САОД
stl, метрика Левенштейна

А. Задорожный

2022

Контрольные вопросы

1. Для каких целей применяются объекты, учитывающие ссылки?
2. Можно ли сказать, что в .Net применяются объекты, учитывающие ссылки?
3. Что позволяют описывать шаблоны в C++?
4. Опишите основные свойства дерева суффиксов.
5. Что такое префикс-функция строки *s*? Как ее использует KMP?
6. Для какой задачи и чем хорош алгоритм Бойера-Мура?

Рассмотрены

- STL
  - Потоки
  - Строки
  - Средства отладки
  - Общие свойства контейнеров
  - Последовательные контейнеры:
    - Список
    - Динамический массив
    - Очереди и стеки
  - Ассоциативные контейнеры:
    - Множества
    - Словари (map)
    - Hash – таблицы (unordered_map, …_set)
  - Другие шаблоны
- Характеристики различных контейнеров
- Примеры применения контейнеров
- Умные указатели
- Хеширование
- Близость строк
  - Редакционное предписание

STL – стандартная библиотека (шаблонов)

STL - **S**tandard **T**emplate **L**ibrary

Набор абстрактных типов данных и алгоритмов.

Основное содержание в заголовочных файлах.

Для использования модуля библиотеки нужно подключить соответствующий заголовок.

Соглашение: Файлы не имеют расширения.

Потоки (STL)

- Обобщенный способ организации ввода/вывода.
  - Потоки ввода (istream),
  - Потоки вывода (ostream),
  - И потоки ввода-вывода (iostream).

Дают общий способ для работы с данными в файлах, памяти, поступающих по сети или из другого источника.

Моды потоков

- Потоки открываются в ***бинарной*** и ***текстовой*** моде

- В ***текстовой*** моде можно управлять форматированием. Например, представлением целых чисел.

```
cout << hex << i << endl;   //Шестнадцатеричный вывод
dec( cout );          // десятеричный по умолчанию
cout << i << endl;
oct( cout );          // восьмеричный по умолчанию
cout << i << endl;
cout << dec << i << endl; // десятеричный 1 раз
```

Специальные потоки

- потоки **cout**, **cin**, **cerr**, **clog** связаны только со стандартным вводом выводом.
- *Потоки стандартного ввода-вывода* ***перенаправляемы***. Т.е. при вызове программы можно указать куда/откуда (*на какой файл*) направить каждый из потоков.
- *Потоки имеют следующие номера*: 0 – стандартный ввод, 1 – стандартный вывод, 2 – стандартный лог и ошибки.
- По умолчанию все они направлены на консоль. Но, например,

  my_prog.exe 0<input.txt

  будет читать входные данные из файла “input.txt ”, а

  my_prog.exe 1>output.txt

  - выводить сообщения в файл “output.txt”

Строковые потоки

Часто потоки связываются *со строкой*.

Так удобно накапливать результат или разбирать содержимое строк.

Для работы со строковыми потоками нужно использовать заголовок <sstream> в котором определен класс string_stream.

Примеры:

```
   ostringstream os;
   os << 123 << " hello world";
//cout << os.str() << endl; //123 hello world

   istringstream is("456");
   is >> x;
//cout << x << endl; //456
```

Файловые потоки

Для работы с *файловыми потоками* нужно использовать заголовок
#include <fstream>

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){

        …

}
```

Файловые потоки

```
int main(){
	char fName[128];
	cout << "File name ? : "; cin.getline(fName, 127);
	cout << fName << endl;

	ifstream ifs;
	ifs.open(fName);		// (fName, ios_base::binary);
	if (ifs.is_open()) {
		char tmp[1024];
		ifs >> tmp;
		cout << tmp << endl;
		ifs.close();
	}
	else
	 cout << fName << " - does not exist" << endl;
	 return 0;
}
```

Файловые потоки
Пример в бинарной моде

```
int main(){
      char fName[128];
      cout << "File name ? : "; cin.getline(fName, 127);
      cout << fName << endl;

      char c = ‘0’;
      int n=5;
      double d = 7;
      ofstream ofs;
      ofs.open (fName, ios_base::binary);      //Открыт в бинарной моде

      if (ofs.is_open()) {
       ofs.put(c);
       os.write((char*)&n, sizeof(n));
              os.write((char*)&d, sizeof(d));
               ofs.close();
      }
      else
       cout << fName << " – can not create or open." << endl;
       return 0;
}
```

Строки

Стандартная библиотека предоставляет класс работы со строками – заголовок <string>.

Теперь вместо:

