ITIVITI

LET IT TRADE

Алгоритмы и структуры данных

Косяков Михаил Сергеевич
к.т.н., доцент кафедры ВТ

УНИВЕРСИТЕТ ИТМО

Содержание курса

- Введение в теорию алгоритмов
- **Алгоритмы сортировок**
- Структуры данных
  - Линейные структуры
  - Бинарные деревья поиска
  - Хеши и хеш-функции
- Алгоритмы на графах
  - Обходы графов в ширину и глубину
  - Минимальные остовные деревья
  - Поиск кратчайших путей в графе

Быстрая сортировка

ITIVITI

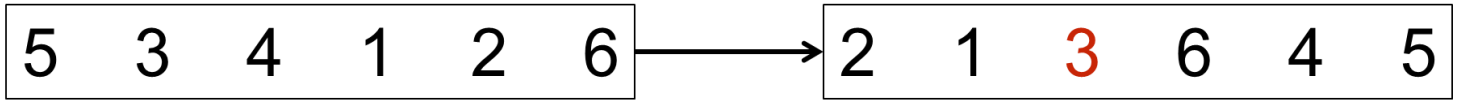
- Один из самых популярных алгоритмов сортировки
- Метод «разделяй и властвуй» (conquer-and-divide):
  - 1) разделим задачу на подзадачи
  - 2) решаем подзадачи с использованием рекурсии
  - 3) объединяем решения подзадач в решение задачи
- При сортировке слиянием основная работа после выполнения рекурсивных вызовов
- При быстрой сортировке основная работа до выполнения рекурсивных вызовов
- Время работы алгоритма зависит от входных данных
- Шаг номер один - разбиение

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Разбиение

- Выберем некоторый элемент в массиве (точка разбиения): опорный (*pivot*) элемент; разделитель *p*
- Изменим порядок элементов в массиве так, что:
  - 1) слева от опорного элемента стоят элементы $A[i] \le p$
  - 2) справа от опорного элемента стоят элементы $A[i] \ge p$
  - ≤ и ≥ для более сбалансированного разбиения при наличии множества дублированных ключей
- С точки зрения сортировки: опорный элемент *p* займет свою окончательную позицию



Разбиение Ломуто

```
1. int partition(int * a, int l, int r) {
2.     int pivot_index = l; // choose_pivot(a, l, r);
3.     if (pivot_index != l) std::swap(a[l], a[pivot_index]);
4.     int p = a[l];
5.     int i = l + 1;
6.     for (int j = l + 1; j <= r; ++j) {
7.         if (a[j] < p) {
8.             std::swap(a[j], a[i]);
9.             ++i;
10.         }
11.     }
12.     std::swap(a[l], a[i-1]);
13.     return i - 1;
14. }
```

- Оценка сложности: $\Theta(N)$; доп. память: $O(1)$
- Где окажутся элементы, равные p?
- Как разобьется массив из одинаковых элементов?

Разбиение Хоара

```
1. int partition(int * a, int l, int r) {
2.     int pivot_index = l; // choose_pivot(a, l, r);
3.     if (pivot_index != l) std::swap(a[l], a[pivot_index]);
4.     int p = a[l];
5.     int i = l; int j = r+1;
6.     for (; ;) {
7.         while (a[++i] < p) if (i == r) break;
8.         while (a[--j] > p);
9.         if (i >= j) break;
10.        std::swap(a[i], a[j]);
11.    }
12.    std::swap(a[l], a[j]);
13.    return j;
14. }
```

- Оценка сложности: $\Theta(N)$; доп. память: $O(1)$
- Где окажутся элементы, равные p?
- Как разобьется массив из одинаковых элементов?

Быстрая сортировка

```
1. void quicksort(int * a, int l, int r)
2. {
3.     if (r <= l) return;
4.     int p = partition(a, l, r);
5.     quicksort(a, l, p-1);
6.     quicksort(a, p+1, r);
7. }
```

- Как алгоритм будет вести себя на отсортированном массиве?
- Оценка сложности:
  - Наихудший случай разбиения (выбираем min / max):
    $T(N) = T(N-1) + \Theta(N)$
  - Наилучший случай разбиения (выбираем медиану):
    $T(N) = 2T(N/2) + \Theta(N)$
- Дополнительная память: ???

