

Array Advance Game

In this lesson, you will learn how to solve the problem of the Array Advance Game in Python.

We'll cover the following ^

- Problem
- Algorithm
 - Greedy Approach
- Implementation
- Explanation

In this lesson, we will be considering the “array advance game”. In this game, you are given an array of non-negative integers. For example:

```
[3,3,1,0,2,0,1]
```

Each number in the array represents the maximum you can advance in the array.

Problem

Now the problem is as follows:

Is it possible to advance from the start of the array to the last element given that the maximum you can advance from a position is based on the value of the array at the index you are currently present on?

We will cover how to solve this problem algorithmically, and then code up a solution to this problem in Python.

Let's start with a simple example. Refer to the slides below:



Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

1 of 5

Array A

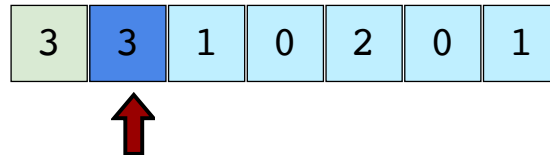
3	3	1	0	2	0	1
---	---	---	---	---	---	---



Current Position : index 0
Maximum that you can advance:
= $A[0] = 3$



Array A

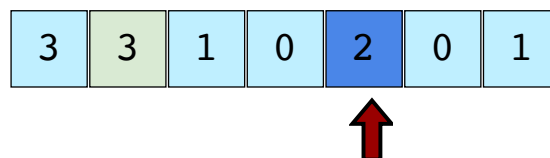


Moved 1 place forward

Current Position : index 1
Maximum that you can advance:
= $A[1] = 3$

3 of 5

Array A



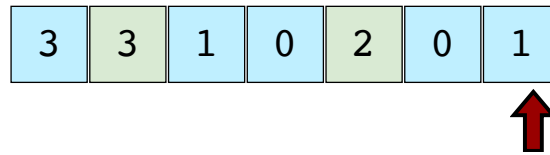
Moved 3 places forward

Current Position : index 4
Maximum that you can advance:
= $A[4] = 2$

4 of 5



Array A



Moved 2 places forward

Current Position : index 6
Success!

5 of 5



In the example above, we have successfully reached the end of the array. Let's look at a case that is *unwinnable* where we can't reach the end of the array.



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

1 of 9

Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Current Position : index 0
Maximum that you can advance:
 $= A[0] = 3$

2 of 9



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Moved 3 places

Current Position : index 3
Maximum that you can advance:
= $A[3] = 0$

3 of 9

Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Unwinnable!

Current Position : index 3
Maximum that you can advance:
= $A[3] = 0$

4 of 9

Let's try another strategy!



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

5 of 9

Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Current Position : index 0
Maximum that you can advance:
= $A[0] = 3$

6 of 9



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Move 1 place

Current Position : index 1
Maximum that you can advance:
= $A[1] = 2$

7 of 9

Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---



Move 2 places

Current Position : index 3
Maximum that you can advance:
= $A[3] = 0$

8 of 9



Array A

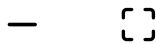
3	2	0	0	2	0	1
---	---	---	---	---	---	---



Unwinnable!

Current Position : index 3
Maximum that you can advance:
= $A[3] = 0$

9 of 9



Algorithm

I hope you are clear about the game and the problem statement at this point. We have to figure out the best way to advance in the array so that we reach the last index.

Greedy Approach

Let's try the *greedy* approach in which we always advance the maximum we can at every index.

The example below illustrates the greedy approach.



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

1 of 6

Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Current Position : index 0
Maximum that you can advance:
= $A[0] = 2$

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

2 of 6



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Current Position : index 2
Maximum that you can advance:
= $A[2] = 1$

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

3 of 6

Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Current Position : index 3
Maximum that you can advance:
= $A[3] = 1$

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

4 of 6



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Current Position : index 4
Maximum that you can advance:
= $A[4] = 0$

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

5 of 6

Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Unwinnable!
Current Position : index 4
Maximum that you can advance:
= $A[4] = 0$

Idea: Use "greedy" strategy.
Advance as much as possible for each number.

6 of 6

The example above perfectly illustrates that the *greedy* approach doesn't work. However, the array is not unwinnable. Have a look at the slides below:



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---

Idea: Use "non-greedy" strategy.

1 of 5



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Current Position : index 0
Maximum that you can advance:
= $A[0] = 2$

2 of 5

Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Moved 1 place
Current Position : index 1
Maximum that you can advance:
= $A[1] = 4$

3 of 5



Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Moved 4 places forward

Current Position : index 5
Maximum that you can advance:
= $A[5] = 2$

4 of 5

Array A

2	4	1	1	0	2	3
---	---	---	---	---	---	---



Success!
Current Position : index 6

5 of 5

As you can see, the greedy algorithm doesn't work for this solution. Let's look at the algorithm which will solve the problem:



- Iterate through each entry in an array, A .
- Track the furthest we can reach from entry $(A[i] + i)$
- If for some i , we don't reach the end and that is the furthest we can reach, then we can't reach the last index. Otherwise, the end is reached.
- i : index processed. Furthest possible to advance from i : $A[i] + i$

The example below will run through each step of the algorithm stated below:

Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

`furthest_reached = 0`

1 of 7



Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

 $i=0$ $\text{furthest_reached} = 0$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[0] + 0)$ $= \max(0, 3+0)$ $= 3$

2 of 7

Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

 $i=1$ $\text{furthest_reached} = 3$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[1] + 1)$ $= \max(3, 3+1)$ $= 4$

3 of 7



Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

 $i=2$ $\text{furthest_reached} = 4$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[2] + 2)$ $= \max(4, 1+2)$ $= 4$

4 of 7

Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---

 $i=3$ $\text{furthest_reached} = 4$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[3] + 3)$ $= \max(4, 0+3)$ $= 4$

5 of 7



Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---



$i=4$

$\text{furthest_reached} = 4$

$\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$

$\text{furthest_reached} = \max(\text{furthest_reached}, A[4] + 4)$

$= \max(4, 2+4)$

$= 6$

6 of 7

Array A

3	3	1	0	2	0	1
---	---	---	---	---	---	---



$i=5$

$\text{furthest_reached} = 6$

$\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$

$\text{furthest_reached} = \max(\text{furthest_reached}, A[5] + 5)$

$= \max(6, 0+5)$

$= 6$

7 of 7

The above example was of a win situation. Let's see how our algorithm works for arrays which are not winnable.



