**Сортировки**

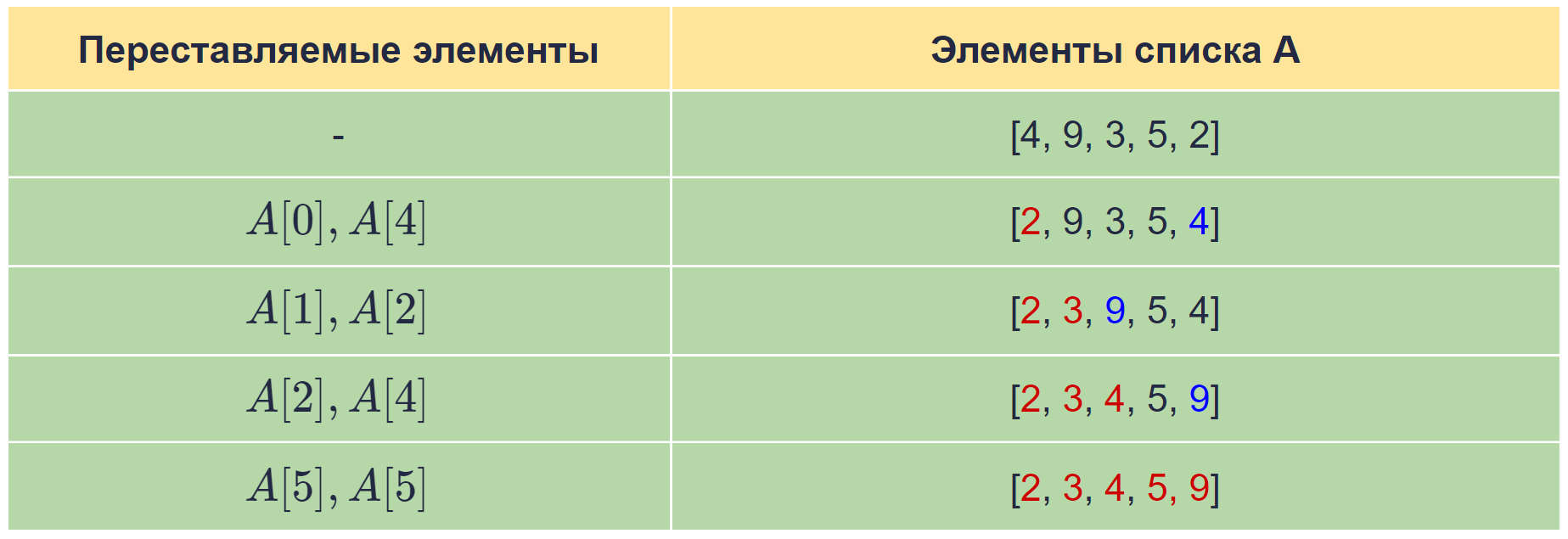
*1. Сортировка выбором*

Будем выбирать для каждой позиции, начиная с нулевой, элемент, который должен на ней стоять. Для нулевого элемента это минимальный элемент во всём списке. Для первого элемента — минимальный из всех оставшихся (то есть кроме нулевого, который уже стоит на своём месте). Для i-го — минимальный элемент списка с индексом больше или равным i. На последнем шаге мы установим на место предпоследний элемент (последний автоматически окажется на нужном месте). Таким образом мы отсортируем массив.

Рассмотрим на примере списка:

[4,9,3,5,2]

Проиллюстрируем работу алгоритма таблицей:



Оценим количество операций сравнения у данного алгоритма. Минимум в списке из n элементов можно найти проходом по всем элементам за n−1 действий, следующий за ним элемент за n−2 действий и так далее. Отсюда следует, что количество операций вычисляется следующей формулой:

(n−1)+(n−2)+(n−3)+…+2+1=n(n−1)/2

Из этого следует, что вычислительная сложность алгоритма — O(n2).

Однако, алгоритм весьма эффективен по количеству операций обмена элементов. Их будет не более n−1. Отсюда следует, что сортировка выбором может быть полезна при упорядочивании больших объектов.

**Реализация алгоритма**

def selection\_sort(a):

for i in range(len(a) - 1):

imin = i

for j in range(i + 1, len(a)):

if a[j] < a[imin]:

imin = j

a[i], a[imin] = a[imin], a[i]

2. Сортировка пузырьком

Обозначим элементы исходного списка a0, a1, …, an−1.

