В билете будет 3 задачи, меня устраивает решение от 2+ задач.

Задачи разделены на 3 группы сложности (сами группы публиковаться не будут). Для каждого билета из каждой группы сложности случайным образом выбрана одна задача. Группа сложности 1 - 25 баллов, 2 - 35 баллов, 3 - 40 баллов.

**Для каждого алгоритма нужно предоставить теоретическое описание и практическую реализацию.**

Работа в семестре 60%, работа на экзамене 40%.

Разбаловка итоговой оценки: от 50 - 3, от 70 - 4, от 85 - 5.

1. Сортировка пузырьком - sort.sln
2. Сортировка вставками - sort.sln
3. Сортировка выбором - sort.sln
4. Шейкерная сортировка - sort.sln
5. Сортировка Шелла - sort.sln
6. Алгоритм бинарного поиска - search.sln
7. Быстрая сортировка - sort.sln
8. Внешняя сортировка слиянием(3 штуки) - externalSort.sln
9. Сортировка с помощью двоичного дерева - sort.sln
10. Поразрядная сортировка (radix sort) - sort.sln
11. Хэш-таблицы с разрешением коллизий методом цепочек – hashChain.sln
12. Хэш-таблицы с разрешением коллизий методом открытой адресации - hash.sln
13. ABC-сортировка (для строк) - sort.sln
14. Реализовать стек и базовые операции работы со стеком, с использованием собственного двусвязного списка - stack.sln
15. Реализовать очередь и базовые операции работы с очередью, с использованием собственного двусвязного списка - stack.sln
16. Алгоритм Крускала (поиска минимального остовного дерева) - search.sln
17. Алгоритм Прима (поиска минимального остовного дерева) - search.sln
18. Обход графа в глубину – graph.sln
19. Обход графа в ширину – graph.sln
20. Алгоритм Дейкстры (поиск кратчайшего пути) - search.sln

1. **Сортировка пузырьком** – sort.sln

Алгоритм заключается в циклических проходах по массиву (слева направо), за каждый проход элементы массива попарно сравниваются и, если их порядок не правильный, то меняем их местами. Обход массива повторяется до тех пор, пока массив не будет отсортирован.  
После первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и т.д. => не более чем после n итераций массив будет отсортирован. => **в худшем и среднем случае – O(n2), в лучшем случае – O(n).**

static void Main(string[] args) //1. сортировка пузырьком - bubbleSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("cортировка пузырьком");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", SortingBubble\_1(list)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(", ", SortingBubble\_1(list2)));

//ввод массива с консоли

//Console.Write("Введите элементы массива: ");

//var parts = Console.ReadLine().Split(new[] { " ", ",", ";" }, StringSplitOptions.RemoveEmptyEntries);

//var array = new int[parts.Length];

//for (int i = 0; i < parts.Length; i++)

//{

// array[i] = Convert.ToInt32(parts[i]);

//}

}

public static int[] SortingBubble\_1(int[] list)

{

for(int i = 0; i < list.Length; i++)

{

for (int j = i + 1; j < list.Length; j++)

{

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

{

int temp = list[i];

list[i] = list[j];

list[j] = temp;

}

}

}

return list;

}

1. **Сортировка вставками** – sort.sln
2. в начале работы упорядоченная часть пуста;
3. добавляем в отсортированную область первый элемент массива из неупорядоченных данных;
4. переходим к следующему элементу в не отсортированных данных, и находим ему правильную позицию в отсортированной части массива, тем самым мы расширяем область упорядоченных данных;
5. повторяем предыдущий шаг для всех оставшихся элементов.

**в среднем и худшем случае – O(n2), в лучшем – O(n).** Реализовывать алгоритм удобнее так: вместо вставки элементов в новый массив, менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

static void Main(string[] args) //2. сортировка вставками - insertionSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("сортировка вставками");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", InsertionSort\_2(list)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(", ", InsertionSort\_2(list2)));

}

public static int[] InsertionSort\_2(int[] list)

{

for (var i = 0; i < list.Length; i++)

{

var key = list[i];

var j = i;

while ((j > 0) && (list[j - 1] > key))

{

var temp = list[j - 1];

list[j - 1] = list[j];

list[j] = temp;

j--;

}

list[j] = key;

}

return list;

}

1. **Сортировка выбором** – sort.sln
2. Для начала определяем позицию минимального элемента массива;
3. Делаем обмен минимального элемента с элементом в начале массива. Получается, что первый элемент массива уже отсортирован;
4. Сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

После i-ой итерации первые i элементов будут стоять на своих местах: **O(n2)** в лучшем, среднем и худшем случае. Эту сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ быстрее.

static void Main(string[] args) //3. сортировка выбором - selectionSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("сортировка выбором");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", SelectionSort\_3(list)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(", ", SelectionSort\_3(list2)));

}

public static int[] SelectionSort\_3(int[] list, int currentIndex = 0)

{

if (currentIndex == list.Length)

return list;

var index = IndexOfMin(list, currentIndex);

if (index != currentIndex)

{

var t = list[index];

list[index] = list[currentIndex];

list[currentIndex] = t;

}

return SelectionSort\_3(list, currentIndex + 1);

}

static int IndexOfMin(int[] array, int n) //метод поиска позиции минимального элемента подмассива, начиная с позиции n

{

int result = n;

for (var i = n; i < array.Length; ++i)

{

if (array[i] < array[result])

{

result = i;

}

}

return result;

}

1. **Шейкерная сортировка** – sort.sln

Проанализировав алгоритм пузырьковой сортировки, можно заметить:  
1) если при обходе части массива не было обменов элементов, то эту часть можно исключить, так как она уже отсортирована;  
2) при проходе от конца массива к началу минимальное значение сдвигается на первую позицию, при этом максимальный элемент перемещается только на один индекс вправо. Поэтому будем идти не только слева направо, но и справа налево.

Исходя из приведенных идей, модифицируем сортировку пузырьком:  
1) на каждой итерации, фиксируем границы части массива в которой происходит обмен;  
2) массив обходиться поочередно от начала массива к концу и от конца к началу. При этом минимальный элемент перемещается в начало массива, а максимальный - в конец, после этого уменьшается рабочая область массива.  
**в худшем и среднем случае – O(n2), в лучшем случае – O(n) -** так же, как и у сортировки пузырьком, однако реальное время работы лучше.

public static int[] ShakerSort\_4(int[] list) //4. шейкерная сортировка - shakerSort

{

for (var i = 0; i < list.Length / 2; i++)

{

var swapFlag = false;

for (var j = i; j < list.Length - i - 1; j++) //проход слева направо

{

if (list[j] > list[j + 1])

{

Swap(ref list[j], ref list[j + 1]);

swapFlag = true;

}

}

for (var j = list.Length - 2 - i; j > i; j--) //проход справа налево

{

if (list[j - 1] > list[j])

{

Swap(ref list[j - 1], ref list[j]);

swapFlag = true;

}

}

if (!swapFlag) //если обменов не было выходим

{

break;

}

}

return list;

}

public static void Swap(ref int x, ref int y) //метод для обмена элементов массива

{

var t = x;

x = y;

y = t;

}

1. **Сортировка Шелла** – sort.sln

Алгоритм сортировки Шелла базируется на двух идеях:  
1) Сортировка вставками показывает лучшие результаты на практически упорядоченных массивах данных;  
2) Сортировка вставками неэффективна для смешанных данных, потому что за одну итерацию элементы смещаются только на одну позицию.

Для устранения недостатков алгоритма сортировки вставками, в сортировке Шелла осуществляется несколько сортировок вставками. При этом в каждой итерации сравниваются элементы, которые размещены на разных расстояниях один от другого, начиная с наиболее отдаленных (d = 1⁄2 длины массива) до сравнения соседних элементов (d = 1). Например, n=16, упорядочивание элементов n/2 - 8 групп по 2 элемента, потом 4 группы из 4 элементов, затем 2 группы из 8 элементов и т.д. на последнем шаге сортируются элементы сразу во всем массиве (1 группа из всех (16) элементов). На каждом шаге упорядочивание происходит используя метод сортировки вставками.  
**в худшем случае – O(n2), в лучшем – O(n log2 n)**

static void Main(string[] args) //5. сортировка Шелла - shellSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("сортировка Шелла");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", ShellSort\_5(list)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(", ", ShellSort\_5(list2)));

}

public static int[] ShellSort\_5(int[] list)

{

var d = list.Length / 2; //расстояние между элементами, которые сравниваются

while (d >= 1)

{

for (var i = d; i < list.Length; i++)

{

var j = i;

while ((j >= d) && (list[j - d] > list[j]))

{

var t = list[j];

list[j] = list[j - d];

list[j - d] = t;

j -= d;

}

}

d /= 2;

}

return list;

