Сортировки

Задача сортировки массива, то есть перестановки элементов массива так, чтобы они были упорядочены по возрастанию, убыванию или другой аналогичной характеристике, является одной из основных технических задач программирования. С этой задачей мы сталкиваемся и при записи фамилий учеников в классном журнале, и при подведении итогов соревнований. Рассмотрим простые алгоритмы сортировки массивов и подсчитаем их вычислительную сложность. Научитесь быстро реализовывать один из алгоритмов сортировки, чтобы в дальнейшем в случае необходимости не тратить время на его отладку, так как очень часто сортировка используется как первый шаг в алгоритме решения более сложной задачи.

Традиционно наиболее простой в реализации считается так называемая *«пузырьковая» сортировка*. Суть ее в случае упорядочения по неубыванию (то есть возрастанию в случае различных элементов) заключается в следующем. Будем просматривать слева направо все пары соседних элементов: a_1 и a_2, a_2 и a_3, …, a_{n-1} и a_n.

Если при этом $a_i > a_{i+1}$, то элементы меняем местами. В результате такого просмотра массива максимальный элемент окажется на крайнем справа (своем) месте. Об остальных элементах ничего определенного сказать нельзя. Будем просматривать массив снова, исключив из рассмотрения правый элемент. На своем месте теперь окажется уже второй по величине элемент. И так далее. В последнем просмотре будут участвовать только первый и второй элементы. Общее число просмотров при этом равно $n - 1$.

При оценке вычислительной сложности алгоритмов сортировки обычно отдельно подсчитывают количество операций сравнения и количество операций присваивания, так как заранее неизвестно, какая из них окажется более трудоемкой, ведь сравниваться между собой могут не только числа, но и строки, и другие достаточно сложные объекты. Иногда обе операции сопоставимы по трудоемкости.

Количество сравнений в данном алгоритме равно

$$(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{n(n-1)}{2}.$$

В худшем случае (когда изначально массив упорядочен по убыванию) обмен будет производиться после каждого сравнения и общее число обменов будет также равно

$$\frac{n(n-1)}{2}.$$

Контрольный вопрос 1. Чему равно количество присваиваний, которые при этом совершаются (это особенно актуально для других языков программирования, где операций обмена нет).

Говорят, что данный алгоритм квадратичный как по числу сравнений, так и по числу присваиваний. «Пузырьковым» он называется потому, что в результате каждого просмотра максимальный из оставшихся элементов оказывается на своем месте — «всплывает». По-другому такая сортировка называется обменной.

Рассмотрим алгоритм сортировки, основанный на принципиально иной идее. Будем выбирать для каждой позиции, начиная с нулевой, элемент, который должен на ней стоять. Для нулевого элемента – это минимальный элемент во всём списке. Будем сравнивать его с каждым другим элементом и, если очередной элемент будет меньше, чем наш, тогда будем менять их местами. Когда мы дойдём до конца массива, то на нулевой позиции у нас окажется искомый элемент. Продолжим таким же образом для первого элемента, для него мы будем перебирать, начиная со второго и до конца. На последнем шаге мы установим на место предпоследний элемент (последний автоматически окажется на нужном месте). Таким образом, мы отсортируем массив.

Данный алгоритм называется сортировка выбором. Суммарное количество сравнений считается по аналогичной форме:

$$(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n(n-1)}{2}.$$

Количество обменов в худшем случае также равно:

$$\frac{n(n-1)}{2}.$$

Вот пример его реализации на P*ython*:

```
arr = list(map(int, input().split()))
for i in range(len(arr) - 1):
    for j in range(i + 1, len(arr)):
        if arr[i] > arr[j]:
            arr[i], arr[j] = arr[j], arr[i]
```

На наш взгляд, эта программа проще, чем та что реализует пузырьковый алгоритм, поэтому на письменных экзаменах, а также при необходимости быстро написать код сортировки мы рекомендуем именно ее.

Контрольный вопрос 2. Что произойдет с массивом, если внутренний цикл данной сортировки выполнять от 0, а не от $i + 1$? Попробуйте ответить на этот вопрос не запуская исправленную программу.

Рассмотрим немного изменённый алгоритм. Будем находить минимальный элемент в массиве и только затем менять его с нулевым элементом массива. В результате он окажется на своем месте. Затем найдем минимальный элемент среди оставшихся, и поменяем его с первым элементом. На $(n-1)$-м шаге мы закончим упорядочивание нашего массива. Такой алгоритм называется сортировкой *прямым выбором*.

Количество выполняемых операций в данном алгоритме всегда одинаково. Для сравнений оно равно

$$(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n(n-1)}{2},$$

а для обменов — всего $(n-1)$.

Итак, данный алгоритм квадратичный по числу сравнений и линейный (эффективный) по числу присваиваний. Заметим, что это самый эффективный по числу обменов из существующих алгоритмов.

Рассмотрим другой алгоритм. Начнём сортировать список, добавляя по одному элементу. 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*.

