Chapter 8

Amortized analysis

The time complexity of an algorithm is often easy to analyze just by examining the structure of the algorithm: what loops does the algorithm contain and how many times the loops are performed. However, sometimes a straightforward analysis does not give a true picture of the efficiency of the algorithm.

Amortized analysis can be used to analyze algorithms that contain operations whose time complexity varies. The idea is to estimate the total time used to all such operations during the execution of the algorithm, instead of focusing on individual operations.

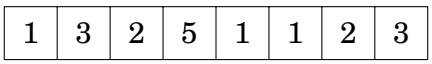
Two pointers method

In the **two pointers method**, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently. Next we discuss two problems that can be solved using the two pointers method.

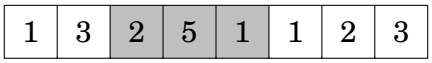
Subarray sum

As the first example, consider a problem where we are given an array of n positive integers and a target sum x, and we want to find a subarray whose sum is x or report that there is no such subarray.

For example, the array

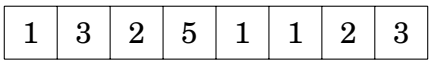


contains a subarray whose sum is 8:

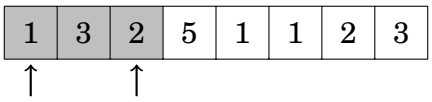


This problem can be solved in $O(n)$ time by using the two pointers method. The idea is to maintain pointers that point to the first and last value of a subarray. On each turn, the left pointer moves one step to the right, and the right pointer moves to the right as long as the resulting subarray sum is at most x. If the sum becomes exactly x, a solution has been found.

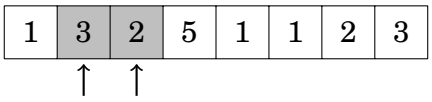
As an example, consider the following array and a target sum $x = 8$:



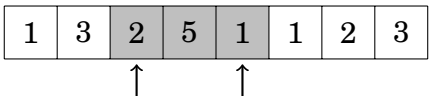
The initial subarray contains the values 1, 3 and 2 whose sum is 6:



Then, the left pointer moves one step to the right. The right pointer does not move, because otherwise the subarray sum would exceed x.



Again, the left pointer moves one step to the right, and this time the right pointer moves three steps to the right. The subarray sum is $2 + 5 + 1 = 8$, so a subarray whose sum is x has been found.



The running time of the algorithm depends on the number of steps the right pointer moves. While there is no useful upper bound on how many steps the pointer can move on a *single* turn. we know that the pointer moves *a total of* $O(n)$ steps during the algorithm, because it only moves to the right.

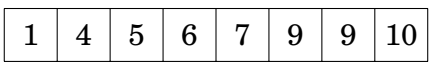
Since both the left and right pointer move $O(n)$ steps during the algorithm, the algorithm works in $O(n)$ time.

2SUM problem

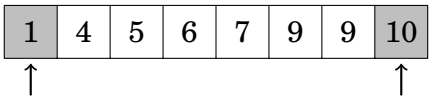
Another problem that can be solved using the two pointers method is the following problem, also known as the **2SUM problem**: given an array of n numbers and a target sum x, find two array values such that their sum is x, or report that no such values exist.

To solve the problem, we first sort the array values in increasing order. After that, we iterate through the array using two pointers. The left pointer starts at the first value and moves one step to the right on each turn. The right pointer begins at the last value and always moves to the left until the sum of the left and right value is at most x. If the sum is exactly x, a solution has been found.

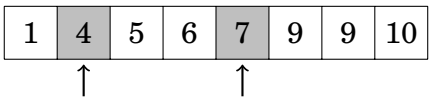
For example, consider the following array and a target sum $x = 12$:



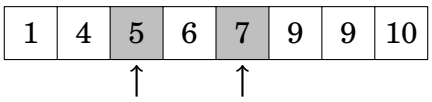
The initial positions of the pointers are as follows. The sum of the values is $1 + 10 = 11$ that is smaller than x.



Then the left pointer moves one step to the right. The right pointer moves three steps to the left, and the sum becomes $4 + 7 = 11$.



After this, the left pointer moves one step to the right again. The right pointer does not move, and a solution $5 + 7 = 12$ has been found.



The running time of the algorithm is $O(n \log n)$, because it first sorts the array in $O(n \log n)$ time, and then both pointers move $O(n)$ steps.

Note that it is possible to solve the problem in another way in $O(n \log n)$ time using binary search. In such a solution, we iterate through the array and for each array value, we try to find another value that yields the sum x. This can be done by performing n binary searches, each of which takes $O(\log n)$ time.

A more difficult problem is the **3SUM problem** that asks to find *three* array values whose sum is x. Using the idea of the above algorithm, this problem can be solved in $O(n^2)$ time[1]. Can you see how?

Nearest smaller elements

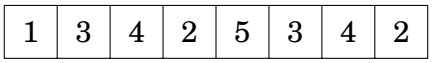
Amortized analysis is often used to estimate the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.

