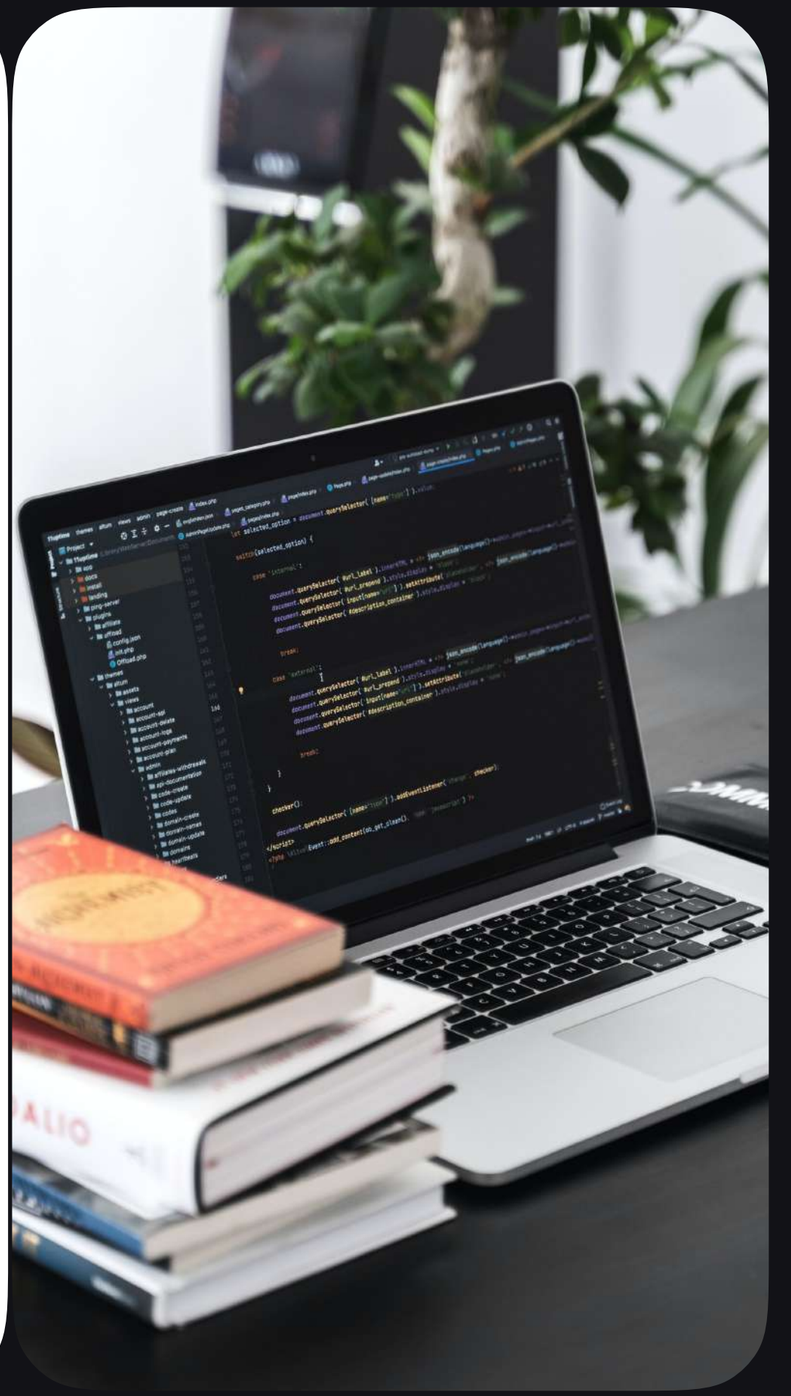
Модуль 2 Занятие 11

Big O (O-нотация). Сложность алгоритмов



Сложность алгоритмов

Сложность алгоритмов поиска

Сложность алгоритмов сортировки

Сложность операций структур данных

Как оценить время работы алгоритма?