}

**Реализация Седжвика в сортировке Шелла**

Седжвик, предложил такую схему вычисления приращений:  
d[i] = 9\*2i - 9\*2i/2 + 1, если i четно  
d[i] = 8\*2i - 6\*2(i+1)/2 + 1, если i нечетно  
используя эту схему производительность алгоритма возрастает  
При расчете приращений по этому методу останавливаться следует на значении d[i-1], если 3\*d[i] > n. Обычно расчет начинается с нулевых значений i(=[0,1,2...]) и продолжается до такого i, когда 3\*d[i+1] > n, как было сказано ранее. Т.о. данная процедура рассчета запускается перед самой сортировкой Шелла и затем хранит полученную таблицу приращений в памяти, а алгоритм сортировки на каждом шаге к ней обращается за очередным значением. **O(n4/3)**

public static int[] ShellSort\_Sedjwik(int[] list, int size)

{

int inc, i, j, s;

var seq = new int[40];

s = Increment(seq, size); //вычисление последовательности приращений

while (s >= 0)

{

inc = seq[s--]; //сортировка вставками с инкрементами

for (i = inc; i < size; i++)

{

var temp = list[i];

for (j = i - inc; (j >= 0) && (list[j] > temp); j -= inc)

list[j + inc] = list[j];

list[j + inc] = temp;

}

}

return list;

}

public static int Increment(int[] inc, int size)

{

int p1 = 1, p2 = 1, p3 = 1, s = -1;

do

{

if (++s % 2 != 0) //нечет

{

inc[s] = 8 \* p1 - 6 \* p2 + 1;

}

else //четно

{

inc[s] = 9 \* p1 - 9 \* p3 + 1;

p2 \*= 2;

p3 \*= 2;

}

p1 \*= 2;

} while (3 \* inc[s] < size);

return s > 0 ? --s : 0;

}

1. **Алгоритм бинарного поиска** – search.sln
2. Определяем значение элемента в середине массива и сравниваем его с искомым;
3. Если они равны, возвращаем индекс середины;
4. Если значение элемента в середине массива больше искомого, то поиск продолжается в левой части массива, иначе в правой;
5. Проверяем не сошлись ли границы массива, если да - искомого значения нет, нет - переходим на первый шаг.  
   на каждой итерации количество элементов в области поиска массива уменьшается в 2  
   **O (log n)**

static void Main(string[] args) //6. Алгоритм бинарного поиска

{

Console.WriteLine("бинарный поиск(рекурсивная реализация)");

//Console.WriteLine("бинарный поиск(итеративная реализация)");

Console.Write("Введите элементы массива: ");

var s = Console.ReadLine().Split(new[] { " ", ",", ";" }, StringSplitOptions.RemoveEmptyEntries);

var list = new int[s.Length];

for (int i = 0; i < s.Length; i++)

{

list[i] = Convert.ToInt32(s[i]);

}

Array.Sort(list); //сортируем массив

Console.WriteLine("Упорядоченный массив: {0}", string.Join(", ", list));

while (true)

{

Console.Write("Введите искомое значение или -90 для выхода: ");

var value = Convert.ToInt32(Console.ReadLine());

if (value == -90)

{

break;

}

var searchResult = BinarySearch\_6\_1(list, value, 0, list.Length - 1);

//var searchResult = BinarySearch\_6\_2(list, value, 0, list.Length - 1);

if (searchResult < 0)

{

Console.WriteLine("Элемент со значением {0} не найден", value);

}

else

{

Console.WriteLine("Элемент найден. Индекс элемента со значением {0} равен {1}", value, searchResult);

}

}

}

//6.1 Алгоритм бинарного поиска (рекурсивный)

public static int BinarySearch\_6\_1(int[] list, int searchedValue, int first, int last)

{

if (first > last) //границы сошлись

{

return -1; //элемент не найден

}

var middle = (first + last) / 2; //средний индекс подмассива

var middleValue = list[middle]; //значение в средине подмассива

if (middleValue == searchedValue)

{

return middle;

}

else

{

if (middleValue > searchedValue) //рекурсивный вызов поиска для левого подмассива

{

return BinarySearch\_6\_1(list, searchedValue, first, middle - 1);

}

else //для правого подмассива

{

return BinarySearch\_6\_1(list, searchedValue, middle + 1, last);

}

}

}

//6.2 Алгоритм бинарного поиска (итеративный - с использованием цикла)

public static int BinarySearch\_6\_2(int[] list, int searchedValue, int left, int right)

{

while (left <= right) //пока не сошлись границы массива

{

var middle = (left + right) / 2; //индекс среднего элемента

if (searchedValue == list[middle])

{

return middle;

}

else if (searchedValue < list[middle])

{

right = middle - 1; //сужаем рабочую зону массива с правой стороны

}

else

{

left = middle + 1; //сужаем рабочую зону массива с левой стороны

}

}

return -1; //ничего не нашли

}

1. **Быстрая сортировка** – sort.sln

1) Выбираем опорный элемент массива, любой элемент, от этого не зависит правильность работы алгоритма;  
2) Разделяем массив на три части, в первой - элементы *меньше* опорного, во второй - *равные* опорному и в третьей - все элементы больше опорного;  
3) Рекурсивно выполняются предыдущие шаги, для подмассивов с меньшими и большими значениями, до тех пор, пока в них содержится больше одного элемента. **O(nlogn) в среднем и лучшем случае, O(n2).**

static void Main(string[] args) //7. быстрая сортировка - quickSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("быстрая сортировка");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", QuickSort\_7(list, 0, list.Length - 1)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(", ", QuickSort\_7(list2, 0, list2.Length - 1)));

}

public static int[] QuickSort\_7(int[] list, int minIndex, int maxIndex)

{

if(minIndex >= maxIndex)

{

return list;

}

int pivotIndex = Partition(list, minIndex, maxIndex);

QuickSort\_7(list, minIndex, pivotIndex - 1);

QuickSort\_7(list, pivotIndex + 1, maxIndex);

return list;

}

public static int Partition(int[] list, int minIndex, int maxIndex) //метод возвращающий индекс опорного элемента

{

var pivot = minIndex - 1;

for(int i = minIndex; i < maxIndex; i++)

{

if(list[i] < list[maxIndex])

{

pivot++;

Swap(ref list[pivot], ref list[i]);

}

}

pivot++;

Swap(ref list[pivot], ref list[maxIndex]);

return pivot;

}

public static void Swap(ref int x, ref int y) //метод для обмена элементов массива

{

var t = x;

x = y;

y = t;

}

1. **Внешняя сортировка слиянием** – externalSort.sln

Слияние – это объединение двух или более упорядоченных массивов в 1 упорядоченный.

**Прямое слияние**

* 1. Разбить имеющиеся элементы массива на пары и осуществить слияние элементов каждой пары, получив отсортированные цепочки длины 2 (кроме, быть может, одного элемента, для которого не нашлось пары).
  2. Разбить имеющиеся отсортированные цепочки на пары, и осуществить слияние цепочек каждой пары.
  3. Если число отсортированных цепочек больше единицы, перейти к шагу 2

**Естественное слияние**

При использовании метода прямого слияния не принимается во внимание то, что исходный файл может быть частично отсортированным, т.е. содержать серии (отсортированная подпоследовательность записей).

Метод естественного слияния основывается на распознавании серий при распределении и их использовании при последующем слиянии.

**естественное слияние:**

* нам понадобятся два сопутствующих файла и файл-вывод
* берём число, смотрим на прошлое число, если это число больше прошлого, кладём в тот же файл, если нет, во второй - меняем файл, когда возрастание сбивается
* записали все числа, и теперь в выходящий файл по очереди складываем эти подпоследовательности
* берем первый файл, пишем оттуда число, например файл 2 15 7 11 4, доп файлы - 0 и 1
* 0: 2 15, на 7 последовательность сбивается, переходим ко второму
* 1: 7 11, на 4 сбилось, пишем опять в первый
* Получили 0: 2 15 4 и 1: 7 11
* теперь переходим к выводу, мы смотрим в каждый файл и берём минимальное из первых
* 2 < 7 output: 2, 7 < 15 output: 2 7, 11 < 15 output: 2 7 11, файл 1 кончился => output: 2 7 11 15 4

Вспомогательные файлы 0 и 0сливаются в *файл* f, при этом серии образуют упорядоченные последовательности.  
Полученный *файл* f вновь обрабатывается, как указано в шагах 1 и 2. Повторяя шаги, сливаем упорядоченные серии до тех пор, пока не будет упорядочен целиком весь *файл*.

