Задача сортировки. Виды алгоритмов сортировки: устойчивые алгоритмы, сортировки сравнением, сортировки на месте (in-place). Сортировка вставками. Сортировка слиянием. Быстрая сортировка. Пирамидальная сортировка. Поразрядная сортировка.

Задача сортировки:

Дана последовательность из n ключей $a_1, a_2, \ldots, a_n$

Требуется упорядочить ключи по не убыванию или по не возрастанию — найти перестановку $(i_1, i_2, \ldots, i_n)$ ключей

По не убыванию *(non-decreasing order)*:

$$a_{i_1} \le a_{i_2} \le \ldots \le a_{i_n}$$

По не возрастанию *(non-increasing order)*:

$$a_{i_1} \ge a_{i_2} \ge \ldots \ge a_{i_n}$$

Виды алгоритмов сортировки

Алгоритм сортировки, не меняющий относительный порядок следования равных ключей, называется **устойчивым** *(stable)*:

(76, Лис), (34, Тигр), (29, Единорог), **(76, Кот),** (95, Дракон)

- **Неустойчивая сортировка:**
  (95, Дракон), **(76, Кот), (76, Лис)** (34, Тигр), (29, Единорог)
  (порядок не соблюден)
- **Устойчивая сортировка:**
  (95, Дракон), **(76, Лис), (76, Кот),** (34, Тигр), (29, Единорог)
  (порядок соблюден)

Внутренние методы сортировки *(Internal sort)* – сортируемые элементы полностью размещены в оперативной памяти компьютера.

Внешняя сортировка *(External sort)* – элементы размещены на внешней памяти *(жесткий диск, USB-флеш)*

Алгоритм сортировки не использующий дополнительной памяти *(кроме сортируемого массива)* называется алгоритмом **сортировки на месте** *(in-place sort)*

Алгоритмы, **основанные на сравнениях** *(comparison sort):*

- Insertion Sort
- Bubble Sort
- Selection Sort
- Shell Sort
- Quick Sort
- Merge Sort
- Heap Sort и другие

Алгоритмы, **не основанные на сравнениях:**

- Counting Sort
- Radix Sort
  (используют структуру ключа)

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Свойства
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)	Устойчивая, на месте, online
Сортировка выбором (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Устойчивость зависит от реализации, на месте
Быстрая сортировка (Quick Sort)	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	Неустойчивая
Сортировка слиянием (Merge Sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Устойчивая
Пирамидальная сортировка (Heap Sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	O(1)	Неустойчивая, на месте
Сортировка подсчетом (Counting Sort)	$O(k + n)$	$O(k + n)$	$O(k + n)$	$O(k + n)$	Устойчивая, целочисленная

7

Сортировка вставками *(Insertion sort)*

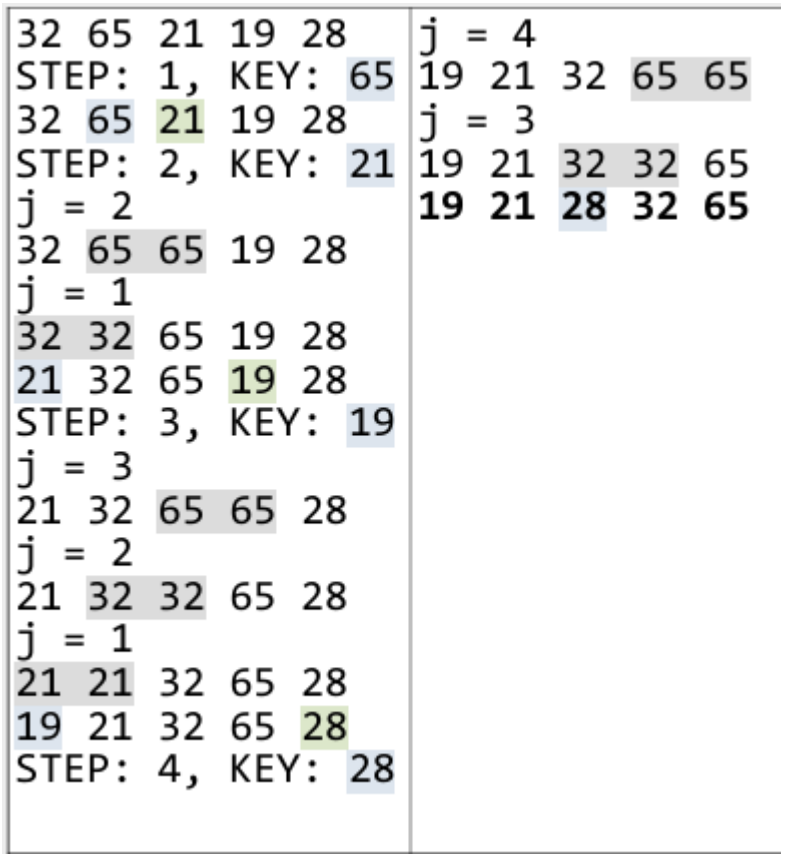
```
void InsertionSort(int *A, int n) {
  for (int i = 1; i < n; i++) {
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key) {
      A[j + 1] = A[j];
      j = j - 1;
    }
    A[j + 1] = key;
  }
}
```

