

# Sliding Window

---

Arnar Bjarni Arnarson

**Árangursrík forritun og lausn verkefna**

School of Computer Science

Reykjavík University

# A Sum Problem

## Problem description

Write a program that, given an integer array of size  $N$ , finds the contiguous subarray of size  $K$  with the highest sum.

## Input description

Input consist of two lines. The first line contains two space separated integers  $N$ , the size of the array, where  $1 \leq N \leq 10^6$ , and  $K$ , the size of the subarrays to consider, where  $1 \leq K \leq N$ . Then second line contains  $N$  space separated integers, the values of the array. Each value in the array is between  $-10^9$  and  $10^9$ .

## Output description

Output one line, the sum of the highest valued contiguous subarray of size  $K$ .

# A Sum Problem

Sample input	Sample output
10 4 17 20 0 1 5 24 8 2 4 1	39

# Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

# Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max(highest, total)
print(highest)
```

- This solution constructs all size  $K$  contiguous subarrays.

# Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max(highest, total)
print(highest)
```

- This solution constructs all size  $K$  contiguous subarrays.
- What is the time complexity?

# Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size  $K$  contiguous subarrays.
- What is the time complexity?
- There are  $N$  starting points, each construction takes  $K$  steps, so  $\mathcal{O}(NK)$ .

# Straightforward Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
highest = float('-inf')
for start in range(n-k+1):
    end = start + k
    total = 0
    for i in range(start, end):
        total += arr[i]
    highest = max((highest, total))
print(highest)
```

- This solution constructs all size  $K$  contiguous subarrays.
- What is the time complexity?
- There are  $N$  starting points, each construction takes  $K$  steps, so  $\mathcal{O}(NK)$ .
- Too slow!



# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
$$a_i + a_{i+1} + \cdots + a_{i+k-1}.$$
- The subarray starting at index  $i + 1$  has the sum  
$$a_{i+1} + a_{i+2} + \cdots + a_{i+k}.$$

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  
 $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  
 $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.
- What changes between starting at  $i$  vs. starting at  $i + 1$ ?

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  
 $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.
- What changes between starting at  $i$  vs. starting at  $i + 1$ ?
- We subtract  $a_i$ .

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  
 $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.
- What changes between starting at  $i$  vs. starting at  $i + 1$ ?
- We subtract  $a_i$ .
- We add  $a_{i+k}$ .

# Wasted Operations

- The subarray starting at index  $i$  has the sum  
 $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  
 $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.
- What changes between starting at  $i$  vs. starting at  $i + 1$ ?
- We subtract  $a_i$ .
- We add  $a_{i+k}$ .
- A shift from the subarray starting at  $i$  to the subarray starting at  $i + 1$  takes  $\mathcal{O}(1)$  time.

# Wasted Operations

- The subarray starting at index  $i$  has the sum  $a_i + a_{i+1} + \cdots + a_{i+k-1}$ .
- The subarray starting at index  $i + 1$  has the sum  $a_{i+1} + a_{i+2} + \cdots + a_{i+k}$ .
- We iterate over the indices  $i + 1, i + 2, \dots, i + k - 1$  twice.
- What changes between starting at  $i$  vs. starting at  $i + 1$ ?
- We subtract  $a_i$ .
- We add  $a_{i+k}$ .
- A shift from the subarray starting at  $i$  to the subarray starting at  $i + 1$  takes  $\mathcal{O}(1)$  time.
- This is known as the sliding window technique, in this case with a fixed window size.



# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?

# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size  $K$  contiguous subarray.

# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size  $K$  contiguous subarray.
- Then,  $N - K$  times, an element is removed and another added.

# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size  $K$  contiguous subarray.
- Then,  $N - K$  times, an element is removed and another added.
- Subtracting and adding numbers is constant time so  $\mathcal{O}(N)$ .

# Sliding Window Solution

