s.size() or s.length()

String Concatenation

```
string s1 = "new";
string s2 = "delhi";

s1.append(s2)

cout << s1; // newdelhi
cout << s2; // delhi
```

```
char s1[] = "new";
char s2[] = "delhi";
cout << s1 + s2;
```
error

```
string s1 = "new";
string s2 = "delhi";
cout << s1 + s2; //newdelhi
```

```
s1 = s1 + s2;
↳ s1 is modified
```

```
string s3 = s1 + s2;
cout << s3; //newdelhi
cout << s1; "new"
cout << s2; "delhi"
```

```
string s = "abc";
s.push_back('d');
cout << s; // abcd
s.pop_back();
cout << s; // abc
```

s[0] or s.front()

```
string s = "hello";
```

0	1	2	3	4	5
h	e	l	l	o	\0

```
int n = s.size(); // 5
cout << s[n-1]; // o
cout << s.back(); // o
```

String Comparation

```
string s1 = "abc";
string s2 = "adc";
```

s2 > s1

98 − 100 = −2

s1.compare(s2)

0	+ve	-ve
equal	s1 > s2	s1 < s2

* string obj i.e var. of string-type can be compared using relational operators

Searching in a String

string::npos

```
string s = "bcdabcdabc";
string substring = "abc";
if (s.find(substring) != string::npos) {
    // substring is found
} else {
    // substring is not found
}
```

s.find

"find"

0 1 2 3 4 5 6 7 8 9

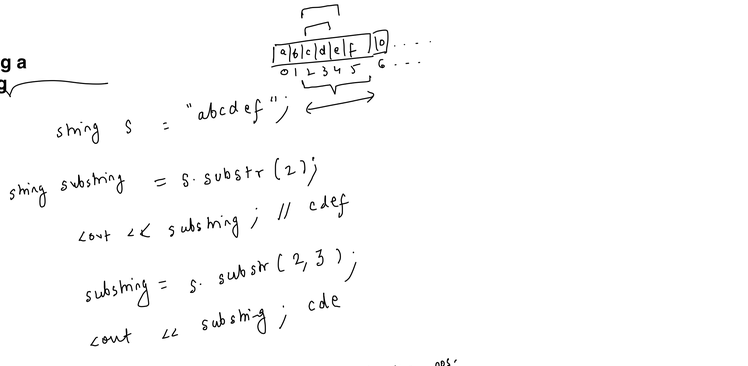
size_t

unsigned int

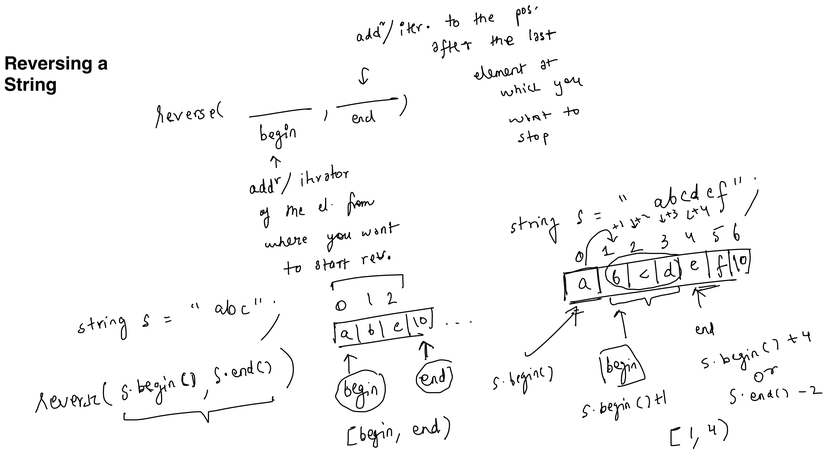
[0, ..., npos]

-2^{31} to $2^{31}-1$ (int)

