# Пузырьковая сортировка

Пузырьковая сортировка - это простой алгоритм сортировки, который работает путем многократного прохода по списку и сравнения соседних элементов. Если два соседних элемента находятся в неправильном порядке, то они меняются местами. По мере того как мы продвигаемся по списку, бОльшие элементы "всплывают" к верху, а меньшие элементы "опускаются" к низу.

Пример пузырьковой сортировки на C#:

public static void BubbleSort(int[] arr)

{

int n = arr.Length;

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

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

{

// меняем местами элементы

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

В данном примере мы имеем функцию BubbleSort, которая принимает массив целых чисел и сортирует его пузырьковым методом. Во внутреннем цикле for мы сравниваем два соседних элемента и, если они находятся в неправильном порядке, меняем их местами.

Сложность алгоритма пузырьковой сортировки в худшем случае составляет O(n^2), так как мы должны выполнить n-1 проходов, и в каждом проходе сравнивать n-i-1 пар элементов. В лучшем случае, если список уже отсортирован, сложность будет O(n), потому что мы проходим по списку только один раз и не выполняем никаких обменов.

# Сортировка выбором

Сортировка выбором - это алгоритм сортировки, который находит минимальный элемент в списке и перемещает его в начало. Затем он ищет следующий минимальный элемент в оставшейся части списка и перемещает его в следующую позицию, и так далее до тех пор, пока весь список не будет отсортирован.

Пример сортировки выбором на C#:

public static void SelectionSort(int[] arr)

{

int n = arr.Length;

// проходим по списку

for (int i = 0; i < n - 1; i++)

{

// находим минимальный элемент в неотсортированной части списка

int minIndex = i;

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

{

if (arr[j] < arr[minIndex])

{

minIndex = j;

}

}

// меняем местами текущий элемент и минимальный элемент

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

В данном примере мы имеем функцию SelectionSort, которая принимает массив целых чисел и сортирует его с использованием сортировки выбором. Во внешнем цикле for мы проходим по списку и ищем минимальный элемент в оставшейся части списка во внутреннем цикле for. Затем мы меняем местами текущий элемент и минимальный элемент.

Сложность алгоритма сортировки выбором в худшем случае также составляет O(n^2), так как мы должны выполнить n-1 проходов, и в каждом проходе выполнять n-i-1 сравнений. В лучшем случае, если список уже отсортирован, сложность также будет O(n^2), потому что мы всегда будем искать минимальный элемент в оставшейся части списка.

# Сортировка вставками

Сортировка вставками - это алгоритм сортировки, который проходит по списку и вставляет каждый элемент в правильную позицию в отсортированной части списка. Другими словами, мы начинаем с первого элемента списка и постепенно увеличиваем отсортированную часть списка, вставляя каждый следующий элемент в правильную позицию.

Пример сортировки вставками на C#:

public static void InsertionSort(int[] arr)

{

int n = arr.Length;

// проходим по списку

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

{

int key = arr[i];

int j = i - 1;

// перемещаем элементы больше ключа на одну позицию вправо

while (j >= 0 && arr[j] > key)

{

arr[j + 1] = arr[j];

j--;

}

// вставляем ключ в правильную позицию

arr[j + 1] = key;

}

}

В данном примере мы имеем функцию InsertionSort, которая принимает массив целых чисел и сортирует его с использованием сортировки вставками. Во внешнем цикле for мы проходим по списку, начиная со второго элемента, и во внутреннем цикле while перемещаем элементы больше ключа на одну позицию вправо. Затем мы вставляем ключ в правильную позицию в отсортированной части списка.

Сложность алгоритма сортировки вставками в худшем случае также составляет O(n^2), так как мы должны выполнить n-1 проходов, и в каждом проходе сравнивать и перемещать до i элементов. В лучшем случае, если список уже отсортирован, сложность будет O(n), потому что мы проходим по списку только один раз и не выполняем никаких перемещений.

# Быстрая сортировка (quicksort)

Быстрая сортировка (quicksort) - это алгоритм сортировки, который использует метод "разделяй и властвуй", чтобы разбить список на меньшие подсписки и рекурсивно сортировать их.

Принцип работы алгоритма заключается в выборе элемента из списка в качестве опорного, после чего элементы, которые меньше опорного, перемещаются перед ним, а элементы, которые больше опорного, перемещаются после него. Затем алгоритм рекурсивно вызывает сам себя для левой и правой частей списка, пока не достигнет отсортированного списка.