```
char fName[128];
cout << "File name ? : "; cin.getline(fName, 127);
cout << fName << endl;
```

можно использовать:

```
string fName;
cout << "File name ? : "; cin >> fName;
cout << fName << endl;
```

Строки
Минимальный набор

- **Конструкторы**: string(), string (const char *p) и string(const string &s) и др.
- **Операции**: length(), [], =, +, +=, <, >, …
- string **substr(**int Offset, int Length)
- int **find**(const string &s, int Offset) - *возвращает индекс ближайшего появления подстроки начиная с Offset и* ***-1*** *если нет.*
- Можно получить и указатель на z-строку:
  const char * c_str() const;

Всего несколько десятков методов.

Отладка
(assert)

Полезный инструмент разработки в STL – макрос **assert** (*подтвердить*)

Часто при реализации алгоритмов нужно убедиться, что в определенном месте алгоритма выполняется некоторое условие.

Можно ставить **if** и т.д., но во-первых сложно, во-вторых все проверки придется убирать при подготовке окончательной версии программы.

Макрос **assert** облегчает этот процесс.

assert(*условие*); // *например,* assert(i < 3)

- При компиляции в **дебаг**-моде в случае нарушения условия, будет выведено в поток ***cerr*** сообщение с номером строки, где не прошла проверка, и содержанием проверки. Приложение будет прервано!

- Если дебаг-мода не указана, то макрос окажется пустым! Ничего убирать не нужно!

Для применения нужно **#include** <cassert>

Примеры в упражнениях

Контейнеры в STL

Важная часть STL – шаблоны контейнеров.

Контейнеры – классы для хранения объектов других классов. Реализуют все базовые структуры данных:

deque (queue, stack**),** - очередь с двумя концами
list, - связный список
vector (*priority_queue*) – динамический массив
hash_map, hash_set, *hash_multimap, hash_multiset*
map, set, *multimap, multiset*

Общие свойства контейнеров

- Контейнеры STL можно параметризовать любыми типами, для которых определены операции =, == и <.
- При включении в контейнер всегда ***создается копия*** объекта.
  Нельзя включить в контейнер объект по ссылке.
- Универсальный способ для доступа к элементам любого контейнера – ***итераторы***.

Итераторы

Итераторы - типы, позволяющие двигаться по контейнеру

- <TYPE NAME>::iterator i = obj.begin();
- <TYPE NAME>::iterator i = obj.end();

Метафора итератора – указатель.

```
for(<TYPE NAME>::iterator i = obj.begin();
        i != obj.end();         i++)
{
// Делаем что хотели с элементами,
//используя *i;
}
```

Итераторы

Начиная с C++11 можно:

```
for (auto &v : obj) {
    // Делаем что хотели с элементами,
    //используя v;
}
```

Auto просит компилятор самостоятельно установить тип.

Обратите внимание на & - амперсанд перед v. Если объект большой, то он не будет копироваться. Если его нужно изменить, то по ссылке можно и изменить!

Как и ожидается, изменять сам контейнер (добавлять или удалять элементы) так нельзя!

Итераторы и .Net

- Как правило, использование итераторов позволяет более эффективно пройти по всему контейнеру, чем другие способы;

- Аналогией в .Net является применение оператора foreach

```
foreach(var p in Lst)
{
    //… обработка элемента контейнера p
}
```

Последовательные контейнеры

Последовательными называются контейнеры, в которых имеет смысл (определен) порядок элементов.

Очередь, массив, список – примеры таких контейнеров.

Последовательные контейнеры
Список

```
typedef list<string> LSTSTR;

                                    //Создание
LSTSTR lst1, lst2(5, “abc”);
LSTSTR lst3(lst2),  lst4(lst2.begin(), --lst2.end());

                                //Проверка на пустоту
cout << lst1.empty() << endl;  //true

                                //Количество элементов
cout << lst2.size() << endl;      //5
```

Список

```
//Добавление элементов
lst1.push_back("2");                    // {2}
lst1.push_front("1");                   // {1,2}
lst1.insert(--lst1.end(), "a");         // list is {1,a,2}
cout << lst1.size() << endl;            // 3

