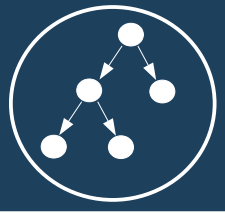


Data Structures and Algorithms

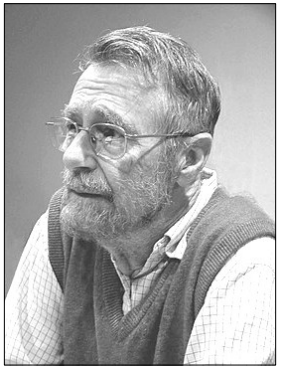
Алгоритмы.
Быстрая сортировка. Трехчастное разбиение

Трехчастное разбиение

На практике в большинстве случаев в сортируемой последовательности содержатся элементы с одинаковыми ключами сортировки. Но при разбиении последовательности на две части элементы равные опорному элементу (по сути в дальнейшей сортировке не нуждающиеся) все равно попадают в левую или правую часть и подвергаются сортировке. Этой проблемы можно избежать если разбивать последовательность на три части (элементы меньше опорного, элементы больше опорного, элементы равны опорному) и для дальнейшей сортировки вызывать только подпоследовательности с элементами меньше опорного и элементами больше опорного.

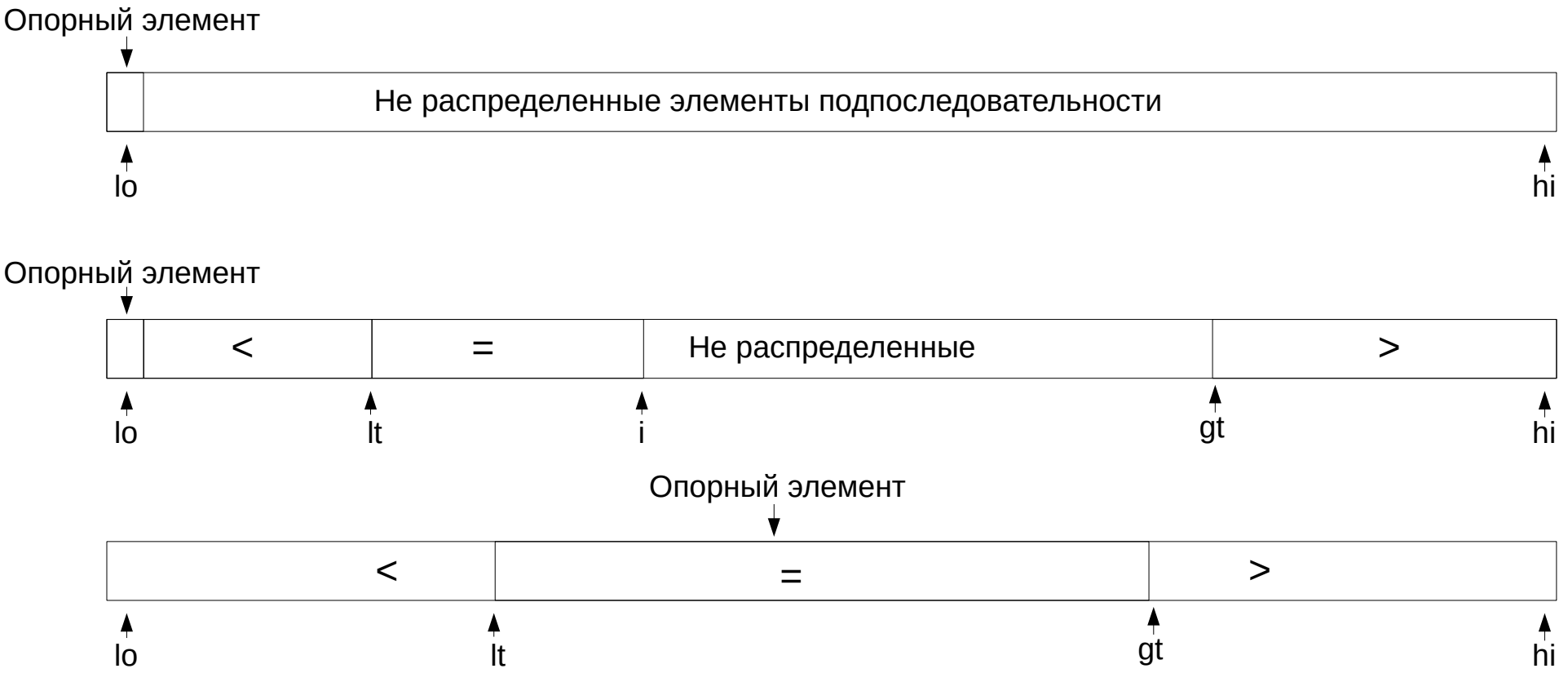
Трехчастное разбиение Дейкстры

Один из простых для реализаций алгоритм троичного разбиения был предложен Дейкстрой. Он предложил использовать три индекса lt, i, gt. После обработки последовательности a[lo..lt-1] меньше опорного элемента, a[lt..gt-1] равны, a[gt..hi] больше.