As an example, consider the problem of finding for each array element the **nearest smaller element**, i.e., the first smaller element that precedes the element in the array. It is possible that no such element exists, in which case the algorithm should report this. Next we will see how the problem can be efficiently solved using a stack structure.

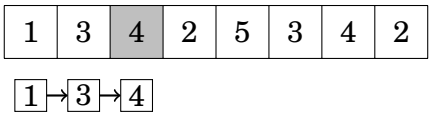
We go through the array from left to right and maintain a stack of array elements. At each array position, we remove elements from the stack until the top element is smaller than the current element, or the stack is empty. Then, we report that the top element is the nearest smaller element of the current element, or if the stack is empty, there is no such element. Finally, we add the current element to the stack.

As an example, consider the following array:

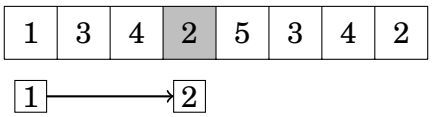
[1] For a long time, it was thought that solving the 3SUM problem more efficiently than in $O(n^2)$ time would not be possible. However, in 2014, it turned out [30] that this is not the case.



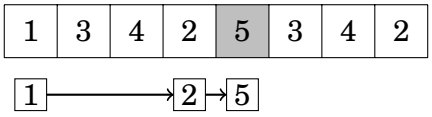
First, the elements 1, 3 and 4 are added to the stack, because each element is larger than the previous element. Thus, the nearest smaller element of 4 is 3, and the nearest smaller element of 3 is 1.



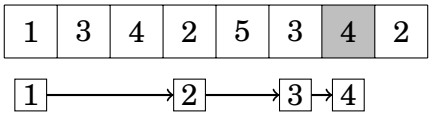
The next element 2 is smaller than the two top elements in the stack. Thus, the elements 3 and 4 are removed from the stack, and then the element 2 is added to the stack. Its nearest smaller element is 1:



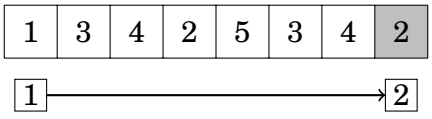
Then, the element 5 is larger than the element 2, so it will be added to the stack, and its nearest smaller element is 2:



After this, the element 5 is removed from the stack and the elements 3 and 4 are added to the stack:



Finally, all elements except 1 are removed from the stack and the last element 2 is added to the stack:



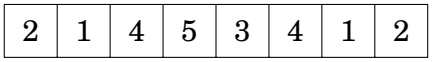
The efficiency of the algorithm depends on the total number of stack operations. If the current element is larger than the top element in the stack, it is directly added to the stack, which is efficient. However, sometimes the stack can contain several larger elements and it takes time to remove them. Still, each element is added *exactly once* to the stack and removed *at most once* from the stack. Thus, each element causes $O(1)$ stack operations, and the algorithm works in $O(n)$ time.

Sliding window minimum

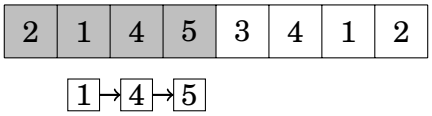
A **sliding window** is a constant-size subarray that moves from left to right through the array. At each window position, we want to calculate some information about the elements inside the window. In this section, we focus on the problem of maintaining the **sliding window minimum**, which means that we should report the smallest value inside each window.

The sliding window minimum can be calculated using a similar idea that we used to calculate the nearest smaller elements. We maintain a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the end of the queue.

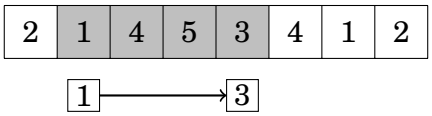
As an example, consider the following array:



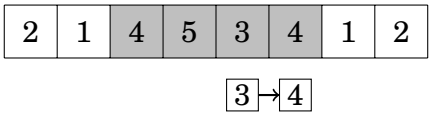
Suppose that the size of the sliding window is 4. At the first window position, the smallest value is 1:



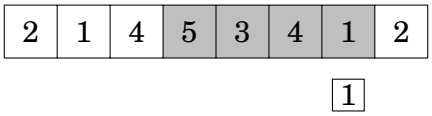
Then the window moves one step right. The new element 3 is smaller than the elements 4 and 5 in the queue, so the elements 4 and 5 are removed from the queue and the element 3 is added to the queue. The smallest value is still 1.



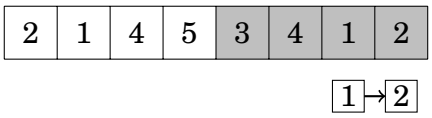
After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the queue and the smallest value is now 3. Also the new element 4 is added to the queue.



The next new element 1 is smaller than all elements in the queue. Thus, all elements are removed from the queue and it will only contain the element 1:



Finally the window reaches its last position. The element 2 is added to the queue, but the smallest value inside the window is still 1.



Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in $O(n)$ time.