//Изменить элемент
*lst1.begin() = "3";                    // {3,a,2}

//Получить первый/последний
cout << lst1.front() << endl;           //3
cout << lst1.back() << endl;            //2
```

Список

```
//Присваивание
lst2 = lst1;

//Удаление элементов
lst1.remove("a");                    // {3,2}
lst1.erase(lst1.begin())             // {2}
lst1.erase(lst1.end())               // empty

//Отсортировать список методом sort.
lst2.sort()                          // {2, 3, a}
```

Sort "знает как устроен список"! Поэтому он более эффективно выполняет сортировку.

Последовательные контейнеры
Динамический массив

В принципе, другие контейнеры рассматриваем аналогично: как **создать**, какие **операции** и **свойства**, как получать **доступ к элементам**, как **добавлять** или **удалять**.

```
typedef vector<int> VINT;

                                    //Создание
VINT v1, v2(100);
VINT v3(v2.begin(), --v2.end());

                                    //Присваивание
v3 = v1;

                                    //Доступ к элементам
v2[i] = 10;
```

Последовательные контейнеры
Динамический массив

```
// добавляем элемент в конец массива
v2.push_back(11);
cout << v2.size() << endl;     //  101

// Можно и v2.insert(v2.begin(), x);
```

Отсортировать массив можно функцией STL sort.

```
//sort vector
sort(v.begin(), v.end());
```

Для сортировки указывается диапазон итераторов. Можно отсортировать только часть массива!

Функция sort размещена в модуле <algorithm>.

Последовательные контейнеры
Динамический массив

В дополнение к списку ***vector*** (как и ***string***) имеет 2 характеристики размера:

- **size()** – количество элементов (*есть и у списка*);
- **capacity ()** – количество элементов, которые может включать без расширения памяти;

Очевидно, **size() <= capacity()**!

При добавлении/удалении элементов *size* изменяется соответствующим образом.

За *capacity* отвечает сам контейнер.

Последовательные контейнеры
Очереди и стеки

<deque> - *Double ended queue*, двусторонняя очередь (сокращенно *Дек*).

Позволяет помещать и получать объекты в порядке поступления как в начало, так и в конец.

Основные операции:

push_back(<…>), push_front (<…>)

<…> pop_back(), <…> pop_front ()

Ограничивая функционал дека можно прийти к очереди (*queue*) и стеку (*stack*).

Ассоциативные контейнеры

В ассоциативных контейнерах порядок элементов не играет значения (*это вопрос реализации*).

Получаем или изменяем элементы **по ключу**! (*ассоциации - связи*)

В **set** и **map** допускают ТОЛЬКО уникальные ключи (*два элемента с одинаковым ключом добавить в них нельзя*).

Ассоциативные контейнеры
Множество

```
typedef set<string> SETSTR;
                                        //Создание
SETSTR s, s2;
                                        //Пустой
cout << s.empty() << endl;              //true
                                    //Добавление элементов
s.insert (“abc”); s.insert (“123”);     // s.size() == 2
s2 = s;                                 // s2 == s
                                    //Поиск элемента
SETSTR::iterator i = s.find(“abc”);   //i != s.end(), *i == “abc”
i = s.find(“efd”);                      //i == s.end()
                                    //Удаление элементов
s.erase (“abc”); s.erase(s.begin());    // s is empty
```

Пары

pair – удобная абстракция <*ключ, значение*>

```
pair<int, string> pr;  //ключом является int, значение – string.
pr.first = 1;
pr.second = “123”;
```

Можно

```
pair<int, string> pr1 = make_pair(2, “abc”);
```

Пары можно копировать, присваивать, сравнивать (сравнивается ключ и значение!)*

Ассоциативные контейнеры
Таблицы

Таблицы (map), содержат пары: <ключ>-<значение>

```
typedef map<string, int> STR2INT;
                                    //Создание
STR2INT m;
                                    //Пустой
cout << m.empty() << endl;          //true

                                    //Добавить и изменить
m.insert(STR2INT::value_type("a",1));   // m.size() == 1
m["b"] = 2;             // m.size() == 2
m["a"] = 3;             // m.size() == 2, т.к. ключ "a" уже был
```

Ассоциативные контейнеры
Таблицы

Таблицы (map), содержат пары:
<ключ>-<значение>

