Сортировки

- Квадратичные сортировки
- Сортировка выбором, вставками, пузырьковая + её варианты
- Сортировки за nlog(n)
- Сортировка слиянием и быстрая сортировка
- Принцип разделяй и властвуй

Что входит в сортировку?

- $\{a_1, a_2, \ldots, a_n\} \rightarrow \{a'_1 \leq a'_2 \leq \ldots \leq a'_n\}$ или $\{a'_1 \geq a'_2 \geq \ldots \geq a'_n\}$
- Сортируем только те элементы к которым применима операция сравнения.
- Процесс сортировки состоит из операции сравнения и перестановки. Так же мы будем прибегать к операции поиска минимума.
- Операция **сравнения** всегда **тяжелее** операции перестановки, поэтому в некоторых случаях мы будем сравнивать количество перестановок и количество сравнений.
- Зачем вообще сортировать? Представление в пользовательском интерфейсе. Упрощает поиск.

Ключ сортировки

Элементы сортируемой последовательности могут иметь любые типы данных.
Обязательное условие — наличие ключа.

Например последовательность:
(Москва, 10000000), (Нью-Йорк, 12000000), (Париж, 9000000), (Токио, 20000000), (Лондон, 10000000), (Дели, 9000000)

Ключ - число жителей.

Устойчивость сортировки

Алгоритм сортировки устойчивый, если он сохраняет относительный порядок элементов.

Начальная последовательность:
(**Москва**, 10000000), (Нью-Йорк, 12000000), (Париж, 9000000), (Токио, 20000000), (**Лондон**, 10000000), (Дели, 9000000)

Устойчивая сортировка:
(Токио, 20000000), (Нью-Йорк, 12000000), (**Москва**, 10000000), (**Лондон**, 10000000), (Париж, 9000000), (Дели, 9000000)

Неустойчивая сортировка:
(Токио, 20000000), (Нью-Йорк, 12000000), (**Лондон**, 10000000), (**Москва**, 10000000), (Париж, 9000000), (Дели, 9000000)

Сортировка выбором

Selection sort

Ищем наименьший элемент в неотсортированной части

Сортировка выбором

Алгоритм

- Делим массив на две части. **Слева** пустая **отсортированная** часть, **справа не отсортированная**.
- На каждой итерации нам необходимо **найти минимум** в правой части массива и перенести ее в конец левой части.
- После каждого нахождения наименьшего элемента справа **сдвигаем границу вправо на 1**
- Другими словами на каждой итерации мы увеличиваем отсортированную часть и уменьшаем неотсортированную

Сортировка выбором

Инварианты

{ 5, 3, 15, 7, 6, **2**, 11, 13 }
{ 2, **3**, 15, 7, 6, 5, 11, 13 }
{ 2, 3, 15, 7, 6, **5**, 11, 13 }
{ 2, 3, 5, 7, **6**, 15, 11, 13 }
{ 2, 3, 5, 6, **7**, 15, 11, 13 }
{ 2, 3, 5, 6, 7, 15, **11**, 13 }
{ 2, 3, 5, 6, 7, 11, 15, **13** }
{ 2, 3, 5, 6, 7, 11, 13, **15** }
{ 2, 3, 5, 6, 7, 11, 13, 15 }

Сортировка выбором

```
function selectionSort(arr) {
    len = len(arr)
    for i = 0 .. len-1 {
        min = i
        for j = i+1 .. len {
            if arr[j] < arr[min] {
                swap(arr[j], arr[min])
            }
        }
    }
}
```

Сортировка выбором

Особенности

- Выбор каждого элемента требует прохода по правой части.
- Для сортировки массива из N элементов требуется N-1 проход (Почему -1? Последний элемент, который останется справа будет являться минимальным, так как он единственный)
- На каждой итерации проходим n-i элементов
- Массив может быть уже частично отсортирован и тогда нам потребуются только операции сравнения.

Сортировка выбором

Выводы

- **Во всех случаях** сложность **$O(n^2)$**
- Алгоритм устойчив.
- in-place.
- Количество операций обмена O(n) — может пригодиться для сортировок массивов с большими элементами.

