Optimization of $O(n^2)$ search: Two Pointers Technic & Sliding Window

Hamza BA-MOHAMMED

ENSIAS IT CLUB - Competitive Programming Cell

Wednesday 23 February 2022

Summary

What is a pointer ?

Definition

Generaly: A variable containing the memory adress of another variable.
Here: A numerical variable p containing the index of another variable S in a data structure D. In other words, $D[p] == S$.
We say that p points to S, even though it's more of an iterator.

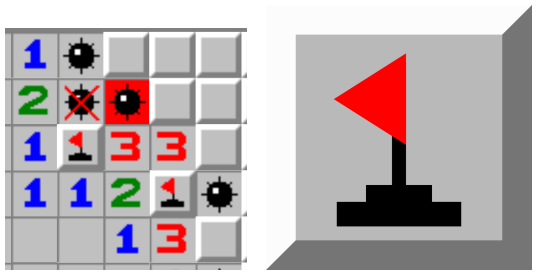


Figure: A pointer can be imagined like a red flag in the minesweeper game: it's a tool that helps keeping track of a special position.

Reversing a string

Problem A

Given a string S such that $1 < |S| < 1000$, write a code that returns its reversal.

Reversing a string

1st approch

1. define an empty string T
2. add characters of S in T starting from its end
3. return T

2nd approch (Two pointers)

1. define 2 pointers i $= 0$ and j $= |S| - 1$
2. while $i < j$, swap S_i and S_j then increase i and decrease j
3. return S

Reversing a string

Example : S = "ABCDEFG"

G	B	C	D	E	F	A
⇑ i						⇑ j

Table: iteration 1

G	F	C	D	E	B	A
	⇑ i				⇑ j	

Table: iteration 2

Reversing a string

```
def inverse(T):
    S = list(T)
    i = 0
    j = len(S) - 1
    while i < j:
        S[i],S[j] = S[j],S[i]
        i += 1
        j -= 1
    return X

s = input()
x = reverse(s)
for e in x:
    print(z, end="")
```

Removing duplicates

Problem B

Given a sorted list A of $1 < n < 10^9$ sorted integers, write a code that returns the same list without duplicates.

Removing duplicates

Brute force

1. create an empty list M
2. in 2 neested loops, for each element x in A, check if $x \in M$
   - if $x \in M$, continue
   - else, append x to M and continue
3. return M

time complexity : $O(n^2)$

Removing duplicates

Two pointers technic

1. create an empty list M and initialize 2 pointers p $= 0$ & q $= 1$
2. in 1 loop, while $q < n$
   - if A[p] != A[q]
     1. append $A[p]$ to M
     2. p $=$ q
   - q++
3. append $A[q-1]$ (or $A[p]$) to M

time complexity: $O(n)$

Removing duplicates

0	1	2	3	3	3	4	5	5	6
	⇑ p	⇑ q							

Table: A, (end of) iteration 1

0

Table: M, iteration 1

Removing duplicates

0	1	2	3	3	3	4	5	5	6
		⇑ p	⇑ q						

Table: A, iteration 2

0	1

Table: M, iteration 2

Removing duplicates

0	1	2	3	3	3	4	5	5	6
			⇑ p	⇑ q					

Table: A, iteration 3

0	1	2

Table: M, iteration 3

Removing duplicates

0	1	2	3	3	3	4	5	5	6
			⇑		⇑				
			p		q				

Table: A, iteration 4

0	1	2	3	3	3	4	5	5	6
			⇑			⇑			
			p			q			

Table: A, iteration 5

Removing duplicates

0	1	2	3	3	3	4	5	5	6
						⇑ p	⇑ q		

Table: A, iteration 6

0	1	2	3

Table: M, iteration 6

Removing duplicates

0	1	2	3	3	3	4	5	5	6	IOR
									⇑ p	⇑ $q > 9$

Table: A, last iteration

0	1	2	3	4	5

Table: M, last iteration

Removing duplicates

```
def remove_duplicates(A):
    p = 0
    q = 1
    n = len(A)
    M =[]
    while q < n:
        if A[p] != A[q]:
            M.append(A[p])
            p = q
        q += 1
    M.append(A[q - 1])
    return M

s = list(map(int,input().split()))
t = remove_duplicates(s)
for e in t: print(e, end=" ")
```

Targeted sum

Problem C

Given an array A of n sorted integers and an integer $K > 1$, find out a pair of integers which sum equals K. If there isn't such a pair in A, return -1.