```
typedef map<string, int> STR2INT;
```

```
                                    //Поиск
STR2INT::iterator i = m.find("a");      // i != m.end(),
        // (*i).first == "a"    или i->first
        // (*i).second == 3  или i->second

m["b"] == 2;

int n = m.count("b");          // 0 или 1.   Для 'b' 1
```

Таблицы*

```
int n = m.count("a");                    // 1 или 0
STR2INT::iterator i = m.find("ab");    //i == m.end()

//Содержат пары!
for (STR2INT::iterator i= m.begin(); i != m.end(); i++)
      cout << i->first << "\t" << i->second << endl;

 //или!
for (auto &p: m)
      cout << p.first << "\t" << p.second << endl;

//Удаление элементов
m.erase("a");     m.erase(m.begin());     // m is empty
```

Ассоциативные контейнеры
Hash-таблицы

В STL имеются ассоциативные контейнеры даже более быстрые, чем *set* и *map*.

Это *unordered_set* и *unordered_map*.

Реализованы в одноименных заголовочных файлах.

По существу, это ***hash***-таблицы – сложность добавления и удаления по ключу *O(1)*.

Тогда как set и map используют упорядоченные деревья.

unordered_...Не упорядочены => Не требуют для ключа операции <

Достаточно сравнения ==.

Стандарт определяет ***hash***-функцию только для **числовых типов, строк и *bitset***.

Для других случаев нужно определяться с применением.

Ассоциативные контейнеры. Hash-таблицы.

Аналогично map

```
typedef unordered_map<string, int> U_STR2INT;
                                    //Создание
U_STR2INT u;
                                    //Пустой
cout << u.empty() << endl;          //true

                                //Добавить и изменить
u[“a”] = 1;          // u.size() == 1
u[“b”] = 2;          // u.size() == 2
u[“a”] = 3;          // u.size() == 2, т.к. ключ “a” уже был
```

см. дальше

Ассоциативные контейнеры. Hash-таблицы.

продолжение

```
//Поиск
U_STR2INT::iterator i = u.find("a"); // i != u.end(),
    // (*i).first == "a"    или i->first
    // (*i).second == 3  или i->second

u["b"] == 2;
int n = u.count("b");          // 0 или 1.   Для 'b' 1
```

Контейнеры, основанные на *hash* работают быстрее (сложность O(1)), но занимают больше памяти!

Прочие шаблоны

Иногда полезно иметь контейнер для хранения нескольких элементов с одинаковым ключом.

В STL это **multiset** и **multimap**.

Определены *multi*-контейнеры в тех же заголовочных файлах <set> и <map>.

Методы *multi*-контейнеров практически совпадают с методами **set** и **map**, за исключением того, что для **multimap** не определен оператор [] – выбора элемента по ключу.

Еще интересный контейнер **bitset**. Он позволяет работать с набором битов не равным 8, 16 или 32, выполняя над ними все побитовые операции.

см. ниже продолжение

Прочие шаблоны

STL имеет набор стандартных алгоритмов.

Описаны в заголовочном файле <**algorithm**> операции с контейнерами для их заполнения, копирования, поиска элементов или пар элементов, сортировки, изменения порядка на обратный, замены одних элементов на другие и пр.

В частности, метод *sort*, который применялся для вектора, расположен именно в <**algorithm**>.

Общий совет:

Прежде чем изобретать собственные контейнеры поищите полезные возможности стандартной

Аналоги контейнеров в .Net

- List<> - динамический массив, аналог vector
- Dictionary<,> - словарь, таблица, аналог map

- Аналог set в расширениях, например HashSet<>

Зачем столько контейнеров?

1. Сложность изменения списка O(1), а чтения элемента с заданным номером O(N);

2. Сложность изменения массива O(N), а сложность чтения элемента с заданным номером O(1);

3. Сложность изменения и чтения ассоциативных контейнеров O(Ln(N)).

4. Сложность изменения и чтения hash-контейнеров O(1), но безусловно выбор по ключу медленнее чем по номеру в vector.

Примеры применения контейнеров.

Задача. Найти N самых частых слов в тексте.

Задачу можно решить в 2 этапа:

1. Посчитать сколько раз каждое слово встречалось в тексте и
2. Найти N наиболее частых.

Примеры применения контейнеров.

