Лабораторная работа 5

**1) Сортировка выбором (Selection Sort)** — это алгоритм сортировки, который находит наименьший (или наибольший) элемент в массиве и перемещает его в начало (или конец). Процесс повторяется, пока все элементы не окажутся упорядоченными.

Пример кода на JAVA

public class SelectionSort {

*// Метод сортировки массива методом выбора*

public static void selectionSort(int[] arr) {

int n = arr.length; *// Получаем длину массива*

for (int i = 0; i < n - 1; i++) { *// Внешний цикл проходит по массиву слева направо*

int minIndex = i; *// Предполагаем, что минимальный элемент находится на текущей позиции*

*// Внутренний цикл ищет наименьший элемент среди оставшихся элементов справа*

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

if (arr[j] < arr[minIndex]) { *// Если найден меньший элемент...*

minIndex = j; *// Запоминаем индекс нового минимального элемента*

}

}

*// Меняем местами текущий элемент с минимальным элементом*

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

*// Вспомогательный метод для вывода отсортированного массива*

public static void printArray(int[] arr) {

System.out.print("Отсортированный массив: ");

for (int value : arr) {

System.out.print(value + " "); *// Вывод каждого элемента массива*

}

System.out.println(); *// Переход на новую строку*

}

*// Основной метод программы*

public static void main(String[] args) {

int[] array = {64, 25, 12, 22, 11}; *// Исходный несортированный массив*

System.out.print("Исходный массив: ");

printArray(array); *// Печать исходного массива*

selectionSort(array); *// Выполняем сортировку выбором*

System.out.print("Результат сортировки: ");

printArray(array); *// Печать отсортированного массива*

}

}

Алгоритм состоит из двух циклов:

1. **Внешний цикл**: выбирает позицию в массиве, начиная с первой (нулевой).
2. **Внутренний цикл**: сравнивает выбранный элемент с остальными элементами массива справа и находит минимальное значение. После завершения внутреннего цикла найденный минимум меняется местами с первым элементом текущего участка массива.

Таким образом, каждый проход внешнего цикла помещает наименьший оставшийся элемент на свою правильную позицию в начале массива.

Важные моменты:

* Алгоритм эффективен для небольших массивов.
* Сложность по времени составляет O(n²), где n — количество элементов в массиве.
* Это нестабильная сортировка, поскольку одинаковые элементы могут менять порядок относительно друг друга.

Пример вывода:

Исходный массив: 64 25 12 22 11

Отсортированный массив: 11 12 22 25 64

Анализ алгоритма: Сортировка выбором (Selection Sort)

Определение:

Сортировка выбором — это алгоритм, который последовательно выбирает минимальный элемент из неотсортированной части массива и переносит его в отсортированную часть.

Анализ:

* Разделяет массив на отсортированную и неотсортированную части.
* На каждой итерации внешнего цикла находит минимальный элемент в неотсортированной части и меняет его с первым элементом.
* Внешний цикл выполняется раз.
* Внутренний цикл проходит по всей оставшейся части массива.

Временная сложность:

* Количество сравнений примерно равно .
* Сложность алгоритма: .

Почему :

Два вложенных цикла приводят к квадратичному росту количества операций по мере увеличения размера массива.

2) **Сортировка пузырьком** (также известна как «сортировка методом обмена») — простой алгоритм сортировки, основанный на попарном сравнении соседних элементов. **Принцип работы**: во время каждого прохода по списку (или массиву) алгоритм сравнивает две соседние позиции. Если элемент слева больше (или, в зависимости от порядка сортировки, меньше) элемента справа, происходит обмен местами. Проход продолжается, пока не будут проверены все пары соседних элементов

Пример на Java

public class BubbleSort {

*// Метод main запускается автоматически при выполнении программы*

public static void main(String[] args) {

int[] array = {64, 34, 25, 12, 22, 11, 90}; *// Исходный массив чисел*

System.out.println("Исходный массив:");

printArray(array); *// Печать исходного массива перед сортировкой*

bubbleSort(array); *// Выполняем алгоритм пузырьковой сортировки*

System.out.println("\nОтсортированный массив:");

printArray(array); *// Печать отсортированного массива*

}

*/\*\**

*\* Реализация метода пузырьковой сортировки*

*\**

*\* @param arr Массив целых чисел для сортировки*

*\*/*

private static void bubbleSort(int[] arr) {

int n = arr.length; *// Получаем длину массива*

boolean swapped; *// Флаг для отслеживания наличия перестановок*

for (int i = 0; i < n - 1; i++) { *// Внешний цикл проходит по всему массиву*

swapped = false; *// Обнуляем флаг на каждом новом проходе*

*// Внутренний цикл сравнивает соседние элементы и меняет местами большие значения*

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

if (arr[j] > arr[j + 1]) { *// Если элемент слева больше элемента справа*

swap(arr, j, j+1); *// Меняем элементы местами*

swapped = true; *// Устанавливаем флаг, что была произведена замена*

}

}

*// Если на каком-то этапе перестановок не было, значит массив уже отсортирован*

if (!swapped)

break;

