# Two Pointer Technique

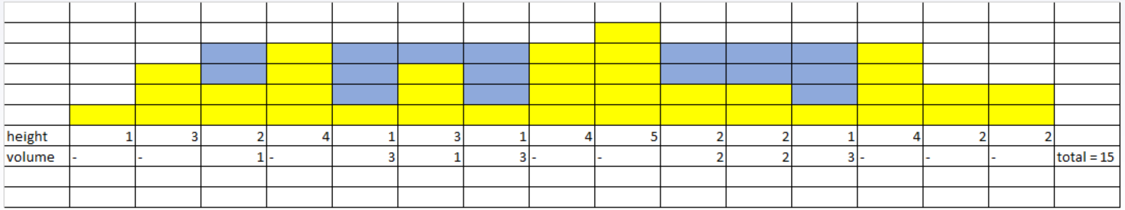
Created by Sheetal Atre

# Brute Force Problems

Bottlenecks

Unnecessary work

Duplicate work



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| height | 1 | 3 | 2 | 4 | 1 | 3 | 1 | 4 | 5 | 2 | 2 | 1 | 4 | 2 | 2 | |
| volume | - | - | 1 | - | 3 | 1 | 3 | - | - | 2 | 2 | 3 | - | - | - | total = 15 |

# Applications of Two Pointer

## Solving problems in array or string

Reverse string

Quick sort, merge sort

Z function in Riemann zeta function, which is useful in number theory for investigating properties of prime numbers.

Prefix function – longest prefix/suffix of a string

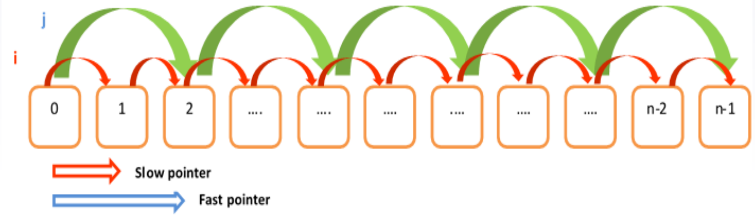
Volume of rainwater trapped between towers

# What is Two Pointer Technique?

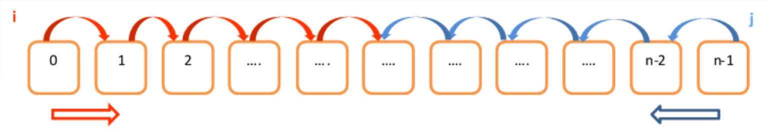
- Pointers =
    - Are index, no relation with C++ pointers
- Constraints =
    - Work on fixed length arrays
- Basic Idea =
    - 2 indexes
    - which move in relation or independently of each other
    - Each index operates on O(N) positions,
    - Thus, total increment/decrement operations = O(N)
- Advantages =
    - Reduces no of iterations in search O(n^2) -> O(n)
    - Reduces complexity of algorithm

# Ways to move the two pointers

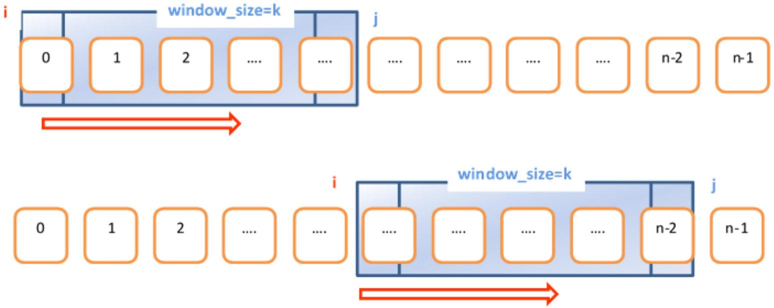
## BOTH POINTERS MOVE IN SAME DIRECTION WITH DIFFERENT SPEEDS (SLOW/FAST)



