Тема 14. Алгоритмы поиска и сортировки

Алгоритмы поиска и сортировки лежат в основе множества прикладных задач: от фильтрации данных в пользовательских интерфейсах до внутренней логики работы баз данных и операционных систем. Знание этих алгоритмов необходимо не только для написания эффективного кода, но и для осознания того, как устроены типовые операции с данными, которые кажутся очевидными в повседневной практике. За простыми действиями — найти значение или упорядочить список — стоит набор решений, различающихся по стратегии, эффективности и устойчивости к изменению входных данных.

На этом этапе целесообразно не просто познакомиться с классическими алгоритмами сортировки и поиска, но и научиться анализировать их применимость в разных ситуациях. Это требует не только изучения теоретических основ и реализаций, но и проведения экспериментов на реальных массивах данных, с замерами времени выполнения и сравнением поведения алгоритмов. Такой подход позволит глубже понять, почему в одних случаях достаточно простой сортировки пузырьком, а в других — требуется использование быстрой или сортировки слиянием.

Поиск в списках/массивах. Бинарный поиск

Нахождение элемента в списке или массиве представляет собой одну из ключевых задач в информатике, являясь неотъемлемой частью как теоретических исследований алгоритмов, так и практических приложений в вычислительной технике. Процесс поиска заключается в определении наличия требуемого значения в заданной структуре данных, а в случае положительного результата — в установлении точного его местоположения. Выбор конкретного метода поиска зависит от ряда характеристик: размера и структуры данных, степени упорядоченности элементов, их распределения, а также наличия дополнительной памяти для реализации алгоритма.

Наиболее элементарной стратегией является ***линейный поиск***, при котором осуществляется последовательный перебор каждого элемента списка до тех пор, пока не будет обнаружено совпадение с искомым значением или пока не исчерпаны все элементы. Данный метод отличается простотой реализации и понятностью алгоритмической идеи, однако его эффективность резко падает при работе с большими объёмами данных, поскольку в худшем случае время работы определяется как $O(n)$, где n — количество элементов в массиве. Это обстоятельство делает линейный поиск менее предпочтительным для систем, где критична скорость обработки информации.

В противоположность линейному поиску применяется ***бинарный поиск***, который является более совершенным и эффективным методом, однако предполагает наличие предварительной сортировки массива. Алгоритм бинарного поиска основывается на принципе деления массива пополам с последующей проверкой центрального элемента: если найденное значение совпадает с искомым, поиск завершается; если искомый элемент меньше центрального, дальнейшие поисковые операции проводятся в левой половине массива, иначе — в правой. Благодаря такому принципу, количество необходимых сравнений существенно сокращается, что позволяет добиться временной сложности, выражаемой как $O(\log n)$. Следует отметить, что хотя бинарный поиск значительно ускоряет процесс доступа к элементу, его применение возможно только в случае отсортированных данных, и предварительная сортировка может внести дополнительные вычислительные затраты.

Ещё одним интересным подходом является ***интерполяционный поиск***, который эффективен при равномерном распределении элементов в массиве — то есть когда значения элементов увеличиваются с приблизительно одинаковым шагом и равномерно заполняют диапазон от минимального до максимального. Этот метод основывается на использовании специальной математической формулы, позволяющей оценить предполагаемое положение искомого элемента, что даёт возможность сразу сузить область дальнейшего поиска. В идеальных условиях интерполяционный поиск способен превосходить по скорости как линейный, так и бинарный методы. Однако его эффективность существенно снижается, если элементы распределены неравномерно, поскольку оценка позиции может оказаться далёкой от реального положения значения, что приводит к необходимости выполнения дополнительных итераций.

Наиболее современные решения задачи поиска часто реализуются на основе ***хеш-таблиц***. *Принцип работы таких структур данных заключается в применении хеш-функции для преобразования ключа элемента в индекс массива, что позволяет осуществлять доступ практически мгновенно. Преимущество данного метода заключается в возможности прямого обращения к элементу без необходимости последовательного или бинарного перебора, что делает его привлекательным для задач с высокими требованиями к скорости обработки. Однако использование хеш-таблиц сопряжено с необходимостью выделения дополнительной памяти и внедрения механизмов разрешения* **коллизий*** — ситуации, когда два различных ключа приводят к одинаковому значению хеш-функции. Эти механизмы, будь то метод цепочек или открытая адресация, добавляют алгоритму определённую сложность, но при правильном подходе позволяют существенно ускорить поиск, особенно при работе с большими объёмами данных. Следует учитывать, что хеш-таблицы не подходят для задач, связанных с диапазонными запросами или сохранением порядка элементов.

Выбор оптимального метода поиска определяется не только размерами и упорядоченностью данных, но и особенностями их распределения, а также возможностями выделения дополнительных вычислительных ресурсов. Линейный поиск, обладая временной сложностью $O(n)$, подходит для небольших или неструктурированных списков, тогда как бинарный поиск с его $O(\log n)$ эффективен для отсортированных массивов, но требует предварительной сортировки. Интерполяционный поиск может оказаться крайне продуктивным при равномерном распределении элементов, хотя и не всегда стабилен в

эффективности, а применение хеш-таблиц позволяет достичь максимальной скорости доступа за счёт более сложной реализации и дополнительных требований к памяти. Подобный аналитический подход к выбору метода поиска способствует повышению общей производительности вычислительных систем, что имеет решающее значение при реализации как теоретических моделей, так и практических приложений в области обработки данных.

Рассмотрим пример реализации бинарного поиска:

In [ ]:
```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Функция `binary_search` принимает два аргумента: отсортированный массив `arr` и элемент `x`, который нужно найти. Внутри функции создаются переменные `low` и `high`, которые задают границы поиска — начальный и конечный индексы массива. Затем запускается цикл `while`, который продолжается, пока `low` не станет больше `high`.

Внутри цикла вычисляется индекс среднего элемента массива с помощью формулы `(low + high) // 2`. Если значение в середине равно искомому элементу, то функция возвращает его индекс. Если значение в середине меньше искомого, то поиск продолжается только в правой половине массива, поэтому переменная `low` устанавливается на значение `mid + 1`. Если значение в середине больше искомого, то поиск продолжается только в левой половине массива, поэтому переменная `high` устанавливается на значение `mid - 1`. Если же элемент не найден, то функция возвращает `-1`.

Создадим список из двадцаци случайных чисел:

In [ ]:
```
import random
```

In [ ]:
```
arr = random.sample(range(100), 20)
```

In [ ]:
```
print(f"Исходный список: {arr}")
```

```
Исходный список: [94, 95, 87, 42, 81, 92, 45, 5, 30, 79, 32, 13, 40, 22, 50, 98, 80, 59, 93, 11]
```

Предварительно отсортируем список и определении элемент для поиска:

In [ ]:
```
arr.sort()
```

In [ ]:
```
print(f"Отсортированный список: {arr}")
```

```
Отсортированный список: [5, 11, 13, 22, 30, 32, 40, 42, 45, 50, 59, 79, 80, 81, 87, 92, 93, 94, 95, 98]
```

In [ ]:
```
x = int(input("Введите искомый элемент: "))
```

```
Введите искомый элемент: 87
```

Найдём нужный элемент в списке:

In [ ]:
```
result = binary_search(arr, x)
```

In [ ]:
```
print("Элемент найден на позиции {}".format(result) if result != -1
      else "Элемент не найден")
```

```
Элемент найден на позиции 14
```

Таким образом, бинарный поиск представляет собой эффективный алгоритм, позволяющий значительно сократить количество проверок за счёт поэтапного деления отсортированного массива на две части. В приведённом примере видно, как с помощью простой логики сравнения и обновления границ поиска можно быстро определить наличие искомого элемента и его позицию. Однако важно помнить, что бинарный поиск применим только к отсортированным данным — в противном случае его результат будет некорректен.

Задачи поиска неизбежно возникают в самых разных прикладных и теоретических контекстах, а потому владение различными стратегиями и понимание их применимости к конкретным ситуациям играет важную роль в эффективной работе с данными. Независимо от выбранного подхода — будь то простой линейный перебор, логически организованный бинарный или интерполяционный поиск, либо использование хеш-таблиц — ключевым остаётся умение учитывать характеристики исходных данных и выбирать алгоритм, обеспечивающий наилучший баланс между скоростью, сложностью реализации и потреблением ресурсов.

При анализе алгоритмов поиска важно учитывать не только временные характеристики в худшем и среднем случаях, но и

влияние структурных особенностей данных на общую производительность методов. Например, бинарный поиск в отсортированном массиве демонстрирует эффективность с временной сложностью $O(\log n)$ как в худшем, так и в среднем случаях, что обеспечивает надёжное применение метода при условии предварительной сортировки. В отличие от бинарного поиска, линейный поиск всегда работает со сложностью $O(n)$ независимо от характера данных, что приводит к значительному увеличению времени обработки при росте объёма информации. При использовании хеш-таблиц возникает вероятность ухудшения показателей времени доступа при неблагоприятном распределении ключей, что может привести к деградации сложности до $O(n)$ вследствие возникновения коллизий и необходимости их разрешения.

Роль предварительной сортировки данных становится особенно заметной, когда требуется выполнять множество операций поиска в большом объёме информации. В этом случае однократная сортировка исходного массива с последующим применением бинарного поиска оказывается существенно эффективнее, чем повторяющийся линейный перебор, позволяя снизить временные затраты и оптимизировать общую производительность системы.

Связь алгоритмов поиска с более сложными структурами данных имеет большое практическое значение. Бинарный поиск лежит в основе различных алгоритмов, в том числе алгоритма бинарного поиска по ответу, и широко применяется в деревьях поиска, в AVL- и красно-чёрных деревьях. Эти структуры используют принцип бинарного разделения для обеспечения быстрого доступа к элементам, что является важным условием при разработке эффективных систем управления данными. Кроме того, алгоритмы поиска применимы не только для непосредственного обнаружения значений в массивах, но и при решении задач оптимизации, когда требуется определить минимальное или максимальное значение параметра, при котором выполняется заданное условие, что известно как бинарный поиск по ответу.

Адаптации методов поиска учитывают особенности структур данных: для массивов, отсортированных по убыванию, либо содержащих повторяющиеся элементы, модификации бинарного поиска направлены на определение первого или последнего вхождения заданного значения, что значительно расширяет возможности анализа данных и повышает точность результатов. В совокупности, современные алгоритмы поиска демонстрируют разнообразие подходов, что позволяет выбирать оптимальное решение с учётом конкретных условий задачи, особенностей распределения данных и доступных вычислительных ресурсов, обеспечивая высокую эффективность обработки информации.

Простые методы сортировки

Сортировка — это процесс упорядочивания элементов в определённом порядке, который играет фундаментальную роль в информатике и применяется как в прикладных задачах, так и в теоретических исследованиях алгоритмов. Эффективная сортировка данных необходима для повышения производительности программ, упрощения последующих вычислений, а также подготовки структур к дальнейшему анализу. Она используется в операционных системах, базах данных, сетевых протоколах, алгоритмах оптимизации, машинном обучении, компьютерной графике и множестве других направлений, где требуется организованное представление информации.

Среди основных сценариев применения сортировки выделяются ускорение поиска и выборки данных, предварительная обработка перед бинарным поиском, а также фильтрация и агрегация информации. Например, в информационно-поисковых системах предварительная сортировка позволяет мгновенно находить нужные значения при помощи бинарного поиска, а в базах данных — извлекать записи в заданном порядке по значениям одного или нескольких атрибутов. Кроме того, упорядочивание данных необходимо при построении индексов, что критически важно для быстрого доступа в системах управления большими массивами информации.

Сортировка часто используется и в алгоритмах слияния, включая внешнюю сортировку, применяемую при работе с данными, не помещающимися в оперативной памяти. В области анализа данных она необходима для вычисления медиан, квантилей, построения гистограмм и других статистических характеристик. В задачах машинного обучения сортировка помогает упорядочивать объекты по значению целевой переменной или предсказанного признака — это может быть полезно, например, при отборе наиболее значимых признаков или в алгоритмах ближайших соседей. В системах рекомендаций, основанных на рейтингах, сортировка позволяет отображать наиболее релевантные объекты в нужной последовательности. для работы с данными, не помещающимися в оперативной памяти.

В компьютерной графике и разработке игр сортировка применяется при отрисовке трёхмерных сцен — необходимо отсортировать объекты по глубине, чтобы правильно определить порядок их отображения и обеспечить реалистичное наложение визуальных элементов. При этом от алгоритма сортировки требуется не только корректность, но и высокая скорость выполнения, поскольку производительность в реальном времени имеет решающее значение.

Существует множество алгоритмов сортировки, отличающихся по сложности, стабильности, объёму используемой памяти и подходу к организации данных. Наиболее известные среди них — сортировка пузырьком, выбором, вставками, быстрая сортировка, сортировка слиянием и др. Некоторые из них обладают простой реализацией и подходят для обучения, тогда как другие обеспечивают высокую эффективность при обработке больших объёмов информации. Выбор конкретного метода сортировки зависит от характеристик входных данных, требований к стабильности результата и допустимых затрат по времени и памяти.

Таким образом, сортировка — это не только вспомогательный этап в решении задач, но и ключевой механизм организации данных, влияющий на эффективность вычислений и структуру алгоритмов в целом. Глубокое понимание принципов работы различных алгоритмов сортировки и знание их применимости к конкретным ситуациям позволяет проектировать более

надёжные и производительные программные решения.

Обменные сортировки

Простая обменная сортировка (сортировка пузырьком)

Обменная сортировка представляет собой один из самых простых алгоритмов упорядочивания данных, основанный на последовательном сравнении и попарном обмене соседних элементов массива. Несмотря на свою кажущуюся наивность, данный метод помогает наглядно понять суть процессов, происходящих в ходе упорядочивания, и именно поэтому часто используется в образовательных целях.

Наиболее известной разновидностью обменной сортировки является ***сортировка пузырьком*** (*Bubble Sort*), получившая своё название благодаря характерному поведению элементов: наибольшие значения «всплывают» к концу массива по мере прохождения каждого этапа алгоритма. На каждом проходе по массиву последовательно сравниваются пары соседних элементов, и если текущий элемент больше следующего, производится их обмен. После завершения одного полного прохода самый крупный элемент оказывается в конце массива и больше не участвует в следующих итерациях. Таким образом, каждый проход «фиксирует» ещё один элемент в правильной позиции, а оставшаяся часть массива обрабатывается повторно до полной упорядоченности.

На рис. 1 представлен первый проход алгоритма сортировки пузырьком. Каждая строка отображает результат одного шага, включающего сравнение и возможный обмен пары элементов. Подписи справа позволяют отследить, происходил ли обмен в конкретной паре. Видно, как элемент со значением `93`, будучи максимальным в массиве, перемещается вправо на каждом шаге, пока не достигает своей окончательной позиции. Серая подсветка визуально выделяет текущую зону сравнения и подчёркивает пошаговый характер сортировки. Это делает алгоритм понятным, так как позволяет увидеть, как за несколько итераций крупные элементы постепенно «выталкиваются» к концу массива, а более мелкие перемещаются ближе к началу.

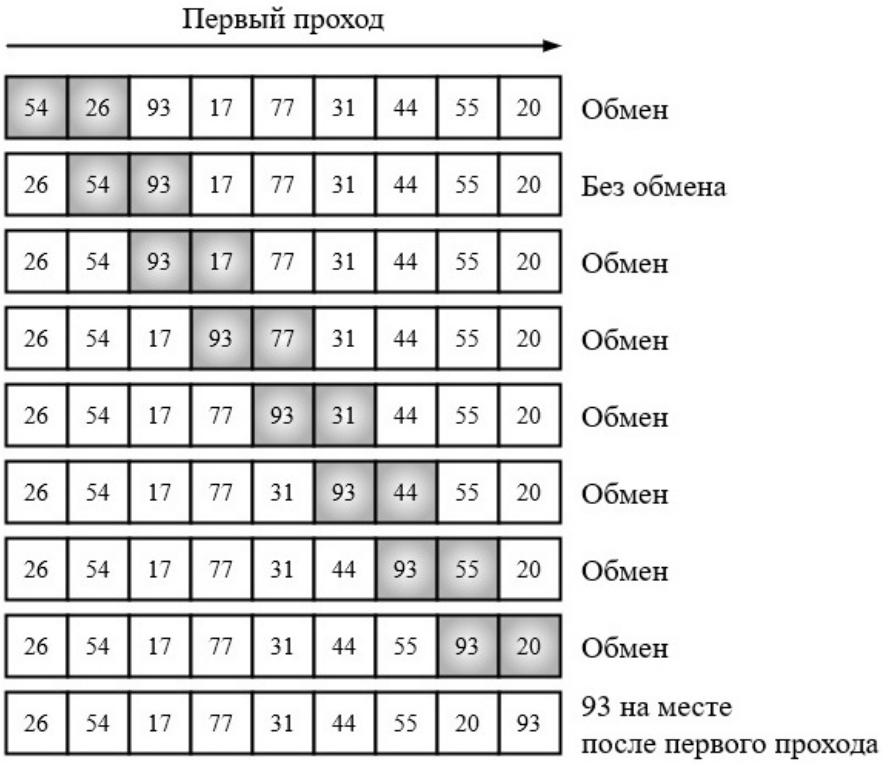
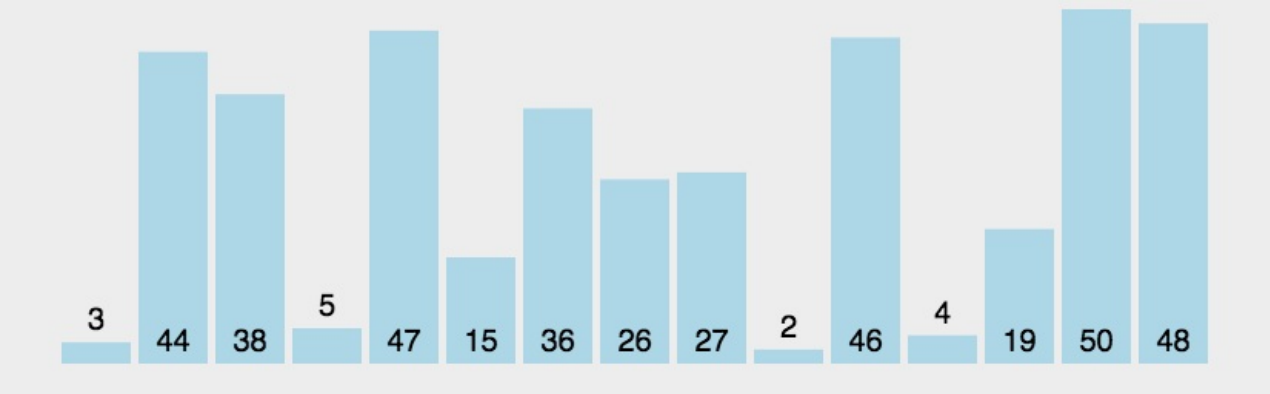


Рисунок 1 — Пример первого прохода алгоритма сортировки пузырьком

Хотя сортировка пузырьком проста в реализации и легко поддаётся визуализации, она обладает низкой эффективностью при работе с большими объёмами данных. Её худшая и средняя временная сложность составляет $O(n^2)$, что делает её непрактичной в реальных проектах при значительных объёмах информации. Однако в случаях, когда массив почти отсортирован, данный алгоритм может завершиться быстрее — особенно в модифицированных версиях, где предусмотрена проверка наличия обменов. Если на одном из проходов не произошло ни одного обмена, выполнение можно завершить досрочно, поскольку массив уже отсортирован. Эта особенность позволяет повысить производительность в частных случаях и делает алгоритм пригодным для задач, где ожидается почти отсортированный вход.



Visualgo

Сортировка пузырьком также служит основой для сравнения с более эффективными алгоритмами, например, быстрой сортировкой или сортировкой слиянием. Именно её очевидная пошаговость позволяет хорошо продемонстрировать, какие именно улучшения привносят более сложные алгоритмы: отказ от избыточных сравнений, уменьшение количества обменов и использование рекурсии или вспомогательных структур. Таким образом, хотя обменная сортировка редко применяется в промышленной разработке, она сохраняет свою значимость как базовый инструмент изучения принципов сортировки и анализа алгоритмов в целом.

Пример реализации алгоритма сортировки пузырьком:

In [ ]:
```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j+1], arr[j]
    return arr
```

Сначала определяется длина списка `n`. Затем внешний цикл `for i in range(n)` проходит по элементам списка `arr` от первого до последнего, а внутренний цикл `for j in range(n - i - 1)` проходит по элементам списка `arr` от первого до предпоследнего, т.е. на каждом проходе внешнего цикла самый большой элемент «всплывает» на последнюю позицию.

Внутри второго цикла проверяется, если текущий элемент `arr[j]` больше следующего за ним элемента `arr[j + 1]`, то они меняются местами с помощью оператора присваивания `arr[j], arr[j + 1] = arr[j + 1], arr[j]`. Такой проход гарантирует продвижение наибольшего из оставшихся элементов к концу массива.

После завершения работы внешнего цикла, список `arr` будет отсортирован в порядке возрастания. Отсортированный список возвращается из функции.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {bubble_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Пример реализации алгоритма сортировки пузырьком с параметром `reverse` (`False` — в порядке возрастания, `True` — в порядке убывания):

In [ ]:
```python
def bubble_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if not reverse:
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
            else:
                if arr[j] < arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{bubble_sort(arr)} — в порядке возрастания\n'
      f'\t{bubble_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Шейкерная сортировка

