10.10 Heap Sort

Heapsort is a comparison-based sorting algorithm and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case $\Theta(nlogn)$ runtime. Heapsort is an in-place algorithm but is not a stable sort.

Performance

Worst case performance: $\Theta(nlogn)$
Best case performance: $\Theta(nlogn)$
Average case performance: $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ total, $\Theta(1)$ auxiliary

For other details on Heapsort refer to the *Priority Queues* chapter.

10.11 Quick Sort

Quick sort is an example of a divide-and-conquer algorithmic technique. It is also called *partition exchange sort*. It uses recursive calls for sorting the elements, and it is one of the famous algorithms among comparison-based sorting algorithms.

Divide: The array $A[low \dots high]$ is partitioned into two non-empty sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$, such that each element of $A[low \dots high]$ is less than or equal to each element of $A[q+1 \dots high]$. The index q is computed as part of this partitioning procedure.

Conquer: The two sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$ are sorted by recursive calls to Quick sort.

Algorithm

The recursive algorithm consists of four steps:

1) If there are one or no elements in the array to be sorted, return.
2) Pick an element in the array to serve as the "*pivot*" point. (Usually the left-most element in the array is used.)
3) Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array.

Implementation

```
import random
def QuickSort( A, low, high ):
    if low < high:
      pivot = Partition( A, low, high)
      QuickSort( A, low, pivot - 1 )
      QuickSort( A, pivot + 1, high)
def Partition( A, low, high ) :
    pivot = low
    swap( A, pivot, high)
    for i in range(low, high ):
      if A[i] <= A[high]:
        swap( A, i, low)
        low += 1
    swap( A, low, high)
    return low
def swap( A, x, y ):
  temp = A[x]
  A[x] = A[y]
  A[y] = temp
A = [534,246,933,127,277,321,454,565,220]
QuickSort(A, 0, len( A ) - 1)
print(A)
```

Analysis

Let us assume that T(n) be the complexity of Quick sort and also assume that all elements are distinct. Recurrence for $T(n)$ depends on two subproblem sizes which depend on partition element. If pivot is i^{th} smallest element then exactly $(i-1)$ items will be in left part and $(n-i)$ in right part. Let us call it as i –split. Since each element has equal probability of selecting it as pivot the probability of selecting i^{th} element is $\frac{1}{n}$.

Best Case: Each partition splits array in halves and gives

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n), \text{ [using } \textit{Divide and Conquer} \text{ master theorem]}$$

Worst Case: Each partition gives unbalanced splits and we get

$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$[*using Subtraction and Conquer master theorem*]

The worst-case occurs when the list is already sorted and last element chosen as pivot.

Average Case: In the average case of Quick sort, we do not know where the split happens. For this reason, we take all possible values of split locations, add all their complexities and divide with n to get the average case complexity.

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} \frac{1}{n}(\textit{runtime with } i-\textit{split}) + n + 1 \\
&= \frac{1}{n}\sum_{i=1}^{N}\big(T(i-1) + T(n-i)\big) + n + 1 \\
&\quad //\text{since we are dealing with best case we can assume } T(n-i) \text{ and } T(i-1) \text{ are equal} \\
&= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1 \\
&= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n + 1
\end{aligned}$$

Multiply both sides by n.

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for $n-1$.

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

Subtract the $n-1$ formula from n.

$$nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n - (2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1))$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n$$

$$nT(n) = (n+1)T(n-1) + 2n$$

Divide with $n(n+1)$.

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= O(1) + 2\textstyle\sum_{i=3}^{n} \frac{1}{i} \\
&= O(1) + O(2\log n) \\
\frac{T(n)}{n+1} &= O(\log n) \\
T(n) &= O\big((n+1)\log n\big) = O(n\log n)
\end{aligned}$$

Time Complexity, $T(n) = O(n\log n)$.

Performance

Worst case Complexity: $O(n^2)$
Best case Complexity: $O(nlogn)$
Average case Complexity: $O(nlogn)$
Worst case space Complexity: O(1)

Randomized Quick sort

In average-case behavior of Quick sort, we assume that all permutations of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in Quick sort.

There are two ways of adding randomization in Quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the Partition algorithm.

In normal Quick sort, *pivot* element was always the leftmost element in the list to be sorted. Instead of always using $A[low]$ as *pivot*, we will use a randomly chosen element from the subarray $A[low..high]$ in the randomized version of Quick sort. It is done by exchanging element $A[low]$ with an element chosen at random from $A[low..high]$. This ensures that the *pivot* element is equally likely to be any of the $high - low + 1$ elements in the subarray.

Since the pivot element is randomly chosen, we can expect the split of the input array to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which occurs in unbalanced partitioning.

Even though the randomized version improves the worst case complexity, its worst case complexity is still $O(n^2)$. One way to improve $Randomized - Quick\ sort$ is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

10.12 Tree Sort

Tree sort uses a binary search tree. It involves scanning each element of the input and placing it into its proper position in a binary search tree. This has two phases:

- First phase is creating a binary search tree using the given array elements.
- Second phase is traversing the given binary search tree in inorder, thus resulting in a sorted array.

Performance

The average number of comparisons for this method is $O(nlogn)$. But in worst case, the number of comparisons is reduced by $O(n^2)$, a case which arises when the sort tree is skew tree.

10.13 Comparison of Sorting Algorithms

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n+d)$, where d is the number of inversions.
Shell	-	$O(nlog^2n)$	1	no	
Merge sort	$O(nlogn)$	$O(nlogn)$	depends	yes	
Heap sort	$O(nlogn)$	$O(nlogn)$	1	no	
Quick sort	$O(nlogn)$	$O(n^2)$	$O(logn)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.
Tree sort	$O(nlogn)$	$O(n^2)$	$O(n)$	depends	Can be implemented as a stable sort.

Note: n denotes the number of elements in the input.

10.14 Linear Sorting Algorithms

In earlier sections, we have seen many examples of comparison-based sorting algorithms. Among them, the best comparison-based sorting has the complexity O($nlogn$). In this section, we will discuss other types of algorithms: Linear Sorting Algorithms. To improve the time complexity of sorting these algorithms, we make some assumptions about the input. A few examples of Linear Sorting Algorithms are:

- Counting Sort
- Bucket Sort
- Radix Sort

10.15 Counting Sort

Counting sort is not a comparison sort algorithm and gives O(n) complexity for sorting. To achieve O(n) complexity, *counting* sort assumes that each of the elements is an integer in the range 1 to K, for some integer K. When K = O(n), the *counting* sort runs in O(n) time. The basic idea of Counting sort is to determine, for each input element X, the number of elements less than X. This information can be used to place it directly into its correct position. For example, if 10 elements are less than X, then X belongs to position 11 in the output.

In the code below, $A[0..n-1]$ is the input array with length n. In Counting sort we need two more arrays: let us assume array $B[0..n-1]$ contains the sorted output and the array $C[0..K-1]$ provides temporary storage.

```
import random
def CountingSort(A, k):
    B = [0 for el in A]
    C = [0 for el in range(0, k+1)]
    for i in xrange(0, k +1):
            C[i] = 0
    for j in xrange(0, len(A)):
            C[A[j]] += 1
    for i in xrange(1, k + 1):
            C[i] += C[i - 1]
    for j in xrange(len(A)-1, 0 - 1, -1):
            tmp = A[j]
            tmp2= C[tmp] -1
            B[tmp2] = tmp
            C[tmp] -= 1
    return B
A = [534,246,933,127,277,321,454,565,220]
print(CountingSort(A, 1000))
```

Total Complexity: O(K) + O(n) + O(K) + O(n) = O(n) if K =O(n).
Space Complexity: O(n) if K =O(n).

Note: Counting works well if K =O(n). Otherwise, the complexity will be greater.

10.16 Bucket Sort (or Bin Sort)

Like *Counting* sort, *Bucket* sort also imposes restrictions on the input to improve the performance. In other words, Bucket sort works well if the input is drawn from fixed set. *Bucket* sort is the generalization of *Counting* Sort. For example, assume that all the input elements from {0, 1, . . . , $K-1$}, i.e., the set of integers in the interval [0, $K-1$]. That means, K is the number of distant elements in the input. *Bucket* sort uses K counters. The i^{th} counter keeps track of the number of occurrences of the i^{th} element. Bucket sort with two buckets is effectively a version of Quick sort with two buckets.

For bucket sort, the hash function that is used to partition the elements need to be very good and must produce ordered hash: if i < k then hash(i) < hash(k). Second, the elements to be sorted must be uniformly distributed.

The aforementioned aside, bucket sort is actually very good considering that counting sort is reasonably speaking its upper bound. And counting sort is very fast. The particular distinction for bucket sort is that it uses a hash function to partition the keys of the input array, so that multiple keys may hash to the same bucket. Hence each bucket must effectively be a growable list; similar to radix sort.

In the below code Insertionsort is used to sort each bucket. This is to inculcate that the bucket sort algorithm does not specify which sorting technique to use on the buckets. A programmer may choose to continuously use bucket sort on each bucket until the collection is sorted (in the manner of the radix sort program below). Whichever sorting method is used on the , bucket sort still tends toward O(n).

```
def insertionsort( A ):
    for i in range( 1, len( A ) ):
      temp = A[i]
      k = i
      while k > 0 and temp < A[k - 1]:
         A[k] = A[k - 1]
         k -= 1
      A[k] = temp

def BucketSort( A ):
    code = Hashing( A )
    buckets = [list() for _ in range( code[1] )]
    for i in A:
      x = ReHashing( i, code )
      buck = buckets[x]
      buck.append( i )

    for bucket in buckets:
      insertionSort( bucket )

    ndx = 0
    for b in range( len( buckets ) ):
      for v in buckets[b]:
        A[ndx] = v
        ndx += 1

    return A

import math

def Hashing( A ):
    m = A[0]
    for i in range( 1, len( A ) ):
      if ( m < A[i] ):
        m = A[i]
    result = [m, int( math.sqrt( len( A ) ) )]
    return result

def ReHashing( i, code ):
    return int( i / code[0] * ( code[1] - 1 ) )

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

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

10.17 Radix Sort

Similar to *Counting* sort and *Bucket* sort, this sorting algorithm also assumes some kind of information about the input elements. Suppose that the input values to be sorted are from base d. That means all numbers are d-digit numbers.

In Radix sort, first sort the elements based on the last digit [the least significant digit]. These results are again sorted by second digit [the next to least significant digit]. Continue this process for all digits until we reach the most significant digits. Use some stable sort to sort them by last digit. Then stable sort them by the second least significant digit, then by the third, etc. If we use Counting sort as the stable sort, the total time is O(nd) ≈O(n).

Algorithm:

1) Take the least significant digit of each element.
2) Sort the list of elements based on that digit, but keep the order of elements with the same digit (this is the definition of a stable sort).
3) Repeat the sort with each more significant digit.

The speed of Radix sort depends on the inner basic operations. If the operations are not efficient enough, Radix sort can be slower than other algorithms such as Quick sort and Merge sort. These operations include the insert and delete functions of the sub-lists and the process of isolating the digit we want. If the numbers are not of equal length then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix sort and also one of the hardest to make efficient.

Since Radix sort depends on the digits or letters, it is less flexible than other sorts. For every different type of data, Radix sort needs to be rewritten, and if the sorting order changes, the sort needs to be rewritten again. In short, Radix sort takes more time to write, and it is very difficult to write a general purpose Radix sort that can handle all kinds of data.

For many programs that need a fast sort, Radix sort is a good choice. Still, there are faster sorts, which is one reason why Radix sort is not used as much as some other sorts.

