**Алгоритмы сортировки в Python**

<https://stackabuse.com/sorting-algorithms-in-python/>

**Введение**

Иногда данные, которые мы храним или извлекаем в приложении, могут иметь небольшой порядок или вообще не иметь его. Возможно, нам придется переупорядочить данные, чтобы правильно их обработать или эффективно использовать. За прошедшие годы компьютерщики создали множество алгоритмов сортировки для организации данных.

В этой статье мы рассмотрим популярные алгоритмы сортировки, поймем, как они работают, и закодируем их на Python. Мы также сравним, насколько быстро они сортируют элементы в списке.

Для простоты реализации алгоритма будут сортировать списки чисел в порядке возрастания. Конечно, вы можете адаптировать их к своим потребностям.

**Пузырьковая сортировка**

Этот простой алгоритм сортировки выполняет итерацию по списку, сравнивая элементы попарно и меняя их местами до тех пор, пока более крупные элементы не «поднимутся» в конец списка, а более мелкие останутся «внизу».

**Объяснение**

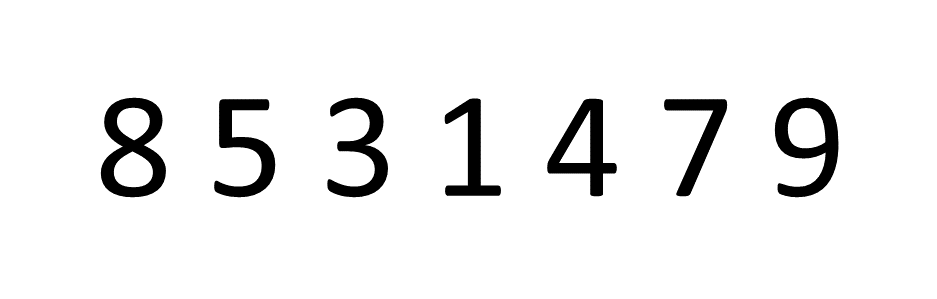
Начнем со сравнения первых двух элементов списка. Если первый элемент больше второго, мы меняем их местами. Если они уже в порядке, мы оставляем их как есть. Затем мы переходим к следующей паре элементов, сравниваем их значения и при необходимости меняем местами. Этот процесс продолжается до последней пары элементов в списке.

По достижении конца списка этот процесс повторяется для каждого элемента. Хотя это крайне неэффективно. Что делать, если в массиве нужно сделать только один обмен? Зачем нам все еще повторять это n ^ 2 раза, даже если оно уже отсортировано?

Очевидно, что для оптимизации алгоритма нам нужно остановить его, когда он закончит сортировку, иначе он будет многократно пересчитывать уже отсортированный массив.

Как мы узнаем, что закончили сортировку? Если бы предметы были в порядке, нам не пришлось бы их менять местами. Таким образом, всякий раз, когда мы обмениваем значения, мы устанавливаем флаг для True повторения процесса сортировки. Если обменов не произошло, флаг останется, False и алгоритм остановится.

Идея пузырьковой сортировки очень проста: мы смотрим на пары соседних элементов в массиве, по одной паре за раз, и меняем их позиции, если первый элемент больше второго, или просто идем дальше, если это не так. Давайте рассмотрим пример и отсортируем массив 8, 5, 3, 1, 4, 7, 9 :



**Оптимизация**

Если обмены не производятся, это означает, что список отсортирован. Хотя с ранее реализованным алгоритмом он будет продолжать оценивать остальную часть списка, даже если в этом нет необходимости. Чтобы исправить это, мы сохраним booleanфлаг и проверим, были ли сделаны какие-либо свопы в предыдущей итерации.

Если обмены не производятся, алгоритм должен остановиться:

Другая оптимизация, которую мы можем сделать, использует тот факт, что пузырьковая сортировка работает таким образом, что самые большие элементы в конкретной итерации оказываются в конце массива.

При первом проходе по списку позиция *n* гарантированно будет самым большим элементом, при втором проходе по списку позиция *n-1* гарантированно будет вторым по величине элементом и так далее.