**Многопутевое слияние**

**многопутевое слияние** это как естественное только файлов больше

* файл будет 8 23 5 65 44 33 1 6
* в прямом слиянии я точно знаю сколько итераций – log2 n
* этот массив длины 8 => 2^3 степени => итераций будет три
* тут мы опять используем два файла  
  мы по очереди через один записываем числа в файлы  
  0: 8 5 44 1 и 1: 23 65 33 6  
  теперь кладём в изначальный файл по очереди беря сначала меньшее, потом большее число из четверок: 8 23 5 65 33 44 1 6
* теперь по очереди два через два пишем  
  0: 8 23 и 1: 5 65 => 0: 8 23 33 44 и 1: 5 65 1 6  
  опять в изначальный кладём парами сначала меньше, потом больше: 5 8 23 65 1 33 6 44
* и третья итерация - 4 через 4, берём четверку - 5 8 23 65 и вторая четвёрка - 1 6 33 44
* 0: 5 8 23 65 и 1: 1 6 33 44 и теперь всю восьмёрку - 1 5 6 8 23 33 44 65

1. **Сортировка с помощью двоичного дерева** – sort.sln
2. Элементы неотсортированного массива данных добавляются в двоичное дерево поиска;
3. Для получения отсортированного массива, производится обход бинарного дерева с переносом данных из дерева в результирующий массив.    
   **O(nlogn) в худшем, среднем и лучшем случае**

static void Main(string[] args) // 9. сортировка с помощью двоичного дерева - treeSort

{

var list = new int[] { 3, 5, 7, 1, 2, 8 };

var list2 = new int[] { 210, 43, 6, 9, 141, 3, 5, 7, 31, 2, 8 };

Console.WriteLine("сортировка с помощью двоичного дерева ");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

Console.WriteLine("oтсортированный массив: {0}", string.Join(" ", TreeSort(list)));

Console.WriteLine("начальный массив2: {0}", string.Join(", ", list2));

Console.WriteLine("oтсортированный массив2: {0}", string.Join(" ", TreeSort(list2)));

////или же создание рандом массива

//Console.Write("n = ");

//var n = int.Parse(Console.ReadLine());

//var a = new int[n];

//var random = new Random();

//for (int i = 0; i < a.Length; i++)

//{

// a[i] = random.Next(0, 100);

//}

//Console.WriteLine("начальный массив: {0}", string.Join(", ", list));

//Console.WriteLine("oтсортированный массив: {0}", string.Join(" ", TreeSort(list)));

}

**public static int[] TreeSort(int[] list)**

**{**

**var treeNode = new TreeNode(list[0]);**

**for (int i = 1; i < list.Length; i++)**

**{**

**treeNode.Insert(new TreeNode(list[i]));**

**}**

**return treeNode.Transform();**

**}**

public class TreeNode //9. простая реализация бинарного дерева

{

public TreeNode(int data) => Data = data;

public int Data { get; set; } //данные

public TreeNode Left { get; set; } //левая ветка дерева

public TreeNode Right { get; set; } //правая ветка дерева

public void Insert(TreeNode node) //рекурсивное добавление узла в дерево

{

if (node.Data < Data)

{

if (Left == null)

{

Left = node;

}

else

{

Left.Insert(node);

}

}

else

{

if (Right == null)

{

Right = node;

}

else

{

Right.Insert(node);

}

}

}

public int[] Transform(List<int> elements = null) //преобразование дерева в отсортированный массив

{

if (elements == null)

{

elements = new List<int>();

}

if (Left != null)

{

Left.Transform(elements);

}

elements.Add(Data);

if (Right != null)

{

Right.Transform(elements);

}

return elements.ToArray();

}

}

1. **Поразрядная сортировка (radix sort)** – sort.sln

Делаем сортировку цифр за цифрой, начиная с наименее значимой цифры до самой значимой цифры. Radix sort использует подсчет сортировки как подпрограмму для сортировки. **Алгоритм сортировки по радиксу:**  
**1)** Делаем (сортировку входной массив, используя сортировку с подсчетом по i-й цифре) для каждой цифры i, где i изменяется от наименее значимой цифры до самой значимой цифры. Например: 170, 45, 75, 90, 802, 24, 2, 66 => 17 **0** , 9 **0** , 80 **2** , **2** , 2 **4**, 4 **5** , 7**5**, 6 **6** => 8 **0** **2,** **2**, **2 4,** **4** **5, 6 6,** 1 **7** **0, 7 5, 9 0** => **2, 24, 45, 66, 75, 90, 1 70, 8 02  
O(n),** если считать, что числа фиксированного размера (а в противном случае нельзя было бы считать, что сравнение двух чисел выполняется за единицу времени)

public static int[] RadixSort\_10(int[] list, int n)

{

int m = GetMax(list, n);//Находим макс число, чтобы узнать количество цифр

for (int exp = 1; m / exp > 0; exp \*= 10) //Делаем подсчет для каждой цифры.

CountSort(list, n, exp);

return list;

}

public static int GetMax(int[] list, int n)

{

int max = list[0];

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

if (list[i] > max)

max = list[i];

return max;

}

public static void CountSort(int[] list, int n, int exp)

{

int[] output = new int[n]; //выходной массив

int[] count = new int[10];

//инициализация всех элементов count до 0

for (int i = 0; i < 10; i++)

count[i] = 0;

//Сохраняем количество вхождений в count[]

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

count[(list[i] / exp) % 10]++;

//Изменить count[i] так, чтобы count[i] теперь содержал фактический положение этой цифры в выходных данных []

for (int i = 1; i < 10; i++)

count[i] += count[i - 1];

for (int i = n - 1; i >= 0; i--) //Создаем выходной массив

{

output[count[(list[i] / exp) % 10] - 1] = list[i];

count[(list[i] / exp) % 10]--;

}

//Копируем выходной массив в list[]

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

list[i] = output[i];

}

1. **Хэш-таблицы с разрешением коллизий методом цепочек** - hashChain.sln

если хэш-функция выделяет один индекс сразу двум элементам, то храниться они будут в одном и том же индексе, но уже с помощью двусвязного списка.

Каждая ячейка массива является указателем на связный список (цепочку). Коллизии приводят к тому, что появляются цепочки длиной более одного элемента.  
**Добавление ключа. В лучшем случае, если не требуется проверять наличие дубля – O(1). В худшем – O(N)**  
1) Вычисляем значение хеш-функции добавляемого ключа – h.  
2) Находим A[h] – указатель на список ключей.  
3) Вставляем в начало списка (в конец списка дольше). Если запрещено дублировать ключи, то придется просмотреть весь список.  
**Удаление ключа. В лучшем – O(1). В худшем – O(N).**  
1) Вычисляем значение хеш-функции удаляемого ключа – h.  
2) Находим A[h] – указатель на список ключей.  
3) Ищем в списке удаляемый ключ и удаляем его.  
**Поиск ключа. В лучшем – O(1). В худшем – O(N).**1) Вычисляем значение хеш-функции ключа – h.  
2) Находим A[h] – указатель на список ключей.  
3) Ищем его в списке.  
**в среднем O(1 + α), где α – коэффициент заполнения таблицы.**

static void Main(string[] args)

{

DoForChainCollection();

}

public static void DoForChainCollection()

{

Random rnd = new Random();

double[] nums = new double[10000];

for (int i = 0; i < nums.Length; i++)

nums[i] = (rnd.NextDouble() \* 1000);

ChainHashCollection<double> hashCol = new ChainHashCollection<double>(1000);

foreach (double num in nums)

hashCol.Add(num);

Console.WriteLine();

Console.WriteLine(" Коэффициент заполнения: " + hashCol.GetLoadFactor());

Console.WriteLine(" Процент эффективности: " + Math.Round(100 \* hashCol.GetEffectiveness()) + "%");

Console.WriteLine(" Длина кратчайшей цепочки: " + hashCol.GetLenghtOfShortestList());

Console.WriteLine(" Длина длиннейшей цепочки: " + hashCol.GetLenghtOfLongestList());

/\*Console.Write("Длины: [");

foreach (int el in hashCol.GetChainLenghts())

Console.Write(el + " ,");

Console.Write("]");\*/

hashCol.GetElementsBy(2);

foreach (double num in nums)

if (!hashCol.Remove(num))

throw new Exception("Хеш не совпал для одного и того же ключа");

}

class ChainHashCollection<T>

{

List<T>[] elements;

public readonly int Size;

public int Count = 0;

public ChainHashCollection(int size)

{

Size = size;

elements = new List<T>[size];

}

int GetHash(T item)

{

double key = 0;

if (typeof(T) == typeof(double) || typeof(T) == typeof(float) || typeof(T) == typeof(int))

key = (double)(object)item;

else if (typeof(T) == typeof(string))

foreach (char el in item as string)

key += el;

else

key = item.GetHashCode();

//return (int)(key\*Size % Size);

//return (int)(Size \* ((key \* 0.82312454123) % 1));

int hash = (int)key;

for (int i = 1; i <= GetNumberLength(key); i++)

hash += (int)(key \* Math.Pow(10, i)) - (int)(key \* Math.Pow(10, i - 1)) \* 10;

return Math.Abs(hash % Size);

}

private static int GetNumberLength(double num)

{

string str = num.ToString();

if (!str.Contains('E'))

return str.Length;

else

return Math.Abs(Int32.Parse(str.Substring(str.IndexOf('E') + 1)));

}

public void Add(T item)

{

int hash = GetHash(item);

if (elements[hash] == null)

elements[hash] = new List<T> { item };

else

elements[hash].Add(item);

Count++;

}

public bool Remove(T item)

{

int hash = GetHash(item);

if (elements[hash] == null)

return false;

else

for (int i = 0; i < elements[hash].Count; i++)

if (elements[hash][i].Equals(item))

{

elements[hash].RemoveAt(i);

if (elements[hash].Count == 0)

elements[hash] = null;

Count--;

return true;

}

return false;

}

public List<T> GetElementsBy(int hash)

{

if (hash > 0 && hash < Size)

return elements[hash];

else

throw new Exception("Неверный хэш");

}

public List<T> GetSameElements(T item)