}

}

*/\*\**

*\* Вспомогательная функция для печати элементов массива*

*\**

*\* @param arr Массив для вывода*

*\*/*

private static void printArray(int[] arr) {

for (int value : arr) {

System.out.print(value + " "); *// Поочередно выводим каждый элемент массива*

}

System.out.println(); *// Переходим на новую строку после вывода всех элементов*

}

*/\*\**

*\* Вспомогательная функция для обмена двух элементов массива*

*\**

*\* @param arr Массив, содержащий элементы*

*\* @param firstIndex Индекс первого элемента*

*\* @param secondIndex Индекс второго элемента*

*\*/*

private static void swap(int[] arr, int firstIndex, int secondIndex) {

int temp = arr[firstIndex]; *// Сохраняем первый элемент временно*

arr[firstIndex] = arr[secondIndex]; *// Переносим второй элемент на позицию первого*

arr[secondIndex] = temp; *// Возвращаем сохранённый ранее первый элемент обратно*

}

}

Подробное пояснение:

**Метод main**:Здесь создаётся простой целочисленный массив, отображаются его элементы до сортировки, выполняется сама сортировка методом пузырька, и снова показываются элементы после завершения процесса.

**Метод bubbleSort:**Этот метод реализует основную логику пузырьковой сортировки.Алгоритм работает следующим образом:

* Во внешней петле (for) мы проходим по каждому элементу массива.
* Во внутренней петле проверяется условие: если левый элемент больше правого, то элементы меняются местами («всплывают»).
* Для оптимизации введён дополнительный флаг (swapped), который позволяет досрочно завершить процесс, если массив уже отсортирован.

**Метод printArray:**Простая вспомогательная функция для удобного вывода элементов массива на экран.

**Метод swap:**Используется для замены значений двух соседних элементов массива.

Код производит следующую последовательность действий:

1. **Создание исходного массива:** {64, 34, 25, 12, 22, 11, 90}
2. **Вывод массива до сортировки:**Исходный массив:64 34 25 12 22 11 90
3. **Выполнение сортировки методом пузырька.** После завершения сортировки:
4. **Вывод отсортированного массива:**Отсортированный массив:11 12 22 25 34 64 90

Анализ алгоритма:

Временная сложность:

Лучшая ситуация (O(n)):

* Когда массив изначально отсортирован. Тогда внутренний цикл сделает всего одну проверку и завершится сразу же благодаря проверке условия на наличие перестановок.

Средняя и худшая ситуации (O(n²)):

* Обычно пузырьковая сортировка требует полного прохода по всем элементам, причем даже если массив почти отсортирован, потребуется значительное число операций сравнения и возможных замен.
* Общий объем вычислений пропорционален количеству парных сравнений и изменений порядка элементов, что составляет около операций.

Почему O(n²)?

* Основная причина временной сложности заключается в наличии двух вложенных циклов:
  + Внешний цикл совершает итерации.
  + Внутренний цикл в самом плохом сценарии (неупорядоченном массиве) выполняет полную проверку оставшихся элементов ( раз), начиная с полной длины и уменьшаясь на единицу при каждом проходе внешнего цикла.
* Суммарно общее количество проверок и возможных замещений растет пропорционально квадрату размера массива, отсюда временная сложность O(n²).

**3) Сортировка вставками (Insertion sort)** — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов.

Пример на Python

def insertion\_sort(arr):  
 # Размер массива  
 n = len(arr)  
  
 # Итерация по элементам массива, начиная со второго элемента  
 for i in range(1, n):  
 # Элемент, который мы хотим вставить в свою позицию  
 current\_element = arr[i]  
  
 # Перемещаемся назад по отсортированной части массива  
 j = i - 1  
  
 # Пока предыдущий элемент больше текущего, двигаем его вперед  
 while j >= 0 and arr[j] > current\_element:  
 arr[j + 1] = arr[j] # Смещаем элемент вперед  
 j -= 1  
  
 # Вставляем текущий элемент на найденную позицию  
 arr[j + 1] = current\_element  
  
  
# Тестируем нашу функцию сортировки  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Исходный массив  
 test\_array = [64, 34, 25, 12, 22, 11, 90]  
  
 # Выводим массив до сортировки  
 print("Массив до сортировки:", test\_array)  
  
 # Применяем сортировку вставками  
 insertion\_sort(test\_array)  
  
 # Выводим отсортированный массив  
 print("Массив после сортировки:", test\_array)

Программа выведит:

Массив до сортировки: [64, 34, 25, 12, 22, 11, 90]

Массив после сортировки: [11, 12, 22, 25, 34, 64, 90]

**Анализ работы алгоритма:**

1. **Идея алгоритма:**
   * Мы рассматриваем массив как состоящий из двух частей: левой отсортированной и правой неотсортированной.
   * Начиная со второго элемента, берем текущий элемент и пытаемся вставить его в правую позицию в отсортированном сегменте.
   * Чтобы сделать это, элементы отсортированного сегмента двигаются вперед, если они больше текущего элемента.
2. **Детали реализации:**
   * Используется два основных цикла:
     + Внешний цикл: проходит по каждому элементу массива, начиная со второго (индекс 1).
     + Внутренний цикл: двигается назад по отсортированной части массива, сравнивая элементы и двигая их вперед, пока не найдет подходящую позицию для вставки текущего элемента.
3. **Особенности реализации:**
   * Каждое сравнение ведет либо к замене элементов, либо к сохранению прежнего положения.
   * Таким образом, каждая новая итерация расширяет отсортированную область массива.

