Алгоритмы и структуры данных

Лекция 5
Сортировка. Поиск.

Сергей Леонидович Бабичев

План лекции

1. Сортировка с использованием свойств элементов.
2. Radix-sort.
3. Внешняя сортировка.
4. Сортировка и параллельные вычисления.
5. Сравнительный анализ методов сортировки.
6. Задача поиска. Абстракция поиска.
7. Абстракция *хранилище*.
8. Последовательный поиск.
9. Поиск с сужением зоны.
10. Распределяющий поиск. Поиск с использованием свойств ключа

Сортировка с использованием свойств элементов

Сортировка подсчётом

- Есть ли алгоритмы сортировки со сложностью меньшей $O(N \log N)$?

  Да, если использовать свойства ключей.
- Пусть множество значений ключей ограничено $D(K) = \{K_{min}, \ldots, K_{max}\}$.
- Тогда при наличии добавочной памяти в $|D(K)|$ ячеек сортировку можно произвести за $O(N)$.

Сортировка подсчётом

- Сортируем массив $S = \{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, 8\}$
- Заранее известно, что значения массива натуральные числа, которые не превосходят 20.
- Заводим массив $F[1..20]$, содержащий вначале нулевые значения.

$$F = \boxed{0}$$

- Проходим по массиву S. $S_1 = 10; F_{10} \leftarrow F_{10} + 1$.

$$F = \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

Сортировка подсчётом

- $S_2 = 5; F_5 \leftarrow F_5 + 1.$

$$F = \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

- $\ldots$
- $S_{12} = 5; F_8 \leftarrow F_8 + 1.$

$$F = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}$$

- Заключительный вывод по ненулевым элементам F:

$$S = \{2, 3, 4, 5, 5, 6, 7, 8, 10, 13, 14, 18\}$$

Сортировка подсчётом: особенности

- Ключи должны быть перечислимы.
- Пространство значений ключей должно быть ограниченным.
- Требуется дополнительная память $O(|D(K)|)$.
- Сложность $O(N) + O(|D(K)|)$

Поразрядная сортировка

- Усложнение варианта сортировки подсчётом - поразрядная сортировка.
- Разобьём ключ на фрагменты — разряды и представим его как массив фрагментов.
- Все ключи должны иметь одинаковое количество фрагментов.
- Пример: ключ 375 можно разбить на 3 фрагмента $\{3, 7, 5\}$, тогда ключ 5 — тоже на 3 $\{0, 0, 5\}$.

Поразрядная сортировка

Требуется отсортировать массив $S = \{153, 266, 323, 614, 344, 993, 23\}$.

- Будем полагать, что разбиение проведено на 3 фрагмента.
- $\{\{1,5,3\},\{2,6,6\},\{3,2,3\},\{6,1,4\},\{3,4,4\},\{9,9,3\},\{0,2,3\}\}$
- Рассматривая последний фрагмент, как ключ, устойчиво отсортируем фрагменты методом подсчёта.
- $\{\{1,5,3\},\{3,2,3\},\{9,9,3\},\{0,2,3\},\{3,4,4\},\{6,1,4\},\{2,6,6\}\}$
- Теперь отсортируем по второму фрагменту.
- $\{\{6,1,4\},\{3,2,3\},\{0,2,3\},\{3,4,4\},\{1,5,3\},\{2,6,6\},\{9,9,3\}\}$
- И, наконец, по первому фрагменту.
- $\{\{0,2,3\},\{1,5,3\},\{2,6,6\},\{3,2,3\},\{3,4,4\},\{6,1,4\},\{9,9,3\}\}$

Поразрядная сортировка: особенности

- Требует ключи, которые можно трактовать как множество перечислимых фрагментов.
- Требует дополнительной памяти $O(|D(K_i)|)$ на сортировку фрагментов.
- Сложность постоянна и равна $O(N \cdot |D(K_i)|)$.

