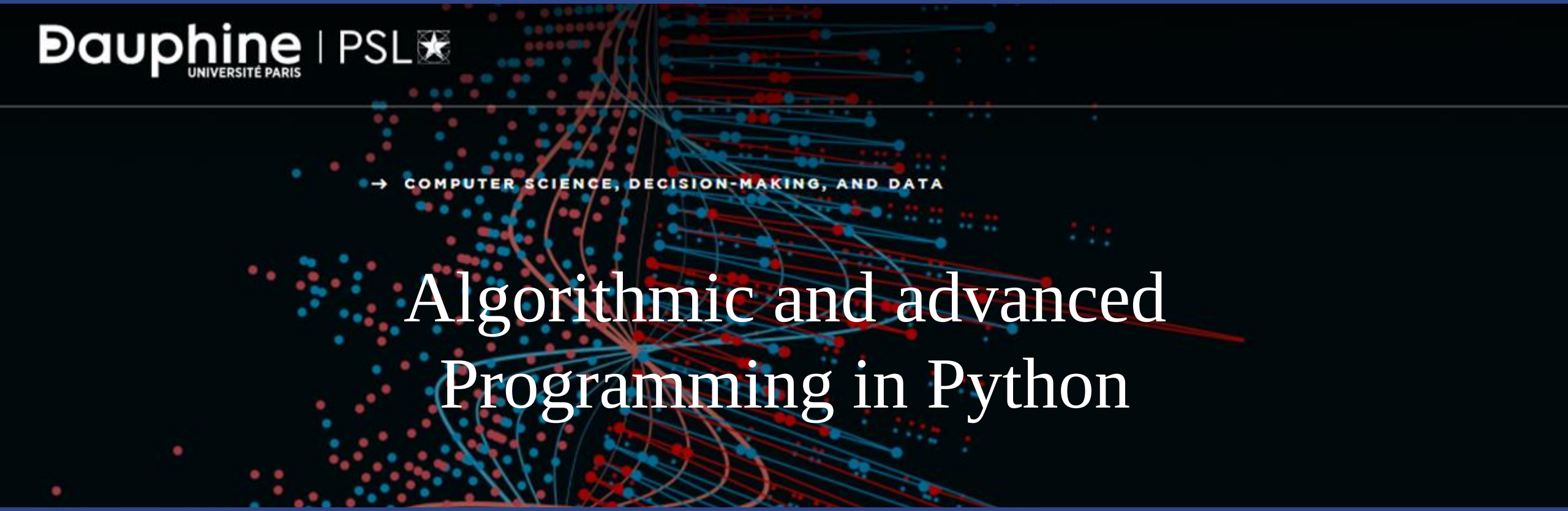


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 8

Outline

1. What is Searching
2. Type of Searching
3. Unordered Linear Search
4. Sorted/Ordered Linear Search
5. Binary Search
6. Interpolation Search
7. Bitonic search

Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask

- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Reminder of previous session

- In Master class 8, we discuss about Gradient boosting decision trees?
- Question: can someone summarize it?

What is Searching?

- In computer science, *searching* is the process of finding an item with specified properties from a collection of items.

- The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or they may be elements of other search spaces. .

- Question: why do we need searching?

We need searching!

- *Searching* is one of the core computer science algorithms.

Google is your best friend!

- We know that today's computers store a lot of information. To retrieve this information proficiently we need very efficient searching algorithms.

- Question: how is this related to our data structure class?

Data structure is fundamental!

- To retrieve this information proficiently we need very efficient searching algorithms.
- There are certain ways of organizing the data that improves the searching process. That means, if we keep the data in proper order, it is easy to search the required element.

- Sorting is one of the techniques for making the elements ordered. In this chapter we will see different searching algorithms.

Types of Searching

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Interpolation search
- Binary Search Trees (this is related to our chapter on trees)

Unordered Linear Search

- Let us assume we are given an array where the order of the elements is not known. That means the elements of the array are not sorted.