Сортировка вставкой

Insertion sort

Берём первый элемент в неотсортированной части и вставляем на своё место в отсортированной части

Сортировка вставкой

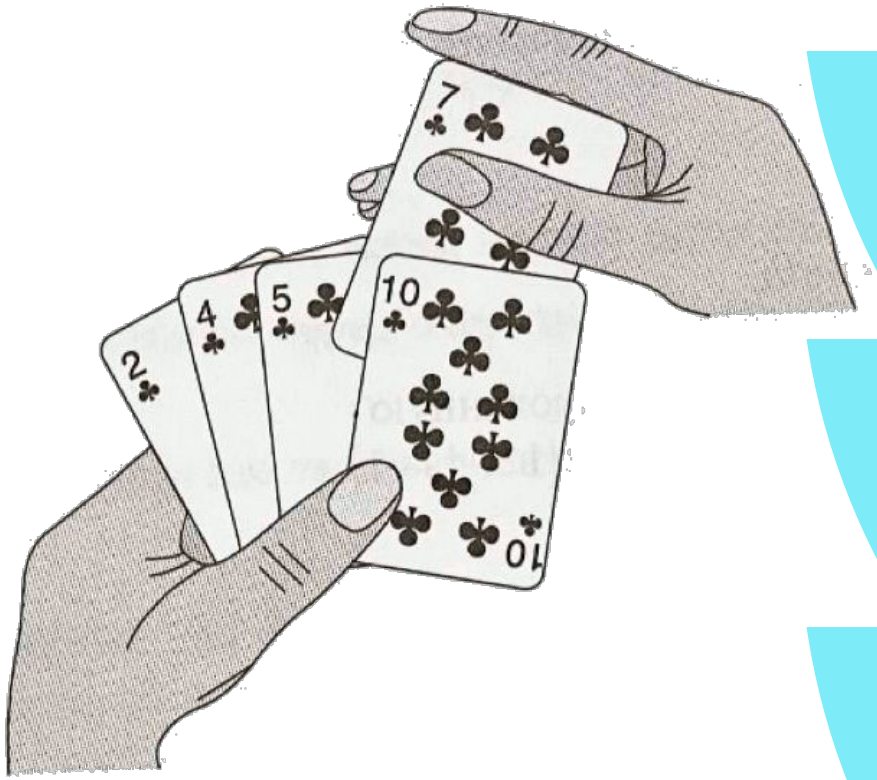
Алгоритм

- Делим массив на две части. **Слева отсортированная** часть, **справа не отсортированная**.
- На каждой итерации берём первый не отсортированный элемент, и **вставляем** в отсортированную часть **сдвигая вправо элементы** которые больше.
- После вставки **сдвигаем границу вправо на 1**.
- Другими словами на каждой итерации мы увеличиваем отсортированную часть и уменьшаем неотсортированную.

Сортировка вставкой

Инварианты

{ 5, **3**, 4, 6, 1, 3 }
{ 3, 5, **4**, 6, 1, 3 }
{ 3, 4, 5, **6**, 1, 3 }
{ 3, 4, 5, 6, **1**, 3 }
{ 1, 3, 4, 5, 6, **3** }
{ 1, 3, 3, 4, 5, 6 }



Сортировка вставкой

Наивная реализация.
В дальнейшем надо будет реализовать вставку с использованием бинарного поиска.

```
function insertionSort(arr) {
    for i = 1 .. len(arr) {
        j = i
        while j > 0 {
            if arr[j-1] > arr[j] {
                swap(arr[j-1], arr[j])
            }
            j--
        }
    }
}
```

Сортировка вставкой

Особенности

- Сортировка упорядоченного массива требует **O(n)**. Будет произведено 0 перестановок и n-1 сравнение.

- В худшем случае **$O(n^2)$**, если массив отсортирован по убыванию. Количество **сравнений** тогда равно: **$O(n^2)$** <= на 1-ой итерации 1, на 2 - 2, на 3-ей - 3 => 1 + 2 + … + n-1 = n*(n-1)/2.
  Количество **копирований $O(n^2)$** <=
  2*(n-1)+(1+2+…+n-1)=2*(n-1)+n*(n-1)/2

- В среднем случае **$O(n^2)$**

Сортировка вставкой

Выводы

- **В среднем** сложность **$O(n^2)$**
- Алгоритм устойчив.
- **in-place**, число дополнительных переменных не зависит от размера.
- Позволяет упорядочивать массив при динамическом добавлении новых элементов — online-алгоритм.

Сортировка пузырьком

bubble sort

Самый “лёгкий” элемент всплывает как пузырёк воздуха в воде

Сортировка пузырьком

Алгоритм

- Один из простейших в реализации алгоритмов.
- Основная идея: до тех пор, пока соседние элементы не в порядке, меняем их местами.
- Элемент всплывает наверх, обмениваясь с соседними местами. Отсюда и название.

