Лабораторная №3.
Работа со строками

Класс String является основным классом, предназначенным для хранения и обработки строк символов. Для создания экземпляров класса String может быть использован один из следующих **конструкторов**:

String()
String(String str)
String(StringBuffer strbuf)
String(char[] arr)
String(char[] arr, int first, int count)

Первый из них создаёт пустую строку, второй и третий копируют содержимое объектов классов String и StringBuffer в созданный объект. Последние два конструктора позволяют создать строку на основе символьного массива или его части. Кроме того, любая объектная ссылка типа String может быть проинициализирована посредством присвоения ей **строкового литерала**, например:

```
String filename = "data.txt";
```

Особенностью класса String является то, что экземпляры этого класса **не могут быть изменены** после их создания. Однако это не создаёт ограничений для их использования, поскольку все методы, которые должны были бы изменять строку, просто создают **новую модифицированную строку**, оставляя исходную без изменений.

Поясним работу этого механизма на примере:

```
String s = "abcd";
s = s.toUpperCase();
```

Здесь метод toUpperCase() создаёт новую строку, содержащую последовательность символов "ABCD", и возвращает ссылку на эту строку, которая присваивается переменной s, старое значение переменной теряется. Исходная строка остаётся в неизменном виде и, поскольку на неё больше не осталось объектных ссылок, будет удалена сборщиком мусора.

Основные методы класса String

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char[] toCharArray()	Получение строки в виде символьного массива
boolean equals(String str)	Сравнение строк на равенство
boolean equalsIgnoreCase(String str)	Сравнение строк без учета регистра
int compareTo(String str)	Лексикографическое сравнение строк
int compareToIgnoreCase(String str)	Лексикографическое сравнение строк без учета регистра
boolean startsWith(String prefix)	Проверка, начинается ли строка с заданной подстроки
boolean endsWith(String suffix)	Проверка, заканчивается ли строка заданной подстрокой
int indexOf(String subStr)	Поиск первого вхождения подстроки в строке с начала строки/с заданной позиции
int indexOf(String subStr, **int** fromIndex)	
int lastIndexOf(String subStr)	Поиск последнего вхождения подстроки в строке с начала строки/с заданной позиции
int lastIndexOf(String subStr, **int** fromIndex)	

String substring(**int** beginIndex, **int** endIndex)	Получение подстроки (символ endIndex не входит в подстроку!)
String substring(**int** beginIndex)	Получение хвоста строки
String concat(String str)	Конкатенация строк
String toUpperCase()	Преобразование строки к верхнему/
String toLowerCase()	нижнему регистру
String trim()	Удаление ведущих и завершающих пробелов в строке
String replace(String target, String replacement)	Замена подстроки другой строкой
boolean matches(String regex)	Проверка строки на соответствие регулярному выражению
String replaceFirst(String regex, String replacement)	Замена первой подстроки/всех подстрок, соответствующих регулярному
String replaceAll(String regex, String replacement)	выражению, заданной подстрокой
String[] split(String regex)	Разбиение строки на подстроки (разделители задаются регулярным выражением)

Преобразование к строке

- Класс String является в некотором смысле исключительным классом в Java, поскольку любой тип данных может быть преобразован к нему.

- Для примитивных типов такое преобразование даёт их естественное строковое представление, для объектов вызывается метод toString(), определённый в классе Object и, следовательно, присутствующий в любом классе Java.

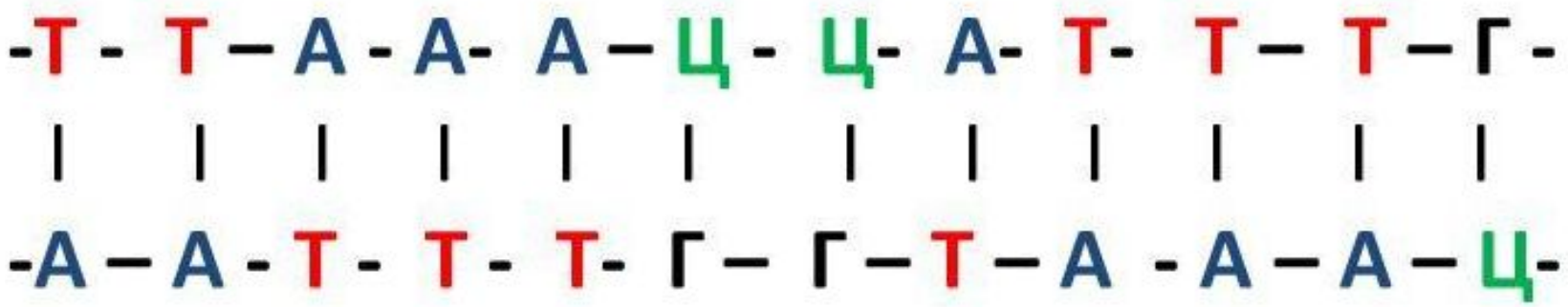
Конкатенация строк

- Для строк определена операция **конкатенации**, обозначаемая знаком +.
- Это бинарная операция, один из аргументов которой должен иметь тип String. Она осуществляет **автоматическое преобразование** другого аргумента к типу String (если это необходимо) и слияние полученных строк. Это единственный случай, когда преобразование к строке осуществляется неявно.