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

`furthest_reached = 0`

1 of 6



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

 $i=0$ $\text{furthest_reached} = 0$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[0] + 0)$ $= \max(0, 3+0)$ $= 3$

2 of 6

Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

 $i=1$ $\text{furthest_reached} = 3$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[1] + 1)$ $= \max(3, 2+1)$ $= 3$

3 of 6



Array A

3	2	0	0	2	0	1
---	---	---	---	---	---	---

 $i=2$ $\text{furthest_reached} = 3$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[2] + 2)$ $= \max(3, 0+2)$ $= 3$

4 of 6

Array A

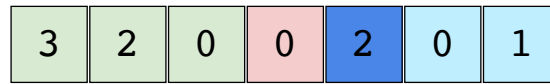
3	2	0	0	2	0	1
---	---	---	---	---	---	---

 $i=3$ $\text{furthest_reached} = 3$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[i] + i)$ $\text{furthest_reached} = \max(\text{furthest_reached}, A[3] + 3)$ $= \max(3, 0+3)$ $= 3$

5 of 6



Array A



$i=4$

$\text{furthest_reached} = 3$

$i > \text{furthest_reached}$ i.e. end not reachable

6 of 6

— []

Implementation

Hopefully, by now, you will clearly understand the algorithm. Let's jump to the implementation in Python, which will be easier to understand once you get the algorithm:

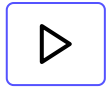
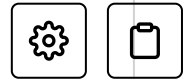
```

1 def array_advance(A):
2     furthest_reached = 0
3     last_idx = len(A) - 1
4     i = 0
5     while i <= furthest_reached and furthest_reach
6         furthest_reached = max(furthest_reached, i + A[i])
7         i += 1
8     return furthest_reached >= last_idx
9
10
11 # True: Possible to navigate to last index in A:
12 # Moves: 1,3,2

```



```
13 A=[3,3,1,0,2,0,1]
14 print(array_advance(A))
15
16 #False:Not possible to navigate to last index in
17 A=[3,2,0,0,2,0,1]
18 print(array_advance(A))
```



Output

0.14s

True
False

array_advance(A)

Explanation

`furthest_reached` and `i` are initialized to 0 on **line 2** and **line 4** respectively. We calculate `last_idx` on **line 3** by subtracting 1 from the length of the array. Next, we proceed to a `while` loop which terminates on the following conditions:

- `i > furthest_reached` : This implies that the end is not reachable.
- `furthest_reached >= last_idx` : This implies that the end is reachable.

In each iteration of the `while` loop, we update `furthest_reached` to the maximum of `furthest_reached` and `(A[i] + i)` on **line 6**. `i` increments by 1 in the next line.

After the `while` loop terminates, we'll check for the condition which terminates the `while` loop. If `furthest_reached >= last_idx`, then `True` is returned from the function. Otherwise, `False` is returned.

In the next lesson, we'll study another interesting and challenging problem.

Stay tuned!



← Back

Next →

Quiz

Arbitrary Precision Increment



Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/array-advance-game__arrays__data-structures-and-algorithms-in-python)

Arbitrary Precision Increment

In this lesson, you will learn how to solve the problem of the Arbitrary Precision Increment using arrays.

We'll cover the following



- Algorithm
- Implementation
- Explanation

In this lesson, we will be considering the following:

Given: An array of non-negative digits that represent a decimal integer.

Problem: Add one to the integer. Assume the solution still works even if implemented in a language with finite-precision arithmetic.

We will cover how to solve this problem algorithmically, and then code a solution to this question in Python.



Arbitrary Precision Increment

Array A

1	4	9
---	---	---



Adding 1



Array A

1	5	0
---	---	---

Algorithm

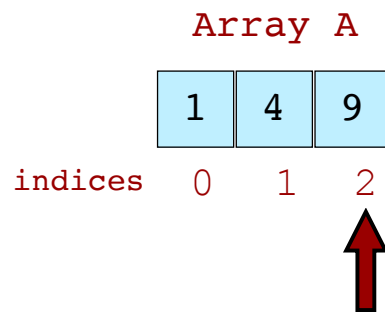
For an array A ,

1. Add 1 to the rightmost digit.
2. Propagate carry throughout the array.

This will be similar to the “standard grade school” approach.



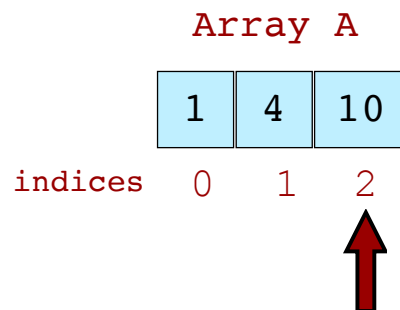
Arbitrary Precision Increment : Example 1



Add 1 to the rightmost digit

1 of 6

Arbitrary Precision Increment : Example 1

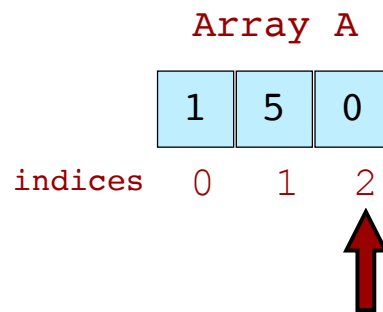


If a sum yields a 10, replace with 0 and add a carry of 1 over to the left.

2 of 6



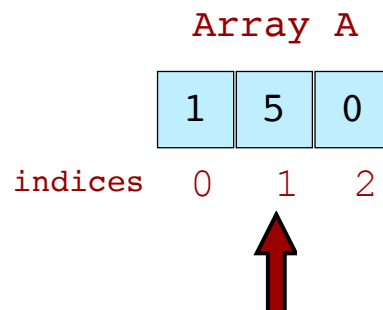
Arbitrary Precision Increment : Example 1



If a sum yields a 10, replace with 0 and add a carry of 1 over to the left.

