

#include <iostream>

#include <string>

#include <queue>

#include <unordered\_map>

using namespace std;

// Структура узла для дерева Хаффмана

struct Node

{

char ch; // Знак, хранимый в узле

int freq; // Частота знака

Node\* left, \* right; // Указатели на левого и правого потомка

};

// Функция для создания нового узла

Node\* getNode(char ch, int freq, Node\* left, Node\* right)

{

// Выделение памяти под новый узел

Node\* node = new Node();

// Присвоение значений атрибутам узла

node->ch = ch; // символ, который будет храниться в данном узле Хаффмана

node->freq = freq; // частоту (количество вхождений) символа, который будет храниться в данном узле Хаффмана

node->left = left; // указывает на узел, который представляет левого потомка текущего узла в дереве

node->right = right;

// Возвращение нового узла

return node;

}

// Структура для сравнения узлов в приоритетной очереди

struct comp

{

// Функция сравнения узлов по частоте

bool operator()(Node\* l, Node\* r)

{

return l->freq > r->freq; // Узлы с меньшей частотой имеют более высокий приоритет

}

};

// Функция для кодирования символов с использованием кодов Хаффмана

void encode(Node\* root, string str, unordered\_map<char, string>& huffmanCode)

{

// Проверяем, что узел существует

if (root == nullptr)

return;

// Если узел - лист, то сохраняем код Хаффмана для символа

if (!root->left && !root->right)

huffmanCode[root->ch] = str;

// Рекурсивно кодируем левое и правое поддеревья

// Для левого потомка добавляем '0' к текущему коду и рекурсивно вызываем encode

encode(root->left, str + "0", huffmanCode);

// Для правого потомка добавляем '1' к текущему коду и рекурсивно вызываем encode

encode(root->right, str + "1", huffmanCode);

}

void decode(Node\* root, int& index, string str)

{

// Проверяем, что узел не является нулевым указателем

if (root == nullptr)

{

return;

}

// Если узел - лист, то выводим символ и завершаем функцию

if (!root->left && !root->right)

{

cout << root->ch;

return;

}

// Увеличиваем индекс для перемещения по закодированной строке

index++;

// Рекурсивно декодируем в зависимости от текущего бита (0 или 1)

// Если текущий бит '0', рекурсивно декодируем левое поддерево

if (str[index] == '0')

decode(root->left, index, str);

// Если текущий бит '1', рекурсивно декодируем правое поддерево

else

decode(root->right, index, str);

}

// Функция для построения дерева Хаффмана и выполнения кодирования/декодирования

void buildHuffmanTree(string text)

{

// Хэш-таблица для хранения частот каждого символа

// хеш-таблица, параметризация, char - это тип ключа (символ), а int - тип значения (частота символа)

unordered\_map<char, int> freq;

// Цикл for используется для итерации по каждому символу в строке 'text'.

for (char ch : text)

{

// Увеличиваем частоту символа 'ch' в хэш-таблице 'freq' на 1.

freq[ch]++;

}

// Приоритетная очередь 'pq' используется для хранения узлов дерева Хаффмана в порядке их приоритета.

// Узлы с более высоким приоритетом (меньшей частотой) имеют больший приоритет.

priority\_queue<Node\*, vector<Node\*>, comp> pq;

// Заполняем приоритетную очередь листовыми узлами

// Цикл for используется для итерации по парам (ключ, значение) в хэш-таблице 'freq'.

for (auto pair : freq)

{

// pair.first - это символ (ключ) из хэш-таблицы 'freq'.

// pair.second - это частота (значение) символа 'pair.first' в тексте.

// Создаем новый узел дерева Хаффмана с использованием функции 'getNode' и добавляем его в приоритетную очередь 'pq'.

pq.push(getNode(pair.first, pair.second, nullptr, nullptr));

}

// Выводим частоту каждого символа

cout << "Частота символов" << endl;

for (auto pair : freq)

{

cout << pair.first << " " << pair.second << endl;

}

// Строим дерево Хаффмана, объединяя узлы с наименьшей частотой

while (pq.size() != 1)//выполняется до тех пор, пока размер приоритетной очереди pq не станет равным 1.