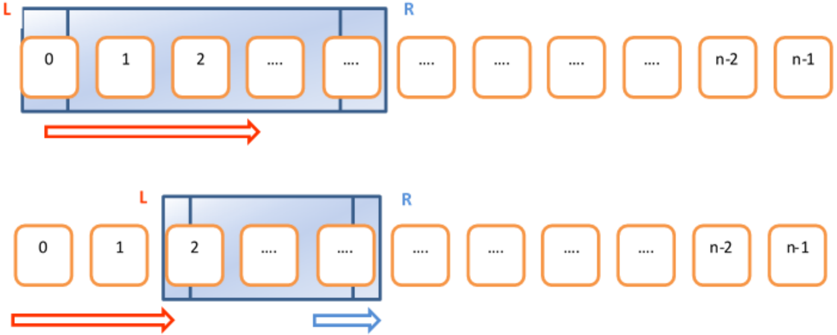
## POINTERS MOVE TOWARDS EACH OTHER FROM OPPOSITE ENDS



## FIXED WIDTH SLIDING WINDOW



## VARIABLE WIDTH SLIDING WINDOW

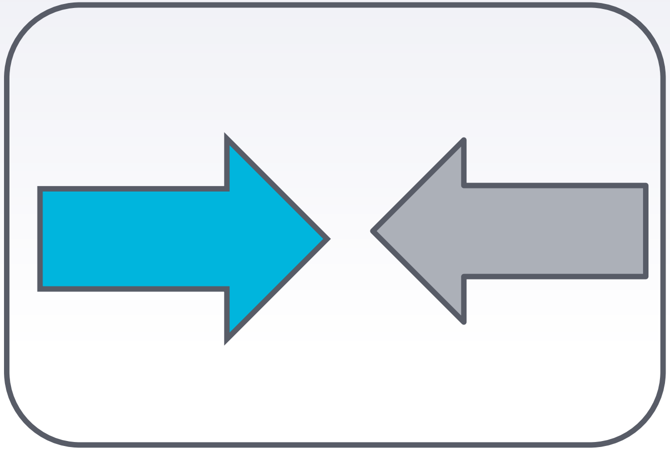


# Explaining each type with an example

# Pointers moving from opposite ends

1

Pointers moving to each other from opposite sides



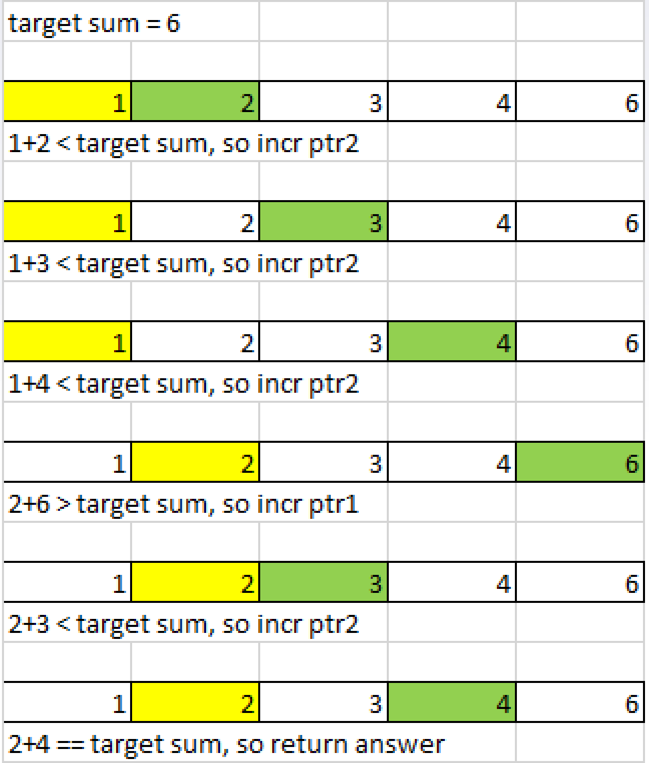
# Sum of 2 numbers problem

- Given:
  - A “sorted” array with N integers
  - Target Sum = X
  - For eg: array contains about 10^5 integers having values about 10^9
- To Find :
  - A “pair” such that arr[i1,j1] + arr[i2,j2] = X
- Possible Approaches:
  - Nested loops iteration and compare sum: O(n^2)
  - Binary search (X-V) for each array value : O(N log N)
  - Two pointer technique : O(N)

# Sum of 2 numbers

## Brute Force Method : O(n^2)

```
def pairExistsBrute( arr,  n,  S):
   for i in range (0, n-2):
      for j in range(i+1 , n-1):
         if(arr[i] + arr[j] == S):
            return (i,j)
   return (-1,-1)


arr = [1,2,3,4,6]
i,j = pairExistsBrute(arr, len(arr), 6)
print(arr[i], arr[j])
```



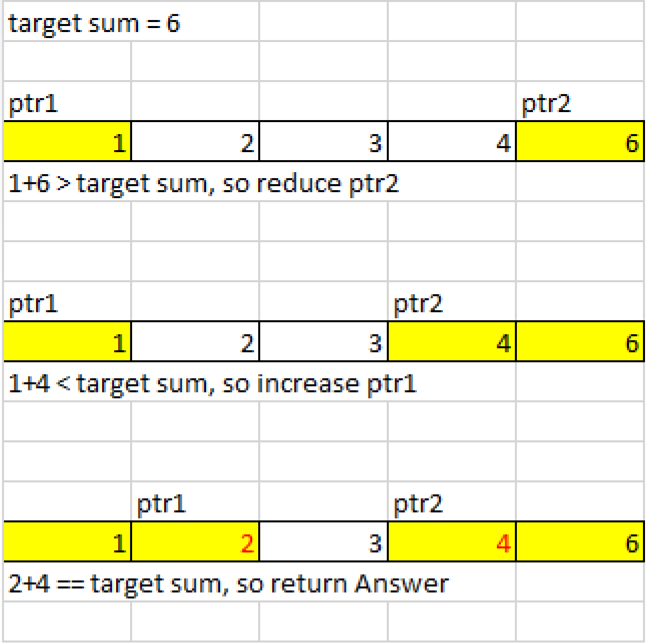
| | | | | |
|---|---|---|---|---|
| target sum = 6 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 1+2 < target sum, so incr ptr2 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 1+3 < target sum, so incr ptr2 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 1+4 < target sum, so incr ptr2 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 2+6 > target sum, so incr ptr1 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 2+3 < target sum, so incr ptr2 | | | | |
| 1 | 2 | 3 | 4 | 6 |
| 2+4 == target sum, so return answer | | | | |

# Sum of 2 numbers

## Two Pointer approach : O(N)

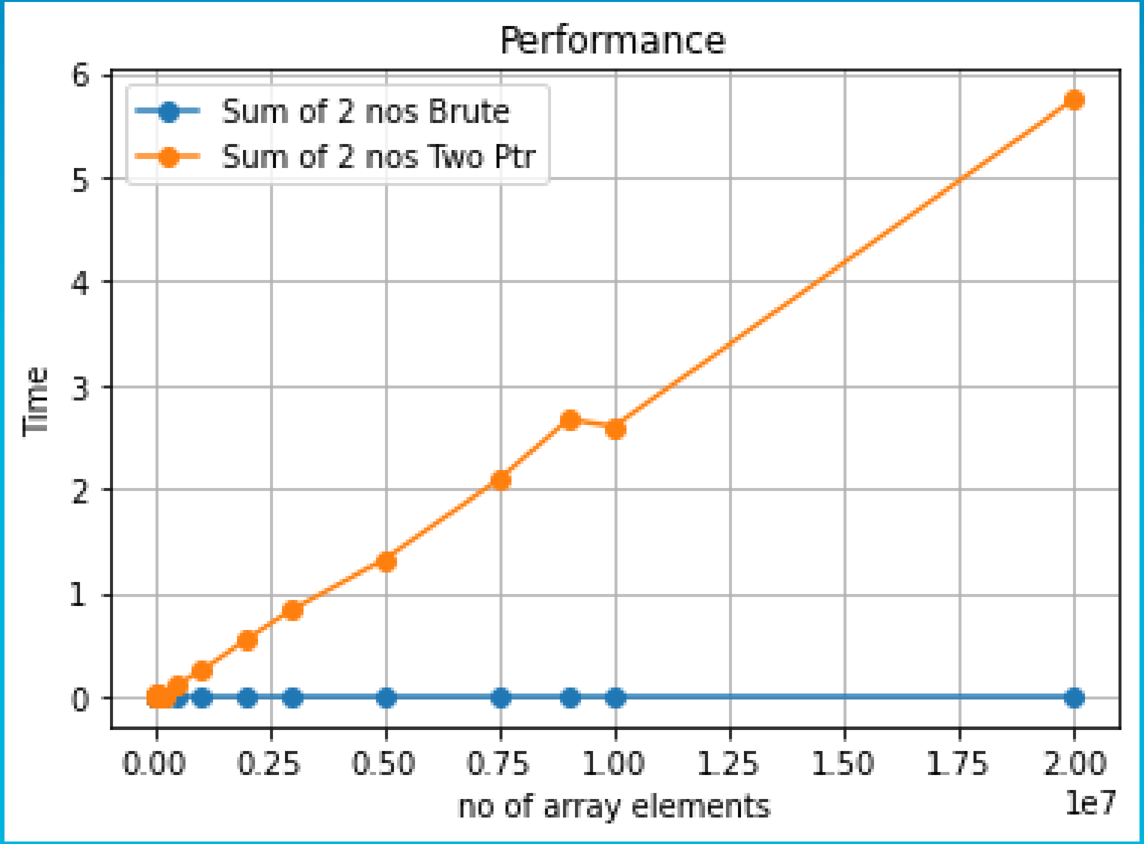
- Let p1 = index to first element of the array
- Let p2 = index to last element of array
- Let Y = arr[ p1] + arr[ p2 ]
- If Y>= X ➔ shift p2 to left ➔ decrease p2
- If Y < X ➔ shift p1 to left ➔ increase p1
- Repeat till Y == X or no way
- Each pointer moves O(N) ➔ total = O(N)

```
def pairExists(arr, n, S):
  i = 0
  j = n-1
  while( i < j):
    Y = arr[ i ] + arr[ j ]
    if ( Y == S):
      return (i,j)
    elif ( Y < S ):
      i = i + 1
    elif ( Y > S ):
      j = j - 1
  return (-1,-1)


arr = [ 1,2,3,4,6 ]
i,j = pairExists(arr, len(arr), 6)
print(arr[ i ], arr[ j ])
```