Шейкерная сортировка, также называемая *двунаправленной пузырьковой сортировкой* или *cocktail sort*, представляет собой модификацию классического пузырькового алгоритма, призванную повысить его эффективность за счёт изменения направления прохода по массиву. Если в сортировке пузырьком элементы продвигаются к нужной позиции только с одного конца, то шейкерная сортировка обеспечивает симметричную обработку: крупные элементы «выталкиваются» к концу, а мелкие — к началу массива. Благодаря этому достигается более быстрая стабилизация упорядоченного порядка в массивах, содержащих значения, удалённые от своих целевых позиций.

Алгоритм реализует попеременные проходы слева направо и справа налево. На прямом проходе сравниваются соседние элементы, и в случае, если текущий элемент больше следующего, они меняются местами. Таким образом, наибольший из оставшихся элементов сдвигается к концу массива. После завершения прямого прохода правый предел сортируемой области уменьшается. Затем начинается обратный проход от правой границы к левой, при котором минимальный элемент среди оставшихся продвигается к началу массива, после чего граница с левой стороны сдвигается вправо. Такой чередующийся процесс продолжается до тех пор, пока массив не будет полностью отсортирован.

Как и в улучшенной версии пузырьковой сортировки, в алгоритме используется специальный флаг, указывающий на наличие обменов во время прохода. Если ни один обмен не был произведён ни на прямом, ни на обратном направлении, это свидетельствует о том, что массив уже отсортирован, и выполнение алгоритма можно завершить досрочно. Эта особенность делает шейкерную сортировку адаптивной и особенно эффективной в тех случаях, когда входные данные близки к отсортированному состоянию.

Одним из основных преимуществ шейкерной сортировки является её способность быстрее обрабатывать массивы с разбросанными минимальными и максимальными элементами. В отличие от пузырькового метода, который «выталкивает» крупные значения только в одну сторону, шейкерная сортировка симметрично оптимизирует продвижение данных, сокращая число необходимых итераций. За счёт сужающихся границ и двустороннего движения она демонстрирует лучшие результаты на массивах, содержащих элементы, значительно удалённые от своих целевых позиций с обеих сторон.

Тем не менее, несмотря на указанное улучшение, алгоритм сохраняет квадратичную временную сложность в худшем и среднем случае — $O(n^2)$. Это ограничивает его применимость при работе с большими объёмами данных, особенно в тех ситуациях, где необходима высокая производительность. Кроме того, при равномерно распределённых случайных данных шейкерная сортировка не даёт существенного выигрыша по сравнению с улучшенной пузырьковой сортировкой, так как всё ещё опирается на попарные сравнения и множественные проходы.

Благодаря своей прозрачной логике и простой реализации, шейкерная сортировка сохраняет значимость. Она иллюстрирует важные концепции: адаптивность, ограничение рабочей области, двунаправленное прохождение массива и оптимизацию за счёт раннего завершения. В этом смысле алгоритм выступает логическим развитием пузырьковой сортировки и позволяет лучше понять принципы обменных методов, их возможности и пределы эффективности.

Таким образом, шейкерная сортировка представляет собой улучшенный обменный алгоритм, сочетающий простоту, адаптивность и симметричность обработки. Хотя она и не входит в число практических инструментов для сортировки больших массивов, её концептуальная ценность и обучающая функция делают её достойной частью арсенала базовых алгоритмов сортировки.

Пример реализации алгоритма шейкерной сортировки:

In [ ]:
```python
def cocktail_sort(arr):
    n = len(arr)
    start = 0
    end = n - 1
    swapped = True
    while swapped:
        swapped = False
        for i in range(start, end):
            if (arr[i] > arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end = end - 1
        for i in range(end - 1, start - 1, -1):
            if (arr[i] > arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start = start + 1
```

```
    return arr
```

Функция `cocktail_sort` принимает на вход список/массив `arr`. В начале определяются вспомогательные переменные: `n` обозначает длину массива, `start` и `end` задают границы неотсортированной области, а логическая переменная `swapped` используется для отслеживания наличия обменов на текущем проходе. Эта переменная также управляет продолжением работы основного цикла.

Цикл `while` выполняется до тех пор, пока в процессе очередного прямого или обратного прохода совершается хотя бы один обмен. В начале каждой итерации флаг `swapped` устанавливается в значение `False`. Затем выполняется прямой проход: перебираются элементы от индекса `start` до индекса `end`. Если текущий элемент больше следующего, происходит обмен значений, а переменная `swapped` принимает значение `True`. Таким образом, максимальный из оставшихся элементов продвигается к правому краю массива. После завершения прямого прохода правая граница `end` сдвигается влево, поскольку последний элемент уже находится на своём месте.

Если в ходе прямого прохода не произошло ни одной перестановки, массив считается отсортированным, и выполнение алгоритма прерывается досрочно. В противном случае флаг `swapped` снова сбрасывается в `False`, и начинается обратный проход. На этом этапе перебираются элементы в обратном направлении — от индекса `end - 1` до `start`. Если текущий элемент больше следующего, происходит обмен, и флаг `swapped` снова устанавливается в значение `True`. После этого наименьший из оставшихся элементов оказывается в начале массива, а граница `start` сдвигается вправо, позволяя продолжить следующий проход уже в более узкой области.

После завершения всех итераций функция возвращает отсортированный список `arr`.

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {cocktail_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

В расширенной версии алгоритма предусмотрен параметр `reverse`, позволяющий задавать направление сортировки. При значении `False` выполняется сортировка по возрастанию, при `True` — по убыванию. Условие сравнения внутри обоих проходов модифицируется в зависимости от значения параметра, сохраняя общую структуру алгоритма.

In [ ]:
```
def cocktail_sort(arr, reverse=False):
    n = len(arr)
    start = 0
    end = n - 1
    swapped = True
    while swapped:
        swapped = False
        for i in range(start, end):
            if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end = end - 1
        for i in range(end - 1, start - 1, -1):
            if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start = start + 1
    return arr
```

Пример вызова функции демонстрирует работу сортировки в двух режимах: сначала массив упорядочивается по возрастанию, затем — по убыванию. Такая универсальность делает реализацию гибкой и удобной для задач с предсказуемыми или частично отсортированными данными, а также в ситуациях, где важна простота алгоритма и пошаговый контроль его работы.

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{cocktail_sort(arr)} — в порядке возрастания\n'
      f'\t{cocktail_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Сортировка расчёской

Сортировка расчёской (*Comb Sort*) представляет собой усовершенствованный вариант пузырьковой сортировки, призванный устранить её основной недостаток — неэффективность при обработке удалённых друг от друга элементов, находящихся в

неправильном порядке. Алгоритм был предложен в 1980 году в исследовательской группе компании Hewlett-Packard и получил своё название благодаря характерной стратегии «прочёсывания» массива с шагом, превышающим единицу, по аналогии с зубьями расчёски.

В основе сортировки расчёской лежит идея о сравнении не только соседних, но и более отдалённых элементов массива. На первых этапах используются крупные интервалы между сравниваемыми элементами, что позволяет быстро устранить грубые нарушения порядка и переместить большие значения ближе к концу массива. По мере выполнения алгоритма величина промежутка постепенно уменьшается до минимального значения, равного единице. На завершающих этапах происходит окончательная доупорядоченность массива, аналогичная последним проходам пузырьковой сортировки.

Алгоритм начинается с установки промежутка, равного длине массива. После этого выполняется серия проходов, в каждом из которых сравниваются элементы, находящиеся друг от друга на заданном расстоянии. Если очередная пара элементов находится в неправильном порядке, происходит обмен. По завершении каждого прохода величина промежутка пересчитывается с использованием фиксированного коэффициента, равного примерно `1.3`. Этот коэффициент был эмпирически подобран с учётом минимизации числа сравнений и ускорения сходимости к упорядоченному состоянию. Когда промежуток становится равным единице, выполняется последний цикл сравнения соседних элементов, аналогичный одному проходу пузырьковой сортировки.

В отличие от пузырьковой сортировки, где элементы могут продвигаться к своей позиции только по одному шагу за проход, в сортировке расчёской крупные значения перемещаются быстрее, поскольку сравниваются и обмениваются с элементами, расположенными на значительном расстоянии. Это полезно при наличии так называемого эффекта «черепах» — малых элементов, задерживающих общее продвижение массива к отсортированному состоянию. Алгоритм эффективно справляется с такими ситуациями на ранних этапах за счёт больших шагов сравнения.

Средняя временная сложность сортировки расчёской ниже, чем у пузырьковой сортировки, несмотря на ту же асимптотику в худшем случае — $O(n^2)$. На практике она демонстрирует высокую эффективность при работе с массивами случайной природы, а также при необходимости упорядочивания данных с минимальными накладными расходами и без использования дополнительной памяти. Алгоритм является адаптивным: если массив уже частично отсортирован, количество необходимых проходов сокращается.

Одним из потенциальных ограничений алгоритма является чувствительность к выбору коэффициента уменьшения промежутка. При слишком быстром уменьшении возможно преждевременное наступление стадии, когда сравниваются только соседние элементы. Это может привести к тому, что удалённые элементы останутся на своих местах, не успев продвинуться в нужную часть массива, в результате чего сортировка окажется менее эффективной. Такое поведение называют эффектом «зубчатости», и его влияние особенно заметно при плохом подборе параметров.

Таким образом, сортировка расчёской представляет собой эффективный обменный алгоритм, сочетающий простоту реализации и способность ускоренно устранять нарушения порядка на ранних этапах. Она занимает промежуточное положение между пузырьковой сортировкой и более сложными методами, демонстрируя лучшие результаты без значительного усложнения логики. Благодаря этим свойствам алгоритм используется в прикладных задачах, где важны сбалансированная производительность, прозрачная логика и компактная реализация без дополнительной памяти.

Пример реализация алгоритма сортировки расчёской:

In [ ]:
```python
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True
    while swapped:
        gap = int(gap/shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = False
        while i + gap < n:
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True
            i += 1
    return arr
```

Функция `comb_sort` принимает на вход список/массив `arr`. В начале работы определяется длина массива `n`, и переменной `gap` присваивается это значение в качестве начального промежутка между сравниваемыми элементами. Параметр `shrink` задаёт коэффициент уменьшения промежутка, который используется для плавного приближения к финальной стадии сортировки, где `gap` должен принять значение, равное единице. Флаг `swapped` используется для отслеживания того, происходили ли обмены на текущем проходе. Его значение контролирует продолжение основного цикла.

Основной цикл `while` продолжается до тех пор, пока совершаются хотя бы одни перестановки элементов. В начале каждой итерации значение промежутка `gap` уменьшается делением на коэффициент `shrink`, при этом производится проверка на нижнюю границу: если полученное значение становится меньше единицы, оно принудительно устанавливается в `1`, чтобы

обеспечить финальный проход по соседним элементам. После этого переменная `i`, задающая индекс начала сравниваемой пары, обнуляется, а флаг `swapped` сбрасывается.

Внутренний цикл `while` организует проход по массиву: он выполняется до тех пор, пока индекс `i + gap` не выходит за пределы длины массива. На каждом шаге сравниваются элементы, находящиеся друг от друга на расстоянии `gap`. Если текущая пара расположена в неправильном порядке, происходит обмен значений, и флаг `swapped` устанавливается в `True`, сигнализируя о необходимости продолжения работы алгоритма. По завершении внутреннего цикла происходит переход к следующей итерации, при этом новый `gap` становится меньше, и алгоритм приближается к финальной стадии, аналогичной пузырьковой сортировке с минимальным шагом. В конечном итоге функция возвращает отсортированный список `arr`.

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {comb_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

В расширенной версии функции, содержащей параметр `reverse`, предоставляется возможность выбора направления сортировки. При значении `False` осуществляется упорядочивание по возрастанию, а при `True` — по убыванию. Условие сравнения внутри цикла модифицируется в зависимости от значения параметра, сохраняя при этом общую логику работы алгоритма. Это позволяет использовать одну реализацию для разных сценариев сортировки, без необходимости дублирования кода.