Это означает, что с каждой последующей итерацией мы можем просматривать на один элемент меньше, чем раньше. Точнее, на *k* -й итерации нужно смотреть только первые *n - k + 1* элементов:

**Заключение**

В самом неэффективном подходе пузырьковая сортировка проходит n-1 итераций, просматривая n-1 пару соседних элементов. Это дает временную сложность O(n\*\*2 ) как в лучшем, так и в среднем случае.

O(n\*\*2 ) считается довольно ужасным для алгоритма сортировки.

def bubble\_sort\_1(\_nums):  
 for i in range(len(\_nums)):  
 for j in range(len(\_nums) - 1):  
 if \_nums[j] > \_nums[j+1]:  
 \_nums[j], \_nums[j+1] = \_nums[j+1], \_nums[j]

# оптимизация 1  
def bubble\_sort\_2(\_nums):  
 swapped = True  
 while swapped:  
 swapped = False  
 for i in range(len(\_nums) - 1):  
 if \_nums[i] > \_nums[i + 1]:  
 \_nums[i], \_nums[i + 1] = \_nums[i + 1], \_nums[i]  
 swapped = True

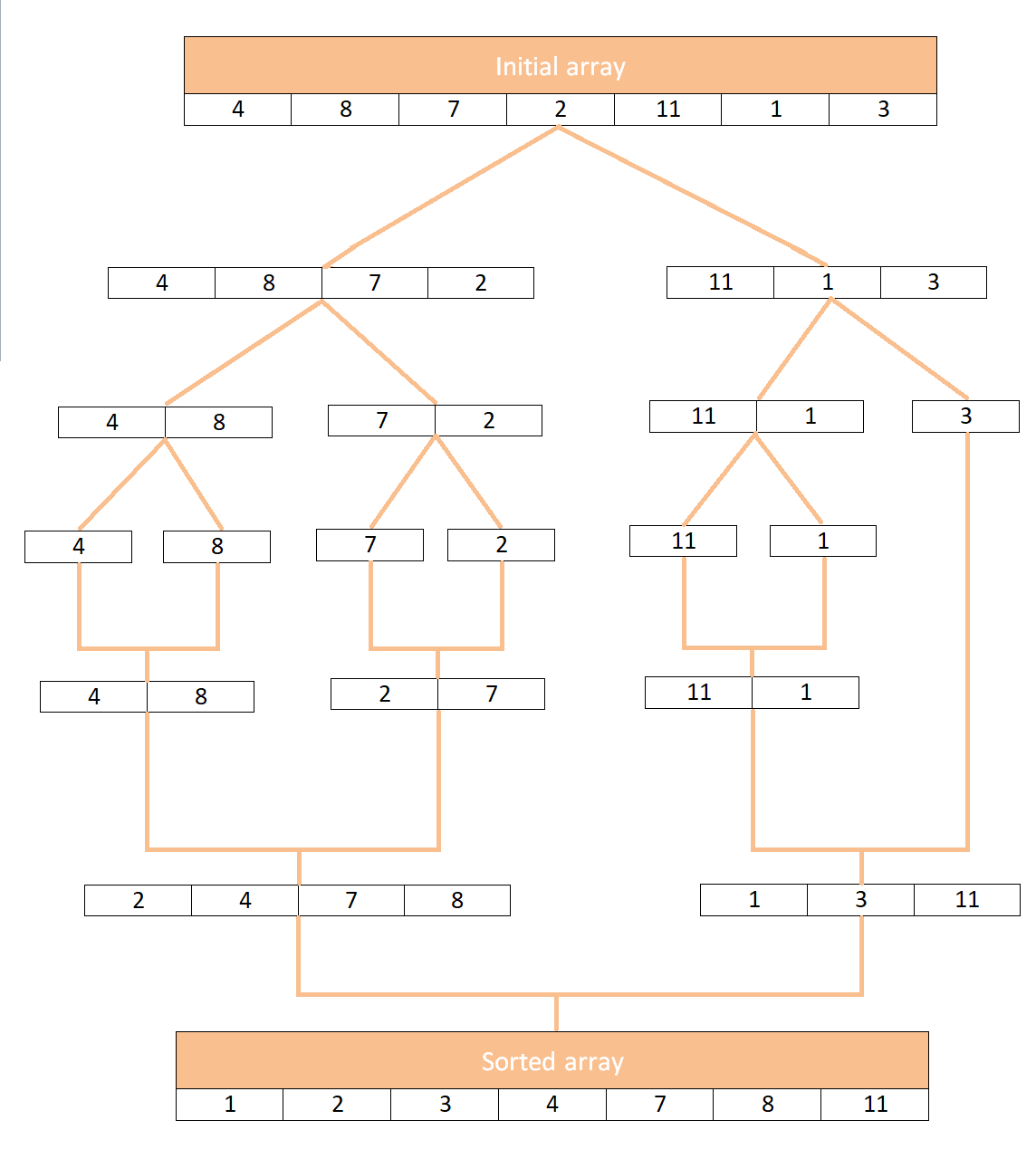
def bubble\_sort\_3(\_nums):  
 has\_swapped = True  
 num\_of\_iterations = 0  
 while(has\_swapped):  
 has\_swapped = False  
 for i in range(len(\_nums) - num\_of\_iterations - 1):  
 if \_nums[i] > \_nums[i+1]:  
 \_nums[i], \_nums[i+1] = \_nums[i+1], \_nums[i]  
 has\_swapped = True  
 num\_of\_iterations += 1