| | | | | |
|---|---|---|---|---|
| target sum = 6 | | | | |
| | | | | |
| ptr1 | | | | ptr2 |
| 1 | 2 | 3 | 4 | 6 |
| 1+6 > target sum, so reduce ptr2 | | | | |
| | | | | |
| | | | | |
| ptr1 | | | ptr2 | |
| 1 | 2 | 3 | 4 | 6 |
| 1+4 < target sum, so increase ptr1 | | | | |
| | | | | |
| | | | | |
| | ptr1 | | ptr2 | |
| 1 | 2 | 3 | 4 | 6 |
| 2+4 == target sum, so return Answer | | | | |
| | | | | |

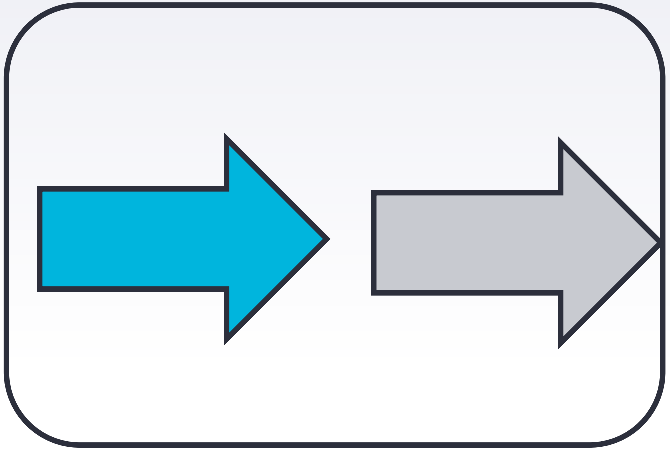
# Sum of 2 numbers problem : Performance



## Pointers moving from same end

2

Pointers moving in same direction, one moves faster, the other slower



# Middle of a linked list problem

- Given :
  - A linked list of size N
- To Find:
  - Data value at the middle node of the list and print it
- Possible Approaches:
  - Find the length of the entire linked list and then iterate till half-length again: O(n^2)
  - Two pointer technique traverse linked list by moving one pointer by one and the other pointer by two: O(N)

## Creating a Linked List in Python:

```
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None


 class LinkedList:
   def __init__(self):
    self.head = None
   def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node


list1 = LinkedList()
list1.push(5)
list1.push(4)
list1.push(2)
list1.push(3)
list1.push(1)
```

# Middle of a linked list

## Brute Force Method : O(n^2)

```
def printMiddleBrute(list1):
    node_count = 0
    temp = list1.head
    while temp:
        node_count = node_count + 1
        temp = temp.next
    temp = list1.head
    while i < node_count/2:
        temp = temp.next
    return temp.data


temp=printMiddle(list1)
print("The middle element is: ", temp)
```