```
n, k = map(int, input().split())
arr = list(map(int, input().split()))
total = 0
for i in range(k):
    total += arr[i]
highest = total
for i in range(n - k):
    total -= arr[i]
    total += arr[i+k]
    highest = max((highest, total))
print(highest)
```

- What is the time complexity?
- This solution constructs the first size  $K$  contiguous subarray.
- Then,  $N - K$  times, an element is removed and another added.
- Subtracting and adding numbers is constant time so  $\mathcal{O}(N)$ .
- Fast enough!

# A Substring Problem

## Problem description

Write a program that, given a string of size  $N$ , finds the longest substring with  $K$  distinct elements.

## Input description

Input consists of two lines. The first line contains two space-separated integers  $N$ , the size of the string, where  $1 \leq N \leq 10^6$ , and  $K$ , the number of distinct elements the substring must have, where  $1 \leq K \leq 26$ . Then the second line contains a string of length  $N$  consisting of English lowercase characters.

## Output description

Output one line, the longest substring with  $K$  distinct elements. If no such string exists, output "DOES NOT EXIST", without quotations.

# A Substring Problem

Sample input	Sample output
14 3 bacdcbcdbabdb	cdcbcdb



# General Framework

```
from string import ascii_lowercase
n, k = map(int, input().split())
s = input()

best_ind, best_len = distinct_k(n, k, s)

if best_len == -1:
    print("DOES NOT EXIST")
else:
    print(s[best_ind:best_ind + best_len])
```

# Straightforward Solution

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            distinct = 0
            for symbol in ascii_lowercase:
                if symbol in substring:
                    distinct += 1
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

# Straightforward Solution

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            distinct = 0
            for symbol in ascii_lowercase:
                if symbol in substring:
                    distinct += 1
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- What is the time complexity?

# Straightforward Solution

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            distinct = 0
            for symbol in ascii_lowercase:
                if symbol in substring:
                    distinct += 1
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- What is the time complexity?
- There are  $\mathcal{O}(N^2)$  substrings of the string

# Straightforward Solution

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            distinct = 0
            for symbol in ascii_lowercase:
                if symbol in substring:
                    distinct += 1
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- What is the time complexity?
- There are  $\mathcal{O}(N^2)$  substrings of the string
- Checking each one takes us  $\mathcal{O}(N)$  time, so  $\mathcal{O}(N^3)$  in total.

# Straightforward Solution

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            distinct = 0
            for symbol in ascii_lowercase:
                if symbol in substring:
                    distinct += 1
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- What is the time complexity?
- There are  $\mathcal{O}(N^2)$  substrings of the string
- Checking each one takes us  $\mathcal{O}(N)$  time, so  $\mathcal{O}(N^3)$  in total.
- Way too slow!

# Constant optimization

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            present = [False for _ in range(26)]
            for symbol in substring:
                present[ord(symbol) - ord('a')] = True
            distinct = sum(present)
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

# Constant optimization

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            present = [False for _ in range(26)]
            for symbol in substring:
                present[ord(symbol) - ord('a')] = True
            distinct = sum(present)
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.



# Constant optimization

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            present = [False for _ in range(26)]
            for symbol in substring:
                present[ord(symbol) - ord('a')] = True
            distinct = sum(present)
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.

# Constant optimization

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            present = [False for _ in range(26)]
            for symbol in substring:
                present[ord(symbol) - ord('a')] = True
            distinct = sum(present)
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.
- Note that counts barely differs between adjacent values of end

# Constant optimization

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        for end in range(start, n+1):
            substring = s[start:end]
            present = [False for _ in range(26)]
            for symbol in substring:
                present[ord(symbol) - ord('a')] = True
            distinct = sum(present)
            cur_len = len(substring)
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- This is a little faster, by a factor of 26 approximately.
- Time complexity is the same.
- Note that counts barely differs between adjacent values of end
- Build it as the substring grows.

# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- Now each substring is processed in constant time.

# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is  $\mathcal{O}(N^2)$

# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is  $\mathcal{O}(N^2)$
- For a given value of ind, adjacent start values have similar values of counts.

# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is  $\mathcal{O}(N^2)$
- For a given value of ind, adjacent start values have similar values of counts.
- Note that adding characters will never decrease distinct.



# Incremental

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    for start in range(n):
        present = [False for _ in range(26)]
        for end in range(start, n):
            present[ord(s[end]) - ord('a')] = True
            distinct = sum(present)
            cur_len = end - start + 1
            if distinct == k and cur_len > best_len:
                best_ind = start
                best_len = cur_len
    return best_ind, best_len
```

- Now each substring is processed in constant time.
- Time complexity is  $\mathcal{O}(N^2)$
- For a given value of ind, adjacent start values have similar values of counts.
- Note that adding characters will never decrease distinct.
- However, removing elements from the front may reduce distinct.