```
def RadixSort( A ):
    RADIX = 10
    maxLength = False
    tmp , placement = -1, 1
    while not maxLength:
        maxLength = True
        buckets = [list() for _ in range( RADIX )]
        for  i in A:
            tmp = i / placement
            buckets[tmp % RADIX].append( i )
            if maxLength and tmp > 0:
                maxLength = False

        a = 0
        for b in range( RADIX ):
            buck = buckets[b]
            for i in buck:
                A[a] = i
                a += 1

        # move to next digit
        placement *= RADIX
A = [534,246,933,127,277,321,454,565,220]
print(RadixSort(A))
```

Time Complexity: $O(nd) \approx O(n)$, if d is small.

10.18 Topological Sort

Refer to *Graph Algorithms* Chapter.

10.19 External Sorting

External sorting is a generic term for a class of sorting algorithms that can handle massive amounts of data. These external sorting algorithms are useful when the files are too big and cannot fit into main memory.

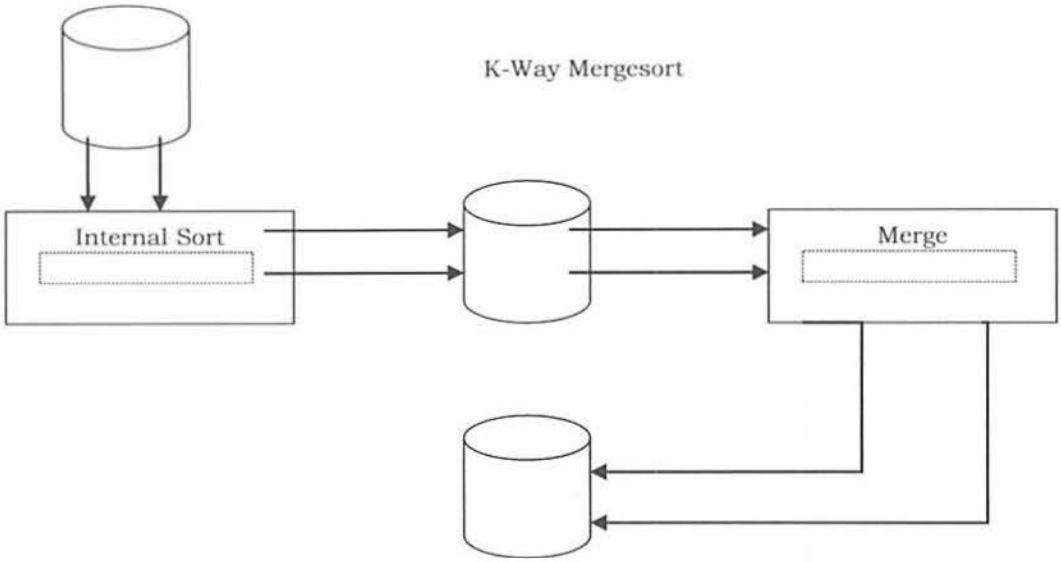
As with internal sorting algorithms, there are a number of algorithms for external sorting. One such algorithm is External Mergesort. In practice, these external sorting algorithms are being supplemented by internal sorts.

Simple External Mergesort

A number of records from each tape are read into main memory, sorted using an internal sort, and then output to the tape. For the sake of clarity, let us assume that 900 megabytes of data needs to be sorted using only 100 megabytes of RAM.

1) Read 100MB of the data into main memory and sort by some conventional method (let us say Quick sort).
2) Write the sorted data to disk.
3) Repeat steps 1 and 2 until all of the data is sorted in chunks of 100MB. Now we need to merge them into one single sorted output file.
4) Read the first 10MB of each sorted chunk (call them input buffers) in main memory (90MB total) and allocate the remaining 10MB for output buffer.

Perform a 9-way Mergesort and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10MB of its associated 100MB sorted chunk; or if there is no more data in the sorted chunk, mark it as exhausted and do not use it for merging.



The above algorithm can be generalized by assuming that the amount of data to be sorted exceeds the available memory by a factor of K. Then, K chunks of data need to be sorted and a K -way merge has to be completed.

If X is the amount of main memory available, there will be K input buffers and 1 output buffer of size $X/(K+1)$ each. Depending on various factors (how fast is the hard drive?) better performance can be achieved if the output buffer is made larger (for example, twice as large as one input buffer).

Complexity of the 2-way External Merge sort: In each pass we read + write each page in file. Let us assume that there are n pages in file. That means we need $\lceil logn \rceil + 1$ number of passes. The total cost is $2n(\lceil logn \rceil + 1)$.

10.20 Sorting: Problems & Solutions

Problem-1 Given an array $A[0 \ldots n-1]$ of n numbers containing the repetition of some number. Give an algorithm for checking whether there are repeated elements or not. Assume that we are not allowed to use additional space (i.e., we can use a few temporary variables, O(1) storage).

Solution: Since we are not allowed to use extra space, one simple way is to scan the elements one-by-one and for each element check whether that element appears in the remaining elements. If we find a match we return true.

```
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)
```

Each iteration of the inner, j-indexed loop uses O(1) space, and for a fixed value of i, the j loop executes $n-i$ times. The outer loop executes $n-1$ times, so the entire function uses time proportional to

$$\sum_{i=1}^{n-1} n-i = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-2 Can we improve the time complexity of Problem-1?

Solution: **Yes**, using sorting technique.

```
def CheckDuplicatesSorting(A):
    A.sort()
    for i in range(0,len(A)-1):
        for j in range(i+1,len(A)):
            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)
```

Heapsort function takes O($nlogn$) time, and requires O(1) space. The scan clearly takes $n - 1$ iterations, each iteration using O(1) time. The overall time is O($nlogn + n$) = O($nlogn$).

Time Complexity: O($nlogn$). Space Complexity: O(1).

Note: For variations of this problem, refer *Searching* chapter.

Problem-3 Given an array $A[0 \ldots n-1]$, where each element of the array represents a vote in the election. Assume that each vote is given as an integer representing the ID of the chosen candidate. Give an algorithm for determining who wins the election.

Solution: This problem is nothing but finding the element which repeated the maximum number of times. The solution is similar to the Problem-1 solution: keep track of counter.

```
def CheckWhoWinsTheElection(A):
    counter = maxCounter = 0
    candidate = A[0]
    for i in range(0,len(A)):
        counter = 1
        for j in range(i+1,len(A)):
            if(A[i]==A[j]):
                counter += 1
        if(counter > maxCounter):
            maxCounter = counter
            candidate = A[i]
    print candidate, "appeared ", maxCounter, " times"
A = [3,2,1,2,2,3]
CheckWhoWinsTheElection(A)
A = [3,3,3,2,2,3]
CheckWhoWinsTheElection(A)
```

Time Complexity: O(n^2). Space Complexity: O(1).

Note: For variations of this problem, refer to *Searching* chapter.

Problem-4 Can we improve the time complexity of Problem-3? Assume we don't have any extra space.

Solution: Yes. The approach is to sort the votes based on candidate ID, then scan the sorted array and count up which candidate so far has the most votes. We only have to remember the winner, so we don't need a clever data structure. We can use Heapsort as it is an in-place sorting algorithm.

```
def CheckWhoWinsTheElection(A):
    A.sort()
    counter = maxCounter = 0
    candidate = maxCandidate = 0
    for i in range(0,len(A)):
        if( A[i] == candidate):
            counter += 1
        else:
            counter = 1
            candidate = A[i]
        if(counter > maxCounter):
            maxCandidate = A[i]
            maxCounter = counter
```

```
        print maxCandidate, "appeared ", maxCounter, " times"
A = [2,3,2,1,2,2,3,2,2]
CheckWhoWinsTheElection(A)
A = [3,3,3,2,2,3]
CheckWhoWinsTheElection(A)
```

Since Heapsort time complexity is $O(nlogn)$ and in-place, it only uses an additional $O(1)$ of storage in addition to the input array. The scan of the sorted array does a constant-time conditional $n - 1$ times, thus using $O(n)$ time. The overall time bound is $O(nlogn)$.

Problem-5 Can we further improve the time complexity of Problem-3?

Solution: In the given problem, the number of candidates is less but the number of votes is significantly large. For this problem we can use counting sort.

Time Complexity: $O(n)$, n is the number of votes (elements) in the array.
Space Complexity: $O(k)$, k is the number of candidates participating in the election.

Problem-6 Given an array A of n elements, each of which is an integer in the range $[1, n^2]$, how do we sort the array in $O(n)$ time?

Solution: If we subtract each number by 1 then we get the range $[0, n^2 - 1]$. If we consider all numbers as 2 –digit base n. Each digit ranges from 0 to n^2 - 1. Sort this using radix sort. This uses only two calls to counting sort. Finally, add 1 to all the numbers. Since there are 2 calls, the complexity is $O(2n) \approx O(n)$.

Problem-7 For Problem-6, what if the range is $[1...n^3]$?

Solution: If we subtract each number by 1 then we get the range $[0, n^3 - 1]$. Considering all numbers as 3-digit base n: each digit ranges from 0 to n^3 - 1. Sort this using radix sort. This uses only three calls to counting sort. Finally, add 1 to all the numbers. Since there are 3 calls, the complexity is $O(3n) \approx O(n)$.

Problem-8 Given an array with n integers, each of value less than n^{100}, can it be sorted in linear time?

Solution: Yes. The reasoning is same as in of Problem-6 and Problem-7.

Problem-9 Let A and B be two arrays of n elements each. Given a number K, give an $O(nlogn)$ time algorithm for determining whether there exists a $\in$ A and b $\in$ B such that $a + b = K$.

Solution: Since we need $O(nlogn)$, it gives us a pointer that we need to sort. So, we will do that.