3 of 6

Arbitrary Precision Increment : Example 1

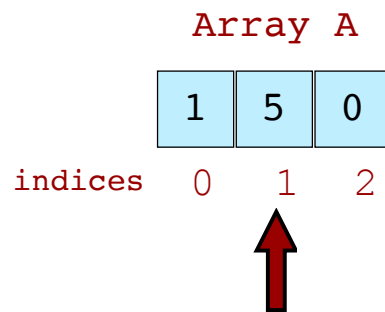


Keep processing from back to front until we don't encounter any 10s (i.e. we don't require further carries)

4 of 6



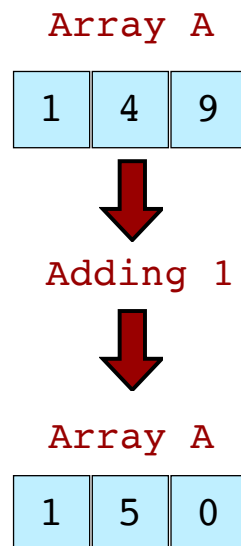
Arbitrary Precision Increment : Example 1



Since there is no 10, we are finished progressing through the array.

5 of 6

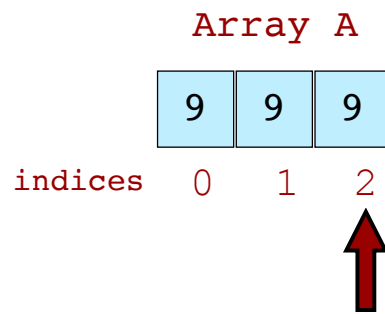
Arbitrary Precision Increment : Example 1



6 of 6



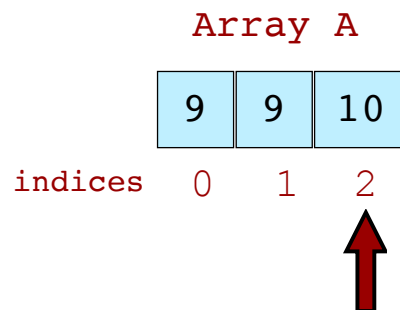
Arbitrary Precision Increment : Example 2



Add 1 to the rightmost digit

1 of 6

Arbitrary Precision Increment : Example 2

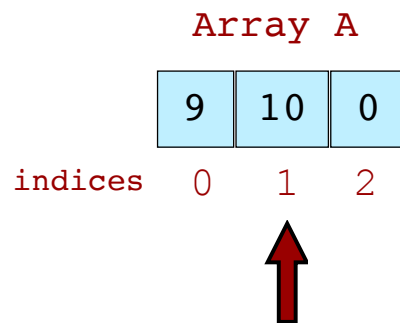


If a sum yields a 10, replace with 0 and add a carry of 1 over to the left.

2 of 6



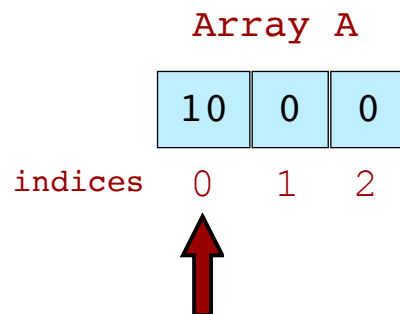
Arbitrary Precision Increment : Example 2



If a sum yields a 10, replace with 0 and add a carry of 1 over to the left.

3 of 6

Arbitrary Precision Increment : Example 2

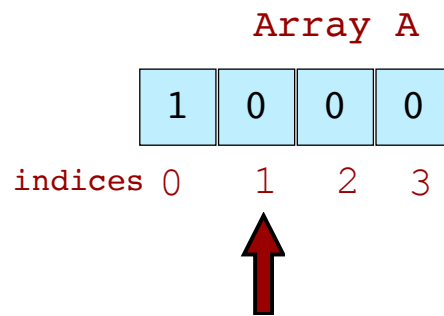


If a sum yields a 10, replace with 0 and add a carry of 1 over to the left.

4 of 6

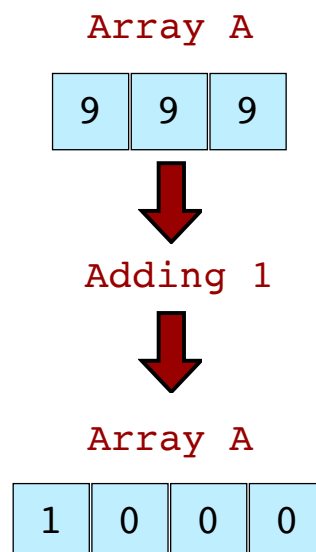


Arbitrary Precision Increment : Example 2



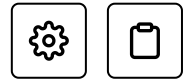
5 of 6

Arbitrary Precision Increment : Example 2



6 of 6

Implementation



In Python, you can solve the problem with just one line.

```
1 A = [1, 4, 9]
2 s = ''.join(map(str, A))
3 print(int(s) + 1)
```



Explanation

On **line 2**, the `map` function returns a list after converting every element of `A` into a string. The `join` function *joins* the elements of that list and returns a string in which the string elements of the sequence have been joined by a string separator (`' '`). All that is left for us to do is to convert `s` to an integer type and add `1` to it. This solution is relatively simple, but you will not be asked for it in an interview.

Let's implement the algorithm that we discussed in the slides.

```
1 A1 = [1, 4, 9]
2 A2 = [9, 9, 9]
3
4 # s = ''.join(map(str, A))
5 # print(int(s) + 1)
6
7
8 def plus_one(A):
9     A[-1] += 1
10    for i in reversed(range(1, len(A))):
11        if A[i] != 10:
12            break
13        A[i] = 0
14        A[i-1] += 1
15    if A[0] == 10:
16        A[0] = 1
17        A.append(0)
18    return A
19
```



```
20
21 print(plus_one(A1))
22 print(plus_one(A2))
```



On **line 9**, we add 1 to the last element of the list A. Now we have to handle the carry as a result of this addition. Therefore, using a for loop on **line 10**, we iterate the list from the last index to the index 1 through the reversed function. Next, on **line 11**, we check if we get 10 as a result of adding 1. If not, then the carry from the addition is 0, and we break the loop on **line 12**. However, if $A[i]$ is 10 then we put 0 in its place and add 1 to the preceding index on **lines 13-14**. This is repeated for every position in the list from the last index to index 1.

