Отчет

1. Сортировка пузырьком

Идея: Попарно сравниваются соседние элементы и меняются местами, если они стоят в неправильном порядке. Проход по массиву повторяется до тех пор, пока массив не будет отсортирован.
Сложность: O(n^2) в худшем и среднем случае.
Особенности: Простая в реализации, но очень медленная на больших данных. Используется только в учебных целях.

```cpp
// 1) Сортировка пузырьком
template<typename T>
void BubbleSort(T* arr, int n) {
    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - 1 - i; ++j) {
            if (arr[j + 1] < arr[j]) {
                SwapElems(arr[j], arr[j + 1]);

            }
        }
    }
}
```

Как работает?
Начальный массив: [4 2 1 5]
1 шаг: [2 4 1 5]
2 шаг: [2 1 4 5]
3 шаг: [2 1 4 5] ничего не изменилось, т.к 4 меньше 5
4 шаг: [1 2 4 5]

2. Шейкерная сортировка

Идея: Улучшение пузырьковой сортировки. Проходы выполняются сначала слева направо, затем справа налево.
Сложность: O(n^2), но на практике немного быстрее пузырьковой.
Особенности: Немного эффективнее пузырьковой, но всё ещё медленная.

```cpp
// 2) Шейкерная сортировка
template<typename T>
void ShakerSort(T* arr, int n) {
    int left = 0, right = n - 1;
    while (left < right) {
        for (int i = left; i < right; ++i) {
            if (arr[i + 1] < arr[i]) {
                SwapElems(arr[i], arr[i + 1]);
            }
        }
        --right;
        for (int i = right; i > left; --i) {
            if (arr[i] < arr[i - 1]) {
                SwapElems(arr[i], arr[i - 1]);
            }
        }
        ++left;
    }
}
```

Как работает?
Массив [5, 3, 8, 4, 2]
Первый проход (слева направо):
Сравниваем 5 и 3, меняем: [3, 5, 8, 4, 2]
Сравниваем 5 и 8, не меняем: [3, 5, 8, 4, 2]
Сравниваем 8 и 4, меняем: [3, 5, 4, 8, 2]
Сравниваем 8 и 2, меняем: [3, 5, 4, 2, 8]
Правую границу уменьшаем: теперь массив выглядит как [3, 5, 4, 2, 8].
Второй проход (справа налево):
Сравниваем 2 и 4, меняем: [3, 5, 2, 4, 8]
Сравниваем 2 и 5, меняем: [3, 2, 5, 4, 8]
Сравниваем 2 и 3, меняем: [2, 3, 5, 4, 8]
Левую границу увеличиваем: массив теперь [2, 3, 5, 4, 8].
Третий проход (слева направо):
Сравниваем 2 и 3, не меняем: [2, 3, 5, 4, 8]
Сравниваем 3 и 5, не меняем: [2, 3, 5, 4, 8]
Сравниваем 5 и 4, меняем: [2, 3, 4, 5, 8]
Правую границу уменьшаем: массив [2, 3, 4, 5, 8].
Четвертый проход (справа налево):
Сравниваем 4 и 5, не меняем: [2, 3, 4, 5, 8]
Левую границу увеличиваем: теперь границы пересеклись.

3. Сортировка расческой

Идея: Улучшение пузырьковой сортировки. Изначально сравниваются элементы на большом расстоянии (шаг вычисляется через "фактор уменьшения"), благодаря чему большие элементы становятся в конец массива(«устранить» элементы с небольшими значения в конце массива, которые замедляют работу алгоритма).
Сложность: В среднем O(n log n), в худшем случае O(n^2).
Особенности: Устраняет главный недостаток пузырьковой сортировки — "черепах" (мелкие элементы в конце массива), поэтому работает намного быстрее.

```cpp
// 3. Расчёска (Comb sort)
template<typename T>
void CombSort(T* arr, int n) {
    const double temp = 1.2473309;
    int step = n;
    bool flag = false;
    while (!flag) {
        step = static_cast<int>(step / temp);
        if (step <= 1) {
            step = 1;
            flag = true;
        }
        for (int i = 0; i + step < n; ++i) {
            if (arr[i + step] < arr[i]) {
                SwapElems(arr[i], arr[i + step]);
                flag = false;
            }
        }
    }
}
```

4. Сортировка вставками

Идея: Массив условно делится на отсортированную и неотсортированную части. Элементы из неотсортированной части по одному вставляются на правильную позицию в отсортированной.
Сложность: O(n^2) в худшем случае, O(n) — в лучшем (если массив уже отсортирован).
Особенности: Эффективна на небольших массивах и практически отсортированных данных.

```cpp
// 4. Сортировка вставками
template<typename T>
void InsertionSort(T* arr, int n) {
    for (int i = 1; i < n; ++i) {
        T key = arr[i];
        int j = i - 1;
        while (j >= 0 && key < arr[j]) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Пример:
5 2 4 3 1
2 5 4 3 1
2 4 5 3 1
2 3 4 5 1
1 2 3 4 5

5. Сортировка выбором

```cpp
// 5. Сортировка выбором
template<typename T>
void SelectionSort(T* arr, int n) {
    for (int i = 0; i < n - 1; ++i) {
        int indexmin = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[indexmin]) {
                indexmin = j;
            }
        }
        if (indexmin != i) {
            SwapElems(arr[i], arr[indexmin]);
        }
    }
}
```

Идея: Массив делится на отсортированную и неотсортированную части. На каждом шаге в неотсортированной части ищется минимальный (или максимальный) элемент и помещается в конец отсортированной части.
Сложность: $O(n^2)$ для всех случаев.
Особенности: Простая, но неэффективная. Главное преимущество — минимум swap'ов (обменов) элементов (всего $O(n)$).

Пример:
4 3 1 5 2
1 3 4 5 2
1 2 4 5 3
1 2 3 4 5

6. Быстрая сортировка

Идея:
1. Выбирается **опорный элемент** (pivot).
2. Массив перераспределяется так, чтобы элементы меньше опорного оказались слева, а больше — справа.
3. Рекурсивно применяются первые два шага к двум подмассивам.

Сложность: $O(n \log n)$ в среднем, $O(n^2)$ в худшем (при неудачном выборе опорного элемента).
Особенности: Один из самых быстрых и распространённых алгоритмов на практике.