Edsger Wybe Dijkstra
1930-2002

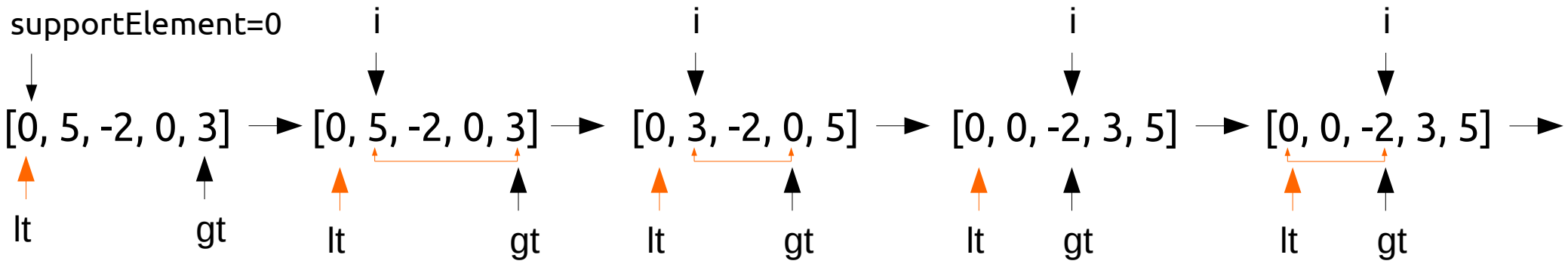
Объяснение принципа разбиения подпоследовательности



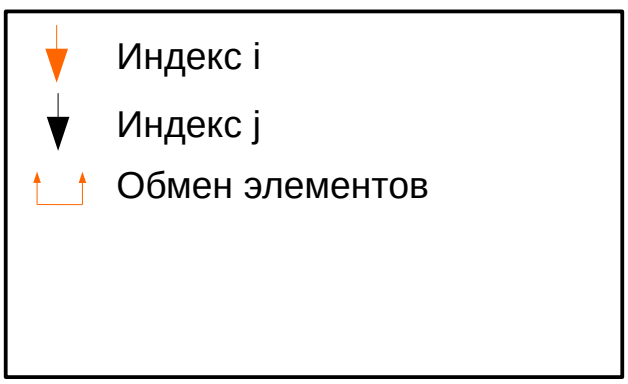
Разбиение Дейкстры

1) В качестве опорного элемента выбирается первый элемент последовательности (supportElement). Объявляется переменные для хранения индексов (в дальнейшем i=lo+1, lt=lo, gt=hi).
2) Выполняем проход по последовательности. Если элемент последовательности a[i]
    - Меньше опорного. Обмен a[lt], a[i] увеличиваем lt и i на единицу
    - Больше опорного. Обмен a[gt],a[i] уменьшаем gt на единицу
    - Равен опорному. Увеличиваем i на единицу
3) Вернуть lt и gt

Графическое пояснение алгоритма разбиения Дейкстры



Обозначения



Python

Реализация алгоритма на Python

Python

Функция для разбиения подпоследовательностей

```
def partition(sequince, lo_index, hi_index):
    support_element = sequince[lo_index]
    i = lo_index + 1
    gt = hi_index
    lt = lo_index
    while i <= gt:
        if sequince[i] < support_element:
            sequince[i], sequince[lt] = sequince[lt], sequince[i]
            i += 1
            lt += 1
        elif sequince[i] > support_element:
            sequince[i], sequince[gt] = sequince[gt], sequince[i]
            gt -= 1
        else:
            i += 1
    return [lt, gt]
```

Python

Реализация алгоритма быстрой сортировки

```
def quick_sort(sequince, lo_index=None, hi_index=None):
    if lo_index is None:
        lo_index = 0
    if hi_index is None:
        hi_index = len(sequince)-1
    if lo_index >= hi_index:
        return None
    h = partition(sequince, lo_index, hi_index)
    quick_sort(sequince, lo_index, h[0]-1)
    quick_sort(sequince, h[1]+1, hi_index)
```