```
typedef map<string, int> STR2INT;
```

Предположим, что все слова в верхнем регистре, т.е. “Hello” и “HELLO” будут поступать как “HELLO”.

```
 STR2INT m;              //Объявили контейнер
string s;
cin>>s;                  //Читаем слово*

 while (!cin.eof()){   //пока не конец
      m[s]++; // вставить, если не было и увеличить счетчик
       cin>>s;       //Читаем следующее слово
}

 cout << m.size() << endl; //количество различных слов
```

Примеры применения контейнеров

```
int N=16;
//подготовили множество пар <целое - строка>
typedef set<pair<int, string>> SET_OF_N_STR;
SET_OF_N_STR mostFriquent;
for (auto &p : m) {
	mostFriquent.insert(SET_OF_N_STR::value_type(p.second, p.first));
//Пары в set располагаются в порядке возрастания (меньшая впереди)
	if (mostFriquent.size() > N)
		mostFriquent.erase(mostFriquent.begin());
}
for (auto &p : mostFriquent)
	cout << p.second << "\t" << p.first << endl;
```

Контрольные вопросы

1. Почему контейнеры называются абстрактными типами данных?
2. Что такое последовательные и ассоциативные контейнеры? Приведите примеры из STL.
3. Что такое итератор в STL?
4. Какие операции должны быть определены для объектов, вставляемых в контейнеры STL?
5. Если в программе используется глобальный контейнер, а в него помещается локальный объект, не нарушится ли контейнер, когда локальный объект будет разрушен? Почему?
6. Дайте рекомендации, для каких целей хорошо использовать каждый из рассмотренных в лекции контейнеров. (string, vector, list, set, map, unordered_map, unordered_set)
7. Какой цели служит макрос assert? Как переводится термин? В чем его удобство?

Умные указатели (smart pointers)

Одна из распространенных ошибок в C++ связана с *new* ... Нет соответствующего *delete*

```
Test * p = new Test;
```

Smart pointers противостоят этой проблеме!

В STL один из видов – шаблон **shared_ptr<>**. Реализованы они в модуле **<memory>**.

```
shared_ptr<Test> pt(new Test);
```

Умные указатели (smart pointers)

В примерах будем использовать класс Test

```
class Test
{
public:
	int val;
	Test() {val = 0;}
};
```

Для наблюдения за динамическими объектами Test можно добавить деструктор и сделать конструктор и деструктор “говорящими”.

Т.е. воспользоваться приемом из Задания 5.

Умные указатели (smart pointers)

```
shared_ptr<Test> pt(new Test);
```

pt – *это* ***объект*** *типа shared_ptr<Test> ,* ***проинициализированный*** *указателем на созданный объект класса Test*.

Его можно использовать как ***указатель на динамически созданный объект****!*

```
cout << pt->val << endl;
```

Когда исчезнет последний умный указатель на динамически созданный объект, сам объект будет разрушен!

Умные указатели
Основные операции

```
shared_ptr<Test> pt(new Test);
```

Конструкторы копирования и инициализации,

= - *присваивание*,

->, * - доступ к членам Test и разыменование;

Имеются и операции сравнения ==, >= , …

bool – *преобразование к булевскому. 1, если инициализирован и 0, если нет!*

int use_count() – сколько объектов ссылается на управляемый объект

Метод **get()** – возвращает просто указатель, которым владеет shared_ptr.

Без достаточного опыта желательно избегать.

Но, например, для массива

```
shared_ptr<Test[]> pv(new Test[5]);
```

без **get()** получить элемент нельзя! Операции **[]** у **shared_ptr** нет!

Умные указатели демонстрация

```
shared_ptr<Test> pt(new Test);
cout << (bool)pt << endl;                  // 1
cout << pt.use_count() << endl;                 // 1
cout << (*pt).val << pt->val << endl;  // 00

shared_ptr<Test> ps;
cout << (bool)ps << endl;                  // 0
cout << pt.use_count() << endl;                 // 0
cout << (*ps).val << ps->val << endl; // ошибка

ps = pt;

