**МАИ**

**Лабораторная работа №13**

 «Методы сортировки I»Вариант №14

Факультет робототехнических и интеллектуальных систем

Кафедра «Системы приводов летательных аппаратов»

**Выполнил:**

Студент группы М7О-114БВ-24

Фельдман Лев Борисович

**Проверил:**Доцент Кафедры 702 Козлова Н.М.

Ассистент Кафедры 702 Милославский Я.Г.

Москва 2025

**Цель работы: знакомство с методами сортировки на основе простого выбора, "пузырька" и простой вставки, а также освоение подходов к анализу алгоритмов.**

**1. Пузырьковая сортировка (Bubble Sort)**

**Идея:  
На каждом проходе по массиву последовательно сравниваются соседние элементы, и, если они стоят в неправильном порядке, меняются местами. «Тяжёлые» (большие) элементы постепенно «всплывают» (bubble) к концу массива.**

**Алгоритм (псевдокод):**

**for i = 1 to n–1**

**swapped = false**

**for j = 1 to n–i**

**if A[j] > A[j+1]**

**swap A[j] и A[j+1]**

**swapped = true**

**if not swapped**

**break // массив уже отсортирован**

* **Подробности работы:**
  1. **Первый проход («i = 1») сравнивает пары (1,2), (2,3), …, (n–1,n).**
  2. **После этого на последней позиции окажется наибольший элемент.**
  3. **Второй проход уже не включает последний элемент (он уже на месте), и так далее.**
  4. **Флаг swapped позволяет прекратить работу, если на каком-то проходе не было ни одного обмена → лучший случай.**
* **Сложности:**
  1. **Лучший: O(n) (массив уже отсортирован, один проход без обменов).**
  2. **Худший / средний: O(n²) (совершить ≈n(n–1)/2 сравнений и, возможно, обменов).**
* **Плюсы и минусы:**
  1. **Плюсы: очень простая реализация; потребляет O(1) дополнительной памяти; при хорошо оптимизированной версии — адаптивна.**
  2. **Минусы: большое число обменов (каждый обмен — 3 присваивания); на больших массивах крайне неэффективна.**

**2. Сортировка вставками (Insertion Sort)**

**Идея:  
Массив делится на «отсортированную» и «неотсортированную» части. С каждым шагом из неотсортированной части берётся следующий элемент и вставляется в нужную позицию внутри отсортированной, сдвигая элементы вправо.**

**Алгоритм (псевдокод):**

**for i = 2 to n**

**key = A[i]**

**j = i–1**

**while j ≥ 1 and A[j] > key**

**A[j+1] = A[j] // сдвигаем вправо**

**j = j–1**

**A[j+1] = key // вставляем «ключ» на место**

* **Подробности работы:**
  1. **На первом шаге берётся второй элемент и сравнивается с первым: либо остаётся на месте, либо меняется.**
  2. **На втором шаге «третий» элемент вставляется в нужное место среди первых двух, и т. д.**
  3. **Сдвиги реализуются через присваивания, а не через тривиальные обмены с временной переменной.**
* **Сложности:**
  1. **Лучший: O(n) (массив уже отсортирован — каждый новый элемент сразу «не требует» сдвигов).**
  2. **Худший / средний: O(n²) (каждый элемент может «проваливаться» в начало → суммарно ≈n(n–1)/2 сдвигов и сравнений).**
* **Плюсы и минусы:**
  1. **Плюсы: очень хорошо работает на почти отсортированных данных; число присваиваний меньше, чем в пузырьке; стабильна; O(1) памяти.**
  2. **Минусы: всё ещё квадратичная; для «рандомных» массивов при больших n уступает более продвинутым алгоритмам.**

**3. Сортировка простым выбором (Selection Sort)**

**Идея:  
На i-м шаге ищется минимальный элемент в подмассиве A[i…n] и меняется местами с A[i]. Таким образом на каждой итерации «ставится» один элемент на своё место.**