```
def binarySearch(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
def findSumInLists(A, B, k):
        A.sort()
        for i in range(0,len(B)):
                c = k-B[i]
                if(BinarySearch(A, c) != -1):
                        return 1
        return 0
A = [2,3,5,7,12,15,23,32,42]
B = [3,13,13,15,22,33]
print findSumInLists(A, B, 270)
```

Note: For variations of this problem, refer to *Searching* chapter.

Problem-10 Let A, B and C be three arrays of n elements each. Given a number K, give an $O(nlogn)$ time algorithm for determining whether there exists $a \in A$, $b \in B$ and $c \in C$ such that $a + b + c = K$.

Solution: Refer to *Searching* chapter.

Problem-11 Given an array of n elements, can we output in sorted order the K elements following the median in sorted order in time $O(n + KlogK)$.

Solution: Yes. Find the median and partition the median. With this we can find all the elements greater than it. Now find the K^{th} largest element in this set and partition it; and get all the elements less than it. Output the sorted list of the final set of elements. Clearly, this operation takes $O(n + KlogK)$ time.

Problem-12 Consider the sorting algorithms: Bubble sort, Insertion sort, Selection sort, Merge sort, Heap sort, and Quick sort. Which of these are stable?

Solution: Let us assume that A is the array to be sorted. Also, let us say R and S have the same key and R appears earlier in the array than S. That means, R is at $A[i]$ and S is at $A[j]$, with $i < j$. To show any stable algorithm, in the sorted output R must precede S.

Bubble sort: Yes. Elements change order only when a smaller record follows a larger. Since S is not smaller than R it cannot precede it.

Selection sort: No. It divides the array into sorted and unsorted portions and iteratively finds the minimum values in the unsorted portion. After finding a minimum x, if the algorithm moves x into the sorted portion of the array by means of a swap, then the element swapped could be R which then could be moved behind S. This would invert the positions of R and S, so in general it is not stable. If swapping is avoided, it could be made stable but the cost in time would probably be very significant.

Insertion sort: Yes. As presented, when S is to be inserted into sorted subarray $A[1..j - 1]$, only records larger than S are shifted. Thus R would not be shifted during S's insertion and hence would always precede it.

Merge sort: Yes, In the case of records with equal keys, the record in the left subarray gets preference. Those are the records that came first in the unsorted array. As a result, they will precede later records with the same key.

Heap sort: No. Suppose $i = 1$ and R and S happen to be the two records with the largest keys in the input. Then R will remain in location 1 after the array is heapified, and will be placed in location n in the first iteration of Heapsort. Thus S will precede R in the output.

Quick sort: No. The partitioning step can swap the location of records many times, and thus two records with equal keys could swap position in the final output.

Problem-13 Consider the same sorting algorithms as that of Problem-12. Which of them are in-place?

Solution:

Bubble sort: Yes, because only two integers are required.

Insertion sort: Yes, since we need to store two integers and a record.

Selection sort: Yes. This algorithm would likely need space for two integers and one record.

Merge sort: No. Arrays need to perform the merge. (If the data is in the form of a linked list, the sorting can be done in-place, but this is a nontrivial modification.)

Heap sort: Yes, since the heap and partially-sorted array occupy opposite ends of the input array.

Quicksort: No, since it is recursive and stores $O(logn)$ activation records on the stack. Modifying it to be non-recursive is feasible but nontrivial.

Problem-14 Among Quick sort, Insertion sort, Selection sort, and Heap sort algorithms, which one needs the minimum number of swaps?

Solution: Selection sort – it needs n swaps only (refer to theory section).

Problem-15 What is the minimum number of comparisons required to determine if an integer appears more than $n/2$ times in a sorted array of n integers?

Solution: Refer to *Searching* chapter.

Problem-16 **Sort an array of 0's, 1's and 2's:** Given an array A[] consisting of 0's, 1's and 2's, give an algorithm for sorting A[]. The algorithm should put all 0's first, then all 1's and all 2's last.
Example: Input = {0,1,1,0,1,2,1,2,0,0,0,1}, Output = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}

Solution: Use Counting sort. Since there are only three elements and the maximum value is 2, we need a temporary array with 3 elements.

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Note: For variations of this problem, refer to *Searching* chapter.

Problem-17 Is there any other way of solving Problem-16?

Solution: Using Quick dort. Since we know that there are only 3 elements, 0, 1 and 2 in the array, we can select 1 as a pivot element for Quick sort. Quick sort finds the correct place for 1 by moving all 0's to the left of 1 and all 2's to the right of 1. For doing this it uses only one scan.

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Note: For efficient algorithm, refer to *Searching* chapter.

Problem-18 How do we find the number that appeared the maximum number of times in an array?

Solution: One simple approach is to sort the given array and scan the sorted array. While scanning, keep track of the elements that occur the maximum number of times.

Time Complexity = Time for Sorting + Time for Scan = $O(nlogn) + O(n) = O(nlogn)$. Space Complexity: $O(1)$.

Note: For variations of this problem, refer to *Searching* chapter.

Problem-19 Is there any other way of solving Problem-18?

Solution: Using Binary Tree. Create a binary tree with an extra field *count* which indicates the number of times an element appeared in the input. Let us say we have created a Binary Search Tree [BST]. Now, do the In-Order traversal of the tree. The In-Order traversal of BST produces the sorted list. While doing the In-Order traversal keep track of the maximum element.

Time Complexity: $O(n) + O(n) \approx O(n)$. The first parameter is for constructing the BST and the second parameter is for Inorder Traversal. Space Complexity: $O(2n) \approx O(n)$, since every node in BST needs two extra pointers.

Problem-20 Is there yet another way of solving Problem-18?

Solution: Using Hash Table. For each element of the given array we use a counter, and for each occurrence of the element we increment the corresponding counter. At the end we can just return the element which has the maximum counter.

Time Complexity: $O(n)$. Space Complexity: $O(n)$. For constructing the hash table we need $O(n)$.

Note: For the efficient algorithm, refer to the *Searching* chapter.

Problem-21 Given a 2 GB file with one string per line, which sorting algorithm would we use to sort the file and why?

Solution: When we have a size limit of 2GB, it means that we cannot bring all the data into the main memory.

Algorithm: How much memory do we have available? Let's assume we have X MB of memory available. Divide the file into K chunks, where $X * K \sim 2\,GB$.

- Bring each chunk into memory and sort the lines as usual (any $O(nlogn)$ algorithm).
- Save the lines back to the file.
- Now bring the next chunk into memory and sort.
- Once we're done, merge them one by one; in the case of one set finishing, bring more data from the particular chunk.

The above algorithm is also known as *external sort*. Step 3 – 4 is known as K-way merge. The idea behind going for an external sort is the size of data. Since the data is huge and we can't bring it to the memory, we need to go for a disk-based sorting algorithm.

Problem-22 **Nearly sorted:** Given an array of n elements, each which is at most K positions from its target position, devise an algorithm that sorts in $O(n\ logK)$ time.

Solution: Divide the elements into n/K groups of size K, and sort each piece in $O(KlogK)$ time, let's say using Mergesort. This preserves the property that no element is more than K elements out of position. Now, merge each block of K elements with the block to its left.

Problem-23 Is there any other way of solving Problem-22?

Solution: Insert the first K elements into a binary heap. Insert the next element from the array into the heap, and delete the minimum element from the heap. Repeat.

Problem-24 **Merging K sorted lists:** Given K sorted lists with a total of n elements, give an $O(nlogK)$ algorithm to produce a sorted list of all n elements.

Solution: Simple Algorithm for merging K sorted lists: Consider groups each having $\frac{n}{K}$ elements. Take the first list and merge it with the second list using a linear-time algorithm for merging two sorted lists, such as the merging algorithm used in merge sort. Then, merge the resulting list of $\frac{2n}{K}$ elements with the third list, and then

merge the resulting list of $\frac{3n}{K}$ elements with the fourth list. Repeat this until we end up with a single sorted list of all n elements.

Time Complexity: In each iteration we are merging K elements.

$$T(n) = \frac{2n}{K} + \frac{3n}{K} + \frac{4n}{K} + \cdots \frac{Kn}{K}(n) = \frac{n}{K}\sum_{i=2}^{K} i$$

$$T(n) = \frac{n}{K}\left[\frac{K(K+1)}{2}\right] \approx O(nK)$$

Problem-25 Can we improve the time complexity of Problem-24?

Solution: One method is to repeatedly pair up the lists and then merge each pair. This method can also be seen as a tail component of the execution merge sort, where the analysis is clear. This is called the Tournament Method. The maximum depth of the Tournament Method is $logK$ and in each iteration we are scanning all the n elements.

Time Complexity: O($n$logK).

Problem-26 Is there any other way of solving Problem-24?

Solution: The other method is to use a *min* priority queue for the minimum elements of each of the K lists. At each step, we output the extracted minimum of the priority queue, determine from which of the K lists it came, and insert the next element from that list into the priority queue. Since we are using priority queue, that maximum depth of priority queue is $logK$.

Time Complexity: O($n$logK).

Problem-27 Which sorting method is better for Linked Lists?

Solution: Merge Sort is a better choice. At first appearance, merge sort may not be a good selection since the middle node is required to subdivide the given list into two sub-lists of equal length. We can easily solve this problem by moving the nodes alternatively to two lists (refer to *Linked Lists* chapter). Then, sorting these two lists recursively and merging the results into a single list will sort the given one.

```
# Definition for singly-linked list.
class ListNode:
        def __init__(self, x):
                self.data = x
                self.next = None
class LinkedListSortWithMergeSort:
  def sortList(self, head):
    if head == None:
      return None
    counter = 0
    temp = head
    while temp != None:
      temp = temp.next
      counter += 1
    return self.sort(head, counter)
  def sort(self,head,size):
    if size ==1:
      return head
    list2 = head
    for i in range(0,size//2):
      list2 = list2.next
    list1 = self.sort(head, size//2)
    list2 = self.sort(list2,size-size//2)
    return self.merge(list1, size//2, list2, size-size//2)
  def merge(self,list1, sizeList1, list2, sizeList2):
    dummy = ListNode(0)
    list = dummy
    pointer1 = 0
    pointer2 = 0
    while pointer1 < sizeList1 and pointer2 < sizeList2:
      if list1.data<list2.data:
        list.next = list1
```

```
            list1 = list1.next
            pointer1 += 1
        else:
            list.next = list2
            list2 = list2.next
            pointer2 += 1
        list = list.next
    while pointer1 < sizeList1:
        list.next = list1
        list1 = list1.next
        pointer1 += 1
        list = list.next
    while pointer2 < sizeList2:
        list.next = list2
        list2 = list2.next
        pointer2 += 1
        list = list.next
    list.next = None
    return dummy.next
```

Note: Append() appends the first argument to the tail of a singly linked list whose head and tail are defined by the second and third arguments.

All external sorting algorithms can be used for sorting linked lists since each involved file can be considered as a linked list that can only be accessed sequentially. We can sort a doubly linked list using its next fields as if it was a singly linked one and reconstruct the prev fields after sorting with an additional scan.

Problem-28 Can we implement Linked Lists Sorting with Quick Sort?

Solution: The original Quick Sort cannot be used for sorting Singly Linked Lists. This is because we cannot move backward in Singly Linked Lists. But we can modify the original Quick Sort and make it work for Singly Linked Lists.

Let us consider the following modified Quick Sort implementation. The first node of the input list is considered a *pivot* and is moved to *equal*. The value of each node is compared with the *pivot* and moved *to less* (respectively, *equal* or *larger*) if the nodes value is smaller than (respectively, *equal* to or *larger* than) the *pivot*. Then, *less* and *larger* are sorted recursively. Finally, joining *less*, *equal* and *larger* into a single list yields a sorted one.

Append() appends the first argument to the tail of a singly linked list whose head and tail are defined by the second and third arguments. On return, the first argument will be modified so that it points to the next node of the list. *Join*() appends the list whose head and tail are defined by the third and fourth arguments to the list whose head and tail are defined by the first and second arguments. For simplicity, the first and fourth arguments become the head and tail of the resulting list.

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.data = x
        self.next = None

def Qsort(first, last):
    lesHEAD = lesTAIL=None
    equHEAD = equTAIL=None
    larHEAD = larTAIL=None
    current = first
    if(current == None):
        return
    pivot = current.data
    Append(current, equHEAD, equTAIL)
    while (current != None):
        info = current.data
        if(info < pivot):
            Append(current, lesHEAD, lesTAIL)
        elif(info > pivot):
            Append(current, larHEAD, larTAIL)
        else:
            Append(current, equHEAD, equTAIL)
```

```
Quicksort(lesHEAD, lesTAIL)
Quicksort(larHEAD, larTAIL)
Join(lesHEAD, lesTAIL,equHEAD, equTAIL)
Join(lesHEAD, equTAIL,larHEAD, larTAIL)
first = lesHEAD
last = larTAIL
```

Problem-29 Given an array of 100,000 pixel color values, each of which is an integer in the range [0,255]. Which sorting algorithm is preferable for sorting them?

Solution: Counting Sort. There are only 256 key values, so the auxiliary array would only be of size 256, and there would be only two passes through the data, which would be very efficient in both time and space.

Problem-30 Similar to Problem-29, if we have a telephone directory with 10 million entries, which sorting algorithm is best?

Solution: Bucket Sort. In Bucket Sort the buckets are defined by the last 7 digits. This requires an auxiliary array of size 10 million and has the advantage of requiring only one pass through the data on disk. Each bucket contains all telephone numbers with the same last 7 digits but with different area codes. The buckets can then be sorted by area code with selection or insertion sort; there are only a handful of area codes.

