АиСД 2020

Æ Куваев
vk.com/vinatorul
t.me/Vinatorul

Модули

1. ~~Временная сложность алгоритма~~
2. Сортировки
3. Алгоритмы на строках
4. Структуры данных стек, очередь, дек, связные списки
5. Двоичные деревья
6. Двоичная куча
7. Сбалансированные деревья поиска
8. Алгоритмы теории графов

Модуль 2: Сортировки

АиСД двадцать двадцатый

здесь
был кот

Постановка задачи

Дан массив **A** из **n** целых чисел. Необходимо упорядочить элементы массива так, чтобы они располагались в массиве по возрастанию (убыванию).

2.1 Квадратичные сортировки

Пузырьковая сортировка

```
for (i = 0; i < n - 1; i++):
    for (j = 0; j < n - i - 1; j++):
        if (A[j] > A[j + 1]):
            A[j] ↔ A[j + 1]
```

Разберём пример:

A = {5, 1, 7, 3, 9, 4, 1}

Оценим сложность

Рассмотрим первый раунд алгоритма i = 0

j = 0, {**5**, **1**, 7, 3, 9, 4, 1} -> {**1**, **5**, 7, 3, 9, 4, 1}

j = 1, {1, **5**, **7**, 3, 9, 4, 1} -> {1, **5**, **7**, 3, 9, 4, 1}

j = 2, {1, 5, **7**, **3**, 9, 4, 1} -> {1, 5, **3**, **7**, 9, 4, 1}

j = 3, {1, 5, 3, **7**, **9**, 4, 1} -> {1, 5, 3, **7**, **9**, 4, 1}

j = 4, {1, 5, 3, 7, **9**, **4**, 1} -> {1, 5, 3, 7, **4**, **9**, 1}

j = 5, {1, 5, 3, 7, 4, **9**, **1**} -> {1, 5, 3, 7, 4, **1**, **9**}

число 9 заняло “свою” позицию

Рассмотрим второй раунд алгоритма i = 1

j = 0, {**1**, **5**, 3, 7, 4, 1, 9} -> {**1**, **5**, 3, 7, 4, 1, 9}

j = 1, {1, **5**, **3**, 7, 4, 1, 9} -> {1, **3**, **5**, 7, 4, 1, 9}

j = 2, {1, 3, **5**, **7**, 4, 1, 9} -> {1, 3, **5**, **7**, 4, 1, 9}

j = 3, {1, 3, 5, **7**, **4**, 1, 9} -> {1, 3, 5, **4**, **7**, 1, 9}

j = 4, {1, 3, 5, 4, **7**, **1**, 9} -> {1, 3, 5, 4, **1**, **7**, 9}

число 7 заняло “свою” позицию

и так далее...

~~Стабильность!~~ Устойчивая сортировка

Сортировку будем называть **стабильной** (**устойчивой**), если она не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи, по которым происходит сортировка.

Вопросы для самоконтроля

1) Является ли пузырьковая сортировка стабильной?
2) Что нужно сделать, чтобы она стала (перестала) быть таковой?
3) Чем полезно свойство стабильности?

Сортировка вставками

```
for (i = 1; i < n; i++):
    j = i - 1
    x = A[i]
    while (j >= 0 and A[j] > x):
            A[j + 1] = A[j]
            j = j - 1
    A[j + 1] = x
```

Рассмотрим первый раунд алгоритма i = 1

x = 1

цикл while:

j = 0, {5, **1**, 7, 3, 9, 4, 1} -> {5, **5**, 7, 3, 9, 4, 1}

j = -1 - цикл прерывается т.к. j < 0

{**5**, 5, 7, 3, 9, 4, 1} -> {**1**, 5, 7, 3, 9, 4, 1}

Рассмотрим второй раунд алгоритма i = 2

x = 7

цикл while:

j = 1 - цикл прерывается т.к. A[j] < x (5 < 7)

{1, 5, **7**, 3, 9, 4, 1} -> {1, 5, **7**, 3, 9, 4, 1}

Рассмотрим третий раунд алгоритма i = 3

x = 3

цикл while:

j = 2, {1, 5, 7, **3**, 9, 4, 1} -> {1, 5, 7, **7**, 9, 4, 1}

j = 1, {1, 5, **7**, 7, 9, 4, 1} -> {1, 5, **5**, 7, 9, 4, 1}

j = 0 - цикл прерывается т.к. A[j] < x (1 < 3)

{1, **5**, 5, 7, 9, 4, 1} -> {1, **3**, 5, 7, 9, 4, 1}

и так далее...