{

Node\* left = pq.top(); pq.pop();//Извлекаем узел с наименьшей частотой из приоритетной очереди и сохраняем его в переменной left

Node\* right = pq.top(); pq.pop();//Извлекаем второй узел с наименьшей частотой из приоритетной очереди и сохраняем его в переменной right

// Объединяем узлы и создаем новый внутренний узел

int sum = left->freq + right->freq;

pq.push(getNode('\0', sum, left, right));//Вставляем новый узел с наименьшей частотой обратно в приоритетную очередь pq

}

// Корень дерева Хаффмана

Node\* root = pq.top();

// Хэш-таблица для хранения кодов Хаффмана для каждого символа

// хеш-таблица, параметризация, char - это тип ключа (символ), а char - тип значения (частота символа)

unordered\_map<char, string> huffmanCode;

// Генерируем коды Хаффмана

encode(root, "", huffmanCode);

// Выводим сгенерированные коды Хаффмана

cout << "Коды Хаффмана :\n" << '\n';

for (auto pair : huffmanCode)

{

cout << "Символ: '" << pair.first << "' Код: " << pair.second << '\n';

}

// Выводим исходную строку

cout << "\nИсходная строка :\n" << text << '\n';

// Кодируем исходную строку с использованием кодов Хаффмана

string str = "";

for (char ch : text)

{

str += huffmanCode[ch];

}

// Выводим закодированную строку

cout << "\nЗакодированная строка :\n" << str << '\n';

int index = -1;

// Декодируем закодированную строку

cout << "\nДекодированная строка: \n";

while (index < (int)str.size() - 2)

{

decode(root, index, str);

}

}

// Основная функция

int main()

{

// Устанавливаем локаль для вывода на русском языке

setlocale(LC\_ALL, "RU");

// Входной текст

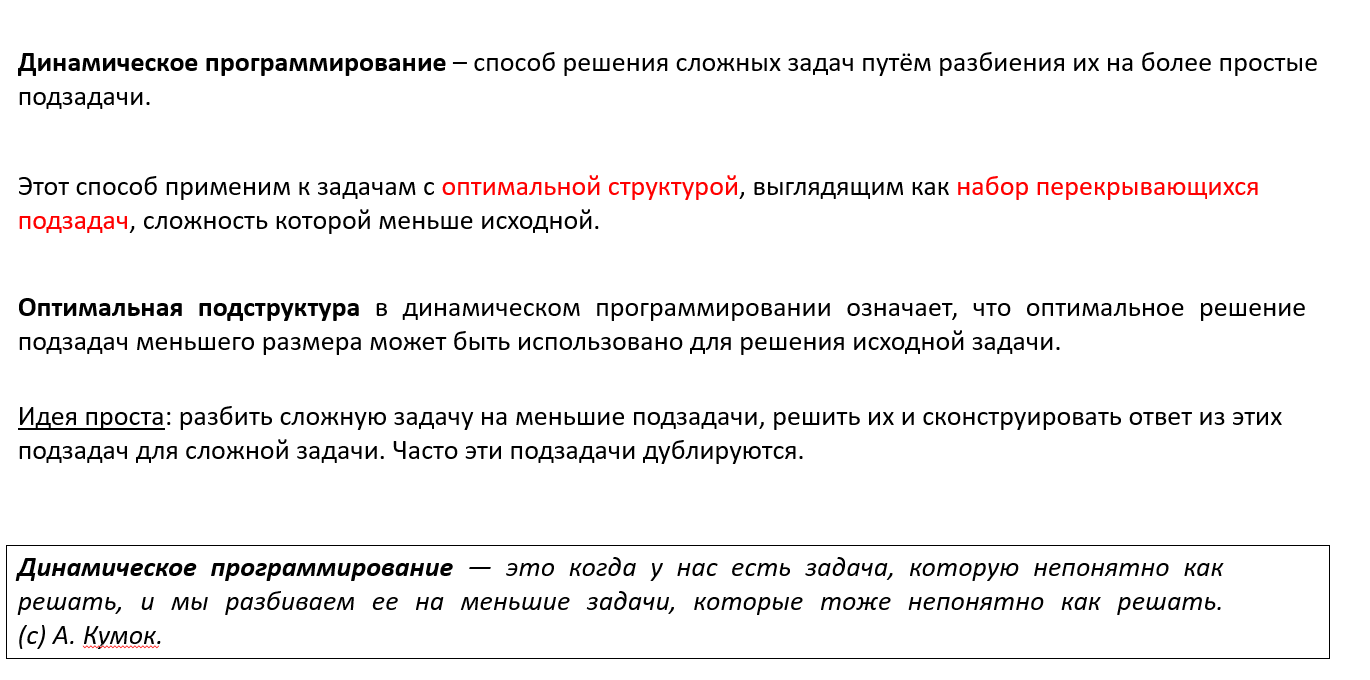
string text = "Сосновец Маша";

