Strings

Topics

1. Definition of a String
2. Problems & Solutions
3. Important References

Definition of a String

In computer science a *string* is any finite sequence of *characters* (*i.e.,* letters, numerals, symbols and punctuation marks).

An important characteristic of each string is its *length*, which is the number of characters in it. The length can be any *natural number* (*i.e.,* zero or any positive integer). A particularly useful string for some programming applications is the *empty string*, which is a string containing no characters and thus having a length of zero. A *substring* is any contiguous sequence of characters in a string.

The String Data Type

How strings and string data types are represented depends largely on the *character set* (e.g., an **alphabet**) for which they are defined and the method of *character encoding* (*i.e.,* how they are represented by bits on a computer). String implementations (formerly) were usually designed to work with **ASCII** (the de facto standard for the character encoding used by computers and communications equipment to represent text) or with its subsequent extensions. In recent years, however, the trend has been to implement strings with **Unicode**, which attempts to provide character codes for all existing and extinct written languages.

The Programming Languages that use ASCII style character encoding are notably C & C++. The Programming Languages that use Unicode style character encoding are notably Java & Python3.

How Do Popular Languages Handle Strings?

1. **C:**
    a. **Basics of C Strings**.
    b. **Character Array & Character Pointer**.
    c. **NYU Material on Strings in C**.
    d. **WikiBook on C Strings**.
    [The aforementioned articles/links are a **must read**, for all programmers]
2. **C++:**
    a. **Basics of C++ Strings**.
    b. **`<string>` library in C++ STL**.

3. **Java:**
    a. **Basics of String Class**.
    b. **Official Documentation of String Class**.

4. **Python:**
    a. **Official Documentation on Strings in Python 3**.
    b. **Python Strings by Google**.

Problems & Solutions

1. Given a string, WAP to count the number of occurrences of each character.
   [Assume that the string only has lowercase english characters]
   Solutions
   - **Approach #1:** Maintain an array called `count[26]`. While iterating over the string `str`, increment the count of each character referring to their respective positions in the `count[]` array by executing `count[str[i]-'a']++`.
     Time Complexity: `O(N)`, Space Complexity: `O(26) = O(1)`

   - **Approach #2:** Maintain a `HashMap<char,int> cnt;` and while iterating through the string `str`, just add any new character into the `HashMap` with value `1` by executing `cnt.add(str[i],1);` or if the character is already inserted into the `HashMap`, then simply increment the character's value by `1` in the `HashMap` by executing `cnt[str[i]]++;`
     Time Complexity: `O(N)`, Space Complexity: `O(26) = O(1)`
     Note: Search/Insert/Delete in Unordered HashMap takes O(1) on an average case. Hence Approach-1 is better than Approach-2.

2. Given a string, WAP to convert all its lowercase characters, to uppercase characters and convert all its uppercase characters to its equivalent lowercase characters.
   Example: Given string: `"sMaRt inTerViewS"`
   Expected Output: `"SmArT INtERvIEWs"`
   Solutions
   - **Approach #1:** While iterating through each character in the string, if the character is an uppercase character, then add 32 to that character, and if the character is a lowercase character, then subtract 32 from that character. Ignore all the characters which are special symbols and digits.
     Time Complexity: `O(N),` Space Complexity: `O(1)`

   - **Approach #2:** For a string `str`, we can XOR each character in `str` with 32.
     [Hint: Understand the bit representation of each uppercase and lowercase character in the character set to understand why this solution works].
     Time Complexity: `O(N)`, Space Complexity: `O(1)`

3. Given two strings `A` & `B`, check if both strings are anagrams of each other.
   [Def^{n} of Anagram: Anagrams are two/more strings of same size which have the same exact characters in each of the strings. Any permutation of a string, is it's anagram.]
   Example 1: `A = "listen"; B = "silent";`
   {Here, both `A` & `B` are anagrams of each other}
   Example 2: `A = "smart"; B = "stack";`
   {Here, both `A` & `B` are not anagrams of each other}
   Solutions
   [Note: Check if both strings `A` & `B` are equal in length before applying any approach. If `A.size() != B.size()`, then both strings `A` & `B` are not anagrams of each other]