{

int hash = GetHash(item);

if (hash > 0 && hash < Size)

return elements[hash];

else

throw new Exception("Неверный хэш");

}

public bool Contains(T item)

{

int hash = GetHash(item);

if (elements[hash] != null)

{

foreach (T el in elements[GetHash(item)])

if (el.Equals(item))

return true;

return false;

}

else

{

return false;

}

}

public T[] GetElements()

{

T[] output = new T[Count];

int i = 0;

foreach (List<T> list in elements)

foreach (T el in list)

{

output[i] = el;

i++;

}

return output;

}

public void Clear()

{

for (int i = 0; i < elements.Length; i++)

{

elements[i] = null;

}

}

public double GetLoadFactor()

{

int loadElNum = 0;

foreach (List<T> el in elements)

if (el != null)

loadElNum++;

return Count / (double)loadElNum;

}

public double GetEffectiveness()

{

int perfectCount = (int)Math.Round(Count / (double)Size);

if (perfectCount == 0)

perfectCount = 1;

int difference = 0;

foreach (List<T> el in elements)

if (el != null)

if (el.Count - perfectCount > 0)

difference += el.Count - perfectCount;

return 1 - (double)difference / Count;

}

public int GetLenghtOfLongestList()

{

int maxLength = 0;

foreach (List<T> el in elements)

if (el != null && el.Count > maxLength)

maxLength = el.Count;

return maxLength;

}

public int GetLenghtOfShortestList()

{

int minLength = Count;

foreach (List<T> el in elements)

if (el == null)

minLength = 0;

else if (el.Count < minLength)

minLength = el.Count;

return minLength;

}

public int[] GetChainLenghts()

{

int[] output = new int[elements.Length];

for (int i = 0; i < elements.Length; i++)

output[i] = elements[i] != null ? elements[i].Count : 0;

return output;

}

}

1. **Хэш-таблицы с разрешением коллизий методом открытой адресации** - hash.sln

Все элементы хранятся непосредственно в массиве. Каждая запись в массиве содержит либо элемент, либо NULL. При поиске элемента систематически проверяем ячейки до тех пор, пока не найдем искомый элемент или не убедимся в его отсутствии.

**Вставка ключа. В лучшем случае, если не требуется проверять наличие дубля – O(1). В худшем – O(N)**1) Вычисляем значение хеш-функции ключа  
2) Систематически проверяем ячейки, начиная от A[h], до тех пор, пока не находим пустую ячейку.3) Помещаем вставляемый ключ в найденную ячейку. В п.2 поиск пустой ячейки выполняется в некоторой последовательности. Последовательность зависит от вставляемого в таблицу ключа. Для определения исследуемых ячеек расширим хеш-функцию, Важно, чтобы для каждого ключа k последовательность представляла собой перестановку множества, чтобы могли быть просмотрены все ячейки таблицы. **Поиск ключа.** Проверяем все ячейки, возвращаем значение или null, если поиск завершается неуспешно **В лучшем – O(1). В худшем – O(N)  
Удаление ключа.** Нельзя при удалении ключа из ячейки i просто пометить ее значением NULL. Иначе в последовательности проб для некоторого ключа (или некоторых) возникнет пустая ячейка, что приведет к неправильной работе алгоритма поиска. **В лучшем – O(1). В худшем – O(N)**  
**в среднем O(1 / 1 - a), где α – коэффициент заполнения таблицы.**

public class HashTableItem

{

public int Value { get; private set; }

public HashTableItem(int value) => Value = value;

public int GetValue() => Value;

public void SetValue(int value) => Value = value;

}

class OpenAddressHashTable

{

public HashTableItem[] Items = new HashTableItem[10000]; //все элементы будут храниться в этом массиве

public HashTableItem DeleteSimbol = new HashTableItem(-1);

public int hashParam = 13; //рандом число, для того чтобы точки например (1,3) и (3,1) не давали одинаковый хэш

public int capacity = 10000;

public List<int> ListLength = new List<int>();

public HashTableItem[] GetItems() => Items;

public OpenAddressHashTable()

{

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

Items[i] = null;

}

public int HashFunction(int value, int i) => (int)Math.Abs((value \* hashParam + i \* Math.PI \* value) % capacity);

//поиск элемента

public int? SearchItem(int value)

{

int i = 0;

int hash = HashFunction(value, i);

while (i < 10000) //проверяем все ячейки

{

if (Items[hash].GetValue() == value)

{

return Items[hash].GetValue();

}

i++;

hash = HashFunction(value, i);

}

return null; //поиск неуспешен

}

//вставка элемента

public void AddItem(int value)

{

var i = 0;

int hash = HashFunction(value, i);

while (Items[hash] != null && !Items[hash].Equals(DeleteSimbol)) //проверяем ячейки, до тех пор, пока не находим пустую ячейку

{

if (Items[hash].GetValue() == value)

{

Items[hash].SetValue(value); //Помещаем вставляемый элемент в найденную ячейку

break;

}

i++;

hash = HashFunction(value, i);

}

ListLength.Add(i);

Items[hash] = new HashTableItem(value);

}

//удаление элемента

public void DeleteItem(int value)

{

int i = 0;

int hash = HashFunction(value, i);

while (Items[hash] != null)

{

if (Items[hash].GetValue() == value)

{

Items[hash] = DeleteSimbol;

break;

}

i++;

hash = HashFunction(value, i);

}

}

//Подсчет длины самого длинного кластера в таблице

public int CalculatingClasterMaxLength()

{

int max = 0;

foreach (var e in ListLength)

{

if (e > max)

max = e;

}

return max;

}

{static void Main(string[] args)

{

var table = new OpenAddressHashTable();

DoingSmth(table);

}

public static void DoingSmth(OpenAddressHashTable table)

{

var list = new List<int>();

GenerationItems(list);

InsertAllItems(list, table);

var max = table.CalculatingClasterMaxLength();

var min = table.CalculatingClasterMinLength();

var average = table.CalculatingClasterAverageLength();

Console.WriteLine($"длина самого длинного кластера в таблице: {max}");

Console.WriteLine($"длина среднего кластера в таблице: {average}");

Console.WriteLine($"длина самого короткого кластера в таблице: {min}");

}

//Генерация 9000 элементов с различными ключами

public static void GenerationItems(List<int> list)

{

var random = new Random();

for (int i = 0; i < 9000; i++)

list.Add(random.Next());

}

//Вставить все элементы в хеш-таблицу

public static void InsertAllItems(List<int> list, OpenAddressHashTable table)

{

foreach (var e in list)

table.AddItem(e);

}

}

//методы из класса OpenAddressHashTable

//Подсчет длины среднего кластера в таблице

public int CalculatingClasterAverageLength()

{

var sum = 0;

int i = 0;

foreach (var e in ListLength)

{

if (e != 0)

{

sum += e;

i++;

}

}

Console.WriteLine(i);

return (sum / i);

}

//Подсчет длины самого min кластера в таблице

public int CalculatingClasterMinLength()

{

int min = capacity;

foreach (var e in ListLength)

{

if (e < min && e != 0)

min = e;

}

return min;

}

1. **ABC-сортировка (для строк)** – sort.sln
2. слова группируются по первой букве. В процессе сортировки, когда происходит переход от разряда к разряду значения в массиве меняются, отражая цепочки слов, начинающихся с одной буквы и имеющие одинаковые литеры в том или ином месте.
3. отмечаются индексы самого первого\последнего слова в списке, в котором в соответствующем разряде находится определённый символ. Отталкиваясь от этого слова, восстанавливается цепочка всех остальных лексем, имеющих в **i**-м разряде соответствующую букву.
4. Создавая и прослеживая подобные цепочки слов, рекурсивно продвигаясь от старших разрядов к младшим, формируются новые последовательности, упорядоченные в алфавитном порядке. Отсортировав слова на «A», затем сортируется «B» и т.д

сортирует не по всем разрядам. Процесс прекращается сразу как только список будет отсортирован, а не до тех пор пока не обработаются все разряды. Так же можно указать количество первых разрядов по которым произведётся упорядочивание, если старшинство по младшим разрядам не имеет особого значения.  
**средняя, худшая - O(k \* n)**, **где k — количество обрабатываемых разрядов.  
лучшая - O(n)**

static void Main(string[] args) //13. ABC - сортировка (для строк) - absSort

{

var list3 = new string[] { "Carmen", "Adela", "Beatrix", "Abbey", "Abigale", "Barbara", "Camalia", "Belinda", "Beckie" };

Console.WriteLine("ABC - сортировка");

Console.WriteLine("начальный массив: {0}", string.Join(", ", list3));

Console.WriteLine("oтсортированный массив: {0}", string.Join(", ", AbsSort\_13(list3)));

}

//13. ABC - сортировка (для строк) - absSort

public static string[] AbsSort\_13(string[] list) => new ABC\_Sorter(list).AbcSort();

public class ABC\_Sorter

{

public static int?[] Indexes;

public static List<int?[]> Level;

public static List<string> Result;

public static string[] List;

public ABC\_Sorter(string[] list)

{

List = list;

Indexes = new int?[List.Length];

Level = new List<int?[]> { new int?[26] };

