Question 1. Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in asymptotic notation) and which would you choose?

Question 2 (K&T 5.1)**.** You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we define to be the nth smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries. If it makes things easier, you may assume that n is a power of two.

Question 3. Recall that the *Fibonacci numbers* F_n are defined recursively by

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \end{aligned}$$

with the first few given by $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ You may assume the following facts:

- F_n is $O(\phi^n)$ (where ϕ is the golden ratio, $(1+\sqrt{5})/2$).
- In addition to their definition, Fibonacci numbers satisfy the following recurrences:

$$\begin{aligned} F_{2n+1} &= F_n^2 + F_{n+1}^2 \\ F_{2n} &= 2F_nF_{n+1} - F_n^2 \end{aligned}$$

3

Design an efficient algorithm to compute F_n, and analyze its running time. Be careful to include the time needed for any multiplications and additions; since the algorithm may need to deal with very large numbers, you may *not* assume that arithmetic operations take $O(1)$ time.

Question 4. 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 $A[i] > A[j]$. You should think of the array elements as, say, JPEG files. However, you *can* answer questions of the form $A[i] = A[j]$ in $O(1)$ time.

(a) Show how to solve this problem in $O(n \log n)$ time. Make sure to prove that your algorithm is correct (via induction) and give a recurrence relation for the running time of your algorithm.

4a

(b) (**Extra credit**) Can you give a linear-time algorithm?

Convolutions and the FFT

You are expecting to receive a Very Important Message over a network, encoded as a sequence of bits. The message will only be sent once. Receiving the message is so important that you decide to redundantly have *two* computers both listening for the message, just in case one of them doesn't work.

4b

This turns out to be an excellent idea; however, you only have the idea at the last minute. When the message starts to arrive, computer A is listening, but you have not quite finished setting up computer B! So computer B does not record some beginning portion of the message. You do finally get computer B set up and it starts recording the message somewhere in the middle. And it's a good thing you do, because a few seconds later, computer A crashes! Thankfully computer B continues to work, and records the rest of the message.

Here, then, is the situation: both computers recorded only part of the message. Computer A recorded from the beginning of the message to somewhere in the middle, and computer B recorded from a different point in the middle to the end. The bits recorded by computer A and computer B overlap, but you have no way to know how many bits are in the overlapping portion, since you do not know how long the message is supposed to be, and network transmission speeds are variable enough that you have no way to know how many bits were transmitted between the time when B started recording and the time when A crashed.

To make things worse, there can be occasional transmission errors, where individual bits are flipped. So even during the portion of the message that both computers were recording, there can be positions where computer A

recorded a 0 but computer B recorded a 1 (for example, A might have correctly recorded the intended bit, but a glitch caused computer B to record the incorrect bit). The message uses an error-correcting code, so you will be able to fix these incorrect bits—but not until you have reconstructed the whole message!

You need to find the correct *alignment*, defined as the index of the bit where computer B started recording. So, for example, if the first bit recorded by computer B was the 973rd bit in the message, the correct alignment would be 972. It is also useful to be able to talk about the *overlap*, defined as the number of bits the two recordings have in common. The alignment and overlap are related by the simple equation

$$\mathit{alignment} + \mathit{overlap} = |A|,$$

where $|A|$ denotes the number of bits in the portion of the message recorded by computer A.

With no way to deduce the correct alignment, you decide to simply try *all possible* alignments and find the one that gives the best match between the overlapping portions of A's bits and B's bits. Because of the occasional errors, the overlap will probably never be perfect, but the hope is that many more bits will correspond with the correct alignment than with any incorrect alignment.[1]

[1] Of course it is easy to imagine scenarios where this does not work—for example, if the middle of the message contains a very long sequence like 101010101010 . . . then many different alignments could all match, but given that no one would bother to send a message with so much redundancy, it is reasonable to assume that the correct alignment will correspond to the best match.

Formally, let $A = a_0a_1a_2 \ldots a_{n-1}$ be the sequence of bits recorded by computer A, and let $B = b_0b_1b_2 \ldots b_{n-1}$ be those recorded by computer B. (For simplicity we assume that A and B have the same length, but it does not really matter.) An alignment of i means that a_i is matched with b_0, a_{i+1} with b_1, . . . and in general a_{i+k} is matched with b_k. We define the *fit* of a given alignment as the average number of mismatches per bit in the overlapping portion, that is,

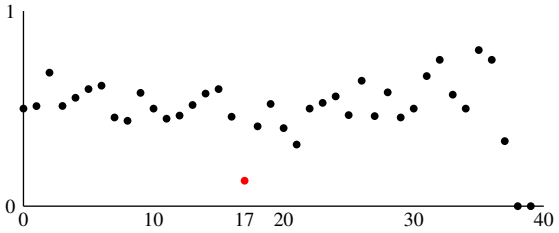
$$\mathit{fit}(i) = \frac{1}{n-i} \sum_{k=0}^{n-1-i} |a_{i+k} - b_k|.$$

For example, suppose A and B have 40 bits each, and are given by

$$A = 1100101101100111100010011000001011111100$$
$$B = 0001000100001101101110011101010001001111$$

With an alignment of, say, 35, A and B overlap by 5 bits, namely, the 11100 at the end of A overlaps with the 00010 at the beginning of B. Four out of five of these bits do not match, giving a fit score of $4/5 = 0.8$. Note that the fit score will always be between 0 and 1. A fit score of 0 means that the sequences match perfectly; a fit score of 1 means that the overlapping portions are exactly inverted from each other, with the first having a 0 whenever the second has a 1, and vice versa. On average, we would expect that two randomly chosen sequences of bits will have a fit score of 0.5 relative to each other.

The absolute best fit between A and B is at alignments 38 and 39: namely, those alignments give a fit of zero, since the two 0 bits at the end of A match perfectly with the two 0 bits at the beginning of B. However, there is a 50% chance that the sequences will align perfectly with an overlap of one, and this is unlikely to actually be the correct alignment. So we ignore alignments near the end like this (specifically, let's say that we will ignore any alignment with 10 or fewer bits of overlap). Making a graph with alignment on the x-axis and fit on the y-axis, an alignment of 17 clearly jumps out as giving the best (lowest) fit value:



Writing A and B underneath each other using this alignment, we can see that they do indeed appear to match very well, with only a few differences; the average number of disagreements per bit is very low:

```
11001011011001111000100110000011011111100
                  00010001000011011011100111010100010011111
```

On the other hand, if we pick another alignment (say, 12):

```
11001011011001111000100110000011011111100
                    00010001000011011011100111010100010011111
```

we can see that A and B do not match very well; the average number of disagreements per bit is relatively high.

Question 5. Before even thinking about designing an algorithm, you decide to do some quick back-of-the-envelope calculations. You note the following facts:

- Each part of the message is about 1GB in length, that is, each part contains $n \approx 2^{33}$ bits.
- Your computer can perform about 1 billion operations per second.

You then imagine using algorithms with various running times and calculate how long they would take to run.

(a) If your algorithm required exactly n^2 operations, approximately how long will it take to run? Express your answer in appropriate, human-comprehensible units (*e.g.* say "10 hours", not "36000 seconds").

(b) If your algorithm required exactly $n \log_2 n$ operations, approximately how long will it take to run?

Question 6. Design and analyze an efficient algorithm which, given two length-n sequences of bits, finds the alignment with the best fit value. Note that "design and analyze" means to describe the algorithm, prove/justify why it is correct, and analyze its asymptotic running time.

6a

6b