Targeted sum

Brute force

Verify if $A[i] + A[j] = K$, $(0 \leq i < j \leq n-1) \Rightarrow \frac{n(n-1)}{2}$ pairs to test. Stop when there is a satisfying pair. If all the pairs doesn't satisfy the condition, return -1.
time complexity: $O(n^2)$

Targeted sum

Two pointers technic

1. initialize 2 pointers i = 0 and j = n-1
2. in 1 loop, while $i != j$, evaluate $A[i] + A[j]$
   - if $A[i] + A[j] = K$, return $(A[i], A[j])$
   - if $A[i] + A[j] > K$, $j--$ (adding smaller number)
   - if $A[i] + A[j] < K$, i++ (adding bigger number)
3. return -1 (since no pair is satisfying)

time complexity: $O(n)$

Targeted sum

Example: K = 9

1	2	4	7	11	12
⇑					⇑
p					q

Table: *i + *j = 13 > 9

1	2	4	7	11	12
⇑				⇑	
p				q	

Table: *i + *j = 12 > 9

Targeted sum

Example: K = 9

1	2	4	7	11	12
⇑			⇑		
p			q		

Table: *i + *j = 8 < 9

1	2	4	7	11	12
	⇑		⇑		
	p		q		

Table: *i + *j = 9

Targeted sum

```
def findSum(A, k):
    i = 0
    n = len(A)
    j = n - 1
    while i != j:
        if A[i] + A[j] == k : return (A[i],A[j])
        elif A[i] + A[j] > k : j -= 1
        else : i += 1
    return -1

k = int(input())
A = list(map(int,input().split()))
print(findSum(A,k))
```

Important notice

If the order of the data isn't important in the problem, we can first sort it in $O(nlog(n))$ then apply the two pointers technic in $O(n)$, which is $O(nlog(n))$ overall. This is still better than $O(n^2)$.
Examples : problems B and C if the order wasn't given.

Maximizing an area

Problem D

Consider an array A of n integers representing the heights of n vertical sticks, in the same arrangement as in the array. Assuming that the distance between each stick is one unit, what is the maximum amount of water one could hold between two sticks (disregarding any sticks in between)?

