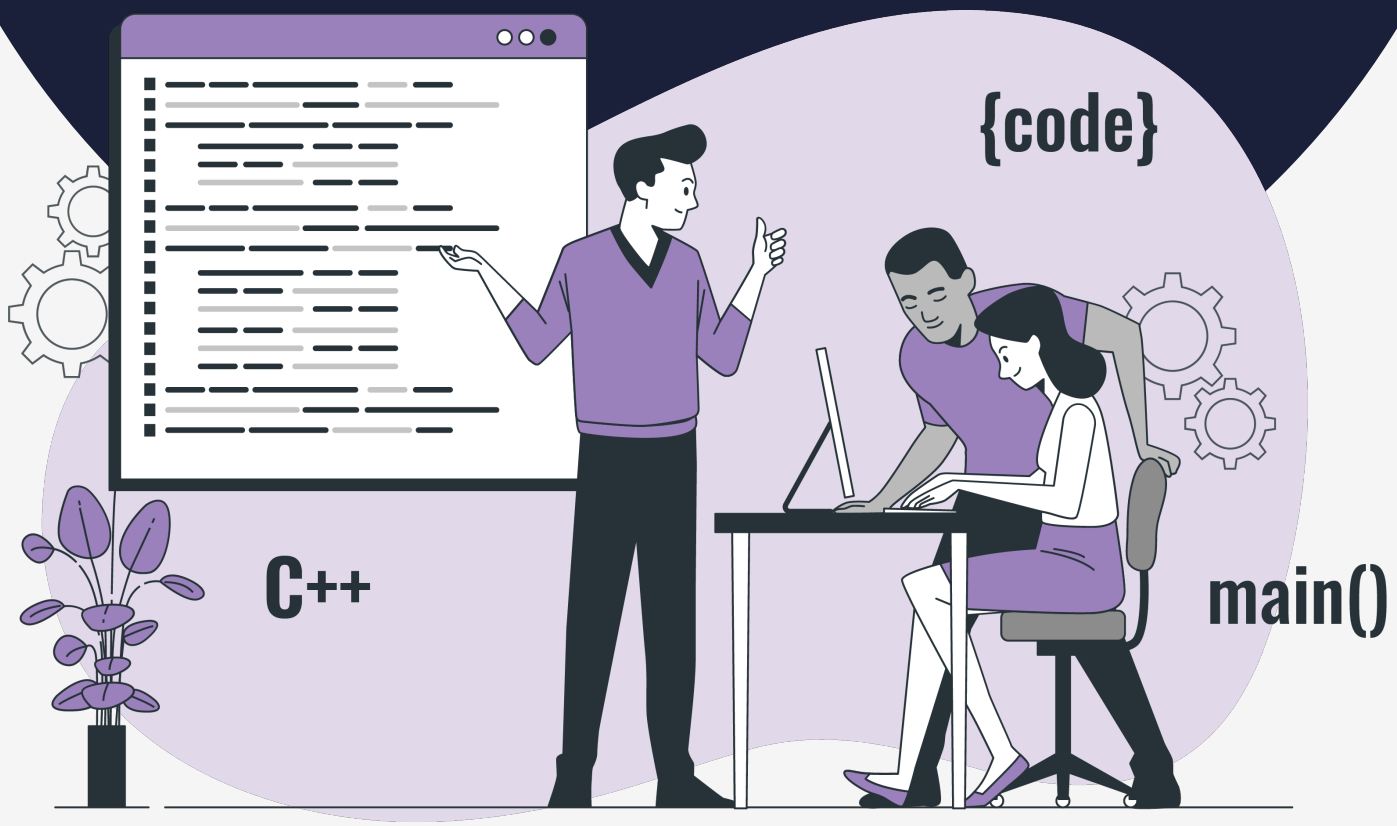
Lesson:

Strings



Pre-Requisites

- Basic knowledge of vectors
- Basic idea of coding in C++

Today's Checklist:

- Introduction to strings
- Indexing of strings
- ASCII table
- User input string
- String vs character array
- Commonly used inbuilt functions
- Bucket sort and its applications
- Given a sentence, split every single word of the sentence and print in a new line.
- Given a sentence 'str', and a word, count the frequency of the given word.
- Sliding window concept
- Questions based on strings.

What is a String in C++?