Пример сортировки быстрой сортировкой на C#:

public static void QuickSort(int[] arr, int low, int high)

{

if (low < high)

{

// выбираем опорный элемент

int pivot = Partition(arr, low, high);

// рекурсивно вызываем QuickSort для левой и правой частей списка

QuickSort(arr, low, pivot - 1);

QuickSort(arr, pivot + 1, high);

}

}

private static int Partition(int[] arr, int low, int high)

{

// выбираем опорный элемент

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++)

{

// если текущий элемент меньше опорного

if (arr[j] < pivot)

{

i++;

// меняем местами элементы

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// меняем местами опорный элемент и элемент в позиции i+1

int temp2 = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp2;

return i + 1;

}

В данном примере мы имеем функцию QuickSort, которая принимает массив целых чисел, а также индексы low и high, которые обозначают начальную и конечную позиции в списке. Внутри функции мы проверяем, если low меньше high, то выбираем опорный элемент с помощью функции Partition, затем рекурсивно вызываем QuickSort для левой и правой частей списка.

Функция Partition выбирает последний элемент в списке в качестве опорного и проходит по списку, сравнивая каждый элемент с опорным и перемещая элементы, которые меньше опорного, перед ним, а элементы, которые больше опорного, после него.

Сложность алгоритма быстрой сортировки в худшем случае составляет O(n^2), когда в качестве опорного элемента выбирается наименьший элемент или наибольший элемент в списке, а список уже отсортирован по возрастанию или убыванию.

В лучшем и среднем случае быстрая сортировка имеет сложность O(n log n), что делает ее одним из наиболее эффективных алгоритмов сортировки.

Однако, худший случай может быть улучшен с помощью оптимизаций, таких как выбор опорного элемента случайным образом или использование медианы в качестве опорного элемента, что делает сложность в худшем случае O(n log n) как и в лучшем и среднем случае.

# Сортировка Шелла (Shell sort)

Сортировка Шелла (Shell sort) является усовершенствованным вариантом сортировки вставками и была предложена Дональдом Шеллом в 1959 году. Она использует последовательность шагов, чтобы сначала сделать сравнения и перестановки на большом расстоянии, а затем на уменьшающихся расстояниях, в результате чего элементы "проваливаются" быстрее на свои места. Это уменьшает количество сравнений и обменов, и делает сортировку более эффективной, чем простая сортировка вставками.

Пример сортировки Шелла на C#:

static void ShellSort(int[] arr)

{

int n = arr.Length;

// Находим начальное значение интервала

int interval = 1;

while (interval < n / 3)

{

interval = interval \* 3 + 1;

}

// Уменьшаем интервал по мере прохода по списку

while (interval > 0)

{

// Сортировка вставками для элементов внутри интервала

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

{

int temp = arr[i];

int j = i;

while (j >= interval && arr[j - interval] > temp)

{

arr[j] = arr[j - interval];

j -= interval;

}

arr[j] = temp;

}

// Уменьшаем интервал

interval = (interval - 1) / 3;

}

}

В этой реализации сначала находим начальное значение интервала, которое определяется как максимальное значение, которое не превышает одну треть размера списка. Затем мы уменьшаем интервал на каждом проходе по списку и выполняем сортировку вставками для элементов внутри интервала.

Сложность алгоритма Шелла зависит от выбора последовательности шагов. В худшем случае сложность составляет O(n^2), но в среднем случае она имеет сложность O(n log n). В некоторых случаях сортировка Шелла может быть более эффективной, чем быстрая сортировка, особенно если список уже частично отсортирован.

# Сортировка на дереве

Одним из методов сортировки на дереве является сортировка с помощью двоичного дерева поиска. Этот метод использует структуру двоичного дерева, в котором каждый узел содержит ключ и ссылки на левого и правого потомков. Ключи левых потомков меньше, чем ключ родительского узла, а ключи правых потомков больше. Таким образом, при обходе дерева ключи будут отсортированы в порядке возрастания.

Пример сортировки на дереве с помощью двоичного дерева поиска на C#:

class Node

{

public int value;

public Node left;

public Node right;

public Node(int value)

{

this.value = value;

left = null;

right = null;

}

}

class BinarySearchTree

{

public Node root;

public BinarySearchTree()

{

root = null;

}

public void Insert(int value)

{

root = InsertHelper(root, value);

}

private Node InsertHelper(Node node, int value)

{

if (node == null)

{

node = new Node(value);

}

else if (value < node.value)

{

node.left = InsertHelper(node.left, value);

}

else

{

node.right = InsertHelper(node.right, value);

}

return node;

}

public void TraverseInOrder()

{

TraverseInOrderHelper(root);

}

private void TraverseInOrderHelper(Node node)

{

if (node != null)

{

TraverseInOrderHelper(node.left);

Console.Write(node.value + " ");

TraverseInOrderHelper(node.right);

}

}

}

В этой реализации мы создаем классы Node и BinarySearchTree для представления двоичного дерева поиска. Метод Insert используется для добавления нового элемента в дерево, а метод TraverseInOrder используется для обхода дерева в порядке возрастания ключей.