1. count no of nodes = n

2. find mid = no of nodes/2

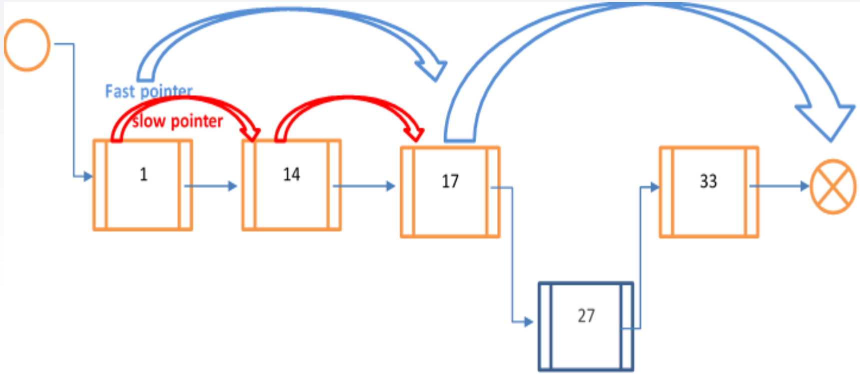
3. find mid of node =

# Middle of a linked list

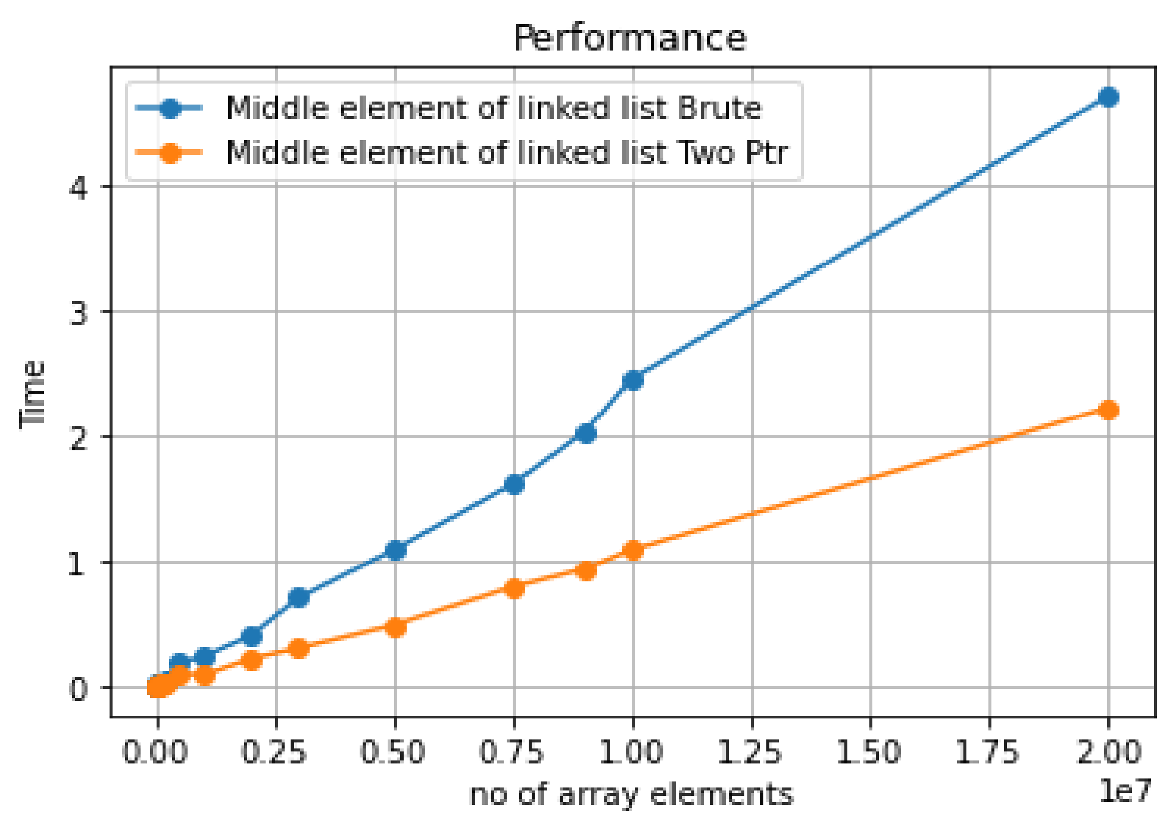
## Two pointer approach : O(N)

- Let slow, fast = head
- shift slow ➔ slow.next
- shift fast ➔fast.next.next
- Repeat till end of list or no way
- Each pointer moves O(N) ➔ total = O(N)

```
def printMiddle(list1):
   slow_ptr = list1.head
   fast_ptr = list1.head

   if list1.head is not None:
      while (fast_ptr is not None and
fast_ptr.next is not None):
         fast_ptr = fast_ptr.next.next
         slow_ptr = slow_ptr.next
   return slow_ptr.data

slow=printMiddle(list1)
print("The middle element is: ", slow)
```



# Middle of a linked list : Performance



3

# Sliding Windows

Fixed window length k

# Max sum of k consecutive numbers problem

- Given:
  - An array of size N
  - Window = k consecutive numbers

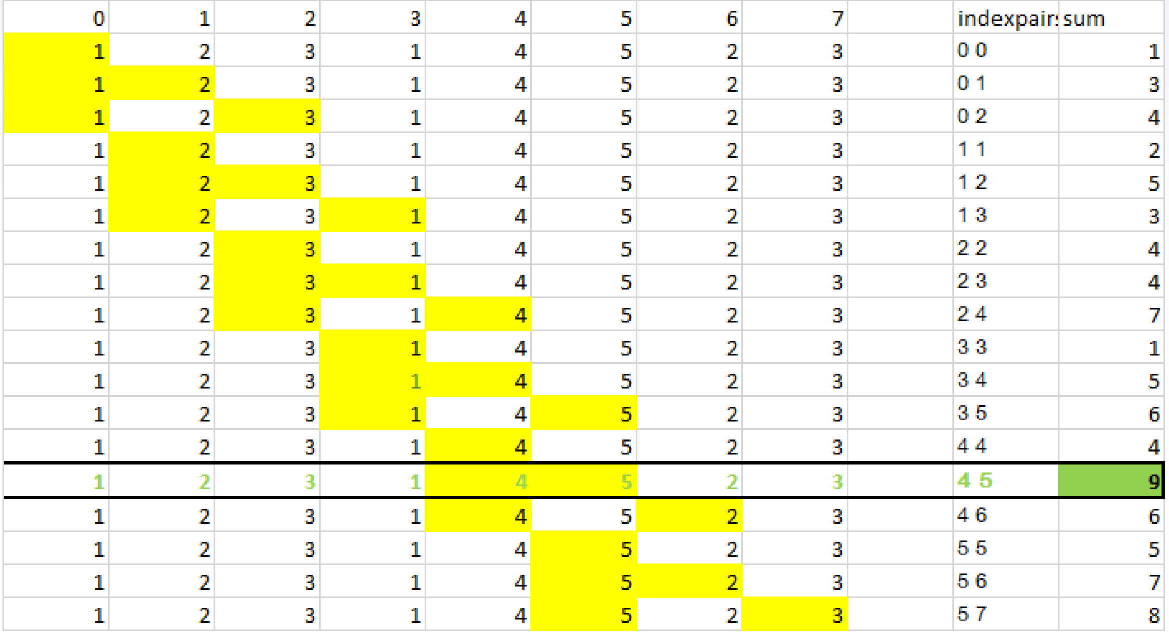
- To Find:
  - Max sum of k consecutive numbers