Заметим, что в отсортированном массиве не будет выполнено ни одного присваивания и будет выполнено всего n сравнений. В худшем же случае, по аналогичным формулам получим количество сравнений и обменов:

$$(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n(n-1)}{2}.$$

Эффективность этого алгоритма можно улучшить, если искать место для вставки очередного элемента двоичным поиском, который будет рассмотрен в нашем курсе позднее. Вам же

достаточно реализовать самый простой вариант алгоритма сортировки вставками. В нем каждый элемент, начиная со второго, последовательно сравнивается с соседями слева и, если оказывается меньше очередного элемента, то меняется с ним местами. Приведем заготовку для соответствующего цикла:

```
while arr[i] < arr[i-1]:
    arr[i], arr[i-1] = arr[i-1], arr[i-1]
    i -= 1
```

В данном цикле намеренно допущена одна ошибка: если вставляемый элемент меньше всех ранее стоящих элементов, то индекс выйдет за границу списка. Исправьте эту ошибку самостоятельно.

Пусть теперь нам надо отсортировать массив, состоящий из десятичных цифр. Оказывается, в этом случае существует гораздо более эффективный алгоритм сортировки по сравнению с рассмотренными выше. А именно, подсчитаем во вспомогательном массиве *d* количество вхождения каждой из цифр в исходный массив *a*. Сделать это можно за один проход массива *a*. А затем заполним массив *a* заново согласно значениям массива *d*. Причем именно в Python это можно сделать тривиальной формулой:

```
[0]*d[0] + [1]*d[1] + … + [9]*d[9]
```

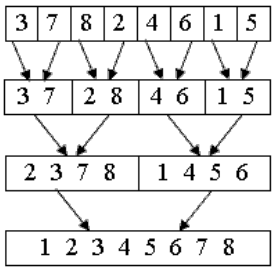
Реализовывать эту формулу удобнее в цикле, особенно в случае, когда подсчитываются произвольные элементы. Заметим, что этот алгоритм работает верно и тогда, когда какой-либо из цифр во вводимой последовательности нет совсем. Подобный алгоритм называют *сортировкой подсчетом*. Количество операций в этом алгоритме пропорционально $N + M$, где N – количество элементов в списке, а M – количество возможных значений, которые встречаются в нем. Алгоритм особенно эффективен, если $M \leq N$.

Рассмотрим последний алгоритм в этом уроке: *сортировка слиянием*. 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 на 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.

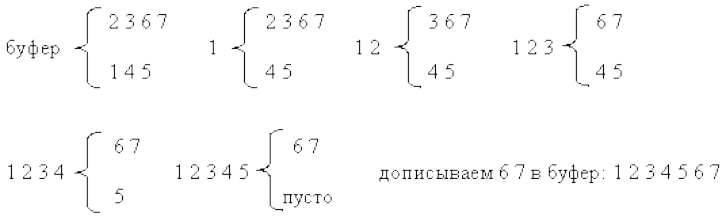
Пример реализации этого алгоритма, записанной на русском языке (вам следует записать этот алгоритм на вашем языке программирования).

```
# Arr – глобальный массив, который мы сортируем
Функция Sort(L, R)
# параметрами являются индексы начала и конца сортируемой части массива
    Если L = R то массив отсортирован
    Иначе
        Sort(L, (L+R)//2)
        Sort((L+R)//2+1, R)
        слить отсортированные части в один список
        поместить полученный список на места с L по R в исходном списке
конец
```

Пример разбиения, которое получится в результате отображения всех уровней рекурсии:



Пример слияния последовательностей 2 3 6 7 и 1 4 5:



Сложность алгоритма: количество уровней рекурсии равно округленному вверх значению $\log_2 n$ (т.е. такому минимальному числу k, что $2^k \geq n$), на каждом уровне, в худшем случае, мы совершим примерно n сравнений, следовательно, суммарное количество сравнений будет:

$$n\log_2 n$$

Аналогично, на каждом уровне мы будем записывать каждый элемент дважды, суммарное количество присваиваний:

$$2n\log_2 n$$

Как видно из показателей, эта сортировка для произвольного списка выполняется быстрее рассмотренных ранее сортировок. Она является примером, так называемых, эффективных универсальных алгоритмов сортировки (сортировка подсчетом универсальной не является).

При выполнении заданий этого урока, в которых требуется реализовать определенный алгоритм, необходимо реализовать именно его. Встроенными алгоритмами при этом пользоваться нельзя. При решении задач на применение сортировки, наоборот, можно использовать не только любой из реализованных ранее алгоритмов, но и встроенные сортировки. Так, в языке Python список по неубыванию можно отсортировать так:

```
Arr.sort()
```

А по невозрастанию – так:

```
Arr.sort(reverse = True)
```

Можно также присвоить другому списку копию отсортированного исходного:

```
Arr2 = sorted(Arr1)
```

Все упомянутые операции являются эффективными.