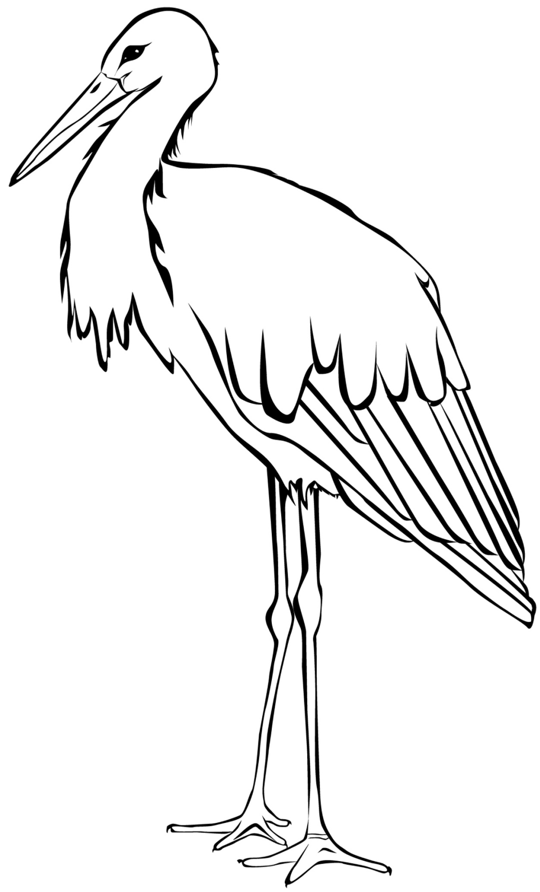


Алгоритмы и структуры данных

Конспекты лекций основного потока

Лектор: С. А. Объедков

Орлов Никита, Евсеев Борис, Рубачев Иван

НИУ ВШЭ, 2017

Содержание

Лекция 1. Асимптотика, простые алгоритмы, сортировка вставками

Пусть перед нами стоит задача: найти в некотором массиве медиану. Техническое задание выглядит так: на вход программе подается массив A, на выходе хотим получить одну из медиан, неважно какую.

Напомним определение медианы m:

$$m \in A = \begin{cases} |\{a \in A \mid a < m\}| \leqslant \frac{|A|}{2} \\ |\{a \in A \mid a > m\}| \leqslant \frac{|A|}{2} \end{cases}$$

Словами: медиана это такое число, что оно не больше половины элементов, но и не меньше половины элементов.

Легко видеть, что разных медиан в массиве может быть не больше двух, в зависимости от четности числа элементов.

Есть несколько способов решить эту задачу. Приведем несколько из них:

Алгоритм 1 Алгоритм поиска медианы

Ввод: Массив A

Вывод: Медиана m массива A

```
function Median(A)
    n := len(A)
    for i := 0 to (n − 1) do
        l := 0
        g := 0
        for j := 0 to (n − 1) do
            if A[j] < A[i] then
                l := l + 1
            else if A[j] > A[i] then
                g := g + 1
        if l ⩽ n/2 and g ⩽ n/2 then
            return A[i]
```

Посмотрим еще на один способ:

Алгоритм 2 Примитивный алгоритм поиска медианы

Ввод: Массив A

Вывод: Медиана m массива A

```
function Median(A)
    n := len(A)
    B := sorted(A)
    return B[⌊n/2⌋]
```

На первый взгляд это сложный подход, так как мы должны отсортировать массив и пока не знаем, как это сделать.

Итак, у нас есть как минимум два способа найти медиану. Возникает абсолютно естественное желание как-нибудь выяснить, какой лучше. Оказывается, в программировании можно провести сразу несколько таких оценок по разным критериям. Два главных ресурса, которые потребляют алгоритмы, это процессорное время и память вычислительного устройства.

Определение 1.1. Время (измеренное в некой абстрактной единице), необходимое алгоритму для завершения своей работы, называется *временем работы алгоритма* и обозначается как $T(n)$, где n - длина входных данных.

Время работы можно считать в разных единицах, например в *секундах*, если реализация алгоритма и исполнитель фиксированы, или в *элементарных операциях*, если речь идет про машину Тьюринга.