- Possible Approaches:
  - Nested loops – generate all pairs(i,j):i<=j and find sum between them: O(N^3)
  - Nested loops - brute force start at each index and compare sum of k values: O(N^2)
  - Two pointer technique : O(N)
    - Use of list, queue or deque to maintain window state
    - Use variables to maintain window state

# Max sum of k consecutive numbers

## Brute Force Method : O(n^2)

```
def findMaxSumBrute(arr, n, k):
  bestSum = -99999999
  for i in range (0, n):
    S = 0
    if i+k <= n:
      for j in range(i,i+k) :
        S = S + arr[j];
      bestSum = max(S, bestSum)
  return bestSum


arr = [1,2,3,1,4,5,2,3]
win_sz=3
S = findMaxSumBrute(arr, len(arr),win_sz)
print(S)
```



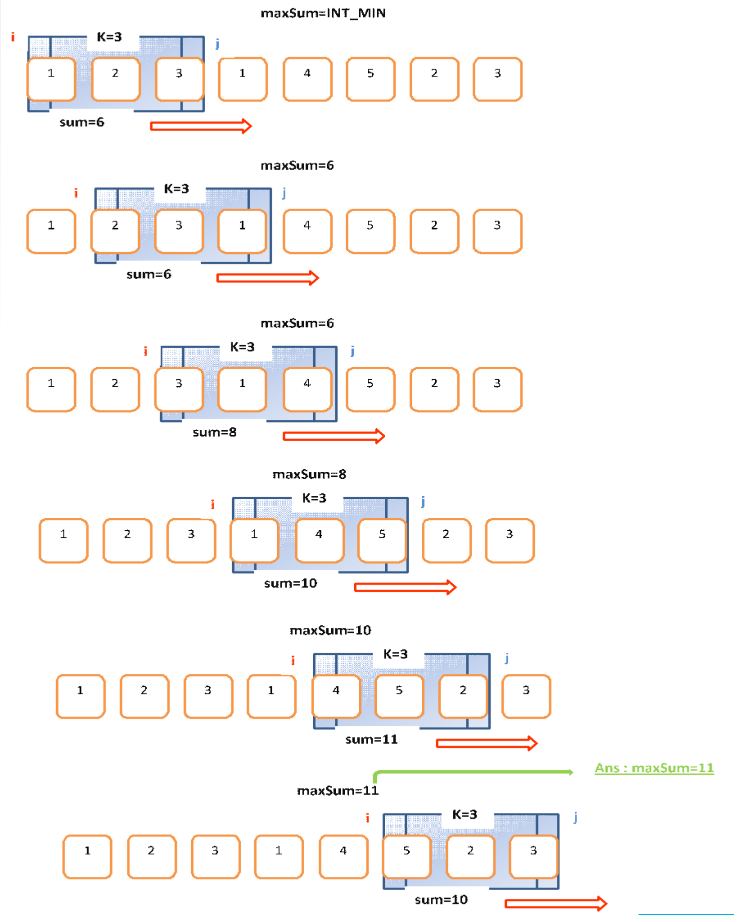
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | indexpair: | sum |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 0 0 | 1 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 0 1 | 3 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 0 2 | 4 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 1 1 | 2 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 1 2 | 5 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 1 3 | 3 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 2 2 | 4 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 2 3 | 4 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 2 4 | 7 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 3 3 | 1 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 3 4 | 5 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 3 5 | 6 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 4 4 | 4 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 4 5 | 9 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 4 6 | 6 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 5 5 | 5 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 5 6 | 7 |
| 1 | 2 | 3 | 1 | 4 | 5 | 2 | 3 | | 5 7 | 8 |

# Max sum of k consecutive numbers

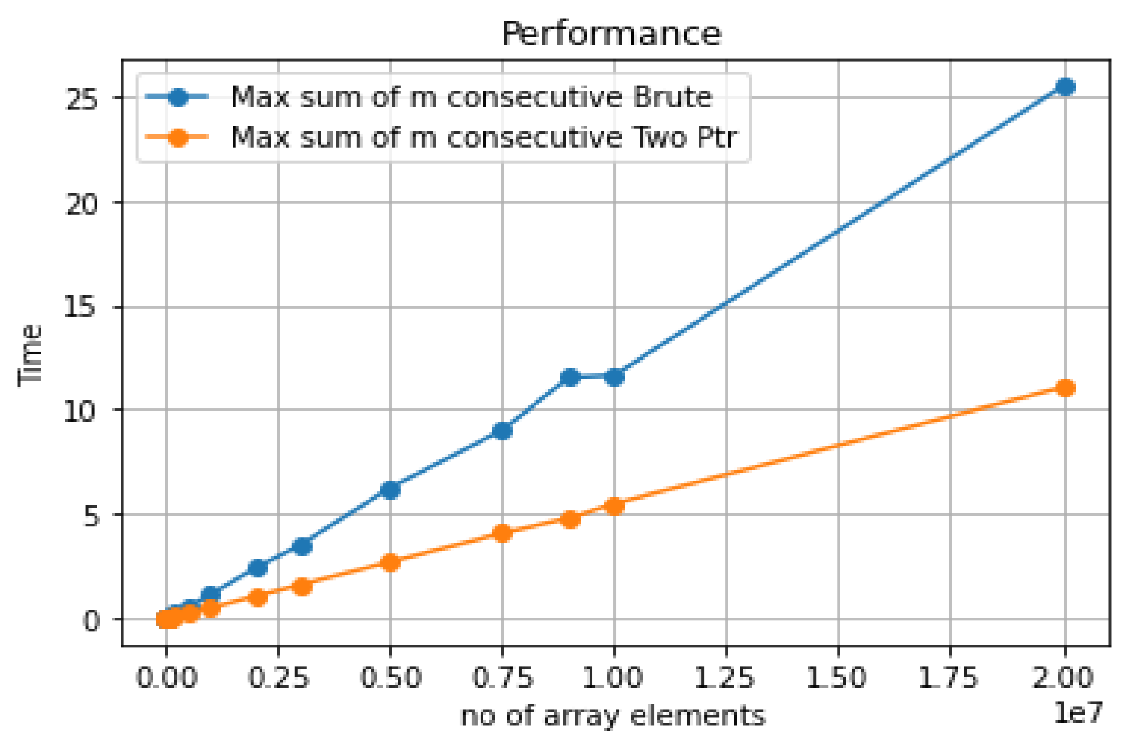
## Two pointer approach : O(N)