In [ ]:
```
def comb_sort(arr, reverse=False):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True
    while swapped:
        gap = int(gap/shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = False
        while i+gap < n:
            if (not reverse and arr[i] > arr[i+gap]) or (reverse and arr[i] < arr[i+gap]):
                arr[i], arr[i+gap] = arr[i+gap], arr[i]
                swapped = True
            i += 1
    return arr
```

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{comb_sort(arr)} — в порядке возрастания\n'
      f'\t{comb_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [64, 90, 22, 25, 34, 11, 12] — в порядке убывания
```

Сортировка выбором (извлечением)

Сортировка выбором (*Selection Sort*), известная также под названием *сортировка извлечением*, представляет собой один из базовых алгоритмов упорядочивания, суть которого заключается в последовательном извлечении наибольшего или наименьшего элемента из неотсортированной части массива и помещении его в нужную позицию в отсортированной части. В отличие от обменных алгоритмов, здесь упор делается не на множественные попарные перестановки, а на последовательный отбор значений, что позволяет сократить общее число обменов.

Алгоритм работает по следующему принципу: на каждой итерации производится поиск наибольшего элемента в оставшейся неотсортированной части массива. После его обнаружения он меняется местами с последним элементом этой части, тем самым закрепляясь на своей окончательной позиции. Далее граница неотсортированной области сдвигается на один элемент влево, и процесс повторяется, пока весь массив не будет приведён к возрастающему порядку. Аналогичным образом может быть реализован и вариант, при котором на каждой итерации находится наименьший элемент и помещается в начало массива. Содержательно оба варианта эквивалентны, различие состоит лишь в направлении сортировки.

На рис. 2 показан поэтапный процесс работы алгоритма при выборе максимального значения. На каждом шаге осуществляется просмотр всех оставшихся элементов и выбор наибольшего среди них. Выбранный элемент обменивается с последним элементом текущей неотсортированной области, после чего исключается из дальнейшего рассмотрения. Серая подсветка помогает отследить, какие элементы находятся в сравнении на текущем шаге, а стрелки и подписи фиксируют найденное максимальное значение. Этот пошаговый процесс визуально демонстрирует, как массив последовательно упорядочивается от конца к началу.

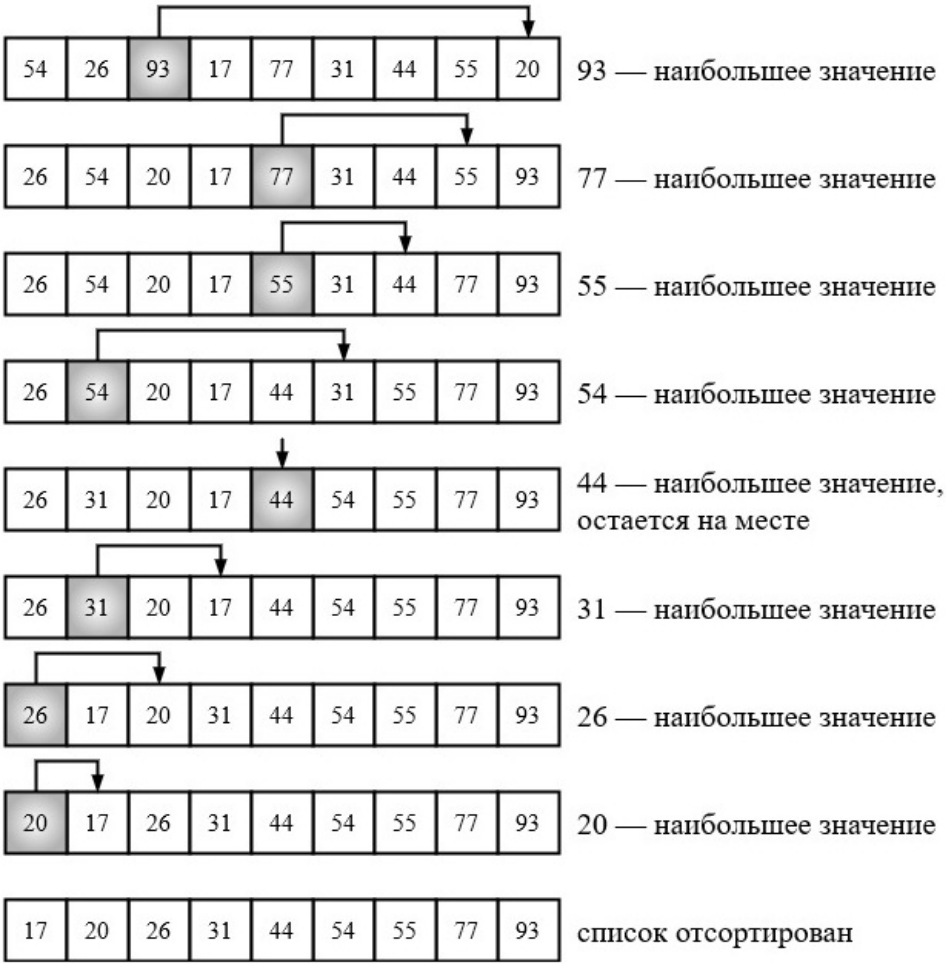


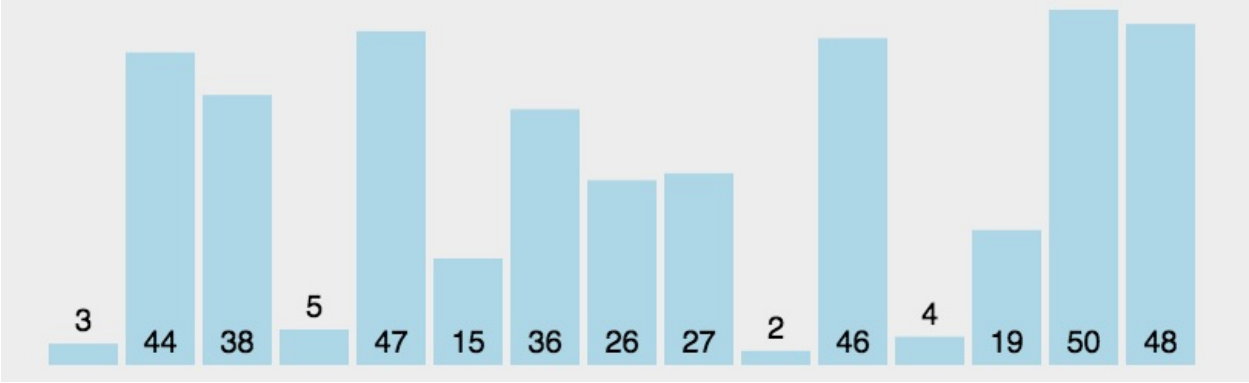
Рисунок 2 — Пример работы алгоритма сортировки выбором

С точки зрения трудоёмкости, алгоритм сортировки выбором производит $n - 1$ проход по неотсортированной части массива, каждый из которых требует поиска наибольшего элемента среди оставшихся. Это даёт общее количество операций сравнения, равное сумме $(n - 1) + (n - 2) + \dots + 1 = \dfrac{1}{2}(n^2 - n)$, что соответствует временной сложности $O(n^2)$. При этом количество самих перестановок минимально и составляет $O(n)$, поскольку на каждом шаге выполняется не более одного обмена. Это делает алгоритм выгодным в задачах, где стоимость обмена велика, а сравнение — относительно дешёвая операция.

Тем не менее, сортировка выбором имеет один существенный недостаток — неспособность эффективно обрабатывать почти отсортированные данные. В отличие от адаптивных методов, алгоритм всегда производит одинаковое количество сравнений вне зависимости от текущего состояния массива. Это делает его непригодным для сценариев, где заранее известно, что входные данные уже частично упорядочены.

Алгоритм сортировки выбором не является устойчивым: при наличии одинаковых элементов их порядок в массиве может измениться. Это связано с тем, что при обмене минимального элемента с текущей позицией порядок равных значений не сохраняется. Такая особенность может быть критичной в ситуациях, где последовательность равных элементов несёт дополнительную семантическую нагрузку, например при сортировке по вторичному критерию. В отличие от устойчивых алгоритмов, например, сортировки вставками или слиянием, метод выбора не гарантирует сохранения исходной структуры при равных значениях.

Визуализация пошагового выполнения позволяет глубже понять внутреннюю механику работы алгоритма.



Visualgo

Одним из естественных направлений улучшения алгоритма сортировки выбором является стремление сократить количество сравнений, необходимых для нахождения максимального или минимального элемента на каждом шаге. Классическая реализация предполагает полный просмотр оставшейся неотсортированной части массива, что требует всё того же порядка $O(n^2)$ сравнений, даже несмотря на минимальное количество перестановок. Однако известно, что результаты некоторых предыдущих сравнений можно использовать для ускорения поиска, особенно в ситуациях, когда требуется найти максимум или минимум не впервые.

В этом контексте особый интерес представляет стратегия ***квадратичного выбора**, *опирающаяся на группировку элементов и поэтапный отбор. Если известно, что общее число элементов n является точным квадратом, массив можно разбить на $\sqrt{n}$ групп по $\sqrt{n}$ элементов в каждой. На первом этапе в каждой группе определяется максимальный элемент — так называемый* лидер группы*. Далее из числа лидеров выбирается глобальный максимум, который и будет перемещён в конец текущей неотсортированной области. Таким образом, задача нахождения одного максимального элемента заменяется двумя этапами: сначала — $\sqrt{n}$ сравнений внутри групп, затем — $\sqrt{n} - 1$ сравнений среди лидеров. Это даёт приблизительно $2\sqrt{n}$ сравнений на каждый шаг, что позволяет значительно сократить общее количество операций.

Подобная схема эффективна при больших значениях n, где количество сравнений в классической реализации достигает десятков тысяч. Метод квадратичного выбора позволяет достичь общей временной сложности порядка $O(n\sqrt{n})$, что существенно превосходит традиционный подход. При этом важно, что количество перестановок остаётся прежним — не более одного обмена на итерацию.

Однако такая оптимизация имеет свои ограничения. Во-первых, метод требует дополнительной памяти или структуры для хранения промежуточных результатов — например, индексов лидеров групп. Во-вторых, эффективность группировки наиболее выражена при значительных объёмах данных и уменьшается для небольших массивов, где накладные расходы на организацию групп не оправдывают себя.

Несмотря на это, идея повторного использования ранее полученной информации о данных, а также введение иерархической структуры поиска, открывает перспективы для дальнейших улучшений не только в сортировке выбором, но и в более сложных алгоритмах. Такой подход близок к идее иерархического отбора, характерной для методов, основанных на принципе ***«разделяй и властвуй»***, когда исходная задача структурируется через последовательность упрощённых шагов с последующим объединением частичных результатов.

Подход с группировкой и промежуточным выбором демонстрирует, как даже простейшие алгоритмы могут быть значительно ускорены при грамотном использовании внутренних закономерностей данных и перераспределении вычислительной нагрузки.

Таким образом, сортировка выбором демонстрирует понятную и наглядную стратегию упорядочивания данных за счёт последовательного выбора крайних значений. Несмотря на ограниченную практическую применимость из-за квадратичной сложности и отсутствия адаптивности, она сохраняет свою ценность и может быть использована как основа для построения более сложных и производительных алгоритмов.

Пример реализации алгоритма сортировки выбором:

In [ ]:
```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Функция `selection_sort` принимает на вход список/массив `arr`. На первом этапе определяется длина массива и сохраняется в переменную `n`, после чего запускается внешний цикл, переменная `i` которого указывает на текущую позицию в массиве, где должен оказаться следующий минимальный элемент. В начале каждой итерации предполагается, что минимальное значение располагается на позиции `i`, и его индекс сохраняется в переменную `min_idx`.

Далее выполняется вложенный цикл по переменной `j`, который последовательно перебирает элементы, расположенные правее текущего. На каждом шаге проверяется, является ли текущий элемент `arr[j]` меньшим, чем значение на позиции `min_idx`. Если это условие выполняется, индекс минимального элемента обновляется. После завершения вложенного цикла происходит обмен между элементом на позиции `i` и элементом с индексом `min_idx`, благодаря чему на текущую позицию устанавливается минимальное значение из оставшейся части массива. В результате выполнения всех итераций массив упорядочивается по возрастанию, и функция возвращает отсортированный список.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список: {arr}\nОтсортированный список: {selection_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Расширение алгоритма за счёт параметра `reverse` делает реализацию более гибкой: оно позволяет задать направление сортировки без необходимости переписывать логику. Это полезно в ситуациях, когда требуется быстро переключиться между возрастающим и убывающим порядком в зависимости от условий задачи или пользовательского запроса. Благодаря компактной модификации условия сравнения, алгоритм сохраняет ясную структуру и легко адаптируется к различным требованиям, не усложняя общий подход к сортировке.

In [ ]:
```python
def selection_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if reverse:
                if arr[j] > arr[min_idx]:
                    min_idx = j
            else:
                if arr[j] < arr[min_idx]:
                    min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{selection_sort(arr)} — в порядке возрастания\n'
      f'\t{selection_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Ниже приведена реализация алгоритма сортировки методом квадратичного выбора по возрастанию. В этой версии массив разбивается на группы длиной, приблизительно равной корню из текущей длины неотсортированной части. На каждом шаге выбирается минимальный элемент среди лидеров групп, который затем перемещается в начало, что позволяет сократить количество сравнений по сравнению с классической сортировкой выбором.

In [ ]:
```python
import math
```

In [ ]:
```python
def quadratic_selection_sort(arr):
    n = len(arr)
    sorted_idx = 0
    while sorted_idx < len(arr) - 1:
        remaining = len(arr) - sorted_idx
        group_size = int(math.isqrt(remaining))
        num_groups = (remaining + group_size - 1) // group_size
        group_min_indices = []

        for g in range(num_groups):
            start = sorted_idx + g * group_size
            end = min(start + group_size, len(arr))
            min_idx = start
            for i in range(start + 1, end):
                if arr[i] < arr[min_idx]:
                    min_idx = i
            group_min_indices.append(min_idx)

        global_min_idx = group_min_indices[0]
        for idx in group_min_indices[1:]:
            if arr[idx] < arr[global_min_idx]:
                global_min_idx = idx