**Временная сложность:**

* **Лучшая ситуация (O(n)):** Когда массив уже отсортирован, внутренний цикл практически не выполняет никаких операций, потому что ни одно смещение не требуется.
* **Средняя и худшая ситуации (O(n²)):** Если массив расположен в обратном порядке или хаотично, придется многократно сравнивать и передвигать элементы, что даёт нам общую сложность порядка .

**Почему O(n²)?**

* Причина в структуре двойного цикла:
  + Внешний цикл проходит по всем элементам, кроме первого.
  + Внутренний цикл на каждой итерации может пройти всю предыдущую отсортированную часть массива.
* Количество необходимых сравнений приблизительно равно сумме арифметической прогрессии от 1 до n-1, что приближенно выражается формулой , приводящей к общей временной сложности .

**4) Сортировка слиянием** (англ. Merge sort) — алгоритм сортировки, основанный на принципе «разделяй и властвуй».Алгоритм сортировки слиянием состоит из двух основных этапов: **Разделение** — массив рекурсивно делится на две равные (или почти равные) части до тех пор, пока каждый подмассив не будет состоять из одного элемента. **Слияние** — подмассивы объединяются в отсортированном порядке, начиная с самых маленьких подмассивов.

Пример на Java

import java.util.Arrays;

public class MergeSortExample {

*// Главный метод, выполняемый при старте программы*

public static void main(String[] args) {

int[] unsortedArray = {38, 27, 43, 3, 9, 82, 10};

System.out.println("Исходный массив: " + Arrays.toString(unsortedArray));

mergeSort(unsortedArray);

System.out.println("Отсортированный массив: " + Arrays.toString(unsortedArray));

}

*// Рекурсивный метод сортировки слиянием*

public static void mergeSort(int[] arr) {

if (arr.length <= 1) return; *// Базовый случай рекурсии: массив длиной 1 уже отсортирован*

*// Определяем середину массива*

int mid = arr.length / 2;

*// Разделяем массив на левую и правую половины*

int[] leftHalf = new int[mid];

int[] rightHalf = new int[arr.length - mid];

*// Копируем первую половину массива в leftHalf*

System.arraycopy(arr, 0, leftHalf, 0, mid);

*// Копируем вторую половину массива в rightHalf*

System.arraycopy(arr, mid, rightHalf, 0, arr.length - mid);

*// Рекурсивно сортируем обе половинки*

mergeSort(leftHalf);

mergeSort(rightHalf);

*// Объединяем отсортированные половинки обратно в основной массив*

merge(arr, leftHalf, rightHalf);

}

*// Метод объединения двух отсортированных массивов*

private static void merge(int[] arr, int[] leftHalf, int[] rightHalf) {

int i = 0; *// Индекс левого массива*

int j = 0; *// Индекс правого массива*

int k = 0; *// Индекс объединенного массива*

*// Пока оба массива содержат элементы*

while (i < leftHalf.length && j < rightHalf.length) {

if (leftHalf[i] <= rightHalf[j]) {

arr[k++] = leftHalf[i++];

} else {

arr[k++] = rightHalf[j++];

}

}

*// Добавляем оставшиеся элементы из левого массива*

while (i < leftHalf.length) {

arr[k++] = leftHalf[i++];

}

*// Добавляем оставшиеся элементы из правого массива*

while (j < rightHalf.length) {

arr[k++] = rightHalf[j++];

}

}

}

Результат исполнения программы:

При запуске данной программы на экране появится следующее:

Исходный массив: [38, 27, 43, 3, 9, 82, 10]

Отсортированный массив: [3, 9, 10, 27, 38, 43, 82]

Описание работы программы:

1. **Разделение массива на две части:**Первым делом массив делится пополам, и обе части рекурсивно сортируются отдельно друг от друга.
2. **Объединение отсортированных частей:**После того как левая и правая половина отсортированы, осуществляется их объединение. Здесь используется техника "слияния": сначала сравниваются первые элементы обеих половинок, наименьший помещается в общий массив, и процедура продолжается дальше.
3. **Рекурсия заканчивается тогда, когда длина массива равна одному элементу**, поскольку массив из одного элемента уже является отсортированным.

**Принцип работы:**

1. **Базовый случай рекурсии:**Если массив состоит из одного элемента или пуст, он уже отсортирован.
2. **Деление массива:**Массив разделяется на две равные (или почти равные) части.
3. **Рекурсивная сортировка:**Обе полученные части рекурсивно сортируются.
4. **Объединение:**После того как обе части отсортированы, они сливаются вместе в единое целое.