- Двигаемся по массиву слева направо: от 2-го до n-го элемента
- На шаге i имеем упорядоченный подмассив A[1...i – 1] и элемент A[i], который необходимо вставить в этот подмассив

- В худшем случае цикл while всегда доходит до первого элемента массива – на вход поступил массив, упорядоченный по убыванию
- Для вставки элемента A[i] на свое место требуется i - 1 итерация цикла while
- На каждой итерации выполняем c действий
- Учитывая, что необходимо найти позиции для n - 1 элемента, время T(n) выполнения алгоритма в худшем случае равно

$$T(n)=\sum_{i=2}^{n} c(i-1)=c+2c+...+(i-1)c+...+(n-1)c=\frac{cn(n-1)}{2}=\Theta(n^2)$$

Пример сортировки 5 элементов для наглядности:

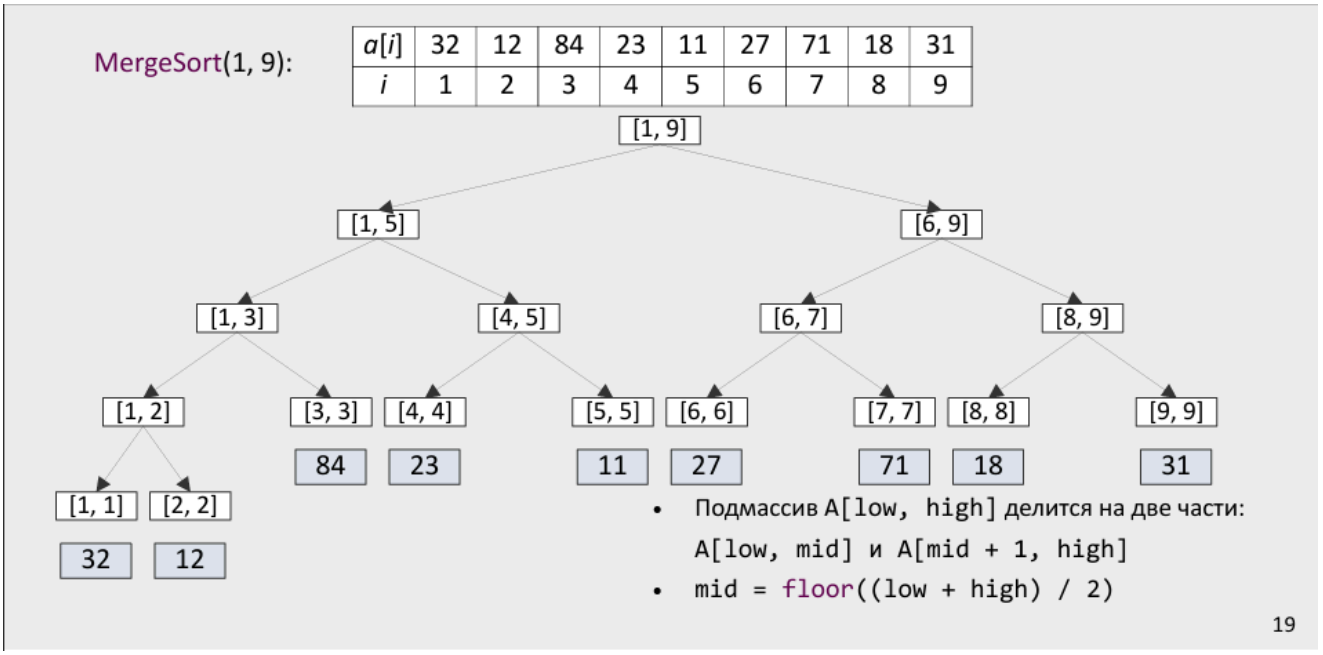


Сортировка слиянием. *(Merge Sort)*

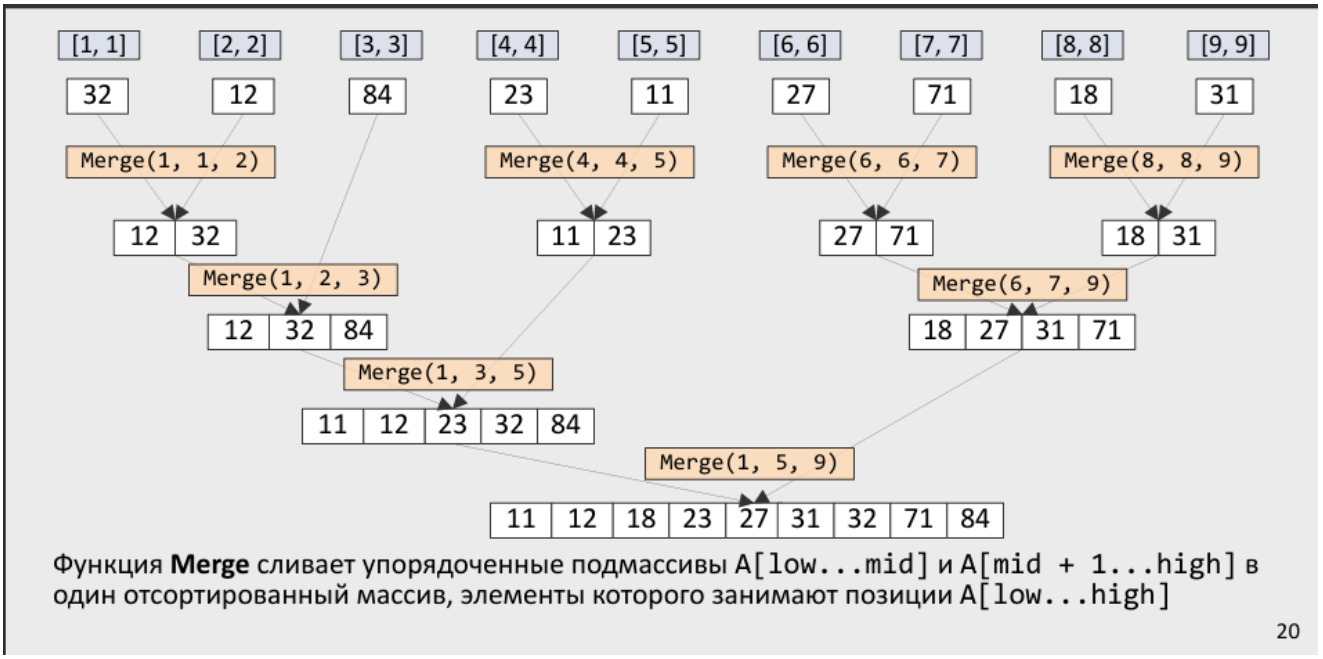
- ***Сортировка слиянием** *(merge sort)* — асимптотически оптимальный алгоритм сортировки сравнением, основанный на методе декомпозиции *(«разделяй и властвуй», decomposition)*.
- Требуется упорядочить заданный массив A[1..n] по не убыванию (non-decreasing order) так, чтобы A[1] ≤ A[2] ≤ … ≤ A[n]
- Алгоритм включает две фазы:

1. **Разделение** (partition) — рекурсивное разбиение массива на меньшие подмассивы, их сортировка.
2. **Слияние** *(merge)* — объединение упорядоченных массивов в один.

