

Submatrices B_{11}, B_{12}, B_{21}, and B_{22} are defined similarly. Finally, let $\hat{A}$ and $\hat{B}$ be the 2×2 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 3.2. 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 $\hat{C} = \hat{A}\hat{B}$.

Solution. We have

$$C = AB = \begin{pmatrix} -11 & 6 & -16 & -1 \\ -9 & 14 & -7 & 17 \\ 7 & 0 & 9 & -2 \\ 16 & -8 & 13 & -25 \end{pmatrix}.$$

Moreover,

$$\hat{C}_{11} = A_{11}B_{11} + A_{12}B_{21} = \begin{pmatrix} -2 & -4 \\ 4 & -2 \end{pmatrix}\begin{pmatrix} -1 & 4 \\ 3 & -3 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -3 & 1 \end{pmatrix}\begin{pmatrix} -1 & 2 \\ -2 & -2 \end{pmatrix} = \begin{pmatrix} -10 & 4 \\ -10 & 22 \end{pmatrix} + \begin{pmatrix} -1 & 2 \\ 1 & -8 \end{pmatrix} = \begin{pmatrix} -11 & 6 \\ -9 & 14 \end{pmatrix} = C_{11}.$$

We leave it as an exercise to verify the other three equations:

$$A_{11}B_{12} + A_{12}B_{22} = C_{12},$$

$$A_{21}B_{11} + A_{22}B_{21} = C_{21},$$

and

$$A_{21}B_{12} + A_{22}B_{22} = C_{22}.$$

□

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 $n/2 \times n/2$ upper left quadrant of A, b the $n/2 \times n/2$ 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 the $n/2 \times n/2$ products ae, bg, af, bh, ce, dg, cf, and dh. Finally, these products are added to obtain

$$\begin{aligned} r &= ae + bg, \\ s &= af + bh, \\ t &= ce + dg, \\ u &= cf + dh. \end{aligned}$$

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.

3.1 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. $A_7 = a - c,\ B_7 = e + f,\ P_7 = A_7B_7 = (a-c)(e+f) = ae - ce - cf + af$

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

Example 3.3. Write r, s, t, u as linear combinations of $P_1, \ldots, P_7$.

Solution (verify!).

$$r = ae + bg = P_5 + P_6 - P_2 + P_4$$

$$ae + ah + de + dh + bg + bh - dg - dh - ah - bh$$
$$dg - de$$

$$s = af + bh = P_1 + P_2$$

$$t = ce + dg = P_3 + P_4$$

$$u = cf + dh = -P_7 + P_5 - P_3 + P_1$$

Divide and Conquer Core 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. Perform the partitioning step of `Quicksort` on the array $6, 2, 7, 6, 7, 10, 10, 1, 9, 8, 4, 3, 5$, where the pivot is chosen using the median-of-three heuristic.

3. 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?

4. In the `Median-of-Five Find Statistic` recall that 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 $n \geq 180$. Re-write the above chain of inequalities if the algorithm used groups of seven rather than groups of five. Provide a new bound for n that must be satisfied in order for the final inequality (i.e. $\geq n/4$) to hold.

5. Use Strassen's algorithm to compute the matrix product
$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$
Make sure to compute integer values for P_1 through P_7 and apply them to Strassen's formulas to determine the matrix product.

6. Consider Karatsuba's algorithm which we'll call `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. Justify all aspects of your recurrence. Hint: for simplicity you may assume that n is even.

7. For the two binary integers $x = 11011110$ and $y = 10111011$, determine the top-level values of P_1, P_2, and P_3, and verify that $xy = P_1 \times 2^{2\lfloor\frac{n}{2}\rfloor} + (P_3 - P_1 - P_2) \times 2^{\lfloor n/2\rfloor} + P_2$.

8. Verify that Karatsuba's 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 the binary representations of both x and y have even length.

9. 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 $a[i] + a[i+1] + a[i+2] + \cdots + a[j]$. An instance of the `Maximum Subsequence Sum (MSS)` problem is an array a of n integers, and the problem is to determine the maximum sum of any of the

subsequences of a. The following divide-and-conquer algorithm may be used to find the MSS. Divide the array a into equal-length subarrays a_L and a_R. Let MSS_L denote the MSS of a_L, MSS_R denote the MSS of a_R. Then calculate $\text{MSS}_{\text{middle}}$ 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 $\text{MSS}_{\text{middle}}$. Provide a divide-and-conquer recurrence relation that describes the running time $T(n)$ of the `MSS` algorithm and use the Master Theorem to provide its big-Θ running time. Make sure to defend all aspects of the recurrence.

