Quicksort — достаточно эффективный алгоритм. Его реализация новичку может показаться несколько сложной, хотя сам принцип, заложенный в его основу очень прост и стар, как мир. Это “разделяй и властвуй”. Вот как он работает.

1. Для начала нужно выбрать опорный элемент в массиве. Чаще всего это средний элемент, но также можно выбрать “опору” случайным образом. Проще всего — просто выбрать первый или последний элемент.
2. Начинаем разделение. Делим массив на две части таким образом, что элементы, которые меньше опорного, перемещаются влево от него, а те, которые больше — вправо.
3. Повторение процесса разделения. Повторяем процесс рекурсивно для левой и правой части массива до тех пор, пока каждая часть не будет состоять из одного элемента.

Перед реализацией алгоритма
Прежде, чем начинать программировать, обратите внимание на пару важных моментов: **Граничные условия.** Будьте осторожны с границами массива во избежание ошибок выхода за пределы массива. Иными словами перед разделением или рекурсивным вызовом Quicksort убедитесь, что диапазон (верхний и нижний индексы, low и high) валиден. Это означает, что low должен быть меньше high, и оба индекса должны находиться в пределах границ массива. **Стек переполнения.** Из-за рекурсии есть риск переполнения стека вызовов, особенно на больших массивах. Как этого избежать? Например сначала рекурсивно обрабатывать меньший из двух подмассивов, полученных после разделения. **Выбор опорного элемента (pivot) в алгоритме Quicksort** существенно влияет на его производительность. Вот несколько распространенных способов выбора опорного элемента.

- Первый или последний элемент. Самый простой способ, но он может привести к худшей производительности, особенно если массив уже частично или полностью отсортирован.
- Выбор среднего элемента массива как опорного может помочь избежать худших сценариев производительности в некоторых случаях. Это метод вычисления индекса среднего элемента как (low + high) / 2.
- Еще один подход — выбрать медиану из трех элементов (обычно первый, средний и последний) массива. Этот метод обычно обеспечивает более сбалансированное разделение, чем простой выбор первого или последнего элемента.
- Выбор случайного элемента в качестве опорного может снизить вероятность худшего случая производительности, особенно для массивов, которые могут быть уже отсортированы или иметь определенную структуру.
- Для очень больших массивов может использоваться метод называемый "медиана медиан". Это сложный метод, по которому опорный элемент выбирается более сбалансировано, но он требует дополнительных вычислений.

В целом, выбор метода зависит от конкретных условий и требований к производительности. Для начала давайте реализуем алгоритм с последним элементом в качестве опорного — для простоты.

Реализация алгоритма быстрой сортировки
Теперь давайте реализуем быструю сортировку на Java. **import** java.util.Arrays;

```
public class QuickSort {

  public static void main(String[] args) {
    int[] array = {17, 14, 15, 28, 6, 8, -6, 1, 3, 18};
    System.out.println("Unsorted Array: " + Arrays.toString(array));
    quickSort(array, 0, array.length - 1);
    System.out.println("  Sorted Array: " + Arrays.toString(array));


  }


  public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = partition(arr, low, high);


      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
    }
  }


  private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
        i++;


        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }


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


    return i + 1;
  }
}
```

}} Вывод программы следующий:

Unsorted Array: [17, 14, 15, 28, 6, 8, -6, 1, 3, 18] Sorted Array: [-6, 1, 3, 6, 8, 14, 15, 17, 18, 28]