Сортировка пузырьком

Инварианты

{ **5**, **3**, 15, 7, 6, 2, 11, 13}
{ 3, **5**, **15**, 7, 6, 2, 11, 13}
{ 3, 5, **15**, **7**, 6, 2, 11, 13}
{ 3, 5, 7, **15**, **6**, 2, 11, 13}
{ 3, 5, 7, 6, **15**, **2**, 11, 13}
{ 3, 5, 7, 6, 2, **15**, **11**, 13}
{ 3, 5, 7, 6, 2, 11, **15**, **13**}
{ 3, 5, 7, 6, 2, 11, 13, 15}
{ 3, 5, 6, 2, 7, 11, 13, 15}
{ 3, 5, 2, 6, 7, 11, 13, 15}
{ 3, 2, 5, 6, 7, 11, 13, 15}
{ 2, 3, 5, 6, 7, 11, 13, 15}

Сортировка пузырьком

```
function bubblesort(int a[]) {
    bool sorted = false
    while !sorted {
        sorted = true
        for i = 0..len(arr)-1 {
            if (arr[i] > arr[i+1]) {
                swap(arr[i], arr[i+1])
                sorted = false
            }
        }
    }
}
```

Сортировка пузырьком

Особенности

- Малоэффективный алгоритм, который используется исключительно в тренировочных целях.
- Имеет множество вариаций и улучшений.
- Учитывая, что сравнения долгие операции, а перемещения быстрые, то сортировка вставками более предпочтительна.

Сортировка пузырьком

Выводы

- Крайне проста в реализации и понимании.
- Алгоритм устойчив.
- **in-place**, сортирует на месте.
- Сложность в наилучшем случае O(n).
- Сложность в наихудшем случае $O(n^2)$

Быстрые сортировки

- Квадратичные устойчивые сортировки слишком медленны для того, чтобы сортировать большие последовательности.
- Мечта: а что, если бы мы имели два отсортированных массива, за какое время можно получить отсортированный массив, содержащий элементы обоих массивов?
- Разделяй и властвуй!

Разделяй и властвуй

Сортировка слиянием

Merge sort

Слияние — объединение отсортированных массивов.

