1. Сортировка выбором (Selection Sort) — это алгоритм, который разделяет массив на две части: отсортированную и неотсортированную. На каждом шаге он находит минимальный элемент в неотсортированной части и меняет его местами с первым элементом этой части. (сортировка выбором) O(*n*2) Для каждого прохода выполняется ещё один внутренний цикл сравнения, что даёт общее количество операций порядка .
2. Сортировка обменом (или пузырьковая сортировка) — это простой алгоритм, который многократно проходит по массиву, сравнивая соседние элементы и меняя их местами, если они расположены в неправильном порядке. **Временная сложность:**
   1. **Худший случай: O(n²)** — массив отсортирован в обратном порядке.
   2. **Лучший случай: O(n)** — массив уже отсортирован, срабатывает оптимизация.
   3. **Средний случай: O(n²)**

**Почему O(n²):** В худшем случае, как и сортировка выбором, требует n\*(n-1)/2 сравнений и потенциальных обменов.

1. Сортировка вставками (Insertion Sort) — это алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. **Временная сложность:**
   1. **Худший случай: O(n²)** — массив отсортирован в обратном порядке. Каждый элемент может потребовать i сдвигов.
   2. **Лучший случай: O(n)** — массив уже отсортирован. Внутренний цикл while не выполняется.
   3. **Средний случай: O(n²)**

**Почему O(n²):** В худшем случае общее количество сдвигов и сравнений составляет 1 + 2 + 3 + ... + (n-1) = n\*(n-1)/2.

1. Сортировка слиянием (Merge Sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

**Временная сложность: O(n log n)**

**Почему O(n log n):**

* 1. Глубина рекурсии: Массив делится пополам на каждом уровне рекурсии. Количество уровней (глубина дерева рекурсии) составляет log n.
  2. Работа на уровне: На каждом уровне рекурсии мы сливаем n элементов (все элементы массива). Функция merge для всего массива на одном уровне выполняет O(n) операций.
  3. Общая сложность: log n уровней \* n элементов на уровне = O(n log n).

1. Сортировка Шелла («Shellsort») представляет собой улучшенный вариант сортировки вставками. Она сначала сортирует элементы, находящиеся далеко друг от друга, постепенно уменьшая расстояние между ними, пока не перейдет к обычным соседним элементам.

**Время выполнения:**

* **Лучшая ситуация:** — практически отсортированный массив.
* **Средняя ситуация:** От до — сильно зависит от выбора шагов.
* **Худшая ситуация:** До — неправильный выбор начальных расстояний (например, постоянно уменьшающихся шагов).

Факторы, влияющие на производительность:

1. **Выбор расстояния (интервала)**: Важно правильно выбирать последовательность шагов (например, серии Хиббарда или Шелла). Неправильный выбор ухудшает эффективность.
2. **Размер массива**: Алгоритм эффективен для средних размеров массивов, но может уступать другим алгоритмам при обработке огромных данных.

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

**Временная сложность:**

* 1. **Средний случай: O(n log n)** — если опорный элемент делит массив примерно пополам на каждом шаге.
  2. **Худший случай: O(n²)** — если опорный элемент всегда является минимальным или максимальным (например, массив уже отсортирован, и опорный всегда последний). В этом случае дерево рекурсии имеет глубину n, а на каждом уровне выполняется n сравнений.
  3. **Лучший случай: O(n log n)** — если опорный всегда делит массив ровно пополам.

**Почему O(n log n) или O(n²):**

* 1. **Средний/Лучший:** Глубина рекурсии log n, на каждом уровне n сравнений (в partition). n \* log n.
  2. **Худший:** Глубина рекурсии n, на каждом уровне до n, n-1, n-2, ... сравнений. n + (n-1) + ... + 1 = n(n+1)/2 ≈ O(n²).

7 . **Пирамидальная сортировка (Heapsort)**

* **Временная сложность: O(n log n)**
* **Почему O(n log n):**
  + **Построение кучи:** build\_max\_heap выполняет heapify для n/2 узлов. Каждый вызов heapifyможет иметь глубину log n. В сумме это O(n) (доказывается математически, так как большинство узлов находятся ближе к листьям).
  + **Сортировка:** Цикл for выполняется n-1 раз. Внутри него вызывается heapify, который работает за O(log n) (глубина дерева). Итого n \* O(log n) = O(n log n).
  + Общая сложность: O(n) (построение) + O(n log n) (сортировка) = O(n log n).

**8. последовательный поиск**

* **Временная сложность:**
  + **Худший случай: O(n)** — элемент находится в конце массива или отсутствует.
  + **Лучший случай: O(1)** — элемент находится в начале массива.
  + **Средний случай: O(n/2) ≈ O(n)**
* **Почему O(n):** В худшем случае нужно проверить все n элементов.

**9. Временная сложность: O(log n)**

* **Почему O(log n):** На каждом шаге размер области поиска уменьшается примерно вдвое. Количество шагов, необходимых для сокращения n до 1, равно log₂ n.

**10. Интерполирующий поиск**

* **Временная сложность:**
  + **Средний случай (равномерное распределение): O(log log n)**
  + **Худший случай (неравномерное распределение): O(n)**
* **Почему O(log log n) или O(n):**
  + **Средний:** При равномерном распределении, на каждом шаге размер области поиска уменьшается быстрее, чем в бинарном поиске. Количество шагов приблизительно равно log log n.
  + **Худший:** Если данные распределены неравномерно (например, большинство элементов сосредоточено в начале), pos может вычисляться близко к lo, и алгоритм может деградировать до линейного сканирования, проверяя каждый элемент по одному.

11.  **Фибоначчи поиск**

* **Временная сложность: O(log n)**
* **Почему O(log n):** Количество чисел Фибоначчи до n примерно равно log n. Каждая итерация цикла while уменьшает размер области поиска, используя меньшие числа Фибоначчи, что приводит к O(log n) итераций.