- **Approach #1:** Maintain two count arrays `cnt1[26]` and `cnt2[26]` for strings `A` & `B` respectively. Iterate through both strings `A` & `B`, while incrementing the count of each character by executing `cnt1[A[i]]++;` and `cnt2[B[i]]++;` respectively. Then iterate through all the values of `cnt1[]` and `cnt2[]` by comparing each value of `cnt1[]` and `cnt2[]` at a time. If at any moment `cnt1[i] != cnt2[i]`, then we say that strings `A` & `B` are not anagrams of each other, else the strings are anagrams of each other. Can we do it using a single count array?
  Time Complexity: $O(N)$, Space Complexity: $O(26) = O(1)$

- **Approach #2:** Sort strings `A` & `B`. If the sorted strings are equal, they're anagrams of each other, else they're not anagrams.
  Time Complexity: $O(N \times \log_2(N))$, Space Complexity: $O(1)$

4. Given a string `str`, WAP to find the:
   - First Non-repeating character in `str` in:
     i. Relative Scale. Example: `str = "abbajjnkkltld"`, o/p - `'n'`
     Solution:
     ```
     int cnt[26] = {0};
     ∀ i=0 to str.size()-1 { cnt[str[i]-'a']++; }
     ∀ i=0 to str.size()-1 { if(cnt[str[i]] == 1) return str[i]; }
     ```

     ii. Absolute Scale. Example: `str = "abbajjnkkltld"`, o/p - `'c'`
     Solution:
     ```
     int cnt[26] = {0};
     ∀ i=0 to str.size()-1 { cnt[str[i]-'a']++; }
     ∀ i=0 to 25 { if(cnt[i] == 1) return 'a'+i; }
     ```

   - First Repeating character in `str` in:
     i. Relative Scale. Example: `str = "axcdefqrfstuxzvwxyz"`, o/p - `'x'`
     Solution:
     ```
     int cnt[26] = {0};
     ∀ i=0 to str.size()-1 { cnt[str[i]-'a']++; }
     ∀ i=0 to str.size()-1 { if(cnt[str[i]] > 1) return str[i]; }
     ```

     ii. Absolute Scale. Example: `str = "axcdefqrfstuxzvwxyz"`, o/p - `'f'`
     Solution:
     ```
     int cnt[26] = {0};
     ∀ i=0 to str.size()-1 { cnt[str[i]-'a']++; }
     ∀ i=0 to 25 { if(cnt[i] > 1) return 'a'+i; }
     ```

     Time Complexity: $O(N)$, Space Complexity: $O(26) = O(1)$

5. Given a string A_N, check whether the string is a palindrome or not.
   Solution: Use **two-pointer** technique, with p1 at 0 and p2 at N-1.
   Time Complexity: $O(N)$, Space Complexity: $O(1)$

6. Enclosing Substring Problem: Given 2 strings A_N & B_M, WAP to find the smallest substring of A_N which covers all the characters in B_M.
   [Note: Here N & M are sizes of strings A & B respectively, and $M \leq N$]
   Example: A_N = "maqxbbcvayaxyzab", B_M = "axyabcy";

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15..
A_N:	m	a	q	x	b	b	c	v	a	y	a	x	y	z	a	b