**Сортировка слиянием**

Исходный массив разбивается на две примерно равные части. Если массив имеет нечетное количество элементов, одна из этих «половинок» на один элемент больше другой.

Подмассивы снова и снова делятся на половины, пока не получатся массивы, в каждом из которых содержится только один элемент.

Затем вы объединяете пары одноэлементных массивов в двухэлементные массивы, сортируя их в процессе. Затем эти отсортированные пары объединяются в четырехэлементные массивы и так далее, пока вы не получите исходный отсортированный массив.



Как видите, то, что массив нельзя разделить на равные половины, не проблема, 3 просто «ждет», пока не начнется сортировка.

Существует два основных способа реализации алгоритма сортировки слиянием, один из которых использует нисходящий подход, как в приведенном выше примере, именно так чаще всего вводится сортировка слиянием.

Другой подход, т.е. снизу вверх , работает в обратном направлении, без рекурсии (работает итеративно) - если наш массив имеет N элементов, мы делим его на N подмассивов одного элемента и сортируем пары соседних одноэлементных массивов, затем сортируем соседние пары двухэлементных массивов и так далее.

Примечание.

Восходящий подход обеспечивает интересную оптимизацию, которую мы обсудим позже. Мы будем реализовывать нисходящий подход, так как он более простой и интуитивно понятный в сочетании с тем фактом, что нет реальной разницы между временной сложностью между ними без специальной оптимизации.

Возможны два особых случая:

1-Оба подмассива имеют один и тот же элемент. Мы можем двигаться вперед в любом из них и добавить элемент в отсортированный массив. Технически мы можем продвинуться вперед в обоих массивах и добавить оба элемента в отсортированный массив, но это потребует особого поведения, когда мы встретим одни и те же элементы в обоих массивах.

2-У нас «заканчиваются» элементы в одном подмассиве. Например, у нас есть массив с {1, 2, 3} и массив с {9, 10, 11}. Ясно, что мы пройдемся по всем элементам первого массива, не продвинувшись вперед ни разу во втором. Всякий раз, когда у нас заканчиваются элементы в подмассиве, мы просто добавляем элементы второго подмассива за другим.

Имейте в виду, что мы можем сортировать как хотим — в этом примере целые числа сортируются в порядке возрастания, но мы можем так же легко сортировать в порядке убывания или сортировать пользовательские объекты.

**Выполнение**

Сортировка выполняется путем сравнения наименьших элементов каждой половины. Первый элемент каждого списка сравнивается первым. Если первая половина начинается с меньшего значения, мы добавляем его в отсортированный список. Затем мы сравниваем второе наименьшее значение первой половины с первым наименьшим значением второй половины. Каждый раз, когда мы выбираем меньшее значение в начале половины, мы перемещаем индекс элемента, который нужно сравнить, на единицу.