Быстрая сортировка (медиана из трех элементов)

- Давайте получим оценку медианы массива:

```
1. inline void comp_swap(int &a, int &b) {
2.     if (a > b) std::swap(a, b); }
3.
4. void quicksort(int * a, int l, int r) {
5.     if (r <= l) return;
6.     std::swap( a[(l+r)/2], a[l+1] );
7.     comp_swap(a[l], a[l+1]);
8.         comp_swap(a[l], a[r]);
9.             comp_swap(a[l+1], a[r]);
10.     if (r - l <= 2) return;
11.     int p = partition(a, l+1, r-1);
12.     quicksort(a, l, p-1);
13.     quicksort(a, p+1, r);
14. }
```

Рандомизированная быстрая сортировка

ITIVITI

- Мы не знаем потребуется ли на практике сортировка почти упорядоченных массивов
- Добавим в алгоритм рандомизацию

```
1. inline int choose_pivot(int * a, int l, int r) {
2.     return l + (rand() % (r-l+1));
3. }
```

- Любой из `r - l + 1` элементов может стать опорным с одинаковой вероятностью
- Время работы – случайная величина: на одном и том же массиве будет разное время работы
- Имеет смысл говорить о мат. ожидании (среднем) времени работы (при случайном выборе опорных элементов)

LET IT TRADE

Вспоминаем понятия теории вероятностей

ITIVITI

- Элементарное событие e – возможный исход некоторого эксперимента
  - Кидаем кубик: эксперимент – бросание кубика, исход – выпадение одной из граней
- Пространство выборок S – множество элементарных событий
  - Кидаем два кубика: S = {(1,1), (1,2), (1,3), … (6,5), (6,6)}
- Событие A – подмножество пространства выборок S
  - Кидаем два кубика: событие – выпало одинаковое число очков
- Вероятность Pr(A) события A – отношение числа благоприятных исходов к общему числу равновозможных исходов
  - Кидаем два кубика: какова вероятность того, что число очков на обоих кубиках будут одинаковы?

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Вспоминаем понятия теории вероятностей

ITIVITI

- $X(e)$ – функция из S в множество действительных чисел R
  - Кидаем два кубика: сумма числа очков на обоих кубиках
  - Какая из сумм будет наиболее вероятной?
  - Какая сумма имеет большую вероятность: 3 или 10?
- Математическое ожидание $E[X]$ – среднее арифметическое значений случайной величины для всех исходов
  - Кидаем кубик: мат. ожидание числа очков на выпавшей грани?
  - (1+2+3+4+5+6)/6 = 3.5
  - Если платить три с полтиной за участие в игре, где можно бросить кубик и получить столько рублей, сколько на нём выпадет очков, то при очень большом числе игр в среднем играющий будет оставаться «при своих»

LET IT TRADE

Вспоминаем понятия теории вероятностей

ITIVITI

- Математическое ожидание $E[X] = \sum_x x \cdot \Pr(X = x)$
  - Кидаем два кубика: мат. ожидание суммы числа очков?
  - 2·1/36 + 3·2/36 + 4·3/36 + … + 10·3/36 + 11·2/36 + 12 ·1/36 = 7
- Линейность мат. ожидания $E[X+Y] = E[X] + E[Y]$
  - Кидаем два кубика: мат. ожидание суммы числа очков?
  - Случайная величина X – "число, выпавшее на первом кубике", Y – "число, выпавшее на втором кубике"
  - $E[X+Y] = E[X] + E[Y] = 3.5 + 3.5 = 7$
  - Кидаем два кубика: мат. ожидание числа выпавших единиц?
  - X – "число единиц на первом кубике ", Y – "число единиц на втором кубике" (индикаторные случайные величины)
  - $E[X] + E[Y] = \Pr(X=1) + \Pr(Y=1) = 1/6 + 1/6 = 1/3$

LET IT TRADE

Рандомизированная быстрая сортировка

ITIVITI

- Сложность работы алгоритма определяется общим числом сравнений, выполняемых при всех вызовах `partition`
  - `partition` вызывается не более N раз (опорный элемент не принимает участия в рекурсивных вызовах)
  - Внутри `partition` опорный элемент сравнивается с другими элементами массива
- Пространство выборок S – все возможные исходы случайного выбора последовательности опорных элементов
- Случайная величина $X(e)$ – число сравнений двух элементов массива друг с другом при выполнении быстрой сортировки для фиксированного выбора опорных элементов e
- $E[X] = O(N \log N)$?