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(array)
x = 8

for i in range(n):
    if x == array[i]:
        print(i)
        break
else:
    print(-1)
```

Вспомним программу линейного поиска в списке. Как оценить время работы данной программы?

Сложность алгоритма

Сложность алгоритма — это зависимость времени выполнения (временная сложность) и используемой памяти (пространственная сложность) от размера входных данных.

Временная сложность — это зависимость количества элементарных операций, совершаемых алгоритмом, для решения задачи от размера входных данных. Обычно говорят о времени выполнения в худшем случае, но также можно рассматривать время выполнения в лучшем и в среднем случае.

Пространственная сложность — зависимость используемой памяти от размера входных данных.

Размер входных данных — это то, что алгоритм получает на вход.

Асимптотическая сложность

При оценке времени работы используется понятие **асимптотическая сложность** — сложность при стремлении размера входных данных к бесконечности.

Например: алгоритм выполняет $5n^2 + 3n$ элементарных операций, при размере входных данных n. При увеличении n, на время работы будет больше влиять возведение n в квадрат, чем умножение его на 5 или прибавление 3n. В этом случае говорят, что временная сложность этого алгоритма равна $O(n^2)$ — алгоритм имеет квадратичную сложность.

Обозначение O большое (Big O, O-нотация) используется в математике для сравнения асимптотического поведения функций.

O(n) — линейная сложность

Вернемся к программе линейного поиска в списке:

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(array)
x = 8


for i in range(n):
    if x == array[i]:
        print(i)
        break
else:
    print(-1)
```

Алгоритм линейного поиска имеет сложность O(n), говорят: «Сложность алгоритма — O(n)» или «алгоритм работает за O(n)».

Как уже было сказано ранее, в O-нотации не учитываются константы и коэффициенты. То есть если в алгоритме совершается 2n + 1 операций или n + 2 операции, то его сложность все равно будет O(n).

O(1) — константная сложность

Рассмотрим другую задачу: необходимо вычислить сумму чисел от 1 до n. Задачу можно решить так:

```
n = int(input())
total = 0
for i in range(1, n + 1):
    total = total + i
print(total)
```

Сложность O(n) — время работы алгоритма линейно зависит от входных данных.

или так

```
n = int(input())
print((1 + n) * n // 2)
```

Сложность O(1) — время работы алгоритма не зависит от размера входных данных.

O(logn) — логарифмическая сложность

Вспомним программу двоичного поиска в списке:

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(array)
x = 9

left = 0
right = n
while left < right:
    mid = (left + right) // 2
    if array[mid] == x:
        print(mid)
        break
    elif array[mid] < x:
        left = mid + 1
    else:
        right = mid
    else:
        print(-1)
```

На каждом шаге двоичного поиска исходный массив разделяется на две половины и затем выбирается та половина, в которой может находиться искомый элемент x. Затем такая половина опять разбивается на две половины и поиск продолжается до тех пор, пока искомый элемент не будет найден.

Двоичный поиск работает быстрее линейного и имеет сложность O(logn).

O(logn) — логарифмическая сложность

$\log_a b$ (читается как «логарифм b по основанию a») — это такое число, которое показывает в какую степень надо возвести число a чтобы получить число b.

Примеры: $\log_2 8 = 3$ ($2^3 = 8$) или $\log_5 25 = 2$ ($52 = 2^5$)

На каждом шаге двоичного поиска мы делим наш массив размером n на два массива размером n // 2, то есть, чтобы найти искомый элемент необходимо выполнить операцию деления $\log_2 n$ раз.

В нашем примере в массиве 10 элементов. Т.к. $\log_2 10 \approx 3.32$, а нас интересуют только целые числа, значит в худшем случае необходимо будет выполнить всего 4 операции чтобы найти (или не найти) искомый элемент x.

В асимптотической оценке часто пишут просто O(logn) без указания основания.

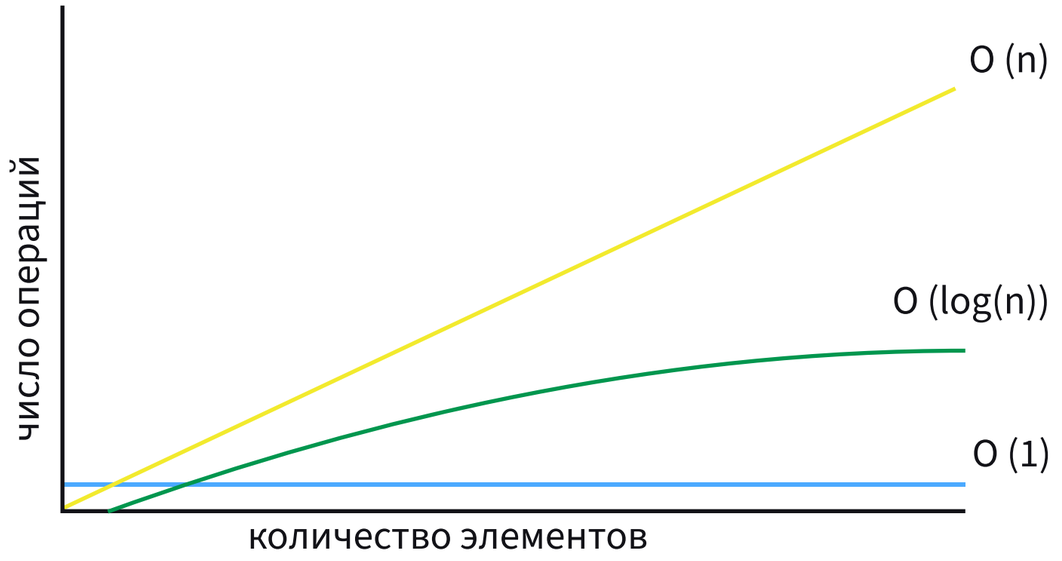
Будем искать x = 8.
array[mid] < x,
продолжаем поиск в правой половине:
left = mid + 1

array[mid] > x,
продолжаем поиск в левой половине:
right = mid

array[mid] = x,
элемент найден

Сложность алгоритмов поиска

Алгоритм	Структура данных	Время в среднем	Время в худшем
Линейный поиск	Массив размером n	O (n)	O (n)
Двоичный поиск	Отсортированный массив размером n	O (logn)	O (logn)



$O(n^2)$ — квадратичная сложность

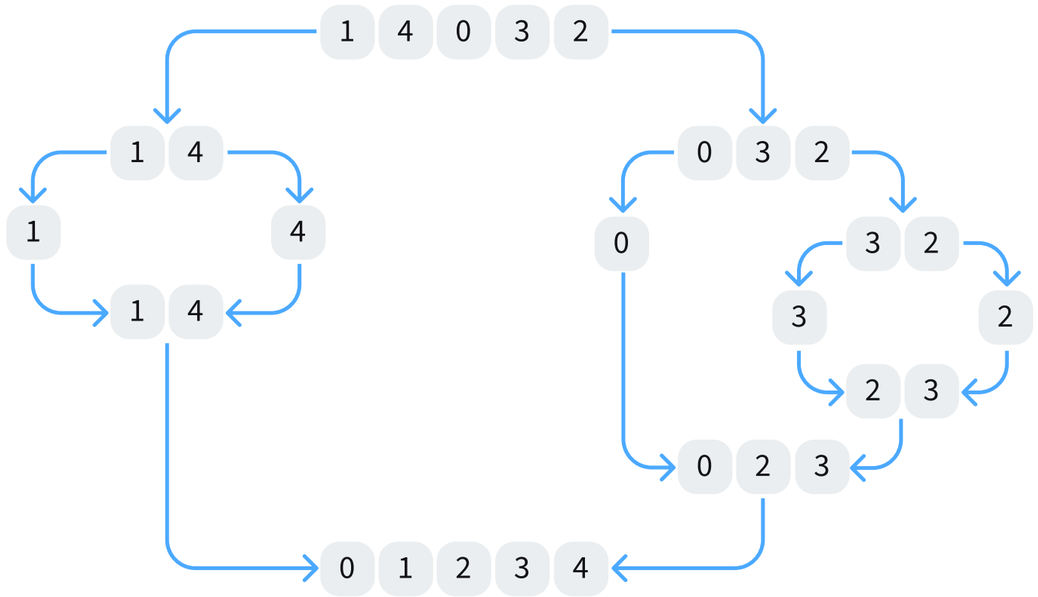
Вспомним программу сортировки пузырьком:

```
array = [1, 4, 0, 3, 2]
n = len(array)

for i in range(n - 1):
    for j in range(n - i - 1):
        if array[j] > array[j + 1]:
            array[j], array[j + 1] = array[j + 1], array[j]

print(array)
```

Алгоритм сортировки пузырьком имеет сложность $O(n^2)$

O(nlogn) — линейно-логарифмическая сложность

Такую сложность, например, имеет сортировка слиянием.

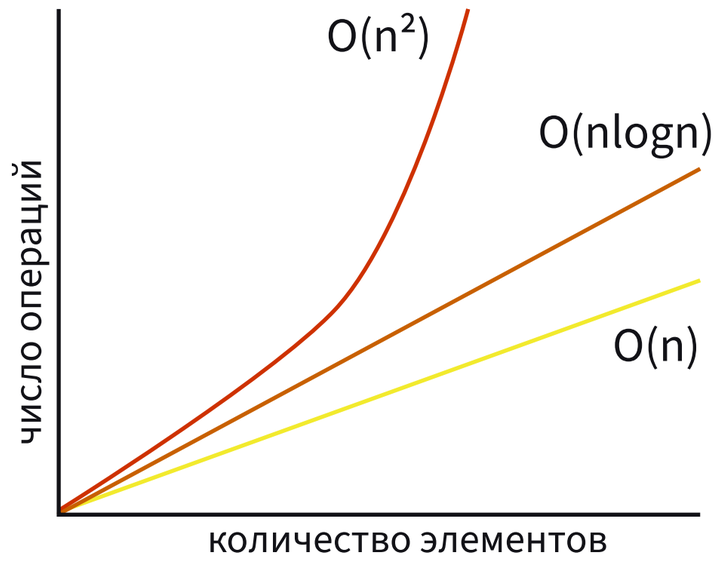


На каждом шаге сортировки слиянием мы делим наш массив размером n на два массива размером n // 2, то есть количество шагов деления будет O(logn). Также, на каждом шаге, мы выполняем операцию слияния — это O(n). Поэтому общее число операций будет O(nlogn).