**Сортировка массивов**

Обратите внимание, что merge\_sort() функция, в отличие от предыдущих алгоритмов сортировки, возвращает новый отсортированный список, а не сортирует существующий список. Поэтому для сортировки слиянием требуется место для создания нового списка того же размера, что и входной список.

**Сложность времени**

Давайте сначала посмотрим на merge функцию. Он принимает два списка и повторяется n раз, где n — размер их объединенного ввода.

Функция merge\_sort разбивает заданный массив на 2 и рекурсивно сортирует подмассивы. Поскольку рекурсивный ввод составляет половину того, что было дано, время, необходимое для обработки, увеличивается логарифмически до n .

Следовательно, общая временная сложность алгоритма сортировки слиянием составляет **O(nlog(n))** .

**Сортировка пользовательских объектов**

Теперь, когда у нас есть базовый алгоритм, мы можем взглянуть на то, как сортировать пользовательские классы. Мы можем переопределить \_\_eq\_\_, и другие операторы \_\_le\_\_, \_\_ge\_\_необходимые для этого.

Это позволяет нам использовать тот же алгоритм, что и выше, но ограничивает нас только одним способом сортировки наших пользовательских объектов, что в большинстве случаев не является тем, что нам нужно. Лучше сделать сам алгоритм более универсальным и вместо этого передать ему функцию сравнения.

Однако сортировка слиянием относительно неэффективна (как по времени, так и по пространству), когда речь идет о меньших массивах, и часто оптимизируется путем остановки, когда мы достигаем массива из ~ 7 элементов, вместо перехода к массивам с одним элементом и вызова сортировки вставками для вместо этого отсортируйте их перед объединением в больший массив.

Это связано с тем, что сортировка вставками очень хорошо работает с небольшими и/или почти отсортированными массивами.

**Заключение**

Сортировка слиянием — это эффективный алгоритм сортировки общего назначения. Его главное преимущество — надежное время работы алгоритма и его эффективность при сортировке больших массивов. В отличие от быстрой сортировки, она не зависит от каких-либо неудачных решений, которые приводят к ухудшению времени выполнения.

Одним из основных недостатков является дополнительная память, которую сортировка слиянием использует для хранения временных копий массивов перед их слиянием.

**Сортировка выбором**

Этот алгоритм делит список на две части: отсортированную и несортированную. Мы постоянно удаляем наименьший элемент несортированного сегмента списка и добавляем его к отсортированному сегменту.

**Объяснение**

На практике нам не нужно создавать новый список для отсортированных элементов, мы обрабатываем крайнюю левую часть списка как отсортированный сегмент. Затем мы ищем во всем списке наименьший элемент и меняем его местами с первым элементом. Теперь мы знаем, что первый элемент списка отсортирован, мы получаем наименьший элемент из оставшихся элементов и меняем его местами со вторым элементом. Это повторяется до тех пор, пока последний элемент списка не станет оставшимся элементом для проверки.

Мы видим i, что по мере увеличения нам нужно проверять меньше элементов.

**Сложность времени**

Для списка с n элементами внешний цикл повторяется n раз. Внутренний цикл повторяет n-1, когда i равно 1, затем n-2, когда i равно 2, и так далее. Количество сравнений равно (n - 1) + (n - 2) + ... + 1, что дает сортировке выбором временную сложность **O(n\*\*2).**

**Сортировка кучей**

Этот популярный алгоритм сортировки, разделяет список на отсортированные и несортированные части. Он преобразует несортированный сегмент списка в структуру данных кучи, чтобы мы могли эффективно определить самый большой элемент.

**Объяснение**

Начнем с преобразования списка в Max Heap — двоичное дерево, где самым большим элементом является корневой узел. Затем мы помещаем этот элемент в конец списка. Затем мы перестраиваем нашу максимальную кучу, которая теперь имеет на одно значение меньше, помещая новое наибольшее значение перед последним элементом списка. Мы повторяем этот процесс построения кучи, пока все узлы не будут удалены.

Сортировка кучей работает путем «удаления» элементов из кучи массива один за другим и добавления их в отсортированную часть массива. Прежде чем мы углубимся в объяснение и вернемся к структуре данных кучи, мы должны упомянуть несколько атрибутов самой сортировки кучи.