10. For the `MSS` divide-and-conquer algorithm described in Exercise 9 provide the entire recursion tree for instance $a = 4, -3, 2, 1, -4, 5, -3, 2$. Label each node with the problem instance that is being solved at that stage of the recursion. Also, next to each node write the MSS its value and (in case the node is interal) indicate if the MSS comes from the left subproblem, right subproblem, or middle of both subproblems.

11. An instance of the `Minimum Positive Subsequence Sum (MPSS)` problem is an array a of n integers, and the problem is to determine the *minimum positive* subsequence sum of a (see Exercise 9 for the definition of *subsequence sum*). The following divide-and-conquer algorithm is similar to the `MSS` algorithm described in Exercise 9. The only difference is in how to calculate $\text{MPSS}_{\text{middle}}$. Since both positive and negative sums may be useful for finding a small positive subsequence sum, we must compute all the subsequence sums of a_L that end with the last element of a_L and sort them in increasing order and call the resulting array b_L. Similarly, compute all the subsequence sums of a_R that begin with the first element of a_R and sort them in *decreasing* order and call the resulting array b_R. Now initialize variable mpss to ∞. Initialize two index variables i and j to 0. Repeat the following until termination. If $0 < b_L[i] + b_R[j]$, then if $b_L[i] + b_R[j] <$ mpss, assign mpss the value $b_L[i] + b_R[j]$. In either case increment j if possible (or terminate the algorithm in case $j = |b_R|$). Otherwise, $b_L[i] + b_R[j] \leq 0$ and so we must increment i if possible (or terminate the algorithm in case $i = |b_L|$). Provide a divide-and-conquer recurrence relation that describes the running time $T(n)$ of the `MPSS` algorithm and use the Master Theorem to provide its big-Θ running time. Make sure to defend all aspects of the recurrence.

12. For the `MPSS` algorithm described in Exercise 11 provide each of the left sums that must be sorted and well as each of the right sums that must be sorted in order to compute $\text{MPSS}_{\text{Middle}}$ at the top level of recursion (only!) for the array

$$a = 48, -37, 32, -33, 70, -64, 46, -34, 45, -72, 34, -52.$$

Then compute both b_L and b_R. Show each iteration of the algorithm until the algorithm is forced to terminate according to the termination conditions described in the algorithm. Do so by making a table whose columns are i, j, $b_L[i]$, $b_R[j]$, $b_L[i] + b_R[j]$, and the current mpss value. Finally, indicate the beginning and ending indices of the subsequence of a that sums to the mpss.

Solutions to Divide and Conquer Core Exercises

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

2. $a_{\text{left}} = 5, 2, 3, 4, 1$, $a_{\text{right}} = 10, 7, 9, 8, 6, 7, 10$, while pivot 6 is permanently placed at index 5.

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

4. With groups of 7, we replace $\lceil \frac{n}{5} \rceil$ with $\lceil \frac{n}{7} \rceil$ since there are $\lceil \frac{n}{7} \rceil$ groups and hence $\lceil \frac{n}{7} \rceil$ medians that form the "medians of medians" array whose median serves as the pivot. Also, the leading 3 is replaced by 4 since, e.g., if the pivot is greater than or equal to the median of some group, then it is also greater than or equal to three other members of the group, and we have $1+3 = 4$. Therefore, we have

$$4(\lfloor \frac{1}{2} \lceil \frac{n}{7} \rceil \rfloor - 2) \geq 4(\frac{1}{2} \cdot \frac{n}{7} - 3) = \frac{2n}{7} - 12 \geq n/4$$

   so long as $n \geq (28)(12) = 336$.