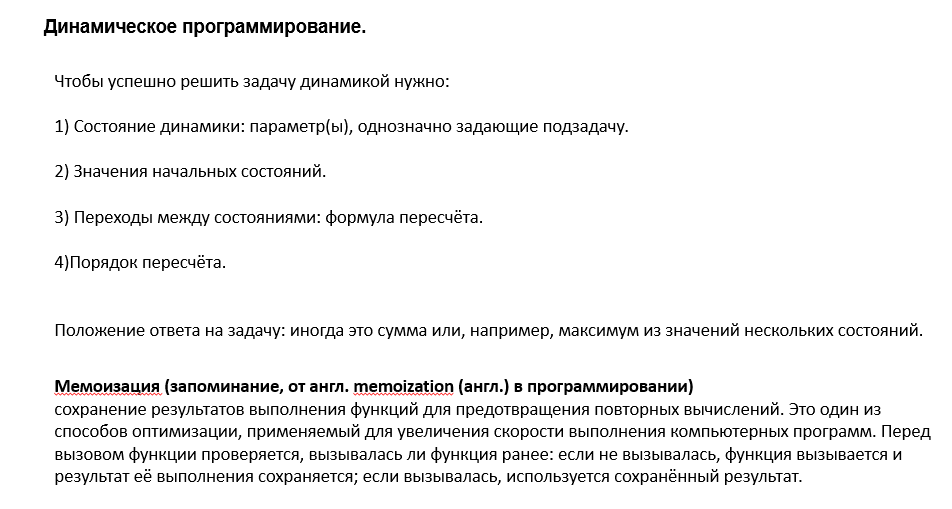
// Строим дерево Хаффмана и выполняем кодирование/декодирование

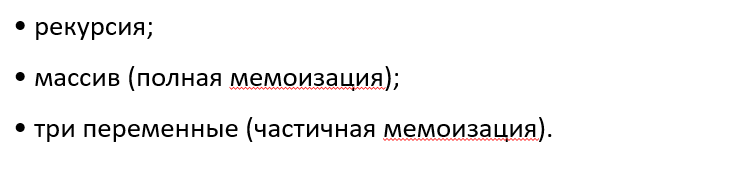
buildHuffmanTree(text);

return 0;