Maximizing an area

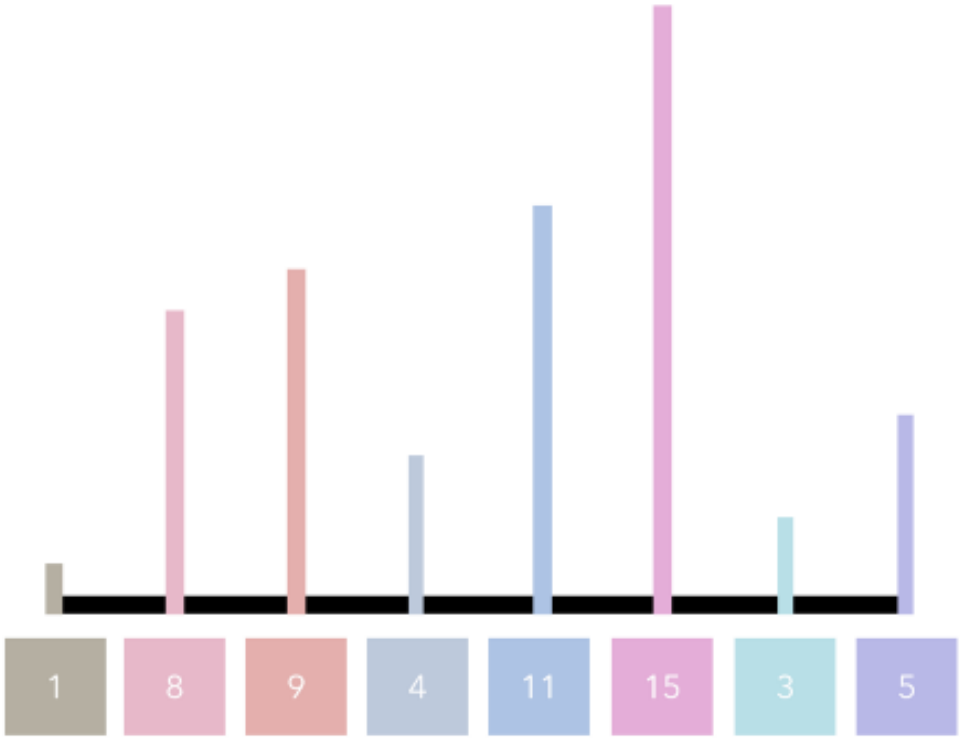


Figure: Illustration

Maximizing an area

Brute force

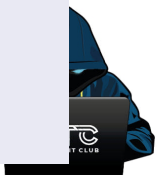
Calculate the legal area between each 2 sticks $\Rightarrow \frac{n(n-1)}{2}$ area to evaluate, while keeping track of the maximum.
time complexity: $O(n^2)$

Maximizing an area

Two pointers technic

1. initialize 2 pointers i=0 and j=n-1
2. initialize a variable max = 0
3. while $i < j$:
   1. evaluate the area S between sticks A_i and A_j
   2. if $S > max$, update max
   3. if $A_i \geq A_j$, $j--$
   4. else, i++
4. return max

