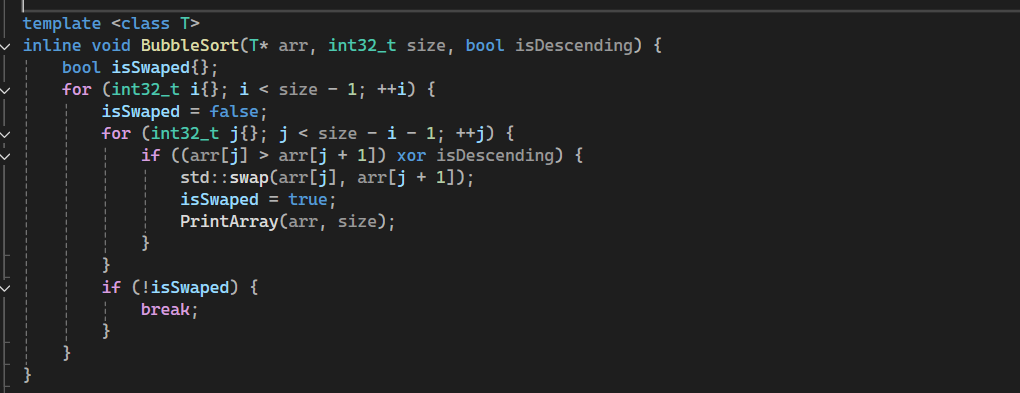
**Отчёт**

**1. Сортировка пузырьком (Bubble Sort)**

* **Временная сложность:** в среднем O(n^2), в лучшем случае O(n), в худшем O(n^2).



**Принцип работы**

Просто идём по массиву и сравниваем каждые два соседних элемента. Если левый больше правого, меняем их местами. Повторяем пока все большие элементы не всплывут.

**Плюсы**

* **Простой**.
* Если массив уже отсортирован, он это поймёт и закончит за один проход.

**Минусы**

* **Медленный.**

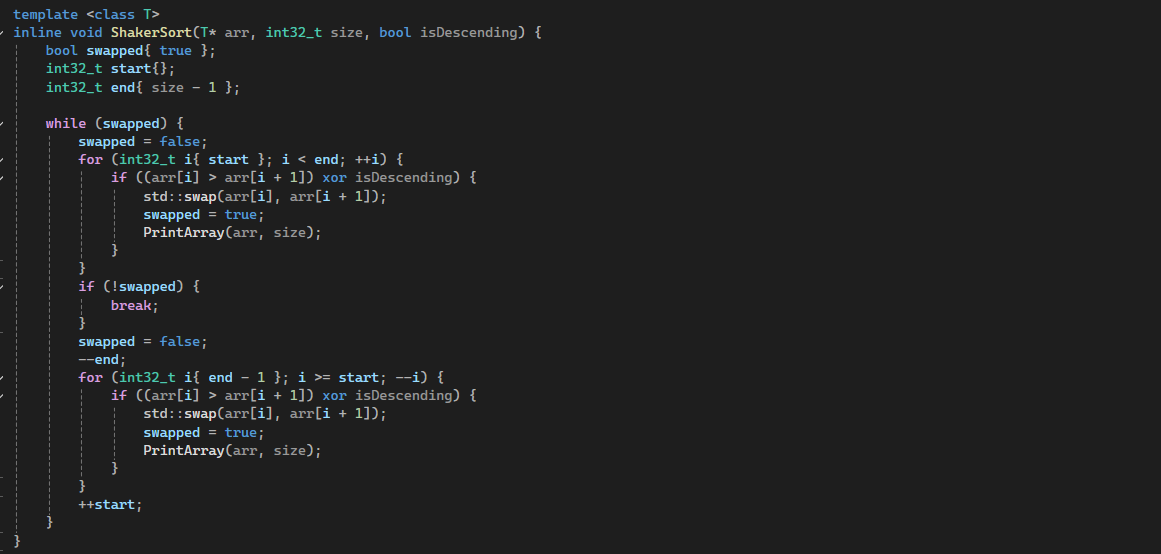
**Время работы**

* На 50 элементах: **4.38 секунды**.