Сортировка вставками в процессе ввода

```
read(A[0])
for (i = 1; i < n; i++):
    j = i - 1
    read(x)
    while (j >= 0 and A[j] > x):
            A[j + 1] = A[j]
            j = j - 1
    A[j + 1] = x
```

Сортировка выбором (поиском минимума)

```
for (i = 0; i < n - 1; i++):
    min_idx = i
    for (j = (i + 1); j < n; j++):
        if (A[j] < A[min_idx]):
            min_idx = j
    A[i] ↔ A[min_idx]
```

Рассмотрим первый раунд алгоритма i = 0, min_idx = 0

j = 1, {5, **1**, 7, 3, 9, 4, 1}, 1 < 5 -> min_idx = 1

j = 2, {5, 1, **7**, 3, 9, 4, 1}

j = 3, {5, 1, 7, **3**, 9, 4, 1}

j = 4, {5, 1, 7, 3, **9**, 4, 1}

j = 5, {5, 1, 7, 3, 9, **4**, 1}

j = 6, {5, 1, 7, 3, 9, 4, **1**}

{**5**, **1**, 7, 3, 9, 4, 1} -> {**1**, **5**, 7, 3, 9, 4, 1}

Рассмотрим второй раунд алгоритма i = 1, min_idx = 1

j = 2, {1, 5, **7**, 3, 9, 4, 1}

j = 3, {5, 1, 7, **3**, 9, 4, 1}, 3 < 5 -> min_idx = 3

j = 4, {5, 1, 7, 3, **9**, 4, 1}

j = 5, {5, 1, 7, 3, 9, **4**, 1}

j = 6, {5, 1, 7, 3, 9, 4, **1**}, 1 < 3 -> min_idx = 6

{1, **5**, 7, 3, 9, 4, **1**} -> {1, **1**, 7, 3, 9, 4, **5**}

и так далее...

Вопросы для самоконтроля

1) Являются ли сортировка вставками и сортировка выбором стабильными?
2) Что нужно сделать, чтобы они стали (перестали) быть таковыми?
3) Какой можно добиться лучшей временной сложности по данным сортировкам? (если можно)
4) Сколько требуется дополнительной памяти?

Количество обменов

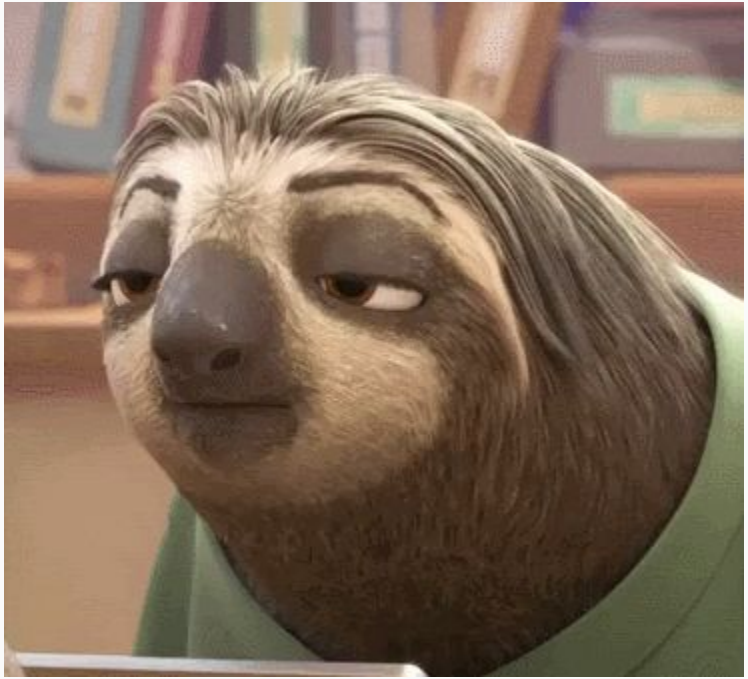
Если объекты имеют большой размер, то количество обменов может выступать важным параметров при выборе алгоритма сортировки.

