1 Array Algorithms

The problems in this chapter might look familiar, but we'll examine them through the lens of designing and analyzing algorithms (rather than writing executable code). In particular, through these problems, we'll explore two fundamental concepts used throughout theoretical computer science (TCS): correctness proofs and running time analysis. Note that we generally index arrays starting at 1 (not 0).

1.1 Max in Array

Problem statement. Let A be an array of n integers. Our goal is to find the largest integer in A.

Algorithm. We scan the elements of A, and whenever we encounter a value $A[i]$ larger than m (initially $-\infty$), we set m to $A[i]$. After the scan, we return m.

Algorithm 1 Max in Array

```
Input: Array $A$ of $n$ integers
1: $m \leftarrow -\infty$
2: for $i \leftarrow 1, \ldots, n$ do
3:     if $A[i] > m$ then
4:         $m \leftarrow A[i]$
5: return $m$
```

Theorem 1.1. *Algorithm 1 finds the largest integer in A.*

Proof. Let i^* denote the smallest index containing the largest integer in A. In Algorithm 1, when $i = i^*$, m is either $-\infty$ or $A[j]$ for some $j < i^*$. Either way, $A[i^*] > m$, so m becomes $A[i^*]$. In the rest of the scan, $A[k] \leq A[i^*]$ for all k from $i^* + 1$ through n, so m remains equal to $A[i^*]$ and gets returned at the end. □

The running time of an algorithm is the number of primitive operations that it makes on a worst-case input (i.e., one that maximizes the number of such operations), in terms of the input length. (An array containing n integers has length n.) Each of the following is a primitive operation:

- Assigning a value to a variable (e.g., $x \leftarrow 0$)
- Performing a comparison (e.g., $x > y$)
- Performing an arithmetic operation (e.g., $x + y$)
- Indexing into an array (e.g., accessing $A[3]$)
- Calling or returning from a method

Since asymptotic notation is ubiquitous in the field of algorithms, the exact number of primitive operations does not matter, so it is not important to count extremely rigorously. (For example, it does not matter exactly which operations are involved in a for-loop.) In this chapter, we'll count primitive operations somewhat more meticulously than we will in later chapters.

Theorem 1.2. *Algorithm 1 runs in $O(n)$ time.*

Proof. Line 1 is an assignment, which is 1 (primitive) operation. Line 2 assigns 1 to the variable i, then increments i $n-1$ times, so Line 2 contributes n operations over the course of the algorithm. In the body of the loop, we do at most 4 primitive operations: index into A, perform a comparison against m, index into A again, and set m to $A[i]$. Thus, Lines 3–4 contribute a total of at most $4n$ operations. Finally, returning m is a primitive operation. Adding everything together, we see that the total number of primitive operations is at most $1 + n + 4n + 1 = 5n + 2 = O(n)$. □

1.2 Two Sum

Problem statement. Let A be an array of n distinct integers sorted in increasing order, and let t be an integer. Our goal is to find indices i, j such that $i < j$ and $A[i] + A[j] = t$, or report that no solution exists.

Algorithm. The algorithm repeats the following, starting with $i = 1$ and $j = n$: if $A[i] + A[j] = t$, return (i, j). If $A[i] + A[j] < t$ then increment i (which increases $A[i]$), and if $A[i] + A[j] > t$ then decrement j (which decreases $A[j]$). If we ever hit $i = j$, then return 0 to indicate that no solution exists.

Algorithm 2 Two Sum

```
Input: Sorted array $A$ of $n$ distinct integers, $t$
1: $i, j \leftarrow 1, n$
2: while $i \neq j$ do
3:     if $A[i] + A[j] = t$ then
4:         return $(i, j)$
5:     if $A[i] + A[j] < t$ then
6:         $i \leftarrow i + 1$
7:     else
8:         $j \leftarrow j - 1$
9: return 0
```

Theorem 1.3. *Algorithm 2 correctly solves the Two Sum problem.*

Proof. We need to show two things: (1) if the algorithm returns a solution (i, j), then $A[i] + A[j] = t$, and (2) if the algorithm returns 0, then the input has no solution. Part (1) is relatively simple: if the algorithm returns (i, j), then $A[i] + A[j] = t$ because the algorithm checks this equality immediately before returning (i, j).