**Алгоритм (псевдокод):**

**for i = 1 to n–1**

**minIdx = i**

**for j = i+1 to n**

**if A[j] < A[minIdx]**

**minIdx = j**

**if minIdx ≠ i**

**swap A[i] и A[minIdx]**

* **Подробности работы:**
  1. **Первый проход ищет наименьший элемент во всём массиве и ставит его в A[1].**
  2. **Второй проход ищет минимальный в A[2…n], ставит его в A[2], и т. д.**
  3. **В итоге каждый элемент обменивается не более одного раза (n–1 обмен).**
* **Сложности:**
  1. **Лучший / средний / худший: всегда O(n²), так как двойной цикл всегда пробегает все пары (i,j).**
* **Плюсы и минусы:**
  1. **Плюсы: число обменов минимально (≤n–1); простота реализации; O(1) памяти.**
  2. **Минусы: неадаптивен (не учитывает степень упорядоченности входа); нестабилен в классическом варианте; по количеству сравнений и времени проигрывает вставкам на «почти упорядоченных» данных.**

**Сравнение и выбор метода**

|  |  |  |  |
| --- | --- | --- | --- |
| **Метод** | **Bubble Sort** | **Insertion Sort** | **Selection Sort** |
| **Асимптотика (л/с/х)** | **O(n) / O(n²) / O(n²)** | **O(n) / O(n²) / O(n²)** | **O(n²) / O(n²) / O(n²)** |
| **Адаптивность** | **есть (при флаге)** | **есть (интегрально)** | **нет** |
| **Стабильность** | **да** | **да** | **нет (в базовой)** |
| **Число обменов** | **много (Θ(n²))** | **сдвиги (Θ(n²))** | **минимальное (n–1)** |
| **Простота реализации** | **очень простая** | **простая** | **простая** |
| **Практическая скорость** | **самая медленная** | **на «малых» n быстрая** | **средняя (постоянные)** |
| **Когда применять** | **Для ознакомления** | **почти всегда для n ≲ 200** | **если критично число обменов** |

* **Если массив почти отсортирован → Insertion Sort даст наилучшее время (O(n)).**
* **Если важна минимизация обменов (например, при дорогостоящем копировании) → Selection Sort.**
* **Для демонстрации и простоты объяснения часто показывают Bubble Sort, но на практике он редко используется при n > 20–30.**

**Сортировка пузырьком (Числа)**

function [A\_sorted, compCount] = traceBubbleSort(A)

% Пузырьковая сортировка

A\_sorted = A; % копия, чтобы не портить оригинал

n = numel(A\_sorted);

compCount = 0; % счётчик сравнений

% Печать исходного массива

fprintf('Исходный массив: ');

fprintf('%4d', A\_sorted);

fprintf('\n');

% Основной алгоритм

for pass = 1:n-1

fprintf('--------- Проход %02d ---------\n', pass);

swapped = false;

for j = 1:(n-pass)

compCount = compCount + 1;

% Лог сравнения

fprintf('Сравнение %03d (%02d,%02d):', ...

compCount, j, j+1);

fprintf('%4d', A\_sorted);

fprintf('\n');

if A\_sorted(j) > A\_sorted(j+1)

% обмен

[A\_sorted(j), A\_sorted(j+1)] = deal(A\_sorted(j+1), A\_sorted(j));

swapped = true;