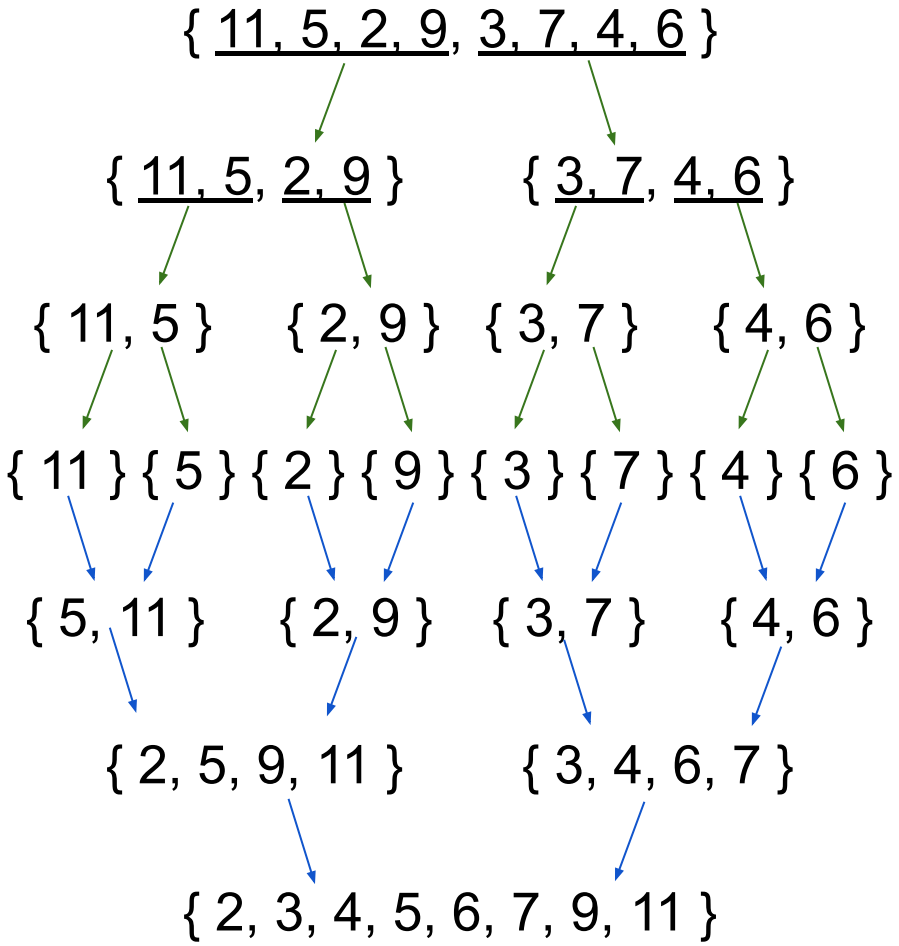
Сортировка слиянием

Алгоритм

- Делим массив пополам.
- Сортируем каждую половину независимо.
- Объединяем две отсортированные последовательности.
- Алгоритм слияния - два указателя.

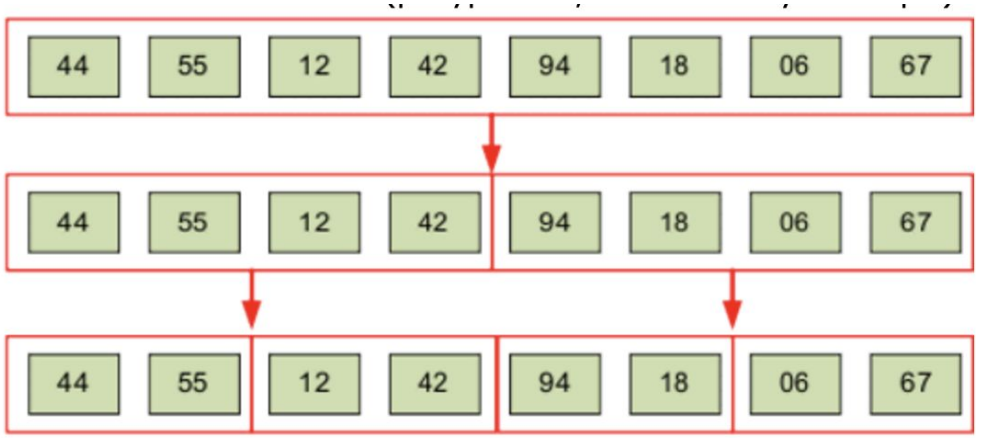
Сортировка слиянием

Инварианты



Сортировка слиянием

разделяй и властвуй,
нисходящая версия сверху вниз




```
function mergeSort(arr) {
    if len(arr) < 2 {
        return arr
    }
    mid = len(arr) / 2

    leftSide  = arr[0, mid]
    rightSide = arr[mid, len(arr)]

    return merge(mergeSort(leftSide),
                 mergeSort(rightSide))
}
```

Сортировка слиянием

слияние (два указателя)

```
function merge(a, b) {
    result = [], i = 0; j = 0
    while i < len(a) and j < len(b) {
        if a[i] < b[i] {
            result.append(a[i]); i++
        } else {
            result.append(b[j]); j++
        }
    }
    while i < len(a) {
        result.append(a[i]); i++
    }
    while j < len(b) {
        result.append(b[j]); j++
    }
    return result
}
```

Сортировка слиянием

Особенности

Принцип разделяй и властвуй — мастер-теорема в действии.

$$T(N) = T\left(\left\lceil \frac{N}{2} \right\rceil\right) + T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + \Theta(N)$$

Количество делений = log n
На каждое слияние нужно N операций

Сложность = n log n

Сортировка слиянием

Выводы

- Требует добавочно Θ(n) памяти.
- Сложность не зависит от входа и равна всегда Θ(n log n).
- Устойчива!
- Прекрасно подходит для внешней сортировки.
- Прекрасно подходит для параллельной сортировки.

Быстрая сортировка

Quick sort

Выбираем опорный элемент, всё что меньше переносим влево, всё что больше переносим вправо. Рекурсивно повторяем для каждой части.

Быстрая сортировка

Алгоритм

- Выбираем опорный элемент - pivot. В идеале медиану.
- Разбиваем массив “in place” на меньшие и больше значения.
- Повторяем алгоритм для большей и меньшей части.