Java

Реализация алгоритма на Java

Метод для разбиения подмассивов

```
public static int[] breakPartition(int[] array, int lo, int hi) {
    int i = lo + 1;
    int lt = lo;
    int gt = hi;
    int supportElement = array[lo];
    for (; i <= gt;) {
        if (array[i] < supportElement) {
            swap(array, i, lt);
            i += 1;
            lt += 1;
        } else if (array[i] > supportElement) {
            swap(array, i, gt);
            gt -= 1;
        } else {
            i += 1;
        }
    }
    return new int[] { lt, gt };
}
```

Метод для для обмена местами элементов массива

```
public static void swap(int[] array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

Реализация алгоритма на Java

Метод для запуска рекурсивного метода сортировки

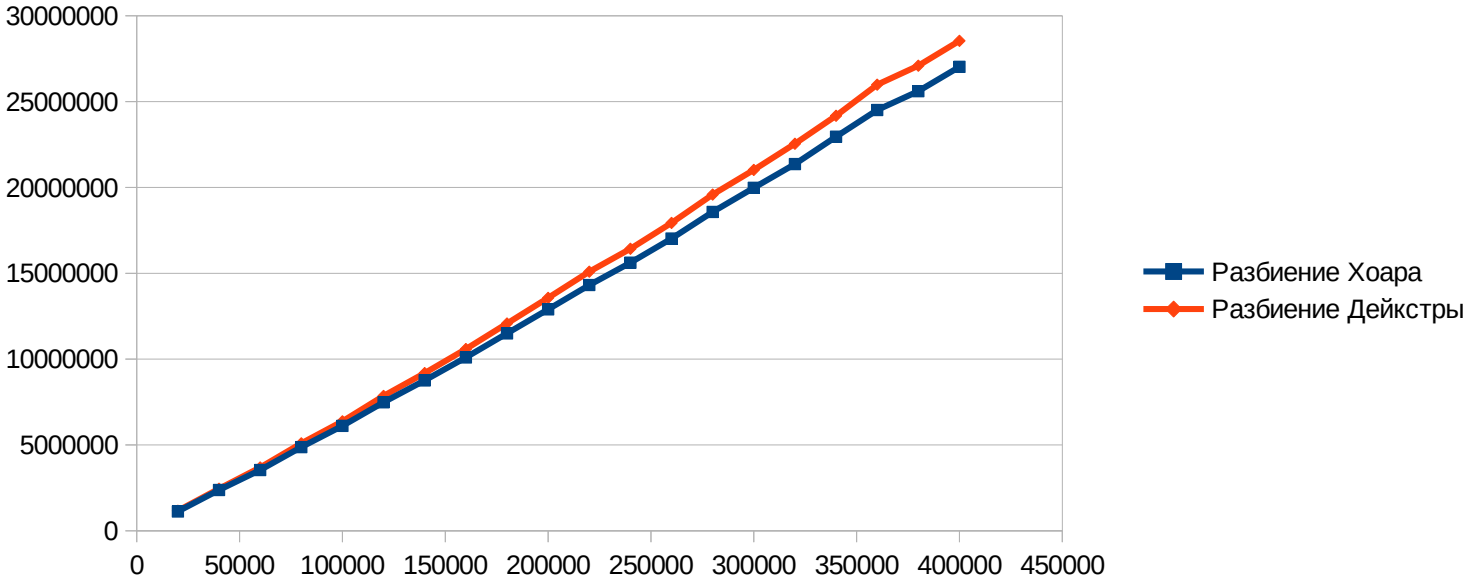
```
public static void quickSort(int[] array) {
    quickSort(array, 0, array.length - 1);
}
```

Рекурсивный метод реализующий быструю сортировку

