Лекция 3. Бинарный поиск. Порядковые статистики.

1. Рандомизированные алгоритмы и быстрая сортировка Хоара

До сих пор мы анализировали **детерминированные** алгоритмы: для одного и того же входного данных алгоритм всегда выполняет одну и ту же последовательность операций. **Рандомизированные** алгоритмы используют источник случайности (например, генератор случайных чисел), поэтому их поведение и время работы могут меняться от запуска к запуску на одних и тех же данных.

Время работы такого алгоритма становится **случайной величиной**. Для его анализа мы будем использовать **математическое ожидание** времени работы.

Математическое ожидание (обозначается $E[X]$) – это среднее значение случайной величины X при многократном повторении эксперимента. Грубо говоря, это то, что мы ожидаем в среднем. Для дискретной величины оно вычисляется как сумма произведений возможных значений на их вероятности:

$$E[X] = \sum_i x_i \cdot P(X = x_i).$$

Нас будет интересовать **математическое ожидание времени работы в худшем случае** (т.е. для самого неблагоприятного входа).

Ярчайший пример рандомизированного алгоритма – **быстрая сортировка (Quicksort)** Тони Хоара.

Идея алгоритма (выбор случайного опорного элемента):

1. Выбрать в массиве **опорный элемент (pivot)** x случайным образом.
2. **Разделить (partition)** массив на три части:

- Элементы **меньше** x.
- Элементы **равные** x (необязательно выделять отдельно, но это улучшает работу с повторяющимися элементами).
- Элементы **больше или равные** x.

3. **Рекурсивно** отсортировать левую и правую части.

Реализация:

```python
import random
from typing import List, Tuple


def quicksort(arr: List[int], left: int = 0, right: int = None) -> None:
    """Сортирует подмассив arr[left:right] на месте с помощью быстрой
сортировки."""
    if right is None:
        right = len(arr) - 1
    # Базовый случай: массив из 0 или 1 элемента уже отсортирован
    if left >= right:
        return
```

```python
    # Рекурсивный случай
    # Выбираем случайный опорный элемент и ставим его в конец (для удобства
разделения)
    pivot_idx = random.randint(left, right)
    arr[right], arr[pivot_idx] = arr[pivot_idx], arr[right]
    x = arr[right]

    # Разделение: получаем индекс 'm' - начало правой части (>= x)
    m = partition(arr, left, right, x)

    # Рекурсивно сортируем части [left, m-1] и [m, right]
    quicksort(arr, left, m - 1)
    quicksort(arr, m, right)


def partition(arr: List[int], left: int, right: int, pivot: int) -> int:
    """
    Разделяет подмассив arr[left:right] на элементы < pivot и >= pivot.
    Возвращает индекс начала второй части (>= pivot).
    """
    # Индекс, по которому мы вставляем следующий элемент, меньший опорного
    m = left
    for i in range(left, right):
        if arr[i] < pivot:
            arr[i], arr[m] = arr[m], arr[i]
            m += 1
    # Ставим опорный элемент на границу
    arr[m], arr[right] = arr[right], arr[m]
    return m
```

Анализ времени работы:

Время работы $T(n)$ – случайная величина. Обозначим её математическое ожидание как $T^*(n) = E[T(n)]$.

Рекуррентное соотношение для $T^*(n)$:

$$T^*(n) = n + T^*(L) + T^*(R),$$

где:

- n – время на разделение массива (цикл в partition).
- L – размер левой части ($< x$).
- R – размер правой части ($\geq x$), причём $L + R = n - 1$ (так как один элемент – это сам x).

Размеры L и R зависят от выбранного x. Поскольку x выбирается случайно равновероятно из n элементов, можно считать, что значение x делит массив на части в некоторой случайной пропорции.

Ключевое наблюдение. Для хорошей асимптотики нужно, чтобы опорный элемент делил массив примерно пополам. Худший случай ($L = 0$, $R = n - 1$ или наоборот) приводит к $T^*(n) = n + T^*(n-1) = O(n^2)$. Но в среднем (по всем выборам x) получается лучше.

Можно показать, что математическое ожидание времени работы удовлетворяет соотношению:

$$T^*(n) \le n + \frac{1}{n} \cdot \sum_{k=0}^{n-1} [T^*(k) + T^*(n-1-k)].$$

Домножим обе части на n:

$$T^*(n) \le n^2 + 2 \cdot \left(T^*(0) + T^*(1) + \ldots + T^*(n-1)\right).$$

Методом подстановки можно доказать, что решением этого неравенства является $T^*(n) = O(n \log n)$. Более того, можно получить точную константу.