Result = new List<string>();

}

public string[] AbcSort()

{

for (var i = 0; i < List.Length; i++)

{

var letter = char.ToUpper(List[i][0]) - 65;

Indexes[i] = Level[0][letter];

Level[0][letter] = i;

}

ClearLevel(0);

return Result.ToArray();

}

public static void ClearLevel(int depth)

{

if (Level.Count == depth + 1)

Level.Add(new int?[26]);

for (var i = 0; i < 26; i++)

{

if (Level[depth][i] != null)

{

var pos = Level[depth][i].GetValueOrDefault();

if (Indexes[pos] == null)

{

Result.Add(List[pos]);

Level[depth][i] = null;

}

else

{

MarkChain(pos, depth);

Level[depth][i] = null;

ClearLevel(depth + 1);

}

}

} }

public static void MarkChain(int pos, int depth)

{

while (true)

{

var nextPos = Indexes[pos];

if (depth + 1 >= List[pos].Length)

{

Result.Add(List[pos]);

Indexes[pos] = null;

}

else

{

int letter = char.ToUpper(List[pos][depth + 1]) - 65;

Indexes[pos] = Level[depth + 1][letter];

Level[depth + 1][letter] = pos; }

if (nextPos == null)

break;

else

pos = nextPos.GetValueOrDefault();

}

} }

1. **Реализовать стек и базовые операции работы со стеком, с использованием собственного двусвязного списка** – stack.sln

**Почти все операции, а может и все - O(1) , но если они проходят через весь список , то - O(n)**

class Item<T>

{

public T Value { get; set; }

public Item<T> Next { get; set; }

public Item<T> Previous { get; set; }

public Item(T value, Item<T> next, Item<T> previous)

{

Value = value;

Next = next;

Previous = previous;

}

}

class DoublyLinkedList<T> : IEnumerable<T>

{

Item<T> Head; //первый

Item<T> Tail; //последний

int CountOfItems; //количество элементов

public int Count() => CountOfItems;

public bool IsEmptyList() => CountOfItems == 0;

public void PlaceLast(T value) //добавить элемент

{

Item<T> item = new Item<T>(value, null, null);

if (Head == null)

Head = Tail = item;

else

{

Tail.Next = item;

item.Previous = Tail;

Tail = item;

}

CountOfItems++;

}

public T DeleteLastItem()

{

if (Head == null)

throw new InvalidOperationException();

var value = Tail.Value;

Tail = Tail.Previous;

Tail.Next = null;

CountOfItems--;

return value;

}

public T ViewLastItem() => Tail.Value;

public void ClearList() //очистить список

{

Head = null;

Tail = null;

CountOfItems = 0;

}

public bool ContainsItem(T value) //проверить наличие элемента и вернуть true при его наличии

{

Item<T> current = Head;

while (current != null)

{

if (current.Value.Equals(value))

return true;

current = current.Next;

}

return false;

}

public override string ToString()

{

StringBuilder sb = new StringBuilder();

Item<T> item = Head;

while (item != Tail)

{

sb.Append(item.Value.ToString() + ", ");

item = item.Next;

}

sb.Append(item.Value.ToString());

return sb.ToString();

}

IEnumerator IEnumerable.GetEnumerator() => ((IEnumerable)this).GetEnumerator();

IEnumerator<T> IEnumerable<T>.GetEnumerator()

{

Item<T> current = Head;

while (current != null)

{

yield return current.Value;

current = current.Next;

}

}

public IEnumerable<T> BackEnumerator()

{

Item<T> current = Tail;

while (current != null)

{

yield return current.Value;

current = current.Previous;

}

}

}

**class Stack<T> : DoublyLinkedList<T> //добавить в конец, удалить с конца**

**{**

**public void Push(T value) => PlaceLast(value); //добавить элемент в конец стека**

**public T Pop() => DeleteLastItem(); //удалить и посмотреть последний элемент стека**

**public T Peek() => ViewLastItem(); //посмотреть последний элемент стека без его удаления**

**public bool IsEmpty() => IsEmptyList();**

**public int Size() => Count();**

**public void Print() => Console.WriteLine("элементы стека: " + ToString());**

**public void Clear() => ClearList(); //очистить стек**

**public bool Contains(T value) => ContainsItem(value);**

**}**

static void Main(string[] args)

{

//stack

Console.WriteLine("stack");

var list = new Stack<int>();

list.Push(1);

list.Push(3);

list.Push(5);

list.Print();

Console.WriteLine("размер стека = " + list.Size());

Console.WriteLine("последний элемент: " + list.Peek());

list.Pop();

Console.WriteLine("удалили последний элемент:");

list.Print();

Console.WriteLine("теперь размер стека = " + list.Size());

Console.WriteLine("stack is empty: " + list.IsEmpty());

}

1. **Реализовать очередь и базовые операции работы с очередью, с использованием собственного двусвязного списка** – stack.sln

**Почти все операции, а может и все - O(1) , но если они проходят через весь список , то - O(n)**

class Item<T>

{

public T Value { get; set; }

public Item<T> Next { get; set; }

public Item<T> Previous { get; set; }

public Item(T value, Item<T> next, Item<T> previous)

{

Value = value;

Next = next;

Previous = previous;

}

}

class DoublyLinkedList<T> : IEnumerable<T>

{

Item<T> Head; //первый

Item<T> Tail; //последний

int CountOfItems; //количество элементов

public int Count() => CountOfItems;

public bool IsEmptyList() => CountOfItems == 0;

public void PlaceLast(T value) //добавить элемент

{

Item<T> item = new Item<T>(value, null, null);

if (Head == null)

Head = Tail = item;

else

{

Tail.Next = item;

item.Previous = Tail;

Tail = item;

}

CountOfItems++;

}

public T DeleteFirstItem()

{

if (Head == null)

throw new InvalidOperationException();

var value = Head.Value;

Head = Head.Next; //переустанавливаем head

if (Head == null)

Tail = null;

CountOfItems--;

return value;

}

public T ViewFirstItem() => Head.Value;

public void ClearList() //очистить список

{

Head = null;

Tail = null;

CountOfItems = 0;

}

public bool ContainsItem(T value) //проверить наличие элемента и вернуть true при его наличии

{

Item<T> current = Head;

while (current != null)

{

if (current.Value.Equals(value))

return true;

current = current.Next;

}

return false;

}

public override string ToString()

{

StringBuilder sb = new StringBuilder();

Item<T> item = Head;

while (item != Tail)

{

sb.Append(item.Value.ToString() + ", ");

item = item.Next;

}

sb.Append(item.Value.ToString());

return sb.ToString();

}

IEnumerator IEnumerable.GetEnumerator() => ((IEnumerable)this).GetEnumerator();

IEnumerator<T> IEnumerable<T>.GetEnumerator()

{

Item<T> current = Head;

while (current != null)

{

yield return current.Value;

current = current.Next;

}

}

public IEnumerable<T> BackEnumerator()

{

Item<T> current = Tail;

while (current != null)

{

yield return current.Value;

current = current.Previous;

}

}

}

**class Queue<T> : DoublyLinkedList<T> //добавить в конец, удалить из начала**

**{**

**public void Enqueue(T value) => PlaceLast(value); //добавить элемент в конец очереди**

**public T Dequeue() => DeleteFirstItem(); //удалить и посмотреть первый элемент очереди**

**public T Peek() => ViewFirstItem(); //посмотреть первый элемент очереди без его удаления**

**public bool IsEmpty() => IsEmptyList();**

**public int Size() => Count();**

**public void Print() => Console.WriteLine("элементы очереди: " + ToString());**

**public void Clear() => ClearList(); //очистить очередь**

**public bool Contains(T value) => ContainsItem(value);**

**}**

static void Main(string[] args)

{

//queue

Console.WriteLine("queue");

var list2 = new Queue<int>();

list2.Enqueue(2);

list2.Enqueue(4);

list2.Enqueue(6);

list2.Print();

Console.WriteLine("размер очереди = " + list2.Size());

Console.WriteLine("первый элемент: " + list2.Peek());

list2.Dequeue();

Console.WriteLine("удалили первый элемент:");

list2.Print();

Console.WriteLine("теперь размер очереди = " + list2.Size());

Console.WriteLine("queue is empty: " + list2.IsEmpty());

}

1. **Алгоритм Крускала (поиска минимального остовного дерева)** - search.sln

**Задача: найти такой связный ациклический (без циклов) подграф *T* ⊂ *G*, содержащего все вершины, что суммарный вес его ребер будет минимален.**

Алгоритм Крускала объединяет вершины графа в несколько связных компонент, каждая из которых является деревом.