Теперь будем просматривать слева направо все пары соседних элементов: a0 и a1, a1 и a2, …, an−2 и an−1. Если ai > ai+1, то элементы меняем местами. В результате такого просмотра массива максимальный элемент окажется на крайнем справа (своём) месте. Об остальных элементах ничего определённого сказать нельзя. Будем просматривать массив снова, исключив из рассмотрения правый элемент. На своём месте теперь окажется уже второй по величине элемент. И так далее до тех пор, пока список не станет отсортирован.

Рассмотрим на примере списка:

[5,2,9,3,4]

Проиллюстрируем работу алгоритма таблицей:



Заметим, что если за один проход алгоритма не выполнилось ни одной операции перестановки элементов местами, то массив уже отсортирован и можно закончить выполнение алгоритма.

Оценим сложность пузырьковой сортировки. На первом шаге выполняется  n−1 операций сравнения, на втором — n−2 и так далее до 1 операции сравнения на (n−1)-м шаге. Получим формулу: (n−1) + (n−2) + ... + 2 + 1 = n(n−1)/2

Следовательно, вычислительная сложность алгоритма — O(n2).

Однако, если массив "близок к упорядоченному", то алгоритм сортировки пузырьком будет выполнять меньше сравнений: как только на каком-то шаге не будет произведено ни одного обмена, алгоритм можно завершать.

Реализация:

**def** bubble\_sort(a):

n = len(a)

unordered = True

**while** unordered:

unordered = False

**for** j **in** range(n - 1):

**if** a[j] > a[j + 1]:

a[j], a[j + 1] = a[j + 1], a[j]

unordered = True

n -= 1

3. Сортировка вставками

Будем сортировать список, добавляя в рассмотрение его элементы по одному, в том порядке в каком они изначально находятся. Изначально имеем отсортированный список из одного элемента. Затем берём следующий элемент и вставляем его на нужное место в уже отсортированном списке, сдвигая на один всех соседей справа. На каждом шаге мы будем увеличивать отсортированную область на один элемент. Такой алгоритм называется **сортировка вставками**. Именно она часто используется в жизни при сортировке материальных предметов.

Рассмотрим на примере списка:

[5,3,9,2,4].

Проиллюстрируем работу алгоритма таблицей:



Особенностью алгоритма сортировки вставками является то, что он работает быстро на упорядоченном или близком к упорядоченному массиве. Например, если мы запустим его на уже отсортированном массиве, то алгоритм выполнит ровно n−1 сравнение.

Однако, если мы рассмотрим худший случай и запустим алгоритм на упорядоченном по убыванию массиве, то он выполнит 1 + 2 + ... + (n−2) + (n−1) = n(n−1)/2 операций. Следовательно, асимптотика алгоритма составляет O(n2).

**Реализация**

**def** insertion\_sort(a):

**for** i **in** range(1, len(a)):

tmp = a[i]

j = i - 1

**while** j >= 0 **and** a[j] > tmp:

a[j + 1] = a[j]

j -= 1

a[j + 1] = tmp

4. Сортировка подсчётом

Пусть нам надо отсортировать массив, состоящий только из десятичных цифр. Оказывается, в этом случае существует более эффективный алгоритм сортировки по сравнению с рассмотренными ранее. А именно подсчитаем во вспомогательном массиве cnt количество вхождений каждой из цифр в исходный массив a. Сделать это можно за один проход массива a. А затем заполним массив a заново согласно значениям массива cnt.

Рассмотрим на примере массива:

[2,3,5,1,3,2,0,9]

После подсчёта количества каждой из цифр в массиве cnt будут храниться следующие значения:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| *cnt[i]* | 1 | 1 | 2 | 2 | 0 | 1 | 0 | 0 | 0 | 1 |

Далее, пройдя с помощью индекса i по массиву cnt, заполним массив a заново, а именно запишем в него cnt[i] раз число i. Получим:

[0,1,2,2,3,3,5,9]

Можно использовать рассмотренную идею не только для сортировки десятичных цифр, но и для сортировки массива, элементам которого можно поставить в соответствие числа от 0 до M−1.

Очень удобно использовать метод подсчёта для массивов символов. В таком случае каждому символу можно поставить в соответствие его ASCII-код. Если это строчные латинские символы, то их можно отображать в массив от 0 до 25.

Несложно понять, что вычислительная сложность полученного алгоритма — O(N+M), где N — количество элементов в массиве a, а M — количество возможных значений, которые встречаются в нём. Также алгоритм зависит от M и по количеству используемой памяти, поэтому его полезно использовать, только когда M сравнительно небольшое (в первую очередь когда M≤N).

Реализация:

Приведём основной фрагмент программы сортировки массива цифр:

cnt = [0] \* 10

**for** el **in** a:

cnt[el] += 1

a = []

**for** d **in** range(len(cnt)):

a += [d] \* cnt[d]

5. Сортировка слиянием

Кроме квадратичных универсальных сортировок существуют и более эффективные сортировки со сложностью O(n log n), где n — количество элементов в массиве.

