1.4 A Case Study in Algorithm Analysis

Having presented the general framework for describing and analyzing algorithms, we now present a case study in algorithm analysis to make this discussion more concrete. Specifically, we show how to use the big-Oh notation to analyze three algorithms that solve the same problem but have different running times.

The problem we focus on is one that is reportedly often used as a job interview question by major software and Internet companies—the ***maximum subarray problem***. In this problem, we are given an array of positive and negative integers and asked to find the subarray whose elements have the largest sum. That is, given

$$A = [a_1, a_2, \ldots, a_n],$$

find the indices j and k that maximize the sum

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.$$

Or, to put it another way, if we use $A[j : k]$ to denote the subarray of A from index j to index k, then the maximum subarray problem is to find the subarray $A[j : k]$ that maximizes the sum of its values. In addition to being considered a good problem for testing the thinking skills of prospective employees, the maximum subarray problem also has applications in pattern analysis in digitized images. An example of this problem is shown in Figure 1.13.

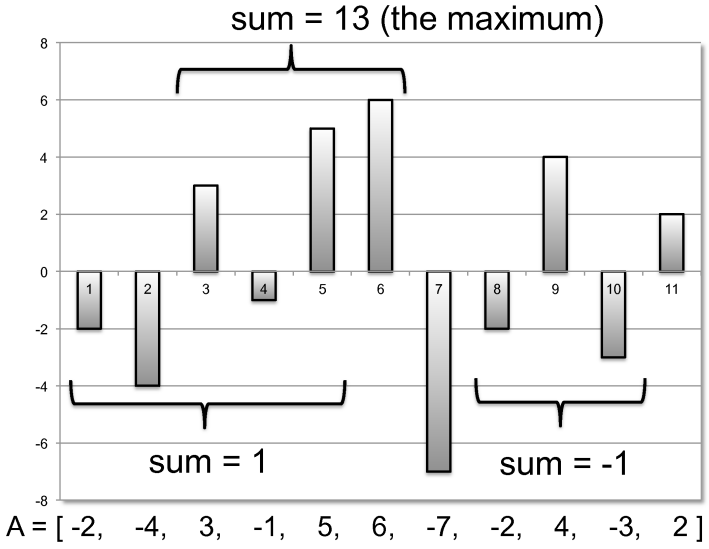

Figure 1.13: An instance of the maximum subarray problem. In this case, the maximum subarray is $A[3 : 6]$, that is, the maximum sum is $s_{3,6}$.

1.4.1 A First Solution to the Maximum Subarray Problem

Our first algorithm for the maximum subarray problem, which we call MaxsubSlow, is shown in Algorithm 1.14. It computes the maximum of every possible subarray summation, $s_{j,k}$, of A separately.

Algorithm MaxsubSlow(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The subarray summation value such that $A[j] + \cdots + A[k]$ is maximized.

```
for j ← 1 to n do
    m ← 0      // the maximum found so far
    for k ← j to n do
        s ← 0      // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return max
```

Algorithm 1.14: Algorithm MaxsubSlow.

It isn't hard to see that the MaxsubSlow algorithm is correct. This algorithm calculates the partial sum, $s_{j,k}$, of every possible subarray, by adding up the values in the subarray from a_j to a_k. Moreover, for every such subarray sum, it compares that sum to a running maximum and if the new value is greater than the old, it updates that maximum to the new value. In the end, this will be maximum subarray sum.

Incidentally, both the calculating of subarray summations and the computing of the maximum so far are examples of the ***accumulator*** design pattern, where we incrementally accumulate values into a single variable to compute a sum or maximum (or minimum). This is a pattern that is used in a lot of algorithms, but in this case it is not being used in the most efficient way possible.

Analyzing the running time of the MaxsubSlow algorithm is easy. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the inner-most loop, for index i, will iterate at most n times. Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$. Unfortunately, in spite of its use of the accumulator design pattern, giving the MaxsubSlow algorithm as a solution to the maximum subarray problem would be a bad idea during a job interview. This is a slow algorithm for the maximum subarray problem.

1.4.2 An Improved Maximum Subarray Algorithm

We can design an improved algorithm for the maximum subarray problem by observing that we are wasting a lot of time by recomputing all the subarray summations from scratch in the inner loop of the MaxsubSlow algorithm. There is a much more efficient way to calculate these summations. The crucial insight is to consider all the ***prefix sums***, which are the sums of the first t integers in A for $t = 1, 2, \ldots, n$. That is, consider each prefix sum, S_t, which is defined as

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i.$$

If we are given all such prefix sums, then we can compute any subarray summation, $s_{j,k}$, in constant time using the formula

$$s_{j,k} = S_k - S_{j-1},$$

where we use the notational convention that $S_0 = 0$. To see this, note that

$$\begin{aligned} S_k - S_{j-1} &= \sum_{i=1}^{k} a_i - \sum_{i=1}^{j-1} a_i \\ &= \sum_{i=j}^{k} a_i = s_{j,k}, \end{aligned}$$

where we use the notational convention that $\sum_{i=1}^{0} a_i = 0$. We can incorporate the above observations into an improved algorithm for the maximum subarray problem, called MaxsubFaster, which we show in Algorithm 1.15.