LET IT TRADE

Рандомизированная быстрая сортировка

ITIVITI

- Как посчитать X? X изменяется от $N \log N$ и N^2
  - Пусть z_i – i-ый наименьший элемент массива (i-ая порядковая статистика)
  - Для фиксированного выбора опорных элементов e и индексов $i < j$ определим случайную величину $X_{ij}(e)$ как число сравнений двух элементов z_i и z_j в ходе работы алгоритма
- Сколько раз могут сравниться два фиксированных элемента в ходе выполнения алгоритма?
  - Опорный элемент сравнивается с другими ровно один раз
  - X_{ij} – индикаторная случайная величина (z_i и z_j сравнились или нет)
  - $X = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} X_{ij}$ и $E[X] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} E[X_{ij}]$
- Т.о. $E[X] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \Pr[X_{ij} = 1]$

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Рандомизированная быстрая сортировка

ITIVITI

- Как посчитать вероятность того, что z_i сравнивается с z_j ?
  - Рассмотрим множество $Z_{ij} = \{z_i, z_{i+1}, \dots, z_{j-1}, z_j\}$, которое содержит элементы, расположенные между z_i и z_j включительно
  - Пока опорный элемент выбирается вне Z_{ij}, все элементы Z_{ij} передаются в один и тот же рекурсивный вызов
  - Рано или поздно опорный элемент будет выбран из Z_{ij}
  - Если в качестве опорного элемента выбирается z_i или z_j, то мы сравним эти два элемента
  - Если опорным элементом становится элемент из $z_{i+1}, \dots, z_{j-1}$ вперед z_i или z_j, то он поместит z_i и z_j в разные подмассивы, эти элементы уйдут в разные рекурсивные вызовы и больше никогда не встретятся
- Поэтому Pr(z_i сравнивается с z_j) = $2 / (j - i + 1)$

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Рандомизированная быстрая сортировка

ITIVITI

LET IT TRADE

- Т.о. мат. ожидание общего числа операций сравнения:

$$E[X] = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{2}{j-i+1} =$$

$$= \sum_{i=1}^{N-1} 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N-i+1}\right) <$$

$$< \sum_{i=1}^{N-1} 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N-i+1} + \cdots + \frac{1}{N}\right) <$$

$$< 2N \cdot \sum_{k=2}^{N} \frac{1}{k}$$

- Как оценить верхнюю границу гармонического ряда $1/2 + 1/3 + \ldots + 1/N$?
  - $1/x$ - монотонно убывающая функция, используем приближение интегралами