- In this case, to search for an element we have to scan the complete array and see if the element is there in the given list or not..

```
def UnOrderedLinearSearch (numbersList, value):
    for i in range(len(numbersList)):
        if numbersList[i] == value:
            return i
    return -1
A = [534,246,933,127,277,321,454,565,220]
print(UnOrderedLinearSearch(A,277))
```

- This is obviously very inefficient
- Question: what is the complexity?
- Question: How can we improve?

Sorted/Ordered Linear Search

- If the elements of the array are already sorted, then in many cases we don't have to scan the complete array to see if the element is there in the given array or not.

- In the algorithm below, it can be seen that, at any point if the value at $A[i]$ is greater than the $data$ to be searched, then we just return -1 without searching the remaining array.

```
def OrderedLinearSearch (numbersList, value):
    for i in range(len(numbersList)):
            if numbersList[i] == value:
                return i
            elif numbersList[i] > value:
                return -1
    return -1
A = [34,46,93,127,277,321,454,565,1220]
print(OrderedLinearSearch(A,565))
```

Sorted/Ordered Linear Search

- Question: what is Time complexity and space complexity?

Sorted/Ordered Linear Search

- Time complexity of this algorithm is O(n). This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is the same.

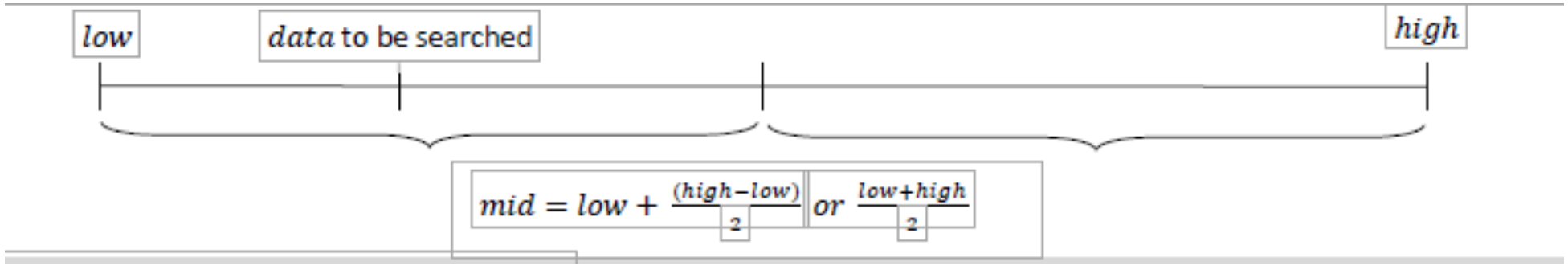
- Space complexity: O(1).
- Note: For the above algorithm we can make further improvement by incrementing the index at a faster rate (say, 2). This will reduce the number of comparisons for searching in the sorted list.

Can we improve searching?

- Question: can you find of a mechanism to improve searching?

Binary Search

- Let us consider the problem of searching a word in a dictionary. Typically, we directly go to some approximate page [say, middle page] and start searching from that point. If the *name* that we are searching is the same then the search is complete. If the page is before the selected pages then apply the same process for the first half; otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.



Code: iterative version

```
def BinarySearchIterative(numbersList, value):
    low = 0
    high = len(numbersList) - 1
    while low <= high:
        mid = (low + high) // 2
        if numbersList[mid] > value: high = mid - 1
        elif numbersList[mid] < value: low = mid + 1
        else: return mid
    return -1

A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
print(BinarySearchIterative(A, 277))
```

And recursive version

```
def BinarySearchRecursive(numbersList, value, low=0, high=-1):
    if not numbersList: return -1
    if(high == -1): high = len(numbersList) - 1
    if low == high:
        if numbersList[low] == value: return low
        else: return -1
    mid = low + (high - low) // 2
    if numbersList[mid] > value: return BinarySearchRecursive(numbersList, value, low, mid - 1)
    elif numbersList[mid] < value: return BinarySearchRecursive(numbersList, value, mid + 1, high)
    else: return mid

A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
print(BinarySearchRecursive(A, 277))
```

What is the complexity?