Оценим предыдущие алгоритмы по количеству обменов

1) Пузырьковая сортировка - **$O(n^2)$**
2) Сортировка вставками - **$O(n^2)$**
3) Сортировка выбором (поиском минимума) - **$O(n)$**

Резюмируем

1) Пузырьковая сортировка - самая простая в написании
2) Сортировка вставками - умеет сортировать элементы в процессе ввода
3) Сортировка выбором - одна из немногих (и единственная квадратичная) сортировка с линейной зависимостью от количества обменов

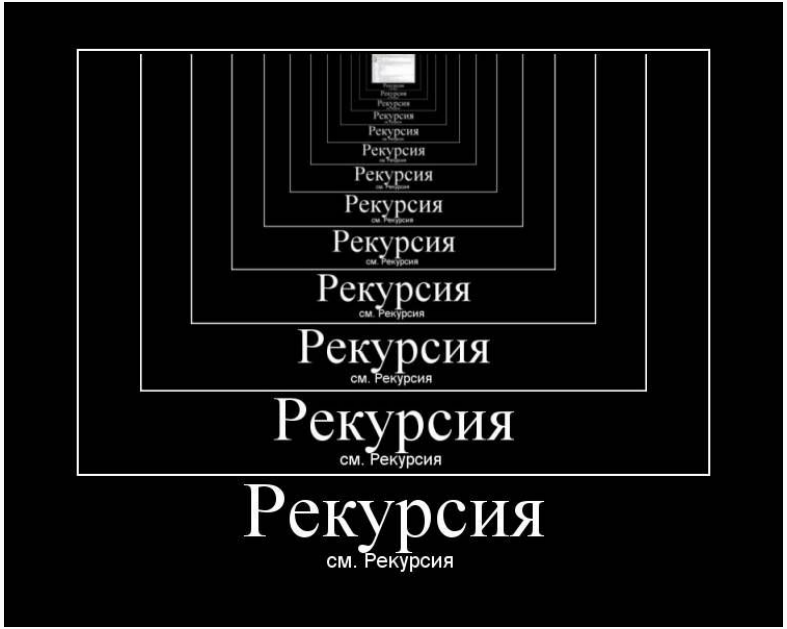


А можно как-то побыстрее?

2.2 Рекурсивные алгоритмы сортировки

Рекурсия (конечная)

Для любого конечного аргумента за конечное число рекурсивных обращений приводит к одному из отдельно определённых частных случаев, вычисляемых без рекурсии.



Сортировка слиянием (идея)

1) Разобьём массив на две половины
2) Каждую половину отсортируем слиянием (рекурсивно)
3) Объединим упорядоченные половины

Рекурсивное разбиение будем проводить, пока в массиве есть хотя бы два элемента.

3) Объединим упорядоченные массивы

```
function merge(A, B):
    i = 0, j = 0, k = 0, C = array[len(A) + len(B)]
    while (k < len(C)):
        if (j == len(B) or (i < len(A) and A[i] <= B[j])):
            C[k] = A[i]
            i = i + 1
        else:
            C[k] = B[j]
            j = j + 1
        k = k + 1
    return C
```

Пример работы функции merge (1 слайд из 2)

A = {1, 3, 5}, B = {2, 4, 5}

C = { , , , , , }

i = 0, j = 0, k = 0, A[i] <= B[j] (1 <= 2) -> C[k] = A[i], i++, k++

{ , , , , , } -> {**1**, , , , , }

i = 1, j = 0, k = 1, A[i] > B[j] (3 > 2) -> C[k] = B[j], j++, k++

{1, , , , , } -> {1, **2**, , , , }

i = 1, j = 1, k = 2, A[i] <= B[j] (3 <= 4) -> C[k] = A[i], i++, k++

{1, 2, , , , } -> {1, 2, **3**, , , }

i = 2, j = 1, k = 3, A[i] > B[j] (5 > 4) -> C[k] = B[j], j++, k++

{1, 2, 3, , , } -> {1, 2, 3, **4**, , }

Пример работы функции merge (2 слайд из 2)

A = {1, 3, 5}, B = {2, 4, 5}

C = {1, 2, 3, 4, , }

i = 2, j = 2, k = 4, A[i] <= B[j] (5 <= 5) -> C[k] = A[i], i++, k++

{1, 2, 3, 4, , } -> {1, 2, 3, 4, **5**, }

i = 3, j = 2, k = 5, i == len(A) -> C[k] = B[j], j++, k++

{1, 2, 3, 4, 5, } -> {1, 2, 3, 4, 5, **5**}

Аналогично работает случай j == len(B)

Оценка

1) Время работы: **O(len(A) + len(B))**
2) Объем дополнительной памяти: **O(len(A) + len(B))**
3) Количество обменов: **O(len(A) + len(B))**

1 + 2

```
function merge_sort(A):
    if (len(A) == 1):
        return A
    L = A[0 .. len(A) / 2 - 1]
    R = A[len(A) / 2 .. len(A) - 1]
    L = merge_sort(L)
    R = merge_sort(R)
    return merge(L, R)
```

Оценка

1) Время работы: **O(nlogn)**
2) Объем дополнительной памяти: **O(n)**, можно модифицировать до **O(1)**
3) Количество обменов **O(nlogn)**

Вопросы для самоконтроля

1) Является ли сортировка слиянием стабильной?
2) Что нужно сделать, чтобы она стала (перестала) быть таковой?
3) Что нужно изменить в идее алгоритма, чтобы сортировка не требовала дополнительной памяти?