        arr[sorted_idx], arr[global_min_idx] = arr[global_min_idx], arr[sorted_idx]
        sorted_idx += 1
    return arr
```

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список: {arr}\n'
      f'Отсортированный список: {quadratic_selection_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Здесь на каждом шаге работы алгоритма остающаяся неотсортированной часть массива разбивается на группы длиной, приблизительно равной квадратному корню из текущей длины этой части. Значение размера группы определяется с использованием функции `math.isqrt`, которая возвращает целую часть квадратного корня. Далее определяется количество таких групп, которое может быть чуть больше из-за округления вверх, если длина массива не является точным квадратом.

Для каждой группы последовательно определяется индекс элемента с минимальным значением. Эти индексы сохраняются в список, который затем используется для второго этапа отбора — нахождения глобального минимума среди всех лидеров групп. Такой двухуровневый подход позволяет уменьшить общее количество сравнений, поскольку вместо линейного просмотра всей неотсортированной части массива осуществляется сначала локальный выбор внутри компактных подмножеств, а затем финальное сравнение между их представителями.

Найденный глобальный минимум обменивается с элементом, находящимся в начале текущей неотсортированной части массива. После этого граница отсортированной области сдвигается на одну позицию вправо, и процедура повторяется для оставшихся элементов. Цикл продолжается до тех пор, пока отсортированной не станет вся последовательность. Благодаря такому подходу достигается уменьшение количества операций поиска минимального значения с $O(n)$ до порядка $O(\sqrt{n})$ на каждом шаге, что в сумме даёт временную сложность порядка $O(n\sqrt{n})$. Алгоритм не требует дополнительной памяти, кроме списка индексов, и сохраняет простоту реализации при повышенной эффективности по сравнению с классической сортировкой выбором.

Сортировка вставками (включением)

Сортировка вставками (*Insertion Sort*) — это последовательный алгоритм упорядочивания, в котором каждый элемент массива поочерёдно вставляется в подходящую позицию внутри уже отсортированной части. Алгоритм имитирует способ, которым человек может упорядочить игральные карты в руке: берётся по одной карте и помещается в нужное место между ранее отсортированными. При этом сохраняется инвариант — все элементы левее текущей позиции отсортированы.

В начале считается, что первый элемент массива уже образует отсортированную последовательность. Далее каждый следующий элемент извлекается из неотсортированной части и сравнивается с элементами отсортированной области, начиная с конца. Пока текущий элемент меньше проверяемого, последние сдвигаются вправо, освобождая место для вставки. После нахождения подходящей позиции элемент помещается на своё место. Этот процесс повторяется до тех пор, пока весь массив не окажется упорядоченным.

Рис. 3 демонстрирует пошаговое формирование отсортированной части массива: каждый новый элемент извлекается из неупорядоченной области и вставляется на корректную позицию. Подписи фиксируют, какой элемент был вставлен, а серая подсветка выделяет сравниваемые элементы, показывая, как левая часть постепенно превращается в отсортированную последовательность.

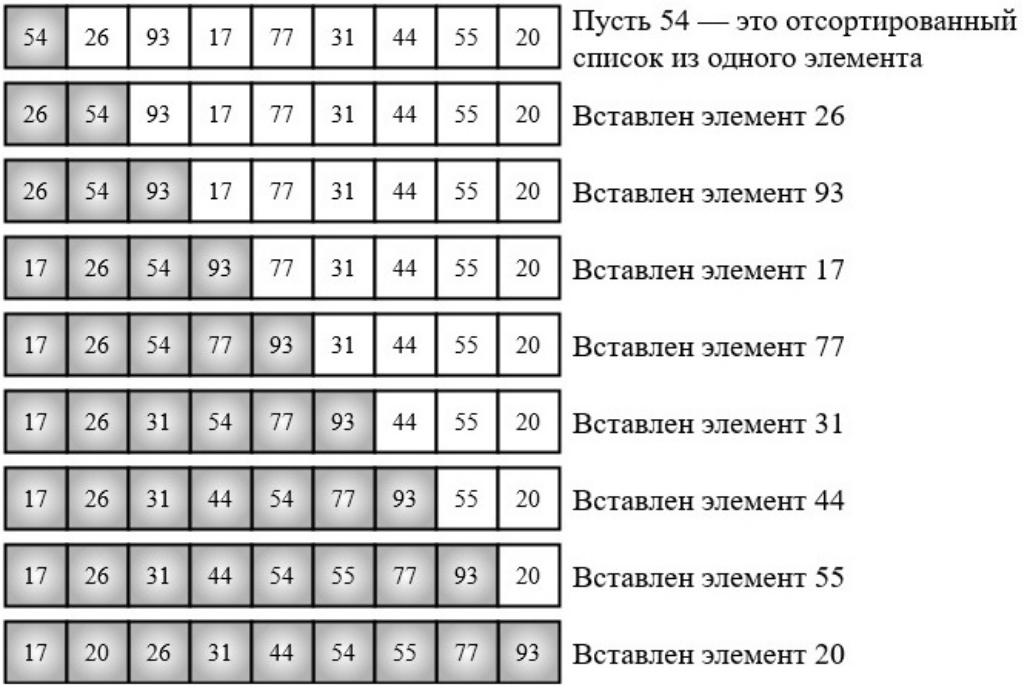


Рисунок 3 — Пример работы алгоритма сортировки вставками

На рис. 4 наглядно представлен пятый проход алгоритма, в ходе которого вставляется значение `31`. На иллюстрации видно, как `31` поочерёдно сравнивается с элементами отсортированной части массива справа налево. Каждый элемент, превышающий `31`, сдвигается вправо до тех пор, пока не будет найден первый элемент, не меньший вставляемого. После этого `31` помещается в освободившуюся позицию. Этот пример подчёркивает характерную механику работы внутреннего цикла вставки.

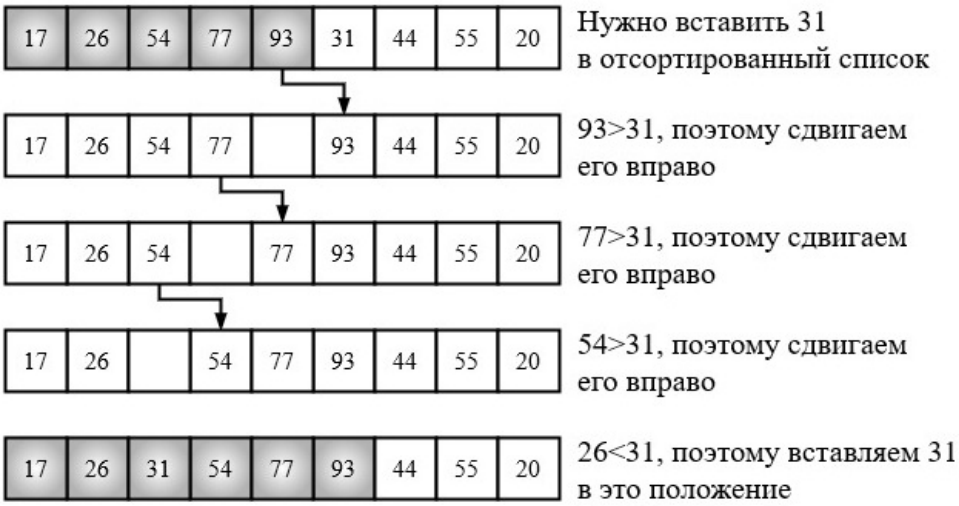
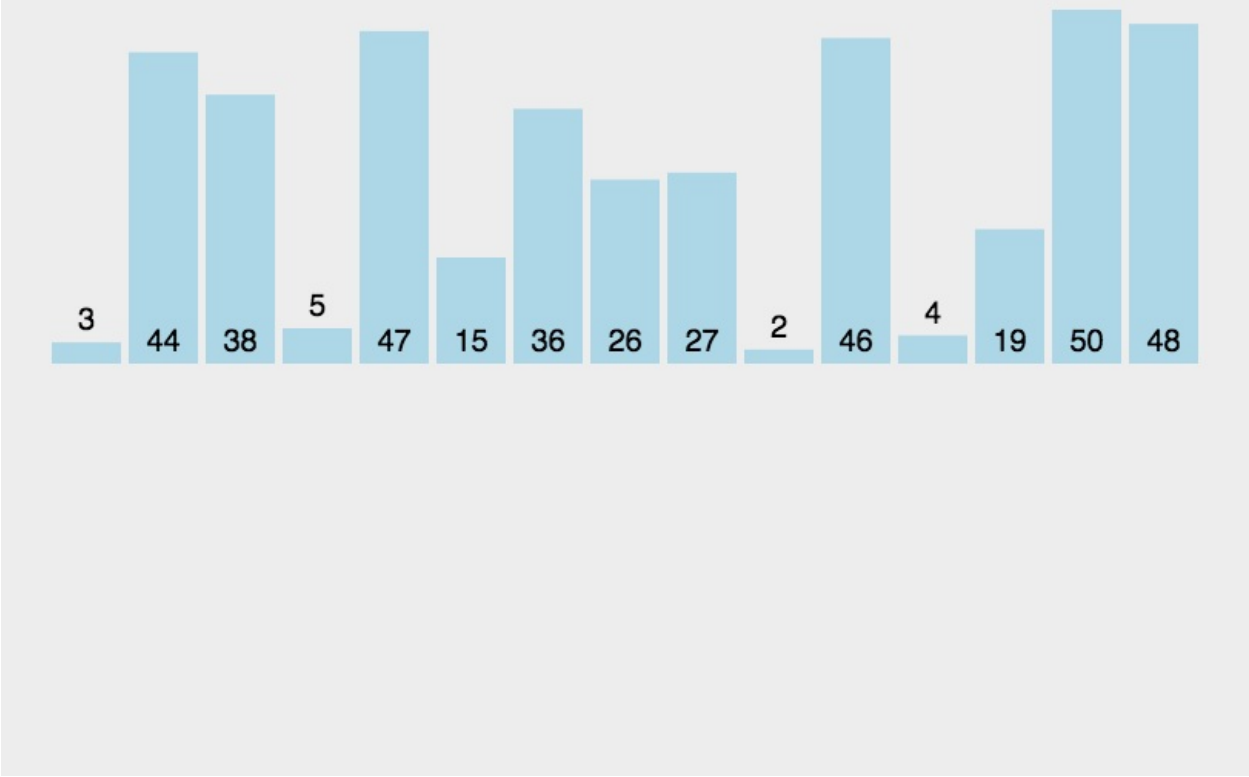


Рисунок 4 — Пример пятого прохода алгоритма сортировки вставками

С точки зрения трудоёмкости, алгоритм сортировки вставками обладает временной сложностью $O(n^2)$ в среднем и худшем случаях, что свойственно большинству простых квадратичных методов. Однако при частично или полностью отсортированных данных количество операций резко сокращается, поскольку элементы почти не сдвигаются. В таких ситуациях внутренняя сложность приближается к $O(n)$, и алгоритм проявляет адаптивность. Это выгодно отличает его от сортировки пузырьком и выбором, в которых количество сравнений не зависит от порядка входных данных.

Сортировка вставками является устойчивым алгоритмом, то есть она сохраняет порядок следования равных элементов. Это свойство важно в задачах, где одинаковые значения могут различаться по вторичным признакам, и начальный порядок имеет значение. Например, при последовательной сортировке сначала по фамилии, а затем по имени, устойчивость алгоритма позволяет сохранить внутреннюю упорядоченность без лишних перестановок.

Благодаря своей простоте, эффективности на малых объёмах данных и хорошей адаптивности, сортировка вставками часто применяется в реальных системах как часть более сложных гибридных алгоритмов.



Visualgo

Одним из направлений оптимизации сортировки вставками является сокращение количества сравнений, выполняемых при поиске позиции вставки. В классической реализации этот поиск осуществляется линейно, начиная с конца отсортированной части массива. В худшем случае это требует до n сравнений на каждом шаге, что в сумме даёт квадратичную трудоёмкость $O(n^2)$ по числу сравнений.

Однако если рассматривать отсортированную часть массива как упорядоченную последовательность, становится возможным применить ***бинарный поиск*** для определения позиции вставки. Этот подход позволяет находить нужную позицию не за $O(n)$, а за $O(\log n)$ сравнений. В результате общее число сравнений уменьшается с $O(n^2)$ до $O(n \log n)$, что заметно

на больших объёмах данных. При этом сдвиг элементов для освобождения места всё равно требует $O(n)$ операций, поэтому общая временная сложность алгоритма по-прежнему остаётся $O(n^2)$, но прирост производительности достигается именно за счёт снижения числа сравнений.

Применение бинарного поиска не делает алгоритм асимптотически быстрее по числу перемещений, но делает его эффективнее в условиях, когда сравнение — дорогая операция, а копирование — относительно дешёвое. Это может встречаться, например, при работе с объектами, где сравнение требует обращения к внешним данным или выполнению дополнительных проверок.

Важно, что модификация алгоритма не влияет на его устойчивость, так как элементы вставляются точно в позицию перед первым элементом, не меньшим текущему. Это позволяет сохранить порядок равных значений, как и в базовом варианте сортировки вставками.

Пример реализации алгоритма сортировки вставками:

In [ ]:
```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

В этом коде на каждой итерации внешнего цикла выбирается элемент `key`, который должен быть вставлен в отсортированную часть массива, расположенную левее текущей позиции. В переменной `j` сохраняется индекс последнего элемента отсортированной области, после чего запускается внутренний цикл, сдвигающий все элементы, большие `key`, на одну позицию вправо. Это создаёт свободное место для вставки элемента. Как только найдено подходящее положение (либо достигнут левый край массива, либо найден элемент не больше `key`), значение `key` помещается на позицию `j + 1`. После завершения всех итераций возвращается отсортированный массив.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список: {arr}\nОтсортированный список: {insertion_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Расширенный вариант функции предусматривает параметр `reverse`, который позволяет выбирать направление сортировки. При значении `True` выполняется сортировка по убыванию, а при `False` — по возрастанию. Логика выбора направления реализуется через условие сравнения внутри цикла, которое изменяется в зависимости от значения `reverse`, при этом структура алгоритма остаётся неизменной.

In [ ]:
```python
def insertion_sort(arr, reverse=False):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and ((not reverse and arr[j] > key) or (reverse and arr[j] < key)):
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Благодаря незначительному изменению логики сравнения сохраняется структура алгоритма и достигается универсальность. Одна и та же функция может быть использована для двух разных сценариев, что удобно при построении адаптивных решений и пользовательских интерфейсов.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{insertion_sort(arr)} — в порядке возрастания\n'
      f'\t{insertion_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Реализация сортировки вставками с использованием бинарного поиска:

In [ ]:
```python
def binary_insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        left = 0
```

```
        right = i
        while left < right:
            mid = (left + right) // 2
            if key < arr[mid]:
                right = mid
            else:
                left = mid + 1
        for j in range(i, left, -1):
            arr[j] = arr[j - 1]
        arr[left] = key
    return arr
```

В начале каждой итерации цикла `for` выбирается элемент `key`, который необходимо вставить в отсортированную часть массива, состоящую из элементов с индексами от `0` до `i - 1`. Далее запускается бинарный поиск: переменные `left` и `right` обозначают текущие границы поиска, и цикл `while` продолжается до тех пор, пока не будет найден индекс `left`, в который следует вставить `key`. В теле этого цикла вычисляется средний индекс `mid`, и в зависимости от того, больше или меньше `key`, диапазон поиска сужается соответственно влево или вправо.

После завершения бинарного поиска начинается второй цикл, сдвигающий все элементы отсортированной части, начиная с индекса `i - 1` до позиции `left`, на одну позицию вправо. Это освобождает нужную ячейку, в которую затем помещается `key`. Таким образом, обеспечивается корректная вставка элемента без нарушения порядка в отсортированной области. Цикл продолжается до тех пор, пока не будет обработан весь массив.

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```
print(f'Исходный список: {arr}\nОтсортированный список: {binary_insertion_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Эффективные методы сортировки

Быстрая сортировка

Быстрая сортировка (*Quick Sort*) — это эффективный алгоритм упорядочивания, основанный на парадигме ***«разделяй и властвуй»***. Предложенный Тони Хоаром в 1960 году, он стал одним из наиболее широко применяемых методов сортировки благодаря своей высокой производительности и простоте реализации. Алгоритм демонстрирует хорошие результаты не только в теоретических оценках, но и на практике, особенно при работе с большими массивами данных.

Принцип работы заключается в выборе одного из элементов массива в качестве ***опорного*** и последующем разбиении массива на две части: элементы, меньшие опорного, и элементы, большие либо равные ему. После этого каждая из частей сортируется рекурсивно по той же схеме. Все операции производятся на месте, без создания дополнительных массивов, за счёт обмена элементов в пределах исходной последовательности.

Результативность алгоритма во многом зависит от выбора опорного элемента. Идеальный вариант — медиана текущего диапазона, поскольку она обеспечивает равномерное разбиение и минимальную глубину рекурсии. Однако точное вычисление медианы требует затрат, поэтому на практике чаще выбираются первый, последний или случайный элемент. Одной из распространённых эвристик является медиана трёх — элемент, занимающий промежуточную позицию между первым, средним и последним значениями.

На рис. 5 показан исходный массив, в котором для первого шага алгоритма в качестве разделяющего (опорного) элемента выбран `54`, он выделен серым цветом.

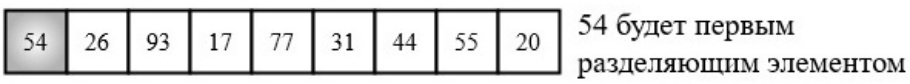


Рисунок 5 — Выбор опорного элемента при выполнении быстрой сортировки

Рис. 6 иллюстрирует пошаговый процесс разделения массива с помощью двух указателей — `leftmark` и `rightmark`, движущихся навстречу друг другу. Сначала `leftmark` смещается вправо, пока не найдёт элемент больше опорного, а `rightmark` — влево, пока не встретит элемент меньше опорного. После нахождения такой пары (`93` и `20`) они меняются местами. Далее аналогичная перестановка происходит для элементов `77` и `44`. В финальной стадии этого шага указатели «пересекаются», что означает завершение разбиения: `54` меняется местами с последним элементом левой подгруппы — `31`, определяя точку разделения массива.

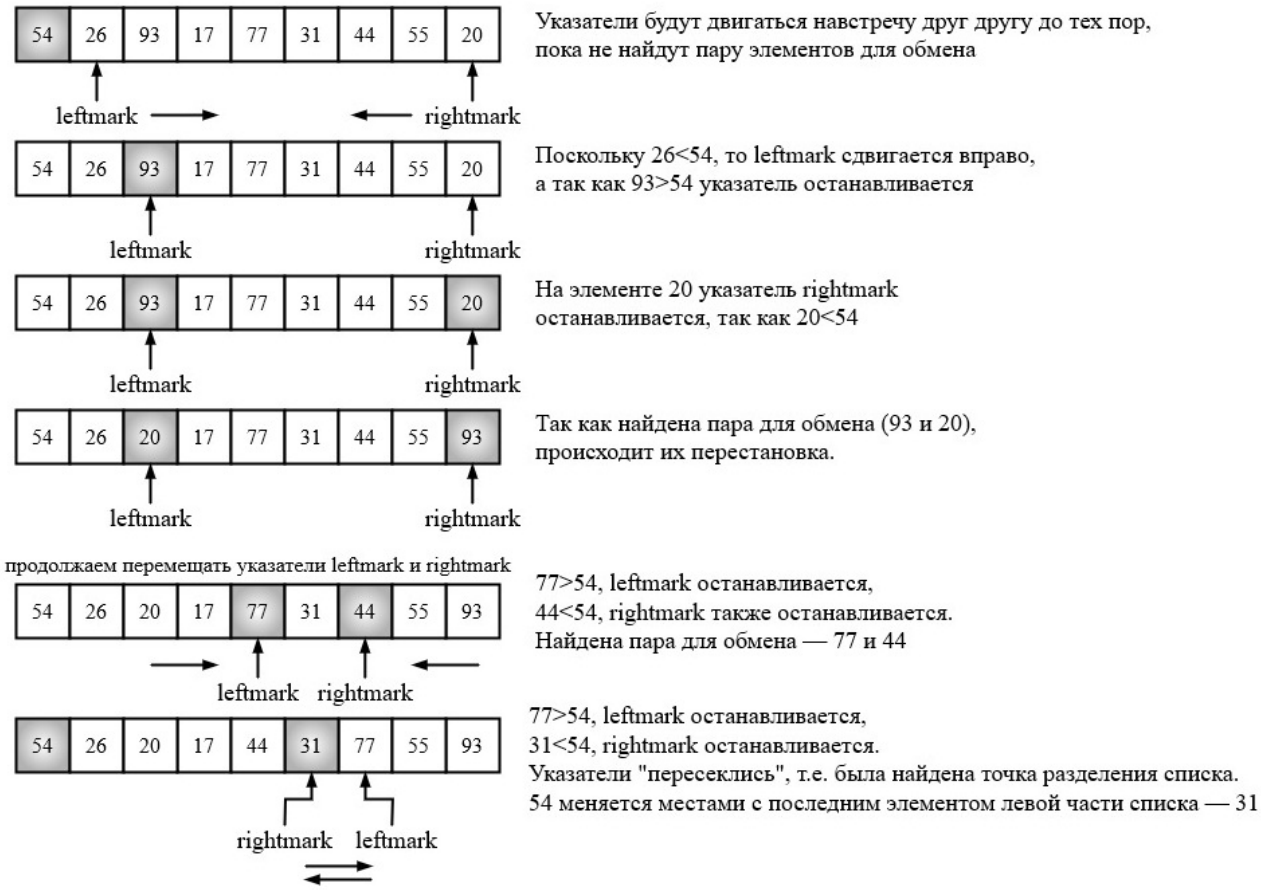


Рисунок 6 — Разбиение массива на подмассивы относительно опорного элемента

На рис. 7 проиллюстрирован переход к рекурсивной сортировке каждой из двух частей. Подписи фиксируют границы новых диапазонов, а метки обозначают, какие участки будут обработаны следующими вызовами.

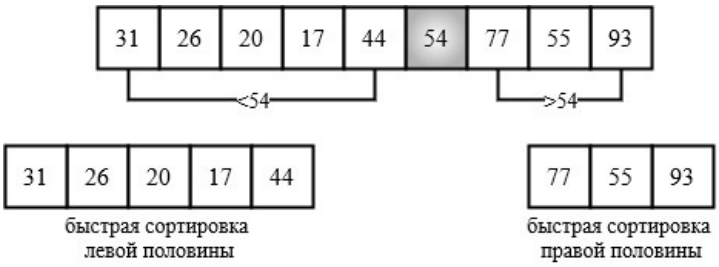
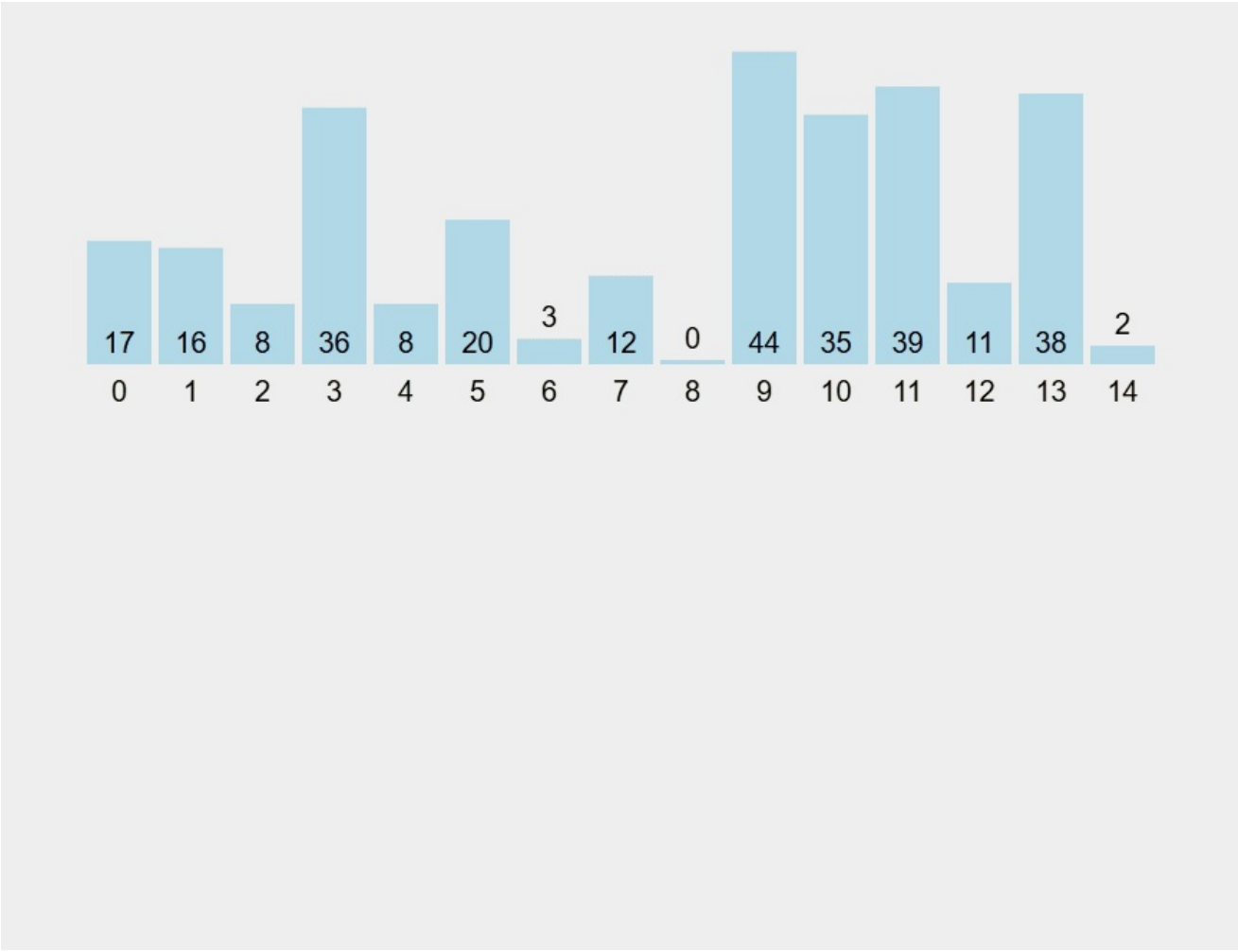


Рисунок 7 — Подготовка к рекурсивной сортировке каждого из подмассивов

Визуализация работы алгоритма позволяет наглядно проследить процесс разбиения массива, перемещения элементов и рекурсивной обработки подмассивов.



Visualgo

С точки зрения временных затрат, алгоритм быстрой сортировки обладает асимптотической сложностью $O(n \log n)$ в среднем и при благоприятном выборе опорного элемента. Такая оценка объясняется тем, что на каждой итерации происходит разбиение массива на две приблизительно равные части. Поскольку каждая операция разбиения требует линейного времени — необходимо пройти по всем элементам текущего подмассива для распределения их относительно опорного значения, — а глубина рекурсии при равномерном делении составляет $\log n$, совокупное число операций на всех уровнях оценивается как $O(n \log n)$.

Худший случай возникает тогда, когда выбор опорного элемента приводит к сильно несбалансированному разбиению, например, при выборе наименьшего или наибольшего значения в текущем диапазоне. В этом случае одна из частей оказывается пустой, а вторая включает почти все элементы массива. Глубина рекурсии в таком сценарии возрастает до n, поскольку каждый следующий вызов работает с подмножеством лишь на один элемент короче. При этом на каждом уровне также выполняется линейное число операций, что приводит к общей временной сложности $O(n^2)$. Подобное поведение характерно для уже отсортированных или почти отсортированных массивов, если опорный элемент выбирается без учёта структуры данных.

Для снижения вероятности возникновения худшего случая на практике применяются эвристики выбора опорного элемента. Одной из наиболее известных является стратегия медианы трёх, при которой в качестве опорного выбирается медиана из трёх значений: первого, среднего и последнего. Это позволяет сгладить влияние неблагоприятной структуры входных данных и приблизить поведение алгоритма к среднему случаю.

Анализ показывает, что в среднем количество сравнений при быстрой сортировке приближается к $2n \log n$, а число перестановок — порядка $n \log n$, что делает её существенно более производительной по сравнению с элементарными сортировками, такими как пузырьковая или выбором, где выполняется $O(n^2)$ операций независимо от структуры данных.

Алгоритм выполняется на месте: все действия по обмену значениями происходят в пределах исходного массива, а дополнительная память требуется лишь для поддержки рекурсивных вызовов. Объём используемой памяти при этом пропорционален глубине стека вызовов, то есть $O(\log n)$ при сбалансированных разбиениях и $O(n)$ в худшем случае. Благодаря этому быстрая сортировка особенно эффективна в системах с ограниченным объёмом оперативной памяти.

Быстрая сортировка является неустойчивым алгоритмом: при наличии равных значений их относительный порядок может нарушаться. В ситуациях, когда сохранение этого порядка критично — например, при последовательной сортировке по нескольким признакам — предпочтение отдают устойчивым алгоритмам. Выбор метода сортировки определяется не только его асимптотической сложностью, но и адаптивностью к реальным данным, устойчивостью, предсказуемостью поведения. Поэтому в ряде языков программирования применяются адаптивные и устойчивые решения. Например, в Python в функциях `sorted()`

и `list.sort()` используется устойчивый гибридный алгоритм Timsort, сочетающий преимущества сортировки вставками и слиянием и обеспечивающий стабильную производительность даже при частично упорядоченных данных.

Несмотря на это, благодаря своей логарифмической глубине, линейному числу операций на каждом уровне и способности к сортировке без выделения дополнительных массивов, быстрая сортировка считается одной из самых эффективных на практике. По этой причине она реализована в стандартных библиотеках многих языков программирования и используется как метод сортировки по умолчанию.

Рассмотрим реализацию алгоритма быстрой сортировки. В приведённой ниже функции `quick_sort()` сортировка выполняется ***на месте***, без создания дополнительных массивов. Алгоритм разбит на три части: основная функция, рекурсивный вспомогательный метод и процедура разбиения массива относительно опорного значения.

In [ ]:
```python
def quick_sort(arr):
    quick_sort_helper(arr, 0, len(arr) - 1)
    return arr

