2.12. How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume n is a power of 2.

```
function f(n)
  if n > 1:
    print_line(''still going'')
    f(n/2)
    f(n/2)
```

2.13. A binary tree is *full* if all of its vertices have either zero or two children. Let B_n denote the number of full binary trees with n vertices.

(a) By drawing out all full binary trees with 3, 5, or 7 vertices, determine the exact values of B_3, B_5, and B_7. Why have we left out even numbers of vertices, like B_4?

(b) For general n, derive a recurrence relation for B_n.

(c) Show by induction that B_n is $\Omega(2^n)$.

2.14. You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

2.15. In our median-finding algorithm (Section 2.4), a basic primitive is the `split` operation, which takes as input an array S and a value v and then divides S into three sets: the elements less than v, the elements equal to v, and the elements greater than v. Show how to implement this `split` operation *in place*, that is, without allocating new memory.

2.16. You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are *not* given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

2.17. Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

2.18. Consider the task of searching a sorted array $A[1 \ldots n]$ for a given element x: a task we usually perform by binary search in time $O(\log n)$. Show that any algorithm that accesses the array only via comparisons (that is, by asking questions of the form "is $A[i] \leq z$?"), must take $\Omega(\log n)$ steps.

2.19. *A k-way merge operation.* Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(a) Here's one strategy: Using the `merge` procedure from Section 2.3, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

(b) Give a more efficient solution to this problem, using divide-and-conquer.

2.20. Show that any array of integers $x[1 \ldots n]$ can be sorted in $O(n + M)$ time, where

$$M = \max_i x_i - \min_i x_i.$$

For small M, this is linear time: why doesn't the $\Omega(n \log n)$ lower bound apply in this case?

2.21. *Mean and median.* One of the most basic tasks in statistics is to summarize a set of observations $\{x_1, x_2, \ldots, x_n\} \subseteq \mathbb{R}$ by a single number. Two popular choices for this summary statistic are:

- The median, which we'll call μ_1
- The mean, which we'll call μ_2

(a) Show that the median is the value of μ that minimizes the function

$$\sum_i |x_i - \mu|.$$

You can assume for simplicity that n is odd. (*Hint:* Show that for any $\mu \neq \mu_1$, the function decreases if you move μ either slightly to the left or slightly to the right.)

(b) Show that the mean is the value of μ that minimizes the function

$$\sum_i (x_i - \mu)^2.$$

One way to do this is by calculus. Another method is to prove that for any $\mu \in \mathbb{R}$,

$$\sum_i (x_i - \mu)^2 \;=\; \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2.$$

Notice how the function for μ_2 penalizes points that are far from μ much more heavily than the function for μ_1. Thus μ_2 tries much harder to be close to *all* the observations. This might sound like a good thing at some level, but it is statistically undesirable because just a few outliers can severely throw off the estimate of μ_2. It is therefore sometimes said that μ_1 is a more robust estimator than μ_2. Worse than either of them, however, is μ_∞, the value of μ that minimizes the function

$$\max_i |x_i - \mu|.$$

(c) Show that μ_∞ can be computed in $O(n)$ time (assuming the numbers x_i are small enough that basic arithmetic operations on them take unit time).

2.22. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

2.23. An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time.

(a) Show how to solve this problem in $O(n \log n)$ time. (*Hint:* Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (*Hint:* Here's another divide-and-conquer approach:

- Pair up the elements of A arbitrarily, to get $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if and only if A does.)

2.24. On page 62 there is a high-level description of the quicksort algorithm.

(a) Write down the pseudocode for quicksort.

(b) Show that its *worst-case* running time on an array of size n is $\Theta(n^2)$.

(c) Show that its *expected* running time satisfies the recurrence relation

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

Then, show that the solution to this recurrence is $O(n \log n)$.

2.25. In Section 2.1 we described an algorithm that multiplies two n-bit binary integers x and y in time n^a, where $a = \log_2 3$. Call this procedure `fastmultiply`(x, y).

(a) We want to convert the decimal integer 10^n (a 1 followed by n zeros) into binary. Here is the algorithm (assume n is a power of 2):

```
function pwr2bin(n)
  if n = 1:  return 1010_2
  else:
    z =???
    return fastmultiply(z, z)
```

Fill in the missing details. Then give a recurrence relation for the running time of the algorithm, and solve the recurrence.

(b) Next, we want to convert any decimal integer x with n digits (where n is a power of 2) into binary. The algorithm is the following:

```
function dec2bin(x)
  if n = 1:  return binary[x]
  else:
    split x into two decimal numbers x_L, x_R with n/2 digits each
    return ???
```

Here `binary`$[\cdot]$ is a vector that contains the binary representation of all one-digit integers. That is, `binary`$[0] = 0_2$, `binary`$[1] = 1_2$, up to `binary`$[9] = 1001_2$. Assume that a lookup in `binary` takes $O(1)$ time.