Упрощённый анализ через «хорошее» разделение:

Предположим, что с вероятностью $\frac{1}{2}$ нам «везёт» и опорный элемент попадает в среднюю половину (т.е. размер обеих частей не превышает, скажем, $\frac{3n}{4}$). С вероятностью $\frac{1}{2}$ нам «не везёт».

Тогда рекуррентное соотношение можно записать как:

$$T^*(n) \le n + \frac{1}{2} \cdot T^*\left(\frac{3n}{4}\right) + \frac{1}{2} \cdot T^*(n)$$

Это грубая, но рабочая оценка.

Перенесём $\frac{1}{2} \cdot T^*(n)$ в левую часть:

$$T^*(n) - \frac{1}{2} \cdot T^*(n) \le n + \frac{1}{2} \cdot T^*\left(\frac{3n}{4}\right)$$

$$\frac{1}{2} \cdot T^*(n) \le n + \frac{1}{2} \cdot T^*\left(\frac{3n}{4}\right)$$

Умножим обе части на 2:

$$T^*(n) \le 2n + T^*\left(\frac{3n}{4}\right)$$

Это стандартное рекуррентное соотношение. Распишем его:

$$T^*(n) \le 2n + T^*\left(\frac{3n}{4}\right) \le 2n + 2 \cdot \left(\frac{3n}{4}\right) + T^*\left(\left(\frac{3}{4}\right)^2 n\right) \le$$

$$\le 2n + \frac{3n}{2} + T^*\left(\frac{9}{16} n\right) \le \ldots$$

Получим убывающую геометрическую прогрессию:

$$T^*(n) \le 2n \cdot \left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \ldots\right)$$

Сумма бесконечно убывающей геометрической прогрессии

$$S = \frac{1}{1-q} = \frac{1}{1 - \frac{3}{4}} = 4.$$

Следовательно,

$$T^*(n) \le 2n \cdot 4 = 8n = O(n).$$

Но это оценка только на «уровень» рекурсии, связанный с разделением. На каждом уровне размер уменьшается в $\frac{4}{3}$ раза. Количество уровней рекурсии L удовлетворяет условию $\left(\frac{3}{4}\right)^L \cdot n = 1$, откуда $L = \log_{\frac{4}{3}} n$. На каждом уровне i суммарное время разделения всех подмассивов этого уровня составляет $O(n)$.

Итоговое время:
$T^*(n) = O(n) \cdot \text{количество уровней} = O(n) \cdot \log_{\frac{4}{3}} n = O(n \log n)$.

Вывод. Математическое ожидание времени работы рандомизированной быстрой сортировки составляет $O(n \log n)$.

2. Порядковые статистики. Алгоритм Хоара и алгоритм BFPRT

Задача. Найти k-ю **порядковую статистику** (элемент, который стоял бы на k-й позиции (начиная с 0) в отсортированном массиве).

- $k = 0$ – минимум.
- $k = n - 1$ – максимум.
- $k = (n - 1)//\ 2$ – медиана.

Наивное решение – отсортировать массив за $O(n \log n)$ и взять элемент по индексу k. Но можно сделать быстрее – за $O(n)$ в среднем и даже в худшем случае!

2.1. Рандомизированный алгоритм Хоара (Quickselect)

Алгоритм очень похож на быструю сортировку, но рекурсивно обрабатывает только ту часть массива, где находится искомая порядковая статистика.

Идея:

1. Выбрать случайный опорный элемент x.

2. Разделить массив с помощью $partition$ на элементы $< x$ и $\geq x$. Пусть m – индекс, с которого начинается часть $\geq x$. Это также означает, что есть ровно m элементов, строго меньших x.

3. Сравнить k с m:

- Если $k < m$, то искомый элемент находится в левой части ($< x$). Рекурсивно ищем k-й элемент в $arr[left{:}\,m - 1]$.
- Если $k \geq m$, то искомый элемент находится в правой части ($\geq x$). Однако в правой части мы ищем элемент не с индексом k, а с индексом $k - m$, потому что мы уже «прошли» m элементов, которые меньше x.