Problem-31 Give an algorithm for merging K-sorted lists.

Solution: Refer to *Priority Queues* chapter.

Problem-32 Given a big file containing billions of numbers. Find maximum 10 numbers from this file.

Solution: Refer to *Priority Queues* chapter.

Problem-33 There are two sorted arrays A and B. The first one is of size $m+n$ containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size $m+n$ such that the output is sorted.

Solution: The trick for this problem is to start filling the destination array from the back with the largest elements. We will end up with a merged and sorted destination array.

```
def Merge(A, m, B, n):
    i = n - 1
    j = k = m - 1
    while k>=0:
        if(B[i] > A[j] or j < 0):
            A[k] =B[i]
            i -= 1
            if(i<0):
                break
        else:
            A[k] = A[j]
            j -= 1
        k -= 1
```

Time Complexity: $O(m+n)$. Space Complexity: $O(1)$.

Problem-34 **Nuts and Bolts Problem:** Given a set of n nuts of different sizes and n bolts such that there is a one-to-one correspondence between the nuts and the bolts, find for each nut its corresponding bolt. Assume that we can only compare nuts to bolts: we cannot compare nuts to nuts and bolts to bolts.

Alternative way of framing the question: We are given a box which contains bolts and nuts. Assume there are n nuts and n bolts and that each nut matches exactly one bolt (and vice versa). By trying to match a bolt and a nut we can see which one is bigger, but we cannot compare two bolts or two nuts directly. Design an efficient algorithm for matching the nuts and bolts.

Solution: Brute Force Approach: Start with the first bolt and compare it with each nut until we find a match. In the worst case, we require n comparisons. Repeat this for successive bolts on all remaining gives $O(n^2)$ complexity.

Problem-35 For Problem-34, can we improve the complexity?

Solution: In Problem-34, we got $O(n^2)$ complexity in the worst case (if bolts are in ascending order and nuts are in descending order). Its analysis is the same as that of Quick Sort. The improvement is also along the same lines.

To reduce the worst case complexity, instead of selecting the first bolt every time, we can select a random bolt and match it with nuts. This randomized selection reduces the probability of getting the worst case, but still the worst case is $O(n^2)$.

Problem-36 For Problem-34, can we further improve the complexity?

Solution: We can use a divide-and-conquer technique for solving this problem and the solution is very similar to randomized Quick Sort. For simplicity let us assume that bolts and nuts are represented in two arrays B and N.

The algorithm first performs a partition operation as follows: pick a random bolt $B[i]$. Using this bolt, rearrange the array of nuts into three groups of elements:

- First the nuts smaller than $B[i]$
- Then the nut that matches $B[i]$, and
- Finally, the nuts larger than $B[i]$.

Next, using the nut that matches $B[i]$, perform a similar partition on the array of bolts. This pair of partitioning operations can easily be implemented in $O(n)$ time, and it leaves the bolts and nuts nicely partitioned so that the "*pivot*" bolt and nut are aligned with each other and all other bolts and nuts are on the correct side of these pivots – smaller nuts and bolts precede the pivots, and larger nuts and bolts follow the pivots. Our algorithm then completes by recursively applying itself to the subarray to the left and right of the pivot position to match these remaining bolts and nuts. We can assume by induction on n that these recursive calls will properly match the remaining bolts.

To analyze the running time of our algorithm, we can use the same analysis as that of randomized Quick Sort. Therefore, applying the analysis from Quick Sort, the time complexity of our algorithm is $O(nlogn)$.

Alternative Analysis: We can solve this problem by making a small change to Quick Sort. Let us assume that we pick the last element as the pivot, say it is a nut. Compare the nut with only bolts as we walk down the array. This will partition the array for the bolts. Every bolt less than the partition nut will be on the left. And every bolt greater than the partition nut will be on the right.

While traversing down the list, find the matching bolt for the partition nut. Now we do the partition again using the matching bolt. As a result, all the nuts less than the matching bolt will be on the left side and all the nuts greater than the matching bolt will be on the right side. Recursively call on the left and right arrays.

The time complexity is $O(2nlogn) \approx O(nlogn)$.

Problem-37 Given a binary tree, can we print its elements in sorted order in $O(n)$ time by performing an In-order tree traversal?

Solution: Yes, if the tree is a Binary Search Tree [BST]. For more details refer to *Trees* chapter.

Problem-38 An algorithm for finding a specific value in a row and column sorted matrix of values. The algorithm takes as input a matrix of values where each row and each column are in sorted order, along with a value to locate in that array, then returns whether that element exists in the matrix. For example, given the matrix along with the number 7, the algorithm would output *yes*, but if given the number 0 the algorithm would output *no*.

1	2	2	2	3	4
1	2	3	3	4	5
3	4	4	4	4	6
4	5	6	7	8	9

Solution: One approach for solving this problem would be a simple exhaustive search of the matrix to find the value. If the matrix dimensions are mn, this algorithm will take time $O(mn)$ in the worst-case, which is indeed linear in the size of the matrix but takes no advantage of the sorted structure we are guaranteed to have in the matrix. Our goal will be to find a much faster algorithm for solving the same problem.

One approach that might be useful for solving the problem is to try to keep deleting rows or columns out of the array in a way that reduces the problem size without ever deleting the value (should it exist). For example, suppose that we iteratively start deleting rows and columns from the matrix that we know do not contain the value. We can repeat this until either we've reduced the matrix down to nothingness, in which case we know that the element is not present, or until we find the value. If the matrix is mn, then this would require only $O(m + n)$ steps, which is much faster than the O(mn) approach outlined above.

In order to realize this as a concrete algorithm, we'll need to find a way to determine which rows or columns to drop. One particularly elegant way to do this is to look at the very last element of the first row of the matrix. Consider how it might relate to the value we're looking for. If it's equal to the value in question, we're done and can just hand back that we've found the entry we want. If it's greater than the value in question, since each column is in sorted order, we know that no element of the last column could possibly be equal to the number we

want to search for, and so we can discard the last column of the matrix. Finally, if it's less than the value in question, then we know that since each row is in sorted order, none of the values in the first row can equal the element in question, since they're no bigger than the last element of that row, which is in turn smaller than the element in question. This gives a very straightforward algorithm for finding the element - we keep looking at the last element of the first row, then decide whether to discard the last row or the last column. As mentioned above, this will run in $O(m + n)$ time.

```
def matrixFind(matrix, value):
    m = len(matrix)
    if m == 0:
        return 0
    n = len(matrix[0])
    if n == 0:
        return 0
    i = 0
    j = n - 1
    while i < m and j >= 0:
        if matrix[i][j] == value:
            return 1
        elif matrix[i][j] < value:
            i = i + 1
        else:
            j = j - 1
    return 0
```

Problem-38 Sort elements of list by frequency.

Soution: Sorting lists in Python is very simple (list.sort()), but we often need to sort a list of objects based on the one of the objects' attributes. Say we have a list of objects, each of which has an attribute called 'score'. We can sort the list by object score like so:

```
myList.sort(key = lambda x: x.score)
```

This passes a lambda function to sort, which tells it to compare the score attributes of the objects. Otherwise, the sort function works exactly as normal (so will, for example, order strings alphabetically. We can also use this technique to sort a dictionary by its values:

```
sortedKeys = sorted(myDict.keys(), key=lambda x: myDict[x])
for k in sortedKeys:
    print myDict[k]
```

The code creates a list of the dictionary keys, which it sorts based on the value for each key (note that we can't simply sort myDict.keys()). Alternatively we can loop through the keys and values in one go:

```
for k, v in sorted(myDict.items(), key=lambda (k,v): v):
    print k, v
```

Example:

```
myString = "We want to get the counts for each letter in this sentence"
counts = {}
for letter in myString:
    counts[letter] = counts.get(letter, 0) + 1
print counts
sortedKeys = sorted(counts.keys(), key=lambda x: counts[x])
for k in sortedKeys:
    print k , "-->" , counts[k]
```

CHAPTER

11

SEARCHING

11.1 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.

11.2 Why do we need Searching?

Searching is one of the core computer science algorithms. We know that today's computers store a lot of information. 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.

11.3 Types of Searching

Following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

11.4 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))
```

Time complexity: O(n), in the worst case we need to scan the complete array. Space complexity: O(1).

11.5 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))
```

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.

11.6 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.

$$mid = low + \frac{(high - low)}{2} \text{ or } \frac{low + high}{2}$$

```
//Iterative Binary Search Algorithm
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 = [534,246,933,127,277,321,454,565,220]
print(BinarySearchIterative(A,277))

//Recursive Binary Search Algorithm
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 = [534,246,933,127,277,321,454,565,220]
print(BinarySearchRecursive(A,277))
```

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. 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, $T(n) = O(logn)$.

Time Complexity: $O(logn)$. Space Complexity: O(1) [for iterative algorithm].

11.7 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.

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.

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

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.

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. For this algorithm to give best results, the dataset should be ordered and uniformly distributed.

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

    if numbersList[low] == value:
        return low
    return None
```

11.8 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	$log(logn)$

Note: For discussion on binary search trees refer *Trees* chapter.

11.9 Symbol Tables and Hashing

Refer to *Symbol Tables* and *Hashing* chapters.

11.10 String Searching Algorithms

Refer to *String Algorithms* chapter.

11.11 Searching: Problems & Solutions

Problem-1 Given an array of n numbers, give an algorithm for checking whether there are any duplicate elements in the array or no?

Solution: 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:

```
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).

Problem-2 Can we improve the complexity of Problem-1's solution?

Solution: 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.

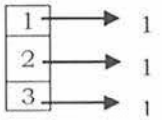
```
def CheckDuplicatesSorting(A):
    A.sort()
    for i in range(0,len(A)-1):
            for j in range(i+1,len(A)):
                    if(A[i] == A[i+1]):
                            print("Duplicates exist:", A[i])
                            return;
    print("No duplicates in given array.")

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

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

Problem-3 Is there any alternative way of solving *Problem-1*?

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):



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: $O(n)$.

Problem-4 Can we further improve the complexity of Problem-1 solution?

Solution: Let us assume that the array elements are positive numbers and 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, $A = \{3,2,1,2,2,3\}$.

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

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

3	2	1	-2	2	3
0	1	2	3	4	5

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

3	2	-1	-2	2	3
0	1	2	3	4	5

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

3	-2	- 1	-2	2	3
0	1	2	3	4	5

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

3	-2	- 1	-2	2	3
0	1	2	3	4	5

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)
```

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.

Problem-5 Given an array of n numbers. Give an algorithm for finding the element which appears the maximum number of times in the array?

Brute Force Solution: One simple solution to this is, for each input element check whether there is any element with the same value, and for each such occurrence, increment the counter. Each time, check the

current counter with the *max* counter and update it if this value is greater than *max* counter. This we can solve just by using two simple *for* loops.

```
def MaxRepititionsBruteForce(A):
    n = len(A)
    count = max = 0
    for i in range(0,n):
            count  = 1
            for j in range(0,n):
                    if( i != j and A[i] == A[j]):
                            count += 1
            if max< count:
                    max = count
                    maxRepeatedElement = A[i]
    print maxRepeatedElement, "repeated for ", max
A = [3,2,1,2,2,3,2,1,3]
MaxRepititionsBruteForce(A)
```

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

Problem-6 Can we improve the complexity of Problem-5 solution?