def quick_sort_helper(arr, first, last):
    if first < last:
        split_point = partition(arr, first, last)
        quick_sort_helper(arr, first, split_point - 1)
        quick_sort_helper(arr, split_point + 1, last)

def partition(arr, first, last):
    pivot_value = arr[first]
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        while left_mark <= right_mark and arr[left_mark] <= pivot_value:
            left_mark += 1
        while right_mark >= left_mark and arr[right_mark] >= pivot_value:
            right_mark -= 1
        if right_mark < left_mark:
            done = True
        else:
            arr[left_mark], arr[right_mark] = arr[right_mark], arr[left_mark]

    arr[first], arr[right_mark] = arr[right_mark], arr[first]
    return right_mark
```

Основная функция `quick_sort()` запускает алгоритм, передавая массив и границы сортировки — начальный и конечный индексы — во вспомогательную функцию `quick_sort_helper()`. Эта функция реализует рекурсивную стратегию деления: сначала проверяется, содержит ли текущий подмассив более одного элемента (условие `first < last`). Если да, вызывается функция `partition()`, которая находит позицию, в которой должен оказаться опорный элемент после разбиения. Затем выполняются два рекурсивных вызова — для левой и правой частей массива, соответственно до и после позиции опорного элемента. Таким образом, каждый уровень рекурсии занимается сортировкой всё меньших подмассивов, пока не останутся только участки длиной `1`, которые по определению считаются отсортированными.

Функция `partition()` отвечает за ключевую операцию алгоритма — разбиение массива относительно опорного значения. В качестве опорного (`pivot_value`) выбирается первый элемент текущего подмассива. Далее устанавливаются два указателя: `left_mark`, начинающийся сразу после опорного, и `right_mark`, указывающий на последний элемент. Указатели двигаются навстречу друг другу: `left_mark` ищет первый элемент, который больше опорного, а `right_mark` — первый элемент, меньший или равный опорному (в пределах текущего диапазона). Как только находятся такие элементы, они обмениваются местами, обеспечивая постепенное перемещение меньших значений влево, а больших — вправо. Этот процесс продолжается до тех пор, пока указатели не пересекутся.

Когда указатели пересекаются, условие выхода из цикла `while not done` выполняется. После этого происходит финальный обмен: опорный элемент (`arr[first]`) меняется местами с элементом, на который указывает `right_mark`. Таким образом, опорный элемент оказывается точно в той позиции, где он должен находиться в окончательно отсортированном массиве. Эта позиция возвращается как точка разбиения (`split_point`) и используется для рекурсивной сортировки подмассивов слева и справа от опорного.

Такой подход обеспечивает сортировку *на месте*, без создания дополнительных массивов, и сохраняет ключевую особенность быстрой сортировки — эффективную работу на больших объёмах данных при логарифмической глубине рекурсии.

Пример использования:

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список: {arr}\nОтсортированный список: {quick_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Расширим реализацию, добавив параметр `reverse`, позволяющий управлять направлением сортировки. При значении `False` выполняется сортировка по возрастанию, при `True` — по убыванию. Логика изменения направления реализуется через условия в циклах поиска позиции перестановки.

In [ ]:
```python
def quick_sort(arr, reverse=False):
    quick_sort_helper(arr, 0, len(arr) - 1, reverse)
    return arr

def quick_sort_helper(arr, first, last, reverse):
    if first < last:
        split_point = partition(arr, first, last, reverse)
        quick_sort_helper(arr, first, split_point - 1, reverse)
        quick_sort_helper(arr, split_point + 1, last, reverse)

def partition(arr, first, last, reverse):
    pivot_value = arr[first]
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        if not reverse:
            while left_mark <= right_mark and arr[left_mark] <= pivot_value:
                left_mark += 1
            while right_mark >= left_mark and arr[right_mark] >= pivot_value:
                right_mark -= 1
        else:
            while left_mark <= right_mark and arr[left_mark] >= pivot_value:
                left_mark += 1
            while right_mark >= left_mark and arr[right_mark] <= pivot_value:
                right_mark -= 1

        if right_mark < left_mark:
            done = True
        else:
            arr[left_mark], arr[right_mark] = arr[right_mark], arr[left_mark]

    arr[first], arr[right_mark] = arr[right_mark], arr[first]
    return right_mark
```

Благодаря добавлению переменной `reverse` и изменению условий сравнения, алгоритм адаптируется к двум направлениям сортировки без изменения основной структуры. Это делает реализацию универсальной и удобной в применении.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{quick_sort(arr)} — в порядке возрастания\n'
      f'\t{quick_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Сортировка Шелла

Сортировка Шелла (*Shell Sort*) — это алгоритм, представляющий собой обобщение сортировки вставками, предложенное Дональдом Шеллом в 1959 году. В отличие от классического варианта, где сравниваются только соседние элементы, сортировка Шелла использует промежуточные расстояния (шаги), позволяющие сравнивать и перемещать элементы, удалённые друг от друга. Такой подход ускоряет выравнивание элементов, находящихся далеко от своих финальных позиций, и тем самым сокращает общее число перестановок.

Ключевая идея алгоритма заключается в поэтапной сортировке элементов с постепенно уменьшающимся интервалом `gap`. На первых стадиях производится упорядочивание элементов, находящихся далеко друг от друга, что позволяет устранить грубые нарушения порядка. На завершающем этапе, когда шаг становится равным `1`, выполняется сортировка вставками, аналогичная классическому алгоритму, но действующая значительно эффективнее благодаря предварительной расстановке элементов ближе к их целевым позициям. Алгоритм не требует дополнительной памяти, работает на месте и демонстрирует хорошую производительность на частично отсортированных массивах.

Сложность алгоритма зависит от выбранной последовательности шагов. Наиболее простая последовательность — деление длины массива пополам на каждом этапе (например, $n/2, n/4, n/8, \ldots, 1$). При этом худшая временная сложность составляет $O(n^2)$. Однако при использовании более сложных последовательностей шагов, предложенных Седжвиком, Хиббардом и другими, удаётся добиться средней сложности порядка $O(n \log^2 n)$, а в некоторых случаях — близкой к $O(n \log n)$.

На рис. 8 представлено разбиение исходного массива на три подсписка с шагом `3`. Первый подсписок включает элементы с

индексами `0`, `3` и `6`, второй — с индексами `1`, `4` и `7`, третий — с индексами `2`, `5` и `8`. Такое разбиение позволяет на начальном этапе устранить глобальные нарушения порядка между удалёнными элементами.

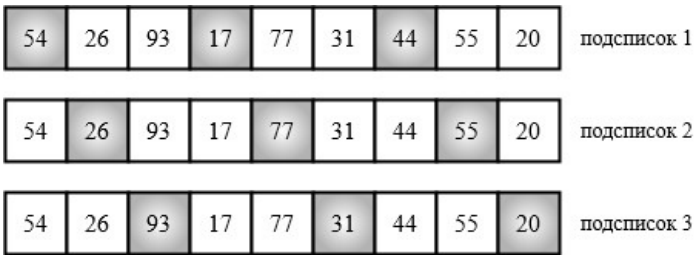


Рисунок 8 — Разбиение массива на подсписки с шагом 3

На рис. 9 показан результат сортировки каждого подсписка по отдельности. В каждом из них применяется сортировка вставками. После этого подсписки интегрируются обратно в исходный массив, при этом он ещё не отсортирован полностью, но структура уже приближена к финальному порядку.

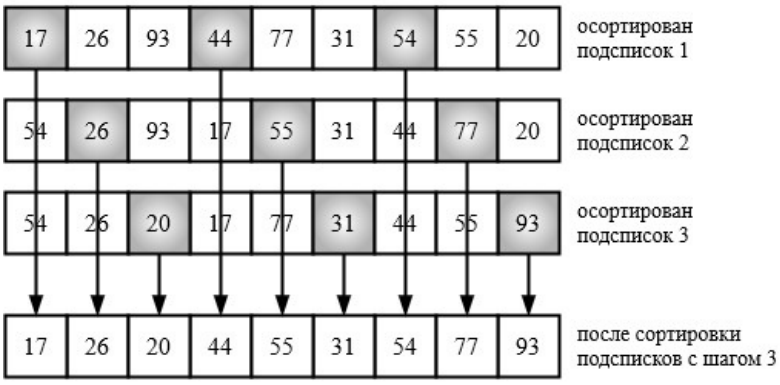


Рисунок 9 — Состояние массива после сортировки подсписков с шагом 3

Рис. 10 иллюстрирует финальную стадию сортировки, когда шаг равен `1`. Элементы упорядочиваются с помощью обычной сортировки вставками. На примере показано, как каждый элемент (например, `20`, `31`, `54`) перемещается в нужную позицию за счёт последовательных перестановок, завершая процесс сортировки массива.

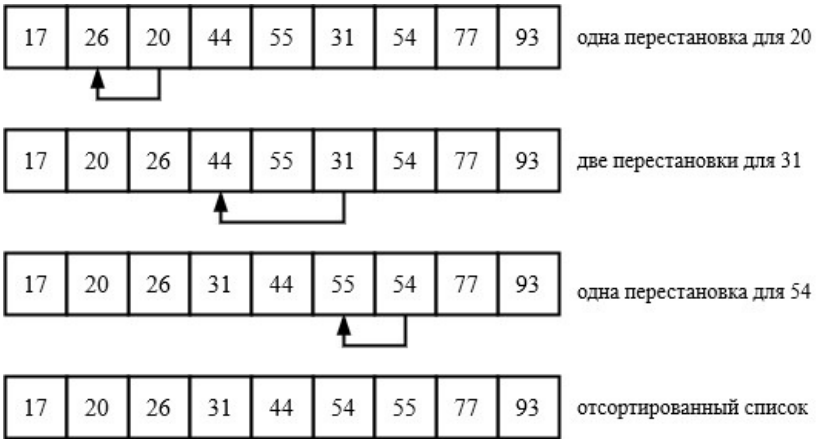


Рисунок 10 — Заключительная сортировка массива с шагом 1

На рис. 11 приведён пример альтернативного варианта: разбиение массива на четыре подсписка с шагом `4`. Формируются четыре группы, в каждой из которых элементы расположены через `4` позиции. Такое разбиение даёт иной характер промежуточной упорядоченности и демонстрирует влияние выбранного шага на процесс сортировки.

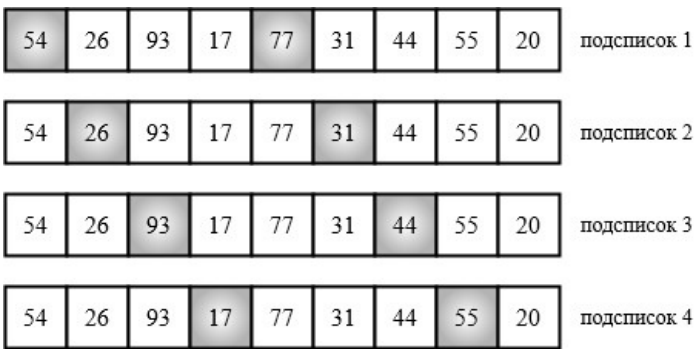


Рисунок 11 — Разбиение массива на подсписки с шагом 4

Ниже представлена базовая реализация алгоритма сортировки Шелла. Алгоритм начинается с определения расстояния между сравниваемыми элементами — начальное значение `gap` устанавливается равным половине длины массива. На каждом этапе `gap` уменьшается вдвое, и внутри цикла происходит сортировка элементов, отстоящих друг от друга на это расстояние, с использованием модифицированного алгоритма вставок.

In [ ]:
```python
def shell_sort(arr):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

На каждом шаге внешний цикл определяет текущий шаг `gap`, а внутренние циклы выполняют перестановки элементов, отстоящих на это расстояние. Если элемент в позиции `j - gap` превышает временное значение `temp`, то происходит сдвиг вправо и обновление позиции `j`. Таким образом, массив постепенно упорядочивается сначала на больших расстояниях, затем на всё меньших, вплоть до заключительного прохода при `gap = 1`, что аналогично классической сортировке вставками, но выполняется значительно быстрее из-за предварительного устранения глобальных нарушений порядка.