Реализация:

```
def quickselect(arr: List[int], k: int, left: int = 0, right: int = None) ->
int:
    """Возвращает k-ю порядковую статистику в подмассиве arr[left:right]."""
    if right is None:
        right = len(arr) - 1
    # Если подмассив содержит один элемент, возвращаем его
    if left == right:
        return arr[left]

    # Выбираем случайный опорный элемент
    pivot_idx = random.randint(left, right)
    # Ставим его в конец для функции partition
    arr[right], arr[pivot_idx] = arr[pivot_idx], arr[right]
```

```python
    x = arr[right]

    # Разделяем массив
    m = partition(arr, left, right, x) # m - начало части >= x

    if k < m:
        # Искомый элемент в левой части (< x)
        return quickselect(arr, k, left, m - 1)

    elif k >= m:
        # Искомый элемент в правой части (>= x)
        # Теперь его индекс в правой части: k - m
        return quickselect(arr, k, m, right)
    # Элемент с индексом 'm' - это сам x, и если k == m, мы его уже нашли.
    # Условие k >= m покрывает этот случай, и рекурсия продолжается до
базового случая.
```

Анализ времени работы:

Анализ аналогичен быстрой сортировке. В среднем на каждом шаге размер задачи уменьшается примерно вдвое.

$T^*(n) = n + T^*\left(\frac{n}{2}\right)$ в среднем (если x делит массив пополам).

$$T^*(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots \leq 2n = O(n).$$

Более строгий анализ, учитывающий все возможные разбиения, также даёт $T^*(n) = O(n)$.

Однако в **худшем случае** (если каждый раз опорный элемент оказывается наибольшим или наименьшим), время работы будет $O(n^2)$. Но вероятность такого неблагоприятного стечения обстоятельств крайне мала.

2.2. Детерминированный алгоритм BFPRT (медиана медиан)

Для гарантированной линейной сложности даже в худшем случае используется более сложный алгоритм, известный как алгоритм BFPRT (по фамилиям Blum, Floyd, Pratt, Rivest, Tarjan).

Идея алгоритма. На каждом шаге выбирать опорный элемент x не случайно, а детерминированно, так чтобы он гарантированно делил массив на части не хуже, чем 3:7.

Шаги алгоритма для нахождения k-й порядковой статистики:

1. Разбить массив на группы по 5 элементов (последняя группа может быть меньше).

2. Найти медиану в каждой группе. Это можно сделать сортировкой вставками для 5 элементов за константное время.

3. Рекурсивно найти медиану x из найденных медиан (медиану медиан). Это и будет наш опорный элемент.

4. Разделить массив вокруг x с помощью $partition$. Пусть m – индекс границы.

5. Как и в алгоритме Хоара, рекурсивно продолжить поиск в левой или правой части в зависимости от k и m.

Почему это работает за O(n)?

Ключевой момент – оценка количества элементов, которые гарантированно будут меньше или больше x.

- Хотя бы половина от $\frac{n}{5}$ медиан групп будет больше или равна медиане медиан x.
- В каждой такой группе как минимум 3 элемента (половина от 5) будут больше или равны своей медиане, а значит, и больше или равны x.
- Следовательно, элементов, **больших или равных** x, как минимум: $\left(\frac{n}{10}\right) \cdot 3 = \frac{3n}{10}$.
- Аналогично, элементов, **меньших или равных** x, также как минимум $\frac{3n}{10}$.

Таким образом, после разделения вокруг x размер большей из двух частей не превысит $n - \frac{3n}{10} = \frac{7n}{10}$.

Рекуррентное соотношение для времени работы $T(n)$:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

- $T\left(\frac{n}{5}\right)$ – время на нахождение медианы медиан.
- $T\left(\frac{7n}{10}\right)$ – время на рекурсивный вызов для большей части.
- $O(n)$ – время на разбиение на группы, нахождение медиан в группах и итоговое разделение вокруг x.

Докажем по индукции, что $\boldsymbol{T(n) = O(n)}$.

Предположим, что $T(n) \leq c \cdot n$ для всех $n < N$. Покажем, что это верно для $n = N$.

$$T(n) \leq c \cdot \frac{N}{5} + c \cdot \frac{7N}{10} + a \cdot N = c \cdot N \cdot \left(\frac{1}{5} + \frac{7}{10}\right) + a \cdot N = c \cdot N \cdot \frac{9}{10} + a \cdot N$$

Мы хотим, чтобы $c \cdot N \cdot \frac{9}{10} + a \cdot N \leq c \cdot N$.