A string in C++ is a type of object representing a collection (or sequence) of different characters. Strings in C++ are a part of the standard string class (std::string). The *string class* stores the characters of a string as a collection of bytes in contiguous memory locations.

To use string one must include the header the file #include<strings> or the universal header file #include<bits/stdc++.h>.

Syntax :
string String_Name;

Example: string str = "pwskills";
string subject = "C++";

1. string str_name = "hello coders";
2. string str_name("physics wallah");

Indexing of characters in a string:

Let's assume any string to be "pwcoder". The indexing is similar to indexing in an array. It begins with 0 from the very first character and ends with a null character(\0). An extra space is for a null character after the end of any string.

0	1	2	3	4	5	6	7
p	w	c	o	d	e	r	/0

ASCII values: Each character has an associated integer/numeric value.
For example 'a' to 'z' ranges from 97 to 122.
Where a has a value 97, b has a value 98, c for 99 and so on.
A -> 65
B ->66
|
|
|
Z->90
There are numeric values for other characters such as #, @, $ etc.
For reference one can check the ascii table for numeric value of any character whose link is provided as below.
https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html

Taking string input:

To provide our program's input from the user, we generally use the cin keyword along with the extraction operator (>>). By default, the extraction operator considers white space (such as space, tab, or newline) as the terminating character. So, suppose the user enters "Physics Wallah" as the input. In that case, only "Physics" will be considered input, and "Wallah" will be discarded.

Let us take a simple example to understand this:

Code Link: **https://pastebin.com/dbrSjZss**

Output:

```
Enter your string: physics wallah
You have entered: physics
PS C:\Users\atiba\OneDrive\Desktop\C++programs>
```

In the above example, the user entered "Physics Wallah" in the input. As " " is the terminating character, anything written after " " was discarded. Hence, we got "Physics" as the output.

To counter this limitation of the extraction operator, we can specify the maximum number of characters to read from the user's input using the cin.get() function.

Getline(cin,str)

Let us take an example to understand this:

Code link: **https://pastebin.com/7es4Gwbt**

Output:

```
Enter your string: Physics wallah
You have entered: Physics wallah
```

Here we have declared a character array, of maximum size of 50 characters.

We have taken input by writing the statement:

cin.get(str_name , lengthOfString);

String vs. Character Array:

C++ supports both strings and character arrays. Although both help us store data in text form, strings and character arrays have a lot of differences. Both of them have different use cases. C++ strings are more commonly used because they can be used with standard operators while character arrays can not. Let us see the other differences between the two.

Comparison	String	Character Array
Definition	String is a C++ class while the string variables are the objects of this class	Character array is a collection of variables with the data type *char*.
Syntax	string string_name;	char arrayname[array_size];
Access Speed	Slow	Fast
Indexing	To access a particular character, we use "str_name.charAt(index)" or "str[index]".	A character can be accessed by its index in the character array.
Operators	Standard C++ operators can be applied.	Standard C++ Operators can not be applied.
Memory Allocation	Memory is allocated dynamically. More memory can be allocated at run time.	Memory is allocated statically. More memory can not be allocated at run time.

Commonly used inbuilt string functions:

1. Reverse() : This function accepts 2 parameters and reverses the string from beginning pointer to the end pointer.

Syntax: reverse(ptr1 , ptr2). The first pointer ptr1 is included and the second pointer ptr2 is not included. That means the string from ptr1 to ptr2 - 1 will be reversed.

Time complexity: let n = ptr2 - ptr1. Therefore time taken by the reverse() function is O(n) where n is the number of characters involved in the reverse process.

The code showing reverse function is illustrated below:

Code link: https://pastebin.com/6fke7hSh

Output:

```
enter your string: PWskills
The reversed string is: sllikswP
```

2. substr(): This function is used to generate a substring of a given string.

Syntax: str_name.substr(position , length)

This is the general syntax where we provide the name of the string whose substring is required. The first parameter indicates the position from where the substring extraction begins, and the second parameter indicates the length till where you want the substring.

An example illustrating the function is given below:

Code link: https://pastebin.com/4AJaHMTp

Output:

```
enter your string: PhysicsWallah
The substring obtained is: sicsW
```

Here we have given the starting index as 3 and want the substring of length 5.

The time complexity of the given function is again the length of the substring.