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список: {arr}\nОтсортированный список: {shell_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Поддержка параметра `reverse` расширяет возможности алгоритма, позволяя использовать единую функцию как для сортировки по возрастанию, так и по убыванию. Такая универсальность полезна в задачах, связанных с анализом данных, построением визуализаций или реализацией пользовательских интерфейсов, где необходимо быстро переключаться между направлениями сортировки. Благодаря лаконичной адаптации условия сравнения, функция остаётся компактной, предсказуемой и легко интегрируемой в разнообразные прикладные сценарии.

In [ ]:
```python
def shell_sort(arr, reverse=False):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and ((not reverse and arr[j - gap] > temp)
                                or (reverse and arr[j - gap] < temp)):
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

In [ ]:
```python
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```python
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{shell_sort(arr)} — в порядке возрастания\n'
      f'\t{shell_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Результаты экспериментального сравнения количества перестановок элементов при сортировке случайных массивов представлены в таблице ниже. Видно, что сортировка Шелла обеспечивает существенно меньшее число перемещений по сравнению с классической сортировкой вставками при увеличении размерности входных данных.

Размер массива	Сортировка Шелла	Сортировка вставками
50	7 700	150
2 100 000	240 000	2,5 млрд

Эти данные подчёркивают практическую эффективность алгоритма Шелла, особенно на больших массивах, и объясняют его

востребованность в приложениях, где критично количество операций перемещения.

Многие алгоритмы, применяемые для решения сложных задач, обладают рекурсивной природой. Их структура предполагает многократный вызов самих себя с модифицированными входными данными, что позволяет постепенно сократить исходную задачу до более простых составляющих. Разработка таких алгоритмов осуществляется по методу ***декомпозиции***, или пошагового разбиения задачи. Этот метод строится на следующих принципах:

- исходная задача делится на несколько подзадач меньшего размера, аналогичных исходной по структуре;
- каждая из подзадач решается независимо, зачастую с использованием рекурсии;
- полученные решения объединяются в единое решение исходной задачи.

Основу этой стратегии составляет парадигма ***«разделяй и властвуй»***, в рамках которой каждый уровень рекурсии включает три этапа:

1. ***Разделение*** задачи на несколько частей.
2. ***Покорение*** — рекурсивное решение каждой из подзадач, либо непосредственное — при достижении минимального объёма.
3. ***Комбинирование*** — сборка финального результата из частичных решений.

Такая организация вычислений лежит, в частности, в основе быстрой сортировки, сортировки слиянием, ряда алгоритмов поиска и других методов, демонстрирующих высокую эффективность на больших объёмах данных.

Сортировка слиянием

Сортировка слиянием (*Merge Sort*) — это классический алгоритм упорядочивания, основанный на парадигме ***«разделяй и властвуй»***. Он представляет собой строго рекурсивный метод, в котором решение исходной задачи строится путём последовательного разбиения массива на части, рекурсивной сортировки этих частей и последующего их слияния. Благодаря стабильной эффективности и устойчивости сортировка слиянием широко применяется как в теории алгоритмов, так и в промышленных реализациях.

Алгоритм состоит из трёх логических этапов. На первом этапе происходит ***разделение*** массива: каждый подмассив делится пополам до тех пор, пока не будут получены последовательности длины один. Эти элементы считаются тривиально отсортированными. Затем начинается этап ***покорения**, в ходе которого рекурсивно сортируются все образовавшиеся подмассивы. После этого выполняется* **слияние*** — объединение двух отсортированных последовательностей в одну. Эта операция требует линейного времени от длины сливаемых массивов и производится с сохранением устойчивости: элементы с одинаковыми значениями сохраняют свой относительный порядок.

На рис. 12 схематически показан процесс разбиения массива. Исходный список из девяти элементов последовательно делится пополам до получения одноэлементных подмассивов. Визуализация построена в виде дерева: каждая вершина соответствует подмножеству исходного массива, а ребро — шагу декомпозиции. Стрелки указывают направление разбиения, подчёркивая иерархическую природу рекурсии.

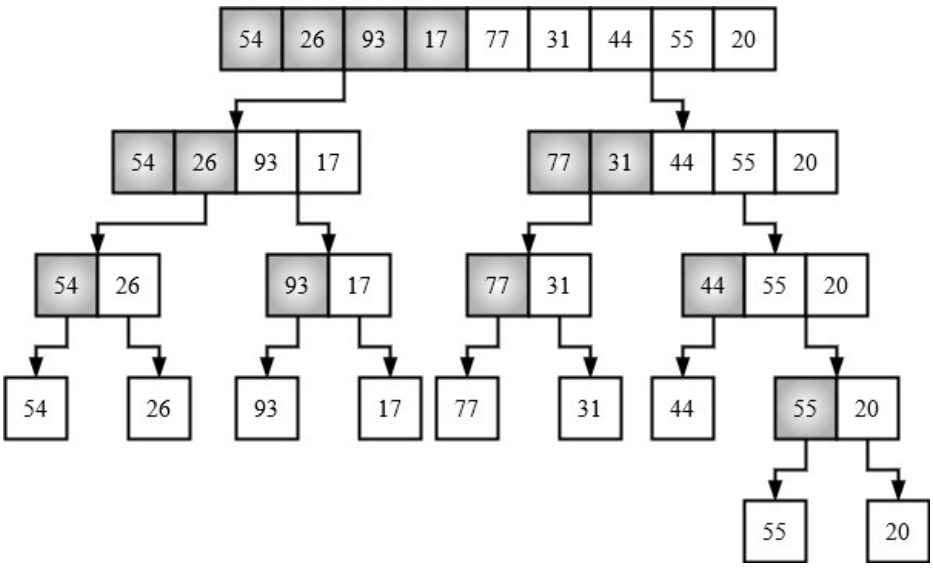


Рисунок 12 — Разбиение массива на элементарные подмассивы

На рис. 13 представлен этап слияния. Два отсортированных массива объединяются в один с помощью пошагового сравнения их элементов. Из каждой пары выбирается минимальный (или максимальный — при сортировке по убыванию) и добавляется в результирующий массив. Процесс продолжается до тех пор, пока не будут исчерпаны все элементы обоих подмассивов.

Итоговый список содержит все значения в отсортированном порядке. Визуализация также представлена в виде дерева, но с обратным направлением — от нижнего уровня к верхнему, показывая, как результат поднимается к корню.

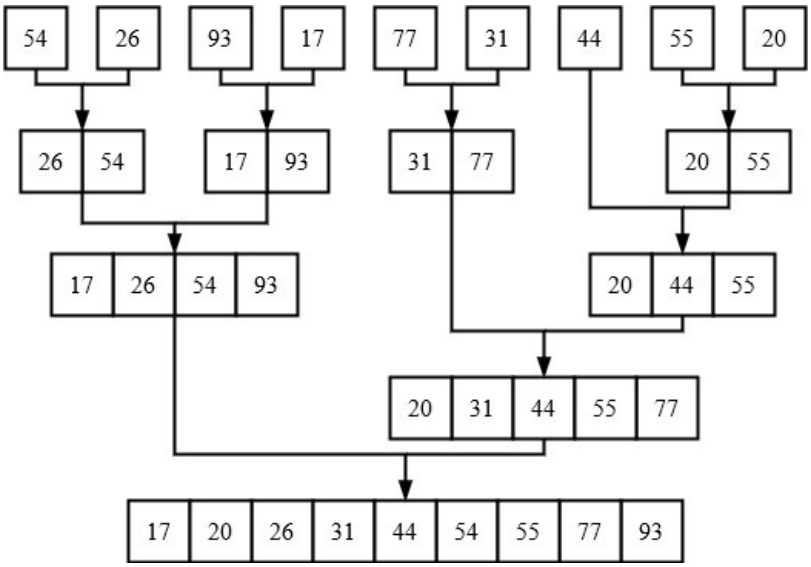
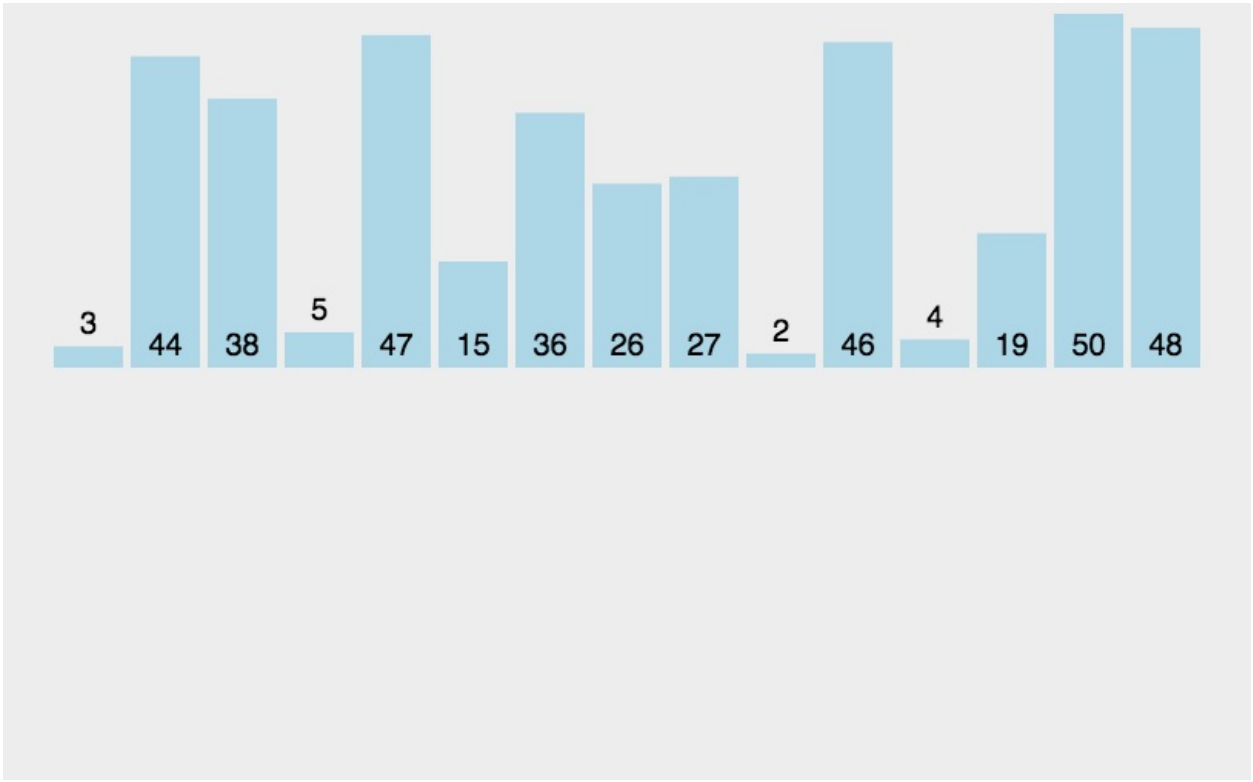


Рисунок 13 — Объединение отсортированных подмассивов в процессе сортировки

Сортировка слиянием относится к числу алгоритмов с гарантированной асимптотической эффективностью. Независимо от характера входных данных — будь то случайный, отсортированный или упорядоченный в обратном порядке массив — алгоритм стабильно демонстрирует временную сложность $O(n \log n)$. Это обусловлено структурой рекурсивного разбиения: исходная последовательность длины n делится на два подмассива приблизительно равного размера, что соответствует построению двоичного дерева глубины $\log n$. На каждом уровне этой рекурсивной схемы осуществляется слияние всех подмассивов, охватывающее в совокупности все n элементов.

Таким образом, совокупная трудоёмкость всех уровней даёт $O(n \log n)$. В отличие от быстрой сортировки, эффективность которой может существенно снижаться при неудачном выборе опорных значений, поведение сортировки слиянием не зависит от распределения данных. Это делает её предпочтительной в тех случаях, где требуется строгое соблюдение верхней оценки времени работы. Помимо устойчивости, которая обеспечивает сохранение относительного порядка равных значений, алгоритм отличается высокой пригодностью к параллельному исполнению. Благодаря независимости операций на разных ветвях рекурсивного дерева, сортировка слиянием может быть эффективно распределена между несколькими потоками или вычислительными узлами, что важно при обработке больших объёмов данных.

Ниже представлена визуализация сортировки слиянием, иллюстрирующая процесс разбиения и слияния массива на каждом уровне рекурсии.



Visualgo

Рассмотрим базовую реализацию алгоритма сортировки слиянием, в которой последовательно реализованы все этапы, соответствующие парадигме «разделяй и властвуй». Главная функция `merge_sort()` отвечает за рекурсивное разбиение массива, а вспомогательная функция `merge()` — за корректное объединение отсортированных подмассивов.

In [ ]:
```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left_half, right_half):
    result = []
    i = 0
    j = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] <= right_half[j]:
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
            j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
```

Алгоритм начинается с проверки длины массива. Если он содержит не более одного элемента, то он уже считается отсортированным, и дальнейшая обработка не требуется. Это и есть базовый случай рекурсии, на котором происходит завершение вложенных вызовов.

Если длина массива превышает один, он делится на две равные (или почти равные) части: `left_half` и `right_half`. Деление осуществляется с помощью индексной операции среза. Для каждой из этих частей рекурсивно вызывается функция `merge_sort()`, и на выходе получаются отсортированные левые и правые подмассивы. Затем вызывается функция `merge()`, объединяющая два отсортированных списка в один, упорядоченный по возрастанию.

Функция `merge()` реализует пошаговое объединение двух отсортированных списков. Создаётся пустой список `result`, в который по одному добавляются наименьшие из оставшихся элементов в подмассивах `left_half` и `right_half`. Два счётчика, `i` и `j`, отслеживают текущую позицию в соответствующих списках. На каждой итерации цикла `while` происходит сравнение элементов, и в `result` добавляется либо элемент из левого, либо из правого подмассива, в зависимости от результата сравнения.

Как только один из подмассивов исчерпывается (то есть счётчик доходит до конца), оставшиеся элементы другого подмассива просто добавляются в конец результирующего списка. Это корректно, поскольку оставшиеся элементы уже упорядочены.

На выходе будет получен новый отсортированный список. Следует отметить, что реализация не изменяет исходный массив, а возвращает новый отсортированный объект, что делает её безопасной при использовании в различных контекстах, где важна неизменяемость исходных данных.

Для расширения функциональности сортировки предусмотрена модификация, позволяющая выполнять упорядочивание в прямом или обратном порядке с помощью параметра `reverse`. Это даёт возможность динамически управлять направлением сортировки, не изменяя общую логику алгоритма.

In [ ]:
```python
def merge_sort(arr, reverse=False):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half, reverse=reverse)
    right_half = merge_sort(right_half, reverse=reverse)

    return merge(left_half, right_half, reverse=reverse)

def merge(left_half, right_half, reverse=False):
    result = []
    i = 0
```

```
    j = 0

    while i < len(left_half) and j < len(right_half):
        if (not reverse and left_half[i] <= right_half[j]) or (reverse and left_half[i] >= right_half[j]):
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
            j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
```

In [ ]:
```
arr = [64, 34, 25, 12, 22, 11, 90]
```

In [ ]:
```
print(f'Исходный список:\n\t{arr}\n\nОтсортированный список:\n'
      f'\t{merge_sort(arr)} — в порядке возрастания\n'
      f'\t{merge_sort(arr, reverse=True)} — в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] — в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] — в порядке убывания
```

Сравнение различных сортировок

Существует множество алгоритмов сортировки, различающихся как по принципу действия, так и по вычислительной трудоёмкости. Их выбор зависит от структуры входных данных, требований к устойчивости, доступной памяти и предполагаемого размера массива. Ниже приведён обзор наиболее известных алгоритмов, классифицированных по методу сортировки и средней асимптотической сложности.

Обменные алгоритмы сортировки (в том числе простая обменная сортировка, шейкерная сортировка и сортировка расчёской) основаны на последовательном сравнении и обмене соседних элементов до достижения упорядоченности. Эти методы просты в реализации, но обладают квадратичной трудоёмкостью $O(n^2)$, что делает их неэффективными при увеличении объёма данных. Используются преимущественно в учебных целях или при сортировке очень малых массивов.

Сортировка выбором реализует иной подход: на каждой итерации находится минимальный (или максимальный) элемент среди оставшихся и перемещается в нужную позицию. Несмотря на меньшее число обменов по сравнению с обменными методами, данный алгоритм также имеет временную сложность $O(n^2)$ и не демонстрирует существенных преимуществ при увеличении размера входных данных.

Сортировка вставками предполагает поэлементное построение отсортированной части массива: каждый новый элемент вставляется в уже упорядоченный фрагмент на соответствующую позицию. Хотя её худшая сложность также составляет $O(n^2)$, алгоритм показывает хорошую производительность на малых массивах или частично отсортированных данных. Благодаря своей простоте и стабильности он часто применяется в гибридных алгоритмах для обработки малых подмассивов.

Сортировка Шелла представляет собой обобщение сортировки вставками, в котором используется серия сравнений и вставок с убывающим шагом между элементами. Эффективность алгоритма напрямую зависит от выбранной последовательности шагов. При использовании оптимальных (например, последовательности Седжвика) можно достичь трудоёмкости, близкой к $O(n \log^2 n)$, что делает этот алгоритм значительно более производительным на больших массивах по сравнению с обычной сортировкой вставками.