Now, we need to handle one other edge case, i.e., the value at the first index is 10. If $A[0]$ is 10, we set $A[0]$ to 1 and append 0 to the list on **lines 16-17**. A is finally returned from the function on **line 18** after being incremented by 1.

Hope the above solution was easy to understand! Let's look at another problem regarding arrays in the next lesson.

[← Back](#)[Next →](#)[Array Advance Game](#)[Two Sum Problem](#)

Completed

[Report an Issue](#)[Ask a Question](#)https://discuss.educative.io/tag/arbitrary-precision-increment__arrays__data-structures-and-algorithms-in-python



Two Sum Problem

In this lesson, you will learn how to solve the Two Sum Problem using three solutions in Python.

We'll cover the following



- Solution 1
 - Implementation
 - Explanation
- Solution 2
 - Implementation
 - Explanation
- Solution 3
 - Implementation
 - Explanation

In this lesson, we are going to be solving the “Two-Sum Problem”. Let’s begin by defining the problem. Given an array of integers, return `True` or `False` if the array has two numbers that add up to a specific target. You may assume that each input would have exactly one solution.



-2	1	2	4	7	11
----	---	---	---	---	----

target = 13



(2, 11)
True

We investigate three different approaches to solving this problem.

Solution 1

A brute-force approach that takes $O(n^2)$ time to solve with $O(1)$ space where we loop through the array and create all possible pairings of elements.

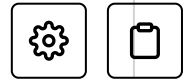
Implementation

Have a look at the code below:

```
1 #TimeComplexity:O(n^2)
2 #SpaceComplexity:O(1)
3 def two_sum_brute_force(A, target):
4     for i in range(len(A)-1):
5         for j in range(i+1, len(A)):
6             if A[i] + A[j] == target:
7                 print(A[i], A[j])
8                 return True
```



```
9  ..return False
10
11 A=[-2,1,2,4,7,11]
12 target=13
13 print(two_sum_brute_force(A,target))
14 target=20
15 print(two_sum_brute_force(A,target))
16
17
```



Output

0.47s

```
2 11
True
False
```

Explanation

The outer loop on **line 4** iterates over all the elements in `A` while the inner loop on **line 5** iterates from the next index of `i` to the last index of `A` in every iteration of the outer loop. These nested loops help us form all possible pairs of array elements in every iteration. On **line 6**, we check if the pair `A[i]` and `A[j]` add up to the `target`. If they do, we have found a solution, and we return `True` on **line 8**. Otherwise, we move on to the next pair. If we are unable to find a solution after both the loops terminate, we return `False` on **line 9**.

This solution is highly inefficient as the code has a time complexity of $O(n^2)$ because of the nested loops. Let's see if we can improve it.

Solution 2

A slightly better approach time-wise, taking $O(n)$ time, but worse from a space standpoint, with space complexity of $O(n)$. In this approach, we make use of an auxiliary hash table to keep track of whether it's possible to construct the target based on the elements we've processed thus far in the array.



Implementation

```
1 # Time Complexity: O(n)
2 # Space Complexity: O(n)
3 def two_sum_hash_table(A, target):
4     ht = dict()
5     for i in range(len(A)):
6         if A[i] in ht:
7             print(ht[A[i]], A[i])
8             return True
9         else:
10            ht[target - A[i]] = A[i]
11    return False
12
13 A = [-2, 1, 2, 4, 7, 11]
14 target = 13
15
16 print(two_sum_hash_table(A, target))
```



Output

0.19s

2 11
True

Explanation

In the code above, we use a Python dictionary and name it `ht` on **line 4**. On **line 5**, using a for loop, we iterate over `A` and check if the element `A[i]` is present in `ht` or not (**line 6**). If it's not, we calculate `target - A[i]` and using that as a key, we store `A[i]` as its value in `ht` on **line 10**. Now it is easy for us to check for the next elements if we ever come across an element which is equal to `target - A[i]`. If `A[i]` in any iteration is present as a key in `ht`, the execution jumps to **line 7**. Note that we have been storing `target - A[i]` as keys and `A[i]` as values in the previous iterations. Therefore, if `A[i]` of the current iteration is present as a key in `ht`, we have found a pair as the sum of that key and its value will equal `target`. In this case, we print out the pair and return `True` (**lines 7-8**). On the other hand, if we never come across such a pair after all the iterations of the for loop, we return `False` on **line 11**.

As we iterate the list once, the time complexity for Solution 2 is $O(n)$ where n is the length of the list, and as we make an entry in the dictionary for every element in the list, the space complexity is also in $O(n)$.

Solution 3

This approach has a time complexity of $O(n)$ and a constant space complexity, $O(1)$. Here, we have two indices that we keep track of, one at the front and one at the back. We move either the left or right indices based on whether the sum of the elements at these indices is either greater than or less than the target element.

Implementation

```
1 #Time Complexity: O(n)
2 #Space Complexity: O(1)
3 def two_sum(A, target):
4     i = 0
5     j = len(A) - 1
6     while i < j:
7         if A[i] + A[j] == target:
```



```
8     .....print(A[i],A[j])
9     .....return True
10    .....elif A[i]+A[j]<target:
11    .....i+=1
12    .....else:
13    .....j-=1
14    ..return False
15
16    A=[-2,1,2,4,7,11]
17    target=13
18
19    print(two_sum(A,target))
```



Output

0.14s

2 11

True

Explanation

This approach assumes that the array is sorted. So i and j are set to 0 and $\text{len}(A) - 1$ respectively (**lines 4-5**), i.e., the minimum and the maximum value. On **line 6**, a while loop is set which will run as long as $i < j$. If at any iteration, $A[i] + A[j]$ equal target, we have found the pairs and we return True from the function after printing the pairs (**line 7-9**). However, if the sum of $A[i]$ and $A[j]$ is less than target, we increment i by 1 on **line 11**, so that $A[i]$ will be a greater value in the next iteration and will produce a greater sum than the current sum. On the other hand, if the sum of $A[i]$ and $A[j]$ is greater than target, we decrement j on **line 13** to

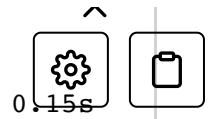
achieve a lesser sum by moving to an index with a smaller value. After the while loop terminates, we return **False (line 14)** as we never come across a pair which sums to `target` during the execution of the while loop.