1. Первоначально из графа удаляются все ребра. Ребра сортируются по возрастанию весов.
2. На каждом шаге из всех ребер, соединяющих вершины из различных компонент связности, выбирается ребро с наименьшим весом. (Ребра последовательно, по возрастанию их весов, включаются в подграф только в том случае, если данное ребро соединяет две разные компоненты связности, одним из которых является наш подграф. То есть, на каждом шаге добавляется минимальное по весу ребро, один конец которого содержится в нашем подграфе, а другой - еще нет.)
3. Алгоритм заканчивают работу, когда все вершины будут объединены в одно множество, при этом оставшиеся ребра не включаются в остовное дерево.

static void Main(string[] args) //16. Алгоритм Крускала

{

var graph = new Graph(4);

graph.Connect(0, 1, 1);

graph.Connect(0, 2, 2);

graph.Connect(0, 3, 6);

graph.Connect(1, 3, 4);

graph.Connect(2, 3, 2);

foreach (var e in Kruskal(graph))

{

Console.WriteLine(e.Weight);

}

}

public static IEnumerable<Edge> Kruskal(Graph graph)

{

var tree = new List<Edge>(); //пустое дерево

foreach (var edge in graph.Edges.Where(t => t.Weight > 0).OrderBy(x => x.Weight))

{

tree.Add(edge);

var temp = MakeGraph(tree);

if (HasCycle(temp))

tree.Remove(edge);

}

return tree;

}

public class Edge //16, 17 ребро

{

public Node From;

public Node To;

public int Weight;

public Edge(Node first, Node second, int weight = 0)

{

From = first;

To = second;

Weight = weight;

}

public bool IsIncident(Node node) => From == node || To == node;

public Node OtherNode(Node node)

{

if (!IsIncident(node)) throw new ArgumentException();

if (From == node) return To;

return From;

}

}

public class Node //16, 17 вершина

{

private List<Edge> \_incidentEdges = new List<Edge>();

public int NodeNumber;

public Node(int number) => NodeNumber = number;

public override string ToString() => NodeNumber.ToString();// чтобы нормально в дебаге отображалось

public IEnumerable<Node> IncidentNodes //перечислить ноды кт инцидентны данной, но не позволит изменить

{

get { return \_incidentEdges.Select(z => z.OtherNode(this)); }

// get

// {

// foreach (var edge in \_incidentEdges) yield return edge.OtherNode(this);

// }

}

public IEnumerable<Edge> IncidentEdges // тоже самое, что и с узлами

{

get

{

foreach (var edge in \_incidentEdges) yield return edge;

}

}

public static Edge Connect(Node node1, Node node2, int weight = 0) // создание связи в неориентированном графе

{

var edge = new Edge(node1, node2, weight);

node1.\_incidentEdges.Add(edge);

node2.\_incidentEdges.Add(edge);

return edge;

}

}

public class Graph //16, 17

{

public Node[] \_nodes;

public Graph(int nodesCount) // чтобы избежать повторения номеров вершин, узнаем их количество

{

\_nodes = Enumerable.Range(0, nodesCount).Select(x => new Node(x)).ToArray();

}

public int Length => \_nodes.Length;

public Node this[int index] => \_nodes[index]; // для извлечения вершин, чтобы их соединить

public IEnumerable<Node> Nodes

{

get

{

foreach (var node in \_nodes) yield return node;

}

}

public Edge Connect(int index1, int index2, int weight = 0) => Node.Connect(\_nodes[index1], \_nodes[index2], weight);

public IEnumerable<Edge> Edges

{

get { return \_nodes.SelectMany(z => z.IncidentEdges).Distinct(); }

}

public static Graph MakeGraph(params int[] incidentNodes)

{

var graph = new Graph(incidentNodes.Max() + 1);

for (var i = 0; i < incidentNodes.Length - 1; i += 2)

graph.Connect(incidentNodes[i], incidentNodes[i + 1]);

return graph;

}

public bool IsCyclicUtil(Node current, bool[] visited, Node parent)

{

visited[current.NodeNumber] = true;

foreach (var i in current.IncidentNodes)

{

if (!visited[i.NodeNumber])

{

if (IsCyclicUtil(i, visited, current))

return true;

}

else if (i != parent)

return true;

}

return false;

}

}

public static Graph MakeGraph(List<Edge> tree) //16, 17. создание множества из набора вершин

{

List<Node> nodes = new List<Node>();

foreach (var edge in tree)

{

if (!nodes.Contains(edge.From))

nodes.Add(edge.From);

if (!nodes.Contains(edge.To))

nodes.Add(edge.To);

}

Graph graph = new Graph(nodes.Count);

nodes.OrderBy(t => t.NodeNumber);

foreach (var n in tree)

{

if (!graph.Edges.Contains(n))

graph.Connect(graph.Nodes.Where(t => t.NodeNumber == nodes.IndexOf(n.To))

.First().NodeNumber,

graph.Nodes.Where(t => t.NodeNumber == nodes.IndexOf(n.From))

.First().NodeNumber, n.Weight);

}

return graph;

}

public static bool HasCycle(Graph graph) //16, 17

{

int length = graph.Nodes.Count();

bool[] visited = new bool[length];

for (int i = 0; i < length; i++) visited[i] = false;

for (int u = 0; u < length; u++)

if (!visited[u])

if (graph.IsCyclicUtil((Node)graph.Nodes.Where(t => t.NodeNumber == u)

.First(), visited, null))

return true;

return false;

}

1. **Алгоритм Прима (поиска минимального остовного дерева)** - search.sln

**Задача: найти такой связный подграф без циклов, содержащий все вершины, что суммарный вес его ребер будет минимален.**1) Построение начинается с дерева, включающего в себя одну (произвольную) вершину.2)На каждом шаге алгоритма к текущему дереву присоединяется самое лёгкое(ребро с наименьшим весом) из рёбер, соединяющих вершину из построенного дерева, и вершину не из дерева (один конец ребра – это вершина, входящая в дерево, а другой – нет.)3) Алгоритм продолжается до тех пор, пока в дерево не будут включены все вершины данного графа. На выходе имеем минимальное остовное дерево.

static void Main(string[] args) //17. Алгоритм Прима

{

var graph = new Graph(4);

//var weights = new Dictionary<Edge, double>

//{

// [graph.Connect(0, 1)] = 1,

// [graph.Connect(0, 2)] = 2,

// [graph.Connect(0, 3)] = 6,

// [graph.Connect(1, 3)] = 4,

// [graph.Connect(2, 3)] = 2

//};

graph.Connect(0, 1, 1);

graph.Connect(0, 2, 2);

graph.Connect(0, 3, 6);

graph.Connect(1, 3, 4);

graph.Connect(2, 3, 2);

var t = Prim(graph);

foreach (var m in t)

{

Console.WriteLine(m.To.NodeNumber + " " + m.From.NodeNumber + " " + m.Weight);

}

}

public static bool HasCycle(Graph graph) //17

{

int length = graph.Nodes.Count();

bool[] visited = new bool[length];

for (int i = 0; i < length; i++) visited[i] = false;

for (int u = 0; u < length; u++)

if (!visited[u])

if (graph.IsCyclicUtil((Node)graph.Nodes.Where(t => t.NodeNumber == u)

.First(), visited, null))

return true;

return false;

}

public class Edge //16, 17 ребро

{

public Node From;

public Node To;

public int Weight;

public Edge(Node first, Node second, int weight = 0)

{

From = first;

To = second;

Weight = weight;

}

public bool IsIncident(Node node) => From == node || To == node;

public Node OtherNode(Node node)

{

if (!IsIncident(node)) throw new ArgumentException();

if (From == node) return To;

return From;

}

}

public class Node //16, 17 вершина

{

private List<Edge> \_incidentEdges = new List<Edge>();

public int NodeNumber;

public Node(int number) => NodeNumber = number;

public override string ToString() => NodeNumber.ToString();// чтобы нормально в дебаге отображалось

public IEnumerable<Node> IncidentNodes //перечислить ноды кт инцидентны данной, но не позволит изменить

{

get { return \_incidentEdges.Select(z => z.OtherNode(this)); }

// get

// {

// foreach (var edge in \_incidentEdges) yield return edge.OtherNode(this);

// }

}

public IEnumerable<Edge> IncidentEdges // тоже самое, что и с узлами

{

get

{

foreach (var edge in \_incidentEdges) yield return edge;

}

}

public static Edge Connect(Node node1, Node node2, int weight = 0) // создание связи в неориентированном графе

{

var edge = new Edge(node1, node2, weight);

node1.\_incidentEdges.Add(edge);

node2.\_incidentEdges.Add(edge);

return edge;

}

}

public class Graph //16, 17

{

public Node[] \_nodes;

public Graph(int nodesCount) // чтобы избежать повторения номеров вершин, узнаем их количество

{

\_nodes = Enumerable.Range(0, nodesCount).Select(x => new Node(x)).ToArray();

}

public int Length => \_nodes.Length;

public Node this[int index] => \_nodes[index]; // для извлечения вершин, чтобы их соединить

public IEnumerable<Node> Nodes

{

get

{

foreach (var node in \_nodes) yield return node;

}

}

public Edge Connect(int index1, int index2, int weight = 0) => Node.Connect(\_nodes[index1], \_nodes[index2], weight);

public IEnumerable<Edge> Edges

{

get { return \_nodes.SelectMany(z => z.IncidentEdges).Distinct(); }

}

public static Graph MakeGraph(params int[] incidentNodes)

{

var graph = new Graph(incidentNodes.Max() + 1);

for (var i = 0; i < incidentNodes.Length - 1; i += 2)

graph.Connect(incidentNodes[i], incidentNodes[i + 1]);

return graph;