Быстрая сортировка — один из наиболее эффективных универсальных методов. При удачном выборе опорных элементов она демонстрирует среднюю временную сложность $O(n \log n)$. Алгоритм выполняется на месте, обладает хорошей производительностью и адаптируем к различным входным данным. Однако в худшем случае (например, при неудачном разбиении массива) его сложность возрастает до $O(n^2)$. На практике применяется множество эвристик, позволяющих избежать таких сценариев и обеспечить стабильную эффективность.

Сортировка слиянием представляет собой устойчивый алгоритм, работающий с гарантированной сложностью $O(n \log n)$ вне зависимости от структуры входных данных. В отличие от быстрой сортировки, она требует дополнительной памяти, но сохраняет стабильный порядок одинаковых элементов и легко распараллеливается, что делает её подходящей для надёжной обработки больших объёмов информации.

Для оценки практического поведения алгоритмов важно учитывать не только асимптотику, но и фактическое число операций. Например, при $n = 1000$ сортировка выбором выполнит порядка 10^6 сравнений, тогда как быстрая сортировка в среднем — лишь около 10^4. Этот порядок различий существенно влияет на время выполнения программ.

Сводная таблица по сложности и применимости алгоритмов:

Алгоритм	Средняя сложность	Худшая сложность	Устойчивость	Использование
Простая обменная	$O(n^2)$	$O(n^2)$	Нет	Учебные примеры, малые массивы
Сортировка выбором	$O(n^2)$	$O(n^2)$	Нет	Простые реализации
Сортировка вставками	$O(n^2)$	$O(n^2)$	Да	Малые или частично упорядоченные массивы
Сортировка Шелла	$O(n \log^2 n)$	До $O(n^2)$	Нет	Средние и большие массивы
Быстрая сортировка	$O(n \log n)$	$O(n^2)$	Нет	Стандартный выбор в системных библиотеках
Сортировка слиянием	$O(n \log n)$	$O(n \log n)$	Да	Большие массивы, параллельные вычисления

На практике, для массивов размером до нескольких сотен элементов, предпочтение часто отдаётся сортировке вставками или Шелла. Для массивов средней и большой размерности (от 10^3 и выше) используются быстрая сортировка и сортировка слиянием — в зависимости от требований к устойчивости и доступной памяти. Сортировка Шелла может выступать промежуточным решением, демонстрируя баланс между производительностью и простотой реализации.

Таким образом, понимание свойств каждого алгоритма позволяет не только выбрать оптимальный метод под конкретную задачу, но и эффективно комбинировать различные подходы для достижения наилучшей производительности.

Пример решения задачи

Для организации сравнения времени выполнения различных алгоритмов сортировки создаётся пользовательский класс `Sorting`, включающий статические методы, реализующие восемь классических подходов к сортировке. Каждый метод принимает в качестве аргументов список `arr` и булевый параметр `reverse`, определяющий направление сортировки: по возрастанию (по умолчанию) или по убыванию.

Применение статических методов оправдано тем, что все алгоритмы сортировки реализуются независимо от состояния конкретного экземпляра класса. Это позволяет вызывать их напрямую через имя класса, без создания объекта, снижая накладные расходы на память и ускоряя выполнение.

В классе реализованы следующие алгоритмы: простая обменная сортировка (`bubble_sort`), шейкерная сортировка (`cocktail_sort`), сортировка расчёской (`comb_sort`), сортировка выбором (`selection_sort`), сортировка вставками (`insertion_sort`), быстрая сортировка (`quick_sort`), сортировка Шелла (`shell_sort`) и сортировка слиянием (`merge_sort`).

Для измерения времени выполнения используется декоратор `measure_time`, принимающий на вход функцию `func` и возвращающий обёртку `wrapper`. Обёртка фиксирует момент начала и завершения выполнения функции, рассчитывает продолжительность работы и выводит её на экран. Внутри `wrapper` используется модуль `time`, а форматированный вывод обеспечивает точность до шестого знака после запятой. Аргументы `*args` и `**kwargs` обеспечивают универсальность декоратора, позволяя применять его к любым функциям с произвольными параметрами.

Метод `sort` реализует единый интерфейс вызова алгоритмов. Он принимает список, наименование метода сортировки и параметр `reverse`, после чего вызывает соответствующий статический метод. Декоратор `measure_time` применяется к `sort`, обеспечивая автоматическое измерение времени выполнения любого выбранного алгоритма.

In [ ]:
```python
import time


class Sorting:
    """
    Класс содержит статические методы реализации различных алгоритмов сортировки,
    а также декоратор для измерения времени выполнения и универсальный интерфейс вызова.
    """

    @staticmethod
    def bubble_sort(arr, reverse=False):
        """Реализация сортировки пузырьком."""
        n = len(arr)
        for i in range(n):
            for j in range(n - i - 1):
                if (not reverse and arr[j] > arr[j + 1]) or (reverse and arr[j] < arr[j + 1]):
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
        return arr

    @staticmethod
    def cocktail_sort(arr, reverse=False):
        """Реализация шейкерной сортировки."""
        n = len(arr)
        start = 0
        end = n - 1
        swapped = True
        while swapped:
```

```
            swapped = False
            for i in range(start, end):
                if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                    arr[i], arr[i + 1] = arr[i + 1], arr[i]
                    swapped = True
            if not swapped:
                break
            swapped = False
            end -= 1
            for i in range(end - 1, start - 1, -1):
                if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                    arr[i], arr[i + 1] = arr[i + 1], arr[i]
                    swapped = True
            start += 1
        return arr

    @staticmethod
    def comb_sort(arr, reverse=False):
        """Реализация сортировки расчёской."""
        n = len(arr)
        gap = n
        shrink = 1.3
        swapped = True
        while swapped:
            gap = int(gap / shrink)
            if gap < 1:
                gap = 1
            i = 0
            swapped = False
            while i + gap < n:
                if (not reverse and arr[i] > arr[i + gap]) or (reverse and arr[i] < arr[i + gap]):
                    arr[i], arr[i + gap] = arr[i + gap], arr[i]
                    swapped = True
                i += 1
        return arr

    @staticmethod
    def selection_sort(arr, reverse=False):
        """Реализация сортировки выбором."""
        n = len(arr)
        for i in range(n):
            min_idx = i
            for j in range(i + 1, n):
                if (not reverse and arr[j] < arr[min_idx]) or (reverse and arr[j] > arr[min_idx]):
                    min_idx = j
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
        return arr

    @staticmethod
    def insertion_sort(arr, reverse=False):
        """Реализация сортировки вставками."""
        for i in range(1, len(arr)):
            key = arr[i]
            j = i - 1
            while j >= 0 and ((not reverse and arr[j] > key) or (reverse and arr[j] < key)):
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr

    @staticmethod
    def quick_sort(arr, reverse=False):
        """Рекурсивная быстрая сортировка с разбиением и выбором первого элемента в качестве опорного."""
        Sorting._quick_sort_helper(arr, 0, len(arr) - 1, reverse)
        return arr

    @staticmethod
    def _quick_sort_helper(arr, first, last, reverse):
        if first < last:
            split_point = Sorting._partition(arr, first, last, reverse)
            Sorting._quick_sort_helper(arr, first, split_point - 1, reverse)
            Sorting._quick_sort_helper(arr, split_point + 1, last, reverse)

    @staticmethod
    def _partition(arr, first, last, reverse):
        pivot_value = arr[first]
        left_mark = first + 1
        right_mark = last
        done = False
        while not done:
            if not reverse:
                while left_mark <= right_mark and arr[left_mark] <= pivot_value:
                    left_mark += 1
```

```
                while right_mark >= left_mark and arr[right_mark] >= pivot_value:
                    right_mark -= 1
            else:
                while left_mark <= right_mark and arr[left_mark] >= pivot_value:
                    left_mark += 1
                while right_mark >= left_mark and arr[right_mark] <= pivot_value:
                    right_mark -= 1
            if right_mark < left_mark:
                done = True
            else:
                arr[left_mark], arr[right_mark] = arr[right_mark], arr[left_mark]
        arr[first], arr[right_mark] = arr[right_mark], arr[first]
        return right_mark

    @staticmethod
    def shell_sort(arr, reverse=False):
        """Реализация сортировки Шелла."""
        gap = len(arr) // 2
        while gap > 0:
            for i in range(gap, len(arr)):
                temp = arr[i]
                j = i
                while j >= gap and ((not reverse and arr[j - gap] > temp) or (reverse and arr[j - gap] < temp)):
                    arr[j] = arr[j - gap]
                    j -= gap
                arr[j] = temp
            gap //= 2
        return arr

    @staticmethod
    def merge_sort(arr, reverse=False):
        """Реализация сортировки слиянием."""
        if len(arr) <= 1:
            return arr
        mid = len(arr) // 2
        left_half = Sorting.merge_sort(arr[:mid], reverse=reverse)
        right_half = Sorting.merge_sort(arr[mid:], reverse=reverse)
        return Sorting._merge(left_half, right_half, reverse=reverse)

    @staticmethod
    def _merge(left_half, right_half, reverse=False):
        result = []
        i = j = 0
        while i < len(left_half) and j < len(right_half):
            if (not reverse and left_half[i] <= right_half[j]) or (reverse and left_half[i] >= right_half[j]):
                result.append(left_half[i])
                i += 1
            else:
                result.append(right_half[j])
                j += 1
        result += left_half[i:]
        result += right_half[j:]
        return result

    @staticmethod
    def measure_time(func):
        """Декоратор, измеряющий время выполнения функции и выводящий его в консоль."""
        def wrapper(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            end = time.time()
            print(f"\nВремя выполнения {tuple(kwargs.items())[0][1]}_sort: {end - start:.6f} сек.")
            return result
        return wrapper

    @staticmethod
    @measure_time
    def sort(arr, method='bubble', reverse=False):
        """Универсальный интерфейс вызова метода сортировки."""
        copy_arr = arr[:]
        if method == 'bubble':
            return Sorting.bubble_sort(copy_arr, reverse)
        elif method == 'cocktail':
            return Sorting.cocktail_sort(copy_arr, reverse)
        elif method == 'comb':
            return Sorting.comb_sort(copy_arr, reverse)
        elif method == 'selection':
            return Sorting.selection_sort(copy_arr, reverse)
        elif method == 'insertion':
            return Sorting.insertion_sort(copy_arr, reverse)
        elif method == 'quick':
            return Sorting.quick_sort(copy_arr, reverse)
        elif method == 'shell':
```

```
            return Sorting.shell_sort(copy_arr, reverse)
        elif method == 'merge':
            return Sorting.merge_sort(copy_arr, reverse)
        else:
            raise ValueError(f"Метод сортировки '{method}' не поддерживается.")
```

Пример использования метода `comb_sort` из класса `Sorting`:

In [ ]:
```
import random
import copy
```

In [ ]:
```
arr = [random.randint(-100,100) for _ in range(40)]
```

In [ ]:
```
print('Исходный список:\n\t', arr)
print('\nОтсортированный список:\n\t', Sorting.sort(arr, method='comb'))
```

```
Исходный список:
        [49, 38, 81, -71, 59, -79, 92, -39, 39, 36, -74, -61, 34, -69, -54, -72, 81, -78, 22, 48, -89, -72, -92
, -26, -59, -53, -96, -60, 14, -61, -99, -92, -82, 86, -91, 38, 76, 67, 93, 44]

Время выполнения comb_sort: 0.000068 сек.

Отсортированный список:
        [-99, -96, -92, -92, -91, -89, -82, -79, -78, -74, -72, -72, -71, -69, -61, -61, -60, -59, -54, -53, -3
9, -26, 14, 22, 34, 36, 38, 38, 39, 44, 48, 49, 59, 67, 76, 81, 81, 86, 92, 93]
```

Создаётся список `arr` из 15 000 уникальных целых чисел в диапазоне от `0` до `14999`, перемешанных в случайном порядке. Функция `random.sample(population, k)` возвращает список длины `k`, элементы которого случайным образом выбраны из последовательности `population` без повторений. Здесь она используется для моделирования неотсортированного набора данных, на котором можно сравнивать эффективность различных алгоритмов сортировки.

In [ ]:
```
arr = random.sample(range(15000), 15000)
arr[:10]
```

Out[ ]:
```
[14312, 13524, 13148, 10157, 14040, 316, 4550, 1564, 7576, 8793]
```

Создаётся список `sorting_methods`, содержащий названия реализованных алгоритмов сортировки, после чего запускается цикл, в котором для каждого метода создаётся глубокая копия исходного массива `arr`, чтобы избежать изменения оригинальных данных, и выполняется сортировка с помощью универсального метода `Sorting.sort`, принимающего на вход название алгоритма.

In [ ]:
```
sorting_methods = [
    'bubble', 'cocktail', 'comb', 'selection',
    'insertion', 'quick', 'shell', 'merge'
    ]
```

In [ ]:
```
for sorting_method in sorting_methods:
    arr_copy = copy.deepcopy(arr)
    sorted_arr = Sorting.sort(arr_copy, method=sorting_method)
```

```
Время выполнения bubble_sort: 13.007318 сек.

Время выполнения cocktail_sort: 11.375020 сек.

Время выполнения comb_sort: 0.105752 сек.

Время выполнения selection_sort: 7.119220 сек.

Время выполнения insertion_sort: 5.094423 сек.

Время выполнения quick_sort: 0.025645 сек.

Время выполнения shell_sort: 0.045120 сек.

Время выполнения merge_sort: 0.039636 сек.
```

Представленные результаты ярко демонстрируют различия в производительности алгоритмов сортировки при работе с массивом из 15 000 элементов. Наиболее медленно себя показали обменные и простые алгоритмы: `bubble_sort`, `cocktail_sort`, `selection_sort` и `insertion_sort`, чья трудоёмкость составляет $O(n^2)$ — особенно это видно по `bubble_sort`, который завершился лишь спустя 13 секунд. На их фоне значительно быстрее работают сортировки улучшенного класса: `comb_sort` потребовала всего около 0.1 секунды. Быстрая сортировка (`quick_sort`), сортировка Шелла (`shell_sort`) и сортировка слиянием (`merge_sort`), обладающие асимптотической сложностью $O(n \log n)$, показали наилучшие результаты — менее 0.05 секунды каждая. Это подтверждает практическую эффективность этих алгоритмов на больших объёмах данных и оправдывает их использование в промышленных задачах.

Вывод первых десяти элементов отсортированного списка:

```
```

In [ ]:
```
sorted_arr[:10]
```

Out[ ]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Для альтернативной оценки времени выполнения алгоритмов сортировки можно использовать магическую команду `%timeit`, предназначенную для интерактивной среды Jupyter Notebook. Эта команда автоматически выполняет заданный код многократно и вычисляет среднее время выполнения, что позволяет получить более точную и воспроизводимую оценку производительности. Кроме того, `%timeit` самостоятельно подбирает оптимальное количество повторений и предоставляет сводную статистику, включая среднее время и стандартное отклонение.

Для интеграции с магической командой создаётся вспомогательная функция `sort_timeit()`, служащая точкой входа для вызова нужного алгоритма сортировки. Она принимает массив `arr`, название метода `method` и параметр `reverse`, указывающий порядок сортировки. Внутри функции происходит выбор соответствующего метода сортировки из класса `Sorting`:

In [ ]:
```
def sort_timeit(arr, method='bubble', reverse=False):
    if method == 'bubble':
        return Sorting.bubble_sort(arr, reverse)
    elif method == 'cocktail':
        return Sorting.cocktail_sort(arr, reverse)
    elif method == 'comb':
        return Sorting.comb_sort(arr, reverse)
    elif method == 'selection':
        return Sorting.selection_sort(arr, reverse)
    elif method == 'insertion':
        return Sorting.insertion_sort(arr, reverse)
    elif method == 'quick':
        return Sorting.quick_sort(arr, reverse)
    elif method == 'shell':
        return Sorting.shell_sort(arr, reverse)
    elif method == 'merge':
        return Sorting.merge_sort(arr, reverse)
```

Создаётся также список `arr` из 15 000 уникальных целых чисел:

In [ ]:
```
import random
import copy
```

In [ ]:
```
arr = random.sample(range(15000), 15000)
arr[:10]
```

Out[ ]:
```
[4374, 5069, 13091, 12164, 1588, 14821, 4933, 12149, 14164, 12650]
```

Для сгенерированного списка целых чисел применяется каждый алгоритм сортировки:

In [ ]:
```
sorting_methods = [
    'bubble', 'cocktail', 'comb', 'selection',
    'insertion', 'quick', 'shell', 'merge'
    ]
```