Это алгоритм на месте, что означает, что он требует постоянного объема дополнительной памяти, т. е. необходимая память не зависит от размера самого исходного массива, кроме памяти, необходимой для хранения этого массива.

Например, не нужны копии исходного массива, нет рекурсии и стеков рекурсивных вызовов. В простейшей реализации Heap Sort обычно используется второй массив для хранения отсортированных значений. Мы будем использовать этот подход, так как он гораздо более интуитивно понятен и прост в использовании в коде, но его можно реализовать полностью на месте.

Сортировка кучей нестабильна, что означает, что она не поддерживает относительный порядок элементов с одинаковыми значениями. Это не проблема с примитивными типами (такими как целые числа и символы...), но может быть проблемой, когда мы сортируем сложные типы, такие как объекты.

Например, представьте, что у нас есть пользовательский класс Person с age полями name и несколько объектов этого класса в массиве, в том числе человек по имени «Майк» в возрасте 19 лет и «Дэвид», которому также 19 лет, — в указанном порядке.

Если бы мы решили отсортировать этот массив людей по возрасту, не было бы никакой гарантии, что «Майк» появится перед «Дэвидом» в отсортированном массиве, даже если они появились в этом порядке в исходном массиве. Это может случиться, но это не гарантировано.

Забавный факт: Heap Sort — предпочтительный алгоритм сортировки в ядре Linux.

**Структура данных кучи (The Heap Data Structure)**

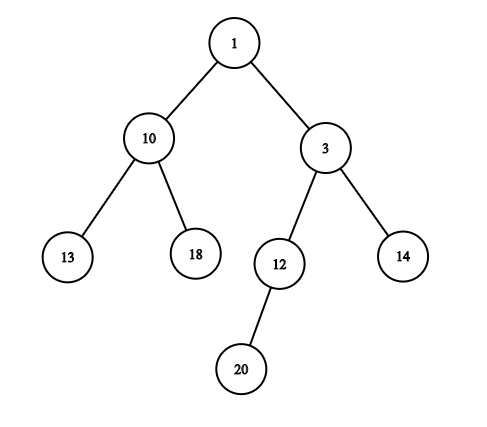
Кучи — одна из самых популярных и широко используемых **структур данных** в компьютерных науках, не говоря уже о том, что они очень популярны во время интервью по разработке программного обеспечения.

Мы будем говорить о кучах, отслеживающих наименьший элемент (min-heap), но их можно так же легко реализовать для отслеживания самого большого элемента (max-heap).

Проще говоря, мини-куча — это древовидная структура данных, в которой каждый узел меньше, чем все его дочерние элементы. Чаще всего используется бинарное дерево. Кучи имеют три поддерживаемые операции — delete\_minimum(), get\_minimum()и add().

Вы можете удалить только первый элемент в куче, после чего он "пересортируется". Кучи «пересортировывают» себя после добавления или удаления элемента, так что наименьший элемент всегда находится в первой позиции.

**Примечание.**

Это никоим образом не означает, что кучи являются отсортированными массивами. Тот факт, что каждый узел меньше, чем его дочерние элементы, недостаточен для гарантии того, что вся куча находится в возрастающем порядке.

Давайте посмотрим на пример кучи:

Как мы видим, приведенный выше пример подходит под описание кучи, но не отсортирован. Мы не будем вдаваться в подробности реализации кучи, так как это не является предметом рассмотрения в этой статье. Решающее преимущество структуры данных кучи, которую мы используем при сортировке кучей, заключается в том, что следующий наименьший элемент всегда является первым элементом в куче.

**Примечание.**

Благодаря тому, как элементы сортируются кучами после удаления элемента, сложность перемещения следующего наименьшего элемента в первую позицию при сохранении массива в виде кучи занимает **O(logn)** времени, что является очень эффективной операцией.

**Выполнение**

Python предоставляет методы для создания и использования кучи, поэтому нам не нужно реализовывать их самостоятельно:

**heappush(list, item):** Добавляет элемент в кучу, а затем повторно сортирует его, чтобы он оставался кучей. Может использоваться в пустом списке.