```
public static void quickSort(int[] array, int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    int[] part = breakPartition(array, lo, hi);
    quickSort(array, lo, part[0] - 1);
    quickSort(array, part[1] + 1, hi);
}
```

Вычислительный эксперимент

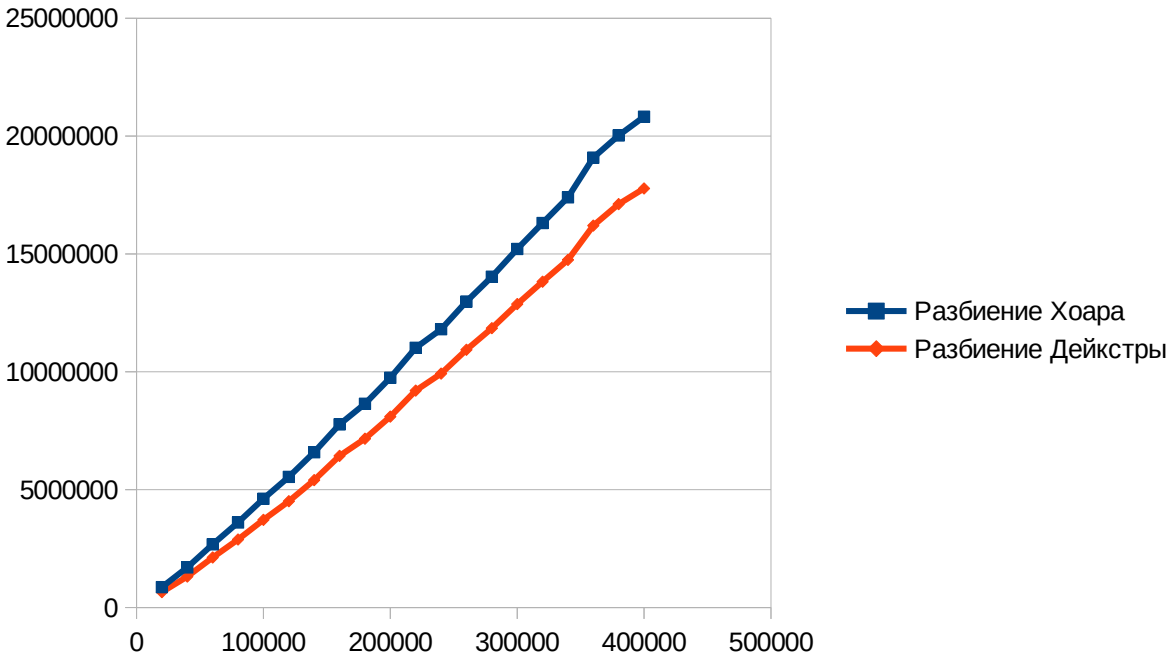
Для определения оптимальности использования трехчастного разбиения Дейкстры проведем вычислительный эксперимент. Сравним скорость сортировки массивов с использованием разбиения Хоара и разбиения Дейкстры. Таких сравнений будет несколько. В одних элементы массива будут примерно уникальны, в других массивы будут содержать совпадающие элементы. Алгоритмы реализованы на Java.



Элементы массива примерно уникальны

Вычислительный эксперимент

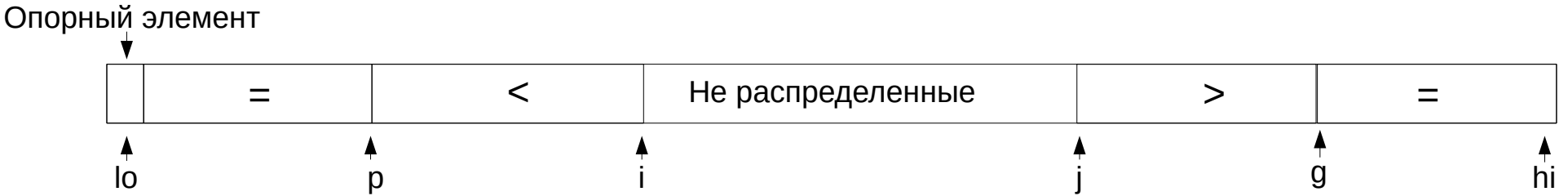
Массивы содержат одинаковые элементы.



И хотя создается впечатление что разбиение Дейкстры обладает явным преимуществом над разбиением Хоара, это не так. Как только в массиве появятся частично упорядоченные элементы работа разбиения Дейкстры очень быстро замедлится. При применении оптимизаций все преимущества перед разбиением Хоара теряются.

Трехчастное разбиение предложенное Бентли и Макилроем

Интересный метод трехчастного разбиения (three-way partitioning) был предложен в 1993 году Бентли и Макилроем (Bentley and McIlroy). Он представляет собой следующую модификацию стандартной схемы разбиения: помещаем ключи из левой части подпоследовательности равные центральному элементу, в левый конец , а ключи из правой части подпоследовательности, равные центральному элементу - в правый конец.



После того как i и j пересекутся то вернуть элементы с краев в центр.

Разбиение Бентли и Макилроя

1) В качестве опорного элемента выбирается первый элемент последовательности (supportElement). Объявляются переменные для хранения индексов (в дальнейшем i=lo+1, j=hi, p=lo, g=h+1). Переходим к пункту **2**.
2) Начиная со второго элемента ищем элемент такой что a[i] ≥ supportElement, начиная с конца последовательности ищем такой элемент a[j] ≤ supportElement. Если i ≥ j. Переходим к **5**. В противном случае переходим к **3**.
3) Выполняем обмен a[i], a[j]. Если после обмена:
   - a[i]=supportElement. Увеличиваем p на единицу. Выполняем обмен a[i], a[p]
   - a[j]=supportElement. Уменьшаем g на единицу. Выполняем обмен a[j], a[g]

   Переходим к **4**.
4) Увеличиваем i на единицу, уменьшаем j на единицу. Переходим к **2**.
5) Производим обмен от lo до p индекса с элементом на индексе j, при каждом обмене j уменьшаем на единицу. Производим обмен от hi до g индекса (в обратном порядке) с элементом на индексе i при каждом обмене увеличиваем i. **Вернуть** {j,i}

Python

Реализация алгоритма на Python

Python

Функция для разбиения подпоследовательностей