Time complexity: O(length)

To get a substring after a particular character:

Syntax 2: str_name.substr(position)

As per this syntax, the complete string after the mentioned "position" will be extracted as a substring as shown below in the example.

Code Link: https://pastebin.com/VAFKfmsV

Output:

```
enter your string: physicswallah
The substring obtained is: sicswallah
```

Here the complete string after index 3 (including index 3) is obtained as a substring.

To get a substring before any character:

Syntax 3: str_name.substr(0 , end_position)

Here the string from 0 to end_position - 1 is considered as the required substring.

This is a specific case of general syntax.

3. push_back(): This function is used to push another character or string at the end of the current string(string with which the function is used/called).

Syntax: str_name.push_back(str2_name)

"Str2_name" string will be pushed at the end of the string "str_name" as shown in the following example.

Code Link: https://pastebin.com/BZHDmSZE

Output:

```
enter your string: pwskill
enter the new string or character to be pushed: s
The new string obtained is: pwskills
```

4. The "+" operator: This is directly used to concatenate 2 strings.

Code link: https://pastebin.com/q5SW9sLp

Output:

```
enter your string: physics
enter the new string or character to be pushed: wallah
The new string obtained is: physicswallah
```

Here it was written as s+=t that is equivalent to s = s + t, had it been written s = t +s then the resulting string would be "wallahphysics".

When we write s+=t that signifies we are appending the string "t" at the back of string s.

Statement 2 s = s + t shows we have created a new copy of string s and added string t at the back of string s.

The only difference in the above two statements is that in the second case extra space will be taken by the reformation of string 's' whereas in the first case no such extra space of string 's' will be occupied.

Homework: Try t+s functionality on your own.

5. strcat(): This function is used to concatenate 2 character arrays.

Syntax: strcat(s1_name , s2_name)

This is equivalent to s1_name = s1_name + s2_name as illustrated below:

Code link: https://pastebin.com/eairj3tq

Output:

```
enter your string: physics
enter the new string or character to be pushed: wallah
The new string obtained is: physicswallah
```

6. size(): This function is used to find the size of the string.

Syntax: str_name.size()

Code link: https://pastebin.com/dKXWDmAB

Output:

```
pwcoders
8
```

Difference between size() and strlen() functions:

- size() function is used to know the length of the strings whereas strlen() function is used to know the length of the character array.
- strlen() function takes O(n) time whereas size() function uses O(1) time, where n = length of the array.

7. to_string(): This function is used to convert a numeric value into string type.

Syntax: Suppose n is an integer. To convert this integer to string we have to write the following way:

to_string(n).

This function is used when we have to perform operations on digits of a number. Converting into string makes the digits accessible in an indexed manner which is quite easy to use.

The following code illustrates:

Code link: https://pastebin.com/330dW56A

Output:

```
enter the number: 3452
3452
3 4
```

Bucket sort on strings

We have 128 different types of characters available that can be A-Z, a-z, special characters such as !, @, # etc, digits from '0' to '9'etc.
Since the maximum number of characters can be at max 128 we can use an array or vector of size 128 such that each of the index of the array represents a particular character.

Now, we can use the ASCII values of the characters to mark the indices of the characters. For example,
The ASCII value of 'a' = 97
'b' = 98 and so on
So if we created an array 'arr' of size 128 then arr[97] will be reserved for the frequency of 'a', similarly arr[98] will be reserved for frequency of 'b' and so on.

Once the frequencies of every character are stored we can use it to build a sorted string, because it is sure that 'a' will always be ahead of 'b', 'b' will always be ahead of 'c' and so on. This is the concept of bucket sort. Below example will clarify this more.

Problem 1: Given a string 'str', sort the given string using count sort technique.

Constraints: The string will contain only alphabetical characters from a-z.