Внешняя сортировка

Сортировка больших данных

- Сортировка больших данных — очень трудоёмкая задача.
- Две основных проблемы:
  1. Для сортируемых данных недостаточно быстрой оперативной памяти.
  2. Время сортировки превосходит приемлемые границы.
- При недостатке оперативной памяти применяют внешнюю сортировку.

Сортировка больших данных

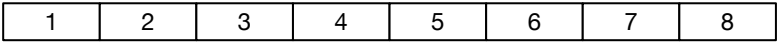
- Обработка всех данных одновременно невозможна.
- Используется абстракция **лента**, имеющая следующие методы:
- create/destroy
- open/close/rewind
- getdata/putdata

Сортировка больших данных

- Одним из хороших способов отсортировать внешние данные является использование операции *слияние*.
- Входными данными *двухпутевого слияния* являются две отсортированные ленты.
- Выходные данные — другая отсортированная лента.
- Чанк (chunk) — фрагмент данных, помещающихся в оперативной памяти.

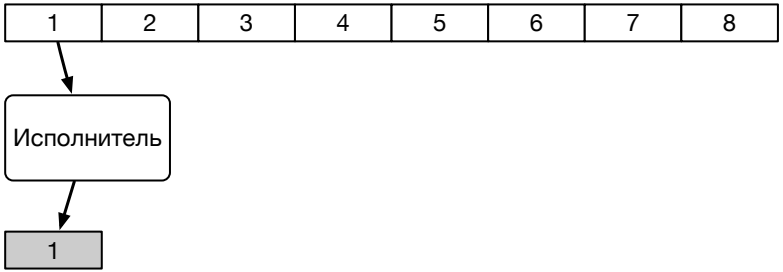
Сортировка слиянием

Пусть исходная лента содержит 8 чанков.



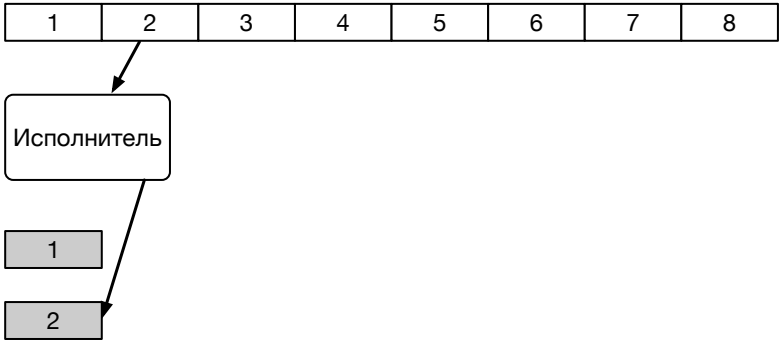
Сортировка слиянием

Первый этап. Считывается первый чанк, сортируется внутренней сортировкой и отправляется на первую временную ленту.



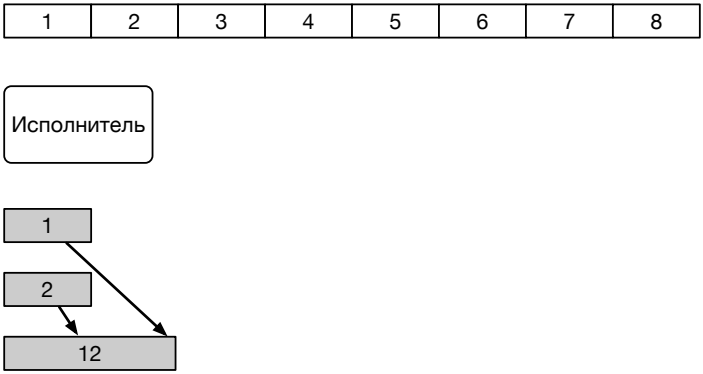
Сортировка слиянием

Второй этап. Второй чанк сортируется и отправляется на вторую временную ленту.