   A_N[5:12] = "bcvayaxy" is the minimum substring that encloses string B_M completely.
   Answer: 8 (length=12-5+1)
   Solutions:
   - **Approach #1:** Generate all substrings A_N. Now, while generating each substring of A_N, check if it contains all characters of B_M by using linear search for each character of B_M , in the current substring of A_N. Take care of repeating characters carefully.
     Time Complexity: $O(N^2 \times M \times N)$, Space Complexity: $O(N)$
   - **Approach #2:** Generate all substrings A_N. Now, while generating each substring of A_N, check if it contains all characters of B_M. To do so, sort the string B_M and also the substring of A_N and check using two-pointer technique.
     Time Complexity: $O(M \times \log_2(M) + N^2(N \times \log_2(N) + N + M)$, Space Complexity: $O(N)$
   - **Approach #3:** Generate the $cnt_B[]$ from B_M. Now, while generating each substring of A_N, check if $\forall_{i=0}^{25} cnt_B[i] \leq cnt_A[i]$ and also maintain a variable which indicates the smallest length enclosing substring for string B_M.
     Time Complexity: $O(M + N^2(N + 26))$, Space Complexity: $O(26) = O(1)$
   - **Approach #4:** In the previous approach, when comparing $cnt_B[i]$ & $cnt_A[i]$, we are simply generating $cnt_A[]$ freshly everytime we generate a new substring of A_N. Instead, we can carry forward the previously taken $cnt_A[]$ into the next $cnt_A[]$ that we are going to generate. That will reduce the time complexity by a linear factor.
     Time Complexity: $O(M + N^2(1 + 26))$, Space Complexity: $O(26) = O(1)$
   - **Approach #5:** If we can enclose the string B_M using substring of A_N of length K, where $M \leq K \leq N$, then we can also say that the substrings of A_N with lengths [K+1, K+2, K+3,..., N] will also suffice and enclose the string B_M successfully. Using this insight, we can apply Binary Search for generating a certain length, and then see if a substring of that length encloses B_M successfully. If it does, then we try and look for a smaller substring than the previous length, else we try and look for a substring of greator length to enclose the string B_M.
     i. Step 1: `lo = B.size(); hi = A.size();`
        `/* M = A.size(); N = B.size(); M ≤ N; */`

ii. Step 2: `mid = lo+(hi-lo)/2;` // Binary Search for length allocation
iii. Step 3: Generate all substrings of length `mid` using carry forward technique to count each character and for each substring check if $\forall_{i=0}^{25}$ $cnt_B[i] \leq cnt_A[i]$.
iv. Step 4: If the condition in Step 3 is `true`, then we consider smaller substring for the next iteration of the binary search by making `hi = mid;` If the condition is `false`, then we generate a longer substring for the next iteration of the binary search by making `lo = mid + 1;`
v. Step 5: Run Step 2 through Step 4 until `lo < hi`.

Time Complexity: $O(M + N \times \log_2(N))$ Space Complexity: $O(1)$

- **Approach #6:** Apply **two-pointer** technique.
  i. Step 1: Prepare $cnt_B[]$ of string B_M.
  ii. Step 2: Initiate `i = j = 0;`
  iii. Step 3: Iterate on string A_N using `j`. While iterating, maintain $cnt_A[]$, and keep on checking whether $\forall_{k=0}^{25}$ $cnt_B[k] \leq cnt_A[k]$, for every iteration of `j`.
  iv. Step 4: If the condition in Step 3 yields `false`, then increment `j` by `1`, otherwise, start to find a tighter and smaller substring of A_N by incrementing `i`, by `1`, and for every incrementation of `i`, check the condition in Step 3. The generation of $cnt_A[]$ is done by using carry forward technique (to decrease the time complexity by a linear factor) on string A_N.
  v. Step 5: Repeat Step 3 through Step 4, while maintaining a answer variable to store the length of the smallest substing of A_N which encloses B_N , until `j < N` [until `j` reaches the end of string A_N].

  Time Complexity: $O(M + N(1+26))$, Space Complexity: $O(1)$

7. Substring Problem: Given two strings A_N & B_M, WAP to check if B_M is a substring of A_N, considering that $M \leq N$ [`M = B.size(); N = A.size();`].
Example: A_N = `"maqxbbcvayaxyzab"`; B_M = `"xbbcva"`;

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15..
A_N:	m	a	q	x	b	b	c	v	a	y	a	x	y	z	a	b

A_N`[3:8] = "xbbcva"` is a substring that is exactly equal to B_M.

Solutions

- **Approach #1:** First thing to know is that, there are `N-M+1` substrings of length `M`, in A_N. We generate `N-M+1` substrings from A_N and for each substring, we do a string equality check where we compare the substring of A_N with B_M character by character. If we find a substring in A_N that exactly equates to string B_M, then we have a match.
  Time Complexity: $O((N-M+1) \times M)$, Space Complexity: $O(1)$