```
def partition(sequince, lo_index, hi_index):
    support_element = sequince[lo_index]
    i = lo_index + 1
    g = hi_index + 1
    p = lo_index
    j = hi_index
    while True:
        while i < hi_index and sequince[i] < support_element:
            i += 1
        while sequince[j] > support_element:
            j -= 1
        if i >= j:
            if i == j and sequince[i] == support_element:
                p += 1
                sequince[i], sequince[p] = sequince[p], sequince[i]
            break
        sequince[i], sequince[j] = sequince[j], sequince[i]
        if sequince[i] == support_element:
            p += 1
            sequince[i], sequince[p] = sequince[p], sequince[i]
        if sequince[j] == support_element:
            g -= 1
            sequince[j], sequince[g] = sequince[g], sequince[j]
        i += 1
        j -= 1
    for k in range(lo_index, p+1):
        sequince[k], sequince[j] = sequince[j], sequince[k]
        j -= 1
    for k in range(hi_index, g+1, -1):
        sequince[i], sequince[k] = sequince[k], sequince[i]
        i += 1
    return [j, i]
```

Python

Реализация алгоритма быстрой сортировки

```
def quick_sort(sequince, lo_index=None, hi_index=None):
    if lo_index is None:
        lo_index = 0
    if hi_index is None:
        hi_index = len(sequince)-1
    if lo_index >= hi_index:
        return None
    h = partition(sequince, lo_index, hi_index)
    quick_sort(sequince, lo_index, h[0])
    quick_sort(sequince, h[1], hi_index)
```

Java

Реализация алгоритма на Java

Метод для разбиения подмассивов

```
public static int[] breakPartition(int[] array, int lo, int hi) {
        int i = lo + 1;
        int p = lo;
        int j = hi;
        int g = hi + 1;
        int supportElement = array[lo];
        for (;;) {
                for (; i < hi && array[i] < supportElement;) {
                        i++;
                }
                for (; array[j] > supportElement;) {
                        j--;
                }
                if (i >= j) {
                        if (i == j && array[i] == supportElement) {
                                swap(array, i, ++p);
                        }
                        break;
                }
                swap(array, i, j);
                if (array[i] == supportElement) {
                        swap(array, i, ++p);
                }
                if (array[j] == supportElement) {
                        swap(array, j, --g);
                }
                i++;
                j--;
        }
        for (int k = lo; k <= p; k++) {
                swap(array, k, j--);
        }
        for (int k = hi; k >= g; k--) {
                swap(array, k, i++);
        }
        return new int[] { j, i };
}
```

Метод для для обмена местами элементов массива

```
public static void swap(int[] array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