- Question: what is the complexity?

Complexity

- Recurrence for binary search is $T(n) = T(\mathrm{N}/2) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using $Divide\ and\ Conquer$ master theorem, we get, $T(n) = \mathrm{O}(logn)$.

- Time Complexity: $\mathrm{O}(logn)$. Space Complexity: O(1) [for iterative algorithm].

Alternative?

- Question: can we do better than binary search?

Interpolation Search

- Undoubtedly binary search is a great algorithm for searching with average running time complexity of $logn$.

- It always chooses the middle of the remaining search space, discarding one half or the other, again depending on the comparison between the key value found at the estimated (middle) position and the key value sought. The remaining search space is reduced to the part before or after the estimated position.

Interpolation Search

- In the mathematics, interpolation is a process of constructing new data points within the range of a discrete set of known data points. In computer science, one often has a number of data points which represent the values of a function for a limited number of values of the independent variable. It is often required to interpolate (i.e. estimate) the value of that function for an intermediate value of the independent variable.

Interpolation Search

- For example, suppose we have a table like this, which gives some values of an unknown function f. Interpolation provides a means of estimating the function at intermediate points, such as x = 5.5.

x	$f(x)$
1	10
2	20
3	30
4	40
5	50
6	60
7	70

Question: What kind of interpolation can we do?

Linear interpolation

- There are many different interpolation methods, and one of the simplest methods is linear interpolation.

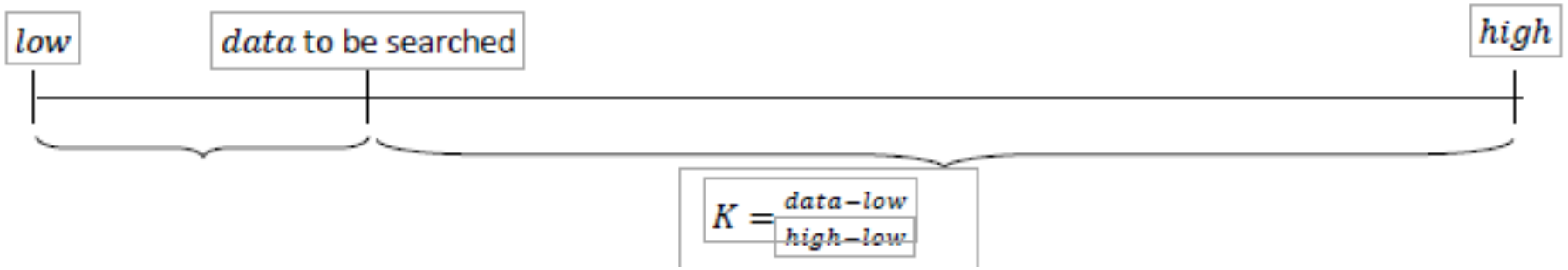
- Consider the above example of estimating f(5.5). Since 5.5 is midway between 5 and 6, it is reasonable to take $f(5.5)$ midway between $f(5)$ = 50 and $f(6)$ = 60, which yields 55 ((50+60)/2).

- Linear interpolation takes two data points, say ($x1$, $y1$) and ($x2$, $y2$, and the interpolant is given by:

$$y = y_1 + (y_2 - y_1)\frac{x - x_1}{x_2 - x_1} \; at \; point \; (x, y)$$

Interpolation search

- With above inputs, what will happen if we don't use the constant ½, but another more accurate constant "K", that can lead us closer to the searched item.



Intuition

- This algorithm tries to follow the way we search a name in a phone book, or a word in the dictionary.
- We, humans, know in advance that in case the name we're searching starts with a "m", like "monk" for instance, we should start searching near the middle of the phone book. Thus if we're searching the word "career" in the dictionary, you know that it should be placed somewhere at the beginning.
- This is because we know the order of the letters, we know the interval (a-z), and somehow we intuitively know that the words are dispersed equally.
- These facts are enough to realize that the binary search can be a bad choice.
- Indeed the binary search algorithm divides the list in two equal sub-lists, which is useless if we know in advance that the searched item is somewhere in the beginning or the end of the list.
- Yes, we can use also jump search if the item is at the beginning, but not if it is at the end, in that case this algorithm is not so effective.