}

public bool IsCyclicUtil(Node current, bool[] visited, Node parent)

{

visited[current.NodeNumber] = true;

foreach (var i in current.IncidentNodes)

{

if (!visited[i.NodeNumber])

{

if (IsCyclicUtil(i, visited, current))

return true;

}

else if (i != parent)

return true;

}

return false;

}

}

public static IEnumerable<Edge> Prim(Graph graph)

{

var tree = new List<Edge>();

var possibleEdges = new List<Edge>();

var nodes = new List<Node>();

Random rnd = new Random();

int num = rnd.Next(0, graph.Length);

Node node = graph.Nodes.Where(t => t.NodeNumber == num).First(); //берем вершину

nodes.Add(node);

while (tree.Count < graph.Length - 1) //пока не включены все вершины графа

{

foreach (var t in node.IncidentEdges) //находим самые легкие ребра

if (!possibleEdges.Contains(t) && t.Weight > 0 &&

tree.Find(b => b.To == t.From && b.From == t.To) == null

&& !tree.Contains(t))

possibleEdges.Add(t);

possibleEdges = possibleEdges.OrderBy(b => b.Weight).ToList();

foreach (var t in possibleEdges)

{

tree.Add(t); //присоединяем к дереву

if ((nodes.Contains(t.To) && nodes.Contains(t.From)) || HasCycle(MakeGraph(tree)))

{

tree.Remove(t);

continue;

}

node = t.To == node ? t.From : t.To;

possibleEdges.Remove(t);

break;

}

}

return tree; //мин остовное дерево

}

public static Graph MakeGraph(List<Edge> tree)

{

List<Node> nodes = new List<Node>();

foreach (var edge in tree)

{

if (!nodes.Contains(edge.From))

nodes.Add(edge.From);

if (!nodes.Contains(edge.To))

nodes.Add(edge.To);

}

Graph graph = new Graph(nodes.Count);

nodes.OrderBy(t => t.NodeNumber);

foreach (var n in tree)

{

if (!graph.Edges.Contains(n))

graph.Connect(graph.Nodes.Where(t => t.NodeNumber == nodes.IndexOf(n.To))

.First().NodeNumber,

graph.Nodes.Where(t => t.NodeNumber == nodes.IndexOf(n.From))

.First().NodeNumber, n.Weight);

}

return graph;

}

1. **Обход графа в глубину** – graph.sln
2. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная.
3. Движемся в произвольную смежную вершину, помечаем ее как посещенную и из этой вершины обходим все возможные пути до смежных вершин.
4. Если таких путей нет или мы не достигли конечной вершины, то возвращаемся назад к вершине с несколькими исходящими ребрами и идем по другому пути. Алгоритм повторяется, пока не будут исследованы все вершины и достигнута конечная вершина, т.е до тех пор, пока все вершины не будут помечены как посещенные.  
   **Если применена****матрица смежности, то O(n2), а если нематричное** **представление – O(n+m)****, где n вершин и m ребер**

public class Program {

public delegate void WriteLog(string text);

public static void Main()

{

WriteLog writeLog = WriteInConsole;

TestDFT(writeLog);

}

public static void WriteInConsole(string text) => Console.Write(text);

public static void TestDFT(WriteLog writeLog) //Обход в глубину - создание графа

{

Graph graph = new();

GraphAlgorithms algorithms = new(graph, writeLog);

graph.AddLink(1, 2);

graph.AddLink(1, 7);

graph.AddLink(1, 8);

graph.AddLink(2, 3);

graph.AddLink(2, 6);

graph.AddLink(3, 4);

graph.AddLink(3, 5);

graph.AddLink(8, 9);

graph.AddLink(8, 12);

graph.AddLink(9, 10);

graph.AddLink(9, 11);

algorithms.DFT("1");

}

}

public class Node //узел

{

public string Name { get; set; }

public List<string> Neighbours { get; set; }

public Node(string name)

{

Name = name;

Neighbours = new();

}

public void UpdateName(string newName) => Name = newName;

public void AddNeighbour(string neighbour)

{

if (!Neighbours.Contains(neighbour)) Neighbours.Add(neighbour);

}

}

public class Graph

{

public Graph()

{

Nodes = new();

Links = new();

}

public Dictionary<string, Node> Nodes { get; set; } //узлы

public List<Link> Links { get; set; } //ребра

public void AddNode(string nodeName) //добавление узла

{

if (!Nodes.ContainsKey(nodeName)) Nodes.Add(nodeName, new Node(nodeName));

}

public void AddLink<T>(T source, T target, int weight = 1) //добавление ребра

{

string s = source.ToString();

string t = target.ToString();

AddNode(s);

AddNode(t);

Links.Add(new Link(s, t, weight));

Nodes[s].AddNeighbour(t);

}

}

public class Link //ребрo

{

public string Source { get; set; }

public string Target { get; set; }

public int Weight { get; set; }

public Link(string source, string target, int weight)

{

Source = source;

Target = target;

Weight = weight;

}

public override string ToString() => Source + "-" + Target;

}

public class GraphAlgorithms

{

public Graph graph;

public WriteLog WriteLog;

public GraphAlgorithms(Graph graph, WriteLog writeLog)

{

this.graph = graph;

WriteLog = writeLog;

}

public void DFT(string node) //Обход в глубину

{

Dictionary<string, bool> visitedNodes = new(); //Список посещенных узлов

foreach (var e in graph.Nodes)

{

visitedNodes.Add(e.Key, false);

}

WriteLog($"Обход в глубину\n" + $"Начинаем с узла {node}\n");

DFTRecursive(node, visitedNodes); //Вызов рекурсивного метода

}

private void DFTRecursive(string node, Dictionary<string, bool> visited) //Рекурсивный метод

{

WriteLog($"Отмечаем узел {node}\n");

visited[node] = true; //Отмечаем полученный узел

List<string> neighbours = graph.Nodes[node].Neighbours; //Выполняем подобное для каждого соседнего узла

WriteLog($"Соседи узла {node}:");

foreach (var e in neighbours)

{

WriteLog($" {e}");

}

WriteLog("\n------\n");

foreach (var n in neighbours)

{

if (!visited[n]) DFTRecursive(n, visited);

}

}

}

1. **Обход графа в ширину** – graph.sln

1) Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная (и заносится в очередь).

2) Посещается первая вершина из очереди (если она не помечена как посещенная). Все ее соседние вершины заносятся в очередь. После этого она удаляется из очереди.

3) Повторяется шаг 2 до тех пор, пока очередь не пуста.  
**Сложность** **при нематричном представлении графа равна O(n+m), где n вершин и m ребер. Использование****матрицы смежности приводит к оценке O(n2)**

public class Program

{

public delegate void WriteLog(string text);

public static void Main()

{

WriteLog writeLog = WriteInConsole;

TestBFT(writeLog);

}

public static void WriteInConsole(string text) => Console.Write(text);

public static void TestBFT(WriteLog writeLog) //Обход в ширину - создание графа

{

Graph graph = new();

GraphAlgorithms algorithms = new(graph, writeLog);

graph.AddLink(1, 2);

graph.AddLink(1, 3);

graph.AddLink(1, 4);

graph.AddLink(2, 5);

graph.AddLink(3, 6);

graph.AddLink(3, 7);

graph.AddLink(4, 8);

graph.AddLink(5, 9);

graph.AddLink(6, 10);

algorithms.BFT("1");

}

}

public class Node //узел

{

public string Name { get; set; }

public List<string> Neighbours { get; set; }

public Node(string name)

{

Name = name;

Neighbours = new();

}

public void UpdateName(string newName) => Name = newName;

public void AddNeighbour(string neighbour)

{

if (!Neighbours.Contains(neighbour)) Neighbours.Add(neighbour);

}

}

public class Graph

{

public Graph()

{

Nodes = new();

Links = new();

}

public Dictionary<string, Node> Nodes { get; set; } //узлы

public List<Link> Links { get; set; } //ребра

public void AddNode(string nodeName) //добавление узла

{

if (!Nodes.ContainsKey(nodeName)) Nodes.Add(nodeName, new Node(nodeName));

}

public void AddLink<T>(T source, T target, int weight = 1) //добавление ребра

{

string s = source.ToString();

string t = target.ToString();

AddNode(s);

AddNode(t);

Links.Add(new Link(s, t, weight));

Nodes[s].AddNeighbour(t);

}

public class Link //ребрo

{

public string Source { get; set; }

public string Target { get; set; }

public int Weight { get; set; }

public Link(string source, string target, int weight)

{

Source = source;

Target = target;

Weight = weight;

}

public override string ToString() => Source + "-" + Target;

}

public class GraphAlgorithms

{

public Graph graph;

public WriteLog WriteLog;

public GraphAlgorithms(Graph graph, WriteLog writeLog)

{

this.graph = graph;

WriteLog = writeLog;

}

public void BFT(string node) //Обход в ширину

{

if (!graph.Nodes.ContainsKey(node)) throw new Exception("Error");

Dictionary<string, bool> visitedNodes = new(); //Список посещенных узлов

foreach (var e in graph.Nodes)

{

visitedNodes.Add(e.Key, false);