Различают несколько оценок времени работы:

1. *Худший случай* - максимально возможное $T(n)$ на входе длины n. Чаще всего используется на практике, так как дает верхнюю оценку времени работы алгоритма.
2. *Средний случай* - математическое ожидание $T(n)$ на входе длины n. Используется на практике реже, чем худший случай, в силу частой неопределенности вероятностного пространства для вычисления матожидания.
3. *Лучший случай* - минимально возможное $T(n)$ на входе длины n. На практике не используется, так как к любому сколько угодно неэффективному алгоритму можно приписать проверку на оптимальность входных данных и выдать ответ быстрее, чем средний или худший случай. Например, в задаче про поиск медианы можно проверять, отсортирован ли массив, и, если он не отсортирован, честно запускать поиск.

Для всего зоопарка алгоритмов существует инструмент их анализа - *асимптотический анализ*. Это методология, в которой время работы и занимаемая память алгоритма ставятся в соответствие классу функций.

Для начала дадим несколько определений.

Определение 1.2. О-большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\underline{\underline{O}}(g(n)) = \{f(n) \mid \exists c_2 > 0, n_0 > 0 \; \forall n \geqslant n_0 : 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

Иными словами, запись $f(n) \in O(g(n))$ означает, что $f(n)$ растет не быстрее, чем $g(n)$.

Определение 1.3. о-малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\overline{\overline{o}}(g(n)) = \{f(n) | \forall c_2 > 0 \exists n_0 > 0 : \forall n \geqslant n_0 : 0 \leqslant f(n) < c_2 g(n)\}$$

Определение 1.4. Ω-большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$f(n) \in \Omega(g(n)) \leftrightarrow g(n) = \underline{\underline{O}}(f(n))$$

Определение 1.5. ω-малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию
$$f(n) \in \omega(g(n)) \leftrightarrow g(n) = \overline{\overline{o}}(f(n))$$

Определение 1.6. $\Theta(g(n))$ называется такое множество функций, которое удовлетворяет условию
$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \ \forall n \geqslant n_0 : \ 0 \leqslant c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)\}$$
Иными словами, $\Theta(g(n))$ растет примерно также, как и $f(n)$.

В нашем курсе мы часто будем писать что-то похожее на
$$T(n) = \Theta(f(n))$$
Такая запись с точки зрения математики некорректна, но мы будем понимать знак равенства как
$$T(n) \in \Theta(f(n))$$

Например:
$$4n^2 + 12n + 12 = \Theta(n^2)$$
$$c_1 = 1, \ c_2 = 16, \ n_0 = 2$$
$$\forall n \geqslant n_0 : \ 0 \leqslant n^2 \leqslant 4n^2 + 12n + 12 \leqslant 16n^2$$

В общем случае верно следующее:

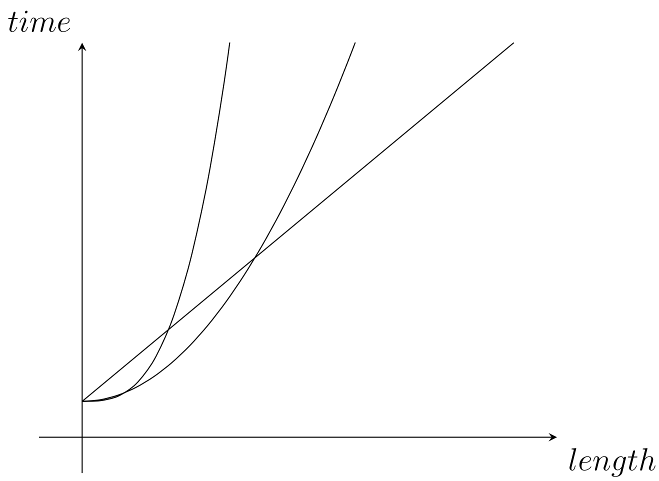
Лемма 1.7. *Если многочлен $p(n)$ представим в виде*
$$p(n) = \sum_{i=0}^{d} a_i n^i, \ d = \deg(p), \ a_d > 0,$$
то
$$p(n) = \Theta(n^d)$$

Замечание 1.8. Обычно функцию, описывающую время работы или память, занимаемую алгоритмом, называют *оценкой времени работы или памяти* алгоритма.

Замечание 1.9. По соглашению мы рассматриваем *асимптотически неотрицательные* функции, то есть такие, что
$$\exists n_0 \ \forall n > n_0 : \ f(n) \geqslant 0$$

Теперь поймем, что скрывается за классами функций Θ.

Пусть есть классы $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$. Для некоторых алгоритмов существует *оценка*, принадлежащая одному из этих классов. Помня про константу, можно сказать, что на достаточно большом объеме данных алгоритм с меньшей оценкой будет работать в среднем быстрее. Это ключевая мысль асимптотического анализа. Представить ее можно, построив графики неких линейной, квадратичной и кубической функций.