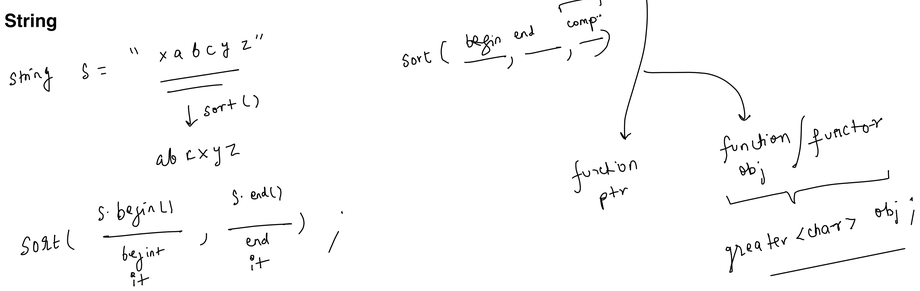
Extracting a Substring



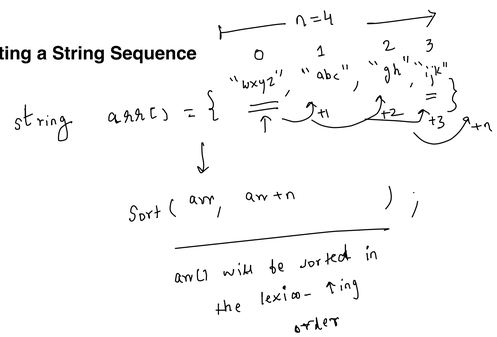
Reversing a String

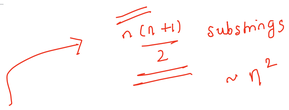


Sorting a String



Sorting a String Sequence





Generate Sub-Strings I

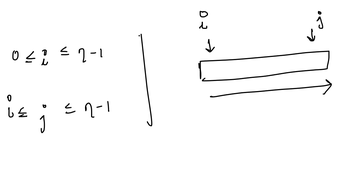
Given a string, design an algorithm to generate all of its **sub-strings**.

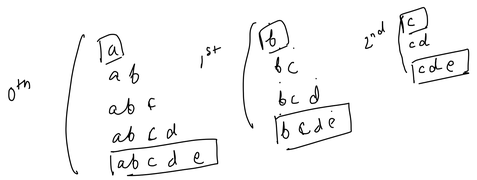
We define a **sub-string** of an string as a **contiguous** part of the given string.

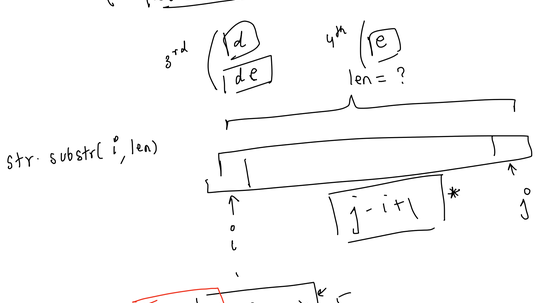
Example

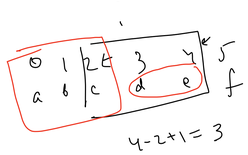
Input : "abcde"

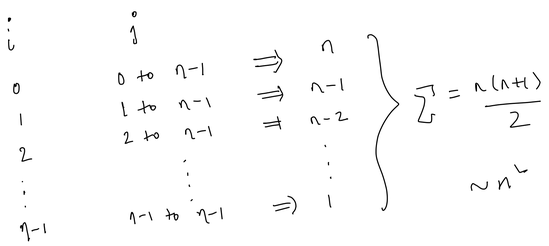
Output : "a", "ab", "abc", "abcd", "abcde"
"b", "bc", "bcd", "bcde"
"c", "cd", "cde"
"d", "de"
"e"











