To analyze the last line, we use the fact that for any $(2n)^{\text{th}}$ root of unity $\omega \neq 1$, we have $\sum_{s=0}^{2n-1} \omega^s = 0$. This is simply because ω is by definition a root of $x^{2n} - 1 = 0$; since $x^{2n} - 1 = (x-1)(\sum_{t=0}^{2n-1} x^t)$ and $\omega \neq 1$, it follows that ω is also a root of $(\sum_{t=0}^{2n-1} x^t)$.

Thus the only term of the last line's outer sum that is not equal to 0 is for c_t such that $\omega_{t+j,2n} = 1$; and this happens if $t+j$ is a multiple of $2n$, that is, if $t = 2n - j$. For this value, $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. So we get that $D(\omega_{j,2n}) = 2nc_{2n-j}$. Evaluating the polynomial $D(x)$ at the $(2n)^{\text{th}}$ roots of unity thus gives us the coeffients of the polynomial $C(x)$ in reverse order (multiplied by $2n$ each). We sum this up as follows.

(5.14) *For any polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$, and corresponding polynomial $D(x) = \sum_{s=0}^{2n-1} C(\omega_{s,2n}) x^s$, we have that $c_s = \frac{1}{2n} D(\omega_{2n-s,2n})$.*

We can do all the evaluations of the values $D(\omega_{2n-s,2n})$ in $O(n \log n)$ operations using the divide-and-conquer approach developed for step (i).

And this wraps everything up: we reconstruct the polynomial C from its values on the $(2n)^{\text{th}}$ roots of unity, and then the coefficients of C are the coordinates in the convolution vector $c = a * b$ that we were originally seeking.

In summary, we have shown the following.

(5.15) *Using the Fast Fourier Transform to determine the product polynomial $C(x)$, we can compute the convolution of the original vectors a and b in $O(n \log n)$ time.*

Solved Exercises

Solved Exercise 1

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is *unimodal*: For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.)

You'd like to find the "peak entry" p without having to read the entire array—in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A.

Solution Let's start with a general discussion on how to achieve a running time of $O(\log n)$ and then come back to the specific problem here. If one needs to compute something using only $O(\log n)$ operations, a useful strategy that we discussed in Chapter 2 is to perform a constant amount of work, throw away half the input, and continue recursively on what's left. This was the idea, for example, behind the $O(\log n)$ running time for binary search.

We can view this as a divide-and-conquer approach: for some constant $c > 0$, we perform at most c operations and then continue recursively on an input of size at most $n/2$. As in the chapter, we will assume that the recursion "bottoms out" when $n = 2$, performing at most c operations to finish the computation. If $T(n)$ denotes the running time on an input of size n, then we have the recurrence

(5.16)

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

when $n > 2$*, and*

$$T(2) \leq c.$$

It is not hard to solve this recurrence by unrolling it, as follows.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n, which takes time at most c plus the time spent in all subsequent recursive calls. The next level has one problem of size at most $n/2$, which contributes another c, and the level after that has one problem of size at most $n/4$, which contributes yet another c.
- *Identifying a pattern:* No matter how many levels we continue, each level will have just one problem: level j has a single problem of size at most $n/2^j$, which contributes c to the running time, independent of j.
- *Summing over all levels of recursion:* Each level of the recursion is contributing at most c operations, and it takes $\log_2 n$ levels of recursion to reduce n to 2. Thus the total running time is at most c times the number of levels of recursion, which is at most $c \log_2 n = O(\log n)$.

We can also do this by partial substitution. Suppose we guess that $T(n) \leq k \log_b n$, where we don't know k or b. Assuming that this holds for smaller values of n in an inductive argument, we would have

$$\begin{aligned} T(n) &\leq T(n/2) + c \\ &\leq k \log_b(n/2) + c \\ &= k \log_b n - k \log_b 2 + c. \end{aligned}$$

The first term on the right is exactly what we want, so we just need to choose k and b to negate the added c at the end. This we can do by setting $b = 2$ and $k = c$, so that $k \log_b 2 = c \log_2 2 = c$. Hence we end up with the solution $T(n) \leq c \log_2 n$, which is exactly what we got by unrolling the recurrence.

Finally, we should mention that one can get an $O(\log n)$ running time, by essentially the same reasoning, in the more general case when each level of the recursion throws away any constant fraction of the input, transforming an instance of size n to one of size at most an, for some constant $a < 1$. It now takes at most $\log_{1/a} n$ levels of recursion to reduce n down to a constant size, and each level of recursion involves at most c operations.

Now let's get back to the problem at hand. If we wanted to set ourselves up to use (5.16), we could probe the midpoint of the array and try to determine whether the "peak entry" p lies before or after this midpoint.

So suppose we look at the value $A[n/2]$. From this value alone, we can't tell whether p lies before or after $n/2$, since we need to know whether entry $n/2$ is sitting on an "up-slope" or on a "down-slope." So we also look at the values $A[n/2 - 1]$ and $A[n/2 + 1]$. There are now three possibilities.

- If $A[n/2 - 1] < A[n/2] < A[n/2 + 1]$, then entry $n/2$ must come strictly before p, and so we can continue recursively on entries $n/2 + 1$ through n.
- If $A[n/2 - 1] > A[n/2] > A[n/2 + 1]$, then entry $n/2$ must come strictly after p, and so we can continue recursively on entries 1 through $n/2 - 1$.
- Finally, if $A[n/2]$ is larger than both $A[n/2 - 1]$ and $A[n/2 + 1]$, we are done: the peak entry is in fact equal to $n/2$ in this case.

In all these cases, we perform at most three probes of the array A and reduce the problem to one of at most half the size. Thus we can apply (5.16) to conclude that the running time is $O(\log n)$.

Solved Exercise 2

You're consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. A typical instance of the problem is the following. They're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1,000 shares on some day and sell all these shares on some (later) day. They want to know: When should they have bought and when should they have sold in order to have made as much money as possible? (If