Для теоретического анализа сложности алгоритма берутся достаточно большие числа, но нужно понимать, что на практике может оказаться так, что входные данные могут быть меньше, чем n_0

Теперь получив матаппарат, оценим время работы алгоритмов поиска медианы.

Замечание 1.10. Будем считать, что элементарные арифметические операции, операции присваивания, копирования и тому подобные выполняются за $\Theta(1)$, иначе говоря, время их выполнения константо.

Первый алгоритм:

1. Лучший случай: медиана на первом месте. Тогда алгоритм выполнит одну итерацию внешнего цикла, n итераций внутреннего цикла, каждая из которых занимает константное время, и завершит работу. Сложность: $\Theta(1 \cdot n) = \Theta(n)$. Такая сложность считается достаточно хорошей.

2. Худший случай: медиана на последнем месте. Тогда алгоритм выполнит n итераций внешнего цикла, на каждой итерации произойдет n итераций внутреннего цикла. Сложность: $\Theta(n^2 - n) = \Theta(n^2)$.

Доказательство корректности заключается в том, что алгоритм *реализует* определение медианы. В таком случае он корректен, пока нет ошибок на уровне написания кода.

Второй алгоритм:

Второй алгоритм сложнее для оценки, так как мы не знаем, как сортируем массив. Операция взятия элемента выполняется за $\Theta(1)$. Остается сортировка, которую можно выполнить разными способами за разное время.

Давайте возьмем простой алгоритм сортировки и оценим его сложность.

Алгоритм 3 Сортировка вставками

Ввод: Массив A с заданным на нем порядком $<$.

Вывод: Отсортированный по возрастанию массив A.

```
function INSERTION_SORT(A)
    n := len(A)
    for i := 1 to (n − 1) do
        k := A[i]
        for j := i − 1 to 0 do
            if k < A[j] then
                A[j + 1] := A[j]
            else
                break
        A[j + 1] := k
    return A
```

Словами: смотрим каждый i элемент и ищем его место среди первых $i-1$ элементов.

Для начала докажем корректность алгоритма. Для этого будем использовать *инвариант* - свойство математического объекта, которое не меняется после преобразования объекта.

Теорема 1.11. *Пусть есть неупорядоченный пронумерованный набор А элементов с заданным на них порядком меньше, и мы исполняем над ним алгоритм. Инвариант: элементы* $A[0], \ldots, A[i-1]$ *являются перестановкой исходных элементов в правильном порядке.*

Доказательство. Докажем по индукции. База $i = 1$ верна. Пусть инвариант верен для $i-1$ шага. Тогда смотрим $k = A[i]$ элемент.

Возможны 3 случая:

1. k - самый большой среди первых i элементов. Тогда алгоритм пропустит эту итерацию и перейдет к следующему.

2. k - самый маленький среди первых i элементов. Тогда алгоритм передвинет его в начало, пройдя весь цикл.

3. В противном случае, мы начинаем перебирать все элементы среди первых i до тех пор, пока операция сравнения на "меньше" не вернет ложь. Это означает, что мы в отсортированном массиве нашли элемент под номером j, который меньше либо равен k:

$$A[j] \leqslant k \ \leqslant \ A[j+1]$$

   Тогда мы сдвигаем элементы с $j+1$ до i на одну позицию вправо, и на $j+1$ место ставим k.

$$A[j] \leqslant k = A[j+1] < A[j+2]$$

   Все элементы с 0 по j позицию меньше либо равны k, а элементы с $j+2$ по i позицию они больше k.

[:|||:]

Из доказательства корректности инварианта прямо следует доказательство корректности алгоритма: когда алгоритм закончит свою работу, $i = n$, а значит инвариант верен для n элементов, а значит массив отсортирован.

Теперь можно оценить время работы сортировки вставками:

1. Лучший случай: массив уже отсортирован. Но тогда внешний цикл совершит $n-1$ итерацию, на каждой из которых произойдет одно сравнение. Сложность получилась $\Theta(n)$.

2. Худший случай: массив отсортирован в обратном порядке. Тогда на каждой итерации число шагов внутреннего цикла будет уменьшаться на 1. Значит

$$T(n) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$

Лекция 2. Merge sort, Binary search, рекуррентные соотношения

Поговорим еще немного про сортировки. Сортировка вставками имеет квадратичную сложность, что не оптимально. Есть более быстрые алгоритмы, один из них называется *сортировкой слиянием.*

Сортировка слиянием

Суть сортировки достаточно проста: если у нас есть две отсортированные последовательности, мы можем их объединить в одну отсортированную, а последовательность длины один уже отсортирована, а значит мы можем разбить наш массив на блоки одинаковой длины, и рекурсивно отсортировать.