Comparing with binary search

- The interpolation search algorithm tries to improve the binary search. The question is how to find this value? Well, we know bounds of the interval and looking closer to the image above we can define the following formula.

$$K = \frac{data - low}{high - low}$$

- This constant K is used to narrow down the search space. For binary search, this constant K is $(low + high)/2$.

- Now we can be sure that we're closer to the searched value. On average the interpolation search makes about $log(logn)$ comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to O(n) comparisons.

- In interpolation-sequential search, interpolation is used to find an item near the one being searched for, then linear search is used to find the exact item

Efficiency

- Question: when is this algorithm appropriate and efficient?

Efficiency

- Question: when is this algorithm appropriate and efficient?
- For this algorithm to give best results, the dataset should be ordered and uniformly distributed

Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Average Case
Unordered Array	n	$n/2$
Ordered Array (Binary Search)	$logn$	$logn$
Unordered List	n	$n/2$
Ordered List	n	$n/2$
Binary Search Trees (for skew trees)	n	$logn$
Interpolation search	n	$log(logn)$

Application

- Given an array of n numbers, give an algorithm for checking whether there are any duplicate elements in the array or not?

- Question: any solution?

Application

- Given an array of n numbers, give an algorithm for checking whether there are any duplicate elements in the array or not?

- This is one of the simplest problems. One obvious answer to this is exhaustively searching for duplicates in the array. That means, for each input element check whether there is any element with the same value. This we can solve just by using two simple for loops. The code for this solution can be given as:

Check duplicates brute force

```
def CheckDuplicatesBruteForce(A):
    for i in range(0, len(A)):
        for j in range(i + 1, len(A)):
            if(A[i] == A[j]):
                print("Duplicates exist:", A[i])
                return;
    print("No duplicates in given array.")

A = [3, 2, 10, 20, 22, 32]
CheckDuplicatesBruteForce(A)
A = [3, 2, 1, 2, 2, 3]
CheckDuplicatesBruteForce(A)
```

- Question: time and space complexity?

Check duplicates brute force

```
def CheckDuplicatesBruteForce(A):
    for i in range(0, len(A)):
        for j in range(i + 1, len(A)):
            if(A[i] == A[j]):
                print("Duplicates exist:", A[i])
                return;
    print("No duplicates in given array.")

A = [3, 2, 10, 20, 22, 32]
CheckDuplicatesBruteForce(A)
A = [3, 2, 1, 2, 2, 3]
CheckDuplicatesBruteForce(A)
```

- Time Complexity: O(n^2), for two nested *for* loops.
- Space Complexity: O(1).

Question: can we improve?

Can we improve?

- **Yes, Sort the given array.**

- After sorting, all the elements with equal values will be adjacent. Now, do another scan on this sorted array and see if there are elements with the same value and adjacent.

Corresponding algorithm

```
def CheckDuplicatesSorting(A):
    A.sort()
    for i in range(0, len(A) - 1):
        if(A[i] == A[i + 1]):
            print("Duplicates exist:", A[i])
            return
    print("No duplicates in given array.")

A = [33, 2, 10, 20, 22, 32]
CheckDuplicatesSorting(A)
A = [3, 2, 1, 2, 2, 3]
CheckDuplicatesSorting(A)
```

- Question: complexity?

Complexity

- Time Complexity: O($nlogn$), for sorting (assuming $nlogn$ sorting algorithm). Space Complexity: O(1).

Can you think about another solution?