```python
def merge_sort(array):
    print(array)
    if len(array) == 1:
        return array
    left = merge_sort(array[: len(array) // 2])
    right = merge_sort(array[len(array) // 2:])
    result = []
    l, r = 0, 0
    while l < len(left) and r < len(right):
        if left[l] <= right[r]:
            result.append(left[l])
            l += 1
    else:
        result.append(right[r])
        r += 1
    if l >= len(left):
        result += right[r:]
    else:
        result += left[l:]
    return result


array = [1, 4, 0, 3, 2]
print(merge_sort(array)
```

Сложность алгоритмов сортировки

Алгоритм	Лучшее время	Среднее время	Худшее время	Память
Сортировка пузырьком (bubble sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Сортировка выбором (selection sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Сортировка вставками (insertion sort)	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Сортировка слиянием (merge sort)	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Быстрая сортировка (quick sort)	O(nlogn)	O(nlogn)	$O(n^2)$	O(n)



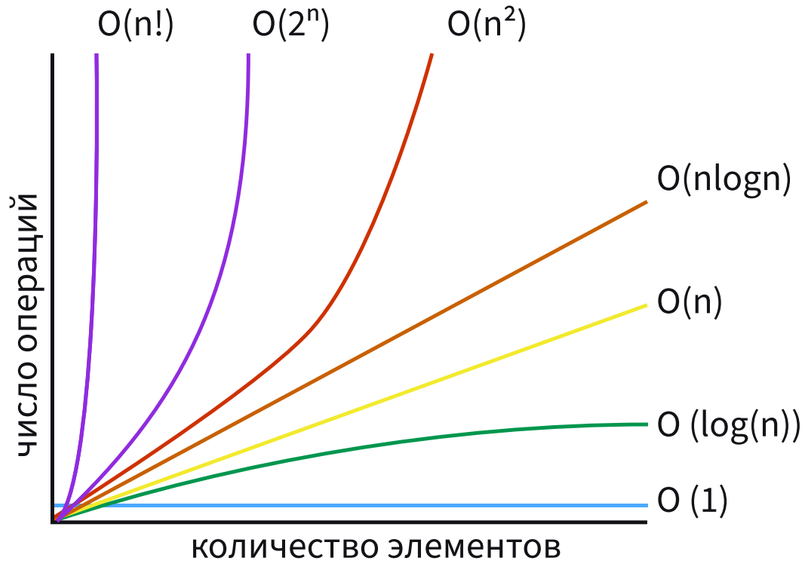
Сложность операций структур данных

Структура данных	Среднее время				Худшее время			
	Обращение по индексу	Поиск по значению	Вставка	Удаление	Обращение по индексу	Поиск по значению	Вставка	Удаление
Динамический массив	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Связный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Хеш-таблица	—	O(1)	O(1)	O(1)	—	O(n)	O(n)	O(n)

Стек и очередь должны позволять быстро добавлять элемент в конец и быстро получать последний элемент (или первый в случае очереди), другие операции они могут либо не поддерживать, либо выполнять неэффективно.

Оценка времени работы

размер n/сложность	O(1)	O(nlogn)	O(n)	$O(n^2)$
10^3	0.000001 секунды	0,00001 секунды	0.001 секунды	1 секунда
10^6	0.000001 секунды	0,00002 секунды	1 секунда	11 дней
10^9	0.000001 секунды	0,00003 секунды	16 минут	?



Если условно предположить, что компьютер выполняет 10^6 элементарных операций в секунду, тогда можно составить таблицу зависимости примерного времени выполнения алгоритма от размера входных данных.

Итоги

- Сложность алгоритма — это зависимость времени выполнения (временная сложность) и используемой памяти (пространственная сложность) от размера входных данных.
- Временная сложность — это зависимость количества элементарных операций, совершаемых алгоритмом, для решения задачи от размера входных данных.
- Пространственная сложность — зависимость используемой памяти от размера входных данных.
- Для оценки сложности используется O-нотация — асимптотическая сложность при стремлении размера входных данных к бесконечности.