% Лог обмена

fprintf('Обмен (%02d,%02d): ', j, j+1);

fprintf('%4d', A\_sorted);

fprintf('\n');

end

end

if ~swapped

% если за этот проход не было обменов — выходим

break;

end

end

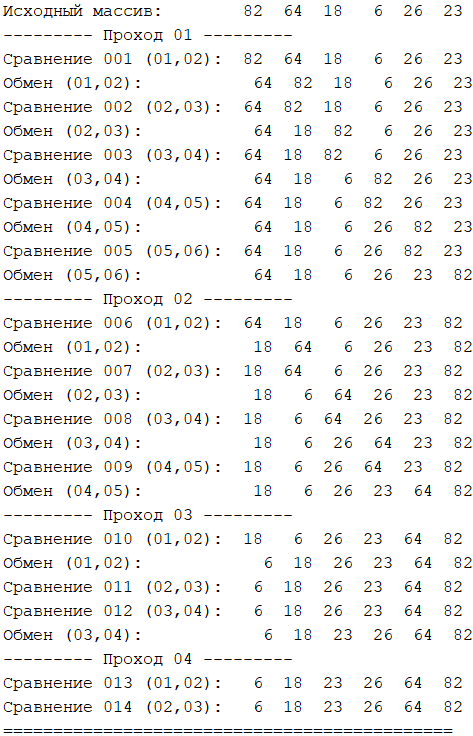
fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>data = [82 64 18 6 26 23];

>>[sortedData, totalComps1] = traceBubbleSort(data);

****

**Сортировка пузырьком (Буквы)**

function [A\_sorted, compCount] = traceLetterBubbleSort(A)

% Пузырьковая сортировка

if isstring(A)

A\_sorted = char(A);

else

A\_sorted = A;

end

n = numel(A\_sorted);

compCount = 0;

% Печать исходного массива

fprintf('Исходный массив: ');

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

% Основной цикл пузырька

for pass = 1:n-1

fprintf('--------- Проход %02d ---------\n', pass);

swapped = false;

for j = 1:(n-pass)

compCount = compCount + 1;

% Лог сравнения

fprintf('Сравнение %03d (%02d,%02d):', ...

compCount, j, j+1);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

% Сравнение символов по их коду (алфавит)

if A\_sorted(j) > A\_sorted(j+1)

% Меняем местами

[A\_sorted(j), A\_sorted(j+1)] = deal(A\_sorted(j+1), A\_sorted(j));

swapped = true;

% Лог обмена

fprintf('Обмен (%02d,%02d): ', j, j+1);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

end

end

if ~swapped

break; % если за проход не было обменов — массив отсортирован

end

end

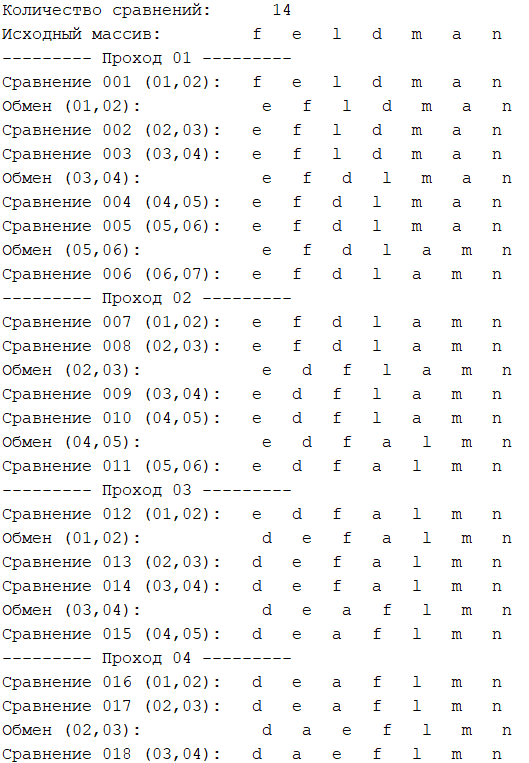
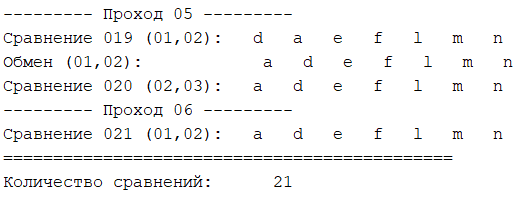
fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>letters = ['f', 'e', 'l', 'd', 'm', 'a', 'n'];