Идея заключается в следующем: рассмотрим сортируемый список. Если он состоит из одного элемента, то он уже отсортирован. В противном случае разобьём его на две примерно равные (с точностью до единицы) части и отсортируем их, рекурсивно применив для каждой из частей аналогичную сортировку. После этого начнём «сливать» отсортированные части. Для этого будем хранить индексы начала двух отсортированных частей. Новый список будет создаваться следующим образом: мы будем выбирать меньший из двух элементов, на которые указывают индексы, и дописывать его в новый список. После этого сдвинем индекс, элемент которого мы взяли, на следующий элемент этого списка. Когда у нас закончится одна из сливаемых частей, то элементы второй мы просто добавим в конец итогового списка. В итоге получим отсортированный список, который надо поместить на место сливаемых частей.

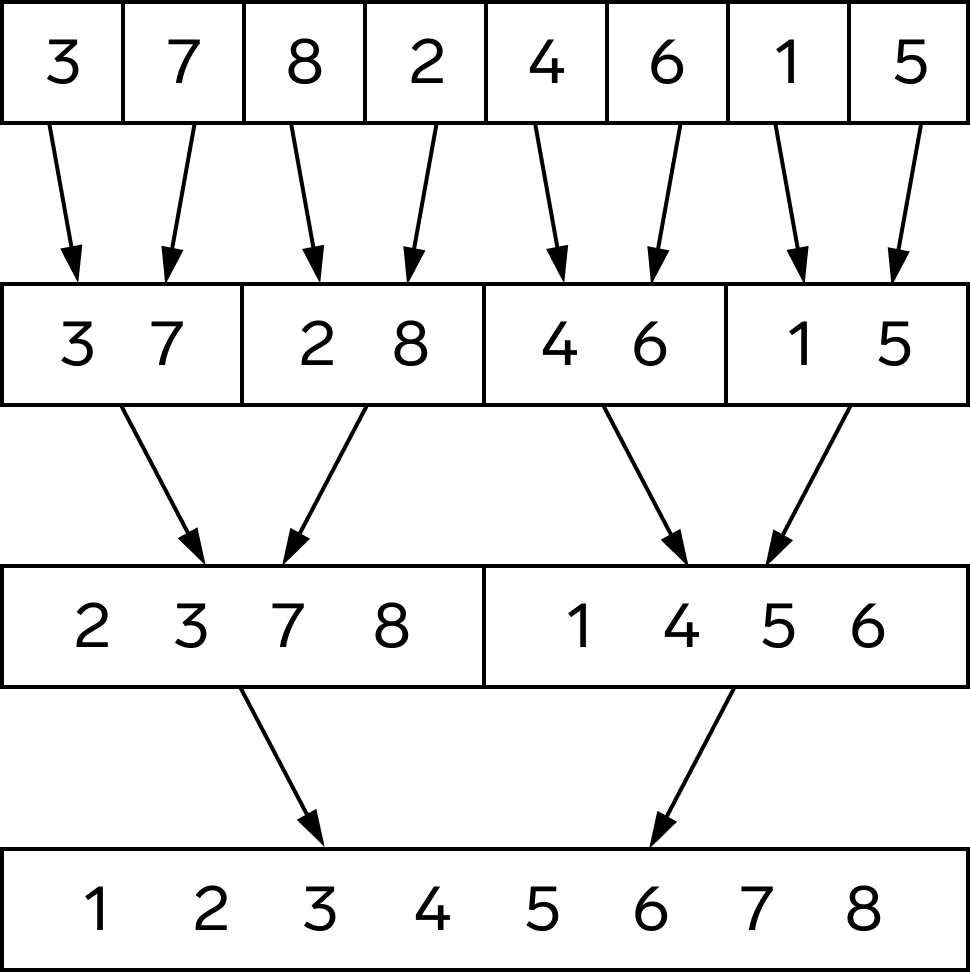
Рассмотрим работу процедуры "слияния" двух отсортированных частей на примере:

[4,5,5,7,8] и [2,4,6,8,9]



Осталось только разобраться, как упорядочить каждую из двух половинок исходного массива. Сделаем это также при помощи сортировки слиянием. Разобьём эти части на две, упорядочим их и сольём результат вместе. Меньшие части в свою очередь также разбиваются на две ещё меньшие части, которые упорядочиваются тем же алгоритмом и затем сливаются.