Fill in the missing details. Once again, give a recurrence for the running time of the algorithm, and solve it.

2.26. Professor F. Lake tells his class that it is asymptotically faster to square an n-bit integer than to multiply two n-bit integers. Should they believe him?

2.27. The *square* of a matrix A is its product with itself, AA.

(a) Show that five multiplications are sufficient to compute the square of a 2×2 matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

> "Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$."

(c) In fact, squaring matrices is no easier than matrix multiplication. In this part, you will show that if $n \times n$ matrices can be squared in time $S(n) = O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

i. Given two $n \times n$ matrices A and B, show that the matrix $AB + BA$ can be computed in time $3S(n) + O(n^2)$.

ii. Given two $n \times n$ matrices X and Y, define the $2n \times 2n$ matrices A and B as follows:

$$A = \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix}.$$

What is $AB + BA$, in terms of X and Y?

iii. Using (i) and (ii), argue that the product XY can be computed in time $3S(2n) + O(n^2)$. Conclude that matrix multiplication takes time $O(n^c)$.

2.28. The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $\begin{bmatrix} 1 \end{bmatrix}$
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

2.29. Suppose we want to evaluate the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ at point x.

(a) Show that the following simple routine, known as *Horner's rule*, does the job and leaves the answer in z.

```
z = a_n
for i = n − 1 downto 0:
    z = zx + a_i
```

(b) How many additions and multiplications does this routine use, as a function of n? Can you find a polynomial for which an alternative method is substantially better?

2.30. This problem illustrates how to do the Fourier Transform (FT) in modular arithmetic, for example, modulo 7.

(a) There is a number ω such that all the powers $\omega, \omega^2, \ldots, \omega^6$ are distinct (modulo 7). Find this ω, and show that $\omega + \omega^2 + \cdots + \omega^6 = 0$. (Interestingly, for any prime modulus there is such a number.)

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 1, 1, 5, 2)$ modulo 7; that is, multiply this vector by the matrix $M_6(\omega)$, for the value of ω you found earlier. In the matrix multiplication, all calculations should be performed modulo 7.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 7.)

(d) Now show how to multiply the polynomials $x^2 + x + 1$ and $x^3 + 2x - 1$ using the FT modulo 7.

2.31. In Section 1.2.3, we studied Euclid's algorithm for computing the *greatest common divisor* (gcd) of two positive integers: the largest integer which divides them both. Here we will look at an alternative algorithm based on divide-and-conquer.

(a) Show that the following rule is true.

$$\gcd(a,b) = \begin{cases} 2\gcd(a/2,b/2) & \text{if } a,b \text{ are even} \\ \gcd(a,b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ \gcd((a-b)/2,b) & \text{if } a,b \text{ are odd} \end{cases}$$

(b) Give an efficient divide-and-conquer algorithm for greatest common divisor.

(c) How does the efficiency of your algorithm compare to Euclid's algorithm if a and b are n-bit integers? (In particular, since n might be large you cannot assume that basic arithmetic operations like addition take constant time.)

2.32. In this problem we will develop a divide-and-conquer algorithm for the following geometric task.

CLOSEST PAIR
Input: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$
Output: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

is minimized.

For simplicity, assume that n is a power of two, and that all the x-coordinates x_i are distinct, as are the y-coordinates.

Here's a high-level overview of the algorithm:

- Find a value x for which exactly half the points have $x_i < x$, and half have $x_i > x$. On this basis, split the points into two groups, L and R.
- Recursively find the closest pair in L and in R. Say these pairs are $p_L, q_L \in L$ and $p_R, q_R \in R$, with distances d_L and d_R respectively. Let d be the smaller of these two distances.
- It remains to be seen whether there is a point in L and a point in R that are less than distance d apart from each other. To this end, discard all points with $x_i < x - d$ or $x_i > x + d$ and sort the remaining points by y-coordinate.
- Now, go through this sorted list, and for each point, compute its distance to the *seven* subsequent points in the list. Let p_M, q_M be the closest pair found in this way.
- The answer is one of the three pairs $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$, whichever is closest.

(a) In order to prove the correctness of this algorithm, start by showing the following property: any square of size $d \times d$ in the plane contains at most four points of L.

(b) Now show that the algorithm is correct. The only case which needs careful consideration is when the closest pair is split between L and R.

(c) Write down the pseudocode for the algorithm, and show that its running time is given by the recurrence:

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

Show that the solution to this recurrence is $O(n \log^2 n)$.

(d) Can you bring the running time down to $O(n \log n)$?