**Вывод**

Слишком медленный для реальных задач.

**2. Шейкерная сортировка (Shaker Sort)**

* **Временная сложность:** в среднем O(n^2), в лучшем случае O(n), в худшем O(n^2).
* 

**Принцип работы**

Усовершенствованный пузырек. Элементы не только всплывают, но и тонут — в начало.

**Плюсы**

* Чуть-чуть лучше пузырька так как мелкие элементы быстрее попадают на свои места.

**Минусы**

* **Всё равно очень медленный.** Проблема та же, что и у пузырька — куча лишних сравнений и обменов.

**Время работы**

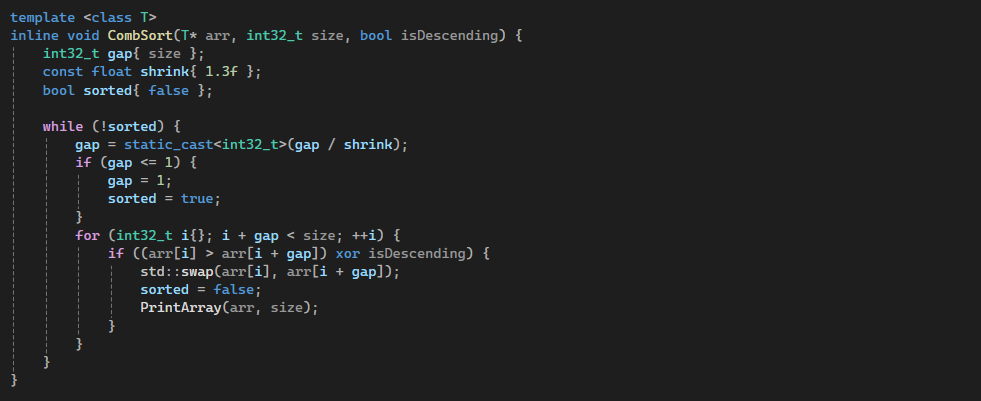
* На 50 элементах: **5.53 секунды**.

**Вывод**

На практике бесполезная. Сложность та же — O(n^2).

**3. Сортировка расчёской (Comb Sort)**

* **Временная сложность:** в среднем O(nlogn), в худшем O(n^2).



**Принцип работы**

Это улучшенный пузырёк в котором сначала сравниваются элементы, которые далеко друг от друга (большой "шаг"), а потом постепенно этот шаг уменьшается. Это помогает быстро перекинуть мелкие элементы из конца в начало. Когда шаг становится равен 1, алгоритм превращается в обычный пузырёк, но массив к этому моменту уже почти отсортирован.

**Плюсы**

* **Намного быстрее** пузырька и шейкера.
* Решает главную проблему пузырька — "черепах" (мелкие элементы в конце).

**Минусы**

* Чуть сложнее в реализации.
* Теоретически может попасть в плохой случай и работать за O(n^2).

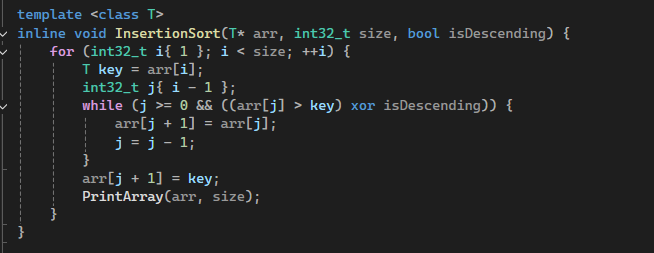
**Время работы**

* На 50 элементах: **0.94 секунды**.

**Вывод**

Простой и довольно эффективный.

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

* **Временная сложность:** в среднем O(n^2), в лучшем случае O(n), в худшем O(n2).
* 

**Принцип работы**

Делим массив на две части: отсортированную (вначале она пустая) и остальную. Берём по одному элементу из второй части и вставляем его в первую, но сразу на своё место, сдвигая другие элементы.

**Плюсы**

* Очень быстрый на маленьких или почти отсортированных массивах.
* Простой в написании.