Проиллюстрируем эту идею:



Таким образом, сортировка слиянием — это рекурсивный алгоритм, который будет разбивать массив на две части и упорядочивать две меньшие части тем же самым алгоритмом, а затем сливать результат вместе.

Реализация:

**def** merge(a, b):

res = []

i = 0

j = 0

**while** i < len(a) **and** j < len(b):

**if** a[i] <= b[j]:

res.append(a[i])

i += 1

**else**:

res.append(b[j])

j += 1

res.extend(a[i:])

res.extend(b[j:])

**return** res

**def** msort(a):

**if** len(a) <= 1:

**return** a

k = len(a) // 2

**return** merge(msort(a[:k]), msort(a[k:]))

Оценим сложность работы алгоритма сортировки слиянием. Сложность алгоритмов сортировки обычно измеряют в количестве наиболее типичных операций — сравнений элементов массива.

Изучим ход нашего алгоритма от самого внешнего рекурсивного вызова. Мы разбили массив из n элементов на два массива из n/2 элементов, а затем их слили. Слияние двух массивов выполняется за время, пропорциональное сумме их длин, то есть у нас будет не более n операций сравнения.

На следущем уровне рекурсии мы делим массивы длиной n/2 на массивы длиной n/4 и затем объединяем 4 массива длиной n/4 попарно в два массива длиной n/2. Для этого понадобится также число сравнений, равное суммарной длине исходных массивов, то есть тоже n операций сравнения.

На следующем уровне рекурсии массивы размером n/8 объединяются попарно в массивы размером n/4, и для этого тоже понадобится суммарно n операций сравнения.

На каждом уровне рекурсии общее число операций равно n. А чтобы определить количество уровней рекурсии, нужно заметить, что размер одного кусочка на следующем шаге сокращается в два раза и рекурсия остановится, когда размер кусочка станет равен 1. Это произойдёт, когда будет выполнено не более, чем ⌈log2n⌉ шагов. То есть общее число операций будет равно

O(n log(n)).

Это гораздо лучше, чем рассмотренные раньше квадратичные алгоритмы сортировки, сложность которых равна O(n2). Можно доказать, что не существует универсального алгоритма сортировки, который требует меньше, чем O(n log(n)) сравнений. Под универсальным алгоритмом подразумевается алгоритм, который только сравнивает элементы массива между собой и не использует иной информации о свойствах элементов массива.

Может показаться, что сортировка подсчётом имеет меньшую сложность, то есть работает быстрее, но при этом сортировка подсчётом не является универсальным алгоритмом сортировки, поскольку применима только для массивов, набор значений в которых ограничен.

6. Быстрая сортировка (Quicksort, сортировка Хаора)

Во многом идея быстрой сортировки такая же, как у алгоритма сортировки слиянием. Выберем некоторый элемент q, называемый барьерным элементом. Разобьем массив на две части, переупорядочив его элементы. В первой части соберем элементы, меньшие или равные q, а во второй части — большие или равные q. Теперь достаточно отсортировать обе части, после чего выполнить их конкатенацию безо всякого дополнительного слияния.

Реализация:

**def QuickSort(A, l, r):**

**if l >= r:**

**return**

**else:**

**q = random.choice(A[l:r + 1])**

**i = l**

**j = r**

**while i <= j:**

**while A[i] < q:**

**i += 1**

**while A[j] > q:**

**j -= 1**

**if i <= j:**

**A[i], A[j] = A[j], A[i]**

**i += 1**

**j -= 1**

**QuickSort(A, l, j)**

**QuickSort(A, i, r)**

Сложность алгоритма быстрой сортировки Хоара зависит от метода выбора барьерного элемента. В лучшем случае при каждом выборе барьерного элемента должен выбираться медианный элемент массива. Но поиск медианного элемента — сложная задача, её нельзя решить быстро. Если выбрать первый элемент фрагмента списка A[l] или последний A[r], то если список A уже упорядочен, сложность алгоритма будет O(n2), так как на каждом рекурсивном вызове от большей части списка будет отделяться всего один элемент.

Поэтому в алгоритме быстрой сортировки Хоара, как правило, в качестве барьерного элемента выбирается случайный элемент списка. Тогда алгоритм становится вероятностным — время его работы зависит от того, каким будет случайно выбранный элемент. Возможна (но крайне маловероятна) ситуация, когда всегда будет выбираться наименьший элемент, и в этом случае алгоритм будет работать за O(n2).

В теории вероятностей доказывается, чти при случайном выборе элемента списка и разбиении его на две части, размер большей из двух получившихся частей будет в среднем равен 3n/4. В этом случае глубина рекурсии в среднем будет составлять порядка log n, а средняя сложность алгоритма быстрой сортировки Хоара — O(n log n)