Solution: Yes. Sort the given array. After sorting, all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see which element is appearing the maximum number of times.

```
def MaxRepititionsWithSort(A):
    A.sort()
    print A
    j = 0
    count = max = 1
    element = A[0]
    for i in range(1,len(A)):
            if (A[i] == element):
                    count += 1
                    if count > max:
                            max =  count
                            maxRepeatedElement = element
            else:
                    count = 1
                    element = A[i]
    print maxRepeatedElement, "repeated for ", max
A = [3,2,1,3,2,3,2,3,3]
MaxRepititionsWithSort(A)
```

Time Complexity: O(*nlogn*). (for sorting). Space Complexity: O(1).

Problem-7 Is there any other way of solving Problem-5?

Solution: Yes, using hash table. For each element of the input, keep track of how many times that element appeared in the input. That means the counter value represents the number of occurrences for that element.

```
def MaxRepititionsWithHash(A):
    table = {}          # hash
    max = 0
    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 table[element] > max:
                    max = table[element]
                    maxRepeatedElement = element
    print maxRepeatedElement, "repeated for ", max, " times"
A = [3,2,1,3,2,3,2,3,3]
```

```
MaxRepititionsWithHash(A)
```

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

Problem-8 For Problem-5, can we improve the time complexity? Assume that the elements' range is 0 to $n-1$. That means all the elements are within this range only.

Solution: Yes. We can solve this problem in two scans. We *cannot* use the negation technique of Problem-3 for this problem because of the number of repetitions. In the first scan, instead of negating, add the value n. That means for each occurrence of an element add the array size to that element. In the second scan, check the element value by dividing it by n and return the element which gives the maximum value. The code based on this method is given below.

```
def MaxRepititionsEfficient(A):
    n = len(A)
    max = 0
    for i in range(0,len(A)):
            A[A[i]%n] += n
    for i in range(0,len(A)):
            if(A[i]/n > max):
                    max = A[i]/n
                    maxIndex =i
    print maxIndex, "repeated for ", max, " times"
A = [3,2,2,3,2,2,2,3,3]
MaxRepititionsEfficient(A)
```

Notes:

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

Time Complexity: O(n). Since no nested *for* loops are required. Space Complexity: O(1).

Problem-9 Given an array of n numbers, give an algorithm for finding the first element in the array which is repeated. For example, in the array $A = \{3, 2, 1, 2, 2, 3\}$, the first repeated number is 3 (not 2). That means, we need to return the first element among the repeated elements.

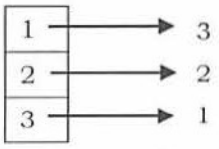
Solution: We can use the brute force solution that we used for Problem-1. For each element, since it checks whether there is a duplicate for that element or not, whichever element duplicates first will be returned.

Problem-10 For Problem-9, can we use the sorting technique?

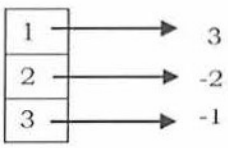
Solution: No. For proving the failed case, let us consider the following array. For example, $A = \{3, 2, 1, 2, 2, 3\}$. After sorting we get $A = \{1, 2, 2, 2, 3, 3\}$. In this sorted array the first repeated element is 2 but the actual answer is 3.

Problem-11 For Problem-9, can we use hashing technique?

Solution: Yes. But the simple hashing technique which we used for Problem-3 will not work. For example, if we consider the input array as $A = \{3, 2, 1, 2, 3\}$, then the first repeated element is 3, but using our simple hashing technique we get the answer as 2. This is because 2 is coming twice before 3. Now let us change the hashing table behavior so that we get the first repeated element. Let us say, instead of storing 1 value, initially we store the position of the element in the array. As a result the hash table will look like (after inserting 3, 2 and 1):



Now, if we see 2 again, we just negate the current value of 2 in the hash table. That means, we make its counter value as -2. The negative value in the hash table indicates that we have seen the same element two times. Similarly, for 3 (the next element in the input) also, we negate the current value of the hash table and finally the hash table will look like:



After processing the complete input array, scan the hash table and return the highest negative indexed value from it (i.e., -1 in our case). The highest negative value indicates that we have seen that element first (among repeated elements) and also repeating.

```
def FirstRepeatedElementAmongRepeatedElementsWithHash(A):
    table = {}          # hash
    max = 0
    for element in A:
            if element in table and table[element] == 1:
                    table[element] = -2
            elif element in table and table[element] < 0:
                    table[element] -= 1
            elif element != " ":
                    table[element] = 1
            else:
                    table[element] = 0
    for element in A:
            if table[element] < max:
                    max = table[element]
                    maxRepeatedElement = element
    print maxRepeatedElement, "repeated for ", abs(max), " times"
A = [3,2,1,1,2,1,2,5,5]
FirstRepeatedElementAmongRepeatedElementsWithHash(A)
```

What if the element is repeated more than twice? In this case, just skip the element if the corresponding value i is already negative.

Problem-12 For Problem-9, can we use the technique that we used for Problem-3 (negation technique)?

Solution: No. As an example of contradiction, for the array $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated element is 3. But with negation technique the result is 2.

Problem-13 **Finding the Missing Number:** We are given a list of $n - 1$ integers and these integers are in the range of 1 to n. There are no duplicates in the list. One of the integers is missing in the list. Given an algorithm to find the missing integer. **Example**: I/P: $[1, 2, 4, 6, 3, 7, 8]$ O/P: 5

Brute Force Solution: One simple solution to this is, for each number in 1 to n, check whether that number is in the given array or not.

```
def FindMissingNumber(A):
    n = len(A)
    for i in range(1,n+1):
            found  = 0
            for j in range(0,n):
                    if(i == A[j]):
                            found = 1
            if found == 0:
                    print "Missing number is ", i
A = [8,2,1,4,6,5,7,9]
FindMissingNumber(A)
```

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-14 For Problem-13, can we use sorting technique?

Solution: Yes. Sorting the list will give the elements in increasing order and with another scan we can find the missing number.

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

Problem-15 For Problem-13, can we use hashing technique?

Solution: Yes. Scan the input array and insert elements into the hash. For inserted elements, keep *counter* as 1 (assume initially all entires are filled with zeros). This indicates that the corresponding element has occurred already. Now, scan the hash table and return the element which has counter value zero.

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

Problem-16 For Problem-13, can we improve the complexity?

Solution: Yes. We can use summation formula.

1) Get the sum of numbers, $sum = n \times (n + 1)/2$.

2) Subtract all the numbers from *sum* and you will get the missing number.

Time Complexity: O(n), for scanning the complete array.

Problem-17 In Problem-13, if the sum of the numbers goes beyond the maximum allowed integer, then there can be integer overflow and we may not get the correct answer. Can we solve this problem?

Solution:

1) *XOR* all the array elements, let the result of *XOR* be *X*.
2) *XOR* all numbers from 1 to n, let *XOR* be Y.
3) *XOR* of *X* and *Y* gives the missing number.

```
def FindMissingNumber(A):
    n = len(A)
    X = 0
    for i in range(1,n+2):
            X = X ^ i
    for i in range(0,n):
            X = X ^ A[i]
    print "Missing number is ", X
A = [8,2,1,4,6,5,7,9]
FindMissingNumber(A)
```

Time Complexity: O(n), for scanning the complete array. Space Complexity: O(1).

Problem-18 **Find the Number Occurring an Odd Number of Times:** Given an array of positive integers, all numbers occur an even number of times except one number which occurs an odd number of times. Find the number in O(n) time & constant space. **Example**: I/P = [1,2,3,2,3,1,3] O/P = 3

Solution: Do a bitwise *XOR* of all the elements. We get the number which has odd occurrences. This is because, $A\ XOR\ A = 0$.

Time Complexity: O(n). Space Complexity: O(1).

Problem-19 **Find the two repeating elements in a given array:** Given an array with *size*, all elements of the array are in range 1 to n and also all elements occur only once except two numbers which occur twice. Find those two repeating numbers. For example: if the array is 4,2,4,5,2,3,1 with *size* = 7 and $n = 5$. This input has $n + 2 = 7$ elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

Solution: One simple way is to scan the complete array for each element of the input elements. That means use two loops. In the outer loop, select elements one by one and count the number of occurrences of the selected element in the inner loop. For the code below, assume that *PrintRepeatedElements* is called with $n + 2$ to indicate the size.

```
def PrintTwoRepeatedElementsBruteForce(A):
    n = len(A)
    for i in range(0,n):
            for j in range(i+1,n):
                    if(A[i] == A[j]):
                            print A[i]
A = [3,5,7,4,2,4,2,1,9]
PrintTwoRepeatedElementsBruteForce(A)
```

Time Complexity: O(n^2). Space Complexity: O(1).

Problem-20 For Problem-19, can we improve the time complexity?

Solution: Sort the array using any comparison sorting algorithm and see if there are any elements which are contiguous with the same value.

Time Complexity: O(*nlogn*). Space Complexity: O(1).

Problem-21 For Problem-19, can we improve the time complexity?

Solution: Use Count Array. This solution is like using a hash table. For simplicity we can use array for storing the counts. Traverse the array once and keep track of the count of all elements in the array using a temp array *count*[] of size n. When we see an element whose count is already set, print it as duplicate. For the code below assume that *PrintRepeatedElements* is called with $n + 2$ to indicate the size.

```
def PrintTwoRepeatedElementsHash(A):
```

```
    table = {}          # hash
    for element in A:
            #print element
            if element in table and table[element] == 1:
                    print element
                    table[element] += 1
            elif element in table:
                    table[element] += 1
            elif element != " ":
                    table[element] = 1
            else:
                    table[element] = 0
A = [3,5,7,4,2,4,2,1,9]
PrintTwoRepeatedElementsHash(A)
```

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

Problem-22 Consider Problem-19. Let us assume that the numbers are in the range 1 to n. Is there any other way of solving the problem?

Solution: Yes, by using XOR Operation. Let the repeating numbers be X and Y, if we XOR all the elements in the array and also all integers from 1 to n, then the result will be $X\ XOR\ Y$. The 1's in binary representation of $X\ XOR\ Y$ correspond to the different bits between X and Y. If the k^{th} bit of $X\ XOR\ Y$ is 1, we can XOR all the elements in the array and also all integers from 1 to n whose k^{th} bits are 1. The result will be one of X and Y.

```
# Approach is same for two repeated and two missing numbers
def findTwoRepeatingNumbersWithXOR (A):
    XOR = A[0]
    X= Y = 0
    n = len(A) - 2
    for i in range(1,len(A)):
            XOR ^= A[i]
    for i in range(1,n+1):
            XOR ^= i
    rightMostSetBitNo = XOR & ~( XOR -1)
    for i in range(0,len(A)):
            if(A[i] & rightMostSetBitNo):
                    X = X^ A[i]
            else:   Y = Y ^ A[i]
    for i in range(1,n+1):
            if(i & rightMostSetBitNo):
                    X = X ^ i
            else:   Y = Y ^ i
    print X, Y
A=[4, 2, 4, 5, 2, 3, 1]
findTwoRepeatingNumbersWithXOR(A)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-23 Consider Problem-19. Let us assume that the numbers are in the range 1 to n. Is there yet other way of solving the problem?

Solution: We can solve this by creating two simple mathematical equations. Let us assume that two numbers we are going to find are X and Y. We know the sum of n numbers is $n(n+1)/2$ and the product is $n!$. Make two equations using these sum and product formulae, and get values of two unknowns using the two equations. Let the summation of all numbers in array be S and product be P and the numbers which are being repeated are X and Y.

$$X + Y = S - \frac{n(n+1)}{2}$$
$$XY = P/n!$$

Using the above two equations, we can find out X and Y. There can be an addition and multiplication overflow problem with this approach.

Time Complexity: O(n). Space Complexity: O(1).

Problem-24 Similar to Problem-19, let us assume that the numbers are in the range 1 to n. Also, $n-1$ elements are repeating thrice and remaining element repeated twice. Find the element which repeated twice.

Solution: If we *XOR* all the elements in the array and all integers from 1 to n, then all the elements which are repeated thrice will become zero. This is because, since the element is repeating thrice and *XOR* another time from range makes that element appear four times. As a result, the output of $a\ XOR\ a\ XOR\ a\ XOR\ a = 0$. It is the same case with all elements that are repeated three times.

With the same logic, for the element which repeated twice, if we *XOR* the input elements and also the range, then the total number of appearances for that element is 3. As a result, the output of $a\ XOR\ a\ XOR\ a = a$. Finally, we get the element which repeated twice.

Time Complexity: O(n). Space Complexity: O(1).

Problem-25 Given an array of n elements. Find two elements in the array such that their sum is equal to given element K.

Brute Force Solution: One simple solution to this is, for each input element, check whether there is any element whose sum is K. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
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)
```

Time Complexity: O(n^2). This is because of two nested *for* loops. Space Complexity: O(1).