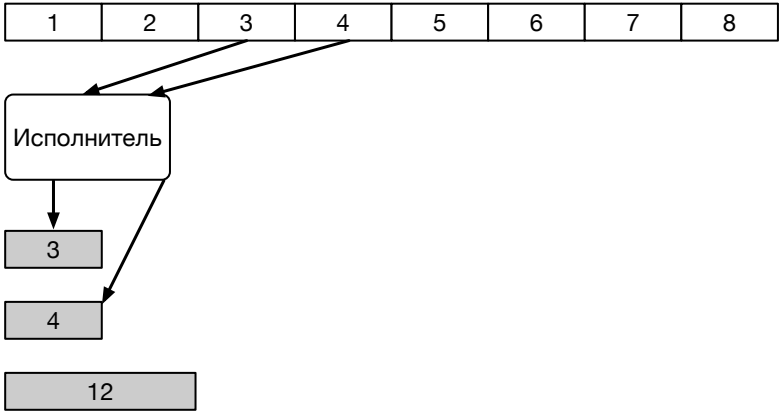
Сортировка слиянием

Третий этап — слияние. Сливаются первая и вторая временные ленты.



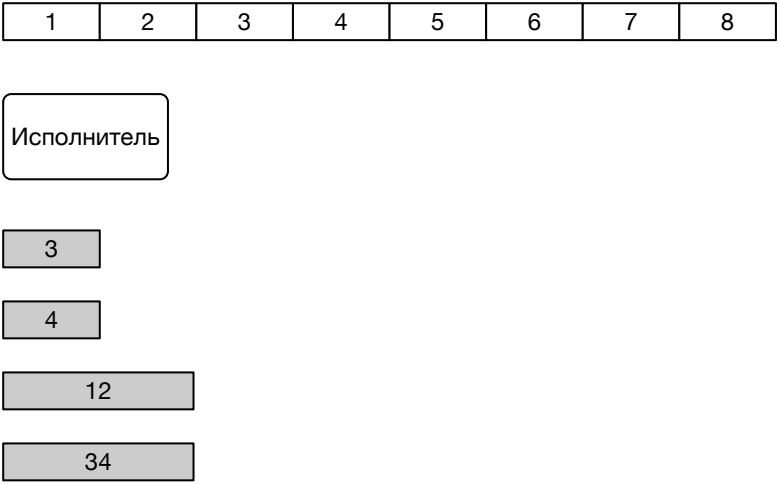
Сортировка слиянием

Аналогично считываются, сортируются и выводятся на временные ленты чанки 3 и 4.



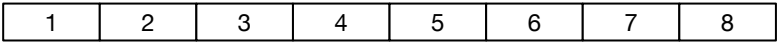
Сортировка слиянием

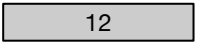
Аналогично временные ленты сливаются в ещё одну, четвёртую. Здесь мы не можем использовать меньшее количество лент.

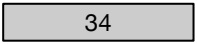


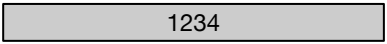
Сортировка слиянием

Сливаем ленты содержащие 12 и 34, получаем ленту 1234.



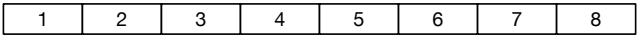


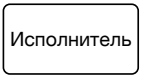


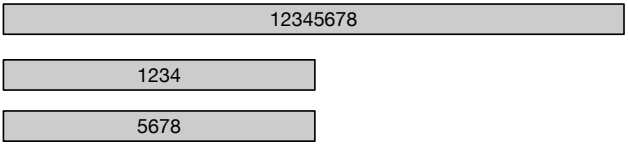


Сортировка слиянием

Для получения ленты 5678 требуется 4 временные ленты. Плюс лента 1234. Итого







— 5 лент.

Оценка сложности внешней сортировки слиянием

- Внутренних сортировок в этом алгоритме — K, количество чанков.
- Сложность внутренней сортировки

$$O\left(\frac{N}{K} \times \log \frac{N}{K}\right) = O(N \log N).$$

- Сложность операции слияния $O(N)$.
- Количество слияний $O(\log K)$.
- Общая сложность $O(N \log N)$.
- Сложность по количеству временных лент $O(\log K)$.