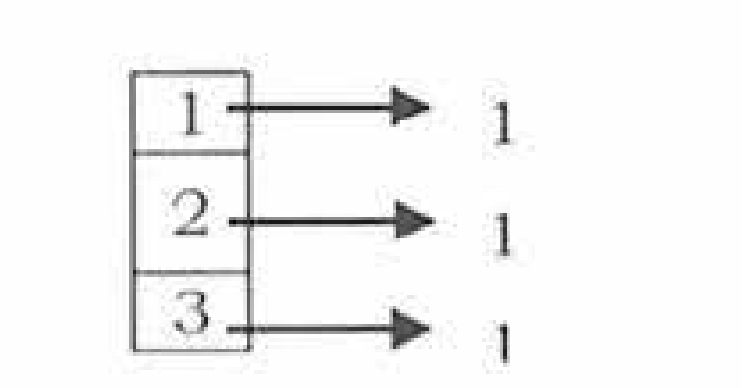
Another solution?

- Yes, using hash table. Hash tables are a simple and effective method used to implement dictionaries. $Average$ time to search for an element is O(1), while worst-case time is O(n). Refer to $Hashing$ chapter for more details on hashing algorithms. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

- Scan the input array and insert the elements into the hash. For each inserted element, keep the $counter$ as 1 (assume initially all entires are filled with zeros). This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting the first three elements 3, 2 and 1):

Another solution with hashing

- Now if we try inserting 2, since the counter value of 2 is already 1, we can say the element has appeared twice.



- Time Complexity: O(n). Space Complexity: O(n).

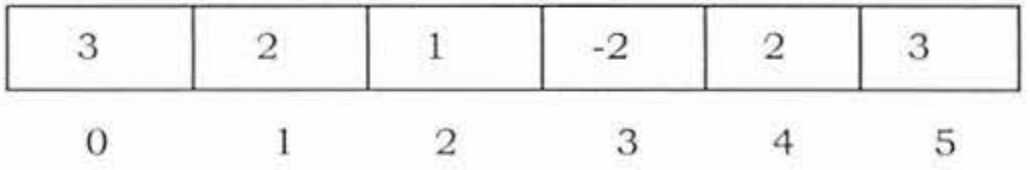
Another improvement

- Let us assume that the array elements are positive numbers and also all the elements are in the range 0 to $n - 1$. For each element $A[i]$, go to the array element whose index is $A[i]$. That means select $A[A[i]]$ and mark - $A[A[i]]$ (negate the value at $A[A[i]]$). Continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