Problem-26 For Problem-25, can we improve the time complexity?

Solution: Yes. Let us assume that we have sorted the given array. This operation takes O(*nlogn*). On the sorted array, maintain indices $loIndex = 0$ and $hiIndex = n-1$ and compute $A[loIndex] + A[hiIndex]$. If the sum equals K, then we are done with the solution. If the sum is less than K, decrement *hiIndex*, if the sum is greater than K, increment *loIndex*.

```
def twoElementsWithSumKBruteForce(A, K):
  loIndex = 0
  hiIndex = len(A)-1;
  while(left < right):
     if(A[loIndex] + A[hiIndex] == K):
        return 1
     elif(A[loIndex] + A[hiIndex] < K):
        loIndex += 1
     else:
        hiIndex -= 1
  return 0
A = [1, 4, 45, 6, 10, -8]
A.sort()
print twoElementsWithSumKBruteForce(A, 11)
```

Time Complexity: O(*nlogn*). If the given array is already sorted then the complexity is O(n).
Space Complexity: O(1).

Problem-27 Does the solution of Problem-25 work even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we get the pair of numbers if they exist.

Problem-28 Is there any other way of solving Problem-25?

Solution: Yes, using hash table. Since our objective is to find two indexes of the array whose sum is K. Let us say those indexes are X and Y. That means, $A[X] + A[Y] = K$. What we need is, for each element of the input array $A[X]$, check whether $K - A[X]$ also exists in the input array. Now, let us simplify that searching with hash table.

Algorithm:

- For each element of the input array, insert it into the hash table. Let us say the current element is $A[X]$.
- Before proceeding to the next element we check whether $K - A[X]$ also exists in the hash table or not.
- Ther existence of such number indicates that we are able to find the indexes.
- Otherwise proceed to the next input element.

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

```
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)
```

Problem-29 Given an array A of n elements. Find three indices, i, j & k such that $A[i]^2 + A[j]^2 = A[k]^2$?

Solution:

Algorithm:

- Sort the given array in-place.
- For each array index i compute $A[i]^2$ and store in array.
- Search for 2 numbers in array from 0 to $i - 1$ which adds to $A[i]$ similar to Problem-25. This will give us the result in $O(n)$ time. If we find such a sum, return true, otherwise continue.

```
A.sort() # Sort the input array
for i in range(0, n):
    A[i] = A[i]*A[i]
for i in range(0, n, -1):
    res = 0
    if(res):
            //Problem-11/12 Solution
```

Time Complexity: Time for sorting + $n \times$ (Time for finding the sum) = $O(nlogn) + n \times O(n) = n^2$.
Space Complexity: $O(1)$.

Problem-30 **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

Brute Force Solution: For each element, find the *sum* with every other element in the array and compare sums. Finally, return the minimum *sum*.

```
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)
```

Time complexity: $O(n^2)$. Space Complexity: O(1).

Problem-31 Can we improve the time complexity of Problem-30?

Solution: Use Sorting.

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

```
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).

Problem-32 Given an array of n elements. Find three elements in the array such that their sum is equal to given element K?

Brute Force Solution: The default solution to this is, for each pair of input elements check whether there is any element whose sum is K. This we can solve just by using three simple for loops. The code for this solution can be given as:

```
def twoElementsWithSumKBruteForce(A, K):
    n = len(A)
    for i in range(0,n-2):
            for j in range(i+1,n-1):
                    for k in range(j+1,n):
                            if(A[i] + A[j] + A[k] == K):
                                    print "yes-->", A[i], " + ", A[j], " + ", A[k], " = ", K
                                    return 1
    return 0
```

```
A = [1, 6, 45, 4, 10, 18]
A.sort()
twoElementsWithSumKBruteForce(A, 22)
```

Time Complexity: $O(n^3)$, for three nested *for* loops. Space Complexity: O(1).

Problem-33 Does the solution of Problem-32 work even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we can find three numbers whose sum is K if they exist.

Problem-34 Can we use sorting technique for solving Problem-32?

Solution: Yes.

```
def threeElementsWithSumKWithSorting(A, K):
    n = len(A)
    left = 0
    right = n-1
    for i in range(0,n-2):
            left = i + 1
            right = n-1
            while(left < right):
                    print A[i] + A[left] + A[right], K
                    if( A[i] + A[left] + A[right] == K):
                            print "yes-->", A[i], " + ", A[left], " + ", A[right], " = ", K
                            return 1
                    elif(A[i] + A[left] + A[right] < K):
                            left += 1
                    else:
                            right -= 1
    return 0
A = [1, 6, 45, 4, 10, 18]
A.sort()
print threeElementsWithSumKWithSorting(A, 23)
```

Time Complexity: Time for sorting + Time for searching in sorted list $= O(nlogn) + O(n^2) \approx O(n^2)$. This is because of two nested *for* loops. Space Complexity: O(1).

Problem-35 Can we use hashing technique for solving Problem-32?

Solution: Yes. Since our objective is to find three indexes of the array whose sum is K. Let us say those indexes are X, Y and Z. That means, $A[X] + A[Y] + A[Z] = K$.

Let us assume that we have kept all possible sums along with their pairs in hash table. That means the key to hash table is $K - A[X]$ and values for $K - A[X]$ are all possible pairs of input whose sum is $K - A[X]$.

Algorithm:

- Before starting the search, insert all possible sums with pairs of elements into the hash table.
- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: The time for storing all possible pairs in Hash table + searching = $O(n^2) + O(n^2) \approx O(n^2)$. Space Complexity: $O(n)$.

Problem-36 Given an array of n integers, the $3 - sum\ problem$ is to find three integers whose sum is closest to *zero*.

Solution: This is the same as that of Problem-32 with K value is zero.

Problem-37 Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $1 \leq k \leq n$ 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.

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

```
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)
```

The recursion equation is $T(n) = 2T(n/2) + c$. Using master theorem, we get O($logn$).

Problem-38 If we don't know n, how do we solve the Problem-37?

Solution: Repeatedly compute $A[1], A[2], A[4], A[8], A[16]$ and so on, until we find a value of n such that $A[n] > 0$.

Time Complexity: O($logn$), since we are moving at the rate of 2. Refer to *Introduction to Analysis of Algorithms* chapter for details on this.

Problem-39 Given an input array of size unknown with all $1's$ in the beginning and $0's$ in the end. Find the index in the array from where $0's$ start. Consider there are millions of $1's$ and $0's$ in the array. E.g. array contents 1111111.....1100000.....0000000.

Solution: This problem is almost similar to Problem-38. Check the bits at the rate of $2^K$where $k = 0, 1, 2 ....$ Since we are moving at the rate of 2, the complexity is O($logn$).

Problem-40 Given a sorted array of n integers that has been rotated an unknown number of times, give a O($logn$) algorithm that finds an element in the array. **Example**: Find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14) **Output**: 8 (the index of 5 in the array)

Solution: Let us assume that the given array is A[]and use the solution of Problem-37 with an extension. The function below *FindPivot* returns the k value (let us assume that this function returns the index instead of the value). Find the pivot point, divide the array into two sub-arrays and call binary search.

The main idea for finding the pivot point is – for a sorted (in increasing order) and pivoted array, the pivot element is the only element for which the next element to it is smaller than it. Using the above criteria and the binary search methodology we can get pivot element in O($logn$) time.

Algorithm:

1) Find out the pivot point and divide the array into two sub-arrays.
2) Now call binary search for one of the two sub-arrays.
   a. if the element is greater than the first element then search in left subarray.
   b. else search in right subarray.
3) If element is found in selected sub-array, then return index *else* return −1.

```
def findInRotatedSortedArray(A, target):
    left = 0
    right = len(A) - 1
    while left <= right:
      mid = (left + right) / 2
      if A[mid] == target:
            return mid
      if A[mid] >= A[left]:
            if A[left] <= target < A[mid]:
              right = mid - 1
            else:
              left = mid + 1
      else:
            if A[mid] < target <= A[right]:
```

```
            left = mid + 1
        else:
            right = mid - 1
    return -1
A = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]
print findInRotatedSortedArray(A, 2)
```

Time complexity: O(*logn*).

Problem-41 For Problem-40, can we solve with recursion?

Solution: Yes.

```
def findInRotatedSortedArrayWithRecursion(A, target):
    if A==None or len(A)==0:
        return -1;
    low=0;
    high=len(A)-1
    return findWithRecursion(A, target, low, high)
def findWithRecursion(A, target, low, high):
    if low>high:
        return -1
    mid=(low+high)/2
    if A[mid]==target:
        return mid
    if A[low]<A[mid]:
        if A[low]<=target<A[mid]:
            return findWithRecursion(A, target, low, mid-1)
        return findWithRecursion(A, target, mid+1, high)
    elif A[low]>A[mid]:
        if A[mid]<target<=A[high]:
            return findWithRecursion(A, target, mid+1, high)
        return findWithRecursion(A, target, low, mid-1)
    else:
        if A[mid]!=A[high]:
            return findWithRecursion(A, target, mid+1, high)
        result=findWithRecursion(A, target, low, mid-1)
        if result!=-1:
            return result
        return findWithRecursion(A, target, mid+1, high)
A = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]
print findInRotatedSortedArrayWithRecursion(A, 5)
```

Time complexity: O(*logn*).

Problem-42 **Bitonic search:** An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array A of n distinct integers, describe how to determine whether a given integer is in the array in O(*logn*) steps.

Solution: The solution is the same as that for Problem-37.

Problem-43 Yet, other way of framing Problem-37.
Let *A*[] be an array that starts out increasing, reaches a maximum, and then decreases. Design an O(*logn*) algorithm to find the index of the maximum value.

Problem-44 Give an O(*nlogn*) algorithm for computing the median of a sequence of n integers.

Solution: Sort and return element at $\frac{n}{2}$.

Problem-45 Given two sorted lists of size m and n, find median of all elements in O($log\ (m+n)$) time.

Solution: Refer to *Divide and Conquer* chapter.

Problem-46 Given a sorted array *A* of n elements, possibly with duplicates, find the index of the first occurrence of a number in O(*logn*) time.

Solution: To find the first occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

```
mid == low && A[mid] == data  ||  A[mid] == data && A[mid-1] < data
```

```
def binarySearchFirstOccurrence(A, target):
    if A==None or len(A)==0:
        return -1;
    high=len(A)-1
  low = 0
  m = 0
  lastFound = -1;
  while( 1 ):
    if ( low>high ): return lastFound
    m = (low+high)/2
    if ( A[m] == target ):
                lastFound = m; high = m-1
    if ( A[m] < target ): low = m+1
    if ( A[m] > target ): high = m-1
  return m
A = [5, 6, 9, 12, 15, 21, 21, 34, 45, 57, 70, 84]
print binarySearchFirstOccurrence(A,21)
```

Time Complexity: O(*logn*).

Problem-47 Given a sorted array A of n elements, possibly with duplicates. Find the index of the last occurrence of a number in O(*logn*) time.

Solution: To find the last occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

```
mid == high && A[mid] == data  ||  A[mid] == data && A[mid+1] > data
```

```
def binarySearchLastOccurrence(A, target):
    if A==None or len(A)==0:
        return -1;
    high=len(A)-1
  low = 0
  m = 0
  lastFound = -1;
  while( 1 ):
    if ( low>high ): return lastFound
    m = (low+high)/2
    if ( A[m] == target ):
                lastFound = m; low = m+1
    if ( A[m] < target ): low = m+1
    if ( A[m] > target ): high = m-1
  return m
A = [5, 6, 9, 12, 15, 21, 21, 34, 45, 57, 70, 84]
print binarySearchLastOccurrence(A,21)
```

Time Complexity: O(*logn*).

Problem-48 Given a sorted array of n elements, possibly with duplicates. Find the number of occurrences of a number.

Brute Force Solution: Do a linear search of the array and increment count as and when we find the element data in the array.