**heappop(list):** извлекает (удаляет) первый (наименьший) элемент и возвращает этот элемент. Куча остается кучей после этой операции, поэтому нам не нужно вызывать heapify().

**heapify(list):** превращает данный список в кучу. Стоит отметить, что этот метод существует, хотя мы не будем его использовать, поскольку не хотим изменять исходный массив.

Как мы видим, основная работа выполняется со структурой данных кучи, все, что нам нужно сделать, это добавить все необходимые элементы и удалить их один за другим. Это почти как машина для подсчета монет, которая сортирует введенные монеты по их стоимости, и мы можем потом их вынуть.

Сложность времени

Давайте сначала посмотрим на временную сложность функции heapify. В худшем случае самый большой элемент никогда не является корневым элементом, это вызывает рекурсивный вызов heapify. Хотя рекурсивные вызовы могут показаться пугающе дорогими, помните, что мы работаем с бинарным деревом.

Визуализируйте бинарное дерево с 3 элементами, оно имеет высоту 2. Теперь визуализируйте бинарное дерево из 7 элементов, оно имеет высоту 3. Дерево растет логарифмически до n . Функция heapifyпроходит это дерево за время O(log(n)) .

Функция heap\_sortперебирает массив n раз. Следовательно, общая временная сложность алгоритма сортировки кучей составляет O(nlog(n)) .

Сортировка слиянием

Этот алгоритм «разделяй и властвуй» делит список пополам и продолжает делить список на 2, пока в нем не останется только одиночные элементы.

Смежные элементы становятся отсортированными парами, затем отсортированные пары объединяются и также сортируются с другими парами. Этот процесс продолжается до тех пор, пока мы не получим отсортированный список со всеми элементами несортированного входного списка.

Объяснение

Мы рекурсивно делим список пополам, пока не получим списки с размером один. Затем мы объединяем каждую половину, которая была разделена, сортируя их в процессе.

Сортировка выполняется путем сравнения наименьших элементов каждой половины. Первый элемент каждого списка сравнивается первым. Если первая половина начинается с меньшего значения, мы добавляем его в отсортированный список. Затем мы сравниваем второе наименьшее значение первой половины с первым наименьшим значением второй половины.

Бесплатная электронная книга: Git Essentials

Ознакомьтесь с нашим практическим руководством по изучению Git с рекомендациями, принятыми в отрасли стандартами и прилагаемой памяткой. Прекратите гуглить команды Git и изучите их!

Скачать электронную книгу

Каждый раз, когда мы выбираем меньшее значение в начале половины, мы перемещаем индекс элемента, который нужно сравнить, на единицу.

Если вы хотите прочитать подробную статью, посвященную сортировке слиянием , мы вам поможем!

Выполнение

def merge(left\_list, right\_list):

sorted\_list = []

left\_list\_index = right\_list\_index = 0

# We use the list lengths often, so its handy to make variables

left\_list\_length, right\_list\_length = len(left\_list), len(right\_list)

for \_ in range(left\_list\_length + right\_list\_length):

if left\_list\_index < left\_list\_length and right\_list\_index < right\_list\_length:

# We check which value from the start of each list is smaller

# If the item at the beginning of the left list is smaller, add it

# to the sorted list

if left\_list[left\_list\_index] <= right\_list[right\_list\_index]:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

# If the item at the beginning of the right list is smaller, add it

# to the sorted list

else:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

# If we've reached the end of the of the left list, add the elements

# from the right list

elif left\_list\_index == left\_list\_length:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

# If we've reached the end of the of the right list, add the elements

# from the left list

elif right\_list\_index == right\_list\_length:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

return sorted\_list

def merge\_sort(nums):

# If the list is a single element, return it

if len(nums) <= 1:

return nums

# Use floor division to get midpoint, indices must be integers

mid = len(nums) // 2

# Sort and merge each half

left\_list = merge\_sort(nums[:mid])

right\_list = merge\_sort(nums[mid:])

# Merge the sorted lists into a new one

return merge(left\_list, right\_list)

# Verify it works

random\_list\_of\_nums = [120, 45, 68, 250, 176]

random\_list\_of\_nums = merge\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

Обратите внимание, что merge\_sort()функция, в отличие от предыдущих алгоритмов сортировки, возвращает новый отсортированный список, а не сортирует существующий список.