time complexity: $O(n)$



Maximizing an area

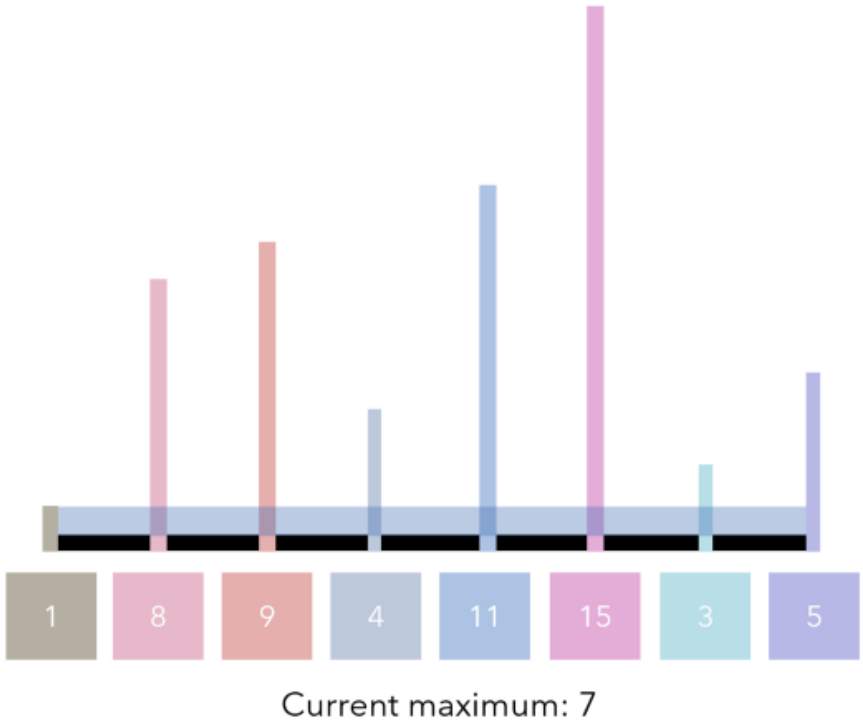


Figure: first position, all other situations can be obtained from it

Maximizing an area

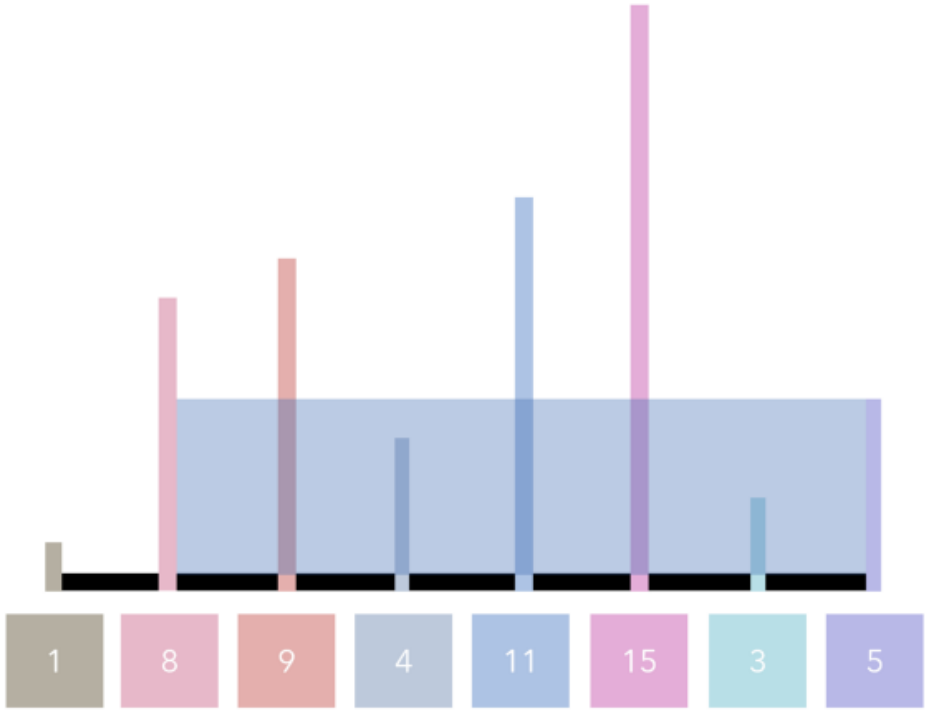
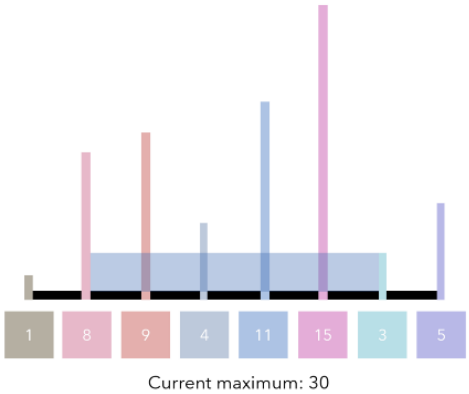


Figure: the best option is to change the shortest stick

Maximizing an area



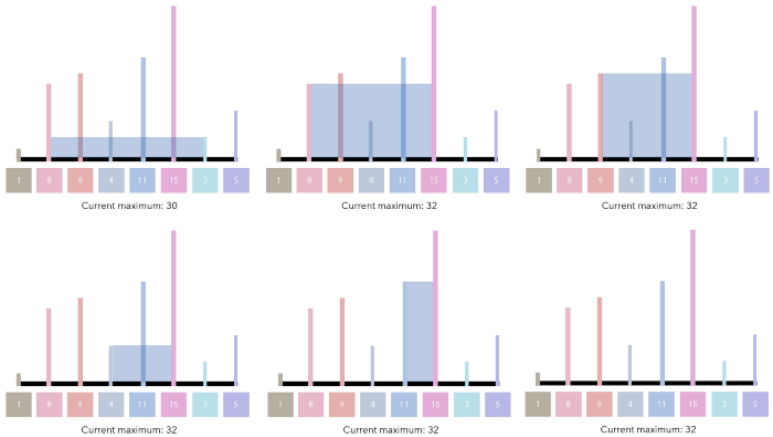


Figure: and so on..

