МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра «Математическая Кибернетика и Информационные технологии»

Системный анализ и исследование операций

Лабораторная работа №1

Анализ алгоритмов и асимптотическая сложность

Выполнил: Студент

группы БВТ2402

Юдин Владимир

Москва

2026

Цель работы:

1. Изучить понятие временной и пространственной эффективности алгоритмов.
2. Освоить асимптотические оценки сложности (O, o, Ω, ω, Θ).
3. Научиться проводить экспериментальный анализ алгоритмов.
4. Сравнить теоретическую и практическую оценку сложности.

Задание 1. Классификация сложности

Определить асимптотическую сложность, определить порядок роста, указать оценку в виде O(…), обосновать решение для следующих циклов:

```
for i in range(n):
    print(i)
```

Цикл выполняется n раз. На каждой итерации выполняется одна операция вывода, которая считается константной по времени.

Количество элементарных операций пропорционально n.

Порядок роста: линейный.
Асимптотическая оценка: O(n).

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Внешний цикл выполняется n раз. Для каждой итерации внешнего цикла внутренний цикл также выполняется n раз.

Общее количество операций равно $n \cdot n = n^2$.

Порядок роста: квадратичный.
Асимптотическая оценка: $O(n^2)$.

```
i = 1
while i < n:
    i *= 2
```

На каждой итерации значение переменной i увеличивается в 2 раза. Количество итераций определяется условием 2^k ≥ n.

Отсюда $k = \log_2(n)$.

Количество выполнений тела цикла пропорционально log n.

Порядок роста: логарифмический.
Асимптотическая оценка: O(log n).

```
for i in range(n):
    for j in range(i):
        print(i, j)
```

Внешний цикл выполняется n раз. Внутренний цикл на i-й итерации выполняется i раз.

Общее число операций равно:

$0 + 1 + 2 + \ldots + (n - 1)$

Сумма арифметической прогрессии равна:

$n(n - 1) / 2$

При больших n доминирует член $n^2 / 2$, поэтому порядок роста определяется как квадратичный.

Порядок роста: квадратичный.
Асимптотическая оценка: $O(n^2)$.

Задание 2. Работа с нотациями

Определить Верхнюю оценку O(...), Нижнюю оценку Ω(...), Точную оценку Θ(...) для функций:

1. $t(n) = 3n^2 + 7n + 5$
2. $t(n) = 5n\log n + 20n$
3. $t(n) = 100$
4. $t(n) = 2^n + n^3$

1. Функция

$t(n) = 3n^2 + 7n + 5$

При $n \to \infty$ наибольший вклад в рост функции вносит член $3n^2$. Линейный член $7n$ и константа 5 являются асимптотически несущественными по сравнению с n^2.

Следовательно, функция растёт квадратично.

Верхняя оценка: $O(n^2)$
Нижняя оценка: $\Omega(n^2)$
Точная оценка: $\Theta(n^2)$

2. Функция

$t(n) = 5n \log n + 20n$

При больших n функция n log n растёт быстрее, чем n. Поэтому доминирующим членом является 5n log n. Линейная часть 20n становится несущественной.

Порядок роста определяется как n log n.

Верхняя оценка: $O(n \log n)$

Нижняя оценка: $\Omega(n \log n)$

Точная оценка: $\Theta(n \log n)$

3. Функция

$t(n) = 100$

Значение функции не зависит от n. Это постоянная функция.

Порядок роста постоянный.

Верхняя оценка: $O(1)$

Нижняя оценка: $\Omega(1)$

Точная оценка: $\Theta(1)$

4. Функция

$t(n) = 2^n + n^3$

Экспоненциальная функция 2^n растёт значительно быстрее любой полиномиальной функции, включая n^3. При больших n член n^3 становится несущественным по сравнению с 2^n.

Порядок роста определяется экспоненциальной частью.

Верхняя оценка: $O(2^n)$

Нижняя оценка: $\Omega(2^n)$

Точная оценка: $\Theta(2^n)$

Задание 3. Эксперимент с пузырьковой сортировкой

Реализуйте пузырьковую сортировку.

1. Замерьте время работы для размеров массива:

n = [100, 200, 400, 800, 1600, 3200]

1. Каждый эксперимент повторить 5 раз и взять среднее значение.
2. Построить график зависимости времени от n.
3. Построить график зависимости времени от n^2.
4. Сделать вывод — приближается ли поведение к параболе?

