Solution - Assignment 2

Question 1 Solution :

It is maximum sub array problem :

Brute force approach O(n^2)

```
int max_Subarray_Sum ( int A[] , int n)
{
   int max_sum = 0
   for ( i = 0 to n-1)
   {
      sum=0
      for( j = i to n-1)
      {
        sum = sum +  A[j]
        if (sum > max_sum)
            max_sum = sum
       }
   }
retun max_sum
}
```

Linear Approach O(n)

```
def maximum_subrray_sum(li):
cur, ans = 0, 0
for num in li:
cur = max(cur + num, 0)
ans = max(ans, cur)
return ans
if __name__ == '__main__':
li = [3, -1, -5, 5, -4, 2, 2, 2]
print li, '->', maximum_subrray_sum(li)
Q5 (
```

Question 2 Solution :

O(nlogn) time:

First sort both sequences A and B using merge sort and then apply this linear algorithm :
Start “i” from first index of A and “j” from last index of B
if(A[i] + B[j] < x) -> update index i to be i + 1
if(A[i] + B[j] > x) -> update index j to be j - 1
if(A[a] + B[b] = x) -> success

O(n) time :

```
countingSort(array):
size = len(array)
output = [0] * size
count = [0] * 10
for i in range(0, size):
count[sqrt(array[i])] += 1//Modification
for i in range(1, 10):
count[i] += count[i - 1]
i = size – 1
while i >= 0:
output[count[array[i]] - 1] = array[i]
count[array[i]] -= 1
i -= 1
for i in range(0, size):

array[i] = (output[i]* output[i]) //Modification
```

Then apply the following linear algorithm
Start “i” from first index of A and “j” from last index of B
if(A[i] + B[j] < x) -> update index i to be i + 1
if(A[i] + B[j] > x) -> update index j to be j - 1
if(A[a] + B[b] = x) -> success

Question 3 Solution :

```
findIndex(A, low, high)

        A ← a sorted list of N numbers
        m ← (low + high)/2
        if A[m] == m then
        return m
        else if A[m] < m then
        return f indIndex(A, low, mid − 1)
        else if A[m] > m then
        return f indIndex(A, mid + 1, high)
        end if
```

Question 4 Solution :

```
def pushZerosToEnd(arr, n):
    count = 0 # Count of non-zero elements

    # Traverse the array. If element
    # encountered is non-zero, then
    # replace the element at index
    # 'count' with this element
    for i in range(n):
        if arr[i] != 0:

            # here count is incremented
            arr[count] = arr[i]
            count+=1

    # Now all non-zero elements have been
    # shifted to front and 'count' is set
    # as index of first 0. Make all
    # elements 0 from count to end.
    while count < n:
        arr[count] = 0
        count += 1

# Driver code
arr = [1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9]
n = len(arr)
pushZerosToEnd(arr, n)
```

```
print("Array after pushing all zeros to end of array:")
print(arr)
```

Question 5 Solution :

Pictorial Representation of Jump Search with an Example

Let us trace the above algorithm using an example:

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77

Step 1: **m = √n = 4** (Block Size)

Step 2: Compare `A[0]` with `item`. Since **A[0] != item** and **A[0]<item**, skip to the next block

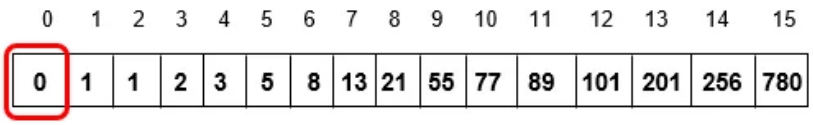

Figure 2: Comparing A[0] and item

Step 3: Compare A[3] with item. Since A[3] != itemand A[3]<item, skip to the next block

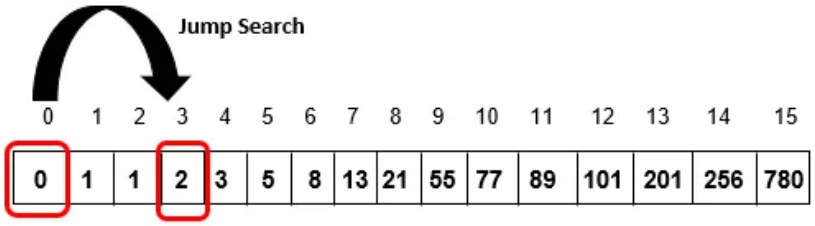

Figure 3: Comparing A[3] and item

Step 4: Compare A[6] with item. Since A[6] != itemand A[6]<item, skip to the next block

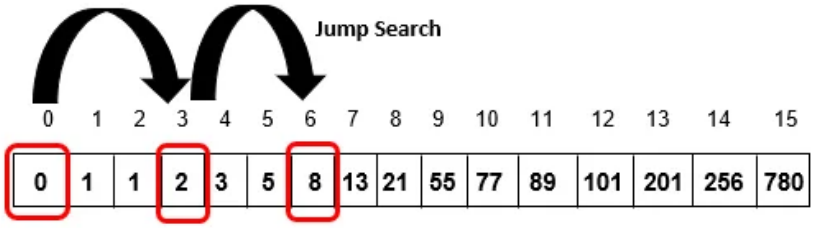

Figure 4: Comparing A[6] and item

Step 5: Compare A[9] with item. Since A[9] != itemand A[9]<item, skip to the next block