Generate Sub-Strings II

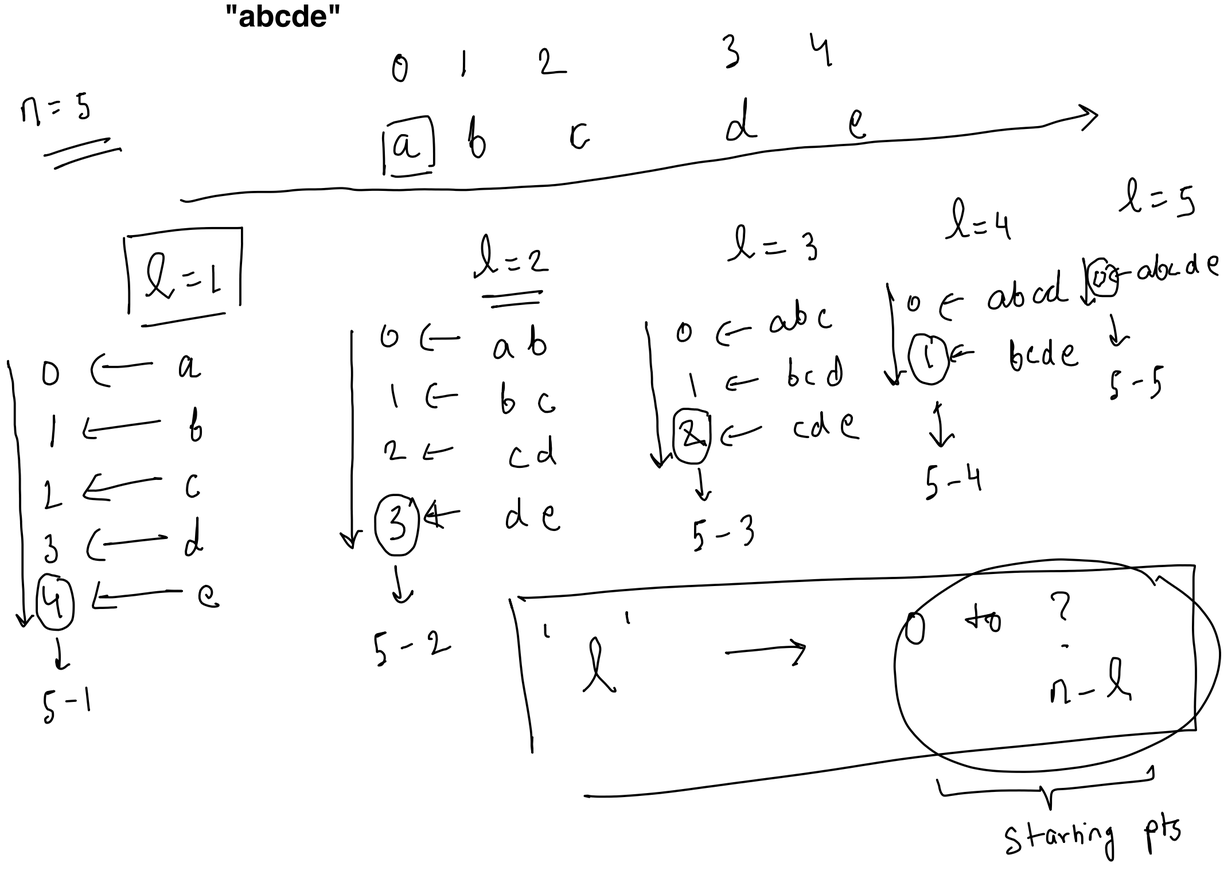
Given a string, design an algorithm to generate all of its **sub-strings** length-wise.

We define a **sub-string** of an string as a **contiguous** part of the given string.

Example

Input : "abcde"

Output : "a", "b", "c", "d", "e"
"ab", "bc", "cd", "de"
"abc", "bcd", "cde"
"abcd", "bcde"
"abcde"



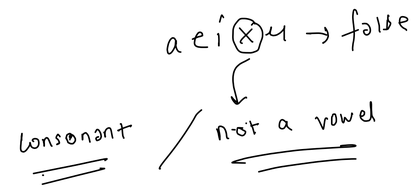
Check Good Strings

Given a string **str**, design an algorithm to check it is a **good string**.

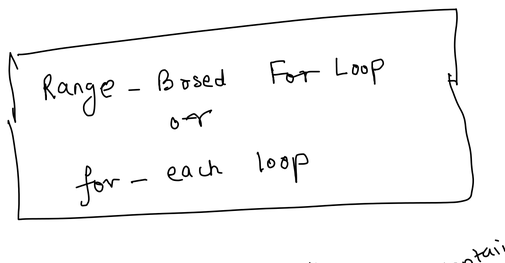
We define a good string as a string only contains **vowels** [a, e, i, o, u].

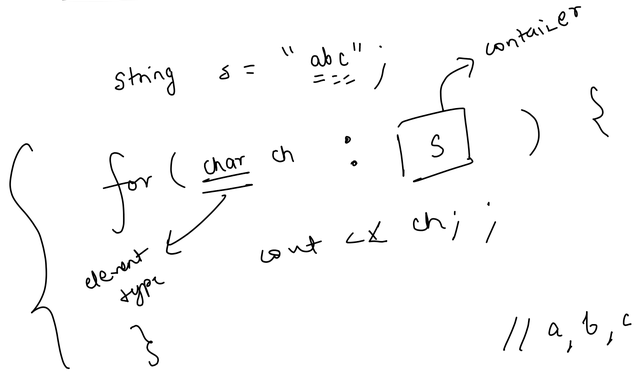
Example

Input : "uoiea"
Output : True



c





Longest Good Sub-String

Given a string **str**, design an algorithm to find the length of its **longest good sub-string**.

We define a good sub-string of **str** as a sub-string that only contains **vowels** [a, e, i, o, u].

Example

Input : "cbaeicdeiou"
Output : 4