Код программы:

```python
import random
import time

def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[i]
    return arr

sizes = [100, 200, 400, 800, 1600, 3200]

for n in sizes:
    arr = [random.randint(0,100000) for _ in range(n)]

    start = time.perf_counter()
    bubble_sort(arr)
    end = time.perf_counter()

    print(f"size - {n}, time = {end - start:.6f} sec")

```

Результаты экспериментов:

1. size - 100, time = 0.000131 sec

   size - 200, time = 0.000482 sec

size - 400, time = 0.002028 sec

size - 800, time = 0.008329 sec

size - 1600, time = 0.029510 sec

size - 3200, time = 0.123648 sec

2. size - 100, time = 0.000104 sec

   size - 200, time = 0.000483 sec

   size - 400, time = 0.001816 sec

   size - 800, time = 0.006443 sec

   size - 1600, time = 0.031749 sec

   size - 3200, time = 0.151313 sec

3. size - 100, time = 0.000136 sec

   size - 200, time = 0.000425 sec

   size - 400, time = 0.001454 sec

   size - 800, time = 0.006633 sec

   size - 1600, time = 0.034052 sec

   size - 3200, time = 0.115664 sec

4. size - 100, time = 0.000145 sec

   size - 200, time = 0.000454 sec

   size - 400, time = 0.001812 sec

   size - 800, time = 0.007042 sec

size - 1600, time = 0.038426 sec

size - 3200, time = 0.125193 sec

5. size - 100, time = 0.000106 sec

size - 200, time = 0.000323 sec

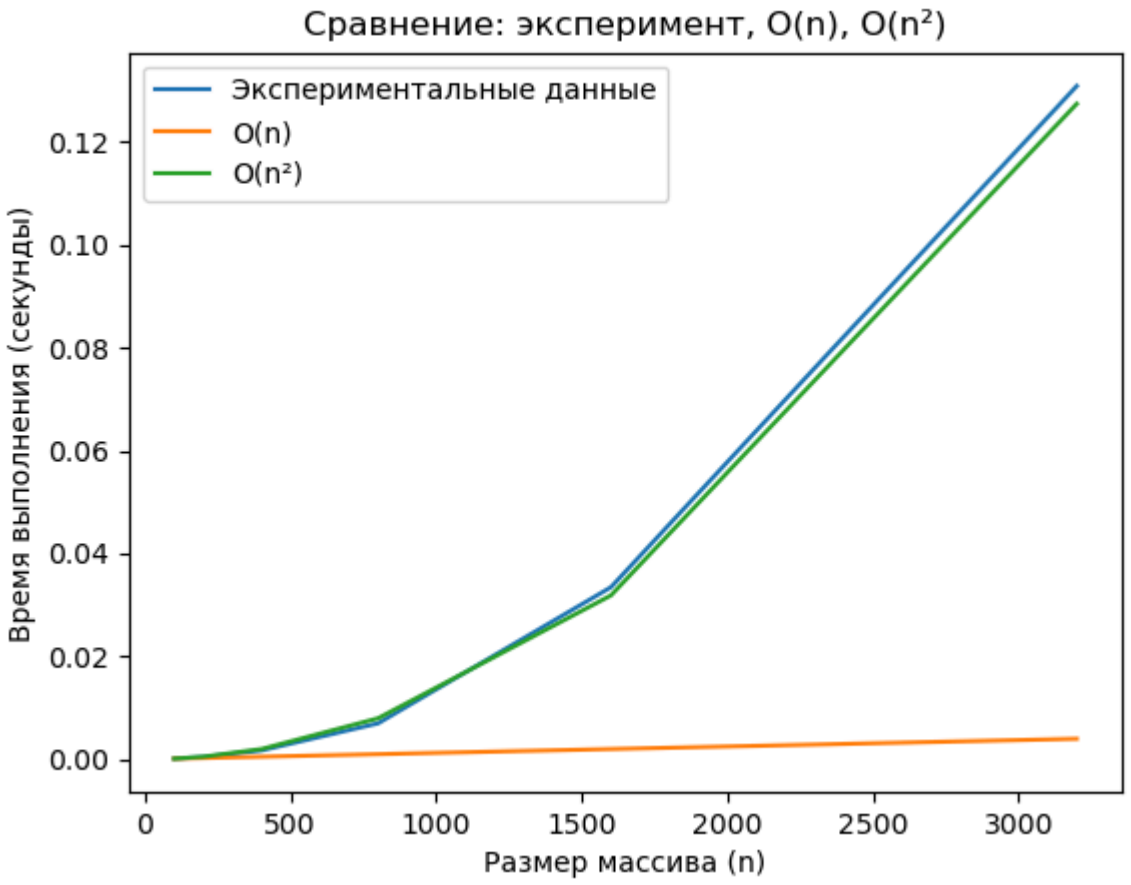
size - 400, time = 0.001495 sec

size - 800, time = 0.006454 sec

size - 1600, time = 0.033466 sec

size - 3200, time = 0.138443 sec

Итоговый график:



Вывод: поведение экспериментального графика приближается к параболе.

Задание 4. Сравнение O(n²) и O(n log n)

1. Реализуйте: пузырьковую сортировку,встроенную сортировку Python (sorted())
2. Замерьте время для: n = [1000, 2000, 5000, 10000]
3. Постройте график сравнения.
4. Ответьте на вопросы:

   При каком n различие становится заметным?

   Почему при больших объёмах данных порядок роста алгоритма важнее постоянных множителей?