Maximizing an area

```
def maxArea(A):
    i = 0
    n = len(A)
    j = n - 1
    mx = 0
    while i < j:
        if A[i] + A[j] > mx : mx = A[i] + A[j]
        if A[i] >= A[j]:: j -= 1
        else : i += 1
    return mx
A = list(map(int,input().split()))
print(maxArea(A))
```

Largest k-subsequence

Problem E

Given an array A of $n \leq 10^9$ non-sorted integers and an integer $k < n$, return the largest sum of k consecutive integers.

Largest k-subsequence

Brute force

in 2 nested loops, evaluate the sum of each k consecutive elements
$\Rightarrow (n-k)n$ operation to perform.
time complexity: $O(n^2)$

Largest k-subsequence

Sliding window algorithm

1. evaluate $\sum_{i=0}^{k-1} A_i$ = current
2. initialize max = current
3. initialize 2 pointers i=0 and j=k
4. in 1 loop, while $j < n$
   - update current = current + A_j - A_i
   - if current > max, update max = current
   - j++ and i++
5. return max

Largest k-subsequence

Figure: a real sliding window *(thanks to Mohamed Nasser!)*

Largest k-subsequence

Example : k=3

11	22	14	17	11	19	15	13
⇑ i			⇑ j				

Table: window A[0:2], current = 47, max = 47

11	22	14	17	11	19	15	13
	⇑ i			⇑ j			

Table: window A[1:3], current = 53, max = 53

Largest k-subsequence

Example : k=3

11	22	14	17	11	19	15	13
		⇑ i			⇑ j		

Table: window A[2:4], current = 42, max = 53

11	22	14	17	11	19	15	13
			⇑ i			⇑ j	

Table: window A[3:5], current = 47, max = 53

Largest k-subsequence

```
def maxSubarray(A, k):
        n = len(A)
        s = sum(A[0:k])
        i = 0
        j = k
        mx = s
        while j < n :
                s -= A[i]
                s += A[j]
                if s > mx : mx = s
                j += 1
                i += 1
        return mx

k = int(input())
A = list(map(int,input().split()))
print(maxSubarray(A,k))
```

Minimum subsequence

Problem F

Given a string S and a string T, with $|T| < 26$ and $|S| \leq 10^9$, find the shortest substring* of S that contains all the characters of T, otherwise return an empty string.
*a substring is a contiguous subpart of a string.

Minimum subsequence

Brute force

iterate in 2 nested loops on all the subsets of the string S from the smallest to the biggest $\Rightarrow \frac{n(n-1)}{2}$ substring to generate and test.
time complexity: $O(n^2)$

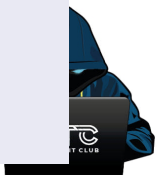
Minimum subsequence

Sliding window algorithm

to resume the idea, we can imagine it as an iterative process of 2 subprocesses :

- adding more characters from S until all the characters in T are there, and keeping track of their number of occurence
- removing characters from the start as long as we still can found the characters of T in the remaining window

and we keep track of the shortest substring satisfying the condition during this process.
time complexity: $O(n)$