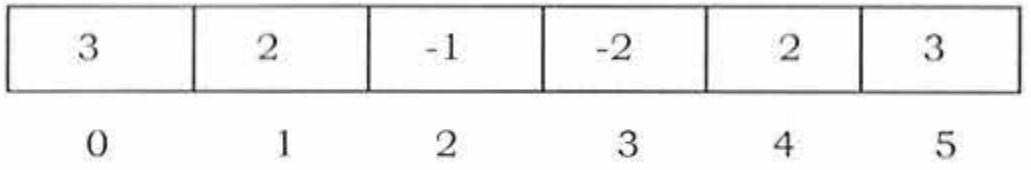
steps

Initially,

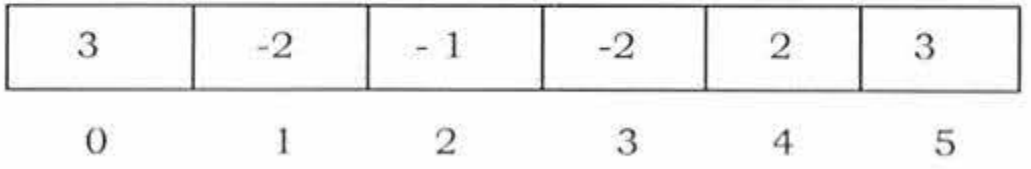
At step-1, negate A[abs(A[0])],



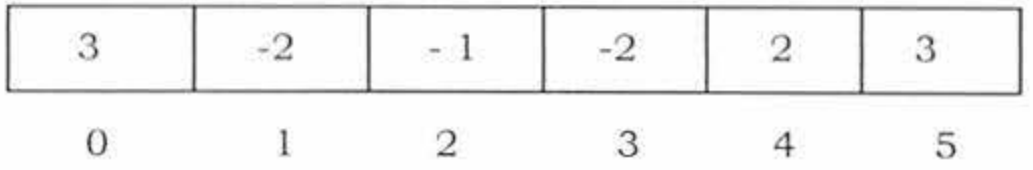
At step-2, negate A[abs(A[1])],



At step-3, negate A[abs(A[2])],



At step-4, negate A[abs(A[3])],



steps

At step-4, observe that $A[abs(A[3])]$ is already negative. That means we have encountered the same value twice.

```
import math
def CheckDuplicatesNegationTechnique(A):
    A.sort()
    for i in range(0,len(A)):
        if(A[abs(A[i])] < 0):
            print("Duplicates exist:", A[i])
            return
        else:
            A[A[i]] = - A[A[i]]
    print("No duplicates in given array.")
A = [3,2,1,2,2,3]
CheckDuplicatesNegationTechnique(A)
```

Question: complexity?

Complexity

- Time Complexity: O(n). Since only one scan is required. Space Complexity: O(1).

Notes:

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to $n-1$ then it may give exceptions.

A little bit more challenging

- **Two elements whose sum is closest to zero**:
- Given an array with both positive and negative numbers, find the two elements such that their sum is closest to zero. For the below array, algorithm should give −80 and 85. Example: 1 60 − 10 70 − 80 85.

- Question: can you solve this?

Brute force approach

- The brute force approach is simple. Loop through each element $A[i]$ and find if there is another value that equals to $target - A[i]$.

```
def TwoElementsClosestToZero(A):
    n = len(A)
    if(n < 2):
        print("Invalid Input")
        return
    minLeft = 0
    minRight = 1
    minSum = A[0] + A[1]
    for l in range(1, n - 1):
        for r in range(l + 1, n):
            sum = A[l] + A[r];
            if(abs(minSum) > abs(sum)):
                minSum = sum
                minLeft = l
                minRight = r
    print(" The two elements whose sum is minimum are ", A[minLeft], A[minRight])

A = [1, 60, -10, 70, -80, 85]
TwoElementsClosestToZero(A)
A = [10, 8, 3, 5, -9, -7, 6]
TwoElementsClosestToZero(A)
```

Can we improve?

- Yes, using sorting

Sorting solution

Algorithm:

1. Sort all the elements of the given input array.
2. Maintain two indexes, one at the beginning ($i = 0$) and the other at the ending ($j = n - 1$). Also, maintain two variables to keep track of the smallest positive sum closest to zero and the smallest negative sum closest to zero.
3. While $i < j$:
    a. If the current pair sum is > zero and < postiveClosest then update the postiveClosest. Decrement j.
    b. If the current pair sum is < zero and > negativeClosest then update the negativeClosest. Increment i.
    c. Else, print the pair.

Sorting solution

```
import sys
def TwoElementsClosestToZero(A):
    n = len(A)
    A.sort()
    if(n < 2):
        print("Invalid Input")
        return
    l = 0
    r = n - 1
    minLeft = l
    minRight = n - 1
    minSum = sys.maxint
    while(l < r):
        sum = A[l] + A[r];
        if(abs(minSum) > abs(sum)):
            minSum = sum
            minLeft = l
            minRight = r
        if sum < 0:
            l += 1
        else: r -= 1
    print(" The two elements whose sum is minimum are ", A[minLeft], A[minRight])

A = [1, 60, -10, 70, -80, 85]
TwoElementsClosestToZero(A)
A = [10, 8, 3, 5, -9, -7, 6]
TwoElementsClosestToZero(A)
```

- Time Complexity: O($nlogn$), for sorting. Space Complexity: O(1).

Variation

- **Find elements whose sum is closest to given target.**

- Given an array with both positive and negative numbers, find the two elements such that their sum is closest to given target. Given A = [2, 7, 11, 15], target = 9. Because, A[0] + A[1] = 2 + 7 = 9, return [0, 1].