Код основной программы:

```python
import random
import time

def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[i]
    return arr

sizes = [1000, 2000, 5000, 10000]

for n in sizes:
    arr = [random.randint(0, 10000) for _ in range(n)]

    # Пузырьковая сортировка
    arr_copy = arr.copy()
    start_time = time.time()
    bubble_sort(arr_copy)
    bubble_time = time.time() - start_time

    # Встроенная сортировка
    arr_copy2 = arr.copy()
    start_time = time.time()
    sorted(arr_copy2)
    built_in_time = time.time() - start_time

    print(f"n={n}: Bubble sort = {bubble_time:.6f}s, Python sorted() = {built_in_time:.6f}s")


```

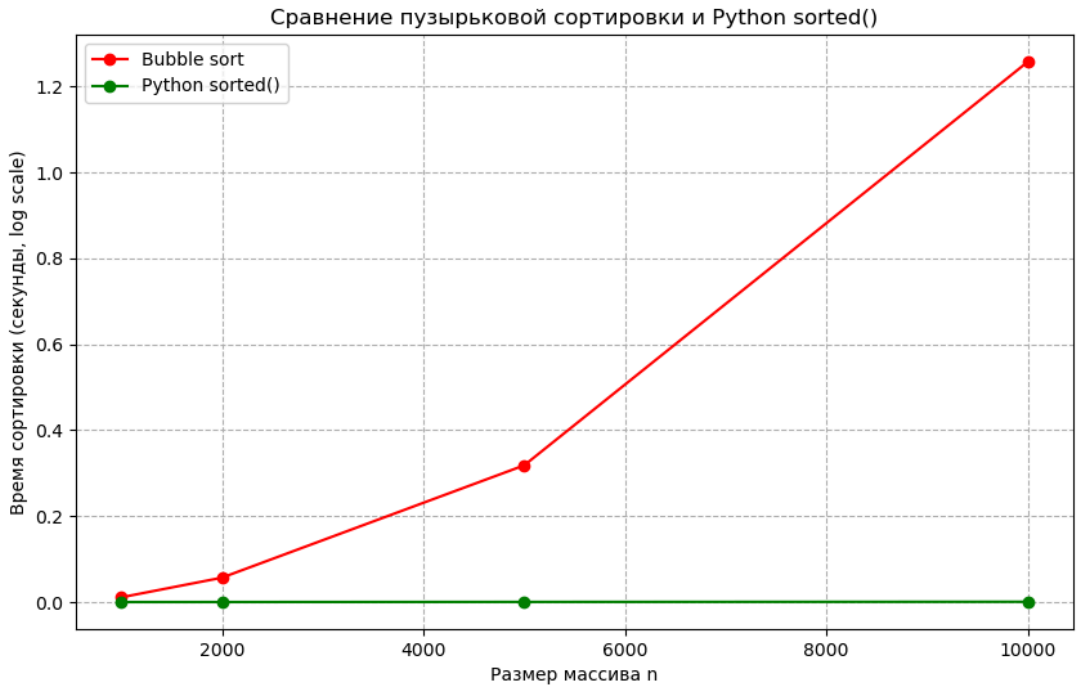
Измеренное время:

n=1000: Bubble sort = 0.011040s, Python sorted() = 0.000059s

n=2000: Bubble sort = 0.056828s, Python sorted() = 0.000125s

n=5000: Bubble sort = 0.318273s, Python sorted() = 0.000371s

n=10000: Bubble sort = 1.257788s, Python sorted() = 0.000726s



Вывод:

Различие в графиках становится заметно уже при n = 2000

При больших объёмах данных важнее порядок роста алгоритма, чем константные множители, потому что время выполнения алгоритма с высокой сложностью (например, $O(n^2)$) быстро растёт и перекрывает любые оптимизации с малыми константами. Алгоритмы с более низкой сложностью ($O(n \log n)$) остаются эффективными даже при больших n.

Задание 5. Оценка времени масштабирования

Алгоритм имеет сложность O(n log n). Для n = 1 000 000 время работы составляет 120 мс. При n = 4 000 000 время работы оценивается как $t_2 = 120 \times (4\,000\,000 \times 22) / (1\,000\,000 \times 19{,}93) \approx 529$ мс, то есть примерно 530 мс.

Контрольные вопросы:

Почему точное указание t(n) затруднительно?

Потому что время работы зависит не только от размера входных данных, но и от констант, накладных расходов, архитектуры процессора, кэшей и конкретных значений данных. Асимптотика учитывает только порядок роста, игнорируя эти детали.

Чем отличается O от Θ?

O(f(n)) задаёт верхнюю границу роста функции (время работы не будет больше этого порядка), Θ(f(n)) задаёт точный порядок роста — функция растёт пропорционально f(n) и сверху, и снизу.

В чём разница между O и o?

O(f(n)) — функция растёт **не быстрее или пропорционально** f(n), o(f(n)) функция растёт строго медленнее f(n), то есть её вклад становится пренебрежимо малым при больших n.

Почему Θ используется реже, чем O?

Потому что точно определить и верхнюю, и нижнюю границу сложнее; на практике чаще важно знать, что алгоритм не хуже некоторого порядка, а точное совпадение не всегда нужно.

Почему при малых n поведение может отличаться от асимптотики?

При малых n константы и накладные расходы (инициализация, кэш, функции библиотек) могут давать сильный эффект, поэтому алгоритмы с худшей асимптотикой иногда работают быстрее, чем «теоретически» лучшие.