Самостоятельное изучение

std::merge (в <algorithm>)
std::inplace_merge (также в <algorithm>) - в теории умеет в O(1) по памяти, но за O(nlogn) по времени

2.3 Быстрая сортировка

Идея алгоритма

1) Выберем из массива элемент, будем называть его опорным. От выбора опорного элемента не зависит корректность, но может зависеть эффективность.
2) Сравним все элементы с опорным и переставим их так, чтобы все элементы меньше опорного оказались левее, а все больше либо равные - справа.
3) Для левой и правой части выполним рекурсивно те же операции, если количество элементов в части больше единицы.

Псевдокод

```
procedure quicksort(A, l, r):
    if (r - l <= 1)
        return
    q = partition(A, l, r)
    quicksort(A, l, q)
    quicksort(A, q, r)
```

Что за partition(A, l, r) ???

Процедура partition - основной шаг работы алгоритма. Она переставляет элементы массива на полуинтервале [l, r) нужным образом. А затем возвращает индекс опорного элемента.

Одна из популярных реализаций - разбиение Хоара.

Стратегия разбиения Хоара

1) В качестве разделяющего элемента выбирается A[(l + r) / 2]
2) Начинается просмотр с левого конца массива, который продолжается до тех пор, пока не будет найден элемент, превосходящий по значению разделяющий элемент
3) Выполняется просмотр, начиная с правого конца массива, который продолжается до тех пор, пока не отыскивается элемент, который по значению меньше разделяющего

Стратегия разбиения Хоара

4) Оба элемента, на которых просмотр был прерван, очевидно, находятся не на своих местах в разделенном массиве, и потому они меняются местами.
5) Так продолжаем дальше, пока не убедимся в том, что слева от левого указателя не осталось ни одного элемента, который был бы больше по значению разделяющего, и ни одного элемента справа от правого указателя, которые были бы меньше по значению разделяющего элемента.

Псевдокод

```
function partition(A, l, r):
    v = A[(l + r) / 2], i = l, j = r - 1
    while (i <= j):
        while (A[i] < v):
            i++
        while (A[j] > v):
            j--
        if (i >= j):
            break
        A[i] ↔ A[j], i++, j--
    return j + 1
```

Пример работы partition

Снова возьмём массив {5, 1, 7, 3, 9, 4, 1}
v = 3 i = 0, j = 6
цикл while A[i] < v не отработает ни разу, т.к. 5 > 3
цикл while A[j] > v также не отработает, т.к. 1 < 3
A[0] меняется с A[6] {**5**, 1, 7, 3, 9, 4, **1**} -> {**1**, 1, 7, 3, 9, 4, **5**}
i = 1, j = 5
цикл while A[i] < v сделает один шаг, т.к. 1 < 3, но 7 > 3, i станет равно 2
цикл while A[j] > v сделает два шага: 4 > 3 и 9 > 3, но 3 не > 3
j станет равно 3
A[2] меняется с A[3] {1, 1, **7**, **3**, 9, 4, 5} -> {1, 1, **3**, **7**, 9, 4, 5}
i = 3 j = 2, условие i <= j не выполняется, разбиение получено.

А что по ассимптотике?

Лемма

Среднее время работы алгоритмы быстрой сортировки составляет O(nlogn)

Доказательство? Для самостоятельного изучения (см Викиконспекты)

Худший случай

Предположим, что после разбиения массива длины **n** одна часть содержит **n-1** элементов, а вторая - **1**.

Время работы процедуры разбиения **Θ(n)**.

T(n) = T(n - 1) + Θ(n) = Θ(1) + Θ(2) + … + Θ(n - 1) + Θ(n) = **Θ(n^2)**

Резюмируем

1) Оценка по времени в среднем O(nlogn)
2) Оценка по памяти в среднем O(logn)
3) Количество обменов в среднем O(nlogn)
4) Неустойчива

2.4 Сортировки с линейным временем исполнения

Сортировка подсчётом

Алгоритм сортировки целых чисел или счётного количества более сложных объектов за линейное время.

Нередко данные необходимо предварительно подготавливать.