Вычтем $c \cdot N \cdot \frac{9}{10}$ из обеих частей: $a \cdot N \leq c \cdot N \cdot \frac{1}{10}$.

Следовательно, $c \geq 10 \cdot a$.

Если выбрать константу $c = 10 \cdot a$, то неравенство выполняется. База индукции проверяется для небольших n выбором достаточно большой константы c.

Таким образом, $T(n) = O(n)$.

Вывод. Алгоритм BFPRT находит k-ю порядковую статистику за линейное время $O(n)$ в худшем случае. Однако скрытая константа c велика, поэтому на практике для не очень больших n рандомизированный алгоритм Хоара часто оказывается быстрее.

3. Бинарный поиск

Бинарный поиск – классический алгоритм поиска элемента в **отсортированном массиве**. Его сложность $O(\log n)$, что намного эффективнее линейного поиска $O(n)$.

3.1. Базовый бинарный поиск

Задача. Проверить, присутствует ли элемент x в отсортированном массиве arr, и если да, то вернуть его индекс.

Идея. На каждом шаге алгоритм сравнивает x с элементом в середине текущего диапазона поиска $[left, right]$.

- Если x равен среднему элементу, поиск завершён.
- Если x меньше среднего элемента, поиск продолжается в левой половине.
- Если x больше среднего элемента, поиск продолжается в правой половине.

Процесс повторяется, пока диапазон не станет пустым.

Реализация (итеративная):

```python
def binary_search(arr: List[int], x: int) -> int:
    """Возвращает индекс элемента x в отсортированном массиве arr.
    Если элемент не найден, возвращает -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2  # Целочисленное деление
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1  # Сужаем диапазон до правой половины
        else:  # arr[mid] > x
            right = mid - 1  # Сужаем диапазон до левой половины
    return -1
```

Важно. Использование mid = (left + right) // 2 может привести к переполнению в некоторых языках для очень больших массивов. Безопасная форма: mid = left + (right - left) // 2.

3.2. Поиск левой и правой границы

Часто требуется найти не любое вхождение элемента, а первое (левую границу) или последнее (правую границу). Это особенно полезно для работы с дубликатами.

Задача. Найти индекс **первого** элемента, **большего или равного** x (lower_bound). Если все элементы меньше x, вернуть len(arr).

Идея. Мы будем поддерживать **инвариант**: arr[left] < x и arr[right] >= x. В начале положим left = -1, right = len(arr). Цикл будет сужать этот диапазон $(left, right]$ до тех пор, пока между $left$ и $right$ не останется ровно один элемент – искомая граница.

Реализация:

```python
def lower_bound(arr: List[int], x: int) -> int:
    """Возвращает индекс первого элемента >= x."""
    left, right = -1, len(arr)
    # Инвариант: arr[left] < x, arr[right] >= x
    while right > left + 1:
        mid = (left + right) // 2
        if arr[mid] < x:
            left = mid
        else:
            right = mid
    return right  # right всегда будет указывать на первый элемент >= x
```

Аналогично ищется индекс **первого элемента, строго большего** x (upper_bound):

```python
def upper_bound(arr: List[int], x: int) -> int:
    """Возвращает индекс первого элемента > x."""
    left, right = -1, len(arr)
    # Инвариант: arr[left] <= x, arr[right] > x
    while right > left + 1:
        mid = (left + right) // 2
        if arr[mid] <= x:
            left = mid
        else:
            right = mid
    return right
```

С помощью lower_bound и upper_bound легко найти диапазон индексов, соответствующих всем вхождениям x:

$$\left[lower\_bound(arr, x), upper\_bound(arr, x)\right).$$

3.3. Бинарный поиск по ответу

Это мощная техника, которая применяется для решения задач оптимизации вида «найдите минимальное значение параметра x, при котором выполняется условие $P(x)$».

Идея:

1. Определить диапазон $[l, r]$, в котором точно лежит ответ.

2. Проверить для среднего значения $m = (l + r)//2$, выполняется ли условие $P(m)$.

• Если условие выполняется ($P(m) == True$), то ответ находится в диапазоне $[l, m]$.

• Если условие не выполняется ($P(m) == False$), то ответ находится в диапазоне $[m + 1, r]$.

3. Повторять шаг 2, пока $l < r$.

Пример задачи. Есть n прямоугольников размером $a \times b$. Можно ли разместить их все в квадрате со стороной s (не поворачивая)? Найти минимальную сторону такого квадрата s_min.