Now let's show (2). We'll actually show the contrapositive: if the input has a solution, then the algorithm returns a solution (i, j). Let (i^*, j^*) denote the solution with the smallest value of i^*. If $1 = i^*$ and $j^* = n$, then the algorithm immediately returns $(1, n)$. If not, then i and j iteratively get closer to each other until one of the following cases occurs:

- i reaches i^* before j reaches j^*: When $i = i^*$, $A[i] + A[j] > A[i] + A[j^*] = t$ for all $j > j^*$ since the integers in A are sorted and distinct. Thus, the algorithm decreases j until $j = j^*$ while keeping $i = i^*$.

- j reaches j^* before i reaches i^* (similar to the previous case): When $j = j^*$, $A[i]+A[j] < t$ for all $i < i^*$, so the algorithm increases i until $i = i^*$ while keeping $j = j^*$.

In either case, the algorithm eventually tests if $A[i^*] + A[j^*] = t$, at which point it correctly returns (i^*, j^*). □

Theorem 1.4. *Algorithm 2 runs in $O(n)$ time.*

Proof. Line 1 makes 2 primitive operations. In each iteration of the while loop, we perform a comparison $(i \neq j)$; at most 2 more comparisons and 4 instances of indexing into A (Lines 3 and 5); and at most one return, increment, or decrement (Lines 4, 6, 8). Since the distance between $j - i$ is initially $n - 1$, and each iteration either decreases $j - i$ by 1 or terminates the algorithm, the number of iterations is at most $n - 1$. Thus, Lines 2–8 contribute at most $(1 + 6 + 1) \cdot (n - 1) = 8n - 8$ operations. Finally, Line 9 contributes at most 1 operation. Adding everything together, we see that the total number of operations is at most $2 + 8n - 8 + 1 = 8n - 5 = O(n)$. □

1.3 Binary Search

Problem statement. The Search problem is the following: Let A be an array of n integers sorted in non-decreasing order, and let t be an integer. Our goal is to find an index x such that $A[x] = t$, or report that no such index exists.

Intuition. We start by checking if the the middle element of A is t. If it's too small, then t must be in the right half of A (if it's in A at all), so we recurse on that subarray; otherwise, we recurse on the left half of A. We continue halving the size of the subarray until we find t or determine it's not in A. In the worst case, $t \notin A$ and the number of rounds is roughly $\log_2 n$, because that's how many times we can halve n until we reach 1. Since each round takes constant time, the overall running time is $O(\log n)$.

Algorithm. Starting with $i = 1$ (representing the left endpoint of the subarray under consideration) and $j = n$ (the right endpoint), we will repeatedly calculate $m = \lfloor (i + j)/2 \rfloor$ and checking if $A[m] = t$. If so, we return m; if $A[m] < t$, we set i to $m + 1$; else, we set j to $m - 1$. If $i > j$ at any point, we return 0, which signifies that t is not in A.

Theorem 1.5. *Algorithm 3 correctly solves the Search problem.*

Algorithm 3 Binary Search

```
Input: Sorted array $A$ of $n$ integers, $t$
1: $i \leftarrow 1, j \leftarrow n$
2: while $i \leq j$ do
3:     $m \leftarrow \lfloor (i+j)/2 \rfloor$
4:     if $A[m] = t$ then
5:         return $m$
6:     if $A[m] < t$ then
7:         $i \leftarrow m + 1$
8:     else
9:         $j \leftarrow m - 1$
10: return 0
```

Proof. We again prove two parts, where first part is much simpler: (1) If the algorithm returns some m, then it must satisfy $A[m] = t$. For part (2), we want to show that if t is in A, then Algorithm 3 finds it.