Фаза разделения



Фаза слияния




```
void Merge(int A[], int low, int mid, int high) {

  int B[high];

  for (int i = low; i < high; i++)

    B[i] = A[i]; /* Копируем массив A */
```

```
  int l = low; /* Начало левого подмассива */
  int r = mid + 1; /* Начало правого подмассива */
  int i = low;
  while (l <= mid && r <= high) {
    if (B[l] <= B[r]) {
      A[i] = B[l];
      l++;
    }
    else {
      A[i] = B[r];
      r++;
    }
    i++;
  }
  /* Копируем остатки подмассивов */
  while (l <= mid) {
    A[i] = B[l];
    l++;
    i++;
  }
  while (r <= high) {
    A[i] = B[r];
    r++;
    i++;
  }
```

```
}
```

- Функция Merge требует порядка **Θ(n)** ячеек памяти для хранения копии B сортируемого массива
- Сравнение и перенос элементов из массива **B** в массив **A** требует **Θ(n)**.

```
void MergeSort(int A[], int low, int high) {

  if (low < high) {

    int mid = (low + high) / 2;

    MergeSort(A, low, mid);

    MergeSort(A, mid + 1, high);

    Merge(A,low,mid,high);

  }

}
```

- Сортируемый массив A[low...high] разделяется (partition) на две максимально равные по длине части .
- Левая часть содержит [n / 2] элементов, правая — [n / 2] элементов.
- Подмассивы рекурсивно сортируются.

Быстрая сортировка *(Quick Sort)*

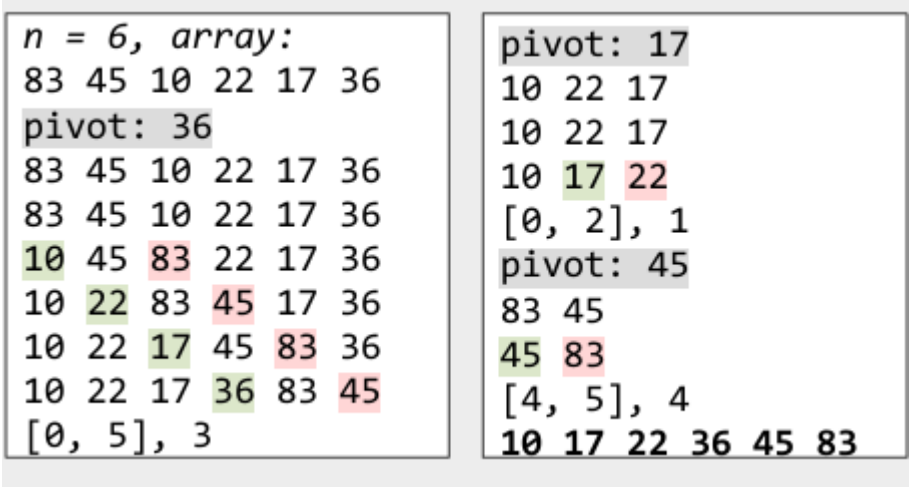
1. Из элементов A[1], A[2], …, A[n] выбирается опорный элемент *(pivot element)*

- A. Опорный элемент желательно выбирать так, чтобы его значение было близко к среднему значению всех элементов массива
- B. Вопрос о выборе опорного элемента открыт (первый, последний, средний из трех, случайный, …)

2. Массив разбивается на две части: элементы массива переставляются так, чтобы элементы, расположенные левее опорного, были не больше (≤), а расположенные правее – не меньше него (≥). На этом шаге определяется граница дальнейшего разбиения массива

- Шаги 1 и 2 рекурсивно повторяются для левой и правой частей

```c
int Partititon(int *array, size_t low, size_t high) {
  int pivot = array[high];
  int j = low;
  for (int i = low; i < high; i++) {
    if (array[i] <= pivot) {
      Swap(&array[i], &array[j]);
          j += 1;
    }
  }
  Swap(&array[j], &array[high]);
  return j;
}
void QuickSort(int *array, size_t low, size_t high) {
  if (low < high) {
    int p = Partititon(array, low, high);
    QuickSort(array, low, p - 1);
    QuickSort(array,p + 1,high);
  }
}
```



(Пример работы)