- Find first window with size k
- sum ➔ sum of all the elements within the current window
- bestSum ➔ max sum among all the windows
- Move the window one step at a time from left to right:
  - 1. L = L+1, R=R+1
  - 2. remove the leftmost element in the current window
  - 3. add the next element of the array
- Repeat until end of the array.

```
def findMaxSum(arr,arr_sz,win_sz):
  S = 0
  bestSum = -999999999
  for i in range (0, win_sz):
    S = S + arr[i]

  for i in range (1, arr_sz):
    bestSum = max(bestSum, S)
    if(i+win_sz)> arr_sz-1:
      break
    S = S - arr[i]
    S = S + arr[i+win_sz]
  return bestSum

arr = [1,2,3,1,4,5,2,3]
S = findMaxSum(arr, len(arr), 3)
print(S)
```



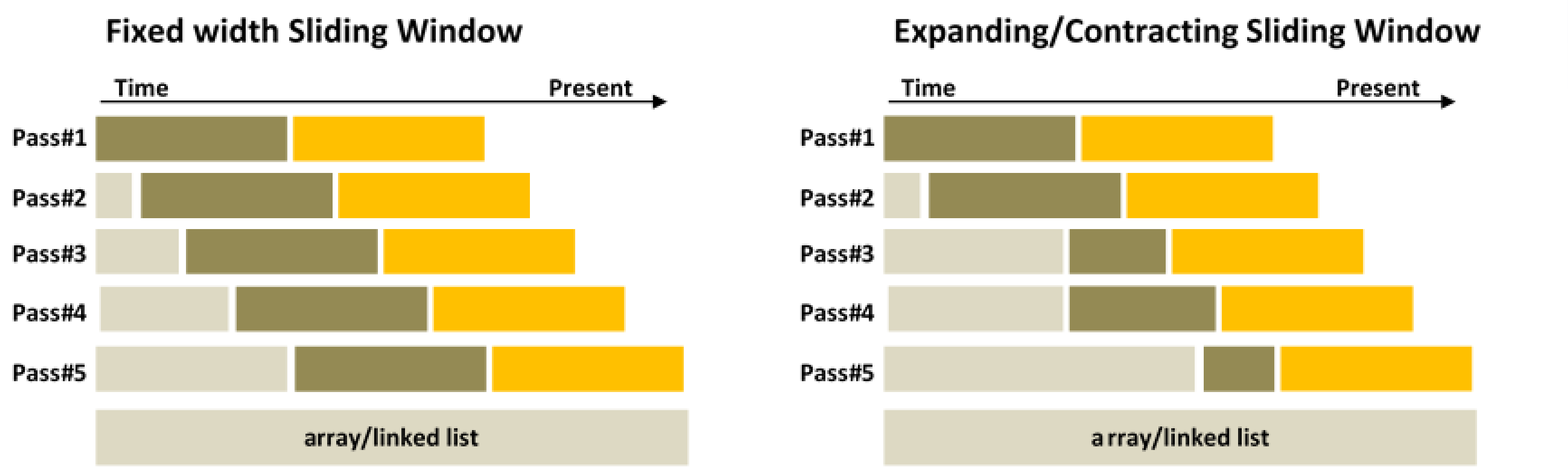
# Max sum of k consecutive numbers : Performance



# Sliding Windows

window length is not fixed

4

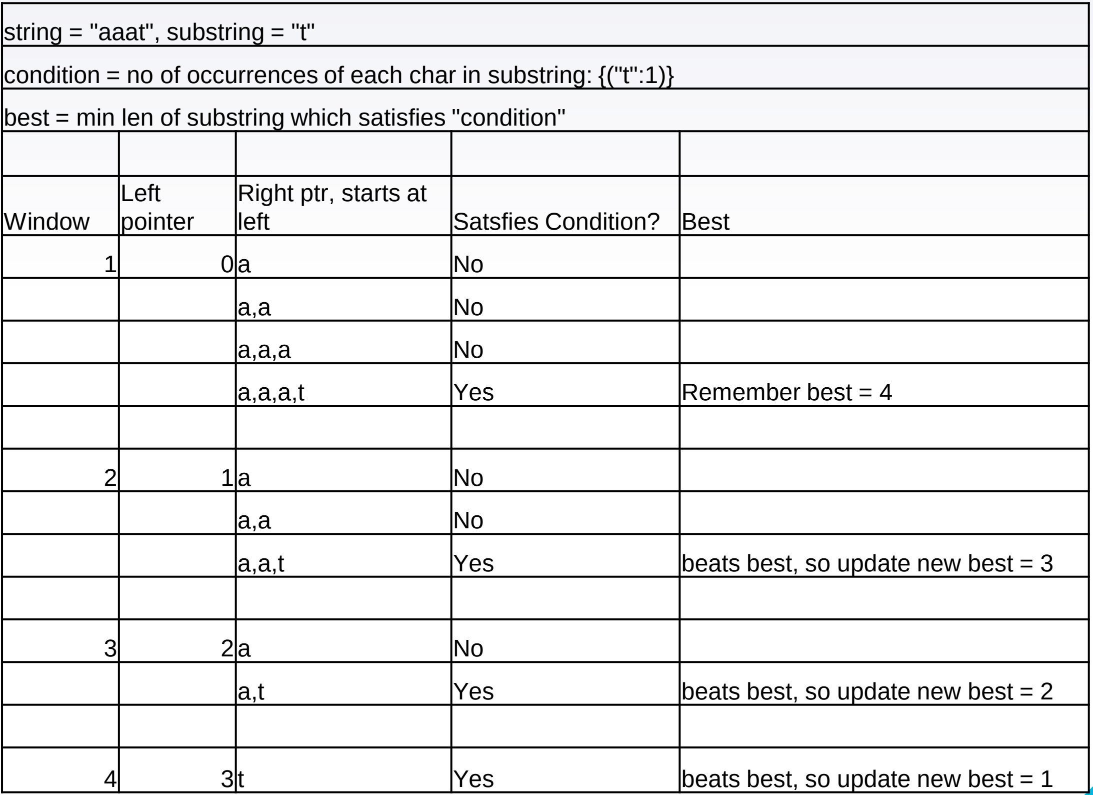


# Min length of a substring problem

- Given:
  - String of n characters
  - Substring of k characters

- To Find:
  - Min length of a substring such that it contains all the distinct characters from a pattern string
  - Order of characters is not important