Реализация алгоритма на Java

Метод для запуска рекурсивного метода сортировки

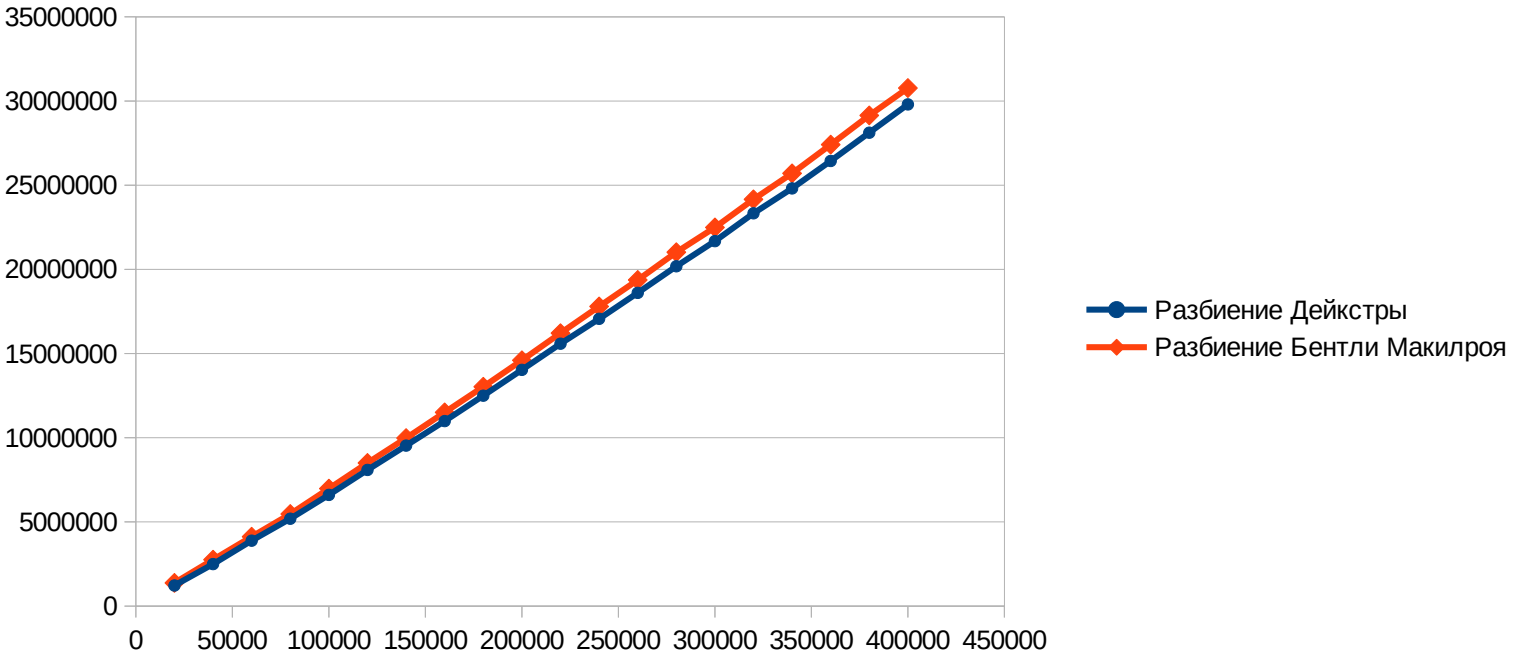
```
public static void quickSort(int[] array) {
    quickSort(array, 0, array.length - 1);
}
```

Рекурсивный метод реализующий быструю сортировку

```
public static void quickSort(int[] array, int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    int[] part = breakPartition(array, lo, hi);
    quickSort(array, lo, part[0]);
    quickSort(array, part[1], hi);
}
```

Вычислительный эксперимент

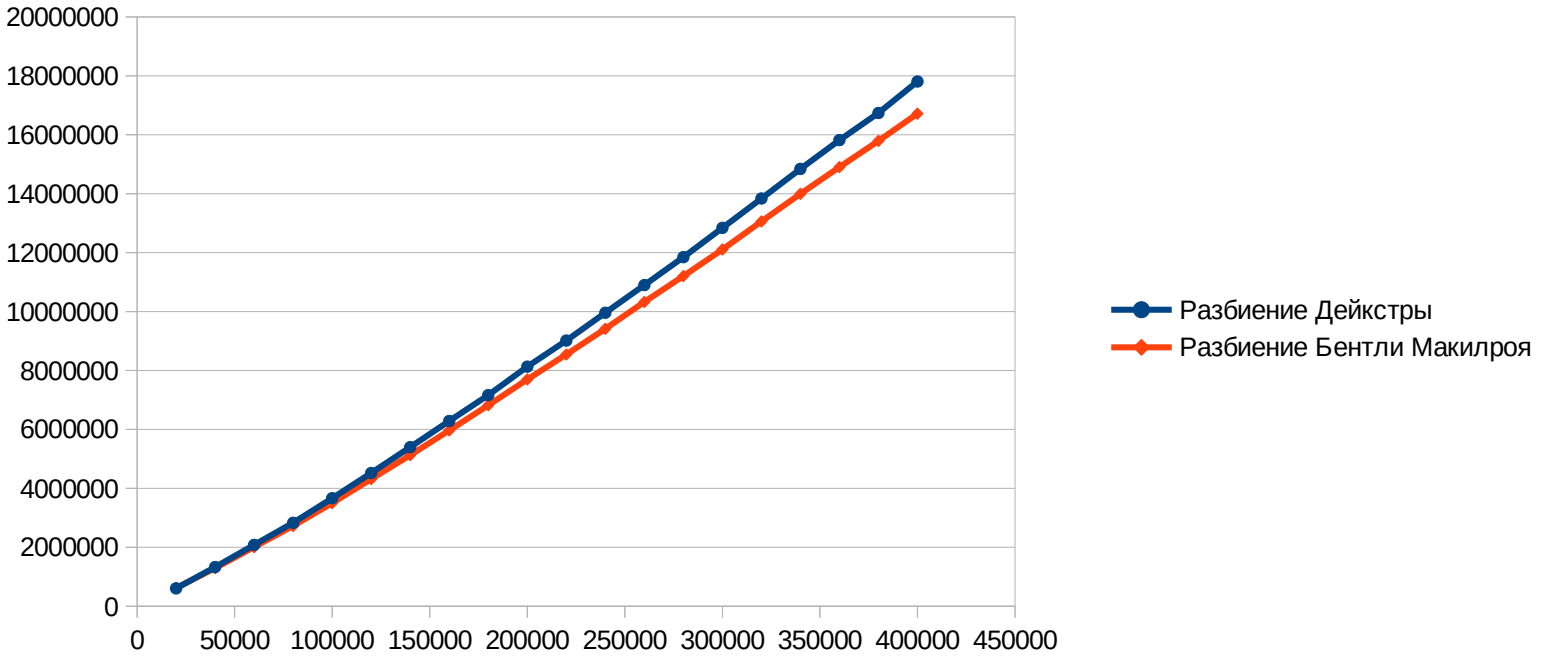
Проведем замеры производительности алгоритмов разбиения Дейкстры и Бентли Макилроя. Для этого замерим время необходимое для сортировки массивов разных размеров. На рисунке приведена зависимость времени сортировки от размера массива.