}

LinkedList<string> queue = new(); //Очередь узлов для посещения

WriteLog($"Обход в ширину\n" + $"Начинаем с узла {node}\n");

WriteLog($"Отмечаем узел {node}\n");

visitedNodes[node] = true; //Отмечаем первый узел и добавляем в очередь

WriteLog($"Добавляем узел {node} в очередь\n");

queue.AddLast(node);

WriteLog("Очередь:");

foreach (var e in queue)

{

WriteLog($" {e}");

}

WriteLog("\n-----\n");

while (queue.Any())

{

node = queue.First(); //Убираем первый узел в очереди

WriteLog($"Берём первый узел из очереди: {node}\n");

queue.RemoveFirst();

//Получаем список соседних узлов, Отмечаем их и добавляем в очередь

List<string> neighbours = graph.Nodes[node].Neighbours;

WriteLog($"Соседи узла {node}:");

foreach (var e in neighbours)

{

WriteLog($" {e}");

}

WriteLog("\n");

foreach (var val in neighbours)

{

if (!visitedNodes[val])

{

visitedNodes[val] = true;

queue.AddLast(val);

}

}

WriteLog("Очередь:");

foreach (var e in queue)

{

WriteLog($" {e}");

}

WriteLog("\n------\n");

}

}

}

1. **Алгоритм Дейкстры (поиск кратчайшего пути между двумя заданными вершинами графа)** - search.sln

Алгоритм можно использовать для графов с положительными значениями весов ребер.  
Для каждой вершины графа вычисляется сумма весов ребер(расстояние) от начальной вершины.

1) **Всем вершинам, за исключением первой, присваивается вес равный бесконечности, а первой вершине – 0. Это показывает что расстояние до других вершин неизвестны. Все кроме первой вершины помечаются как непосещенные.**  
2) Все вершины не выделены.  
3) Первая вершина объявляется текущей.  
4) Вес всех невыделенных вершин пересчитывается по формуле: вес невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса ребра, соединяющего текущую вершину с невыделенной.  
5) Среди невыделенных вершин ищется вершина с минимальным весом. Если таковая не найдена, то есть вес всех вершин равен бесконечности, то маршрут не существует. Следовательно, выход. Иначе, текущей становится найденная вершина. Она же выделяется.  
6) Если текущей вершиной оказывается конечная, то путь найден, и его вес есть вес конечной вершины.  
7) Переход на шаг 4  
**Итерации алгоритма: Алгоритм завершается как только все вершины посещены. В ином случае, из непосещенных вершин, выбирается та, у которой меньшая сумма весов. Затем рассматриваются все маршруты, в которых выбранная вершина стоит на предпоследнем месте.**  
**О(n^2)**

static void Main(string[] args) {

Console.WriteLine("Алгоритм Дейкстры(поиск кратчайшего пути между двумя заданными вершинами графа)");

var graph = new Graph();

graph.AddVertex("A"); //добавление вершин

graph.AddVertex("B");

graph.AddVertex("C");

graph.AddVertex("D");

graph.AddVertex("E");

graph.AddVertex("F");

graph.AddVertex("G");

graph.AddEdge("A", "B", 22); //добавление ребер

graph.AddEdge("A", "C", 33);

graph.AddEdge("A", "D", 61);

graph.AddEdge("B", "C", 47);

graph.AddEdge("B", "E", 93);

graph.AddEdge("C", "D", 11);

graph.AddEdge("C", "E", 79);

graph.AddEdge("C", "F", 63);

graph.AddEdge("D", "F", 41);

graph.AddEdge("E", "F", 17);

graph.AddEdge("E", "G", 58);

graph.AddEdge("F", "G", 84);

var dijkstra = new Dijkstra(graph);

var path = dijkstra.FindShortestPath("A", "G");

Console.WriteLine(path);

}

**public class GraphVertex** //20. Вершина графа

{

public string Name { get; } //Название вершины

public List<GraphEdge> Edges { get; } //Список ребер

public GraphVertex(string vertexName)

{

Name = vertexName;

Edges = new List<GraphEdge>();

}

public void AddEdge(GraphEdge newEdge) => Edges.Add(newEdge); //Добавить ребро

public void AddEdge(GraphVertex vertex, int edgeWeight) => AddEdge(new GraphEdge(vertex, edgeWeight)); //Добавить ребро

public override string ToString() => Name; //Преобразование в строку

}

**public class GraphEdge** //20. Ребро графа

{

public GraphVertex ConnectedVertex { get; } //Связанная вершина

public int EdgeWeight { get; } //Вес ребра

public GraphEdge(GraphVertex connectedVertex, int weight)

{

ConnectedVertex = connectedVertex;

EdgeWeight = weight;

}

}

**public class Graph** //20. Граф

{

public List<GraphVertex> Vertices { get; } //Список вершин графа

public Graph() => Vertices = new List<GraphVertex>();

public void AddVertex(string vertexName) => Vertices.Add(new GraphVertex(vertexName)); //Добавление вершины

public GraphVertex FindVertex(string vertexName) //Поиск вершины

{

foreach (var v in Vertices)

{

if (v.Name.Equals(vertexName))

{

return v;

}

}

return null;

}

public void AddEdge(string firstName, string secondName, int weight) //Добавление ребра

{

var v1 = FindVertex(firstName);

var v2 = FindVertex(secondName);

if (v2 != null && v1 != null)

{

v1.AddEdge(v2, weight);

v2.AddEdge(v1, weight);

}

}

}

**public class GraphVertexInfo** //20. Информация о вершинах графа

{

public GraphVertex Vertex { get; set; } //Вершина

public bool IsUnvisited { get; set; } //Не посещенная вершина

public int EdgesWeightSum { get; set; } //Сумма весов ребер

public GraphVertex PreviousVertex { get; set; } //Предыдущая вершина

public GraphVertexInfo(GraphVertex vertex)

{

Vertex = vertex;

IsUnvisited = true;

EdgesWeightSum = int.MaxValue;

PreviousVertex = null;

}

}

**public class Dijkstra** //20. Алгоритм Дейкстры - реализация поиска кратчайшего пути

{

Graph graph;

List<GraphVertexInfo> infos;

public Dijkstra(Graph graph) => this.graph = graph;

public void InitInfo() //Инициализация информации

{

infos = new List<GraphVertexInfo>();

foreach (var v in graph.Vertices)

{

infos.Add(new GraphVertexInfo(v));

}

}

GraphVertexInfo GetVertexInfo(GraphVertex v) //Получение информации о вершине графа

{

foreach (var i in infos)

{

if (i.Vertex.Equals(v))

{

return i;

}

}

return null;

}

public GraphVertexInfo FindUnvisitedVertexWithMinSum() //Поиск непосещенной вершины с минимальным значением суммы

{

var minValue = int.MaxValue;

GraphVertexInfo minVertexInfo = null;

foreach (var i in infos)

{

if (i.IsUnvisited && i.EdgesWeightSum < minValue)

{

minVertexInfo = i;

minValue = i.EdgesWeightSum;

}

}

return minVertexInfo;

}

public string **FindShortestPath**(string startName, string finishName) //Поиск кратчайшего пути по названиям вершин

{

return FindShortestPath(graph.FindVertex(startName), graph.FindVertex(finishName));

}

public string **FindShortestPath**(GraphVertex startVertex, GraphVertex finishVertex) //Поиск кратчайшего пути по вершинам

{

InitInfo();

var first = GetVertexInfo(startVertex);

first.EdgesWeightSum = 0;

while (true)

{

var current = FindUnvisitedVertexWithMinSum();

if (current == null)

{

break;

}

SetSumToNextVertex(current);

}

return GetPath(startVertex, finishVertex);

}

public void SetSumToNextVertex(GraphVertexInfo info) //Вычисление суммы весов ребер для следующей вершины

{

info.IsUnvisited = false;

foreach (var e in info.Vertex.Edges)

{

var nextInfo = GetVertexInfo(e.ConnectedVertex);

var sum = info.EdgesWeightSum + e.EdgeWeight;

if (sum < nextInfo.EdgesWeightSum)

{

nextInfo.EdgesWeightSum = sum;

nextInfo.PreviousVertex = info.Vertex;

}

}

}

public string GetPath(GraphVertex startVertex, GraphVertex endVertex) //Формирование пути

{

var path = endVertex.ToString();

while (startVertex != endVertex)

{

endVertex = GetVertexInfo(endVertex).PreviousVertex;

path = endVertex.ToString() + path;

}

return path;

}

}

* **Шаг алгоритма:**
* **Если все вершины посещены, алгоритм завершается.**
* **Иначе:**
  + **из ещё не посещённых вершин выбирается вершина *u*, имеющая минимальную метку;**
  + **рассматриваем всевозможные маршруты, в которых *u* является предпоследним пунктом;**
  + **вершины, в которые ведут рёбра из *u*, назовем *соседями* этой вершины;**
  + **для каждого соседа вершины *u*, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки *u* и длины ребра, соединяющего *u* с этим соседом;**
  + **если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины;**
  + **рассмотрев всех соседей, пометим вершину *u* как посещенную и повторим шаг алгоритма;**