The key idea is that in every iteration, the algorithm shrinks $A[i:j]$ (the subarray of A from index i through index j) while ensuring that t is still in the shrunken subarray. To see this, notice that if $A[m] < t$, then t is in $A[m+1:j]$ because A is sorted, and in this case, the algorithm increases i to $m+1$. (This is indeed an increase since the starting value of i is $\lfloor 2i/2 \rfloor \leq \lfloor (i+j)/2 \rfloor = m$.) Similarly, if $A[m] > t$, the algorithm decreases j to $m-1$ while maintaining that t is in $A[i:j]$. So in each iteration, the algorithm either finds t or shrinks the size of $A[i:j]$. This continues until it returns some m or $i = j$, at which point $m = i = j$ so $A[i:j] = A[m] = t$. □

Theorem 1.6. *Algorithm 3 runs in $O(\log n)$ time.*

Proof. At any point in the algorithm, the size of the subarray under consideration is $j - i + 1$ (initially n), and in the worst case, t is not in A so we iterate until $i > j$. Let $d = j - i + 1$ at the beginning of some iteration, and let $d' = j - i + 1$ at the end of the iteration. If i becomes $m + 1$, then we have

$$d' = j - (m+1) + 1 = j - \left\lfloor \frac{i+j}{2} \right\rfloor \leq j - \frac{i+j}{2} + \frac{1}{2} = \frac{j-i+1}{2} = \frac{d}{2}.$$

Similarly, if j becomes $m-1$, then we have $d' = (m-1) - i + 1 \leq d/2$. Either way, $d' \leq d/2$, i.e., in each round, $A[i:j]$ shrinks to at most half of its size at the beginning of the round.

Initially, $A[i:j] = A$ has n elements, so after k rounds, $A[i:j]$ has at most $n/2^k$ elements. This means if $k > \log_2 n$, then $A[i:j]$ has no elements (i.e., $i > j$), so the algorithm must terminate at this point. In each round, the algorithm performs a constant number of operations (e.g., 3 comparisons, at most 1 return, etc.), so the total number of operations is $O(\log n)$. □

1.4 Selection Sort

Problem statement. Let A be an array of n integers. Our goal is to sort the elements of A (i.e., rearrange them such that $A[1] \leq A[2] \leq \cdots \leq A[n]$).

Algorithm. The algorithm proceeds in rounds, starting with $i = 1$. In each round of the algorithm, we scan $A[i]$ through $A[n]$ to find the index j of the smallest element in this subarray $A[i:n]$. Then we swap the values of $A[i]$ and $A[j]$, increment i, and repeat the process until i reaches n.

Algorithm 4 Selection Sort

```
Input: Array $A$ of $n$ integers
1: for $i \leftarrow 1, \ldots, n$ do
2:     $m \leftarrow i$
3:     for $j \leftarrow i+1, \ldots, n$ do
4:         if $A[j] < A[m]$ then
5:             $m \leftarrow j$
6:     Swap $A[i], A[m]$
7: return $A$
```

Theorem 1.7. *Algorithm 4 sorts the input.*

Proof. Sorting A amounts to setting $A[1]$ as the smallest element of A, $A[2]$ the second smallest element of A, and so on; this is precisely what the algorithm does.

We can proceed more formally by induction. Our claim is that at the end of round i, $A[1:i]$ contains the i smallest i elements of A in sorted order. The theorem follows from this claim when $i = n$. Our base case is when $i = 1$, in which case the algorithm indeed sets $A[1]$ to be the smallest element of A. More generally, in Lines 2–5 of round $i+1$, the algorithm scans $A[i+1:n]$ to find the smallest element of $A[i+1:n]$, which is the $(i+1)$-th smallest element of A (since $A[1:i]$ contains i smaller elements). In Line 6, it places this element at index i. The result is that at the end of round $i+1$, $A[1:i+1]$ contains the $i+1$ smallest elements of A in sorted order, as desired. $\square$