Усовершенствование алгоритма

- Мы заметили, что при операции слияния дополнительной памяти не требуется, достаточно памяти для двух элементов.
- Можно ли произвести внешнюю сортировку без использования большого буфера?
- Да, если отказаться от понятия *чанк* и использовать понятие *серия*.
- *Серия* — неубывающая последовательность на ленте.

Сортировка сериями

Пусть имеется лента

$$\underbrace{14, 4, 2, 7, 5, 9, 6, 11, 3, 1, 8, 10, 12, 13}.$$

Заводим две вспомогательных ленты, в каждую из которых помещаем очередную серию длины 1 из входной ленты.

$$\begin{cases} \underbrace{14}, \underbrace{2}, \underbrace{5}, \underbrace{6}, \underbrace{3}, \underbrace{8}, \underbrace{12} \\ \underbrace{4}, \underbrace{7}, \underbrace{9}, \underbrace{11}, \underbrace{1}, \underbrace{10}, \underbrace{13}. \end{cases}$$

Инвариант: внутри серии длины K все значения упорядочены по неубыванию.

Сортировка сериями

Каждую из серий парами сливаем в исходный файл.

$$\begin{cases} \underbrace{14}, \underbrace{2}, \underbrace{5}, \underbrace{6}, \underbrace{3}, \underbrace{8}, \underbrace{12} \\ \underbrace{4}, \underbrace{7}, \underbrace{9}, \underbrace{11}, \underbrace{1}, \underbrace{10}, \underbrace{13} .. \end{cases}$$

Инвариант операции слияния серий: совокупная последовательность является серией удвоенной длины.

$$\underbrace{4, 14}, \underbrace{2, 7}, \underbrace{5, 9}, \underbrace{6, 11}, \underbrace{1, 3}, \underbrace{8, 10}, \underbrace{12, 13}.$$

Сортировка сериями

Серии длины 2 попеременно помещаем на выходные ленты.

$$\underbrace{4, 14}, \underbrace{2, 7}, \underbrace{5, 9}, \underbrace{6, 11}, \underbrace{1, 3}, \underbrace{8, 10}, \underbrace{12, 13}.$$

$$\begin{cases} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{cases}$$

Сортировка сериями

Снова каждую из серий парами сливаем в исходную ленту.

$$\begin{cases} \underbrace{4, 14}, \underbrace{5, 9}, \underbrace{1, 3}, \underbrace{12, 13} \\ \underbrace{2, 7}, \underbrace{6, 11}, \underbrace{8, 10}. \end{cases}$$

Длина полных серий в выходной ленте не меньше 4. Только последняя серия может иметь меньшую длину.

$$\underbrace{2, 4, 7, 14}, \underbrace{5, 6, 9, 11}, \underbrace{1, 3, 8, 10}, \underbrace{12, 13}.$$

Сортировка сериями

Третий этап: формируем временные ленты сериями по 4.

$$\underbrace{2,4,7,14}, \underbrace{5,6,9,11}, \underbrace{1,3,8,10}, \underbrace{12,13}.$$

$$\begin{cases} \underbrace{2,4,7,14}, & \underbrace{1,3,8,10} \\ \underbrace{5,6,9,11}, & \underbrace{12,13}. \end{cases}$$

Длина полных серий в выходной ленте не меньше 4. Только последняя серия может иметь меньшую длину.

Сортировка сериями

Сливаем серии длины 4.

$$\begin{cases} \underbrace{2, 4, 7, 14}, & \underbrace{1, 3, 8, 10} \\ \underbrace{5, 6, 9, 11}, & \underbrace{12, 13}. \end{cases}$$

Слияние серий длины 4 обеспечивает длину серий 8.

$$\underbrace{2, 4, 5, 6, 7, 9, 11, 14}, \underbrace{1, 3, 8, 10, 12, 13}.$$

Сортировка сериями

Последний этап: разбивка на серии длины 8 с последующим слиянием.