**Под капотом алгоритма:**

* Сначала массив делится на две части.
* Эти две части продолжают дробиться рекурсивно, пока не останутся отдельные элементы.
* Затем начинают происходить операции объединения (merge), при которых отсортированные подпоследовательности соединяются в единую отсортированную последовательность.

**Временная сложность:**

* **Хуже/лучше/среднее время:** Все случаи имеют одинаковую временную сложность , где — количество элементов в массиве.

**Почему ?**

* Массив постоянно делится пополам на каждом уровне рекурсии. Число уровней деления равно .
* На каждом уровне рекурсии происходит операция объединения, которая занимает линейное время .
* Общая временная сложность получается произведением количества уровней () и стоимости объединения на одном уровне (), что и дает итоговую сложность .

**5) Сортировка Шелла** (англ. Shell sort) — алгоритм сортировки, усовершенствованный вариант сортировки вставками. Идея: сравнивать элементы, стоящие не только рядом, но и на определённом расстоянии друг от друга. Это позволяет быстро переставлять далёкие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

public class ShellSortExample {

// Главный метод программы

public static void main(String[] args) {

int[] array = {38, 27, 43, 3, 9, 82, 10};

System.out.println("Исходный массив: " + toString(array)); // Выводим исходный массив

shellSort(array); // Производим сортировку Шелла

System.out.println("Отсортированный массив: " + toString(array)); // Выводим отсортированный массив

}

// Метод сортировки Шелла

public static void shellSort(int[] arr) {

int n = arr.length; // Длина массива

int gap = n / 2; // Начальный интервал (разрыв) между элементами для сравнения

// Продолжаем уменьшать разрыв (gap), пока он не станет равен нулю

while (gap > 0) {

// Сортируем элементы с расстоянием 'gap' друг от друга

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

// Храним текущий элемент

int temp = arr[i];

// Перемещаем предыдущие элементы, пока находим подходящий слот для вставки текущего элемента

int j = i;

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

arr[j] = arr[j - gap]; // Сдвигаем элемент на расстояние gap вправо

j -= gap; // Перемещаемся назад на расстояние gap

}

arr[j] = temp; // Вставляем текущий элемент на свое место

}

gap /= 2; // Уменьшаем разрыв вдвое

}

}

// Вспомогательный метод для преобразования массива в строковое представление

public static String toString(int[] arr) {

StringBuilder sb = new StringBuilder("[");

for (int i = 0; i < arr.length; i++) {

sb.append(arr[i]);

if (i != arr.length - 1) {

sb.append(", ");

}

}

sb.append("]");

return sb.toString();

}

}

После запуска программы на экране появятся следующие строки:

Исходный массив: [38, 27, 43, 3, 9, 82, 10]

Отсортированный массив: [3, 9, 10, 27, 38, 43, 82]

**Анализ:**

1. **Основная идея:**Сначала элементы массива группируются на большом расстоянии друг от друга и сортируются локально. Затем это расстояние сокращается, и вновь проводится сортировка этих новых подгрупп. Наконец, когда дистанция достигает нуля, алгоритм сводится к обычной сортировке вставками, но массив уже значительно ближе к финальной отсортированной форме.
2. **Прохождение массива:**
   * Внешний цикл контролирует уменьшение интервала (gap), на котором происходят операции сортировки.
   * Подгруппы формируются таким образом, что берутся элементы, расположенные на расстоянии "gap" друг от друга.
   * Чем меньше становится "gap", тем точнее выравнивается порядок элементов.
3. **Эффективность:**За счет предварительного сглаживания порядка элементов до стадии обычных вставочных операций, скорость сортировки существенно возрастает по сравнению с классической сортировкой вставками.

**Временная сложность:**

* **Средняя сложность:**В среднем, сортировка Шелла обладает лучшей производительностью по сравнению с простыми методами типа сортировки вставками, выбором или пузырьком. Однако точное значение зависит от последовательности уменьшения промежутков (gap sequence). Типичные оценки варьируют от до или даже лучше, в зависимости от используемой стратегии изменения расстояния между элементами.
* **Худший случай:**Теоретически возможен случай, когда время работы ухудшается до , хотя это крайне редко встречается на практике.

**Почему именно такая сложность?**

* Использование предварительно сглаженной структуры (частичных сортировок на удалённых элементах) значительно снижает среднее количество необходимых перемещений элементов.
* Конкретная оценка производительности сильно зависит от выбранной последовательности изменения промежутков (например, наиболее эффективные последовательности дают лучшее поведение алгоритма).

Сортировка Шелла часто применяется там, где важна устойчивость и относительная простота реализации, обеспечивая заметное ускорение по сравнению с примитивными сортировочными алгоритмами.