**Минусы**

* Для больших и перемешанных данных — медленный.

**Время работы**

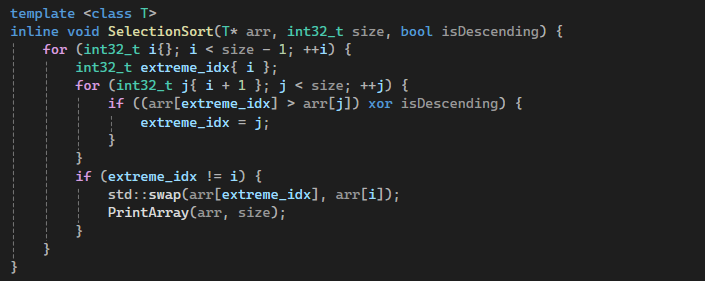
* На 50 элементах: **0.41 секунды**.

**Вывод**

Хорош для небольших задач.

**5. Сортировка выбором (Selection Sort)**

* **Временная сложность:** всегда O(n2).



**Принцип работы**

Ищем минимальный элемент во всём массиве и ставим его на первое место. Потом ищем минимальный из оставшихся и ставим на второе. И так до конца.

**Плюсы**

* Простой
* Делает минимум обменов, если это вдруг важно.

**Минусы**

* **Всегда медленный.**

**Время работы**

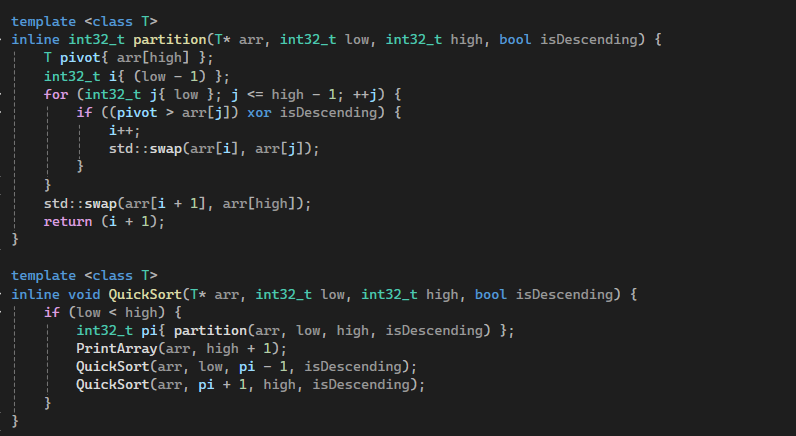
* На 50 элементах: **0.39 секунды**.

**Вывод**

Медленный. На практике почти не используется.

**6. Быстрая сортировка (Quick Sort)**

* **Временная сложность:** в среднем O(nlogn), в худшем O(n^2).



**Принцип работы**

Выбираем один элемент ("опорный"), и раскидываем все остальные: те, что меньше — налево от него, те, что больше — направо. Потом повторяем то же самое для левой и правой частей, пока всё не отсортируется.

**Плюсы**

* **Очень быстрый** в среднем. Почти всегда самый эффективный.
* Не требует дополнительной памяти.

**Минусы**

* Можно нарваться на худший случай (O(n^2)), если неудачно выбирать опорный элемент (например, на уже отсортированном массиве).
* Сложнее в реализации, чем простые сортировки.