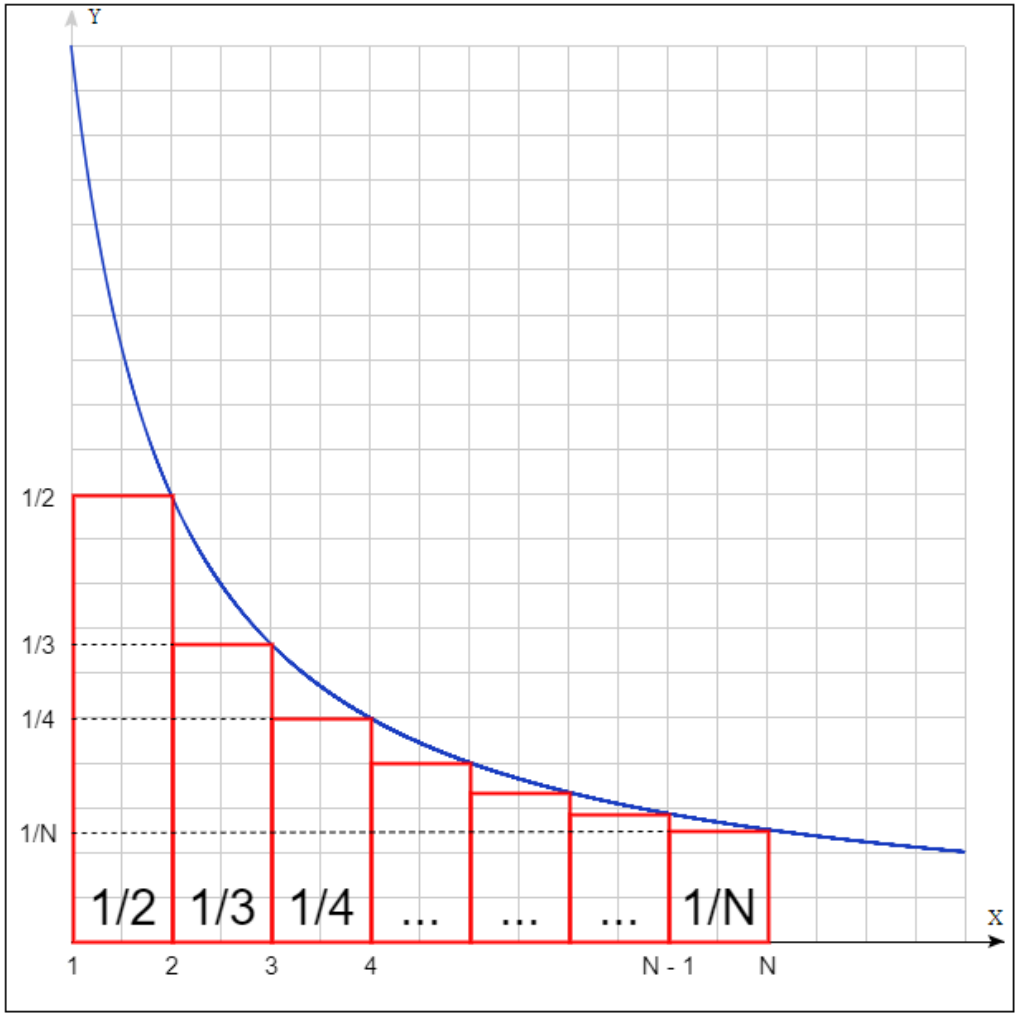
УНИВЕРСИТЕТ ИТМО

Рандомизированная быстрая сортировка

ITIVITI

LET IT TRADE

- Приближение интегралами:
  - Синяя линия: монотонно убывающая функция $y(x) = 1/x$
  - Площадь красных прямоугольников: $1/2 + 1/3 + \ldots + 1/N$
  - Сумма ряда ограничена сверху интегралом от 1 до N
  - $\sum_{k=2}^{N} \frac{1}{k} \le \int_1^N \frac{1}{x} dx = \ln N$
  - Т.о. $E[X] < 2N \cdot \ln(N)$



Быстрая сортировка (маленькие подмассивы)

ITIVITI

- На маленьких подмассивах рекурсивный характер быстрой сортировки неэффективен
- Какой алгоритм ведет себя хорошо на (почти) отсортированном массиве?

```
1. void quicksort(int * a, int l, int r)
2. {
3.     // if (r - l <= k) insertion_sort(a, l, r);
4.     if (r - l <= k) return;
5.     int p = partition(a, l, r);
6.     quicksort(a, l, p-1);
7.     quicksort(a, p+1, r);
8. } ... insertion_sort(a, l, r);
```

- Оценка сложности: $k \approx N / 2^x \Rightarrow O(N k + N \log N/k)$
  - $O(N \log N)$ при $k = 1$; $O(N^2)$ при $k = N$

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

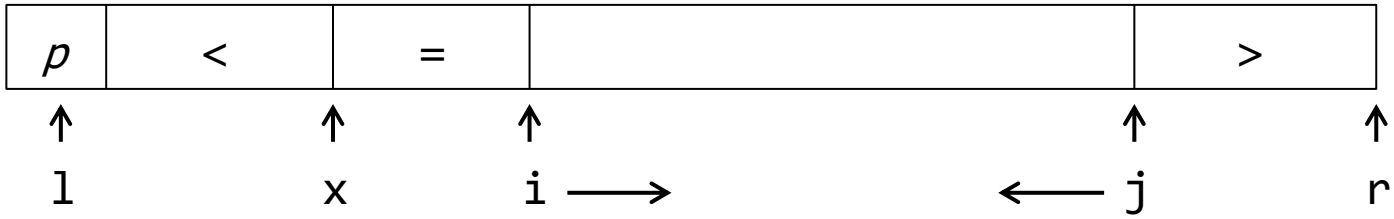
Быстрая сортировка (дублированные ключи)

ITIVITI

- Часто в массиве большое число одинаковых ключей
- Даже если в подмассиве остались только одинаковые ключи быстрая сортировка все равно уходит в рекурсию
- Разделим массив на **три** части так, что:
  - 1) слева элементы $A[i] < p$
  - 2) справа элементы $A[i] > p$
  - 3) посередине элементы $A[i] = p$
- Рекурсивные вызовы только на левой и правой частях!
- Если число различных элементов константно, сортировка отработает за линейное время (по году рождения)
  - На каждом разбиении исключаем все ключи, равные опорному