Решение:

1. Функция проверки is_valid(s):

- В квадрат $s \times s$ можно уложить $(s//a) \cdot (s//b)$ прямоугольников.
- Проверить, что $(s//a) \cdot (s//b) \geq n$.

2. Определение границ:

- $l = 0$ (очевидно, слишком мало).
- $r = max(a, b) \cdot n$ (грубая, но рабочая верхняя оценка).

3. Бинарный поиск:

```
def find_min_square_side(a: int, b: int, n: int) -> int:
    """Находит минимальную сторону квадрата для размещения n прямоугольников
a*b."""

    def is_valid(s: int) -> bool:
        # Количество прямоугольников, которое поместится по горизонтали и
вертикали
        # Учитываем целочисленное деление
        return (s // a) * (s // b) >= n

    l, r = 0, max(a, b) * n
    while l < r:
        mid = (l + r) // 2
        if is_valid(mid):
            r = mid  # Ответ в [l, mid]
        else:
            l = mid + 1  # Ответ в [mid+1, r]
    return l
```

Внимание! При работе с большими числами ($a, b, n \sim 10^9$) вычисление $(s//a) \cdot (s//b)$ может привести к переполнению типа int (если использовать $(s//a) \cdot (s//b)$, промежуточные результаты будут огромными). Правильный порядок операций и целочисленное деление спасают от переполнения в этом конкретном случае. Однако если бы мы вычисляли что-то вроде $(s \cdot s)//(a \cdot b)$, переполнение было бы возможно. Нужно быть аккуратным и использовать проверки или другие математические преобразования.

3.4. Бинарный поиск по вещественному аргументу

Применяется, когда ответ – вещественное число, и мы ищем его с заданной точностью $epsilon$.

Идея. Аналогична целочисленному случаю, но цикл продолжается, пока $(r - l) > epsilon$.

Пример. Найти корень уравнения $f(x) = 0$ на отрезке $[l, r]$, где известно, что функция монотонна.

```
def binary_search_float(l: float, r: float, epsilon: float, f:
Callable[[float], float]) -> float:
    """Находит корень монотонной функции f на отрезке [l, r] с точностью
```

```
epsilon."""
    while r - l > epsilon:
        mid = (l + r) / 2.0
        if f(mid) >= 0:  # Предполагаем, что функция возрастает и f(l) < 0, 
f(r) > 0
            r = mid
        else:
            l = mid
    return (l + r) / 2.0
```

Важно. Для избежания бесконечных циклов из-за погрешностей вычислений с плавающей точкой иногда используют не условие while r - l > epsilon, а фиксированное количество итераций (например, 100). Это гарантирует, что точность будет как минимум $\frac{r-l}{2^{100}}$.

3.5. Тернарный поиск

Применяется для поиска **экстремума** (минимума или максимума) **унимодальной функции** на отрезке. Унимодальная функция – это функция, которая на отрезке сначала строго возрастает, а затем строго убывает (или наоборот), имея ровно одну точку экстремума.

Идея. Разделить отрезок $[l, r]$ на **три** части двумя точками m_1 и m_2 ($m_1 = l + (r - l)/3, m_2 = r - (r - l)/3$).

- Если $f(m_1) < f(m_2)$ для поиска минимума, то минимум не может находиться правее m_2 (правая треть отбрасывается).
- Если $f(m_1) \geq f(m_2)$, то минимум не может находиться левее m_1 (левая треть отбрасывается).

```
def ternary_search(l: float, r: float, epsilon: float, f: Callable[[float], 
float]) -> float:
    """Находит минимум унимодальной функции f на отрезке [l, r]."""
    while r - l > epsilon:
        m1 = l + (r - l) / 3
        m2 = r - (r - l) / 3
        if f(m1) < f(m2):
            r = m2
        else:
            l = m1
    return (l + r) / 2
```

Сложность тернарного поиска $O\left(\log_{\frac{3}{2}} \frac{r-l}{epsilon}\right)$, что асимптотически равно $O(\log n)$, как и у бинарного поиска, но с худшей константой. На практике бинарный поиск по производной (если её можно вычислить) часто эффективнее.

Вывод. Бинарный поиск – это не просто алгоритм для поиска элемента в массиве, а общий мощный принцип «разделяй и властвуй» для решения самых разных задач за логарифмическое время.