Алгоритм поиска наидлиннейшей общей подпоследовательности строк

Строки

- *Строка* – это последовательность символов из некоторого их набора.
- Текст может быть написан с помощью обычного алфавита или некоторого условного набора символов (пример – генетический код из 4-х «букв»).



Последовательности и подпоследовательности

- **Последовательность** представляет собой список элементов, в котором важен их порядок. Определенный элемент может появляться в последовательности несколько раз.
- В нашем случае последовательности – это строки символов.
- **Подпоследовательностью** Z строки X является строка X, возможно, с удаленными элементами.

- Например, если X является строкой нуклеотидов GAC, то он имеет восемь подпоследовательностей:

1) GAC (без удаленных символов),
2) GA (удален C),
3) GC (удален A),
4) AC (удален G),
5) G (удалены A и C),
6) A (удалены G и C),
7) C (удалены G и A) и
8) пустая строка (удалены все символы).

Общая подпоследовательность

- Если X и Y являются строками, то Z является *общей подпоследовательностью* X и Y, если она является подпоследовательностью обеих строк.
- Например, если X — это строка CATCGA, а Y является строкой GTACCGTCA, то CCA является общей подпоследовательностью X и Y, состоящей из трех символов. Однако это не наидлиннейшая общая подпоследовательность, поскольку есть общая подпоследовательность CTCA из четырех символов.

- Следует различать понятия подпоследовательности и подстроки: *подстрока* представляет собой подпоследовательность строки, в которой все символы выбираются из смежных позиций в строке. Для строки CATCGA подпоследовательность ATCG является подстрокой, в то время как подпоследовательность CTCA таковой не является.

Формулировка задачи

- **Задача**: для двух заданных строк X и Y найти наидлиннейшую общую подпоследовательность (НОП) этих строк (обозначим ее Z).
- **Простой способ решения**: перебор подпоследовательностей. Однако если длина X равна m, то она имеет 2^m подпоследовательностей, что дает экспоненциальную зависимость времени поиска от длины X.

Динамическое программирование

- **Требуется**: построить оптимальную подструктуру, т.е. оптимальное решение задачи должно состоять из оптимальных решений ее подзадач.
- **Оптимальная подструктура**: наидлиннейшая общая подпоследовательность двух строк содержит в себе наидлиннейшие общие подпоследовательности префиксов этих двух строк.

- Если X является строкой x_1, x_2, x_3… x_m, то ***i-м префиксом*** X является строка x_1, x_2, x_3… x_i, которую мы будем обозначать как X_i. Величина i должна быть в диапазоне от 0 до m, X_0 является пустой строкой.

- **Пример**: если строка X — CATCGA, то X_4 — CATC.

Оптимальная подструктура

Наидлиннейшая общая подпоследовательность двух строк содержит в себе наидлиннейшие общие подпоследовательности префиксов этих двух строк.

- Пусть две строки $X = x_1, x_2, x_3 \ldots x_m$ и $Y = y_1, y_2, y_3 \ldots y_n$ имеют некоторую наидлиннейшую общую подпоследовательность $Z = z_1, z_2, z_3 \ldots z_k$, где k может иметь значение от 0 до меньшего из значений m и n.
- Посмотрим на последние символы строк X и Y: x_m и y_n.

а) Если x_m и y_n совпадают, последний символ z_k строки Z должен быть таким же, как и этот символ. Остальная часть строки Z, т.е. $Z_{k-1} = z_1, z_2, z_3 \ldots z_{k-1}$, должна быть наидлиннейшей общей подпоследовательностью того, что осталось от X и Y, а именно — $X_{m-1} = x_1, x_2, x_3 \ldots x_{m-1}$ и $Y_{n-1} = y_1, y_2, y_3 \ldots y_{n-1}$.

б) Если x_m и y_n различны, то z_k может быть таким же, как x_m или y_n, но не оба. Кроме того, z_k может не совпадать ни с последним символом X, ни с последним символом Y.