Minimum subsequence

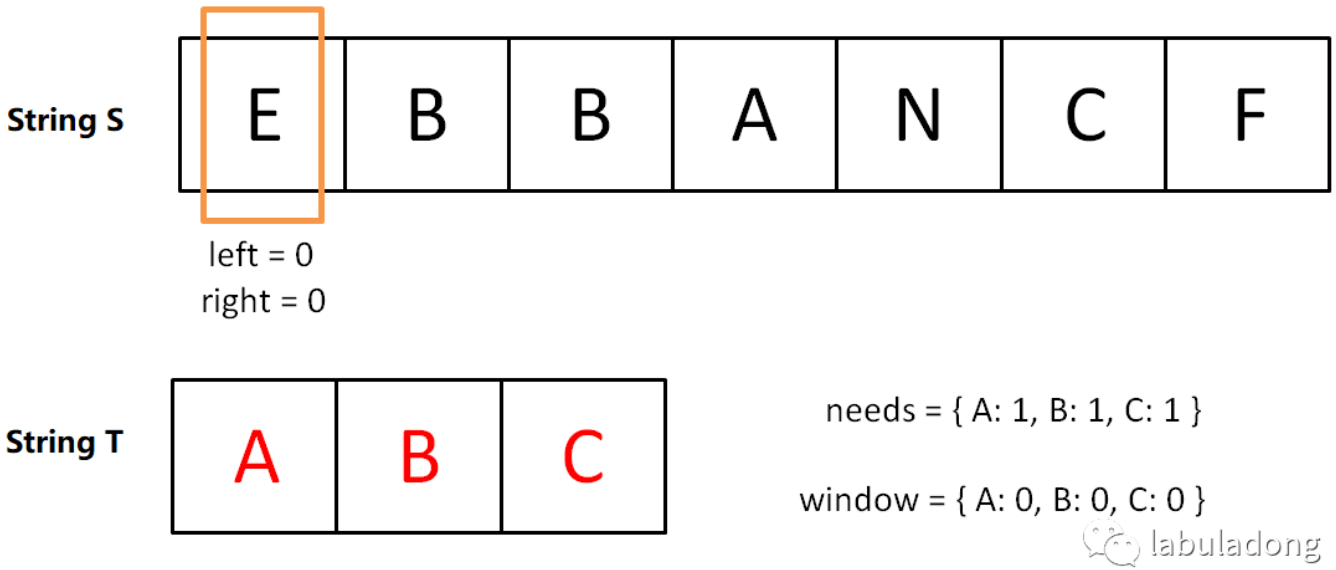


Figure: initialization

Minimum subsequence

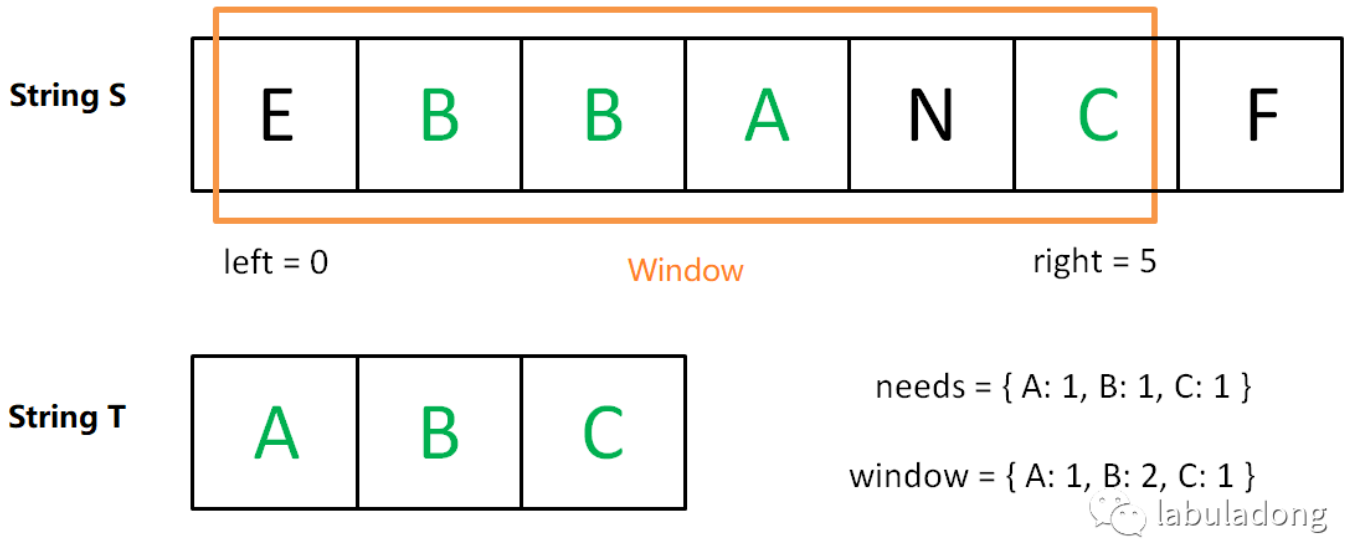


Figure: end of increasing of the right pointer. The current window is valid, not optimal.