>>[sortedLetters, totalComps2] = traceLetterBubbleSort(letters);

**Сортировка вставками (Числа)**

function [A\_sorted, compCount] = traceInsertionSort(A)

% Сортировка вставками

A\_sorted = A(:)'; % копируем и делаем строку

n = numel(A\_sorted);

compCount = 0; % счётчик сравнений

% Печать исходного массива

fprintf('Исходный массив: ');

fprintf('%4d', A\_sorted);

fprintf('\n');

% Сортировка вставками через обмены соседних элементов

for i = 2:n

fprintf('--------- Шаг %02d ---------\n', i-1);

j = i;

% Вставляем элемент A\_sorted(i) в отсортированную часть [1..i-1]

while j > 1

compCount = compCount + 1;

% Лог сравнения

fprintf('Сравнение %03d (%02d,%02d):', compCount, j, j-1);

fprintf('%4d', A\_sorted);

fprintf('\n');

if A\_sorted(j) < A\_sorted(j-1)

% Меняем местами соседние элементы

[A\_sorted(j), A\_sorted(j-1)] = deal(A\_sorted(j-1), A\_sorted(j));

fprintf('Обмен (%02d,%02d): ', j, j-1);

fprintf('%4d', A\_sorted);

fprintf('\n');

j = j - 1;

else

break; % дальше вставлять не нужно

end

end

end

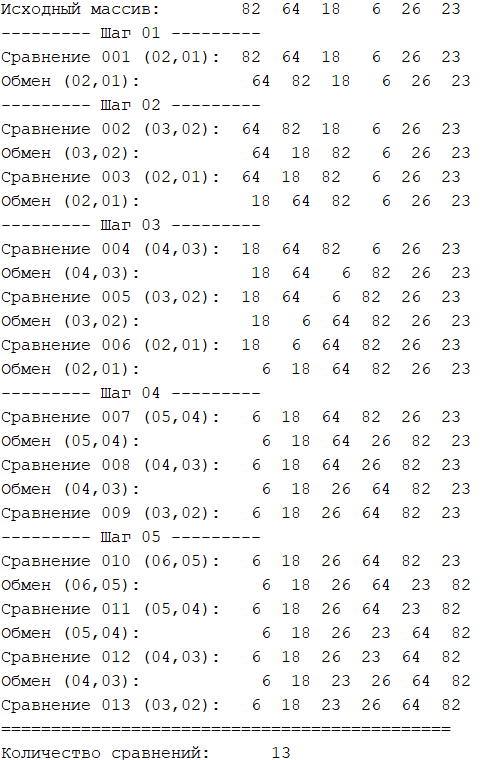
fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>data = [82 64 18 6 26 23];

>>[sortedData, totalComps1] = traceInsertionSort(data);



**Сортировка вставками (Буквы)**

function [A\_sorted, compCount] = traceLetterInsertionSort(A)

% Сортировка вставками

% Приводим к символьному вектору

if isstring(A)

A\_sorted = char(A);

else

A\_sorted = A;

end

n = numel(A\_sorted);

compCount = 0;

% Печать исходного массива

fprintf('Исходный массив: ');

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

% Сортировка вставками через обмены смежных элементов

for i = 2:n

fprintf('--------- Шаг %02d ---------\n', i-1);

j = i;

% На каждой итерации сравниваем и, при необходимости,

% обмениваем A\_sorted(j) и A\_sorted(j-1) до позиции вставки

while j > 1

compCount = compCount + 1;

% Лог сравнения

fprintf('Сравнение %03d (%02d,%02d):', compCount, j, j-1);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

if A\_sorted(j) < A\_sorted(j-1)