Algorithm MaxsubFaster(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The subarray summation value such that $A[j] + \cdots + A[k]$ is maximized.

```
S_0 ← 0    // the initial prefix sum
for i ← 1 to n do
    S_i ← S_{i−1} + A[i]
for j ← 1 to n do
    m ← 0    // the maximum found so far
    for k ← j to n do
        if S_k − S_{j−1} > m then
            m ← S_k − S_{j−1}
return max
```

Algorithm 1.15: Algorithm MaxsubFaster.

Analyzing the MaxsubFaster Algorithm

The correctness of the MaxsubFaster algorithm follows along the same arguments as for the MaxsubSlow algorithm, but it is much faster. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the steps inside that loop will only take $O(1)$ time in each iteration. Thus, the total running time of the MaxsubFaster algorithm is $O(n^2)$, which improves the running time of the MaxsubSlow algorithm by a linear factor.

True story: a former student of one of the authors gave this very algorithm during a job interview for a major software company, when asked about the maximum subarray problem, correctly observing that this algorithm beats the running time of the naive $O(n^3)$-time algorithm by a linear factor. Sadly, this student did not get a job offer, however, and one possible reason could have been because there is an even better solution to the maximum subarray problem, which the student didn't give.

1.4.3 A Linear-Time Maximum Subarray Algorithm

We can improve the running time for solving the maximum subarray further by applying the intuition behind the prefix summations idea to the computation of the maximum itself. That is, what if, instead of computing a partial sum, S_t, for $t = 1, 2, \ldots, n$, of the values of the subarray from a_1 to a_t, we compute a "partial maximum," M_t, which is the maximum summation of a subarray of $A[1 : t]$ that ends at index t?

Such a definition is an interesting idea, but it is not quite right, because it doesn't include the boundary case where we wouldn't want any subarray that ends at t, in the event that all such subarrays sum up to a negative number. So, recalling our notation of letting $s_{j,k}$ denote the partial sum of the values in $A[j : k]$, let us define

$$M_t = \max\{0, \max_{j \le t}\{s_{j,t}\}\}.$$

In other words, M_t is the maximum of 0 and the maximum $s_{j,k}$ value where we restrict k to equal t. This definition implies that if $M_t > 0$, then it is the summation value for a maximum subarray that ends at t, and if $M_t = 0$, then we can safely ignore any subarray that ends at t.

Note that if we know all the M_t values, for $t = 1, 2, \ldots, n$, then the solution to the maximum subarray problem would simply be the maximum of all these values. So let us consider how we could compute these M_t values.

The crucial observation is that, for $t \ge 2$, if we have a maximum subarray that ends at t, and it has a positive sum, then it is either $A[t : t]$ or it is made up of the maximum subarray that ends at $t - 1$ plus $A[t]$. If this were not the case, then we could make an even bigger subarray by swapping out the one we chose to end at $t - 1$ with the maximum one that ends at $t - 1$, which would contradict the fact that we have the maximum subarray that ends at t. In addition, if taking the value

of maximum subarray that ends at $t-1$ and adding $A[t]$ makes this sum no longer be positive, then $M_t = 0$, for there is no subarray that ends at t with a positive summation. In other words, we can define $M_0 = 0$ as a boundary condition, and use the following formula to compute M_t, for $t = 1, 2, \ldots, n$:

$$M_t = \max\{0,\ M_{t-1} + A[t]\}.$$

Therefore, we can solve the maximum subarray problem using the algorithm, MaxsubFastest, shown in Algorithm 1.16.

Algorithm MaxsubFastest(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The subarray summation value such that $A[j] + \cdots + A[k]$ is maximized.

$M_0 \leftarrow 0$ // the initial prefix maximum
for $t \leftarrow 1$ **to** n **do**
  $M_t \leftarrow \max\{0,\ M_{t-1} + A[t]\}$
$m \leftarrow 0$ // the maximum found so far
for $t \leftarrow 1$ **to** n **do**
  $m \leftarrow \max\{m,\ M_t\}$
return m

Algorithm 1.16: Algorithm MaxsubFastest.

Analyzing the MaxsubFastest Algorithm

The MaxsubFastest algorithm consists of two loops, which each iterate exactly n times and take $O(1)$ time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is $O(n)$. Incidentally, in addition to using the accumulator pattern, to calculate the M_t and m variables based on previous values of these variables, it also can be viewed as a simple application of the ***dynamic programming*** technique, which we discuss in Chapter 13. Given all these positive aspects of this algorithm, even though we can't guarantee that a prospective employee will get a job offer by describing the MaxsubFastest algorithm when asked about the maximum subarray problem, we can at least guarantee that this is the way to nail this question. Still, we are nonetheless leaving one small detail as an exercise (C-1.1), which is to modify the description of the MaxsubFastest algorithm so that, in addition to the value of the maximum subarray summation, it also outputs the indices j and k that identify the maximum subarray $A[j : k]$.