cout << (bool)ps << endl;                  // 1
cout << pt.use_count() << endl;                 // 2
cout << (*pt).val << pt->val << endl;  // 00
```

Умные указатели
Заключительные замечания

Умные указатели ***shard_ptr*** реализуют подсчет ссылок на объект владения.

Имеются и другие виды умных указателей. В упражнениях.

Появились в такой форме в C++11. Стали официальным стандартом. Все еще подвижны и развиваются.

Терминология.* Умные указатели реализуют идиому **R**esource **A**cquisition **I**s **I**nitialization (**RAII**) – получение ресурса – это и инициализация.

“Получают ресурс при создании, и освобождают при разрушении, независимо от варианта завершения блока (исключение или нормальный выход)”.

Такая дисциплина делает программирование проще и эффективнее!

Поэтому в C++ не вводят блок **try-finally**! Умные указатели делают его ненужным!

Применяем умные указатели где можно!

Алгоритмы хеширования

Отмечали, что unordered_... являются hash-таблицами.

Хеширование применяется не только для построения *hash*-таблиц, а также: для хранения данных в БД, как контрольные суммы, в блок-чейн технологиях…

В STL реализованы качественные ***hash***-функции для string и чисел.

Для составных объектов, если имеются хеши отдельных полей, можно строить хеш, как h1^(h2<<1)^(<h3<<2)…

В заданиях рассматривается алгоритм Рабина-Карпа, который использует полиномиальный ***hash*** для поиска образцов в “тексте”.

Алгоритмы хеширования
в STL

Шаблон класса std::hash имеет единственный метод

size_t operator ()(const T& v)

Используется так:

```
hash<string> hs;
size_t h = hs("123qbsd");
```

Если нужно найти hash от пары строк, то можно действовать через побитовые операции:

```
size_t h = hs("123qbsd")^hs("qwerty")<<1;
```

Таким же образом можно поступать при вычислении ***hash*** любой последовательности (списка, массива и пр.).

Но в некоторых случаях удобно использовать собственные алгоритмы. В алгоритме Рабина-Карпа используются кольцевые hash'ы.

Кольцевые алгоритмы хеширования

Особенность кольцевых хэшей в том, что если известен хэш от последовательности **a;b;…;c**, можно просто вычислить хэш от **b;…;c;d**.

Популярный кольцевой алгоритм – полиномиальный хэш. Для строки s

$$h(s) = \sum_{i=1}^{|s|} s[i]x^{|s|-i} \ mod\ q$$

x – любое число (2 .. q-1, часто берут 2), а **q** – достаточно больше простое.

Легко увидеть что

$$h(s + c) = s[1]x^{|s|-1} + \sum_{i=2}^{|s|} s[i]x^{|s|-i} + c * x^{|s|}\ mod\ q$$

Т.е. достаточно убрать одно слагаемое и добавить другое!

Детали в заданиях по алгоритму Рабина-Карпа.

Контрольные вопросы

1. Что означает термин “*умные указатели*” (smart pointers)? Зачем они введены в STL?
2. Как называется в stl класс умных указателей, рассмотренных в лекции?
3. Следует ли создавать объекты умных указателей динамически?
4. Назовите основные операции над умными указателями?
5. Являются ли *умные указатели* указателями?
6. Какой механизм управления объектами использует *shared_ptr*?
7. В STL хорошо реализованы хеши для чисел и строк. А если наш объект более сложен. Какой алгоритм можно предложить для него?

Близость строк

Применения:

- задачи нечеткой логики;
- подсказка при проверке правописания;
- коррекция орфографических ошибок в поисковых машинах;
- сравнение (поиск различий) двух текстовых документов (Git);
- задачи биоинформатики и пр. …

Близость строк

В.И. Левенштейн , 1965 г.

Метрика (расстояние) – симметричная неотрицательная функция, удовлетворяющая “правилу треугольника”:

- ✔ M(a, b) = M(b, a);
- ✔ M(a, b) >=0, M(a, b) == 0, только если a и b совпадают;
- ✔ M(a, b) + M(b, c) >= M(a, c)

Расстояния между двумя текстами - ‘дистанция’ редактирования, т. е. наименьшее количество операций:

- замены,
- вставки или
- удаления

символа, которыми одна из строк может быть превращена в другую.

Метрика Дамерау – Левенштейна:

- добавлена операция “*транспозиция 2х- соседних символов*”.

Близость строк
“выравнивание”

Применительно к биоинформатике выявление различий между аминокислотами называется “*выравниванием*”.

Задача выглядит так. Имеется 2 последовательности **X** и **Y**, состоящих из {A,C,D,T} (*алфавит*).
Стоимость пропуска - **g**(*ap*), стоимость несовпадения – **m**(*ismatch*).

Добавляя пропуски в **X** и **Y**, найти “выравнивание” *минимальной стоимости*.

Пример

X = ATCDTCAD **Y** = ACDCTAD

Варианты выравниваний:

```
ATCD-TCAD        ATCD-TCAD
A-CDCT-AD        A-CDCT-AD
```

Штраф: 3g + m

Штраф: 3g

Очевидно, стоимость наиболее дешевого “выравнивания” и есть расстояние между строками. Но само “выравнивание” позволяет и найти различия (сходства) между строками.

Близость строк
Расстояние

Один из алгоритмов вычисления расстояния - Вагнер и Фишер (1974).

Для вычисления расстояния между строками A и B подготовим матрицу D (M+1xN+1) целых чисел, где M и N длины строк A и B соответственно. (*Строка A соответствует строкам матрицы D, а строка B – столбцам.*)

1. Заполним нулевую строку матрицы последовательными числами от 0 до N, а нулевой столбец – от 0 до M.

   Перейдем к заполнению оставшейся части матрицы по строкам.
2. Начиная с первого (*а не нулевого*) элемента строки *i*, последовательно, для каждого столбца j матрицы D:
   a. Вычислим штраф **p**. Если A[i-1] = B[j-1], то штраф **p** = 0, если нет, то 1.
   b. Вычислим D[i, j] как ***min***(D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] **+ p**).
3. Последовательно повторяем пункт 2 для всех строк матрицы D.
4. Возвращаем элемент $d_{M,N}$ (нижний правый элемент матрицы D).

Близость строк

Алгоритм заполнения D после инициализации можно описать рекуррентной формулой:

D[i, j] = min (D[i-1, j] + 1,
D[i, j-1] + 1,
D[i-1, j-1] + A[i-1]==B[j-1]? 0: 1)

“**Стоимость**” преобразования B в A (и наоборот) равна D[M,N] – правый нижний угол таблицы.

Здесь мы считали, что стоимость удаления, вставки и замены символов одинаковы и равны 1.

В зависимости от задач они могут меняться. В этом случае, в каждом из D[…, …] + 1 нужно добавлять не 1, а стоимость операции – неотрицательное целое.

Близость строк

Рассмотрим на примере слов: *но, оно*.

		О	Н
	0	1	2	3
Н	1	?		
О	2			

Вычислим элемент матрицы (1,1). Первые символы не совпали. Наименьшее из **(0, 0)+1**, (0, 1)+1, (1,0)+1. Результат равен **1**.

	0	1	2	3
1	1	?		
2				

Сравниваем второй символ ***оно*** и первый символ ***но***. Они совпали, так что вычисленное значение равно наименьшему из **(0, 1)**, (1, 1)+1, (0,2)+1. Результат равен 1.

…

Близость строк

Анимация алгоритма на примере слов:

но и ***оно***

		О	Н	О
	0	1	2	3
Н	1	1	1	2
О	2	1	2	1

Движемся дальше

Сравниваем символы

Вычисляем значение

Близость строк
динамическое программирование

Алгоритм *Вагнера-Фишера* - пример ***динамического программирования***.

Метод ***динамического программирования*** применим, когда задача разбивается на более простые подзадачи.

Подзадача:

Обозначим $\boldsymbol{x}_i$ и $\boldsymbol{y}_j$ префиксы строк ***x*** и ***y*** длиной *i* и *j* соответственно. Зная расстояние D($\boldsymbol{x}_i$, $\boldsymbol{y}_j$) найти расстояние D($\boldsymbol{x}_{i+1}$, $\boldsymbol{y}_j$).

Без такого разбиения можно было бы решать задачу тем или иным перебором.

Например, попытаться добиться максимального совпадения фрагментов строк сдвигами фрагментов в одном направлении (*вставкой пустых позиций*).