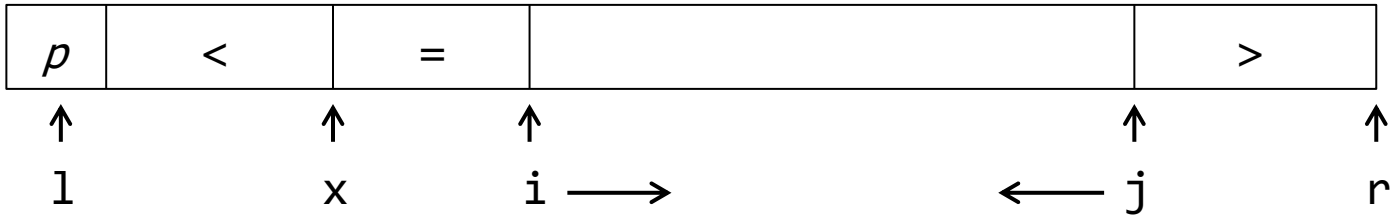
LET IT TRADE

Разбиение на три части (Дейкстра)



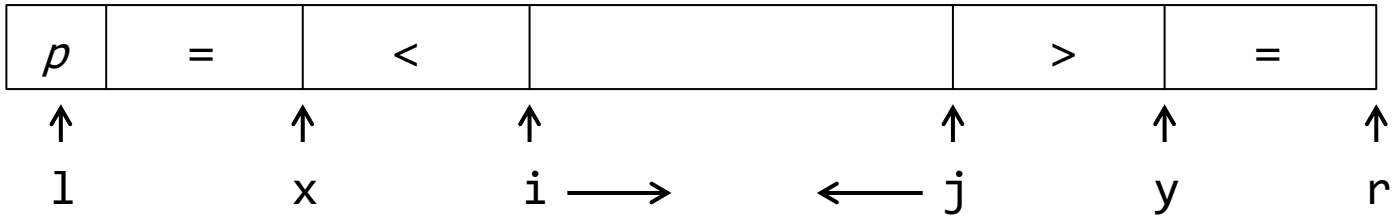
- $A[i] < p$: делаем swap $A[i]$ и $A[x]$ и увеличиваем оба индекса i и x
- $A[i] > p$: делаем swap $A[i]$ и $A[j]$ и уменьшаем индекс j
- $A[i] = p$: увеличиваем индекс i
- Каждая операция поддерживает инвариант разбиения и уменьшает расстояние $(j - i)$
- Но любой элемент, не равный p, приводит к swap-у:
  - Сколько будет swap-ов если всего один дубликат?

Разбиение на три части (Дейкстра)



- $A[i] < p$: делаем swap $A[i]$ и $A[x]$ и увеличиваем оба индекса i и x
- $A[i] > p$: делаем swap $A[i]$ и $A[j]$ и уменьшаем индекс j
- $A[i] = p$: увеличиваем индекс i
- Каждая операция поддерживает инвариант разбиения и уменьшает расстояние $(j - i)$
- Но любой элемент, не равный p, приводит к swap-у:
  - Если дубликатов немного, то число swap-ов будет гораздо больше чем при разбиении на две части!

Разбиение на три части (Бентли и Макилрой)



- Аналогично разбиению Хоара:
  - Увеличиваем i пока $A[i] < p$, и уменьшаем j пока $A[j] > p$
  - Делаем swap $A[i]$ и $A[j]$
  - $A[i] = p$: делаем swap $A[i]$ и $A[x]$ и увеличиваем x
  - $A[j] = p$: делаем swap $A[j]$ и $A[y]$ и уменьшаем y
- Не отработаем за один проход…
  - … но доп. overhead пропорционален числу дубликатов (нет дубликатов – нет доп. overhead-а)

Трехчастная поразрядная быстрая сортировка строк

- Сортируем массив указателей char *
- Сравнение строк работает за время ~ числу символов
- На последних стадиях алгоритма строки различаются только суффиксами: молодость, молоко, молоток
- При разбиении проверяем только один символ в позиции x (предполагая, что символы в позициях от 0 до $x-1$ совпадают):
  - 1) строки, у которых x-ый символ меньше x-ого символа опорного элемента, слева
  - 2) строки, у которых x-ый символ больше x-ого символа опорного элемента, справа
  - 3) строки, у которых x-ый символ равен x-му символу опорного элемента, посередине
- Средний подмассив сортируем с позиции $x+1$

Быстрая сортировка (глубина стека)