6) **Быстрая сортировка (QuickSort)** — алгоритм сортировки, основанный на принципе **«разделяй и властвуй».** **Применение: быстрая сортировка используется в ситуациях, где важна высокая скорость сортировки, особенно когда объём данных велик**

**Пример на Python**

def partition(arr, low, high):  
 *"""  
 Функция разбиения массива для быстрой сортировки.  
 Выбирает последний элемент как опорный.  
 Возвращает индекс, на котором оказался опорный элемент.  
 """* pivot = arr[high] # Выбираем последний элемент как опорный  
 i = low - 1 # Индекс меньшего элемента (элементы слева от i <= pivot)  
  
 for j in range(low, high):  
 # Если текущий элемент меньше или равен опорному  
 if arr[j] <= pivot:  
 i += 1 # Увеличиваем индекс меньшего элемента  
 arr[i], arr[j] = arr[j], arr[i] # Меняем местами arr[i] и arr[j]  
  
 # После цикла, элементы от low до i включительно <= pivot  
 # Элементы от i+1 до high-1 > pivot  
 # Меняем местами опорный элемент (pivot) с arr[i+1]  
 arr[i + 1], arr[high] = arr[high], arr[i + 1]  
 return i + 1 # Возвращаем индекс, на котором теперь стоит pivot  
  
def quick\_sort(arr, low, high):  
 *"""  
 Основная функция быстрой сортировки.  
 Сортирует массив arr от индекса low до high (включительно).  
 """* if low < high:  
 # pi - это индекс разбиения, arr[pi] находится на своем месте  
 pi = partition(arr, low, high)  
  
 # Отсортировать элементы до и после разбиения  
 quick\_sort(arr, low, pi - 1)  
 quick\_sort(arr, pi + 1, high)  
  
def print\_array(arr):  
 *"""Функция для вывода массива."""* print(\*arr)  
  
# Пример использования  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Создаем тестовый массив  
 array = [10, 7, 8, 9, 1, 5]  
  
 print("Исходный массив:")  
 print\_array(array)  
  
 # Сортируем массив  
 quick\_sort(array, 0, len(array) - 1)  
  
 print("Отсортированный массив:")  
 print\_array(array)

**программа выведет**

**Исходный массив:**

**10 7 8 9 1 5**

**Отсортированный массив:**

**1 5 7 8 9 10**

**Принцип работы:**

1. **Выбор опорного элемента:**В данном примере опорный элемент выбирается как последний элемент массива.
2. **Разбиение массива:**Все элементы, меньшие или равные опорному, перемещаются в левую часть массива, а все элементы, большие опорного, — в правую.
3. **Рекурсия:**Процесс повторяется для левой и правой частей массива, пока весь массив не будет отсортирован.

**Временная сложность:**

* **Лучший случай:**Временная сложность равна , когда массив уже отсортирован или почти отсортирован.
* **Средний случай:**Временная сложность также равна , что делает алгоритм одним из самых эффективных для больших массивов.
* **Худший случай:**Временная сложность ухудшается до , если массив уже отсортирован в обратном порядке или все элементы равны. Это происходит из-за неудачного выбора опорного элемента.

**Почему именно такая сложность?**

* В среднем случае алгоритм эффективно разделяет массив на две примерно равные части, что приводит к логарифмической глубине рекурсии.
* В худшем случае, если каждый раз опорный элемент оказывается крайним, массив делится на части размером 1 и N-1, что приводит к квадратичной сложности.

Таким образом, быстрая сортировка является одним из самых эффективных алгоритмов сортировки для больших объемов данных, но требует осторожного выбора опорного элемента для предотвращения ухудшения производительности.

**7)** **Пирамидальная сортировка (Heap Sort, «Сортировка кучей»)** — алгоритм сортировки, основанный на структуре данных **двоичной кучи**. Алгоритм состоит из двух этапов: 1) Построение пирамиды из элементов исходного массива. Например, начиная с нулевого элемента последовательно заполняют уровни дерева слева направо: корень — нулевой элемент массива, его потомки — первый и второй элементы и т. д.. 2) Сортировка на построенной пирамиде. Из кучи извлекают один элемент и помещают его на освободившееся место в конце массива. После каждого извлечения пирамиду обновляют, чтобы сохранить

Пример на Java

public class HeapSortExample {

// Главное исполнение программы

public static void main(String[] args) {

int[] array = {12, 11, 13, 5, 6, 7}; // Исходный массив для сортировки

System.out.println("Исходный массив: " + Arrays.toString(array));

heapSort(array); // Вызов метода сортировки кучей

System.out.println("Отсортированный массив: " + Arrays.toString(array));

}

// Методы для построения max-кучи и самой сортировки

// Строим кучу (heapify)

private static void buildMaxHeap(int[] arr, int heapSize, int rootIndex) {

int largest = rootIndex; // Предполагаем, что корень - максимальный элемент

int leftChild = 2 \* rootIndex + 1; // Левый дочерний узел

int rightChild = 2 \* rootIndex + 2; // Правый дочерний узел

// Проверяем, существует ли левый дочерний узел и больше ли он корня

if (leftChild < heapSize && arr[leftChild] > arr[largest])

largest = leftChild;

// Проверяем, существует ли правый дочерний узел и больше ли он наибольшего узла

if (rightChild < heapSize && arr[rightChild] > arr[largest])

largest = rightChild;

// Если наибольший элемент - не корневой, меняем их местами

if (largest != rootIndex) {

int temp = arr[rootIndex];

arr[rootIndex] = arr[largest];

arr[largest] = temp;