$$\underbrace{2, 4, 5, 6, 7, 9, 11, 14}, \underbrace{1, 3, 8, 10, 12, 13}.$$

Разбивка:

$$\begin{cases} \underbrace{2, 4, 5, 6, 7, 9, 11, 14} \\ \underbrace{1, 3, 8, 10, 12, 13}. \end{cases}$$

Слияние:

$$\underbrace{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}.$$

Сортировка сериями: оценка сложности алгоритма

- За один проход примем операцию разбивки с последующим слиянием.
- На каждом проходе участвуют все элементы лент по два раза.
- Инвариант: длина серии после прохода k равна 2^k.
- Завершение алгоритма: длина серии не меньше N.
- Итого $\log N$ проходов.
- Сложность алгоритма: $O(N \log N)$
- Сложность по памяти: $O(1)$
- Сложность по ресурсам: две временные ленты.

Сортировка сериями: возможные улучшения

- Сортировка сериями: длина серии начинается от 1 и для полной сортировки требуется ровно $\lceil \log_2 N \rceil$ итераций.
- При этом первые итерации производятся с небольшой длиной серии.
- Идея! Сократить число итераций, использовав больше, чем 1 элемент памяти (опять ввести чанки).

Сортировка сериями: улучшенный вариант

- Подбираем такое число k_0, при котором серия длиной 2^{k_0} помещается в доступную память.
- Разбиваем исходную ленту на серии: считывается первый чанк длиной 2^{k_0}, сортируется внутренней сортировкой, пишется на первую ленту. Второй чанк после сортировки пишется на вторую ленту.
- После подготовки возвращаемся к алгоритму сортировки сериями.

Сложность алгоритма — $O(N \log N)$, количество итераций сокращается на k. Пусть $N = 10^8$. Тогда $\log_2 N \approx 27$. Для $k = 20$ в память помещается 2^{20} элементов, что вполне реально. Тогда общее количество итераций составит $27 - 20 + 1 = 8$ вместо 28. Profit!

Сортировка и параллельные вычисления

Сортировка и параллельные вычисления

- Современные компьютеры содержат по несколько исполнителей машинного кода.
- Как использовать несколько исполнителей для сортировки?

Особенности параллельного исполнения

- Каждый из исполнителей может исполнять свой поток инструкций.
- В программном коде это выглядит как одновременное исполнение нескольких функций.
- Все исполнителей могут иметь совместный доступ к общим данным.
- Исполнитель в современной терминологии называется *вычислительный поток*, *thread*.

Проблемы параллельного исполнения

.

- Совместный доступ к общим переменным — и благо и зло одновременно.
- Благо — так как это удобный способ взаимодействия, обмен данными.
- Зло — так как это может привести к конфликтам.

Проблемы параллельного исполнения

.

- Два исполнителя используют совместные переменные:

```
int a = 2, b = 10;

void thread1() {
   a += b; // 1a
   b = 5;  // 1b
}

void thread2() {
   b = 13; // 2b
   a *= b; // 2a
}
```

- Чему равны переменные a и b после окончания обоих исполнителей?

Проблемы параллельного исполнения

- Порядок исполнения недетерминирован и результатов может быть несколько при различных прогонах алгоритма.
- Задача становится комбинаторной.
- Результат зависит от взаимного порядка исполнения.
- Если обозначить за $t(x)$ абсолютное время получения результатов исполнения соответствующих инструкций, то известно лишь, что $t(1a) < t(1b)$ и $t(2b) < t(2a)$.
- Таким образом, возможных путей исполнения несколько:
- $1a \to 1b \to 2a \to 2b$
- $1a \to 2a \to 1b \to 2b$
- $1a \to 2a \to 2b \to 1b$
- $\ldots$

Проблемы параллельного исполнения

- Более того, операции $1a$ и $2a$ атомарны для исполнителя «Язык Си» и не атомарны для исполнителя «современный процессор»!
- Инструкция `a += b` превратится в несколько операций:
  1. Загрузка b в регистр процессора
  2. Загрузка a в регистр процессора
  3. Добавление значения b регистру, содержащему a
  4. Сохранение получившегося значения в a
- На любой из этих операций возможна передача управления другому исполнителю.