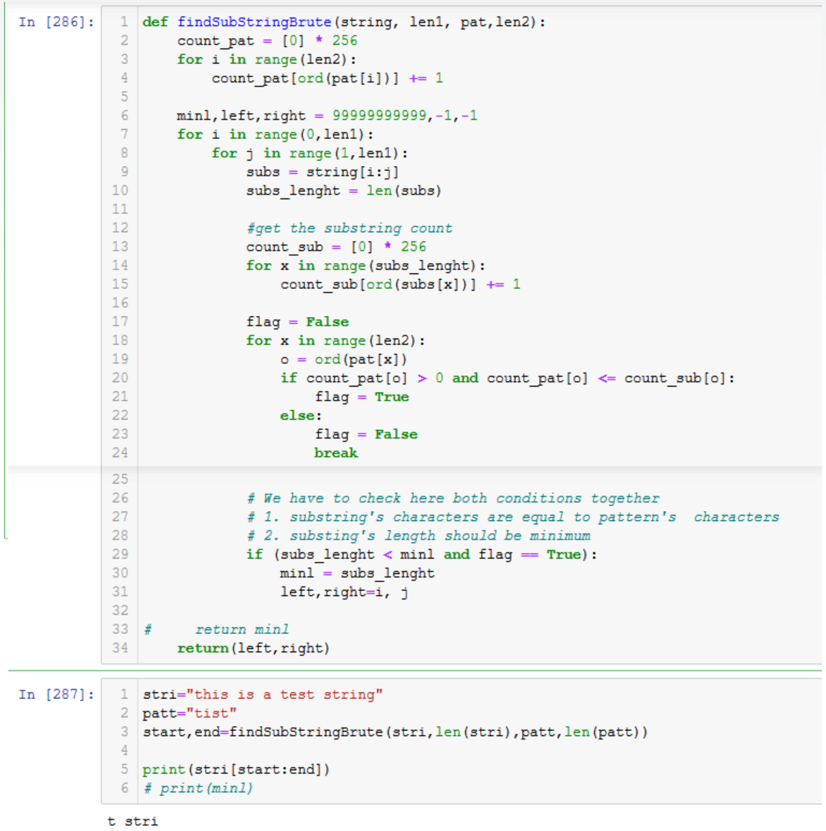
- Possible Approaches:
  - **Generate all substrings of string and print the smallest substring containing all characters of substring: O(N^3)**
  - **Two pointer technique : O(N)**
    - **At each step choose to either expand the window or contract the window until it satisfies the given conditions.**
    - **Use of hashtable to maintain window state**



| | | | | |
|---|---|---|---|---|
| string = "aaat", substring = "t" | | | | |
| condition = no of occurrences of each char in substring: {("t":1)} | | | | |
| best = min len of substring which satisfies "condition" | | | | |
| | | | | |
| Window | Left pointer | Right ptr, starts at left | Satsfies Condition? | Best |
| 1 | 0 | a | No | |
| | | a,a | No | |
| | | a,a,a | No | |
| | | a,a,a,t | Yes | Remember best = 4 |
| | | | | |
| 2 | 1 | a | No | |
| | | a,a | No | |
| | | a,a,t | Yes | beats best, so update new best = 3 |
| | | | | |
| 3 | 2 | a | No | |
| | | a,t | Yes | beats best, so update new best = 2 |
| | | | | |
| 4 | 3 | t | Yes | beats best, so update new best = 1 |

# Min length of a substring

## Brute Force Method : O(n^3)



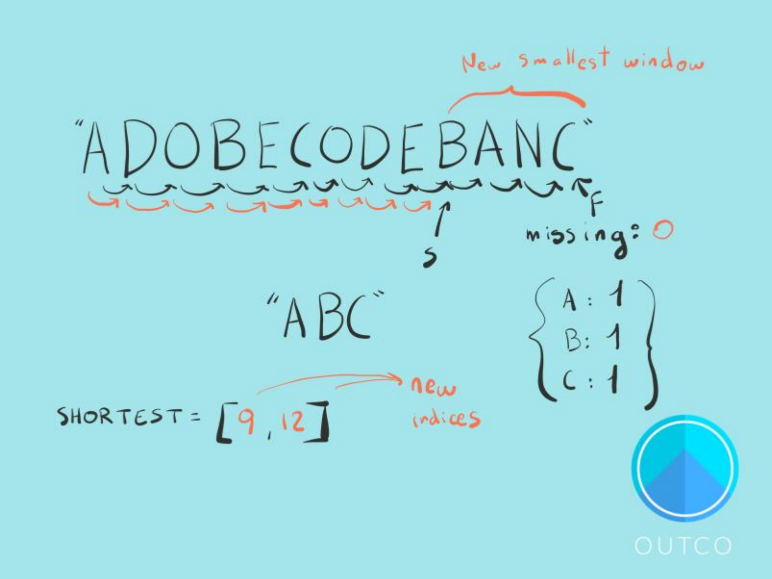
In [286]:

```
def findSubStringBrute(string, len1, pat,len2):
    count_pat = [0] * 256
    for i in range(len2):
        count_pat[ord(pat[i])] += 1

    minl,left,right = 99999999999,-1,-1
    for i in range(0,len1):
        for j in range(1,len1):
            subs = string[i:j]
            subs_lenght = len(subs)

            #get the substring count
            count_sub = [0] * 256
            for x in range(subs_lenght):
                count_sub[ord(subs[x])] += 1

            flag = False
            for x in range(len2):
                o = ord(pat[x])
                if count_pat[o] > 0 and count_pat[o] <= count_sub[o]:
                    flag = True
                else:
                    flag = False
                    break


            # We have to check here both conditions together
            # 1. substring's characters are equal to pattern's  characters
            # 2. substing's length should be minimum
            if (subs_lenght < minl and flag == True):
                minl = subs_lenght
                left,right=i, j

#     return minl
    return(left,right)
```

In [287]:

```
stri="this is a test string"
patt="tist"
start,end=findSubStringBrute(stri,len(stri),patt,len(patt))

print(stri[start:end])
# print(minl)
```