```
def LinearSearchCOunt(A, data):
    count = 0
    for i in range (0, len(A)):
            if(A[i] == data):
                count += 1
    return count
A= [7,3,6,3,3,6,7]
print LinearSearchCOunt(A, 7)
```

Time Complexity: O(n).

Problem-49 Can we improve the time complexity of Problem-48?

Solution: Yes. We can solve this by using one binary search call followed by another small scan.

Algorithm:

- Do a binary search for the *data* in the array. Let us assume its position is K.
- Now traverse towards the left from K and count the number of occurrences of *data*. Let this count be $leftCount$.
- Similarly, traverse towards right and count the number of occurrences of *data*. Let this count be $rightCount$.
- Total number of occurrences = $leftCount + 1 + rightCount$

Time Complexity – $O(logn + S)$ where S is the number of occurrences of *data*.

Problem-50 Is there any alternative way of solving Problem-48?

Solution:

Algorithm:

- Find first occurrence of *data* and call its index as $firstOccurrence$ (for algorithm refer to Problem-46)
- Find last occurrence of *data* and call its index as $lastOccurrence$ (for algorithm refer to Problem-47)
- Return $lastOccurrence - firstOccurrence + 1$

Time Complexity = $O(logn + logn) = O(logn)$.

Problem-51 What is the next number in the sequence 1, 11, 21 and why?

Solution: Read the given number loudly. This is just a fun problem.

One One
Two Ones
One two, one one→ 1211

So the answer is: the next number is the representation of the previous number by reading it loudly.

Problem-52 Finding second smallest number efficiently.

Solution: We can construct a heap of the given elements using up just less than n comparisons (Refer to the *Priority Queues* chapter for the algorithm). Then we find the second smallest using $logn$ comparisons for the GetMax() operation. Overall, we get $n + logn + constant$.

Problem-53 Is there any other solution for Problem-52?

Solution: Alternatively, split the n numbers into groups of 2, perform $n/2$ comparisons successively to find the largest, using a tournament-like method. The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones that the maximum popped. This will yield $logn - 1$ comparison for a total of $n + logn - 2$. The above solution is called the *tournament problem*.

Problem-54 An element is a majority if it appears more than $n/2$ times. Give an algorithm takes an array of n element as argument and identifies a majority (if it exists).

Solution: The basic solution is to have two loops and keep track of the maximum count for all different elements. If themaximum count becomes greater than $n/2$, then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$, then the majority element doesn't exist.

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-55 Can we improve Problem-54 time complexity to $O(nlogn)$?

Solution: Using binary search we can achieve this. Node of the Binary Search Tree (used in this approach) will be as follows.

```
class TreeNode(object):
    def __init__(self, value):
        self.data = value
        self.left = None
        self.right = None
        self.count = None
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if the count of a node becomes more than $n/2$, then return. This method works well for the cases where $n/2 + 1$ occurrences of the majority element are present at the start of the array, for example {1, 1, 1, 1, 1, 2, 3, and 4}.

Time Complexity: If a binary search tree is used then worst time complexity will be $O(n^2)$. If a balanced-binary-search tree is used then $O(nlogn)$. Space Complexity: $O(n)$.

Problem-56 Is there any other of achieving O(*nlogn*) complexity for Problem-54?

Solution: Sort the input array and scan the sorted array to find the majority element.

Time Complexity: O(*nlogn*). Space Complexity: O(1).

Problem-57 Can we improve the complexity for Problem-54?

Solution: If an element occurs more than $n/2$ times in A then it must be the median of A. But, the reverse is not true, so once the median is found, we must check to see how many times it occurs in A. We can use linear selection which takes O(n) time (for algorithm, refer to *Selection Algorithms* chapter).

```
int CheckMajority(int A[], in n) {
    1) Use linear selection to find the median m of A.
    2) Do one more pass through A and count the number of occurrences of m.
        a. If m occurs more than n/2 times then return true;
        b. Otherwise return false.
}
```

Problem-58 Is there any other way of solving Problem-54?

Solution: Since only one element is repeating, we can use a simple scan of the input array by keeping track of the count for the elements. If the count is 0, then we can assume that the element visited for the first time otherwise that the resultant element.

```
def majorityElement(A):
    count = 0
    element = -1
    n = len(A)
    if n == 0:
            return
    for i in range(0, n-1):
            if(count == 0) :
                    element = A[i]
                    count = 1
            elif(element == A[i]):
                    count += 1
            else:
                    count -= 1
    return element
A= [7,3,2,3,3,6,3]
print majorityElement(A)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-59 Given an array of $2n$ elements of which n elements are the same and the remaining n elements are all different. Find the majority element.

Solution: The repeated elements will occupy half the array. No matter what arrangement it is, only one of the below will be true:

- All duplicate elements will be at a relative distance of 2 from each other. Ex: **n**, *1*, **n**, *100*, **n**, *54*, **n** ...
- At least two duplicate elements will be next to each other.
  Ex: $n, n, 1, 100, n, 54, n, \ldots.$
  $n, 1, n, n, n, 54, 100 \ldots$
  $1, 100, 54, n, n, n, n \ldots.$

In worst case, we will need two passes over the array:

- First Pass: compare $A[i]$ and $A[i+1]$
- Second Pass: compare $A[i]$ and $A[i+2]$

Something will match and that's your element. This will cost O(n) in time and O(1) in space.

Problem-60 Given an array with $2n+1$ integer elements, n elements appear twice in arbitrary places in the array and a single integer appears only once somewhere inside. Find the lonely integer with O(n) operations and O(1) extra memory.

Solution: Except for one element, all elements are repeated. We know that $A \; XOR \; A = 0$. Based on this if we *XOR* all the input elements then we get the remaining element.

```
def singleNumber(A):
```

```
    i = res =0
    for i in range (0, len(A)):
            res = res ^ A[i]
    return res
A= [7,3,6,3,3,6,7]
print singleNumber(A)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-61 **Throwing eggs from an n-story building:** Suppose we have an n story building and a number of eggs. Also assume that an egg breaks if it is thrown from floor F or higher, and will not break otherwise. Devise a strategy to determine floor F, while breaking O($logn$) eggs.

Solution: Refer to *Divide and Conquer* chapter.

Problem-62 **Local minimum of an array:** Given an array A of n distinct integers, design an O($logn$) algorithm to find a *local minimum*: an index i such that $A[i-1] < A[i] < A[i+1]$.

Solution: Check the middle value $A[n/2]$, and two neighbors $A[n/2-1]$ and $A[n/2+1]$. If $A[n/2]$ is local minimum, stop; otherwise search in half with smaller neighbor.

Problem-63 Give an $n \times n$ array of elements such that each row is in ascending order and each column is in ascending order, devise an O(n) algorithm to determine if a given element x is in the array. You may assume all elements in the $n \times n$ array are distinct.

Solution: Let us assume that the given matrix is $A[n][n]$. Start with the last row, first column [or first row, last column]. If the element we are searching for is greater than the element at $A[1][n]$, then the first column can be eliminated. If the search element is less than the element at $A[1][n]$, then the last row can be completely eliminated. Once the first column or the last row is eliminated, start the process again with the left-bottom end of the remaining array. In this algorithm, there would be maximum n elements that the search element would be compared with.

Time Complexity: O(n). This is because we will traverse at most $2n$ points. Space Complexity: O(1).

Problem-64 Given an $n \times n$ array a of n^2 numbers, give an O(n) algorithm to find a pair of indices i and j such that $A[i][j] < A[i+1][j], A[i][j] < A[i][j+1], A[i][j] < A[i-1][j]$, and $A[i][j] < A[i][j-1]$.

Solution: This problem is the same as Problem-63.

Problem-65 Given $n \times n$ matrix, and in each row all 1's are followed by 0's. Find the row with the maximum number of 0's.

Solution: Start with first row, last column. If the element is 0 then move to the previous column in the same row and at the same time increase the counter to indicate the maximum number of 0's. If the element is 1 then move to the next row in the the same column. Repeat this process until your reach last row, first column.

Time Complexity: O($2n$) ≈O(n) (similar to Problem-63).

Problem-66 Given an input array of size unknown, with all numbers in the beginning and special symbols in the end. Find the index in the array from where the special symbols start.

Solution: Refer to *Divide and Conquer* chapter.

Problem-67 **Separate even and odd numbers:** Given an array $A[\,]$, write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers. **Example**: Input = {12, 34, 45, 9, 8, 90, 3} Output = {12, 34, 90, 8, 9, 45, 3}

Note: In the output, the order of numbers can be changed, i.e., in the above example 34 can come before 12, and 3 can come before 9.

Solution: The problem is very similar to *Separate 0's and 1's* (Problem-68) in an array, and both problems are variations of the famous *Dutch national flag problem*.

Algorithm: The logic is similar to Quick sort.

1) Initialize two index variables left and right: $left = 0$, $right = n-1$
2) Keep incrementing the left index until you see an odd number.
3) Keep decrementing the right index until youe see an even number.
4) If $left < right$ then swap $A[left]$ and $A[right]$

```
def separateEvenOdd(A):
    left = 0; right = len(A)-1
```

```
    while(left < right):
            while(A[left]%2 == 0 and left < right):
                    left += 1
            while(A[right]%2 == 1 and left < right):
                    right -= 1
            if(left < right):
                    A[left], A[right] = A[right], A[left]
                    left += 1
                    right -= 1
A= [12, 34, 45, 9, 8, 90, 3]
separateEvenOdd(A)
print A
```

Time Complexity: $O(n)$.

Problem-68 The following is another way of structuring Problem-67, but with a slight difference.

Separate 0's and 1's in an array: We are given an array of 0's and 1's in random order. Separate 0's on the left side and 1's on the right side of the array. Traverse the array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0] **Output array** = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Solution: Counting 0's or 1's

1. Count the number of 0's. Let the count be C.
2. Once we have the count, put C 0's at the beginning and 1's at the remaining $n - C$ positions in the array.

Time Complexity: $O(n)$. This solution scans the array two times.

Problem-69 Can we solve Problem-68 in one scan?