Проблемы параллельного исполнения

- Для борьбы с такими проблемами автор алгоритма должен использовать *примитивы синхронизации*
- На исполнение *примитивов синхронизации* требуется значительное время, за которое можно исполнить сотни и тысячи обычных операций.
- Простейший способ избежать проблем — ограничить использование общих данных *критическими секциями*.
- Основные операции лучше всего производить над *локальными* для каждого исполнителя данными и только в отдельные моменты использовать *точки синхронизации* для обмена.

Проблемы параллельного исполнения

- Разработка параллельных версий классических алгоритмов — отдельная, очень сложная задача.
- Алгоритмы, которые наиболее эффективны в варианте для одного исполнителя, чаще всего непригодны для варианта с несколькими исполнителями.

Сортировка и параллельные вычисления

- Параллельная сортировка имеет общие свойства с внешней:
  1. Данные разбиваются на непересекающиеся подмножества.
  2. Каждое подмножество обрабатывается независимо.
  3. После независимой обработки используется слияние.

Сравнительный анализ методов сортировки

Algo	Best case	Average case	Worst case	Memory	Stable?
Bubble	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Shell/Comb	$O(N^{\frac{7}{6}})$	$O(N^{\frac{7}{6}})$	$O(N^{\frac{4}{3}})$	$O(1)$	Нет
Insection	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Selection	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Quick	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(1)$	Нет/Да
Merge	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Да
Heap	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	Нет
Tim	$O(N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Да
Count	$O(N)$	$O(N)$	$O(N)$	$O(\|D\|)$	Да
Radix	$O(N)$	$O(N)$	$O(N)$	$O(\|R\| + N)$	Да/Нет

Абстракция *хранилище*.

Абстракция *хранилище*

- Хранилище содержит *ключи* и *значения*.
- Помимо операций создания и уничтожения хранилища реализуются операции:
  - `create` — создать новую пару;
  - `read` — найти пару по ключу;
  - `update` — изменить значение по ключу;
  - `delete` — удалить пару по ключу.

Структуры данных, реализующие эту абстракцию называются `CRUD`-структурами.

Задача поиска. Абстракция поиска.

Задача поиска. Абстракция поиска

Информация нужна для того, чтобы ей пользоваться.

Расширенная задача поиска:

1. Накопление информации (сбор)
2. Организация информации (переупорядочивание, сортировка)
3. Извлечение информации (собственно поиск)

Расширенная задача поиска

- Задача: построение эффективного хранилища данных.
- Требования:
  - Поддержка больших объёмов информации.
  - Возможность быстро находить данные.
  - Возможность быстро модифицировать данные.
- Реализация абстракций:
  - ***Create***
  - ***Read***
  - ***Update***
  - ***Delete***

Задача поиска. Абстракция поиска

- Имеется множество ключей

$$a_1, a_2, \ldots, a_n$$

- Требуется определить индекс ключа, совпадающего с заданным значением key, `find` — основа для `Read/Update/Delete`.

```
bunch a;
index = a.find(key);
```

$bunch$ — абстрактное хранилище элементов, содержащих ключи (массив, множество, дерево, список...).

Хорошая организация хранилища входит в расширенную задачу поиска.

Последовательный поиск.

Последовательный поиск

Ситуация: к поиску не готовились, ключи не упорядочены.

Индекс	0	1	2	3	4	5	6	7	8	9
Ключ	132	612	232	890	161	222	123	861	120	330
Данные	АВ	СА	ЯФ	АВ	АА	НД	ОР	ОС	ЗЛ	УГ

`find(a, 222) = 5`

`find(a, 999) = 10` (элемент за границей поиска).