Поэтому для сортировки слиянием требуется место для создания нового списка того же размера, что и входной список.

Сложность времени

Давайте сначала посмотрим на mergeфункцию. Он принимает два списка и повторяется n раз, где n — размер их объединенного ввода.

Функция merge\_sortразбивает заданный массив на 2 и рекурсивно сортирует подмассивы. Поскольку рекурсивный ввод составляет половину того, что было дано, как и бинарные деревья, время, необходимое для обработки, увеличивается логарифмически до n .

Следовательно, общая временная сложность алгоритма сортировки слиянием составляет O(nlog(n)) .

Быстрая сортировка

Этот алгоритм «разделяй и властвуй» является наиболее часто используемым алгоритмом сортировки, описанным в этой статье. При правильной настройке он чрезвычайно эффективен и не требует дополнительного пространства, используемого сортировкой слиянием. Мы разбиваем список вокруг опорного элемента, сортируя значения вокруг опорного элемента.

Объяснение

Быстрая сортировка начинается с разделения списка — выбора одного значения из списка, которое будет находиться в отсортированном месте. Это значение называется точкой опоры. Все элементы, меньшие, чем точка поворота, перемещаются влево от нее. Все более крупные элементы перемещаются вправо.

Зная, что опорная точка находится на своем законном месте, мы рекурсивно сортируем значения вокруг опорной точки до тех пор, пока не будет отсортирован весь список.

Если вы хотите прочитать подробную статью, посвященную быстрой сортировке , мы вам поможем!

Выполнение

# There are different ways to do a Quick Sort partition, this implements the

# Hoare partition scheme. Tony Hoare also created the Quick Sort algorithm.

def partition(nums, low, high):

# We select the middle element to be the pivot. Some implementations select

# the first element or the last element. Sometimes the median value becomes

# the pivot, or a random one. There are many more strategies that can be

# chosen or created.

pivot = nums[(low + high) // 2]

i = low - 1

j = high + 1

while True:

i += 1

while nums[i] < pivot:

i += 1

j -= 1

while nums[j] > pivot:

j -= 1

if i >= j:

return j

# If an element at i (on the left of the pivot) is larger than the

# element at j (on right right of the pivot), then swap them

nums[i], nums[j] = nums[j], nums[i]

def quick\_sort(nums):

# Create a helper function that will be called recursively

def \_quick\_sort(items, low, high):

if low < high:

# This is the index after the pivot, where our lists are split

split\_index = partition(items, low, high)

\_quick\_sort(items, low, split\_index)

\_quick\_sort(items, split\_index + 1, high)

\_quick\_sort(nums, 0, len(nums) - 1)

# Verify it works

random\_list\_of\_nums = [22, 5, 1, 18, 99]

quick\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

Сложность времени

В худшем случае в качестве опорного всегда выбирается самый маленький или самый большой элемент. Это создаст разделы размером n-1 , вызывая рекурсивные вызовы n-1 раз. Это приводит нас к наихудшей временной сложности O(n^2) .

Хотя это ужасный худший случай, быстрая сортировка широко используется, потому что ее средняя временная сложность намного быстрее. Хотя partitionфункция использует вложенные whileциклы, она сравнивает все элементы массива, чтобы произвести обмен. Таким образом, он имеет временную сложность O(n) .

При хорошем повороте функция быстрой сортировки разделит массив на половинки, которые растут логарифмически с увеличением n . Следовательно, средняя временная сложность алгоритма быстрой сортировки составляет O(nlog(n)) .

Встроенные функции сортировки Python

Хотя полезно понимать эти алгоритмы сортировки, в большинстве проектов Python вы, вероятно, будете использовать функции сортировки, уже предусмотренные в языке.