% Обмен смежных элементов

[A\_sorted(j), A\_sorted(j-1)] = deal(A\_sorted(j-1), A\_sorted(j));

fprintf('Обмен (%02d,%02d): ', j, j-1);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

j = j - 1;

else

% Если не требует обмена — выходим из цикла вставки

break;

end

end

end

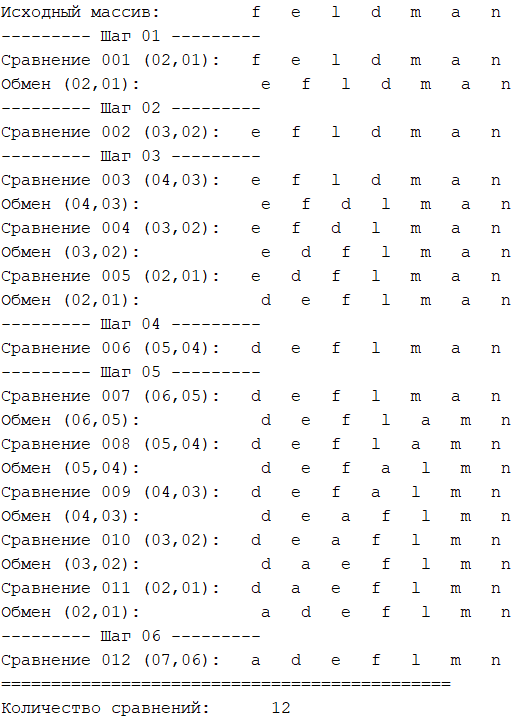
fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>letters = ['f', 'e', 'l', 'd', 'm', 'a', 'n'];

>>[sortedLetters, totalComps2] = traceLetterInsertionSort(letters);



**Сортировка выбором (Числа)**

function [A\_sorted, compCount] = traceSelectionSort(A)

% Сортировка простым выбором

A\_sorted = A(:)'; % копия в виде строки

n = numel(A\_sorted);

compCount = 0;

% Печать исходного массива

fprintf('Исходный массив: ');

fprintf('%4d', A\_sorted);

fprintf('\n');

for i = 1:n-1

fprintf('--------- Шаг %02d ---------\n', i);

minIdx = i;

% Поиск минимального в сегменте i..n

for j = i+1:n

compCount = compCount + 1;

fprintf('Сравнение %03d (%02d,%02d):', compCount, j, minIdx);

fprintf('%4d', A\_sorted);

fprintf('\n');

if A\_sorted(j) < A\_sorted(minIdx)

minIdx = j;

end

end

% Обмен текущего элемента с найденным минимумом

if minIdx ~= i

[A\_sorted(i), A\_sorted(minIdx)] = deal(A\_sorted(minIdx), A\_sorted(i));

fprintf('Обмен (%02d,%02d): ', i, minIdx);

fprintf('%4d', A\_sorted);

fprintf('\n');

end

end

fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>data = [10, 8, 5, 6, 3];

>>[sortedData, compsNum] = traceSelectionSort(data);



**Сортировка выбором (Буквы)**

function [A\_sorted, compCount] = traceLetterSelectionSort(A)

% Сортировка простым выбором

if isstring(A)

A\_sorted = char(A);

else

A\_sorted = A(:)'; % убедимся, что вектор-строка

end

n = numel(A\_sorted);

compCount = 0;

% Печать исходного массива

fprintf('Исходный массив: ');

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

for i = 1:n-1

fprintf('--------- Шаг %02d ---------\n', i);

minIdx = i;

% Поиск «лексикографического минимума»

for j = i+1:n

compCount = compCount + 1;

fprintf('Сравнение %03d (%02d,%02d):', compCount, j, minIdx);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

if A\_sorted(j) < A\_sorted(minIdx)

minIdx = j;

end

end

% Обмен, если минимум не на месте

if minIdx ~= i

[A\_sorted(i), A\_sorted(minIdx)] = deal(A\_sorted(minIdx), A\_sorted(i));

fprintf('Обмен (%02d,%02d): ', i, minIdx);

for k = 1:n

fprintf('%4c', A\_sorted(k));

end

fprintf('\n');