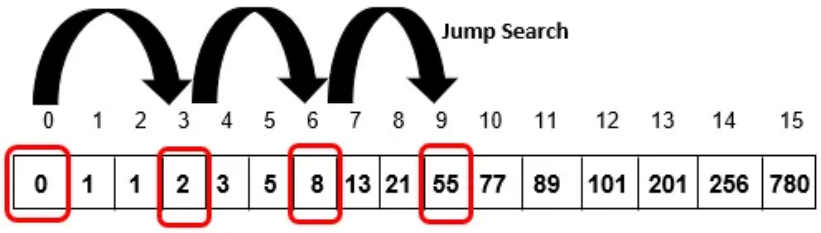

Figure 5: Comparing A[9] and item

Step 6: Compare A[12] with item. Since A[12] != item and A[12] >item, skip to A[9] (beginning of the current block) and perform a linear search.

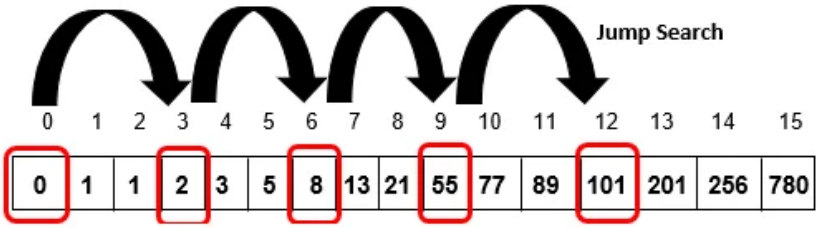

Figure 6: Comparing A[12] and item

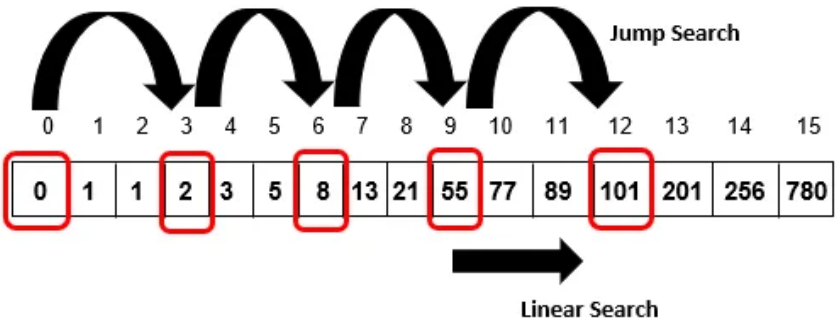

Figure 7: Comparing A[9] and item

(Linear Search)

- Compare A[9] with item. Since A[9] != item, scan the next element
- Compare A[10] with item. Since A[10] == item, index 10 is printed as the valid location and the algorithm will terminate

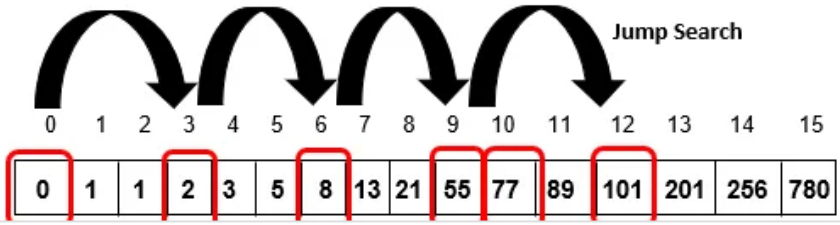

Figure 8: Comparing A[10] and

```
#include<cmath>

using namespace std;
int jump_Search(int a[], int n, int item) {
   int i = 0;
   int m = sqrt(n); //initializing block size= √(n)

   while(a[m] <= item && m < n) {
      // the control will continue to jump the blocks
      i = m;  // shift the block
      m += sqrt(n);
      if(m > n - 1)  // if m exceeds the array size
         return -1;
   }

   for(int x = i; x<m; x++) { //linear search in current block
      if(a[x] == item)
         return x; //position of element being searched
```

Jump search has been implemented above. Students may apply binary search easily as it has been covered in class so I am not mentioning here iterations of binary search

Pros and cons: Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be searched is the smallest element or smaller than the smallest).

Solution 5 Part (b):
The Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side and so on…… Exponential The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater. Once we find an index i

(after repeated doubling of i), we know that the element must be present between i/2 and i (Why i/2? because we could not find a greater value in previous iteration) and so on…….

Question 6 Solution :

Suppose you have numbers {4,1,3,2,16,9,10,14,8,7} then they can be sorted using heap sort by this way : (understand working through this, input is different in assignment 2 for this question)