Для сортировки списка мы создаем экземпляр BinarySearchTree, добавляем каждый элемент списка в дерево с помощью метода Insert, а затем обходим дерево в порядке возрастания ключей с помощью метода TraverseInOrder. Это позволяет получить отсортированный список.

Сложность сортировки на дереве с помощью двоичного дерева поиска в худшем случае может достигать O(n^2), если дерево является несбалансированным. В среднем же случае сложность составляет O(n log n).

Задание:

разработать шаблонные функции сортировки, проверить на массивах размером 10000..40000, вывести рез-ты в файл и изобразить график.

Код:

using System;

using System.Diagnostics;

using System.IO;

using System;

public static class Sort

{

public static void BubbleSort<T>(T[] arr) where T : IComparable<T>

{

for (int i = 0; i < arr.Length - 1; i++)

{

for (int j = 0; j < arr.Length - i - 1; j++)

{

if (arr[j].CompareTo(arr[j + 1]) > 0)

{

Swap(arr, j, j + 1);

}

}

}

}

public static void SelectionSort<T>(T[] arr) where T : IComparable<T>

{

int minIndex;

for (int i = 0; i < arr.Length - 1; i++)

{

minIndex = i;

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

{

if (arr[j].CompareTo(arr[minIndex]) < 0)

{

minIndex = j;

}

}

if (minIndex != i)

{

Swap(arr, i, minIndex);

}

}

}

public static void InsertionSort<T>(T[] arr) where T : IComparable<T>

{

T key;

int j;

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

{

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j].CompareTo(key) > 0)

{

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

private static void Swap<T>(T[] arr, int i, int j)

{

T temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void QuickSort<T>(T[] array, int left, int right) where T : IComparable<T>

{

if (left >= right)

{

return;

}

T pivot = array[(left + right) / 2];

int partitionIndex = Partition(array, left, right, pivot);

QuickSort(array, left, partitionIndex - 1);

QuickSort(array, partitionIndex, right);

}

private static int Partition<T>(T[] array, int left, int right, T pivot) where T : IComparable<T>

{

while (left <= right)

{

while (array[left].CompareTo(pivot) < 0)

{

left++;

}

while (array[right].CompareTo(pivot) > 0)

{

right--;

}

if (left <= right)

{

Swap(array, left, right);

left++;

right--;

}

}

return left;

}

public static void ShellSort<T>(T[] array) where T : IComparable<T>

{

int gap = array.Length / 2;

while (gap > 0)

{

for (int i = gap; i < array.Length; i++)

{

T temp = array[i];

int j = i;

while (j >= gap && array[j - gap].CompareTo(temp) > 0)

{

array[j] = array[j - gap];

j -= gap;

}

array[j] = temp;

}

gap = gap / 2;

}

}

}

class Program

{

static void Main(string[] args)

{

int[] sizes = { 10000, 20000, 30000, 40000 };

string filePath = "sort\_results.txt";

using (StreamWriter writer = new StreamWriter(filePath))

{

foreach (int size in sizes)

{

int[] array = GenerateRandomArray(size);

Stopwatch stopwatch = new Stopwatch();

// Bubble sort

stopwatch.Start();

Sort.BubbleSort(array);

stopwatch.Stop();

writer.WriteLine($"Bubble sort time ({size}): {stopwatch.ElapsedMilliseconds} ms");

// Selection sort

array = GenerateRandomArray(size);

stopwatch.Restart();

Sort.SelectionSort(array);

stopwatch.Stop();

writer.WriteLine($"Selection sort time ({size}): {stopwatch.ElapsedMilliseconds} ms");

// Insertion sort

array = GenerateRandomArray(size);

stopwatch.Restart();

Sort.InsertionSort(array);

stopwatch.Stop();

writer.WriteLine($"Insertion sort time ({size}): {stopwatch.ElapsedMilliseconds} ms");

// Quick sort

array = GenerateRandomArray(size);

stopwatch.Restart();

Sort.QuickSort(array, 0, size - 1);

stopwatch.Stop();

writer.WriteLine($"Quick sort time ({size}): {stopwatch.ElapsedMilliseconds} ms");

// Shell sort

array = GenerateRandomArray(size);

stopwatch.Restart();

Sort.ShellSort(array);

stopwatch.Stop();

writer.WriteLine($"Shell sort time ({size}): {stopwatch.ElapsedMilliseconds} ms");

}

}

Console.WriteLine($"Sorting results written to file: {filePath}");

Console.ReadLine();

}

static int[] GenerateRandomArray(int size)

{

Random random = new Random();

int[] array = new int[size];

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

{

array[i] = random.Next();

}

return array;

}

}