Мы можем изменить наш список, чтобы его содержимое было отсортировано с помощью sort()метода:

apples\_eaten\_a\_day = [2, 1, 1, 3, 1, 2, 2]

apples\_eaten\_a\_day.sort()

print(apples\_eaten\_a\_day) # [1, 1, 1, 2, 2, 2, 3]

Или мы можем использовать sorted()функцию для создания нового отсортированного списка:

apples\_eaten\_a\_day\_2 = [2, 1, 1, 3, 1, 2, 2]

sorted\_apples = sorted(apples\_eaten\_a\_day\_2)

print(sorted\_apples) # [1, 1, 1, 2, 2, 2, 3]

Они оба сортируются в порядке возрастания, но вы можете легко сортировать в порядке убывания, установив reverseфлаг True:

# Reverse sort the list in-place

apples\_eaten\_a\_day.sort(reverse=True)

print(apples\_eaten\_a\_day) # [3, 2, 2, 2, 1, 1, 1]

# Reverse sort to get a new list

sorted\_apples\_desc = sorted(apples\_eaten\_a\_day\_2, reverse=True)

print(sorted\_apples\_desc) # [3, 2, 2, 2, 1, 1, 1]

В отличие от созданных нами функций алгоритма сортировки, обе эти функции могут сортировать списки кортежей и классов. Функция sorted()может сортировать любой итерируемый объект, включая списки, строки, кортежи, словари, наборы и настраиваемые итераторы , которые вы можете создать.

Эти функции сортировки реализуют алгоритм сортировки Тима , основанный на сортировке слиянием и сортировке вставками.

Сравнение скорости

Чтобы получить представление о том, насколько быстро они работают, мы создаем список из 5000 чисел от 0 до 1000. Затем мы измеряем, сколько времени требуется для завершения каждого алгоритма. Это повторяется 10 раз, чтобы мы могли более надежно установить закономерность производительности.

Это были результаты, время в секундах:

Бегать Пузырь Выбор Вставка куча Объединить Быстрый

1 5.53188 1.23152 1.60355 0,04006 0,02619 0,01639

2 4.92176 1.24728 1.59103 0,03999 0,02584 0,01661

3 4.91642 1.22440 1,59362 0,04407 0,02862 0,01646

4 5.15470 1.25053 1,63463 0,04128 0,02882 0,01860

5 4.95522 1,28987 1.61759 0,04515 0,03314 0,01885

6 5.04907 1.25466 1.62515 0,04257 0,02595 0,01628

7 5.05591 1.24911 1.61981 0,04028 0,02733 0,01760

8 5.08799 1.25808 1.62603 0,04264 0,02633 0,01705

9 5.03289 1.24915 1.61446 0,04302 0,03293 0,01762

10 5.14292 1.22021 1,57273 0,03966 0,02572 0,01606

Среднее 5.08488 1.24748 1,60986 0,04187 0,02809 0,01715

Вы получите другие значения, если настроите тест самостоятельно, но наблюдаемые закономерности должны быть одинаковыми или похожими. Пузырьковая сортировка — самый медленный и худший из всех алгоритмов. Хотя это полезно в качестве введения в сортировку и алгоритмы, оно не подходит для практического использования.

Мы также заметили, что быстрая сортировка работает очень быстро, почти в два раза быстрее, чем сортировка слиянием, и для ее работы не требуется столько места. Напомним, что наш раздел был основан на среднем элементе списка, разные разделы могли иметь разные результаты.

Поскольку сортировка вставками выполняет гораздо меньше сравнений, чем сортировка выбором, реализации обычно выполняются быстрее, но в этих запусках сортировка выбором работает немного быстрее.

Сортировка вставками делает гораздо больше обменов, чем сортировка выбором. Если замена значений занимает значительно больше времени, чем сравнение значений, то этот «противоположный» результат будет правдоподобным.

Помните об окружающей среде при выборе алгоритма сортировки, так как это повлияет на производительность.

Заключение

Алгоритмы сортировки дают нам много способов упорядочить данные. Мы рассмотрели 6 различных алгоритмов — сортировку пузырьком, сортировку выбором, сортировку вставками, сортировку слиянием, сортировку кучей, быструю сортировку — и их реализации в Python.

Количество сравнений и перестановок, которые алгоритм выполняет вместе со средой, в которой работает код, являются ключевыми факторами, определяющими производительность. В реальных приложениях Python рекомендуется использовать встроенные функции сортировки Python из-за их гибкости при вводе и скорости.