Minimum subsequence

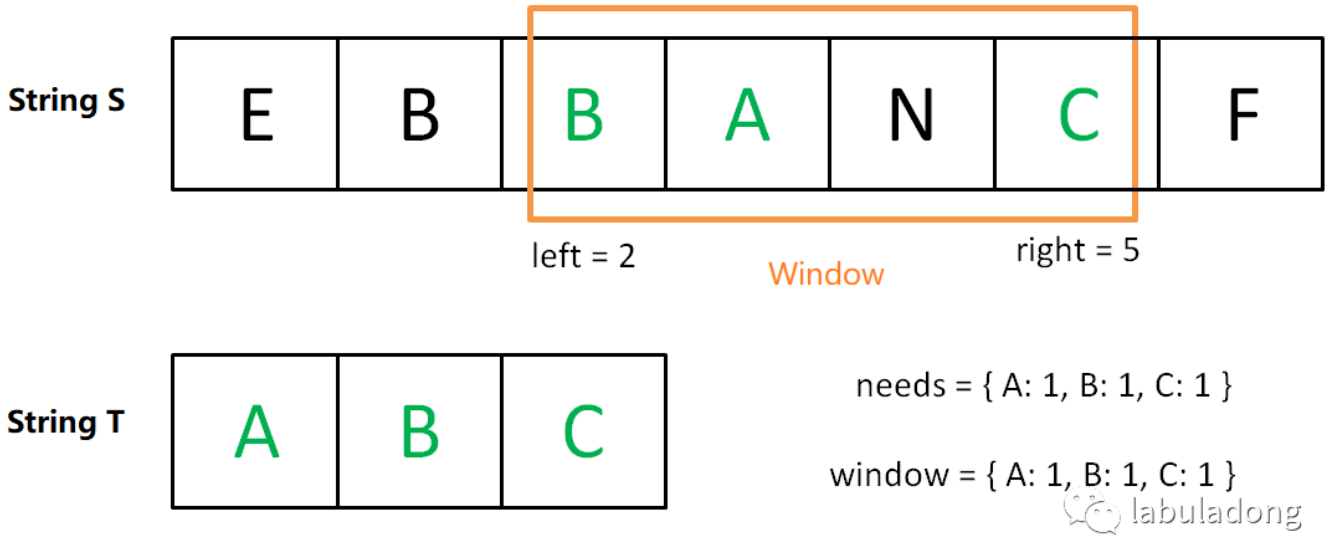


Figure: increasing of left pointer. The current window is still valid but more optimal.

