Быстрая сортировка. Разбиения: Ломуто, Хоара, Толстое. Среднее время работы алгоритма при случайном выборе pivot

Соль Михаил

1 Алгоритм быстрой сортировки

1. Выберем опорный элемент pivot (важно: значение должно присутствовать в массиве, например можно брать самый правый/самый левый/центральный)
2. Разобьем массив на две части: в одной все элементы больше или равны pivot, в другой меньше
3. Поставим pivot на границе раздела (во избежание бесконечной рекурсии)
4. Рекурсивно решим две образованные подзадачи (шаги 1-3)

2 Разбиение Ломуто

1. Заведем два указателя l, i. Изначально оба показывают на начало массива
2. Пойдём слева направо, i – будет показывать границу, до которой все элементы меньше опорного
3. Таким образом для разбиения хватит одного цикла

Вариант реализации разбиения Ломуто на Java

```java
public class LomutoPartition {
  public static int lomutoPartition(int[] a, int l, int r) {
    int pivot = a[l + (r - l) / 2];
    swap(a, l + (r - l) / 2, r); // swapping elements in a by index
    int i = l;
    for (int j = l; j < r; j++) {
      if (a[j] < pivot) {
        swap(a, i, j);
        i++;
      }
    }
    swap(a, i, r);
    return i;
  }
}
```

3 Разбиение Хоара

1. Заведем два указателя l, r, левый начинает с начала, правый с конца
2. Устремим их на встречу друг другу
3. r пропускает все элементы, которые больше pivot, l — все, которые меньше pivot. Когда оба находят элементы, стоящие не так, меняем местами a_l и a_r
4. Когда l, r встретятся — мы получим необходимое разбиение

Вариант реализации разбиения Хоара на Java

```java
public class HoarePartition {
  public static int hoarePartition(int[] a, int l, int r) {
    int piv_idx = l + (r - l) / 2;
    int pivot = a[piv_idx];
    int i = l;
    int j = r;
    while (i <= j) {
      while (a[i] < pivot) {i++;}
      while (a[j] > pivot) {j--;}
      if (i >= j) {
        return j;
      }
      swap(a, i, j); // swapping elements in a by index
      i++;
      j--;
    }
    return j;
  }
}
```

Замечание 3.1.

- *На практике разбиение Хоара обычно работает заметно быстрее, чем разбиение Ломуто*
- *Интуитивно это можно объяснить тем, что при разбиении Хоара сначала меняются местами как можно более удаленные друг от друга элементы, благодаря чему в массиве быстрее уменьшается число инверсий*

4 Толстое разбиение

Представим что у нас есть массив из 1, 2 и 3, и нам необходимо его отсортировать — эта задача носит название задачи флага Нидерландов. Поскольку числа обладают свойством трихотомии, решение задачи флага Нидерландов можно применить к разбиению элементов на три группы: $<$ pivot, $=$ pivot, $>$ pivot.

1. Заведем три указателя: l, mid, r
2. $l = 0, r = n - 1, mid = 0$
3. Будем поддерживать инвариант: a$[r...n-1] >$ pivot, a$[0...l] <$ pivot, a$[l...mid] ==$ pivot.

Вариант реализации толстого разбиения на Java

```java
public class FlagPartition {
  public static void flagPartition(int[] a, int l, int r) {
    int pivot = a[l + (r - l) / 2];
    int mid = l;
    while (mid <= r) {
      if (a[mid] < pivot) {
        swap(a, l, mid); // swapping elements in a by index
        l++;
        mid++;
      } else if (a[mid] == pivot) {
        mid++;
      } else {
        swap(a, mid, r);
        r--;
      }
    }
  }
}
```

5 Стратегии выбора элементов

Рассмотрим стратегию, при котором выбирается центральный элемент

- Заметим, что если при каждом partition в качестве опорного брать самый маленький элемент, то у нас подзадачи будут иметь размеры 0 и $n - 1$.
- Это ужасно плохо, так как при такой работе алгоритм скатится до $\mathcal{O}(n^2)$.
- Таким образом, от того, как мы выбираем опорный элемент зависит время работы нашего алгоритма.
- Неплохой стратегией является выбор случайного элемента.
- Также можно брать медиану трех случайных элементов (или даже 5-ти или 7-ми).
- Стратегия, при которой выбирается случайный элемент сводит на нет возможность подобрать такой массив, на котором наш алгоритм будет работать долго.
- Худшее, что может произойти: нам не повезет n раз подряд, вероятность чего мизерная.
- Проведем анализ работы быстрой сортировки, когда в качестве опорного выбирается случайный элемент.
- Анализ будем проводить из предположения, что все элементы различны.

Лемма 5.1. *Пусть X — число сравнений, выполняемых за время работы сортировки над n-элементным массивом. Тогда время работы быстрой сортировки составляет $\mathcal{O}(n + X)$.*

Доказательство. Быстрая сортировка делает не более n вызовов функции partition, который в свою очередь совершает некоторое количество итераций цикла, в каждой из которых происходит сравнение элементов с pivot. □

Лемма 5.2. $\mathbb{E}[X] = \mathcal{O}(n \log n)$, *где X — число сравнений, выполняемых за время работы сортировки над n-элементным массивом.*

Доказательство. Интуитивное объяснение, НЕ ЯВЛЯЕТСЯ МАТЕМАТИЧЕСКИ ТОЧНЫМ ДОКАЗАТЕЛЬСТВОМ: сделаем вид что массив всегда разбивается «примерно» пополам. Тогда рекуррентное соотношение совпадает с рекуррентным соотношением алгоритма MergeSort:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Из мастер теоремы о рекурсии получаем, что $T(n) = \mathcal{O}(n \log n)$. □