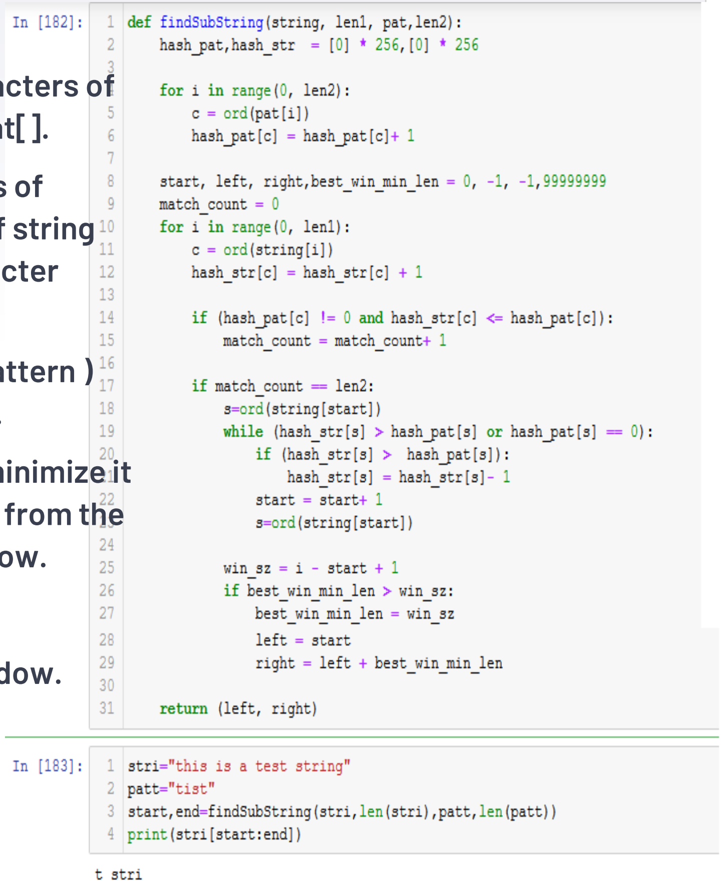
```
t stri
```



# Min length of a substring

## Two pointer approach

- Store the occurrence of characters of the given pattern in a hash_pat[ ].
- Start matching the characters of pattern with the characters of string i.e. increment count if a character matches.
- Check if (count == length of pattern ) this means a window is found.
- If such window found, try to minimize it by removing extra characters from the beginning of the current window.
- Update min_length.
- Print the minimum length window.



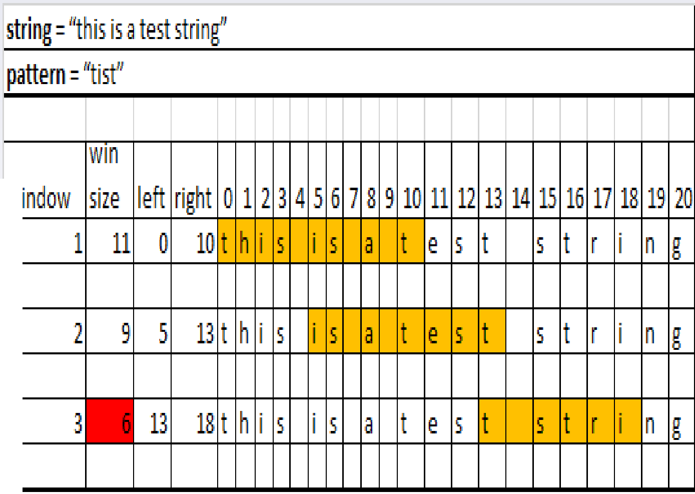
In [182]:

```python
def findSubString(string, len1, pat,len2):
    hash_pat,hash_str  = [0] * 256,[0] * 256

    for i in range(0, len2):
        c = ord(pat[i])
        hash_pat[c] = hash_pat[c]+ 1

    start, left, right,best_win_min_len = 0, -1, -1,99999999
    match_count = 0
    for i in range(0, len1):
        c = ord(string[i])
        hash_str[c] = hash_str[c] + 1

        if (hash_pat[c] != 0 and hash_str[c] <= hash_pat[c]):
            match_count = match_count+ 1

        if match_count == len2:
            s=ord(string[start])
            while (hash_str[s] > hash_pat[s] or hash_pat[s] == 0):
                if (hash_str[s] >  hash_pat[s]):
                    hash_str[s] = hash_str[s]- 1
                start = start+ 1
                s=ord(string[start])

            win_sz = i - start + 1
            if best_win_min_len > win_sz:
                best_win_min_len = win_sz
                left = start
                right = left + best_win_min_len

    return (left, right)
```

In [183]:

```python
stri="this is a test string"
patt="tist"
start,end=findSubString(stri,len(stri),patt,len(patt))
print(stri[start:end])
```

```
t stri
```



string = "this is a test string"

pattern = "tist"

| indow | win size | left | right | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 11 | 0 | 10 | t | h | i | s |  | i | s |  | a |  | t | e | s | t |  | s | t | r | i | n | g |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | 9 | 5 | 13 | t | h | i | s |  | i | s |  | a |  | t | e | s | t |  | s | t | r | i | n | g |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | 6 | 13 | 18 | t | h | i | s |  | i | s |  | a |  | t | e | s | t |  | s | t | r | i | n | g |

# Some more two pointer problems

## Sorting

- Pair With Given Difference
- 3 Sum
- 3 Sum Zero
- Counting Triangles
- Diffk

## Multiple Arrays

Merge Two sorted lists

Intersection of sorted arrays

## Inplace updates

Remove duplicates from sorted array

Remove element from an array

Sort by color

## SubArrays

Counting subarrays

Subarrays with distinct integers

## Others

Max 1s after modification

Max continuous series of 1s

Array 3 pointers

Container with most water

# Wrap-up

Brute Force Problems much efficiently solved by Two pointer techniques

Applications of this technique

Four ways in  which the two pointers move:

- Move towards each other,
- fast/slow pointers,
- fixed/variable width sliding windows

Code and performance of the Brute Force approach Vs. Two pointer approach for each of the above examples

Some more problems which can be solved by this technique

# THANKS!