Theorem 1.8. *Algorithm 4 runs in $O(n^2)$ time.*

Proof. In round i, j iterates from $i+1$ through n. In each of these $n-i$ iterations, the algorithm makes a constant number of operations (increment j, access A twice, perform a comparison, etc.). Besides those, in round i, the algorithm performs a constant number of operations (set m to i, access A twice, etc.). So the total number of operations in round i is at most $c(n-i)$ for some constant c. Summing over all $i = 1, \ldots, n$, we see that the total number of operations is at most

$$c\sum_{i=1}^{n}(n-i) = c \cdot \left(n^2 - \sum_{i=1}^{n} i\right) = c \cdot \left(n^2 - \frac{n(n+1)}{2}\right) = c \cdot \frac{n^2-n}{2} = O(n^2).$$

In fact, regardless of the values in A, Algorithm 4 makes at least $c'(n-i)$ operations in round i for some constant c', so the running time is $\Theta(n^2)$. $\square$

1.5 Merge Sort

Problem statement. Let A be an array of n integers. Our goal is to sort the elements of A (i.e., rearrange them such that $A[1] \le A[2] \le \cdots \le A[n]$).

Algorithm. The algorithm is recursive: if A only has 1 element, then we can simply return A. Otherwise, we split A into two its left and right halves, and recursively sort each half. Then we perform the defining step of this algorithm: in linear time, we merge the two sorted halves into a single sorted array.

Algorithm 5 Merge Sort

```
Input: Array $A$ of $n$ integers
 1: if $n = 1$ then
 2:     return $A$
 3: $k \leftarrow \lfloor n/2 \rfloor$
 4: $A_L, A_R \leftarrow$ MergeSort($A[1:k]$), MergeSort($A[k+1:n]$)
 5: Create empty list $B$
 6: $i, j \leftarrow 1, 1$
 7: while $i \leq k$ and $j \leq n-k$ do                ▷ Extend $B$ by merging $A_L$ and $A_R$
 8:     if $A_L[i] \leq A_R[j]$ then
 9:         Append $A_L[i]$ to $B$, $i \leftarrow i+1$
10:     else
11:         Append $A_R[j]$ to $B$, $j \leftarrow j+1$
12: if $i > k$ then                                    ▷ Append remaining elements
13:     Append each element of $A_R[j:]$ to $B$
14: else
15:     Append each element of $A_L[i:]$ to $B$
16: return $B$ (as an array)
```

Theorem 1.9. *Algorithm 5 sorts the input.*

Proof. Since the algorithm is recursive, it is natural for us to proceed by induction. If $n = 1$, then the algorithm correctly returns A as it is. Otherwise, suppose the algorithm sorts all arrays of size less than n, so A_L and A_R are each sorted.

Consider the first iteration of Line 7: if $A_L[1] \leq A_R[1]$, then because A_L and A_R are sorted, $A_L[1]$ must be the smallest element of A. Therefore, appending $A_L[1]$ as the first element of B is correct. In general, if $A_L[i] \leq A_R[j]$, then among the elements of A that haven't been appended to B, $A_L[i]$ must be the smallest, so appending it to B (in Line 9) is correct. (The same reasoning holds if $A_R[j] < A_L[i]$.) If i exceeds k, then for the value of j at that point, we must have $A_L[k] \leq A_R[j]$. Thus, the subarray $A_R[j:]$ contains all remaining elements of A in sorted order, so appending them to B (in Line 13) is correct. (The same reasoning holds if j exceeds $n-k$.) $\square$

Theorem 1.10. *Algorithm 5 runs in $O(n \log n)$ time.*

Proof. Let $T(n)$ denote the running time of Algorithm 5 on an input of size n. In Line 4, we call the algorithm itself on two inputs of size roughly $n/2$ each.[1] In each iteration of

[1]For simplicity, we are assuming $n/2$ is always an integer, but the full analysis isn't very different.