// Рецурсивно восстанавливаем структуру кучи снизу-вверх

buildMaxHeap(arr, heapSize, largest);

}

}

// Полноценная сортировка кучей

public static void heapSort(int[] arr) {

int n = arr.length;

// Преобразование обычного массива в max-кучу

for (int i = n / 2 - 1; i >= 0; i--)

buildMaxHeap(arr, n, i);

// Один за другим извлекаем корни и переносим их в конец массива

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

// Поменять местами последний элемент с корнем

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Восстанавливаем структуру кучи (без учёта последнего элемента)

buildMaxHeap(arr, i, 0);

}

}

}

После запуска программы вы получите следующий результат:

Исходный массив: [12, 11, 13, 5, 6, 7]

Отсортированный массив: [5, 6, 7, 11, 12, 13]

Анализ алгоритма: Пирамидальная сортировка (Heap Sort)

**Принцип работы:**

1. **Преобразование массива в кучу:**Массив превращается в максимальную кучу (max-heap), где каждый родительский узел больше своих детей. То есть самый большой элемент находится в корне кучи.
2. **Удаление корней:**Самый большой элемент (корень) ставится в конец массива, а оставшаяся часть массива повторно преобразуется в кучу, чтобы сохранить свойства кучи. Этот процесс повторяется до тех пор, пока весь массив не окажется отсортированным.

**Временная сложность:**

* **Лучший, средний и худший случаи:**Общая временная сложность алгоритма равна , где — количество элементов в массиве.
* **Почему именно такая сложность?**Формирование начальной кучи требует операций. Удаление каждого элемента и восстановление свойств кучи занимают операций. Поскольку этот процесс повторяется раз, итоговая сложность получается .

Таким образом, сортировка кучей является эффективным алгоритмом, который показывает хорошую производительность даже на больших объёмах данных, обладая стабильной временной сложностью .

8) **Последовательный (линейный) поиск** — это простейший вид поиска заданного элемента на некотором множестве, при котором каждый элемент проверяется по порядку до тех пор, пока не будет найден искомый элемент или не завершится просмотр всех элементов.

Пример на Java

public class LinearSearchExample {

// Основное тело программы

public static void main(String[] args) {

int[] array = {10, 20, 30, 40, 50, 60, 70, 80, 90}; // Исходный массив

int target = 50; // Значение, которое мы ищем

// Выполняем поиск элемента в массиве

int result = linearSearch(array, target);

// Вывод результата поиска

if (result != -1) {

System.out.println("Элемент " + target + " найден на позиции " + result);

} else {

System.out.println("Элемент " + target + " не найден в массиве.");

}

}

// Метод линейного поиска

public static int linearSearch(int[] arr, int x) {

// Проходим по каждому элементу массива

for (int i = 0; i < arr.length; i++) {

// Если нашли искомое значение, возвращаем индекс

if (arr[i] == x) {

return i;

}

}

// Если элемент не найден, возвращаем -1

return -1;

}

}

При запуске программа выведет следующее сообщение:

Элемент 50 найден на позиции 4

Анализ алгоритма: Линейный поиск (Sequential Search).

**Принцип работы:**

* Алгоритм начинает просмотр с первого элемента массива и последовательно сравнивает каждое значение с искомым значением.
* Если совпадение найдено, возвращается индекс элемента.
* Если пройдены все элементы, а совпадение не обнаружено, возвращает специальное значение (-1), обозначающее отсутствие элемента в массиве.

**Временная сложность:**

* **Лучший случай (элемент найден в первом сравнении):**Временная сложность равна .
* **Средний и худший случаи (полный обход массива):**Временная сложность равна , где — количество элементов в массиве.

**Почему именно такая сложность?**

* Алгоритм последовательно рассматривает каждый элемент массива, пока не найдет искомое значение или не просмотрит весь массив.
* Максимальное количество сравнений равно длине массива, поэтому временная сложность прямо пропорциональна размеру массива.

Таким образом, линейный поиск удобен своей простотой и отсутствием требований к предварительной обработке данных, но неэффективен для больших массивов.

9) **Бинарный поиск** (другие названия — двоичный поиск, метод половинного деления, дихотомия) — **тип поискового алгоритма, который последовательно делит пополам заранее отсортированный массив данных, чтобы обнаружить нужный элемент**

def binary\_search(arr, target):  
 *"""  
 Реализация бинарного поиска.  
  
 Параметры:  
 arr (list): Отсортированный список чисел.  
 target (int): Целевое значение для поиска.  
  