There is no extra data structure to store values which implies that the code has a space complexity of $O(1)$ while the time complexity is $O(n)$ as we traverse the entire list `A` using `i` and `j`.

All the above solutions are implemented in the code widget below. Test them on your list and target value!

```
19     for i in range(len(A)):
20         if A[i] in ht:
21             print(ht[A[i]], A[i])
22             return True
23         else:
24             ht[target - A[i]] = A[i]
25     return False
26
27
28 # Time Complexity: O(n)
29 # Space Complexity: O(1)
30 def two_sum(A, target):
31     i = 0
32     j = len(A) - 1
33     while i < j:
34         if A[i] + A[j] == target:
35             print(A[i], A[j])
36             return True
37         elif A[i] + A[j] < target:
38             i += 1
39         else:
40             j -= 1
41     return False
42
43
44 print(two_sum_brute_force(A, target))
45 print(two_sum_hash_table(A, target))
46 print(two_sum(A, target))
```





Output

```
2 11
True
2 11
True
2 11
True
```

In the next lesson, we will look at the “Optimal Task Assignment” Problem.
See you there!

← Back

Next →

Arbitrary Precision Increment

Optimal Task Assignment



Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/two-sum-problem__arrays__data-structures-and-algorithms-in-python)



Optimal Task Assignment

In this lesson, you will learn how to assign tasks optimally to a set of workers in Python.

We'll cover the following



- Implementation
- Explanation

In this lesson, we will be solving the following problem:

Assign tasks to workers so that the time it takes to complete all the tasks is minimized given a count of workers and an array where each element indicates the duration of a task.

We wish to determine the optimal way in which to assign tasks to some workers. Each worker must work on exactly *two* tasks. Tasks are independent of each other, and each task takes a certain amount of time.

In the slides below, we have been given an array of tasks where the value of each element in the array corresponds to the number of hours required for each task.

Check out the slides below where an example of assigning tasks optimally to workers has been illustrated:



Tasks

6	3	2	7	5	5
---	---	---	---	---	---

Worker 1

Worker 2

Worker 3

Workers

1 of 5

Tasks

6	3	2	7	5	5
---	---	---	---	---	---

(x_1, y_1)

(x_2, y_2)

(x_3, y_3)

Worker 1

Worker 2

Worker 3

Workers

Each worker must be assigned two tasks.

2 of 5



Tasks

6	3	2	7	5	5
---	---	---	---	---	---

(6, 3)

Worker 1

(2, 7)

Worker 2

(5, 5)

Worker 3

Workers

Tasks have been assigned to the workers.

3 of 5

Tasks

6	3	2	7	5	5
---	---	---	---	---	---

(6, 3)

Worker 1

(2, 7)

Worker 2

(5, 5)

Worker 3

Workers

Tasks have been assigned to the workers.

$$\max(6+3, 2+7, 5+5) = 10$$

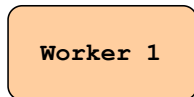
4 of 5



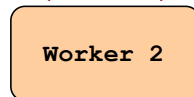
Tasks

6	3	2	7	5	5
---	---	---	---	---	---

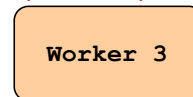
(6, 3)



(2, 7)



(5, 5)



Workers

$$\max(6+3, 2+7, 5+5) = 10$$

So 10 is the maximum number of hours that we have to wait for all the tasks to complete.

Also, it is the minimum number that we can get to solve all the tasks.

5 of 5



Now let's think of a general approach. From the above example, we know that we have to make pairs. If we generate all possible pairs, it would not be efficient as enumerating every possible pair would require generating $\frac{n(n-1)}{2}$ pairs where n is the number of tasks in the given array.

Therefore, we are going to make use of a different approach, i.e., the greedy approach.

In the greedy approach, we'll focus on the following rule:

- Pair the longest task with the shortest one.

Let's find out how to do this in the slides below.



Tasks

6	3	2	7	5	5
---	---	---	---	---	---



2	3	5	5	6	7
---	---	---	---	---	---

Sorted Tasks

1 of 5

Sorted Tasks

2	3	5	5	6	7
---	---	---	---	---	---



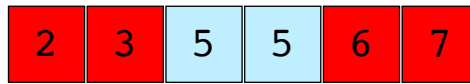
2	7
---	---

Pick the shortest and the longest one.

2 of 5



Sorted Tasks



Pick the shortest and the longest one.

3 of 5

Sorted Tasks



Pick the shortest and the longest one.

4 of 5



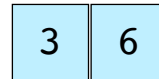
Sorted Tasks



9



10



9

We get the same number of hours as in the slides at the beginning of the lesson.

5 of 5



The time complexity for the algorithm depicted in the slides above will $O(n \log n)$ due to sorting.

Implementation

Now that you are familiar with the algorithm, let's jump to the code in Python:

```
1 A = [6, 3, 2, 7, 5, 5]
2
3 A = sorted(A)
4
5 for i in range(len(A)//2):
6     print(A[i], A[~i])
```





Explanation

Yes, the implementation was that simple! After sorting the array using sorted on **line 3**, the for loop on **line 5** iterates for half the length of the array and prints out the pairs using indexing. So `~i` on **line 6** is the bitwise complement operator which puts a negative sign in front of `i`. Thus, the negative numbers as indexes mean that you count from the right of the array instead of the left. So, `A[-1]` refers to the last element, `A[-2]` is the second-last, and so on. In this way, we are able to pair the numbers from the beginning of the array to the end of the array.

That pretty much does it for this lesson! I hope everything's clear so far. In the next lesson, we'll have a look at a problem involving sorted arrays. See you there!