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- What is the time complexity?

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- What is the time complexity?
- It may seem quadratic at first

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- What is the time complexity?
- It may seem quadratic at first
- Each element gets added and removed once, so  $\mathcal{O}(N)$ .

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- What is the time complexity?
- It may seem quadratic at first
- Each element gets added and removed once, so  $\mathcal{O}(N)$ .
- Lets introduce  $C$ , the number of different symbols possible.

# Sliding Window

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            count[c] += 1
            end += 1
            distinct = sum(x > 0 for x in count)
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        count[ord(s[start]) - ord('a')] -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- What is the time complexity?
- It may seem quadratic at first
- Each element gets added and removed once, so  $\mathcal{O}(N)$ .
- Lets introduce  $C$ , the number of different symbols possible.
- The time complexity is actually  $\mathcal{O}(NC)$ , but we can do better!

# Sliding Window Improved

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            if count[c] == 0:
                distinct += 1
            count[c] += 1
            end += 1
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        c = ord(s[start]) - ord('a')
        count[c] -= 1
        if count[c] == 0:
            distinct -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```



# Sliding Window Improved

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            if count[c] == 0:
                distinct += 1
            count[c] += 1
            end += 1
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        c = ord(s[start]) - ord('a')
        count[c] -= 1
        if count[c] == 0:
            distinct -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- Now adding/removing an element is  $\mathcal{O}(1)$ .

# Sliding Window Improved

```
def distinct_k(n, k, s):
    best_ind, best_len = -1, -1
    start, end, distinct = 0, 0, 0
    count = [0 for _ in range(26)]
    while start < n:
        while end < n:
            c = ord(s[end]) - ord('a')
            if distinct == k and count[c] == 0:
                break
            if count[c] == 0:
                distinct += 1
            count[c] += 1
            end += 1
        cur_len = end - start
        if distinct == k and cur_len > best_len:
            best_ind = start
            best_len = cur_len
        c = ord(s[start]) - ord('a')
        count[c] -= 1
        if count[c] == 0:
            distinct -= 1
        start += 1
        distinct = sum(x > 0 for x in count)
    return best_ind, best_len
```

- Now adding/removing an element is  $\mathcal{O}(1)$ .
- The time complexity is now  $\mathcal{O}(N + C)$ .

- This method is applicable when working with substrings or subarrays.

- This method is applicable when working with substrings or subarrays.
- The data has to be contiguous, or in other words, no gaps between selected elements.

- This method is applicable when working with substrings or subarrays.
- The data has to be contiguous, or in other words, no gaps between selected elements.
- Usually you want the maximal or the minimal window fulfilling a certain condition.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation add which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation `add` which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.
- Define an operation `remove` which removes  $a_i$  from your subarray, finally increasing  $i$  by 1.



# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation `add` which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.
- Define an operation `remove` which removes  $a_i$  from your subarray, finally increasing  $i$  by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation `add` which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.
- Define an operation `remove` which removes  $a_i$  from your subarray, finally increasing  $i$  by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation `add` which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.
- Define an operation `remove` which removes  $a_i$  from your subarray, finally increasing  $i$  by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.
- Step 3: Perform `remove` and go to step 1.

# General Method

- Suppose that your current window is  $[i, j)$  which are both initialized as 0.
- Define an operation `add` which adds  $a_j$  to your subarray, finally increasing  $j$  by 1.
- Define an operation `remove` which removes  $a_i$  from your subarray, finally increasing  $i$  by 1.
- Step 1: If performing `add` does not break your condition, perform it and repeat step 1. Otherwise go to step 2.
- Step 2: Check if the current window is a better answer and possibly update. Then go to step 3.
- Step 3: Perform `remove` and go to step 1.
- Time complexity is  $\mathcal{O}(N \cdot (X + Y))$  where  $X$  and  $Y$  are the cost of `add` and `remove`, respectively.