 Возвращаемое значение:  
 int: Индекс элемента, если элемент найден, иначе -1.  
 """* low = 0 # Нижняя граница поиска  
 high = len(arr) - 1 # Верхняя граница поиска  
  
 # Продолжаем искать, пока границы не пересеклись  
 while low <= high:  
 mid = (low + high) // 2 # Срединный индекс (целочисленное деление)  
  
 # Если целевой элемент найден, возвращаем его индекс  
 if arr[mid] == target:  
 return mid  
  
 # Если середина больше цели, переходим к поиску в левой половине  
 elif arr[mid] > target:  
 high = mid - 1  
  
 # Если середина меньше цели, переходим к поиску в правой половине  
 else:  
 low = mid + 1  
  
 # Если элемент не найден, возвращаем -1  
 return -1  
  
  
# Основная часть программы  
if \_\_name\_\_ == "\_\_main\_\_":  
 sorted\_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91] # Отсортированный список  
 target\_value = 23 # Цель поиска  
  
 # Выполняем бинарный поиск  
 found\_index = binary\_search(sorted\_list, target\_value)  
  
 # Выводим результат поиска  
 if found\_index != -1:  
 print(f"Значение {target\_value} найдено на позиции {found\_index}")  
 else:  
 print(f"Значение {target\_value} не найдено")

После запуска программа выведет следующее сообщение:

Значение 23 найдено на позиции 5

**Принцип работы:**

1. Диапазон поиска задается границами low и high. Первоначально это начало и конец массива соответственно.
2. На каждом шаге рассчитывается средняя точка mid, и происходит сравнение среднего элемента с искомым.
3. Если средний элемент равен искомому, поиск завершен.
4. Если средний элемент больше искомого, поиск продолжается в левой половине.
5. Если средний элемент меньше искомого, поиск продолжается в правой половине.
6. Такие шаги повторяются, пока не останется один элемент или не закончится диапазон поиска.

**Временная сложность:**

* **Лучший случай (искомый элемент находится в середине):**Временная сложность равна .
* **Средний и худший случаи (искаемый элемент отсутствует или находится на краю):**Временная сложность равна , где — количество элементов в массиве.

**Почему именно такая сложность?**

* Алгоритм каждый раз делит диапазон поиска ровно пополам, что гарантирует экспоненциальное снижение размеров поиска.
* Так как после каждого шага размер области поиска уменьшается вдвое, максимальное количество шагов ограничено высотой двоичного дерева, равной , где — количество элементов в массиве.
* Соответственно, количество операций поиска зависит от логарифма размера массива, что дает временную сложность .

Таким образом, бинарный поиск чрезвычайно эффективен для быстрого поиска в больших отсортированных списках.

**10) Интерполяционный поиск (интерполирующий поиск) — алгоритм поиска, который основан на принципе поиска в телефонной книге или, например, в словаре. Суть алгоритма: вместо сравнения каждого элемента с искомым, как при линейном поиске, интерполяционный поиск производит предсказание местонахождения элемента. Поиск происходит подобно двоичному поиску, но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента**

def interpolation\_search(arr, target):  
 *"""  
 Реализация интерполяционного поиска.  
  
 Параметры:  
 arr (list): Отсортированный список чисел.  
 target (int): Целевой элемент для поиска.  
  
 Возвращаемое значение:  
 int: Индекс элемента, если элемент найден, иначе -1.  
 """* low = 0 # Нижняя граница поиска  
 high = len(arr) - 1 # Верхняя граница поиска  
  
 # Продолжаем поиск, пока нижняя граница не превысила верхнюю  
 while low <= high and target >= arr[low] and target <= arr[high]:  
 # Если верхняя и нижняя границы совпадают, элемент точно не найден  
 if arr[low] == arr[high]:  
 if arr[low] == target:  
 return low  
 return -1  
  
 # Интерполяционная формула расчета приблизительного местоположения  
 pos = low + ((target - arr[low]) \* (high - low)) // (arr[high] - arr[low])  
  
 # Если целевой элемент найден, возвращаем его индекс  
 if arr[pos] == target:  
 return pos  
  
 # Если промежуточный элемент больше цели, сужаем верхнее окно поиска  
 elif arr[pos] > target:  
 high = pos - 1  
  
 # Если промежуточный элемент меньше цели, сужаем нижнее окно поиска  
 else:  
 low = pos + 1  
  
 # Если элемент не найден, возвращаем -1  
 return -1  
  
  
# Основная часть программы  
if \_\_name\_\_ == "\_\_main\_\_":  
 sorted\_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91] # Отсортированный список  
 target\_value = 23 # Цель поиска  
  
 # Выполняем интерполяционный поиск  
 found\_index = interpolation\_search(sorted\_list, target\_value)  
  
 # Выводим результат поиска  
 if found\_index != -1:  
 print(f"Значение {target\_value} найдено на позиции {found\_index}")  
 else:  
 print(f"Значение {target\_value} не найдено")

После запуска программа выведет следующее сообщение:

Значение 23 найдено на позиции 5

**Принцип работы:**

1. **Итерационные шаги:**
   * Алгоритм определяет приблизительную позицию целевого элемента исходя из предположения, что распределение элементов равномерно.
   * Новая позиция рассчитывается по формуле, учитывающей соотношение между текущими граничными элементами и целью.
   * По полученной позиции алгоритм либо находит элемент, либо сужает диапазон поиска.
2. **Расчет позииции:**
   * Использует формулу для оценки вероятности местонахождения искомого элемента:
   * Это позволяет быстрее находить цель, если распределение действительно близко к равномерному.