Input 1: "codingwallah"
Output 1: "aacdghillnow"

Code link: https://pastebin.com/y3JHF5Ar

Input 2: "star"
Output 2: "arst"

Code link: https://pastebin.com/y3JHFEAr

Output:

```
Enter the string:
codingwallah
The sorted string is: aacdghillnow
```

Approach:

- In the **"main"** function we have taken a string as input from the user and called for the function **"countSort"** that accepts the string as a parameter and returns the sorted string.
- In the **"countSort"** function we have declared an array of size 26. Here in the question the constraints are specified that there will be only a-z in the string.
- To use that information we have taken the array of size 26.
- Str[i] - 'a' indicates how to fit a value 97 or more than that in an array of size 26 where each character will be treated as an index of the array.
- We know that the ascii value of 'a' = 97 . Let's assume for a second that str[i] = 'a'. We have written the logic str[i] - 'a' , whose value = 'a' - 'a', here subtracting the characters from each other basically means subtracting the ASCII value of the characters.
- Here 'a' - 'a'= 97 - 97 = 0. Now this 0 will be treated as an index for 'a' for storing its frequencies.
- Similarly, 'b' - 'a' = 98 - 97 = 1, meaning that the1st index is reserved for character 'b' and so on. This way because we have 26 different characters each character will be allotted an index from 0 to 25. Starting from 0 for 'a' and 25 for 'z'.
- The second method can also be used where we can declare an array of size 128 but that will only take extra space. Instead this method will be more space and time efficient since it is already mentioned in the question that we have characters only in the range a-z.
- Once all the frequencies are stored, our task is to regenerate the newly sorted string.
- To do this we can iterate from 0 to 25 on every index, and whatever is the frequency of the **'i'th** character we will add that particular character, that many times.
- The while loop will keep track of the frequency of every character whereas the for loop will make sure that each and every character is observed.
- If any particular character is not occurring in the string we have already mentioned its value as 0 in the beginning, hence while loop will not work for that index.
- Here for adding the character we have written the statement i+'a'. To extract the index we have subtracted character 'a' from the str[i], to get back the character we have to add the character 'a' to the index to obtain the original character. Eg. lets take value of i = 1, i + 'a' = 1 + 97 = 98 which is the ascii value of 'b', hence in the string we will add the value 'b'.
- This way the whole string in a sorted fashion will be constructed. Once the for loop will end, we will return this string.

Time complexity: Let 'n' be the length of the string
Then time complexity will be O(n).
Space complexity: Since we have created a constant spaced array of 26 size, then space consumed will be O(26) which is nearly equal to O(1) / constant space.

Problem 2: Given two strings s and t, return true if t is an anagram of s, and false otherwise.
An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Constraints : String s and t will only contain lowercase alphabetical characters.

Input 1: s = “anagram” , t = “nagaram”
Output 1: yes

Input 2: s = “bank” , t = “atm”
Output 2: no

Solution:

Code link: https://pastebin.com/KqEaPzUi

Output:

```
Enter the string:
codingwallah
The sorted string is: aacdghillnow
```

Approach:

- In the main function we have taken 2 strings as input from the user. We have made a call to the function isAnagram(s , t), if they are anagrams then we will print yes otherwise no.
- In the “isAnagram” function we have passed two strings. The return type of this function is “bool”. The concept used here is similar to the bucket sort concept, where we will store each character’s frequency in an array and will use this data further.
- Before storing the frequencies, one very basic check should be if the length of both the strings are not equal they can never be treated as anagrams, so we simply return a false from there only.
- Now if the length of both the strings are same then we have two choices, first is we create two arrays and store the character’s frequency of both the strings independently and then compare each character’s frequency one by one.
- If every character has equal frequencies in both the arrays then we can conclude that the two strings are anagrams.
- The second approach is a bit more intuitive where we have used just a single array of size 26 and to make a difference in both the strings we will increment the character’s frequency whenever we are counting for string ‘s’ and will decrement the frequency of the character when we are counting for string ‘t’.
- Once both the strings are stored frequency wise, we will iterate over the “count” array. If any of the index has value other than zero that means there is a mismatch between the character’s frequency in both the strings, the frequency is not nullified. We can simply return a false from here.
- Once all 26 elements have been checked and no discrepancy is found we can now return true from this function.

Time complexity: Let ‘n’ be the length of the string
Then time complexity will be O(n).

Space complexity: Since we have created a constant spaced array of 26 size, then space consumed will be O(26) which is nearly equal to O(1) / constant space.

fix

Problem 3: Given two strings s and t, determine if they are isomorphic.
Two strings s and t are isomorphic if the characters in s can be replaced to get t.
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself. Two strings s and t are isomorphic.

Constraints: String s and t can contain any valid ASCII character.

Input 1: s = "egg" , t = "add"
Output 1: yes

Explanation: 'e' is mapped with 'a' and 'g' is mapped with 'd'.