- **Approach #2:** Use Rabin Karp's String Matching Algorithm. In the algorithm, we use the concept of **Rolling Hash**, where we carry forward the hash value of the previous substring of A_N (by applying some clever

mathematical operations) to compute the hash value of the next substring of A_N, thereby reducing the overall time complexity by a linear factor. If hash value of the substring of A_N is equal to the hash value of B_M, then we have a match. Of course, this is prone to collisions and we can handle this issue by using a strong hash function and at the same time more than one hash function.
Now, a strong hash function for lesser collisions is theoretically proven to be:

$$\texttt{hash(str)} = \sum_{i=0}^{N-1} (\texttt{str[i]} \times \texttt{P}^{\texttt{N-i}})$$

where, `str[i]` is the ASCII value of the character at position `i` in string `str`; `P` is a relatively large enough prime number(2-digit or 3-digit) like 71, 101, 199, etc; `N = str.size().`

Now, a problem that might occur when computing the hash value of any string is **integer overflow**. That's because $\texttt{P}^{\texttt{N-i}}$ can be very large, which might not fit into the range of even a long long int. Therefore, we take modulus of the result against a very large prime number like `K = 1e9+7` (or 10^9+7) while generating the hash value for the string.

We can compute $\texttt{P}^{\texttt{N-i}}$ while generating the hash value for the string, but that will increase the time complexity by a logarithmic factor. To avoid that, we precompute the powers of `P` and store them in an array say `PR[]`.

Rabin Karp's String Matching Algorithm:
Step 1: `N = A.size(); M = B.size(); K = 1e9+7; PR[M+1]; PR[0] = 1;`
`P = 101(say);` Here, `M` ≤ `N`, and `P` can take any large prime number.
Step 2: Generate `PR[]` by using the Prefix Product technique:
$\forall_{i=1}^{M}$ `PR[i] = (PR[i-1] × P) % K;`
Step 3: $\texttt{hash}_\texttt{B}$ `= 0;`
$\forall_{i=0}^{M-1}$ $\texttt{hash}_\texttt{B}$ `= (`$\texttt{hash}_\texttt{B}$ `+ (B[i] × PR[M-i])) % K;`
Step 4: $\texttt{hash}_\texttt{A}$ `= 0;`
$\forall_{i=0}^{M-1}$ $\texttt{hash}_\texttt{A}$ `= (`$\texttt{hash}_\texttt{A}$ `+ (A[i] × PR[M-i])) % K;`
Step 5: `if(`$\texttt{hash}_\texttt{A}$ `==` $\texttt{hash}_\texttt{B}$`) return true;`
Step 6: $\forall_{i=M}^{N-1}$ $\texttt{hash}_\texttt{A}$ `= (((`$\texttt{hash}_\texttt{A}$`- A[i-M] × PR[M] + A[i]) × P) % K + K)%K;`
`if(`$\texttt{hash}_\texttt{A}$ `==` $\texttt{hash}_\texttt{B}$`) return true;`

[**Note-1**: In Step 6, we add `K` to $\texttt{hash}_\texttt{A}$ because, the value might become less than 0 as it involves subtraction operation, thereby making it a negative value. Hence, we use the rule of distributing modulo operator over subtraction.]
[**Note-2**: To reduce the chances of collisions, we can find two hash values for each string using two hash functions. Then, we can combine those hash values and use it to find the required solution]
Time Complexity: `O(M+N)`, Space Complexity: `O(M)`

Alternative sources to understand Rabin Karp's String Matching Algorithm:
i. **Rabin Karp @Brilliant.org**.
ii. **Rabin Karp @Wikipedia**.

- **Approach #3:** Use **Knuth-Morris-Pratt** [KMP] String Matching Algorithm to check if B_M is a substring of A_N.
  Time Complexity: $O(M+N)$, Space Complexity: $O(M)$ [∵ Prefix Table]
  Resources:
  i. **KMP @Topcoder**.
  ii. **A very good video explanation of KMP @YouTube**.

8. Given a string A_N, find the Largest Palindromic Substring in A_N.
   Example: A_N = "maqxbbcvcbbxyzab";

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15..
A_N:	m	a	q	x	b	b	c	v	c	b	b	x	y	z	a	b

   A_N[3:11] = "xbbcvcbbx" is a substring that is also a palindrome.
   Solutions:
   - **Approach #1:** While generating all the substrings of string A_N, check whether each string is a palindrome or not, using the `is_palindrome()` function defined in Problem 5. Maintain a variable which stores the length of the largest palindromic substring.
     Time Complexity: $O(N^2 \times N)$, Space Complexity: $O(1)$