Медиана — элемент, который находился бы в середине упорядоченного массива.
Median({1, 1, 1, 1, 1, 10}) = 1.0
Average({1, 1, 1, 1, 1, 10}) = 2.5

Быстрая сортировка1

Инварианты

{10, 5, 14, 7, 3, 2, 18, 4, 5, 13, 6, **8**}
{5, 7, 3, 2, 4, **5**, 6, 8}{10, 14, 18, 13}
{5, 3, 2, 4, 5}{7, 6, 8}

Быстрая сортировка

Наивная реализация.
В дальнейшем надо будет реализовать поиск медианы.

```
void qsortRecursive(arr[]) {
    i = 0, j = len(arr)-1
    pivot = arr[len(arr) / 2];
    while(i <= j) {
        while(arr[i] < pivot) i++;
        while(arr[j] > pivot) j--;
        if (i <= j) {
            swap(arr[i], arr[j])
            i++; j--
        }
    }

    if(j > 0)
        qsortRecursive(arr, j + 1);
    if (i < len(arr))
        qsortRecursive(&arr[i], len(arr) - i);
}
```

Быстрая сортировка

Особенности

- Худший случай: ведущим выбирается минимальный или максимальный элемент.
- Вероятность такого события крайне мала.
- Один из способов его избежать выбор медианы из трёх случайных элементов.

Быстрая сортировка

Выводы

- Может проводиться на месте.
- Сложность в наихудшем случае $O(n^2)$, но с крайне малой вероятностью.
- Сложность в среднем $O(n \log n)$.
- В прямолинейной реализации использует до $O(n)$ стека.

Быстрее быстрой сортировки?

- Если использовать особые свойства ключей мы можем сортировать со сложностью меньшей O(nlog n)

Сортировка подсчётом

Counting sort

Подсчитываем количество элементов каждого номинала

Сортировка подсчётом

Алгоритм

- 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 $O(n)$.
- Заранее известно, что значения массива натуральные числа, которые не превосходят 20.
- Заводим массив F[1..20], содержащий счетчики каждого значения.

Сортировка подсчётом

Инварианты

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

F_0 = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

F_1 = [0 0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0]

F_2 = [0 0 0 0 **1** 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]

F_3 = [0 0 0 0 1 0 0 0 0 1 0 0 0 **1** 0 0 0 0 0 0]

F_4 = [0 0 0 0 **2** 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0]

…

F_n = [0 1 1 1 2 1 1 1 0 1 0 0 1 1 0 0 0 1 0 0]

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

Сортировка подсчётом

Выводы

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

Поразрядная сортировка

Radix sort

Сравнения происходят по каждому разряду отдельно

Поразрядная сортировка

Алгоритм

- Разобьём ключ на фрагменты — разряды и представим его как массив фрагментов.
- Все ключи должны иметь одинаковое количество фрагментов.
- Устойчиво сортируем массив по каждому разряду отдельно.

Поразрядная сортировка

Инварианты

Сортируем массив S = {153, 266, 323, 614, 344, 993, 23}

{{1, 5, **3**}, {2, 6, **6**}, {3, 2, **3**}, {6, 1, **4**}, {3, 4, **4**}, {9, 9, **3**}, {0, 2, **3**}}
{{1, **5**, 3}, {3, **2**, 3}, {9, **9**, 3}, {0, **2**, 3}, {3, **4**, 4}, {6, **1**, 4}, {2, **6**, 6}}
{{**6**, 1, 4}, {**3**, 2, 3}, {**0**, 2, 3}, {**3**, 4, 4}, {**1**, 5, 3}, {**2**, 6, 6}, {**9**, 9, 3}}

$S^{/}$= {{0, 2, 3}, {1, 5, 3}, {2, 6, 6}, {3, 2, 3}, {3, 4, 4}, {6, 1, 4}, {9, 9, 3}}

Поразрядная сортировка

Выводы

- Требует ключи, которые можно трактовать как множество перечислимых фрагментов.
- Требуется дополнительная память $O(D(k_i))$ на сортировку фрагментов.
- Сложность $O(n \cdot D(k_i))$.

Итоги

Алгоритм	Лучший случай	Средний случай	Худший случай	Доп. память	Устойчивая?
Пузырьком	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Шелла	$O(N^{7/6})$	$O(N^{7/6})$	$O(N^{4/3})$	$O(1)$	Нет
Вставкой	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Выбором	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	Да
Быстрая	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(1)$	Нет/Да
Слияния	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Да
Пирамида	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	Нет
Подсчетом	$O(N)$	$O(N)$	$O(N)$	$O(D)$	Да
Поразрядная	$O(N)$	$O(N)$	$O(N)$	$O(R+N)$	Да/Нет

Спасибо!