Minimum subsequence

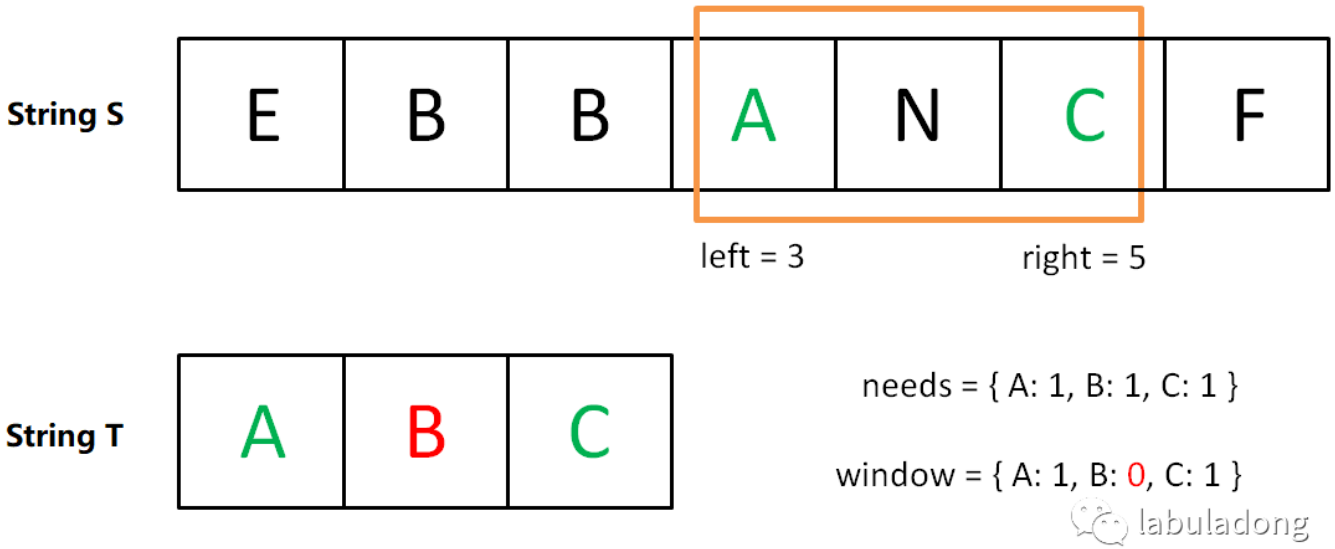


Figure: increasing of left pointer. The current window is not valid. We return to increase the right pointer. The final result is "BANC".

Minimum subsequence

1. initialize a hash table *need* (map/dictionary) with the characters of T as keys and their occurences as values
2. initialize an empty hash table *window*
3. initialize 2 pointers i=0 and j=0
4. initialize 3 variables min = n+1, match = 0 and k = len(need) (number of distinct elements of T)
5. initialize a variable start = 0

Minimum subsequence

6. while $j < n$:
   1. if need[S[j]] != 0 then
      1. window[S[j]]++
      2. if need[S[j]]==window[S[j]] then match++
   2. j++
   3. while match == k :
      1. if j - i $<$ min then update start = i and min = j - i
      2. if need[S[i]] != 0 then
         1. window[S[i]]−−
         2. if need[S[i]]>window[S[i]] then match−−
      3. i++
7. if min $>$ n then return "", else return S[start:start+min]

Minimum subsequence

```
def minWindow(S, T):
        i = 0
        j = 0
        n = len(S)
        start = 0
        minimum = len(S) + 1
        match = 0
        need = dict()
        for e in T:
                if need.get(e,"-1") == "-1" : need[e] = 0
                need[e] += 1
        k = len(need)
        window = dict()
```

Minimum subsequence

```
        while j < n :
                if need.get(S[j],"-1") != "-1" :
                        if window.get(S[j],"-1") == "-1" : window[S[j]] = 0
                        window[S[j]] += 1
                        if window[S[j]] == need[S[j]] : match += 1
                j += 1
                while match == k:
                        if j - i < minimum :
                                start = i
                                minimum = j - i
                        if need.get(S[i],"-1") != "-1" :
                                window[S[i]] -= 1
                                if window[S[i]] < need[S[i]] : match -= 1
                        i += 1
        if minimum > n : return ""
        return S[start:start+minimum]

s = input()
t = input()
print(minWindow(s,t))
```

Conslusion

The Two Pointers Technic and the Sliding Window Algorithm are two powerful tools to deal with search in different types of linear data structures and that can help optimizing it from a polynomial or exponential time to a linear or linear-logarithmic time. This could be very important when the input size passes 10^4.
The only differnce between this 2 algorithms, is that in the two pointers we only consider **the 2 elements pointed** by our pointers, but in the sliding window we consider also **the elements between them**.

Further reading

Amortyzed Analysis repository on GitHub:
`github.com/HamzaBamohammed/amortyzed-analysis`

Webography

1. "Two-pointer technique". *Leetcode*. Visited in 21/02/2022.
2. Andre Ye, "Two-Pointer Technique: Solving Array Problems at Light Speed". *Medium*. Visited in 21/02/2022.
3. labuladong, "Sliding Window Algorithm". *Github*. Visited in 21/02/2022.
4. Antti Laaksonen, Competitive Programmer's Handbook.