[← Back](#)[Next →](#)[Two Sum Problem](#)[Intersection of Two Sorted Arrays](#)[Mark as Completed](#)[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/optimal-task-assignment__arrays__data-structures-and-algorithms-in-python)



Intersection of Two Sorted Arrays

In this lesson, you will learn how to find the intersection of two sorted arrays in Python.

We'll cover the following



- Algorithm
- Implementation
- Explanation

In this lesson, we will be solving the following problem:

Given two sorted arrays, A and B , determine their intersection. What elements are common to A and B ?



Array A

2	3	3	5	7	11
---	---	---	---	---	----

Array B

3	3	7	15	31
---	---	---	----	----



Intersection of A and B

3	7
---	---

There is a one-line solution to this problem in Python that will also work in the event when the arrays are not sorted.

```
1 A = [2, 3, 3, 5, 7, 11]
2 B = [3, 3, 7, 15, 31]
3
4 print(set(A).intersection(B))
```

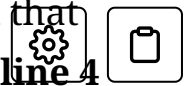


Output

0.2s

{3, 7}

The `set` function in Python operates on a list and returns a set object that contains the unique elements of that list. A is converted into a set on **line 4** by using `set(A)`. Now there is another built-in function in Python called `intersection` which operates on sets. In the code above, `intersection` returns a set which is the intersection of A and B. The method `intersection` is a member of the `Set` class, but it can accept lists as arguments too. This solution is fine considering it works on unsorted arrays as well. However, since we are aware that the arrays A and B are sorted, we can use this to our advantage and solve the problem in a way that leverages this and gives us a slightly better runtime.



Algorithm

In the algorithm that we'll use to solve this problem more efficiently, we will make use of two iterators, `i` and `j`, one for each array. If the `i`th element in `array1` is less than the `j`th element in `array2`, we increment the `i`th iterator to increase the value so that we can reach a match with `array2`. On the other hand, if `array1[i]` is greater than `array2[j]`, `j` will be incremented so that we can get a larger value from `array2`. If both the elements pointed to by the iterators are equal, we have found an intersection and both the iterators are incremented.

Additionally, we need to cater for an edge case of duplicates. If an intersection is recorded for some elements, we need to make sure that we don't count another intersection of their duplicates. In the slides below, this algorithm has been illustrated for you.



i

2	3	3	5	7	11
---	---	---	---	---	----

Array A

j

3	3	7	15	31
---	---	---	----	----

Array B

We'll traverse these arrays according to the rules defined in the algorithm.

1 of 13

i

2	3	3	5	7	11
---	---	---	---	---	----

Array A

j

3	3	7	15	31
---	---	---	----	----

Array B

```
if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++
```

2 of 13



i

2	3	3	5	7	11
---	---	---	---	---	----

Array A

j

3	3	7	15	31
---	---	---	----	----

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

3 of 13

i

2	3	3	5	7	11
---	---	---	---	---	----

Array A

j

3	3	7	15	31
---	---	---	----	----

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

4 of 13



	<i>i</i>
2	
3	
3	
5	
7	
11	

Array A

<i>j</i>
3
3
7
15
31

Array B

```

    if A[i] < B[j]
        i++
    if A[i] > B[j]
        j++
    if A[i] == B[j] and A[i] != A[i-1]
        Intersection Found!
        i++; j++
    if A[i] == B[j]
        i++; j++

```

Intersection:
[3]

5 of 13

	<i>i</i>
2	
3	
3	
5	
7	
11	

Array A

<i>j</i>
3
3
7
15
31

Array B

```

    if A[i] < B[j]
        i++
    if A[i] > B[j]
        j++
    if A[i] == B[j] and A[i] != A[i-1]
        Intersection Found!
        i++; j++
    if A[i] == B[j]
        i++; j++

```

Intersection:
[3]

6 of 13



	<i>i</i>
2	
3	
3	
5	
7	
11	

Array A

	<i>j</i>
3	
3	
7	
15	
31	

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3]

7 of 13

	<i>i</i>
2	
3	
3	
5	
7	
11	

Array A

	<i>j</i>
3	
3	
7	
15	
31	

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3]

8 of 13



				<i>i</i>	
2	3	3	5	7	11

Array A

		<i>j</i>		
3	3	7	15	31

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3]

9 of 13

					<i>i</i>
2	3	3	5	7	11

Array A

		<i>j</i>		
3	3	7	15	31

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3,7]

10 of 13



2	3	3	5	7	11
---	---	---	---	---	----

Array A

3	3	7	15	31
---	---	---	----	----

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3,7]

11 of 13

2	3	3	5	7	11
---	---	---	---	---	----

Array A

3	3	7	15	31
---	---	---	----	----

Array B

```

if A[i] < B[j]
    i++
if A[i] > B[j]
    j++
if A[i] == B[j] and A[i] != A[i-1]
    Intersection Found!
    i++; j++
if A[i] == B[j]
    i++; j++

```

Intersection:
[3,7]

12 of 13



2	3	3	5	7	11
---	---	---	---	---	----

Array A

3	3	7	15	31
---	---	---	----	----

Array B

Final Intersection:
[3,7]

13 of 13



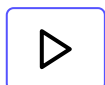
Implementation

As evident from the slides above, the solution will be complete once any iterator is done traversing its respective array. Now all that is left for us is to code the solution in Python. Let's go!

```
1 def intersect_sorted_array(A, B):
2     i = 0
3     j = 0
4     intersection = []
5
6     while i < len(A) and j < len(B):
7         if A[i] == B[j]:
8             if i == 0 or A[i] != A[i - 1]:
9                 intersection.append(A[i])
10            i += 1
11            j += 1
12        elif A[i] < B[j]:
```



```
13         i += 1
14     else:
15         j += 1
16     return intersection
17
18 A = [2, 3, 3, 5, 7, 11]
19 B = [3, 3, 7, 15, 31]
20
21 print(intersect_sorted_array(A, B))
```



Output

0.26s

```
[3, 7]
```

Explanation

i and j have been set to 0 on **lines 2-3**. `intersection` has been initialized to an empty list on **line 4**. The `while` loop on **line 6** is where the crux of the algorithm lies. The `while` loop will terminate as soon as i or j becomes equal or greater than the length of the arrays they are iterating on. The conditions from **lines 7-15** are reflective of the conditions discussed in the algorithm with some slight variations.