**Время работы**

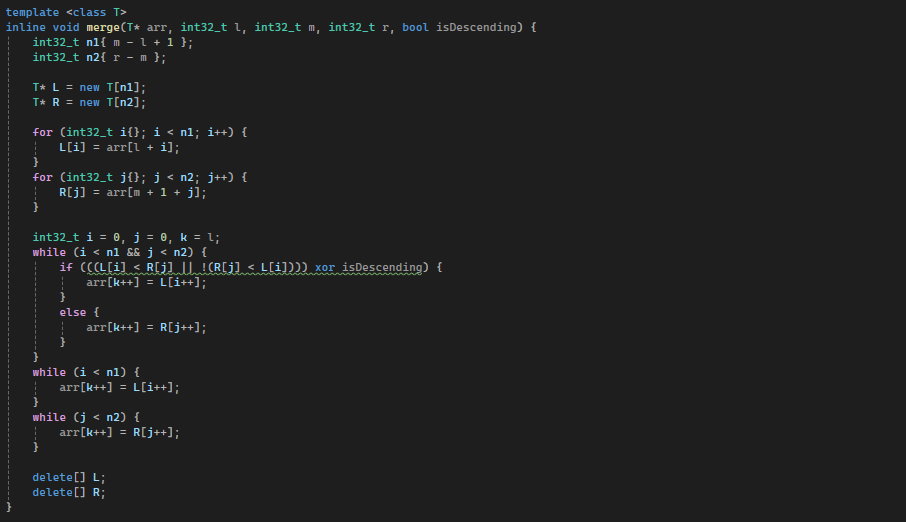
* На 50 элементах: **0.16 секунды**.

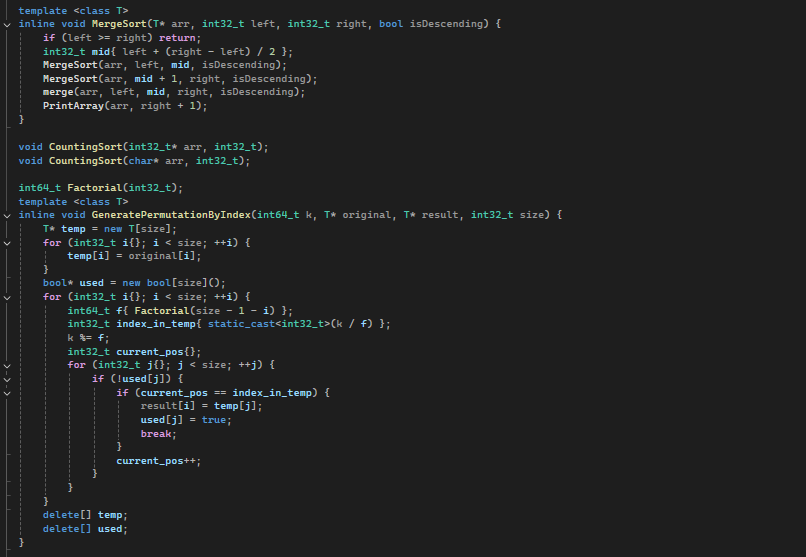
**Вывод**

В среднем самый эфеективный.

**7. Сортировка слиянием (Merge Sort)**

* **Временная сложность:** всегда O(nlogn).





**Принцип работы**

Делим массив пополам, и ещё пополам, пока не останутся массивы из одного элемента. А потом начинаем их сливать обратно, но уже в правильном порядке.

**Плюсы**

* **Стабильное время работы.** Никаких худших случаев, всегда O(nlogn).
* Стабильная сортировка (не меняет порядок одинаковых элементов).

**Минусы**

* **Много затрат памяти.** Ему нужен дополнительный массив такого же размера, что и исходный.

**Время работы**

* На 50 элементах: **0.23 секунды**.

**Вывод**

Надёжный алгоритм.

**8. Сортировка подсчётом (Counting Sort)**

* **Временная сложность:** O(n+k), где k — разброс значений.



**Принцип работы**

Работает только для целых чисел. Мы просто создаём массив-счётчик и считаем, сколько раз каждое число встретилось в исходном массиве. А потом просто выводим числа нужное количество раз.

**Плюсы**

* **Очень быстрая,** если числа лежат в небольшом диапазоне (например, от 0 до 100).

**Минусы**

* **Не универсальная.** Не работает с дробями или строками.
* Если числа очень большие (например, от 0 до миллиарда), то понадобится гигантский массив для счётчиков.

**Время работы**

* На 50 **целых** элементах: **0.176 секунды**.

**Вывод**

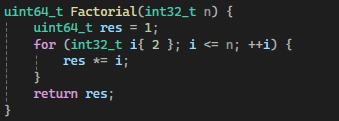
Специализированный инструмент. Если задача подходит под его условия — это лучшее решение.