end

end

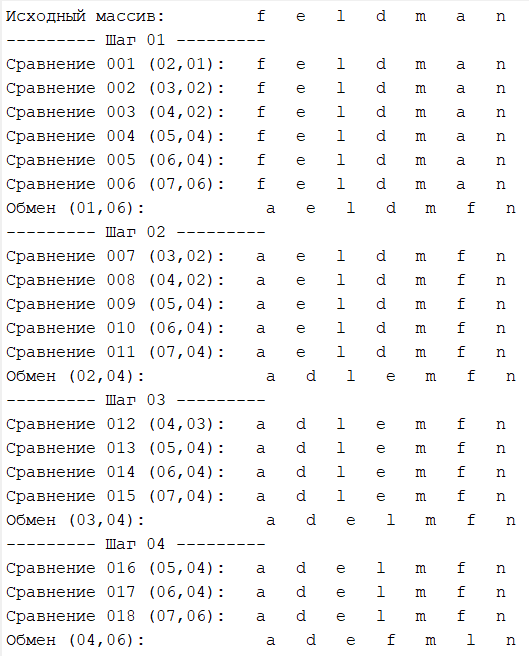
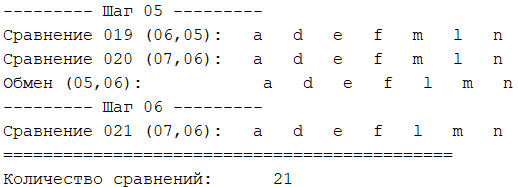
fprintf('=============================================\n');

fprintf('Количество сравнений: %7d\n', compCount);

end

>>letters = ['d','a','c','b','e'];

>>[sortedLetters, compsLet] = traceLetterSelectionSort(letters);

**Вывод**

В ходе проделанной работы мы подробно рассмотрели и реализовали «трассирующие» версии трёх базовых алгоритмов сортировки — пузырьком, вставками и простым выбором — как для чисел, так и для букв. Все они имеют квадратичную асимптотику в среднем и худшем случаях, но существенно различаются по практическим характеристикам:

1. **Простота понимания и обучения**  
   • **Bubble Sort** – интуитивно-понятен, но практически неприменим в реальных задачах из-за избыточных обменов.  
   • **Insertion Sort** и **Selection Sort** чуть сложнее концептуально, но тоже остаются очень наглядными.
2. **Адаптивность (работа с почти упорядоченными данными)**  
   • **Insertion Sort** показывает наилучшие результаты на почти-отсортированных массивах (O(n)).  
   • **Bubble Sort** может остановиться досрочно, но всё равно делает «лишние» сравнения.  
   • **Selection Sort** вовсе не адаптивен.
3. **Число операций обмена/сдвигов**  
   • **Selection Sort** совершает минимальное количество обменов (≤ n–1), но требует полной проверки всех элементов каждый проход.  
   • **Insertion Sort** выполняет сдвиги подряд идущих элементов, что в ряде случаев эффективнее множественных обменов.  
   • **Bubble Sort** использует многократные обмены соседних элементов (каждый — три присваивания).
4. **Стабильность**  
   • Сохраняют порядок равных элементов: **Bubble Sort** и **Insertion Sort**.  
   • **Selection Sort** по умолчанию нестабилен.
5. **Применимость на практике**
   * Для **малых** и **почти отсортированных** массивов (n ≲ 200) предпочтителен **Insertion Sort**.
   * Если важна **минимизация количества обменов** (например, при дорогостоящем копировании данных) — выбирают **Selection Sort**.
   * **Bubble Sort** чаще служит учебным примером, но редко встречается в боевом коде.

В итоге, хотя все три метода обладают одинаковой теоретической сложностью O(n²) в среднем и худшем случаях, на практике выбор зависит от конкретных требований: степени предварительного упорядочения, стоимости операций обмена и необходимости стабильности. При разработке реальных приложений для большинства задач с небольшими объёмами данных наиболее универсальным и эффективным оказывается **Insertion Sort**.