Последовательный поиск

```
int dummysearch(int a[], int N, int key) {
   for (int i = 0; i < N; i++) {
       if (a[i] == key) {
          return i;
       }
    }
  return N;
}
```

Вероятность найти ключ в $i-$м элементе $P_i = \frac{1}{N}$

Матожидание числа поисков $E = \frac{N}{2}$

Число операций сравнения в худшем случае $2N$.

$$T(N) = 2 \cdot N = O(N)$$

Последовательный поиск

Небольшая подготовка:

Индекс	0	1	2	3	4	5	6	7	8	9	10
Ключ	132	612	232	890	161	222	123	861	120	330	999
Данные	АВ	СА	ЯФ	АВ	АА	НД	ОР	ОС	ЗЛ	УГ	??

Результаты не изменились.
`find(a, 222) = 5`
`find(a, 999) = 10` (элемент за границей поиска).

Последовательный поиск

```
int cleversearch(int a[], int N, int key) {
   a[n] = key;
   int i;
   for (i = 0; a[i] != key; i++)
       ;
   return i;
}
```

Число операций сравнения N в худшем случае.

$$T(N) = N = O(N)$$

Поиск ускорен в два раза!
Без подготовки лучших результатов не добиться.

Неупорядоченный массив

- Сложность операций:
  - ***Create*** — $O(1)$
  - ***Read*** — $O(N)$
  - ***Update*** — $O(N)$
  - ***Delete*** — $O(N)$

Поиск с сужением зоны.

Поиск с сужением зоны

Если в зоне поиска имеется упорядочивание — всё становится значительно лучше. Возможное действие: упорядочить по отношению.

- Имеется множество ключей

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$$

- Требуется определить индекс ключа, совпадающего с заданным значением key.

Поиск с сужением зоны

Принцип «разделяй и властвуй».

1. Искомый элемент равен центральному? Да — нашли.
2. Искомый элемент меньше центрального? Да — рекурсивный поиск в левой половине.
3. Искомый элемент больше центрального? Да — рекурсивный поиск в правой половине.

Поиск с сужением зоны

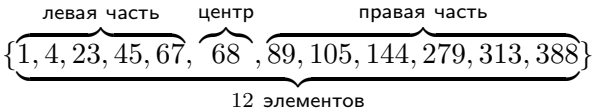
- Вход алгоритма: упорядоченный по возрастанию массив, левая граница поиска, правая граница поиска.
- Выход алгоритма: номер найденного элемента или -1.

```
int binarySearch(int val, int a[], int left, int right) {
   if (left >= right) return a[left] == val? left : -1;
   int mid = (left+right)/2;
   if (a[mid] == val) return mid;
   if (a[mid] < val) {
      return binarySearch(val, a, left, mid-1);
   }  else {
      return binarySearch(val, a, mid+1, right);
   }
}
```

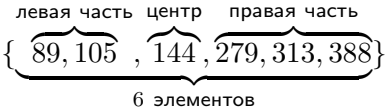
Поиск с сужением зоны

Оценка глубины рекурсии.

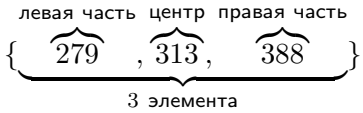
Поиск ключа 313:



$313 > 68 \rightarrow$ ключ справа



$313 > 144 \rightarrow$ ключ справа



$313 = 313 \rightarrow$ ключ найден

Поиск с сужением зоны

Попрактикуемся в основной теореме о рекурсии.

- Количество подзадач $a = 1$.
- Каждая подзадача уменьшается в $b = 2$ раза.
- Сложность консолидации $O(1) = O(N^0) \to d = 0$