Brute force

```
def twoElementsWithSumKBruteForce(A, K):
    n = len(A)
        for i in range(0, n):
        for j in range(i + 1, n):
            if(A[i] + A[j] == K):
                return 1
    return 0

A = [1, 4, 45, 6, 10, -8]
A.sort()
print twoElementsWithSumKBruteForce(A, 111)
```

A better solution

- **Solution**: To improve our run time complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to look up its index. What is the best way to maintain a mapping of each element in the array to its index? A hash table.
- We reduce the look up time from O(n) to O(1) by trading space for speed. A hash table is built exactly for this purpose, it supports fast look up in near constant time. I say "near" because if a collision occurred, a look up could degenerate to O(n) time. But, look up in hash table should be amortized O(1) time as long as the hash function was chosen carefully.

A simple implementation

- A simple implementation uses two iterations. In the first iteration, we add each element's value and its index to the table. Then, in the second iteration we check if each element's complement ($target - A[i]$) exists in the table. Beware that the complement must not be $A[i]$ itself!

- Time complexity: O(n). We traverse the list containing n elements exactly twice. Since the hash table reduces the look up time to O(1), the time complexity is O(n). Space complexity: O(n). The extra space required depends on the number of items stored in the hash table, which stores exactly n elements.

Simple implementation

```
def twoElementsWithSumKWithHash(A, K):
    table = {}  # hash
    for element in A:
        if element in table:
            table[element] += 1
        elif element != " ":
            table[element] = 1
        else:
            table[element] = 0
    for element in A:
        if K - element in table:
            print("yes-->", element, "+", K - element, " = ", K )
A = [1, 4, 45, 6, 10, -8]
A.sort()
twoElementsWithSumKWithHash(A, 11)
```

In one pass!

- It turns out we can do it in one-pass. While we iterate and inserting elements into the table, we also look back to check if current element's complement already exists in the table. If it exists, we have found a solution and return immediately.

In one pass

```
def twoElementsWithSumKWithHash_one_pass(A, K):
    table = {}  # hash
    for element in A:
        if element in table:
            table[element] += 1
        elif element != " ":
            table[element] = 1
        else:
            table[element] = 0
    if K - element in table:
        print("yes-->", element, "+", K - element, " = ", K )
        return
A = [1, 4, 45, 6, 10, -8]
A.sort()
twoElementsWithSumKWithHash_one_pass(A, 11)
```

Complexity

- Time complexity: O(n). We traverse the list containing n elements only once. Each look up in the table costs only O(1) time. Space complexity: O(n). The extra space required depends on the number of items stored in the hash table, which stores at most nn elements.

Bitonic search

- Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $\leq k \leq n$ such that $A[1], \ldots, A[k]$ is an increasing sequence and $A[k+1], \ldots, A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding k.

- **Similar question**: Let us assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing functions]. In this array find the starting index of the positive numbers. Assume that we know the length of the input array. Design a O($logn$) algorithm.

Bitonic search

- **Solution**: Let us use a variant of the binary search.

Algorithm

- Find the mid element of the array.
- If *mid* element > *first element of array* this means that we need to look for the inflection point on the right of *mid*.
- If *mid* element < *first element of array* this that we need to look for the inflection point on the left of mid.
- We stop our search when we find the inflection point, when either of the two conditions is satisfied:
  - A[*mid*] > A[*mid* + 1] Hence, *mid* +1 is the smallest.
  - A[*mid* - 1] > A[*mid*] Hence, *mid* is the smallest.

Corresponding code

```
def findMinimumInRotatedSortedArray(A):
        mid, low, high = 0, 0, len(A) - 1
        while A[low] >= A[high]:
            if high - low <= 1:
                return A[high], high
            mid = (low + high) >> 1
            if A[mid] == A[low]:
                low += 1
            elif A[mid] > A[low]:
                low = mid
            elif A[mid] == A[high]:
                high -= 1
            else:
                high = mid
        return A[low], low

A = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]
print findMinimumInRotatedSortedArray(A)
```

In Lab session

- Lab is done by Remy Belmonte next week