```
ОБЛАК-О
О---КНО
```

Но сложность такого перебора “по времени” была бы неприемлемой!

Близость строк
Редакционное предписание

После достраивания таблицы (красным курсивом)

		О	Н	О
	0	1	2	3
Н	1	1	*1*	*2*
О	2	*1*	*2*	***1***

Выясняем, что расстояние (наименьшее количество операций) между ‘ОНО’ и ‘НО’ равно 1 (нижний правый угол таблицы).

Но какие изменения должны быть сделаны из этого не следует!

Последовательность действий для преобразования одной строки в другую называется “*редакционным предписанием*”.

Редакционное предписание – последовательность операций, которая позволяет *первую строку* (по строкам, т.е. вертикальную) превратить *во вторую* (по столбцам, т.е. горизонтальную).

Будем далее обозначать символами ***i*** – вставку, ***d*** – удаление, ***r*** – замену и ‘***.***’ – отсутствие действий.

Близость строк
Редакционное предписание. Шаг 1.

Редакционное предписание – последовательность операций, которая позволяет *первую строку* (по строкам, т.е. вертикальную) превратить *во вторую* (по столбцам, т.е. горизонтальную).

p = “”

Начинаем с последнего элемента матрицы D: i=N-1;j=M-1;

1. Цикл
   a) Если D[i-1,j - 1] <= min(D[i-1, j], D[i, j-1]) , // мин. на диагонали
      i--;j--; если A[i]==B[j] то p +=‘.’ иначе p+=‘r’
   b) Иначе, если D[i,j-1] <= D[i - 1,j], // мин. слева
      j--; p+=‘i'
   c) Иначе, // мин. вверху
      i--; p+=‘d'
2. Конец цикла 1, если i==0 или j==0

см. следующий слайд …
v

Близость строк
Редакционное предписание. Шаг 2.

^

см. предыдущий слайд …

3. Теперь мы находимся
   или на верхней строке матрицы D
   или на первом столбце

Если j>0 предписание ***p*** нужно дополнить операциями ***i****(nsert)* **j** раз.

Если i>0 предписание ***p*** нужно дополнить операциями ***d****(elete)* ***i*** раз.

4. После чего предписание ***p*** нужно инвертировать.

Близость строк

Редакционное предписание. Пример.

Вернемся к примеру слов: *но, оно*.

		о	н	о
	*0**	*1**	2	3
н	1	1	1*	2
о	2	1	2	**1**

Путь построения предписания отмечен звездочками.
i=2, j=3. Наименьшее из значение соседей нижнего левого угла расположено по диагонали. Соответствующие буквы совпадают. => Ставим точку p = “.” и переходим по диагонали.

i=1, j=2. Опять минимум по диагонали, а буквы совпадают. Переходим по диагонали и добавляем точку p = “..”.

На этом i=0. Теперь просто добавляем (j=1) операций вставки
p = “..i”.

После инвертирования строки предписания p=“i..”. Т.е. чтобы привести “но” к “оно”, достаточно в первой позиции вставить символ из “оно”.

Близость строк

Редакционное предписание. Пояснение.

Длина предписания всегда равна длине наибольшей строки.

Алгоритм восстановления редакционного предписания приводит к однозначному результату.

В действительности, такое предписание может быть не единственным.

Однозначность достигается конкретной схемой проверки условий.

При другой согласованной схеме мы можем получить другой результат.

Близость строк

Можно обобщить алгоритм, считая, что стоимость различных операций различна.

Алгоритм при этом несколько модифицируется*.

Область применения шире чем поиск ближайших строк.

- Можно сравнивать любые последовательности. Например, файлы, тогда символом можно считать строку, а строкой сам файл (последовательность строк).

- Можно искать мутации в геноме.

Близость строк
Сложность

При больших объемах данных важна память и эффективность по времени исполнения.

Алгоритм Вагнера-Фишера - один из самых эффективных по времени. Но он требует памяти O(MxN).

Для анализа “строк” порядка 10^6 это потребует ~4*10^12 байт (и столько же операций).

Если для операций это приемлемо, то для *объема памяти уже нет*. Нужны модификации.

И такие модификации есть (*см. самостоятельно…*). В частности, широкое применяется алгоритм Хиршберга (Hirschberg, 1975).

Один из приемов оптимизации будет в Заданиях.

Контрольные вопросы

1. Что означает понятие “метрика”?
2. В чем заключается метрика Левенштейна для строк?
3. Назовите несколько примеров применения
   a) расстояния
   b) редакционного предписания для строк
4. Какова сложность алгоритма Вагнера-Фишера?
5. Каковы требования алгоритма к объему памяти в рассмотренной реализации?
6. Какова сложность построения редакционного предписания (расстояние уже вычислено)?