Input 2: s = "coding" , t = "ok"
Output 2: no

Solution:
Code link: https://pastebin.com/YCHkixP5

Output:

```
Enter the strings:
egg add
Are the strings isomorphic ?
yes
```

Approach:

- In the main function we have taken two strings as input from the user and called a function **"isIsomorphic"** that takes two strings as parameters.
- In the **"isIsomorphic"** function we have taken two separate arrays of size 128 each to ensure every character is counted because in the question statement it is already mentioned that both the strings can contain any character.
- Here we have already taken an array of size 128 therefore for every ascii value we can reserve an index for it.
- Firstly, we checked the size of the strings. If a mismatch is observed there we can simply return false from there only.
- In order to do mapping, we can simply store a unique integer for any character in both the strings. If anytime the same character repeats but they have different numeric values stored against that character that means one character is mapped to more than one character in the second string, hence they are not isomorphic.
- We know that any index once occurred will never be repeated so in place of that unique integer we have used index value. We have incremented it by 1 since we have initialized the array with 0 value. So if any character is not there the value will be 0. Hence we have started with 1-based counting. Hence we used the value **"i + 1"**.
- If none of the time any "return false" condition arose that means the string is isomorphic, so we can simply return true since both the strings are isomorphic in nature.

Time complexity: Let 'n' be the length of the string
Then time complexity will be O(n).

Space complexity: Since we have created a constant spaced array of 128 size, then space consumed will be O(128) which is nearly equal to O(1) / constant space.

Problem 4: Given an array of strings Write a program to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".

Problem 4: Given an array of strings.Write a program to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".

Input 1: arr = [“flower” , “flight” , “flask”]
Output 1: “fl”

Input 2: arr = [“physics” , “wallah”]
Output 2: “”

Solution:

Code link: https://pastebin.com/As2dpXWz

Output:

```
Enter the number of strings in the list : 3
Enter the strings:
flow flower flight
The prefix is: fl
```

```
Enter the number of strings in the list : 2
Enter the strings:
physics wallah
The prefix is:
```

Approach:

- In the **main** function we have taken the vector of strings as input and called for the function **“longestCommonPrefix”** that accepts the vector of string as a parameter.
- In the **“longestCommonPrefix”** function we initially sorted the array.
- The main advantage we get after sorting is that the most distinct strings will be on the opposite ends of the vector and the highest risk of mismatching of the characters will be in the first and the last strings.
- This way we only need to check for common prefix in the first and last strings. If we find any prefix in these two strings then this prefix will surely be there in the strings left in the middle.
- Therefore we only need to check for the first and last string and will try to match as much as the characters are the same.
- Once the characters are mismatched we will immediately break from there or we can simply return the answer because prefix is a continuous string from the beginning.
- Any mismatch will limit the prefix to the previous character.
- At the end we will return our string.

Time complexity: O(nlogn*m + min(s[0].size() , s.back().size())), overall since we take maximum it will be O(nlogn*m)
Where n = length of the vector / number of strings in the vector
And m = general length of each string in the vector
Here s = name of the vector of strings.

Space complexity: As no extra space apart from input is used therefore space will be constant / O(1) in nature.

Approach 2: second approach could be a simple traversal and checking of prefix over every string in the vector.

Code link: https://pastebin.com/ja8KVgNG

- In this approach we have simply compared every other string relative to the first string.
- We have taken the minimum value of the index that is matching and at last after comparing all strings we will return the substring from the 0th string of length equal to “ans”.

Time complexity: O(n*m),
Where n = length of the vector / number of strings in the vector
And m = general length of each string in the vector

Space complexity: As no extra space apart from input is used therefore space will be constant / O(1) in nature.

Problem 5: An encoded string (s) is given, and the task is to decode it. The encoding pattern is that the occurrence of the string is given at the starting of the string and each string is enclosed by square brackets.

Note: The occurrence of a single string is less than 1000.

Input: s = 1[b]
Output: b
Explanation: 'b' is present only one time.

Input: s = 3[b2[ca]]
Output: bcacabcacabcaca
Explanation: 2[ca] means 'ca' is repeated twice which is 'caca' which concatenated with 'b' becomes 'bcaca'.