Сортировка целых чисел в диапазоне от 0 до k

1) Пусть задан массив **A** с числами, лежащими в диапазоне от **0** до **k** (включительно)
2) Заведём вспомогательный массив **C** размера **k + 1**
3) Последовательно пройдём по массиву **A** и запишем в **C[i]** количество элементов, равных **i**
4) Теперь пройдём по массиву **C** и для каждого **i** в массив **A** запишем последовательно значение **C[i]** раз

Реализация

```
procedure count_sort(A, k):
    C = array[k + 1]
    for (int i = 0; i < len(A); i++):
        C[A[i]]++
    pos = 0
    for (int i = 0; i < len(C); i++):
        for (int j = 0; j < C[i]; j++):
            A[pos++] = i
```

Пример работы

Возьмём массив A = {5, 1, 6, 3, 8, 4, 1}
k = 8, C = {0, 0, 0, 0, 0, 0, 0, 0, 0}
После подсчтёта массив
C = {0, 2, 0, 1, 1, 1, 1, 0, 1}
Заполним массив A
pos = 0, i = 0 - на этом шаге ничего не происходит C[0] == 0
i = 1 - на этом шаге в массив A переносятся две единицы C[1] ==2
{**5**, **1**, 6, 3, 8, 4, 1} -> {**1**, **1**, 6, 3, 8, 4, 1}, pos = 2
i = 2 - на этом шаге ничего не происходит C[2] == 0
i = 3 - на этом шаге в массив A переносится одна тройка C[3] == 1
{1, 1, **6**, 3, 8, 4, 1} -> {1, 1, **3**, 3, 8, 4, 1}, pos = 3
и так далее

Оценка

1) Первый цикл работает за время порядка **len(A)**
2) Второй цикл за время порядка **len(C)**
3) Вложенный в него цикл работает за время порядка **C[i]**, а поскольку сумма всех **C[i]** в точности равна **len(A)**

То временная сложность алгоритма равна O(**len(A)** + **len(C)**) или **O(n + k)**

Плохие новости

1) Сложность по памяти **O(k)**
2) Устойчива?!
3) Количество обменов **O(n + k)**
4) Подготовка данных может занимать время порядка **O(nlogn)**

Поразрядная сортировка (radix)

Один из вариантов сортировки, использующих внутреннее устройство сортируемых объектов.
Предназначена для сортировки массива последовательностей одинаковой длины, на которых задано отношения порядка (в самом простом виде цифры или буквы).
Элементы последовательностей будем называть **разрядами** по аналогии с разрядами в числах.

Идея

Будем сортировать последовательности какой-либо устойчивой сортировкой по каждому разряду, от младшего к старшему (LDS), т.е. справа-налево.

Существует модификация алгоритма с сортировкой от старшего к младшему (MSD), т.е. слева-направо.

Псевдокод сортировки с использованием сортировки подсчётом

```
procedure radix_sort(A, m, k):
    for (int i = m - 1; i >= 0; i--):
        C = array[k + 1], B = array[len(A)]
        for (int j = 0; j < len(A); j++):
            C[A[j][i]]++
        count = 0
        for (int j = 0; j < len(C); j++):
            t = C[j]
            C[j] = count
            count += t
        for (int j = 0; j < len(A); j++):
            B[C[A[j][i]]] = A[j]
            C[A[j][i]]++
        A = B
```

Оценка времени

1) Внешний цикл работает за время порядка количества разрядов **O(m)**
2) В качестве устойчивой сортировки используется модификация сортировки подсчётом с временем **O(n + k)**, где **k** - количество различных значений в **<u>одном разряде</u>**

Итоговая сложность сортировки **O(m(n + k))**

Резюмируем

1) Сложность по памяти и по времени зависит от используемой сортировки
2) Сортирует последовательности одинаковой длины
3) Сложность по времени **O(m(n + k))**, где **m** - количество разрядов, **k** - количество возможных значений одного разряда
4) Сложность по памяти (при использовании сортировки подсчётом) **O(n + k)**, **k** - количество возможных значений одного разряда
5) Количество обменов порядка **O(mn)**

Блочная (карманная сортировка)

Алгоритм сортировки, основанный на предположении о равномерном распределении входных данных.

Если входные элементы подчиняются равномерному закону распределения, то математическое ожидание времени работы алгоритма карманной сортировки является линейным.

Идея

1) Разобьём массив на **k** последовательных непересекающихся блока (отрезка, кармана)
2) Каждый из блоков сортируется либо другой сортировкой, либо рекурсивно
3) После сортировок внутри каждых блоков данные записываются в массив в порядке разбиения на блоки.

Реализация и оценка

Для самостоятельного изучения.

См Викиконспекты и курс на openedu (3 неделя)

1) По времени **$O(n\log_k n)$**
2) По памяти **$O(n)$**