ITIVITI

LET IT TRADE

```
1. void quicksort(int * a, int l, int r) {
2.     if (r <= l) return;
3.     int p = partition(a, l, r);
4.     quicksort(a, l, p-1);
5.     quicksort(a, p+1, r);
6. }
```

- Устраним последний рекурсивный вызов на правом подмассиве (концевая рекурсия) …

```
1. void quicksort(int * a, int l, int r) {
2.     while (r > l) {
3.         int p = partition(a, l, r);
4.         quicksort(a, l, p-1);
5.         l = p+1;
6.     }
7. }
```

Быстрая сортировка (глубина стека)

- Рекурсивный вызов для меньшего из подмассивов...

```
1. void quicksort(int * a, int l, int r) {
2.    while (r > l) {
3.       int p = partition(a, l, r);
4.
5.       if (p-l < r-p) {
6.          quicksort(a, l, p-1);
7.          l = p+1;
8.       }
9.       else {
10.         quicksort(a, p+1, r);
11.         r = p-1;
12.      }
13.   }
14.}
```

Рандомизированная выборка

ITIVITI

- Нахождение *k*-го наименьшего числа (поиск *k* наименьших элементов)
- Например, найти 10% наименьших значений
- Можно вначале отсортировать массив и найти *k*, а можно …

```
1. void select(int * a, int l, int r, int k)
2. {
3.     if (r <= l) return;
4.     int p = partition(a, l, r);
5.     if (p > k) select(a, l, p-1, k);
6.     if (p < k) select(a, p+1, r, k);
7. }
```

- Если не везёт с разбиениями: $T(N) = T(N-1) + \Theta(N)$
- Если не хуже, чем ¾: $T(N) \leq T(3N/4) + \Theta(N)$

LET IT TRADE

Рандомизированная выборка

- Какая будет сложность, если разбиение не хуже, чем ¾?
  - $T(N) \leq T(3N/4) + \Theta(N)$ – геометрическая прогрессия
  - Следовательно, общее время $O(N)$
- Давайте посмотрим как убывает длина массива M (размер задачи) при рекурсивных вызовах
- Будем говорить, что алгоритм выполняется в фазе i, если текущий размер массива M находится между $(¾)^{i+1}N$ и $(¾)^{i}N$
  - X_i – число рекурсивных вызовов в фазе i
- Поскольку на каждом рекурсивном вызове мы делаем не более $c \cdot M$ операций, сложность алгоритма не превосходит $\sum_i X_i \cdot c(3/4)^i N$

Рандомизированная выборка

- Если алгоритм выбирает «хороший» опорный элемент, дающий разбиение не хуже, чем ¾, то текущая фаза заканчивается
- Какая вероятность получить такое разбиение?
  - Вероятность выбора «хороших» опорных элементов ½
- Какое среднее число шагов для получения «хорошего» опорного элемента?
  - Среднее число подбрасываний, чтобы выпал «орёл» = $\sum_{i=0}^{\infty} i \cdot Pr\{\text{«орёл» выпал на } i\text{-ом подбрасывании}\} = \sum_{i=0}^{\infty} i(1/2)^i = 2$
  - Мат. ожидание $E[X_i] = 2$
- Т.о. мат. ожидание времени работы $\leq \sum_i 2c(3/4)^i N \leq 8cN$

Нерекурсивная выборка

- Вместо рекурсивного вызова на левой или правой части массива двигаем указатели на границы массива

```
1. void select(int * a, int l, int r, int k)
2. {
3.    while (r > l) {
4.       int p = partition(a, l, r);
5.       if (p == k) break;
6.       if (p > k) r = p-1;
7.       if (p < k) l = p+1;
8.    }
9. }
```

Рандомизированная выборка (дополнительные вопросы)

- Поиск набора перцентилей (80%, 90%, 95%, 99%, 99.9%)
- Через последовательные вызовы `select()`
- Через сортировку массива и прямой доступ
- Через частичную сортировку (например от 80-перцентиля и до конца массива):
  - найдем 80-перцентиль с помощью выборки и потом отсортируем правый подмассив
  - безусловная рекурсия (сортировка) по одну сторону разбиения, условная рекурсия (выборка) – по другую: например, если попали на 90-перцентиль, то справа сортируем, слева выбираем

https://en.wikipedia.org/wiki/Partial_sorting

ITIVITI

LET IT TRADE

Спасибо за внимание!

УНИВЕРСИТЕТ ИТМО