One of these conditions is when we check for duplicates using the condition $(A[i] \neq A[i - 1])$ on **line 8**. In that case, we need to handle the code when i equals 0 . Therefore, we have an additional check on **line 8**, i.e., $i == 0$. Therefore, if i equals 0 , there is no need to check for the duplicate condition $(A[i] \neq A[i - 1])$ given on **line 8**. Instead, $A[i]$ is appended to `intersection` if $A[i]$ equals $B[j]$ and i equals 0 .

After the `while` loop terminates, `intersection` is returned from the function on **line 16**.



I hope you have enjoyed the chapter so far. In the next lesson, we have an interesting challenge waiting for you!



Exercise: Buy and Sell Stock

Challenge yourself with an exercise in which you'll have to find the maximum profit generated by buying and selling stocks!

We'll cover the following



- Problem
- Coding Time!

Problem

Given an array of numbers consisting of daily stock prices, calculate the maximum amount of profit that can be made from buying on one day and selling on another.

In an array of prices, each index represents a day, and the value on that index represents the price of the stocks on that day.

Here is the way to calculate the profit:

$$\text{Profit} = \text{Selling Price} - \text{Buying Price}$$

Note that you need to buy the stocks before you sell them so the day (index) indicating the buying price should be before the day (index) indicating the selling price.

Have a look at an example illustrated below:



	310	315	275	295	260	270	290	230	255	250
Day	0	1	2	3	4	5	6	7	8	9

Maximum Profit = 30

Buying Price = 260

Selling Price = 290

If stocks are bought on day 4 for a price of 260 and sold on day 6 for a price of 290, we end up with a maximum profit of 30.

Coding Time!

Your task is to return the maximum profit from the function `buy_and_sell_stock_once(prices)` given in the code widget below. The input parameter `prices` is the array of integers that contains the price of stocks at each day where a day is represented by the index.

Good luck!

```
1 def buy_and_sell_stock_once(prices):
2     profit = 0
3     i, j = 0, 1
4     while j < len(prices):
5         if prices[i] > prices[j]:
6             i += 1
7             if i == j:
8                 j += 1
9         else:
10            if prices[j] - prices[i] > profit:
11                profit = prices[j] - prices[i]
12            else:
13                j += 1
14    return profit
```



Show Results

Show Console

0.21s

4 of 4 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	buy_and_sell_stock_once([310, 315, 275, ...])	30	30	Succeeded
✓	buy_and_sell_stock_once([100, 180, 260, ...])	655	655	Succeeded
✓	buy_and_sell_stock_once([50, 40, 30, 20, ...])	0	0	Succeeded
✓	buy_and_sell_stock_once([110, 215, 180, ...])	225	225	Succeeded

← Back

Next →

Intersection of Two Sorted Arrays

Solution Review: Buy and Sell Stock

☒ Mark as Completed

Report an Issue

Ask a Question

(https://discuss.educative.io/tag/exercise-buy-and-sell-stock__arrays__data-structures-and-algorithms-in-python)



Solution Review: Buy and Sell Stock

This lesson contains the solution review for the challenge of finding the maximum profit generated by buying and selling stocks!

We'll cover the following



- Solution 1: Brute Force
 - Implementation
 - Explanation
- Solution 2: Tracking Min Price
 - Implementation
 - Explanation

In this lesson, we will review the solution to the problem in the last exercise. Let's go over the problem statement once again:

Given an array of numbers consisting of daily stock prices, calculate the maximum amount of profit that can be made from buying on one day and selling on another day.

We consider two approaches to this problem. In the first, we consider a brute force approach that solves the problem in $O(N^2)$, where N is the size of the array of numbers. We then improve upon this solution to take our solution to a time complexity of $O(N)$.

Solution 1: Brute Force

Implementation

```
1 A = [310, 315, 275, 295, 260, 270, 290, 230, 255,
2 # Time Complexity: O(n^2)
3 # Space Complexity: O(1)
4 def buy_and_sell_once(A):
5     max_profit = 0
6     for i in range(len(A)-1):
7         for j in range(i+1, len(A)):
8             if A[j] - A[i] > max_profit:
9                 max_profit = A[j] - A[i]
10    return max_profit
11
12 print(buy_and_sell_once(A))
```



Output

0.51s

30

Explanation

This brute force solution is an intuitive approach. We set `max_profit` equal to `0` on **line 5**. The for loop on **line 6** iterates over every element in `A`. Now for every element of `A`, we iterate over all the elements in `A` which are at a greater position than that element in `A` using the for loop on **line 7**. In this way, we calculate all possible profits by selling with all possible prices after we buy the stocks. Now if the calculated profit, i.e., `A[j] - A[i]` is greater than the `max_profit`, we update `max_profit` to its new greater value on **line 9**. Finally, we return `max_profit` on **line 10** which will be the maximum possible profit that we got by buying stocks on each day and selling them on all possible days after the day we buy them. Therefore, this solution has a

time complexity of $O(n^2)$ as we generate all the possible buying and selling combinations using nested for loops. Now we need to think of a better solution. Let's check out Solution 2.



Solution 2: Tracking Min Price


In solution 2, for each index, we calculate the most profit we can make by selling at that time. So, for a given day, the maximum profit can be made if the stock were bought at the minimum price on an earlier day. Therefore, we maintain the minimum price seen so far and subtract it from every future price to keep track of the maximum profit.

We iterate the array once and keep track of the minimum buying price and the maximum profit. We calculate the profit at each iteration by subtracting the minimum buying price seen so far from the current price in each iteration and updating the maximum profit accordingly.