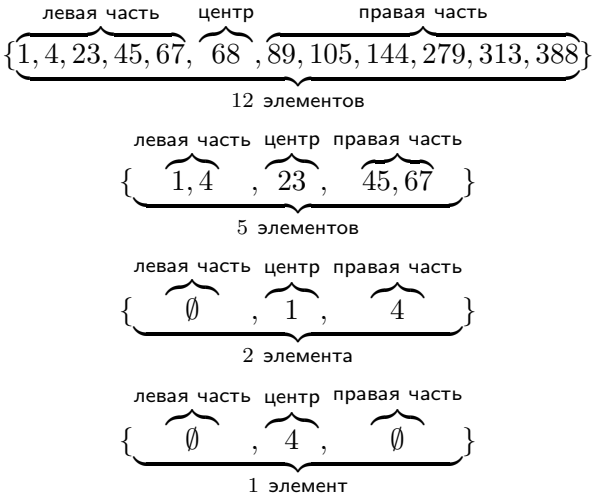
$d = \log_b a \to T(N) = \log N$

Результат можно получить и интуитивно.

Поиск с сужением зоны

Оценка глубины рекурсии.

Поиск отсутствующего 10:



Поиск с сужением зоны

Переход от рекурсии к итерации.

```
int binarySearch(int val, int a[], int left, int right) {
   while (left < right) {
      int mid = (left + right)/2;
      if (a[mid] == val) return mid;
      if (a[mid] < val) {
         right = mid - 1;
      }  else {
         left =  mid + 1;
      }
   }
   return a[left] == val? left : -1;
}
```

Распределяющий поиск. Поиск с использованием свойств ключа.

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- Без вспомогательных данных — нет.

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- Без вспомогательных данных — нет.
- Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- Без вспомогательных данных — нет.
- Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- Без вспомогательных данных — нет.
- Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.
- А быстрее можно?

Распределяющий поиск

- Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- Без вспомогательных данных — нет.
- Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.
- А быстрее можно?
- В некоторых случаях — да.

Распределяющий поиск

- Если $|D(Key)|$ невелико, то имеется способ, похожий на сортировку подсчётом.
- Создаётся **инвертированный массив**.

$a = \{2, 7, 5, 3, 8, 6, 3, 9, 12\}$. $|D(a)| = 12 - 2 + 1 = 11$.
$a_{inv}[2..12] = \{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$
$a[0] = 2 \to a_{inv}[2] = 0$
$a[1] = 7 \to a_{inv}[7] = 1$
$a[2] = 5 \to a_{inv}[5] = 2$
$a_{inv}[2..12] = \{0, 6, -1, -1, 5, -1, 4, 7, -1, -1, 8\}$

Распределяющий поиск

index	0	1	2	3	4	5	6	7	8
key	2	7	5	3	8	6	3	9	12

key	2	7	5	3	8	6	3	9	12
index	0	1	2	3	4	5	6	7	8

Распределяющий поиск

Два этапа. Первый этап — инвертирование.

```
int * prepare(int a[], int N, int *min, int *max) {
   *min = *max = a[0];
   for (int i = 1; i < N; i++) {
      if (a[i] > *max) *max = a[i];
      if (a[i] < *min) *min = a[i];
   }
   if (*max - *min > THRESHOLD) return NULL;
   int *ret = new int[*max - *min + 1];
   for (int i = *min; i <= *max; i++) {
      ret[i] = -1;
   }
   for (int i = 0; i < N; i++) {
      ret[a[i] - *min] = i;
   }
   return ret;
}
```

Распределяющий поиск

Второй этап: поиск.

```
// Preparation
int min, max;
int *ainv = prepare(a, N, &min, &max);
// Where is the key?
result = -1; // Not found value
if (key >= min && key <= max) result = ainv[key - min];
...
delete [] ainv;
```

- $O(N)$ на подготовку.
- $O(M)$ на поиск M элементов.
- $T(N, M) = O(N) + O(M) = O(N)$

Распределяющий поиск

- Сложность операций после подготовки:
  - ***Create*** — $O(1)$
  - ***Read*** — $O(1)$
  - ***Update*** — $O(1)$
  - ***Delete*** — $O(1)$
- Жёсткие ограничения на множество ключей.
- Сложность по памяти — $O(N) + O(|E|)$
- Запрет на повторяющиеся ключи.
- При наличии $f(key)$ сводится к хеш-поиску.

Спасибо за внимание.

Следующая лекция —
списки и деревья.