Пирамидальная сортировка *(Heap Sort)*

```c
// Возвращает индекс левого потомка для узла i (индексация с 0)
int leftF(int i) {
    return 2 * i + 1; // Формула для левого потомка при индексации с 0
}

// Возвращает индекс правого потомка для узла i (индексация с 0)
int rightF(int i) {
    return 2 * i + 2; // Формула для правого потомка при индексации с 0
}

// Функция "просеивания вниз" (heapify) для поддержки свойств кучи
// arr - массив, представляющий кучу
// i - индекс узла, с которого начинается просеивание
// n - текущий размер кучи
void heapify_down(uint32_t arr[], int i, int n) {
    int left = leftF(i);   // Индекс левого потомка
    int right = rightF(i);  // Индекс правого потомка
    int largest = i;       // Инициализируем наибольший элемент как текущий
узел

    // Если левый потомок существует и больше текущего элемента
    if (left < n && (arr[left] > arr[largest])) {
        largest = left;
    }

    // Если правый потомок существует и больше текущего наибольшего
    if (right < n && (arr[right] > arr[largest])) {
        largest = right;
    }

    // Если наибольший элемент - не текущий узел
```

```
    if (largest != i) {
        swap((arr + i), (arr + largest));  // Меняем местами
        heapify_down(arr, largest, n);     // Рекурсивно просеиваем дальше
    }
}

// Построение max-кучи из неупорядоченного массива
// arr - входной массив
// n - размер массива
void build_max_heap(uint32_t arr[], int n) {
    // Начинаем с последнего нелистового узла (родителя последнего элемента)
    // и идем в обратном порядке до корня
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify_down(arr, i, n);
    }
}

// Основная функция пирамидальной сортировки
// arr - сортируемый массив
// n - размер массива
void heap_sort(uint32_t arr[], int n) {
    // Сначала строим max-кучу из массива
    build_max_heap(arr, n);

    // Последовательно извлекаем элементы из кучи
    for (int i = n - 1; i > 0; i--) {
        // Перемещаем текущий максимум (корень) в конец
        swap(arr, arr + i);
        // Восстанавливаем свойства кучи для уменьшенной кучи
        heapify_down(arr, 0, i);
    }
}
```

Принцип работы:

1. **Построение max-кучи** из массива (корень — наибольший элемент).
2. **Извлечение максимума** (перемещение `A[1]` в конец).
3. **Уменьшение кучи** на 1 элемент и восстановление её свойств (`HEAPIFYDOWN`).
4. **Повторение** шагов 2–3, пока куча не опустеет.

Поразрядная сортировка *(Radix Sort).*

```
// Функция для нахождения максимального элемента в массиве
uint32_t get_max(uint32_t* arr, size_t n) {
    uint32_t max = arr[0]; // Предполагаем, что первый элемент -
```

```
максимальный
    for (size_t i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i]; // Обновляем максимум, если находим больший
элемент
        }
    }
    return max;
}

// Функция сортировки подсчетом для определенного разряда (exp)
void counting_sort(uint32_t* arr, size_t n, uint32_t exp) {
    uint32_t* output = (uint32_t*)malloc(sizeof(uint32_t) * n); // Временный
массив для результата
    uint32_t count[10] = {0}; // Массив счетчиков для цифр 0-9

    // 1. Подсчет количества каждой цифры в текущем разряде
    for (size_t i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++; // Извлекаем цифру и увеличиваем
счетчик
    }

    // 2. Преобразование count в массив префиксных сумм
    // Теперь count[i] содержит позицию последнего элемента с цифрой i
    for (size_t i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // 3. Построение выходного массива (обратный проход для стабильности)
    for (size_t i = n - 1; i < n; i--) { // ОШИБКА: должно быть i >= 0, но
из-за беззнакового типа нужно особое условие
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // 4. Копирование отсортированного массива обратно в исходный
    for (size_t i = 0; i < n; i++) {
        arr[i] = output[i];
    }

    free(output); // Освобождаем временную память
}

// Основная функция поразрядной сортировки (Radix Sort)
void radix_sort(uint32_t* arr, size_t n) {
    // Находим максимальное число, чтобы узнать количество разрядов
    uint32_t max = get_max(arr, n);

    // Применяем counting_sort для каждого разряда, начиная с младшего
    for (uint32_t exp = 1; max / exp > 0; exp *= 10) {
```

```
        counting_sort(arr, n, exp);
    }
}
```

- **LSD** *(least significant digit, начиная с последней цифры)* может быть использована для стандартной сортировки чисел.
- **MSD** *(most significant digit, начиная с первой цифры)* может быть использована для сортировки строк .
- Можно сортировать по основанию 2, 8, 10, 16…
- Не использует операцию сравнения
- В данном случае для сортировки разрядов используется **CountingSort**.

```
n = 7, maxnum = 564, array:
12 564 43 2 64 536 263

564
CURRENT: 4
COUNTING:
12 2 43 263 564 64 536
CURRENT: 6
COUNTING:
2 12 536 43 263 564 64
CURRENT: 5
COUNTING:
2 12 43 64 263 536 564
```

(Пример работы)