Have a look at the slides below to understand the algorithm:

Profits

	310	315	275	295	260	270	290	230	255	250
Day	0	1	2	3	4	5	6	7	8	9



Minimum Price = 310

Maximum Profit = 0

1 of 12



Profits 0

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 310

Selling Price = 310

Profit = $310 - 310 = 0$

Maximum Profit = 0

2 of 12

Profits 0 5

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 310

Selling Price = 315

Profit = $315 - 310 = 5$

Maximum Profit = 5

3 of 12



Profits 0 5 0

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 275

Selling Price = 275

Profit = 275 - 275 = 0

Maximum Profit = 5

4 of 12

Profits 0 5 0 20

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 275

Selling Price = 295


Profit = 295 - 275 = 20

Maximum Profit = 20

5 of 12



<i>Profits</i>	0	5	0	20	0					
	310	315	275	295	260	270	290	230	255	250
<i>Day</i>	0	1	2	3	4	5	6	7	8	9



Minimum Buying Price = 260


Selling Price = 260

Profit = 260 - 260 = 0

Maximum Profit = 20

6 of 12

<i>Profits</i>	0	5	0	20	0	10				
	310	315	275	295	260	270	290	230	255	250
<i>Day</i>	0	1	2	3	4	5	6	7	8	9



Minimum Buying Price = 260

Selling Price = 270

Profit = 270 - 260 = 10

Maximum Profit = 20

7 of 12



Profits 0 5 0 20 0 10 30

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 260

Selling Price = 290

Profit = 290 - 260 = 30

Maximum Profit = 30

8 of 12

Profits 0 5 0 20 0 10 30 0

310	315	275	295	260	270	290	230	255	250
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Day 0 1 2 3 4 5 6 7 8 9



Minimum Buying Price = 230

Selling Price = 230

Profit = 230 - 230 = 0

Maximum Profit = 30

9 of 12



<i>Profits</i>	0	5	0	20	0	10	30	0	25	
	310	315	275	295	260	270	290	230	255	250
<i>Day</i>	0	1	2	3	4	5	6	7	8	9



Minimum Buying Price = 230

Selling Price = 255

Profit = 255 - 230 = 25

Maximum Profit = 30

10 of 12

<i>Profits</i>	0	5	0	20	0	10	30	0	25	20
	310	315	275	295	260	270	290	230	255	250
<i>Day</i>	0	1	2	3	4	5	6	7	8	9



Minimum Buying Price = 230

Selling Price = 250

Profit = 250 - 230 = 20

Maximum Profit = 30

11 of 12

<i>Profits</i>	0	5	0	20	0	10	30	0	25	20
	310	315	275	295	260	270	290	230	255	250
<i>Day</i>	0	1	2	3	4	5	6	7	8	9

Minimum Buying Price = 230

Selling Price = 250

Profit = 250 - 230 = 20

Maximum Profit = 30

12 of 12

— []

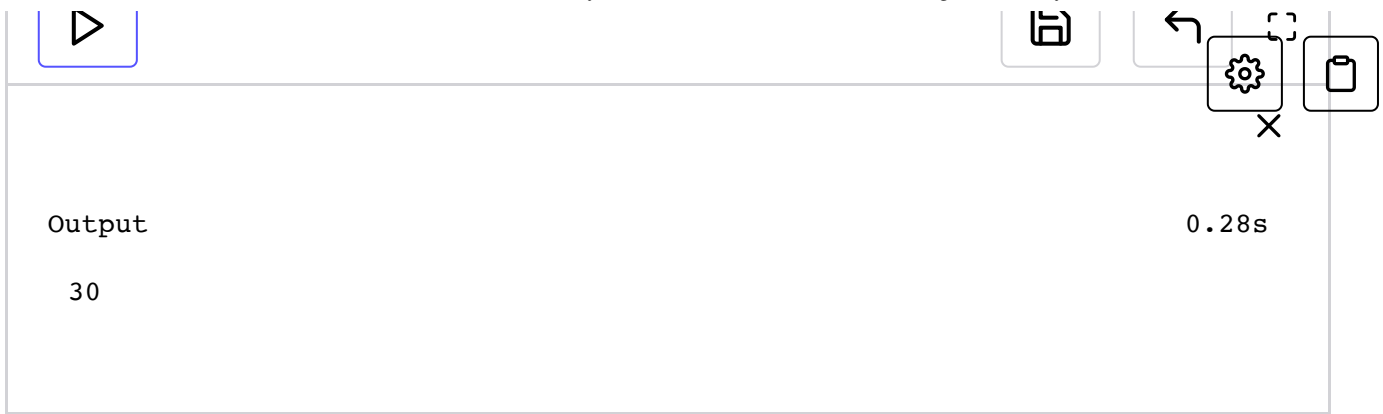
Implementation

Let's jump to the implementation in Python of the algorithm illustrated above:

```

1 A = [310, 315, 275, 295, 260, 270, 290, 230, 255,
2
3 # Time Complexity: O(n)
4 # Space Complexity: O(1)
5 def buy_and_sell_once(prices):
6     max_profit = 0.0
7     min_price = float('inf')
8     for price in prices:
9         min_price = min(min_price, price)
10        compare_profit = price - min_price
11        max_profit = max(max_profit, compare_profit)
12    return max_profit
13
14 print(buy_and_sell_once(A))

```



Output

30

0.28s

Explanation

We set `max_profit` and `min_price` to 0 and infinity respectively (**lines 6-7**). The list `prices` is iterated using a for loop on **line 8**. As we are supposed to keep track of the minimum price, we update `min_price` using the `min` function on **line 9** where `min_price` is the minimum amount between `min_price` and `price` of the current iteration. In the next line, we store the calculated profit (`price - min_price`) in `compare_profit`. As we also need to keep a check on the `max_profit`, we update `max_profit` to the maximum value between `max_profit` and `compare_profit` on **line 11**. After we are done with the iteration of `prices`, we return `max_profit` on **line 12**.

I hope everything is clear so far and you were able to enjoy the exciting problems regarding arrays in Python in this chapter.

Up next we have **Binary Trees**. Stay tuned!

[← Back](#)[Exercise: Buy and Sell Stock](#)[Next →](#)[Introduction](#)☒ Mark as Completed[Report an Issue](#)[Ask a Question](#)

(https://discuss.educative.io/tag/solution-review-buy-and-sell-stock__arrays__data-structures-and-algorithms-in-python)