Элементы массива примерно уникальны

Вычислительный эксперимент

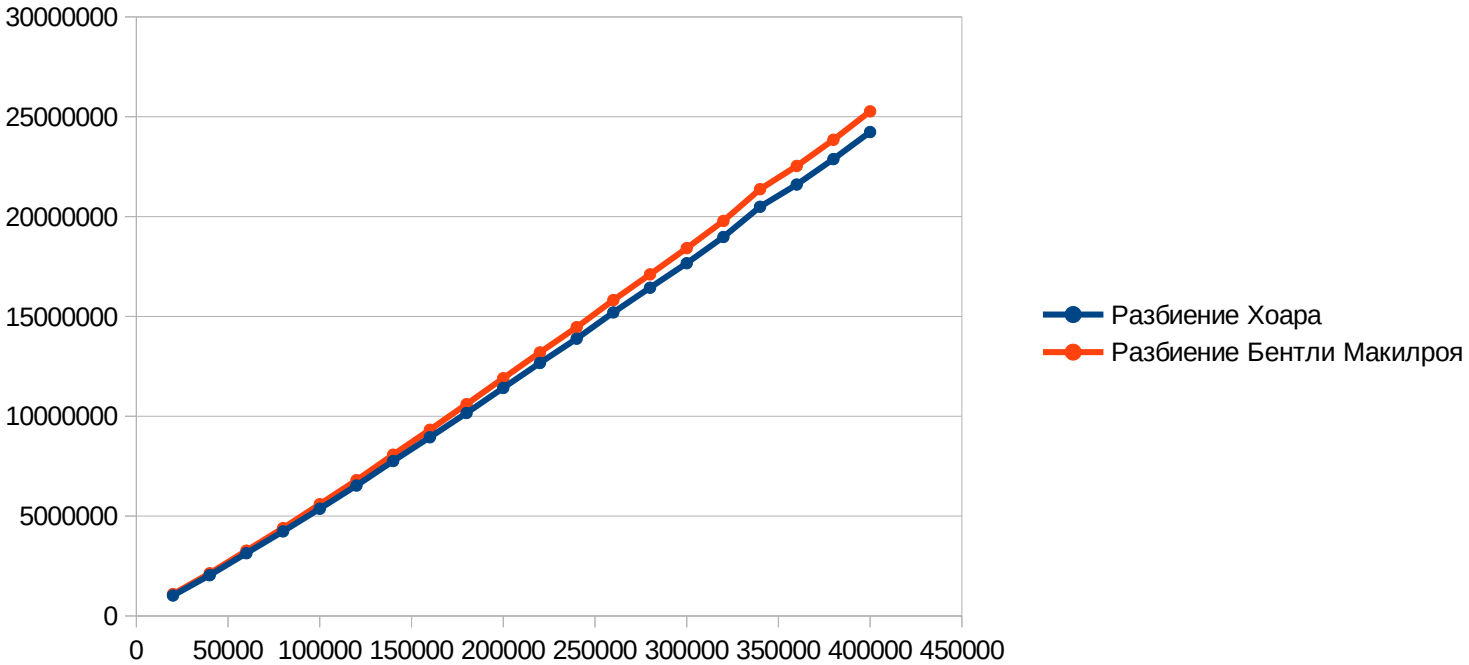
Проведем замеры производительности алгоритмов разбиения Дейкстры и Бентли Макилроя. Для этого замерим время необходимое для сортировки массивов разных размеров. На рисунке приведена зависимость времени сортировки от размера массива.