   - **Approach #2:** While iterating through the string A_N treat each character A[i] as a center and expand around that center considering that the center can be part of an even string, or an odd string.

```
int largest_palindromic_substring(string& s, int& n) {
    int palin_len = 1;
    for(int i = 0; i < n; ++i) {
        // Odd Palindrome Case
        int p1 = i-1, p2 = i+1;
        while(p1 != -1 && p2 != n && s[p1] == s[p2]) {
            --p1;
            ++p2;
        }
        palin_len = max(palin_len, (p2-p1-1));
        // Even Palindrome Case
        p1 = i-1;   p2 = i;
        while(p1 != -1 && p2 != n && s[p1] == s[p2]) {
            --p1;
            ++p2;
        }
        palin_len = max(palin_len, (p2-p1-1));
    }
    return palin_len;
}
```

     Time Complexity: $O(N^2)$, Space Complexity: $O(1)$

   - **Approach #3:** We can use Rolling Hash with Prefix Sum + Binary Search to find the longest palindromic substring in an optimal fashion.
     We use Binary Search for determining (or guessing) the length till where a center A[i] can expand on both the right and the left side making sure whether

the character `A[i]` is a center to an even/odd length palindrome.
The parameters of Binary Search are:

```
lo = 0; hi = min(i, N-i-1); mid = lo+(hi-lo+1)/2;
```

We use a utility function `valid()`, which indicates whether to shrink/expand our search space, *i.e.,* `if (valid(str, i-mid, i+mid) == true)`, then we maximize our search space by executing `lo = mid;` otherwise, we minimise our search space by executing `hi = mid-1;` Our `valid()` should check whether the string from `A[i-mid]` to `A[i+mid]` is a palindrome or not, in constant time. The `valid()` uses Prefix Hash Sum (`PHS[]`) & Suffix Hash Sum (`SHS[]`) to do that. To generate hash value for each `str`, we use the same hash function used in Problem 7, Approach #2. After that, we apply prefix sum to rolled hashes of each substring `[0:i]` being generated when we iterate `str` from `0` to `N-1` and generate our `PHS[]`. We iterate backward and generate our `SHS[]`. *i.e.,*
Initiate `PHS[N]; SHS[N];`

```
PHS[0] = str[0] ✕ PR[1];
∀ (i = 1 to N−1) PHS[i] = PHS[i-1] + (str[i] ✕ PR[i+1]);
SHS[N-1] = str[N-1] ✕ PR[1];
∀ (i = N−2 to 0) SHS[i] = SHS[i+1] + (str[i] ✕ PR[N-i]);
```

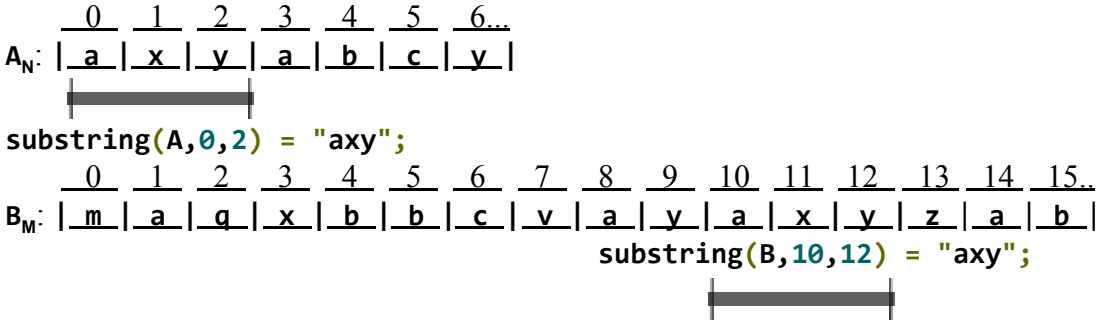
Here, `PR[]` is an array of powers of prime generated in Problem 7, Approach #2, Steps 1 & 2.
Please keep in mind that, while generating `PHS[]` & `SHS[]`, there can be integer overflow, which can be handled by simply taking modulo against a very large prime number like `K = 1e9+7` (or 10^9+7) for every interim result.
Now, to know whether the string from `A[i-mid]` to `A[i+mid]` is a palindrome or not, we check:
`if(PHS[i+mid]-PHS[i-mid-1] == SHS[i-mid]-SHS[i+mid+1])` is `true` or `false`. If it is `true`, then the string from `A[i-mid]` to `A[i+mid]` is a palindrome and the `valid()` returns `true` to the Binary Search function, and if it is not, then the string from `A[i-mid]` to `A[i+mid]` is not a palindrome and the `valid()` returns `false` to the Binary Search function.