Line 7, $i+j$ increases by 1, and since $i+j \leq n$, Lines 7-11 take $O(n)$ time. Appending the remaining elements of A_R or A_L also takes $O(n)$ time. Putting this all together, we have

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn$$

for some constant c, and we can assume $T(1) = k$ for some constant $k \geq c$. We shall prove $T(n) \leq kn(\log n + 1)$ by induction, where the logarithm is of base 2. The base case is true since $T(1) = k \leq k \cdot 1 \cdot (0+1)$. Now we proceed inductively:

$$\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + cn \\
&\leq 2 \cdot \frac{kn}{2}\left(\log \frac{n}{2} + 1\right) + cn \\
&\leq kn \log n + kn \\
&= kn(\log n + 1).
\end{aligned}$$

If $n \geq 2$, then $1 \leq \log n$, so $T(n) \leq kn \cdot (2 \log n) = O(n \log n)$, as desired. □

1.6 Exercises

Throughout these notes, whenever an exercise asks for an algorithm that solves some problem in some amount of time, it is implicitly also asking for a proof of correctness and a running time analysis.

1.1. Suppose we want to maximize the number of operations performed by Algorithm 1 (finding the max in an array). What kind of input would we give it?

1.2. What is a lower bound on the number of operations made by Algorithm 1? Your answer should be in terms of n and as large as possible.

1.3. Run Algorithm 2 (Two Sum) on the input $A = [1, 3, 4, 5, 6, 9]$ and $t = 8$, keeping track of exactly how i and j change.

1.4. Run Algorithm 4 (Selection Sort) on the input $A = [4, 1, 3, 5, 9, 6]$, printing the contents of A at the end of each round.

1.5. Our solution to the Two Sum problem is considered a "two-pointer" approach rather than "brute force." Give an $O(n^2)$-time, brute-force algorithm for the same problem. What are the advantages and disadvantages of each approach?

1.6. Consider the Two Sum problem, but suppose the constraint is $i \leq j$ (rather than $i < j$). How does the two-pointer algorithm and its analysis change, if at all?

1.7. What is the running time of Selection Sort if the input is already sorted in increasing order? And if the input is sorted in decreasing order?

1.8. Let A be an array of n distinct, positive integers, and suppose we want to find the sum of the two largest elements in A. Give an $O(n)$-time algorithm for this problem. Bonus: Your algorithm should loop through the array exactly once.

1.9. Let A be an array of n distinct integers, and let t be an integer. We want to find distinct indices i, j, k such that $A[i] + A[j] + A[k] = t$. Give an $O(n^2)$-time algorithm for this problem.

1.10. Let A be an (unsorted) array containing $n + 1$ integers; every integer $\{1, 2, \ldots, n\}$ is in A, and one of them appears in A twice. Give an $O(n)$-time algorithm that returns the integer that appears twice. Bonus: Do not construct a separate array.

1.11. Let A be an array of n integers. We want to find indices i, j such that $i \leq j$ and the sum from $A[i]$ through $A[j]$ is maximized. Give an $O(n^2)$-time algorithm for this problem. Bonus: $O(n)$ instead of $O(n^2)$.

1.12. Let A be a sorted array of n integers. We want to return a sorted array that contains the squares of the elements of A. Give an $O(n^2)$-time algorithm for this problem. Bonus: $O(n)$ instead of $O(n^2)$.

1.13. Let A be an array of n integers. We want to find indices i, j such that $i < j$ and $A[j] - A[i]$ is maximized. Give an $O(n^2)$-time algorithm for this problem. Bonus: $O(n)$ instead of $O(n^2)$.

1.14. Let A be an array of n positive integers. We want to find indices i, j such that $i < j$ and $(j - i) \cdot \min(A[i], A[j])$ is maximized. Give an $O(n^2)$-time algorithm for this problem. Bonus: $O(n)$ instead of $O(n^2)$.

1.15. Let A be an array of n integers. For any pair of indices (i, j), we call (i, j) an *inversion* if $i < j$ and $A[i] > A[j]$. Give an $O(n \log n)$-time algorithm that returns the number of inversions in A.