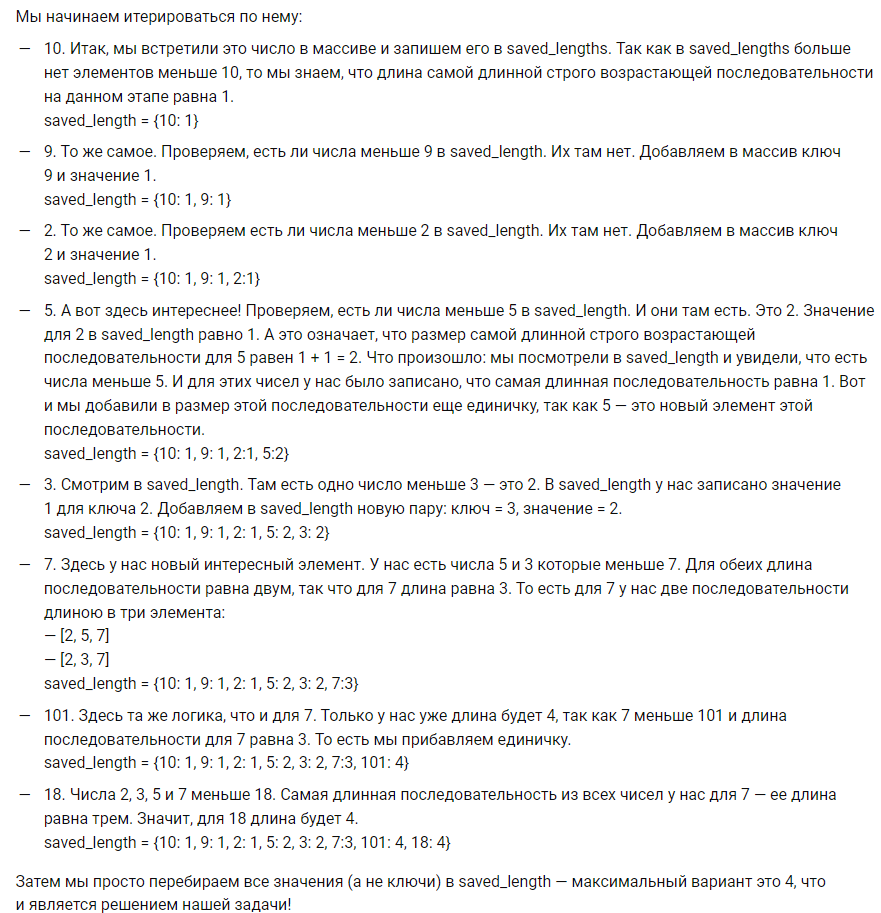
}







 [10,9,2,5,3,7,101,18]



#include <iostream>

using namespace std;

int main() {

setlocale(LC\_ALL, "Rus");

int n;

// Запрашиваем у пользователя длину последовательности

while (true)

{

cout << "Введите N (длина последовательности):" << endl;

cin >> n;

// Проверяем, была ли введена строка после числа (по несколько раз) и исправляем ошибку, если есть

if (cin.get() != (int)'\n')

{

cout << "\n---------- Данные введены неверно, попробуйте ещё раз --------------" << endl;

cin.clear(); // Очищаем флаг ошибки ввода

cin.ignore(32767, '\n'); // Игнорируем все символы до символа новой строки

continue; // Начнем ввод сначала

}

else {

break; // Выходим из цикла, если ввод корректен

}

}

int sequence[1000]; // Массив для хранения элементов последовательности, предполагается, что не будет более 1000 элементов

int maxLen[1000]; // Массив для хранения максимальной длины подпоследовательности

int next[1000]; // Массив для хранения индексов элементов подпоследовательности

// Запрашиваем у пользователя ввод элементов последовательности

cout << "Вводите массив:" << endl;

for (int i = 0; i < n; i++)

{

cout << i << "-й элемент: ";

cin >> sequence[i];

// Проверяем корректность ввода, если ввод некорректен, начинаем ввод сначала

if (cin.get() != (int)'\n')

{

cout << "\n---------- Данные введены неверно, начните сначала (с первого элемента массива) --------------" << endl;

cin.clear(); // Очищаем флаг ошибки ввода

cin.ignore(32767, '\n'); // Игнорируем все символы до символа новой строки

i = -1; // Начнем ввод сначала

}

}

// Алгоритм поиска максимальной возрастающей подпоследовательности

for (int i = n - 1; i >= 0; i--)

{

int maxNextLen = 0; // Инициализируем переменную для хранения максимальной длины подпоследовательности

int bestNext = -1; // Инициализируем переменную для хранения индекса следующего элемента в подпоследовательности

// Проходим через элементы, следующие за текущим элементом

for (int j = i + 1; j < n; j++)

{

// Проверяем, является ли элемент sequence[j] больше текущего элемента sequence[i]

if (sequence[j] > sequence[i])

{

// Если да, то проверяем, была ли найденная длина подпоследовательности для sequence[j] больше текущей максимальной

if (maxLen[j] > maxNextLen)

{

maxNextLen = maxLen[j]; // Обновляем максимальную длину

bestNext = j; // Обновляем индекс следующего элемента в подпоследовательности

}

}

}

next[i] = bestNext; // Сохраняем индекс следующего элемента для текущего элемента

maxLen[i] = maxNextLen + 1; // Максимальная длина текущей подпоследовательности

}

// Переменные для отслеживания максимальной длины подпоследовательности и индекса начала

int max = 0; // Максимальная длина

int bestStart = -1; // Индекс начала

// Находим индекс начала максимальной подпоследовательности

for (int i = 0; i < n; i++)

{

// Проверяем, если длина подпоследовательности, начинающейся с текущего элемента, больше текущей максимальной длины

if (maxLen[i] > max)

{

max = maxLen[i]; // Обновляем максимальную длину, если нашли более длинную подпоследовательность

bestStart = i; // Обновляем индекс начала максимальной подпоследовательности

}

}

// Выводим результат

cout << "Максимальная длина: " << max << endl;

for (int i = bestStart; i != -1; i = next[i])

cout << sequence[i] << " ";

return 0;

}