[**Note-1**: Assume that: `U` = substring `A[i-mid]` to `A[i+mid]` & `V` = substring `A[i+mid]` to `A[i-mid]` (*i.e.,* = `reverse(U)`), we have to balance the hash values of the strings from `U` and the reverse of that string which is `V`. We balance the hash values by multiplying a factor of P^X to either string `U` or to string `V`.
Here, `P` is the prime number used to generate the powers of prime in `PR[]`, and P^X is the factor by which there is an imbalance between the hash values of the strings `U` & `V`. If we do the check mentioned above, *i.e.,* if we check: `(PHS[i+mid]-PHS[i-mid-1] == SHS[i-mid]-SHS[i+mid+1])?` without balancing the hash values, the results yielded would be incorrect.]
[**Note-2**: We have to carefully distribute modulo operator over subtraction]
[**Note-3**: We have to take care that indices: `i-mid-1 & i+mid+1 does not exceed array sizes`]
[**Note-4**: Similarly take care of even length palindromic substrings. Take `i-1,i` as center and check `valid(str, i-1-mid, i+mid) == true)`]

- We apply the Binary Search for each character as the center for the palindrome length guess till we exhaust the entire string.
  Time Complexity: $O(N \times \log_2(N))$, Space Complexity: $O(N)$

- **Approach #4:** Use Manacher's Algorithm. It is an algorithm which uses Dynamic Programming and is a non-trivial algorithm, just used to solve this one specific problem. It has a linear runtime complexity, and takes linear space.
  Best resources to read/learn about Manacher's Algorithm:
  I. **Manacher's Algorithm @Leetcode**.
  II. **Manacher's Algorithm @HackerEarth**.

9. Given two strings A_N & B_M, and two indices for each string, *i.e.,* `i`, `j` for A_N, and `k`, `l` for B_M, WAP to find whether `substring(A,i,j) == substring(B,k,l)`.
   Here, $0 \le i \le j < N$ and $0 \le k \le l < M$, where `N = A.size(); M = B.size();`
   Also, we have '`Q`' Queries of the form `i`, `j`, `k`, `l` given after the two strings.
   Example: A_N = "axyabcy"; B_M = "maqxbbcvayaxyzab";
   i = 0; j = 2; k = 10; l = 12;



   Here, `substring(A,0,2) == substring(B,10,12)` is `true`.
   Solutions:
   - **Approach #1:** Iterate on `substring(A,i,j)` along with the `substring(B,k,l)` for each query and check whether each character of those substrings are equal or not.
     Time Complexity: $O(Q \times N)$, Space Complexity: $O(1)$
   - **Approach #2:** We can use some insights from **Problem 8**, **Approach #3** where, we use Prefix Hash Sum & Suffix Hash Sum for a given string and then apply Binary Search on the string to find maximum possible palindrome size. In this problem, we don't need the usage of Suffix Hash Sum & Binary Search, because here, we are just trying to find whether the `substring(A,i,j) == substring(B,k,l)`, which just needs the calculation of Prefix Hash Sum(`PHS[]`). After that, we just need to check whether the `hash_value(A,i,j) == hash_value(B,k,l)` *i.e.,* check whether, `PSH[j]-PSH[i-1] == PSH[l]-PSH[k-1]` is `true` or not.
     [**Note**: We have to balance out the hash values of `A[i→j]` and `B[k→l]` with a factor of P^X depending on the values of `i` and `k`. P^X is the power of a large enough prime number (defined at Problem 7, Approach #2, Steps 1 & 2). Also, we have to carefully distribute modulo operator over subtraction]

     Time Complexity: $O(N + Q)$, Space Complexity: $O(N)$

Important References

1. **Documentation of C's `string.h` library**.
2. **Documentation on C++'s `std::string` library**.
3. **Documentation on Java's `String Class` and `StringBuilder Class`**.
4. **Documentation on Python's String Library**.
5. **ASCII Table**.