Solution: Yes. Use two indexes to traverse: Maintain two indexes. Initialize the first index left as 0 and the second index right as $n - 1$. Do the following while $left < right$:

1) Keep the incrementing index left while there are 0s in it
2) Keep the decrementing index right while there are 1s in it
3) If left < right then exchange $A[left]$ and $A[right]$

```
def separateZerosAndOnes(A):
    left = 0; right = len(A)-1
    while(left < right):
            while(A[left] == 0 and left < right):
                    left += 1
            while(A[right] == 1 and left < right):
                    right -= 1
            if(left < right):
                    A[left], A[right] = A[right], A[left]
                    left += 1
                    right -= 1
A= [1, 1, 0, 0, 1, 0, 1]
separateZerosAndOnes(A)
print A
```

Time Complexity: $O(n)$. Space Complexity: O(1).

Problem-70 **Sort an array of 0's, 1's and 2's [or R's, G's and B's]:** Given an array A[] consisting of 0's, 1's and 2's, give an algorithm for sorting $A[]$.The algorithm should put all 0's first, then all 1's and finally all 2's at the end. **Example Input** = {0,1,1,0,1,2,1,2,0,0,0,1}, **Output** = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2}

Solution:

```
def sorting012sDutchFlagProblem(A):
    n = len(A)
    zero = 0; two = n-1
    # Write 1 at the beginning; 2 at the end.
    cur = 0
    while cur <= two:
        print cur, A, zero, two
        if A[cur] == 0:
            if cur > zero:
                A[zero], A[cur] = A[cur], A[zero]
                zero += 1
```

```
            else: # TRICKY PART.
                # cur == zero and A[cur] == A[zero] == 0
                cur += 1
                zero += 1
        elif A[cur] == 2:
            if cur < two:
                A[two], A[cur] = A[cur], A[two]
                two -= 1
            else:
                break
        else:
            cur += 1
    print A, '\n'
    return A
sorting012sDutchFlagProblem([2,0,1,0,2,1,2,2,1,1])
sorting012sDutchFlagProblem([2,1,2,1,2,0])
sorting012sDutchFlagProblem([0,0,1,2,2,2,0,0,0])
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-71 **Maximum difference between two elements:** Given an array $A[]$ of integers, find out the difference between any two elements such that the larger element appears after the smaller number in $A[]$.
Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Difference between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then the returned value should be 2 (Difference between 7 and 9)

Solution: Refer to *Divide and Conquer* chapter.

Problem-72 Given an array of 101 elements. Out of 101 elements, 25 elements are repeated twice, 12 elements are repeated 4 times, and one element is repeated 3 times. Find the element which repeated 3 times in O(1).

Solution: Before solving this problem, let us consider the following *XOR* operation property: $a\ XOR\ a = 0$. That means, if we apply the *XOR* on the same elements then the result is 0.

Algorithm:

- *XOR* all the elements of the given array and assume the result is A.
- After this operation, 2 occurrences of the number which appeared 3 times becomes 0 and one occurrence remains the same.
- The 12 elements that are appearing 4 times become 0.
- The 25 elements that are appearing 2 times become 0.
- So just *XOR'ing* all the elements gives the result.

Time Complexity: O(n), because we are doing only one scan. Space Complexity: O(1).

Problem-73 Given a number n, give an algorithm for finding the number of trailing zeros in $n!$.

Solution:

```
def numberOfTrailingZerosOfFactorialNumber(n):
    count = 0
    if(n < 0):
            return -1
    i = 5
    while n/i >0:
            count += n / i
            i *= 5
    return count
print numberOfTrailingZerosOfFactorialNumber(100)
```

Time Complexity: O($logn$).

Problem-74 Given an array of $2n$ integers in the following format $a1\ a2\ a3 \ldots an\ b1\ b2\ b3 \ldots bn$. Shuffle the array to $a1\ b1\ a2\ b2\ a3\ b3 \ldots an\ bn$ without any extra memory.

Solution: A brute force solution involves two nested loops to rotate the elements in the second half of the array to the left. The first loop runs n times to cover all elements in the second half of the array. The second loop rotates the elements to the left. Note that the start index in the second loop depends on which element we are rotating and the end index depends on how many positions we need to move to the left.

```
def rearrangeArrayElementsA1B1A2B2(A):
    n = len(A)//2
    i =0;    q =1; k = n
    while (i<n):

            j = k
            while j > i+ q:
                    A[j], A[j-1] = A[j-1], A[j]
                    j -= 1
            i += 1; k += 1; q += 1
A = [1,3,5,6,2,4,6,8]
rearrangeArrayElementsA1B1A2B2(A)
print A
```

Time Complexity: $O(n^2)$.

Problem-75 Can we improve Problem-74 solution?

Solution: Refer to the *Divide and Conquer* chapter. A better solution of time complexity O(*nlogn*) can be achieved using the *Divide and Concur* technique. Let us look at an example

1. Start with the array: *a*1 *a*2 *a*3 *a*4 *b*1 *b*2 *b*3 *b*4
2. Split the array into two halves: *a*1 *a*2 *a*3 *a*4 : *b*1 *b*2 *b*3 *b*4
3. Exchange elements around the center: exchange *a*3 *a*4 with *b*1 *b*2 and you get: *a*1 *a*2 *b*1 *b*2 *a*3 *a*4 *b*3 *b*4
4. Split *a*1 *a*2 *b*1 *b*2 into *a*1 *a*2 : *b*1 *b*2. Then split *a*3 *a*4 *b*3 *b*4 into *a*3 *a*4 : *b*3 *b*4
5. Exchange elements around the center for each subarray you get: *a*1 *b*1 *a*2 *b*2 and *a*3 *b*3 *a*4 *b*4

Note that this solution only handles the case when $n = 2^i$ where $i = 0, 1, 2, 3$, etc. In our example $n = 2^2 = 4$ which makes it easy to recursively split the array into two halves. The basic idea behind swapping elements around the center before calling the recursive function is to produce smaller size problems. A solution with linear time complexity may be achieved if the elements are of a specific nature. For example, if you can calculate the new position of the element using the value of the element itself. This is nothing but a hashing technique.

Problem-76 Given an array A[], find the maximum j – i such that A[j] > A[i]. For example, Input: {34, 8, 10, 3, 2, 80, 30, 33, 1} and Output: 6 (j = 7, i = 1).

Solution: Brute Force Approach: Run two loops. In the outer loop, pick elements one by one from the left. In the inner loop, compare the picked element with the elements starting from the right side. Stop the inner loop when you see an element greater than the picked element and keep updating the maximum $j - i$ so far.

```
def maxIndexDiff(A):
    maxJ = maxI = maxDiff = -1
    n = len(A)
    for i in range(0,n):
            j =n-1
            while(j > i):
                    if(A[j] > A[i] and maxDiff < (j - i)):
                            maxDiff = j - i
                            maxI =  i;maxJ = j
                    j -= 1
    return maxDiff,  maxI, maxJ
A=[34, 8, 10, 3, 2, 80, 30, 33, 1]
print maxIndexDiff(A)
```

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-77 Can we improve the complexity of Problem-76?

Solution: To solve this problem, we need to get two optimum indexes of A[]: left index *i* and right index *j*. For an element A[i], we do not need to consider A[i] for the left index if there is an element smaller than A[i] on the left side of A[i]. Similarly, if there is a greater element on the right side of A[j] then we do not need to consider this j for the right index.

So we construct two auxiliary Arrays LeftMins[] and RightMaxs[] such that LeftMins[i] holds the smallest element on the left side of A[i] including A[i], and RightMaxs[j] holds the greatest element on the right side of A[j] including A[j]. After constructing these two auxiliary arrays, we traverse both these arrays from left to right.

While traversing LeftMins[] and RightMaxs[], if we see that LeftMins[i] is greater than RightMaxs[j], then we must move ahead in LeftMins[] (or do i++) because all elements on the left of LeftMins[i] are greater than or equal to LeftMins[i]. Otherwise we must move ahead in RightMaxs[j] to look for a greater $j - i$ value.

```
def maxIndexDiff(A):
    n = len(A)
    LeftMins = [0]*(n)
    RightMaxs= [0]*(n)
    LeftMins[0] = A[0]
    for i in range(1,n):
            LeftMins[i] = min(A[i], LeftMins[i-1])
    RightMaxs[n-1] = A[n-1]
    for j in range(n-2,-1,-1):
            RightMaxs[j] = max(A[j], RightMaxs[j+1])
    i = 0; j = 0; maxDiff = -1;
    while (j < n and i < n):
            if (LeftMins[i] < RightMaxs[j]):
                    maxDiff = max(maxDiff, j-i)
                    j = j + 1
            else:
                    i = i+1
    return maxDiff
A=[34, 8, 10, 3, 2, 80, 30, 33, 1]
print maxIndexDiff(A)
```

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

Problem-78 Given an array of elements, how do you check whether the list is pairwise sorted or not? A list is considered pairwise sorted if each successive pair of numbers is in sorted (non-decreasing) order.

Solution:

```
def checkPairwiseSorted(A):
    n = len(A)
    if (n == 0 or n == 1):
            return 1
    for i in range(0,n-1,2):
            if (A[i] > A[i+1]):
                    return 0
    return 1
A=[34, 48, 10, 13, 2, 80, 30, 23]
print checkPairwiseSorted(A)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-79 Given an array of n elements, how do you print the frequencies of elements without using extra space. Assume all elements are positive, editable and less than n.

Solution: Use *negation* technique.

```
def frequencyCounter(A):
    pos = 0
    n = len(A)
    while(pos < n):
            expectedPos = A[pos] - 1
            if(A[pos] > 0 and A[expectedPos] > 0):
                    A[pos], A[expectedPos] = A[expectedPos],A[pos]
                    A[expectedPos] = -1
            elif(A[pos] > 0):
                    A[expectedPos] -= 1
                    A[pos] = 0
                    pos += 1
            else:
                    pos += 1
    for i in range(1,n):
            print  i + 1 ,"--->",abs(A[i])
A = [10, 1, 9, 4, 7, 6, 5,5, 1, 2, 1]
frequencyCounter(A)
```

Array should have numbers in the range [1, n] (where n is the size of the array). The if condition (A[pos] > 0 && A[$expectedPos$] > 0) means that both the numbers at indices pos and $expectedPos$ are actual numbers in the array but not their frequencies. So we will swap them so that the number at the index pos will go to the position where it should have been if the numbers 1, 2, 3, ..., n are kept in 0, 1, 2, ..., $n-1$ indices. In the above example input array, initially $pos = 0$, so 10 at index 0 will go to index 9 after the swap. As this is the first occurrence of 10, make it to -1. Note that we are storing the frequencies as negative numbers to differentiate between actual numbers and frequencies.

The else if condition (A[pos] > 0) means A[pos] is a number and A[$expectedPos$] is its frequency without including the occurrence of A[pos]. So increment the frequency by 1 (that is decrement by 1 in terms of negative numbers). As we count its occurrence we need to move to next pos, so $pos++$, but before moving to that next position we should make the frequency of the number $pos+1$ which corresponds to index pos of zero, since such a number has not yet occurred.

The final else part means the current index pos already has the frequency of the number $pos+1$, so move to the next pos, hence $pos++$.

Time Complexity: O(n). Space Complexity: O(1).

Problem-80 An, array, A contains n integers from the range X to Y. Also, there is one number that is not in A from the range X to Y. Design an O(n) time algorithm for finding that number.

Solution: The algorithm for finding the number that is not in array A:

```
import sys
def findMissingNumberFromGivenRange(A, X, Y):
    n = len(A)
    S = [-sys.maxint]*(n)
    missingNum = -sys.maxint
    for i in range(0,n):
            S[A[i]-X]=A[i]
    for i in range(0,n):
            if(S[i] == -sys.maxint):
                    missingNum = i + X
                    break
    return missingNum
A = [10, 16, 14, 12, 11, 10, 13 ,15, 17, 12, 19]
print findMissingNumberFromGivenRange(A, 10, 20)
```

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