This string repeated thrice becomes the output.
Solution:

Code link: https://pastebin.com/ewxTtUjn

Output:

```
Enter the string : 3[av1[d]]
Desired output is: avdavdavd
```

Approach:

- If we see a digit, it means that we need to form a number, so just do it: multiply the already formed number by 10 and add this digit.
- If we see an open bracket [, it means that we are just right before finishing to form our number: so we put it into our string.
- If we have close bracket], it means that we just finished [...] block and what we have in our string: on the top it is solution for what we have inside brackets, before we have number of repetitions of this string and finally, before we have string built previously: so we concatenate **“result”** and **“str * number”**.
- Finally, if we have some other symbol, that is a letter, we add it to the last element of our stack.
- For better understanding the process, let us consider example s = 3[a5[c]]4[b]:
- At first we have an empty string. ‘i’ is an iterator iterating on the string.
- At i = 0, "3", open bracket: now we have a string with 3.
- At i = 2, "3[a”: build our string
- 'At i = 3, “3[a5", open bracket: add number
- At i = 5, “3[a5[c” build string
- At i = 5, "3[accccc” : now we have closing bracket, so we remove last elements and put “accccc” into our string.

- At i = 7, “acccccacccccaccccc” we again have closing bracket, so we remove last 3 elements and put new one.
- At i = 9, “acccccacccccaccccc4[“: open bracket, add number
- At i = 10, “acccccacccccaccccc4[b” build string
- “acccccacccccacccccbbbb” closing bracket: remove elements.

Final result = acccccacccccacccccbbbb

Time complexity: O(n) where n is the expanded/decoded string length.
Space complexity: O(n) where n is the expanded/decoded string length.

Sliding window: The Sliding window is a problem-solving technique of data structure and algorithm for problems that apply arrays or lists. These problems are painless to solve using a brute force approach in **O(n²)** or **O(n³)**. However, the Sliding window technique can reduce the time complexity to **O(n)**.

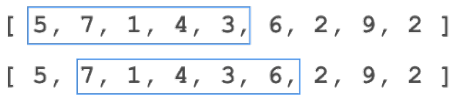


Figure 1: Sliding window technique to find the largest sum of 5 consecutive numbers.
The basic idea behind the sliding window technique is to transform two nested loops into a single loop.

Below are some fundamental clues to identify such kind of problem:
The problem will be based on an array, list or string type of data structure.

- It will ask to find subranges in that array or string and will have to give longest, shortest, or target values.
- Its concept is mainly based on ideas like the longest sequence or shortest sequence of something that satisfies a given condition perfectly.

Let's say that if you have an array like below:

Figure 2: Array of values
A sliding window of **size 3** would run over it like below:

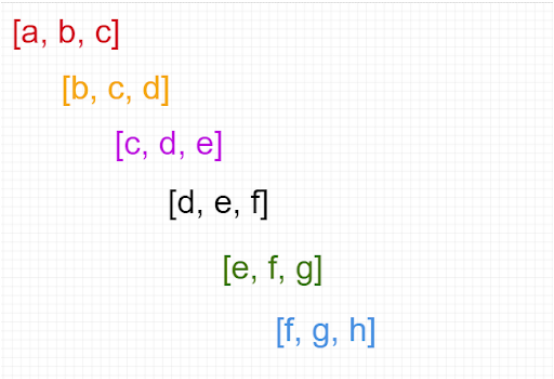


Figure 3: Sliding window of size 3 (Sublist of 3 items)

Problem 6: Given a binary string and an integer k, return the maximum number of consecutive 1's in the string if you can flip at most k 0's.

Input : "0001101011" , k = 2
Output: 7
Explanation: we have flipped these '0's, 0001111111. In total we have 7 '1' consecutively.

Input: "000110111" , k = 0
Output: 3
Explanation: since there are no k's available, therefore the best we can have is the maximum number of the consecutive 1's that are available in the given string. 000110111.

Solution:
Code link: https://pastebin.com/azhxpS8g

Output:

```
Enter the string : 0001101011
Enter the number of flips allowed : 2
Desired output is: 7
```

Approach:

- The question translates to finding the max length of subArray with at most K 0s.
- Using the sliding window technique.
- Keep count of the number of 0s found in the current window.
- If the count is > K, then increment **left** until the count goes to <=K.
- At each iteration find the **maxLength**.

Time complexity : O(n) where n = length of the string.
Space complexity : O(1)

Next class teaser:

- OOPS concept