Давайте разберемся, что тут происходит. Рекурсивный метод quickSort на вход принимает массив, а также нижний (low) и верхний (high) индексы. Если low меньше high, происходит разделение массива с помощью метода partition, а затем рекурсивно вызывается quickSort для левой и правой части. Метод partition определяет положение опорного элемента (здесь выбран последний элемент массива) и переставляет элементы так, чтобы элементы меньше опорного находились слева от него, а большие — справа. Этот метод возвращает индекс опорного элемента после перестановки. Теперь давайте изменим реализацию алгоритма таким образом, чтобы в качестве опорного элемента был выбран средний элемент массива. Import java.util.Arrays;

```
public class QuickSort {
  public static void main(String[] args) {
    int[] array = {17, 14, 15, 28, 6, 8, -6, 1, 3, 18};
    System.out.println("Unsorted Array: " + Arrays.toString(array));
    quickSort(array, 0, array.length - 1);
    System.out.println("  Sorted Array: " + Arrays.toString(array));
  }

  public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
    }
  }

  private static int partition(int[] arr, int low, int high) {
    // Выбор среднего элемента в качестве опорного
    int middle = low + (high - low) / 2;
    int pivot = arr[middle];

    // Обмен опорного элемента с последним, чтобы использовать существующую
логику
    int temp = arr[middle];
    arr[middle] = arr[high];
    arr[high] = temp;

    int i = (low - 1);
    for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
        i++;
```

```
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

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

    return i + 1;
  }
```
} Метод, реализованный таким образом может работать быстрее, чем такой же метод, но с последним элементом в качестве опорного на определенных наборах данных, особенно на частично отсортированных массивах.

Итеративная реализация алгоритма быстрой сортировки
Быстрая сортировка в Java-реализации может быть не только рекурсивной, но и итеративной. Давайте попробуем реализовать такой вариант метода. В этой реализации используется стек для отслеживания индексов подмассивов, которые нужно отсортировать. **import** java.util.Arrays;
import java.util.Stack;

```
public class QuickSort {

  static void quickSort(int[] arr, int l, int h) {
    if (arr == null || arr.length == 0)
      return;


    if (l >= h)
      return;


    Stack stack = new Stack<>();
    stack.push(l);
    stack.push(h);


    while (!stack.isEmpty()) {
      h = stack.pop();
      l = stack.pop();


      int pivotIndex = partition(arr, l, h);


      if (pivotIndex - 1 > l) {
        stack.push(l);
```

```
            stack.push(pivotIndex - 1);
        }


        if (pivotIndex + 1 < h) {
            stack.push(pivotIndex + 1);
            stack.push(h);
        }
    }
}


  private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
        }
    }


    int temp = arr[i];
    arr[i] = arr[high];
    arr[high] = temp;
    return i;
  }


  public static void main(String[] args) {
    int[] array = {17, 14, 15, 28, 6, 8, -6, 1, 3, 18};
    System.out.println("Unsorted Array: " + Arrays.toString(array));
    quickSort(array, 0, array.length - 1);
    System.out.println("  Sorted Array: " + Arrays.toString(array));;
    }
  }
```

Вывод программы будет такой же, как и в предыдущих случаях.

Сложность алгоритма быстрой сортировки

Сложность алгоритма — это мера оценки эффективности алгоритма. Сложность часто выражается в терминах "O-нотации" (Big O notation), которая описывает худший случай поведения алгоритма с увеличением размера входных данных. Она предоставляет общее представление о том, как масштабируется производительность алгоритма при увеличении размера входных данных. Важно понимать, что O-нотация не дает точных временных значений выполнения, а скорее показывает тенденцию изменения производительности. Сложность алгоритма Quicksort зависит от многих

факторов, включая выбор опорного элемента и распределение входных данных. В общем случае сложность быстрой сортировки такова:

- Лучший случай. O(nlogn). Алгоритм делит массив на равные части, что приводит к наиболее эффективному разделению и сокращению времени выполнения.
- Средний случай. Также O(nlogn). Для случайных данных и при хорошем выборе опорного элемента быстрая сортировка чаще всего достигает этой сложности.
- Худший случай. $O(n^2)$. Это уже не так хорошо. Такое происходит, когда каждое разделение делит массив на части размером '1' и 'n-1', что чаще всего случается при выборе опорного элемента как наименьшего или наибольшего элемента в массиве. Например, если массив уже отсортирован и в качестве опорного выбирается первый или последний элемент, алгоритм будет иметь худшую производительность.