5. We have $P_1 = 6, P_2 = 8, P_3 = 72, P_4 = -10, P_5 = 48, P_6 = -12, P_7 = -84$ and so

$$r = ae + bg = P_5 + P_6 - P_2 + P_4 = 18$$

$$s = af + bh = P_1 + P_2 = 14$$

$$t = ce + dg = P_3 + P_4 = 62$$

$$u = cf + dh = -P_7 + P_5 - P_3 + P_1 = 66$$

   which gives the product matrix

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

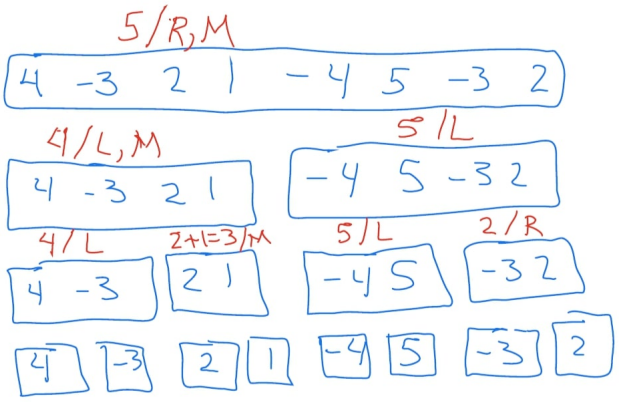
6. $T(n) = 3T(n/2) + \mathrm{O}(n)$ yields $T(n) = \Theta(n^{\log 3})$. There are three subproblems, P_1, P_2, and P_3 each of size $n/2$. As for $f(n)$, both P_1 and P_2 can be formed in $\mathrm{O}(1)$ steps, while P_3 can be formed with a single $\mathrm{O}(n)$ addition. Finally, the output requires four additions/subtractions, each requiring $\mathrm{O}(n)$ addition, while multiplying by $2^{\frac{n}{2}}$ also requires $\mathrm{O}(n)$ steps since it requires shifting bits $\frac{n}{2}$ places to the left.

7. $x = 222$, $y = 187$, $x_L = 13$, $x_R = 14$, $y_L = 11$, and $y_R = 11$. $P_1 = 143$, $P_2 = 154$, $P_3 = 594$. $(256)(143) + (16)(594 - 143 - 154) + 154 = 41514 = (222)(187)$.

8. 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.

9. The running time recurrence satisfies $T(n) = 2T(n/2) + \mathrm{O}(n)$. Why? Therefore, $T(n) = \Theta(n \log n)$.

10. We have the following recursion tree.



11. The left sums are $-64, 6, -27, 5, -32, 16$, while $b_L = -64, -32, -27, 5, 6, 16$. The right sums are $46, 12, 57, -15, 19, -33$, while $b_R = 57, 46, 19, 12, -15. - 33$.

i	j	$b_L[i]$	$b_R[j]$	$b_L[i] + b_R[j]$	mpss
0	0	-64	57	-12	∞
1	0	-32	57	25	25
1	1	-32	46	14	14
1	2	-32	19	-13	14
2	2	-27	19	-8	14
3	2	5	19	24	14
3	3	5	12	17	14
3	4	5	-15	-10	14
4	4	6	-15	-9	14
5	4	16	-15	1	1
5	5	16	-33	-17	1

Therefore, the mpss equals 1 and represents the sum from $a[0]$ to $a[9]$.

12. The running time recurrence satisfies $T(n) = 2T(n/2) + \mathrm{O}(n \log n)$. Why? $T(n) = \Theta(n \log^2 n)$.

Additional Exercises

A. 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.

B. 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 $n-1$.

C. 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.

D. 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.

E. For the `Median-of-Five Find Statistic` algorithm, does the algorithm still run in linear time if groups of three are used instead of groups of five? Explain and show work.

F. Explain how the Median-of-Five Find Statistic Algorithm could be used to modify Hoare's `Quicksort` so that it requires $O(n \log n)$ steps in the worst-case.

G. 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

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

H. 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.

I. For the matrices A and B in Example 3.2, 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.

J. 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.

K. 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.

L. 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.

M. 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?

N. 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.

O. 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.

P. 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.

Q. 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 $a \cup b$.

Solutions to Additional Exercises

A. 173924685 is one possible permutation. Verify!

B. 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.

C. 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.

D. 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.

E. Not true for groups of 3, since, in the worst-case, the new recurrence is $T(n) = T(\lceil n/3 \rceil) + T(2n/3 + 6) + \mathrm{O}(n)$ which yields log-linear growth in the worst case. This can be verified using the substitution method.

F. 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 $T(n) = 2T(n/2) + \mathrm{O}(n)$, since both subarrays are now guaranteed to have size $n/2$.

G. 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.

H. 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 $T(n) = 2T(n/2) + \mathrm{O}(n)$. 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 $\mathrm{O}(n \log q)$ 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 $q\mathrm{O}(n/q) = \mathrm{O}(n)$ running time. Therefore the total running time is $\mathrm{O}(n \log q)$.

I. 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 $C = AB$.

J. 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.

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

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

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

N. 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 $\Theta(k^2 n^{\log 7})$.

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

P. 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.

Q. 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 $\mathrm{O}(\log n)$, since it satisfies the recurrence $T(n) = T(n/2) + 1$.