В массиве часто встречаются одинаковые значения

Вычислительный эксперимент

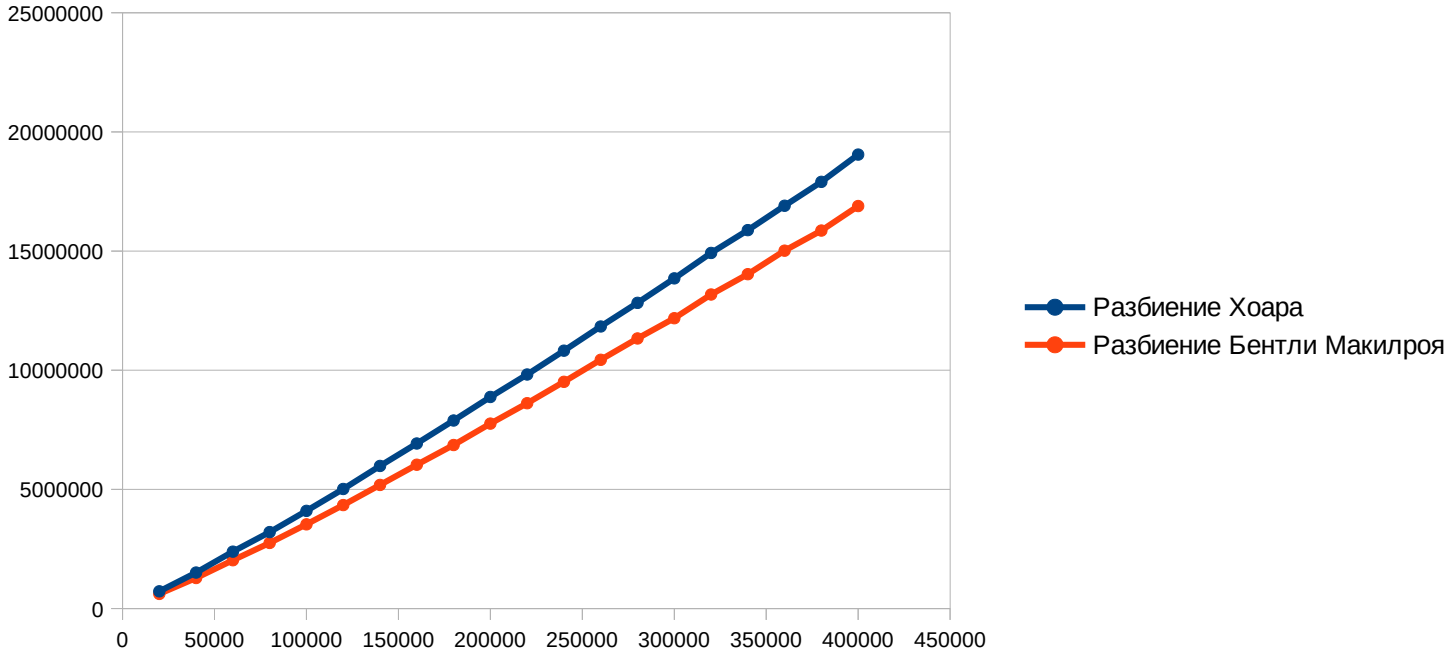
Применим оптимизации указанные ранее (переход на сортировку вставкой) вычисление медианы (девятки Тьюки) к алгоритмам на основании разбиения Хоара, и разбиения Бентли Макилроя. На графике вы видите зависимость времени сортировки от размера массива.



Элементы массива примерно уникальны

Вычислительный эксперимент

Применим оптимизации указанные ранее (переход на сортировку вставкой) вычисление медианы (девятки Тьюки) к алгоритмам на основании разбиения Хоара, и разбиения Бентли Макилроя. На графике вы видите зависимость времени сортировки от размера массива.



В массиве часто встречаются одинаковые значения

Список литературы

1) Д. Кнут. Искусство программирования. Том 3. «Сортировка и поиск», 2-е изд. ISBN 5-8459-0082-4
2) Роберт Седжвик, Кевин Уэйн «Алгоритмы на java 4-е издание» Пер. с англ. - М. : ООО "И.Д. Вильямс", 2013. ISBN 978-5-8459-1781-2.