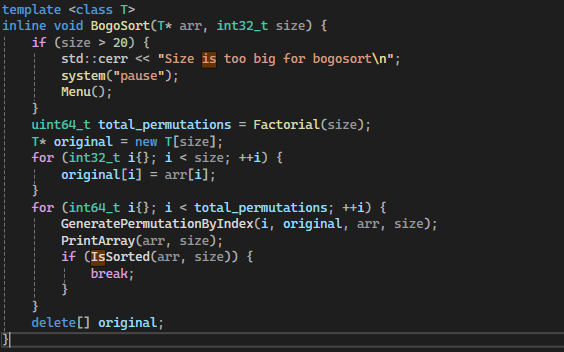
9. Сортировка перестановками (Bogosort)

* Временная сложность: в среднем O(n!), в худшем — O(n!).

A computer screen with white and blue text

AI-generated content may be incorrect.





Принцип работы

Эта версия Bogosort — не случайная, а детерминированная. Вместо того чтобы перемешивать массив наугад, она методично генерирует все возможные перестановки элементов по порядку. Для этого используется факториальная система счисления, которая позволяет получить любую перестановку по её номеру (от 0 до n!-1). Алгоритм просто перебирает все эти варианты один за другим, пока не наткнётся на отсортированный.

Плюсы

* Нету

Минусы

* Неэффективен. Количество перестановок растёт взрывным образом (n!), поэтому даже для 15-20 элементов ждать придётся дольше, чем существует Вселенная. Хоть он и гарантированно найдёт ответ (в отличие от случайной версии, которая может работать вечно).

Время работы

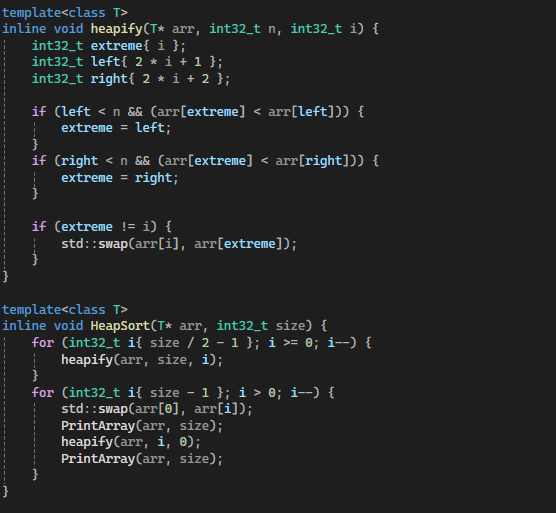
* На 8 элементах: 43.35 секунды.

Вывод

Так делать НЕ надо.

10. Пирамидальная сортировка (Heap Sort)

* Временная сложность: всегда O(nlogn).



Принцип работы

Алгоритм сначала выстраивает все элементы массива в специальную структуру — "кучу" (heap), где самый большой элемент всегда находится на вершине. Затем он забирает этот элемент, ставит его в конец массива, а кучу перестраивает. И так, пока не заберёт все элементы.

**Плюсы**

* Гарантированное время O(nlogn), как у слияния.
* Не требует дополнительной памяти, в отличие от слияния.

**Минусы**

* На практике обычно чуть медленнее, чем QuickSort.
* Нестабильная.

**Время работы**

* На 50 элементах: **0.95 секунды**.

**Вывод**

Надёжный вариант, если нужен быстрый алгоритм без риска нарваться на O(n^2) и без лишних затрат памяти.

**Общий вывод по лабе**

1. **"Быстрые" (**O(nlogn)**):** QuickSort, MergeSort, HeapSort. Самая большая скорость — QuickSort (0.16 с).
2. **"Медленные, но простые" (**O(n^2)**):** Пузырёк, шейкер, вставки, выбор. Очень медленные но простые.
3. **"Специализированные":**

CountingSort — супер быстрый для своих задач.

Bogosort — супер медленный для любых задач.

Для большинства реальных задач :**QuickSort**.

Если важна стабильность и есть лишняя память : **MergeSort**.

Если памяти нет, а худших случаев надо избежать : **HeapSort**.

Остальные — чисто для общего развития.