Если z_k не совпадает с x_m, игнорируем последний символ X: Z должна быть НОП X_{m-1} и Y.

Аналогично, если z_k не совпадает с y_n, игнорируем последний символ Y: Z должна быть НОП X и Y_{n-1}.

Подзадачи

- Если x_m и y_n **совпадают**, то мы решаем только одну подзадачу — поиска НОП X_{m-1} и Y_{n-1} — а затем добавим к ней этот последний символ, чтобы получить НОП X и Y.
- Если x_m и y_n **не совпадают**, то нам надо решить две подзадачи — найти НОП X_{m-1} и Y, а также X и Y_{n-1} — и использовать большую из них в качестве НОП X и Y. Если их длины одинаковы, можно использовать любую из них — конкретный выбор не имеет значения.

Вычисление длины НОП

- Обозначим длину НОП префиксов X_i и Y_j как l[i,j].
- Длина НОП X и Y равна l[m,n].
- Индексы i и j начинаются с 0, т.е. l[0,j] = l[i,0] = 0.
- Когда i и j положительны, имеем два варианта:
- а) если $x_i = y_j$, то l[i,j] = l[i-l,j-l] + 1;
- б) если $x_i \neq y_j$, то l[i,j] равно наибольшему из значений l[i,j-1] и l[i-1,j].
- Для вычисления l[i,j], где i и j > 0, нам необходимо сначала вычислить записи l[i,i-1],

Процедура Compute-LCS-Table(X, Y).

Вход: X и Y – две строки длиной m и n соответственно.

Выход: массив l[0..m, 0..w]. Значение l[m,n] представляет собой длину наидлиннейшей общей подпоследовательности X и Y.

Шаги процедуры:

1. Пусть l[0..m, 0..n] представляет собой новый массив.
2. Для i = 0 до m:
    A. Установить l[i,0] = 0.
3. Для j = 0 до n:
    A. Установить l[0,j] = 0.

4. Для i = 1 до m:
   A. Для j = 1 до n:
      i. Если x_i совпадает с y_j, то установить l[i,j] = l[i-1, j-1] + 1.
      ii. В противном случае (x_i отличается от y_j) установить l[i,j] равным большему из значений l[i, j-1] и l[i-1, j].
      Если l[i, j-1] = l[i-1, j], конкретный выбор не имеет значения.
5. Вернуть массив l.

Пример: последовательности нуклеотидов

		j	0	1	2	3	4	5	6	7	8	9
		y_j		G	T	A	C	C	G	T	C	A
i	x_i	$l[i,j]$										
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Т.к. таблица содержит (m + 1)(n + 1) записей, время работы процедуры Compute-LCS-Table равно $\Theta(mn)$.

Определение самой НОП

- Это рекурсивная процедура, которая собирает искомую подпоследовательность в обратном порядке – с конца к началу.
- Когда она находит в X и Y одинаковые символы, она добавляет этот символ к концу строящейся наидлиннейшей общей подпоследовательности.

Процедура Assemble-LCS(X, Y, l, i, j).

Вход:

- X и Y – две строки,
- l – массив, заполненный процедурой Compute-LCS-Table,
- i и j – индексы как в строках X и У, так и в массиве l.

Выход: наидлиннейшая общая подпоследовательность X_i и Y_j.

Шаги процедуры:

1. Если l[i,j] = 0, вернуть пустую строку.

2. В противном случае (поскольку l[i,j] > 0 и i и j >0), если $x_i = y_j$, вернуть строку, образованную рекурсивным вызовом Assemble-LCS(X, Y, l, i-1, j-1) с добавлением к ней символа x_i (или y_j).
3. В противном случае ($x_i \neq y_j$), если l[i, j-1] > l[i-1, j], вернуть строку, образованную рекурсивным вызовом Assemble-LCS(X, Y, l, i, j-1).
4. В противном случае ($x_i \neq y_j$ и l[i, j-1] ≤ l[i-1, j]) вернуть строку, образованную рекурсивным вызовом Assemble-LCS(X, Y, l, i-1, j).

Пример

		j	0	1	2	3	4	5	6	7	8	9
		y_j		G	T	A	C	C	G	T	C	A
i	x_i	$l[i,j]$										
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Так как в каждом рекурсивном вызове происходит уменьшение на единицу либо значения i, либо значения j, либо обоих одновременно, время работы процедуры Assemble-LCS равно $O(m+n)$.

Задание

1. Создать две произвольных строки
2. Определить длину наидлиннейшей общей подпоследовательности строк
3. Зная длину, определить саму наидлиннейшую общую подпоследовательность строк