Опишем функцию слияния двух массивов разной длины.

Алгоритм 4 Функция слияния отсортированных массивов

Ввод: Отсортированные массивы A, B.
Вывод: Отсортированный массив C

```
function MERGE(A, B)
    i := 0
    j := 0
    vector C
    while i < len(A) and j < len(B) do
        if A[i] ⩽ B[i] then
            C.push_back(A[i])
            i := i + 1
        else
            C.push_back(B[i])
            j := j + 1
    C := C + A[i : len(A)] + B[j : len(B)]
    return C
```

Доказательство корректности. На входе отсортированные массивы. На каждой итерации цикла выбирается наименьший из еще не выбранных элементов. **[:|||:]**

Время работы. Алгоритм как минимум смотрит на каждый элемент каждого массива, тогда его сложность получается $\Theta(len(A) + len(B))$. **[:|||:]**

Теперь сам алгоритм сортировки слиянием:

Алгоритм 5 Merge Sort

Ввод: Массив A

Вывод: Отсортированный массив C

```
function MERGE_SORT(A)
    if len(A) < 2 then
        return A
    else
        n := len(A)
        A_1 := merge_sort(A[0 : n/2])
        A_2 := merge_sort(A[n/2 : n])
        return merge(A_1, A_2)
```

Замечание 2.1. Сортировку слиянием можно написать разными способами. Например, изначально разбить массив на куски длиной 10, каждую из них отсортировать вставками, а потом уже последовательно слить в один отсортированный массив.

Доказательство корректности. Пока корректна функция *merge*, весь алгоритм корректен, так как вся задача разбивается на меньшие подзадачи, а рекурсия остановится на массивах размера 1. **[:|||:]**

Время работы. Время работы данного алгоритма $T(n) = \Theta(n \cdot log(n))$: У нас $log(n)$ - глубина рекурсии, а на каждом шаге мы пройдемся в сумме по всему массиву. **[:|||:]**

Время работы сортировки слиянием можно записать и по-другому, с помощью *рекуррентной форумулы.* В нашем случае она будет иметь вид

$$T(n) = \begin{cases} c, & n = 2; \\ 2T(\frac{n}{2}) + c \cdot O(n), & n > 2; \end{cases}$$

где $2T(\frac{n}{2})$ - сложность рекурсивных вызовов, $c \cdot O(n)$ - сложность слияния. Для таких соотношений хочется получить правило их раскрытия в явную формулу. О таком преоборазовании говорит *основная теорема о рекуррентных соотношениях.*

Теорема 2.2. *Пусть у нас есть реккурентное соотношение, записанное в виде*

$$T(n) = \begin{cases} c, & n = 2 \\ aT(\frac{n}{b}) + cn^d, & n > 2 \end{cases}$$

Тогда его можно представить в следующем виде:

$$T(n) = \begin{cases} O(n^d \log(n)), & a = b^d \\ O(n^d), & a < b^d \\ O(n^{\log_b(a)}), & a > b^d \end{cases}$$

Эта теорема может раскрыть не все рекуррентные соотношения, поэтому покажем еще несколько способов решения.

Теперь давайте разберем другой рекурсивный алгоритм - *бинарный поиск.* Это алгоритм ищет в отсортированном элементе некоторый элемент и возвращает его индекс.

Алгоритм 6 Binary Search

Ввод: Отсортированный массив A, элемент e

Вывод: Индекс элемента e если он есть в массиве, -1 в противном случае.

function BINARY_SEARCH(A, e)
    $n := len(A)/2$
    if $A[n] == e$ **then**
        return n
    else if $A[n] < e$ **then**
        return binary_search($A[n+1:], e$)
    else
        return binary_search($A[:n], e$)

Словами: смотрим в середину, сравниваем, идем в ту сторону, где элемент заведомо будет, повторяем до нахождения.

Время работы. Формула времени нашего алгоритма:

$$T(n) = \begin{cases} c, & n = 2 \\ T(\frac{n}{2}) + c, & n > 2 \end{cases}$$

По основной теореме

$$a = 1,\ b = 2,\ d = 0$$

$$T(n) = O(log(n))$$

[:|||:]