**Временная сложность:**

* **Лучший случай (идеально равномерное распределение):**Временная сложность равна , где — количество элементов в массиве.
* **Средний случай (равномерное распределение):**Временная сложность тоже близка к .
* **Худший случай (очень неравномерное распределение):**Временная сложность может ухудшиться до , если распределение элементов плохо предсказуемо или неоднородно.

**Почему именно такая сложность?**

* В отличие от бинарного поиска, который гарантированно уменьшает область поиска вдвое на каждом шаге, интерполяционный поиск потенциально способен гораздо быстрее добраться до нужного элемента при условии равномерного распределения.
* Но если распределение элементов резко меняется или является непредсказуемым, точность оценки позиций снижается, и алгоритм может оказаться менее эффективным, вплоть до прямой зависимости от размера массива.

Таким образом, интерполяционный поиск отлично проявляет себя на однородных данных, демонстрируя высокую скорость поиска.

**11) Поиск Фибоначчи** — это метод поиска в отсортированном массиве с использованием алгоритма «разделяй и властвуй», который сужает возможные местоположения с помощью чисел Фибоначчи. Вместо того чтобы вычислять середину, как в бинарном поиске, этот метод выбирает точки сравнения в списке.

def fibonacci\_search(arr, target):  
 *"""  
 Реализация поиска Фибоначчи.  
  
 Параметры:  
 arr (list): Отсортированный список чисел.  
 target (int): Целевой элемент для поиска.  
  
 Возвращаемое значение:  
 int: Индекс элемента, если элемент найден, иначе -1.  
 """* def fibonacci\_numbers(limit):  
 *"""Генерирует ряд чисел Фибоначчи, ограничивая максимум указанным пределом."""* fib\_sequence = [0, 1]  
 next\_fib = fib\_sequence[-1] + fib\_sequence[-2]  
 while next\_fib <= limit:  
 fib\_sequence.append(next\_fib)  
 next\_fib = fib\_sequence[-1] + fib\_sequence[-2]  
 return fib\_sequence  
  
 # Генерируем необходимые числа Фибоначчи  
 fib\_seq = fibonacci\_numbers(len(arr))  
  
 offset = -1 # Начальная позиция  
 m = len(fib\_seq) - 1 # Индекс самого большого числа Фибоначчи  
 length = len(arr) # Длина массива  
  
 # Ищем, пока остаётся пространство для поиска  
 while m > 0:  
 # Выбираем минимальное из текущего числа Фибоначчи и длины массива  
 i = min(offset + fib\_seq[m], length - 1)  
  
 # Если элемент меньше цели, переходим в большую часть массива  
 if arr[i] < target:  
 offset = i  
 m -= 1  
  
 # Если элемент больше цели, переходим в меньшую часть массива  
 elif arr[i] > target:  
 m -= 2  
  
 # Если элемент найден, возвращаем его индекс  
 else:  
 return i  
  
 # Если элемент не найден, возвращаем -1  
 return -1  
  
  
# Основная часть программы  
if \_\_name\_\_ == "\_\_main\_\_":  
 sorted\_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91] # Отсортированный список  
 target\_value = 23 # Цель поиска  
  
 # Выполняем поиск Фибоначчи  
 found\_index = fibonacci\_search(sorted\_list, target\_value)  
  
 # Выводим результат поиска  
 if found\_index != -1:  
 print(f"Значение {target\_value} найдено на позиции {found\_index}")  
 else:  
 print(f"Значение {target\_value} не найдено")

После запуска программа выведет следующее сообщение:

Значение 23 найдено на позиции 5

**Принцип работы:**

1. Используя числа Фибоначчи, мы определяем точку разбиения массива, исходя из соотношения индексов соседних чисел Фибоначчи.
2. На каждом шаге выбирается ближайшая возможная точка разбиения, соответствующая числу Фибоначчи, которая помогает определить следующую область поиска.
3. Если выбранный элемент больше искомого, мы исследуем левую часть массива, иначе — правую.
4. Такой подход теоретически позволяет ускорить поиск, если данные хорошо соответствуют распределению, характерному для чисел Фибоначчи.

**Временная сложность:**

* **Лучший случай (точка поиска идеально соответствует ряду Фибоначчи):**Временная сложность равна , где — золотое сечение.
* **Средний и худший случаи:**Хотя в большинстве случаев алгоритм работает достаточно быстро, худшая временная сложность остается , где — количество элементов в массиве.

**Почему именно такая сложность?**

* На каждом шаге алгоритм разбивает массив на неравные части, одна из которых связана с числом Фибоначчи, позволяя быстрее локализовать поиск, если распределение данных благоприятствует такому подходу.
* Тем не менее, несмотря на потенциал ускорения, в худших случаях (особенно при неблагоприятных распределениях) алгоритм ведёт себя сходно с бинарным поиском, сохраняя логарифмическое время работы.

Таким образом, поиск Фибоначчи полезен в ситуациях, когда данные имеют особое распределение, схожее с рядом Фибоначчи, и в некоторых случаях может обеспечить преимущество над классическим бинарным поиском.