```
// 6. Быстрая сортировка
template<typename T>
void QuickSort(T* arr, int left, int right) {
    if (left >= right) return;
    T pivot = arr[(left + right) / 2];
    int i = left, j = right;
    while (i <= j) {
        while (arr[i] < pivot) { ++i; }
        while (pivot < arr[j]) { --j; }
        if (i <= j) {
            SwapElems(arr[i], arr[j]);
            ++i; --j;
        }
    }
    if (left < j)  QuickSort(arr, left, j);
    if (i < right) QuickSort(arr, i, right);
}
```

Пример:

4	3	1	5	2	
<2	**2**	>=2			
1		4	3	5	
			<5	**5**	>=5
	4	3			
	<3	**3**	>=3		
			4		

7. Сортировка слиянием

```
// 7. Сортировка слиянием
template<typename T>
void Merge(T* src, T* buf, int left, int mid, int right) {
    int i = left, j = mid, l = left;
    while (i < mid && j < right) {
        if (src[i] < src[j]) {
            buf[l++] = src[i++];
        } else {
            buf[l++] = src[j++];
        }
    }
    while (i < mid) { buf[l++] = src[i++]; }
    while (j < right) { buf[l++] = src[j++]; }
}

template<typename T>
void MergeSortRec(T* src, T* buf, int left, int right) {
    if (right - left < 2) return;
    int mid = (left + right) / 2;
    MergeSortRec(src, buf, left, mid);
    MergeSortRec(src, buf, mid, right);
    Merge(src, buf, left, mid, right);
    std::memcpy(src + left, buf + left, sizeof(T) * (right - left));
}

template<typename T>
void MergeSort(T* arr, int n) {
    T* buf = new T[n];
    MergeSortRec(arr, buf, 0, n);
    delete[] buf;
}
```

Идея: Алгоритм "разделяй и властвуй".
1. Массив рекурсивно разбивается на две половины до тех пор, пока не останутся подмассивы размером в один элемент.
2. Отсортированные подмассивы сливаются (merge) в один большой отсортированный массив.
Сложность: Всегда O(n log n).
Особенности: Стабильная, надёжная сортировка. Требует дополнительной памяти O(n).

Пример:

		4	3	1	5	2		
	4	3	1			5	2	
4		3		1		5		2
	3	4		1		2	5	
		1	3	4		2	5	
		1	**2**	**3**	**4**	**5**		

8. Сортировка подсчётом

Идея: Не основана на сравнениях. Работает с числами в заданном диапазоне.
1. Создаётся массив-счётчик, где индекс — число из исходного массива, а значение — количество его вхождений.
2. На основе массива-счётчика формируется результирующий отсортированный массив.
Сложность: O(n + k), где k — размер диапазона чисел.
Особенности: Очень быстрая, но только для целых чисел с небольшим диапазоном. Требует много памяти, если диапазон велик.

Пример:
1 3 1 2 0 4 2 3
Кол-во = 8
К = 5

0	1	2	3	4
1	2	2	2	1

Получаем отсортированный массив 0 1 1 2 2 3 3 4

```cpp
// 8. Сортировка подсчётом (Counting sort)
void CountingSort(int* arr, int n) {
    if (n <= 0) return;
    int minVal = arr[0], maxVal = arr[0];
    for (int i = 1; i < n; ++i) {
        if (arr[i] < minVal) { minVal = arr[i]; }

if (arr[i] > maxVal) { maxVal = arr[i]; }
    }
    int range = maxVal - minVal + 1;
    int* cnt = new int[range];
    std::memset(cnt, 0, sizeof(int) * range);
    for (int i = 0; i < n; ++i) {
        ++cnt[arr[i] - minVal];
    }
    for (int i = 1; i < range; ++i) {
        cnt[i] += cnt[i - 1];
    }
    int* out = new int[n];
    for (int i = n - 1; i >= 0; --i) {
        out[--cnt[arr[i] - minVal]] = arr[i];
    }
    std::memcpy(arr, out, sizeof(int) * n);
    delete[] cnt;
    delete[] out;
}
```

9. BogoSort

Идея:

1. Проверить, отсортирован ли массив.
2. Если нет — перемешать элементы случайным образом.
3. Повторять шаги 1 и 2 до тех пор, пока массив не окажется отсортирован.

Сложность: O((n+1)!) в среднем, бесконечность в худшем случае.
Особенности: Абсолютно не практичный алгоритм.

```cpp
// 9. BogoSort (только для эксперимента)
template<typename T>
bool IsSorted(T* arr, int n) {
    for (int i = 1; i < n; ++i) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

template<typename T>
void BogoSort(T* arr, int n) {
    while (!IsSorted(arr, n)) {
        for (int i = 0; i < n; ++i) {
            std::swap(arr[i], arr[std::rand() % n]);
        }
    }
}
```