In [ ]:
```
for sorting_method in sorting_methods:
    print(f'{sorting_method:>10}_sort:', end='\t')
    %timeit sort_timeit(copy.deepcopy(arr), method=sorting_method)
    print()
```

```
   bubble_sort:	13.2 s ± 252 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

 cocktail_sort:	11.1 s ± 370 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

     comb_sort:	64.9 ms ± 742 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

selection_sort:	6.48 s ± 497 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

insertion_sort:	5.63 s ± 574 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

    quick_sort:	28.3 ms ± 646 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

    shell_sort:	67.9 ms ± 18.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

    merge_sort:	46.5 ms ± 990 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Результаты, полученные с помощью магической команды `%timeit`, также демонстрируют резкое различие в производительности между различными алгоритмами сортировки при работе с массивом из 15 000 элементов. Обменные алгоритмы `bubble_sort` (13.2 сек) и `cocktail_sort` (11.1 сек), показали наихудшие результаты: их временная сложность $O(n^2)$ приводит к значительным затратам времени даже при относительно небольшом размере входных данных. Сортировки

`selection_sort` (6.48 сек) и `insertion_sort` (5.63 сек) оказались быстрее, но также существенно отстают от более оптимальных алгоритмов.

Наиболее эффективными оказались `quick_sort` (28.3 мс), `merge_sort` (46.5 мс), `comb_sort` (64.9 мс) и `shell_sort` (67.9 мс), продемонстрировав стабильную и предсказуемую работу благодаря улучшенным стратегиям разбиения и слияния данных. Особенно заметен разрыв между быстрыми алгоритмами ($O(n \log n)$ в среднем) и наивными — в десятки, а иногда и в сотни раз. Эти результаты ещё раз подчёркивают важность выбора алгоритма в зависимости от объёма данных и требований к производительности.

ПРИМЕР

Необходимо вывести на экран топ-5 товаров, которые принесли наибольшую выручку за определенный период времени. При решении использовать алгоритмы сортировки: сортировку пузырьком, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Информация о товарах хранится в файле.

I. Подготовка данных для CSV-файла

Прежде чем решить поставленную задачу, программно сформируем CSV-файл, который будет хранить данные о товарах. Создадим словарь, где ключи — это категории товаров, а значения — это списки словарей, где каждый словарь представляет отдельный товар с его названием и ценовым диапазоном. Данный словарь будем использовать для автоматического создания списка объектов класса «Товар» и последующей записи их в файл.

In [ ]:
```
products = {
    'Электроника': [
        {'Смартфон': (10000, 30000)},
        {'Ноутбук': (20000, 50000)},
        {'Планшет': (8000, 25000)},
        {'Телевизор': (15000, 50000)},
        {'Наушники': (500, 5000)}
        ],
    'Одежда': [
        {'Футболка': (500, 1500)},
        {'Джинсы': (1000, 3000)},
        {'Рубашка': (800, 2500)},
        {'Юбка': (1200, 3500)},
        {'Куртка': (2000, 5000)}
        ],
    'Обувь': [
        {'Кроссовки': (1500, 4000)},
        {'Ботинки': (2500, 6000)},
        {'Туфли': (2000, 4500)},
        {'Сандалии': (1000, 3000)}
        ],
    'Косметика': [
        {'Шампунь': (50, 500)},
        {'Крем для лица': (100, 800)},
        {'Тушь для ресниц': (50, 300)},
        {'Помада': (80, 500)}
        ],
    'Продукты питания': [
        {'Молоко': (20, 80)},
        {'Хлеб': (15, 50)},
        {'Яйца': (30, 100)},
        {'Мясо': (200, 1000)},
        {'Фрукты': (50, 500)}
        ],
    'Канцелярские товары': [
        {'Ручка': (5, 50)},
        {'Бумага': (50, 500)},
        {'Карандаш': (3, 20)},
        {'Ластик': (10, 50)},
        {'Клей': (20, 100)},
        {'Ножницы': (50, 200)}
        ],
    'Спортивные товары': [
        {'Мяч для футбола': (300, 1000)},
        {'Гантели': (500, 1500)},
        {'Велосипед': (5000, 20000)},
        {'Теннисная ракетка': (1000, 5000)}
        ],
    'Автотовары': [
        {'Автомобильное масло': (500, 2000)},
        {'Автомобильные шины': (2000, 8000)},
        {'Автомобильный аккумулятор': (3000, 12000)}
        ],
    'Домашний текстиль': [
        {'Постельное белье': (1000, 5000)},
```

```
        {'Подушка': (500, 2000)},
        {'Одеяло': (1500, 7000)}
        ]
    }
```

Класс `Product_for_CSV` предназначен для представления информации о товаре, предназначенной для последующего сохранения в CSV-файл. Он инкапсулирует основные характеристики товара и предоставляет набор методов для анализа и отображения данных.

Инициализация объекта осуществляется через конструктор `__init__`, который принимает пять параметров: наименование (`name`), категорию (`category`), оптовую цену (`wholesale_price`), розничную цену (`retail_price`) и список объёмов продаж по месяцам за год (`sales`). Все параметры сохраняются в соответствующих атрибутах экземпляра.

Метод `__str__` возвращает строковое представление объекта в удобочитаемом формате, включая наименование, категорию, цены и значения продаж.

Метод `total_sales` вычисляет суммарное количество проданных единиц товара за год, а метод `total_revenue` определяет полную выручку от продаж, умножая общее количество продаж на розничную цену. Метод `average_price` возвращает среднее значение объёма продаж за месяц.

Метод `add_sales` позволяет дополнять список продаж, добавляя новое значение к имеющимся данным.

Структура класса позволяет удобно работать с товарными позициями, формировать сводки по продажам и экспортировать данные в формат CSV для последующей обработки или визуализации.

In [ ]:
```
class Product_for_CSV:
    def __init__(self, name, category, wholesale_price, retail_price, sales):
        self.name = name
        self.category = category
        self.wholesale_price = wholesale_price
        self.retail_price = retail_price
        self.sales = sales

    def __str__(self):
        return (
            f'Наименование товара: {self.name}\n'
            f'Категория товара: {self.category}\n'
            f'Оптовая цена: {self.wholesale_price}\n'
            f'Розничная цена: {self.retail_price}\n'
            f'Продажи за год: {self.sales}\n'
            )

    def total_sales(self):
        return sum(self.sales)

    def total_revenue(self):
        return sum(self.sales) * self.retail_price

    def average_price(self):
        return sum(self.sales) / len(self.sales)

    def add_sales(self, sales):
        self.sales.append(sales)
```

Следующий фрагмент программы формирует коллекцию товарных объектов на основе словаря `products`, где хранятся категории товаров и соответствующие диапазоны оптовых цен. Для каждой итерации случайным образом выбирается категория, затем внутри неё случайным образом выбирается товар. К названию этого товара добавляется шесть случайных цифр, образующих артикул. После этого случайным образом выбирается оптовая цена в заданных границах, а розничная цена определяется путём применения случайного коэффициента наценки. Далее формируется список месячных продаж за год. Все эти параметры передаются в конструктор класса `Product_for_CSV`, после чего созданный объект добавляется в итоговый список `list_of_products`.

In [ ]:
```
import random
```

In [ ]:
```
list_of_products = []
```

In [ ]:
```
for _ in range(5000):
    category = random.choice(list(products.keys()))
    product_dict = random.choice(products[category])
    name = list(product_dict.keys())[0]
    article = ''.join([str(random.randint(0, 9)) for _ in range(6)])
    product_name = f"{name} Арт.{article}"

    min_price, max_price = list(product_dict.values())[0]
    wholesale_price = float(round(random.randint(min_price, max_price), -1))

    markup = random.choice([1.2, 1.25, 1.3, 1.35, 1.4, 1.45, 1.5, 1.55])
```

```
    retail_price = float(round(markup * wholesale_price, -1))

    sales = [round(random.randint(10, 50), -1) for _ in range(12)]

    list_of_products.append(
        Product_for_CSV(
            product_name,
            category,
            wholesale_price,
            retail_price,
            sales
        )
    )
```

II. Формирование CSV-файла

Следующий фрагмент кода осуществляет сохранение списка объектов класса `Product_for_CSV` в файл формата CSV. Сначала открывается (или создаётся) файл `products.csv` в режиме записи с использованием модуля `csv`. Создаётся объект `writer`, предназначенный для последовательной записи строк в файл. Заголовок таблицы формируется в переменной `headers` и включает наименования полей — «name», «category», «wholesale_price», «retail_price» — а также номера месяцев от 1 до 12, соответствующие значениям продаж по каждому месяцу. Для каждого товара из списка `list_of_products` формируется строка, содержащая его характеристики и годовой вектор продаж. Эти строки последовательно записываются в CSV-файл с помощью метода `writerow`.

In [ ]:
```
import csv
```

In [ ]:
```
with open("products.csv", "w", newline="") as file:
    writer = csv.writer(file)

    headers = [
        "name", "category",
        "wholesale_price", "retail_price"
        ] + [str(i) for i in range(1, 13)]
    writer.writerow(headers)

    for product in list_of_products:
        row = [
            product.name,
            product.category,
            product.wholesale_price,
            product.retail_price
            ] + product.sales
        writer.writerow(row)
```

После выполнения кода формируется структурированный CSV-файл, пригодный для дальнейшего анализа или визуализации данных.

III. Чтение CSV-файла и создание списка объектов класса Product

Класс `Product` предназначен для представления информации о товаре и содержит набор атрибутов и методов, обеспечивающих базовые операции с данными о продажах. При создании объекта класса указываются следующие характеристики: наименование (`name`), категория (`category`), оптовая и розничная цены (`wholesale_price` , `retail_price`), а также список месячных продаж за год (`sales`). Метод `__str__()` формирует строковое представление объекта с отображением всех его основных параметров. Метод `total_sales()` возвращает общее количество проданных единиц товара за год. Метод `total_revenue(period=0)` вычисляет выручку: либо за полный год, либо за заданное число последних месяцев, если указан аргумент `period` . Метод `average_price()` рассчитывает среднее количество продаж в месяц. Метод `add_sales(sales)` добавляет новую запись о продажах, предварительно удаляя самую старую, тем самым поддерживая актуальный 12-месячный период продаж.

In [ ]:
```
class Product:
    def __init__(self, name, category, wholesale_price, retail_price, sales):
        self.name = name
        self.category = category
        self.wholesale_price = wholesale_price
        self.retail_price = retail_price
        self.sales = sales

    def __str__(self):
        return (
            f'Наименование товара: {self.name}\n'
            f'Категория товара: {self.category}\n'
            f'Оптовая цена: {self.wholesale_price}\n'
            f'Розничная цена: {self.retail_price}\n'
            f'Продажи за год: {self.sales}\n'
        )

    def total_sales(self):
```

```
        return sum(self.sales)

    def total_revenue(self, period=0):
        if not period:
            return sum(self.sales) * self.retail_price
        return sum(self.sales[len(self.sales) - period:]) * self.retail_price

    def average_price(self):
        return sum(self.sales) / len(self.sales)

    def add_sales(self, sales):
        del self.sales[-12]
        self.sales.append(sales)
```

Для загрузки данных из CSV-файла и последующего создания объектов класса `Product` используется модуль `csv`. Открытие файла `products.csv` осуществляется в режиме чтения с автоматическим управлением ресурсами с помощью конструкции `with`. Далее создаётся итератор `reader`, читающий строки файла. Первая строка, содержащая заголовки столбцов, пропускается вызовом `next(reader)`. Затем каждая последующая строка интерпретируется как запись о товаре, в которой значения атрибутов извлекаются из соответствующих столбцов. Оптовая и розничная цены приводятся к типу `float`, а список месячных продаж формируется из оставшихся значений с преобразованием в тип `int`. Полученные данные используются для создания экземпляра класса `Product`, который добавляется в общий список `list_of_products`.

In [ ]:
```
import csv
```

In [ ]:
```
list_of_products = []
```

In [ ]:
```
with open('products.csv', 'r') as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        name = row[0]
        category = row[1]
        wholesale_price = float(row[2])
        retail_price = float(row[3])
        sales = [int(sale) for sale in row[4:]]
        product = Product(name, category, wholesale_price, retail_price, sales)
        list_of_products.append(product)
```

IV. Создание декоратора, вычисляющего время выполнения функции и выводящий его на экран

Для измерения времени выполнения произвольной функции можно использовать специальный декоратор `measure_time`. Этот декоратор принимает функцию в качестве аргумента и возвращает обёртку — внутреннюю функцию `wrapper`, которая позволяет отследить продолжительность выполнения. Обёртка принимает произвольное количество позиционных и именованных аргументов, передаёт их исходной функции `func`, и сохраняет результат её выполнения. Перед вызовом фиксируется время начала работы (`start`), после — время завершения (`end`). Разница между этими значениями отражает длительность выполнения функции и выводится в консоль с точностью до шести знаков после запятой. Такой подход обеспечивает универсальный механизм профилирования, пригодный для оценки производительности любых функций, включая сортировку.

In [ ]:
```
def measure_time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"\nВремя выполнения сортировки: {end - start:.6f} сек.")
        return result
    return wrapper
```

V. Описание функций, реализующих алгоритмы сортировки

В следующем фрагменте представлены реализации трёх алгоритмов сортировки: пузырьковой, выбором и быстрой сортировки, применённых к списку объектов `products`. Каждый из алгоритмов учитывает выручку от продаж товаров за заданный период времени (`period`) и направление сортировки (`reverse`). Особое внимание уделено функции `quick_sort`, которая реализована через вспомоговые функции `quick_sort_helper` и `partition`, что обеспечивает эффективную работу с исходным массивом без создания дополнительных списков. Универсальная функция `sort` служит точкой входа: в зависимости от выбранного метода (`method` равен `'1'`, `'2'` или `'3'`), она вызывает соответствующий алгоритм сортировки и измеряет время его выполнения с помощью декоратора `measure_time`.

In [ ]:
```
# реализация алгоритма сортировки пузырьком
def bubble_sort(products, period=0, reverse=False):
    n = len(products)
    for i in range(n):
        for j in range(n - i - 1):
            if not reverse:
                if products[j].total_revenue(period) > products[j + 1].total_revenue(period):
```

```
                    products[j], products[j + 1] = products[j + 1], products[j]
            else:
                if products[j].total_revenue(period) < products[j + 1].total_revenue(period):
                    products[j], products[j + 1] = products[j + 1], products[j]
    return products

# реализация алгоритма сортировки выбором
def selection_sort(products, period=0, reverse=False):
    n = len(products)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if reverse:
                if products[j].total_revenue(period) > products[min_idx].total_revenue(period):
                    min_idx = j
            else:
                if products[j].total_revenue(period) < products[min_idx].total_revenue(period):
                    min_idx = j
        products[i], products[min_idx] = products[min_idx], products[i]
    return products

# реализация алгоритма быстрой сортировки с помощью вспомогательных функций
def quick_sort(products, period=0, reverse=False):
    quick_sort_helper(products, 0, len(products) - 1, period, reverse)
    return products

def quick_sort_helper(products, first, last, period, reverse):
    if first < last:
        split_point = partition(products, first, last, period, reverse)
        quick_sort_helper(products, first, split_point - 1, period, reverse)
        quick_sort_helper(products, split_point + 1, last, period, reverse)

def partition(products, first, last, period, reverse):
    pivot_value = products[first].total_revenue(period)
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        if not reverse:
            while left_mark <= right_mark and products[left_mark].total_revenue(period) <= pivot_value:
                left_mark += 1
            while right_mark >= left_mark and products[right_mark].total_revenue(period) >= pivot_value:
                right_mark -= 1
        else:
            while left_mark <= right_mark and products[left_mark].total_revenue(period) >= pivot_value:
                left_mark += 1
            while right_mark >= left_mark and products[right_mark].total_revenue(period) <= pivot_value:
                right_mark -= 1

        if right_mark < left_mark:
            done = True
        else:
            products[left_mark], products[right_mark] = products[right_mark], products[left_mark]

    products[first], products[right_mark] = products[right_mark], products[first]
    return right_mark

# выбор метода сортировки в зависимости от переданного параметра
@measure_time
def sort(products, method='1', period=0, reverse=False):
    if method == '1':
        return bubble_sort(products, period, reverse)
    elif method == '2':
        return selection_sort(products, period, reverse)
    elif method == '3':
        return quick_sort(products, period, reverse)
```

VI. Определение топ-5 товаров по выручке

На заключительном этапе решается основная задача: определить топ-5 товаров по объёму выручки за заданный период. Сначала создаётся полная копия исходного списка `list_of_products` с помощью функции `deepcopy` из модуля `copy`, чтобы исключить изменение оригинальных данных. Затем пользователь вводит число месяцев, за которые требуется рассчитать выручку, а также выбирает метод сортировки: пузырьком, выбором или быструю сортировку. После этого вызывается функция `sort`, в которую передаются копия списка товаров, выбранный метод и период. Итогом выполнения является список из пяти товаров с наибольшей выручкой. В завершение информация о каждом товаре выводится на экран: отображаются его порядковый номер, наименование, категория и сумма выручки за указанный промежуток времени.

In [ ]:
```
import copy
```

In [ ]:
```
list_of_products_copy = copy.deepcopy(list_of_products)
```

In [ ]:
```
months = int(input(
    'За какой период надо посмотреть выручку?\n'
    'Введите количество месяцев (0 – год): '
    ))
```

```
За какой период надо посмотреть выручку?
Введите количество месяцев (0 – год): 6
```

In [ ]:
```
method = input(
    'Методы сортировки:\n'
    '1. Сортировка пузырьком\n'
    '2. Сортировка выбором\n'
    '3. Быстрая сортировка\n'
    'Выберете метод сортировки: '
    )
```

```
Методы сортировки:
1. Сортировка пузырьком
2. Сортировка выбором
3. Быстрая сортировка
Выберете метод сортировки: 1
```

In [ ]:
```
top_5_products = sort(
    list_of_products_copy,
    method=method,
    period=months,
    reverse=True
    )[:5]
```

```
Время выполнения сортировки: 13.732395 сек.
```

In [ ]:
```
print(f"\nТоп-5 товаров по выручке за последние {months} мес.:")
for i, product in enumerate(top_5_products, 1):
    print(f"{i}. {product.name:<25} {product.category:<15} "
          f"{product.total_revenue(months):<10} руб.")
```

```
Топ-5 товаров по выручке за последние 6 мес.:
1. Телевизор Арт.574249      Электроника     16163700.0 руб.
2. Телевизор Арт.012986      Электроника     15080000.0 руб.
3. Ноутбук Арт.964449        Электроника     14746200.0 руб.
4. Ноутбук Арт.453080        Электроника     14440000.0 руб.
5. Ноутбук Арт.264818        Электроника     14434000.0 руб.
```

Таким образом, была реализована задача определения топ-5 товаров с наибольшей выручкой за заданный период времени. Для этого была создана система генерации и хранения товарных данных в формате CSV, реализованы два класса для представления товарных объектов, подготовлен декоратор для оценки производительности и разработаны три алгоритма сортировки — пузырьковая, выбором и быстрая, адаптированные под специфические критерии сравнения. Сравнительный анализ показал различие во времени выполнения различных алгоритмов при сортировке по выручке. Реализация задачи позволила не только отработать навыки работы с файлами, классами и сортировками, но и продемонстрировала практическое применение алгоритмического анализа в условиях приближённых к реальным задачам бизнес-аналитики.