Вычислительная сложность - это зависимость объема работы алгоритма от размера входных данных. Сложность может быть временной и пространственной. Временная сложность оценивает время работы алгоритма, а именно выражает зависимость количества выполняемых элементарных операций от размера входных данных. По аналогии с временной сложностью определяют пространственную сложность, только здесь говорят не о количестве элементарных операций, а об объеме используемой памяти.

Несмотря на то, что функция временной сложности алгоритма в некоторых случаях может быть определена точно, в большинстве случаев искать точное её значение бессмысленно. Точное значение временной сложности зависит от определения "элементарных операций" (например, сложность можно измерять в количестве арифметических операций или битовых операций), и при увеличении размера входных данных вклад постоянных множителей и слагаемых низших порядков, фигурирующих в выражении для точного времени работы, становятся незначительными.

Рассмотрение входных данных большого размера и оценка порядка роста времени работы алгоритма приводят к понятию асимптотической сложности алгоритма. Для их записи используются асимптотические обозначения:

Обозн.	Объяснение
$f(n)=O(g(n))$	f ограничена сверху функцией g (с точностью до постоянного множителя) асимптотически
$f(n)=\Omega(g(n))$	f ограничена снизу функцией g (с точностью до постоянного множителя) асимптотически
$f(n)=\Theta(g(n))$	f ограничена снизу и сверху функцией g асимптотически

Сортировка пузырьком

Сортировка пузырьком - метод сортировки массивов путем последовательного сравнения и обмена соседних элементов, если предшествующий оказывается больше последующего. В процессе выполнения данного алгоритма элементы с большими значениями оказываются в конце списка, а элементы с меньшими значениями постепенно перемещаются по направлению к началу списка. Образно говоря, тяжелые элементы падают на дно, а легкие медленно всплывают подобно пузырькам воздуха.

```
def bubble_sort(a):
    for i in range(len(a)-1):
        for j in range(len(a)-i-1):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
```

Сортировка слиянием

Алгоритм сортировки слиянием основан на идее, что два отсортированных списка можно слить в один отсортированный список за время, равное суммарной длине этих списков.

Для этого сравним первые элементы данных списков. Тот элемент, который меньше, скопируем в конец результирующего списка (который первоначально пуст) и в этом списке перейдем к следующему элементу. Будем повторять этот процесс (выбираем из начала двух списков наименьший элемент, копируем его в результирующий список), пока один из исходных списков не кончится. После этого оставшиеся элементы (один из двух исходных списков будет непуст) скопируем в результирующий список.

Для того, чтобы не удалять начальные элементы из списков, заведем два индекса i и j, указывающие на текущие элементы в каждом списке. Вместо удаления элементов будем передвигать эти индексы. В конце добавим к результирующему списку оставшиеся элементы из двух исходных списков A[i:] + B[j:] (один из этих срезов будет пустым, это не должно нас смущать).

```
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 += A[i:] + B[j:]
    return Res
```

Заметим, что сложность работы функции merge — линейная от суммарных длин списков A и B, так как каждый элемент обрабатывается ровно один раз за O(n).

Теперь можно реализовать сортировку слиянием. Это — рекурсивная функция, которая получает на вход исходный список и список, составленный из тех же элементов, но отсортированный. Если длина исходного списка равна 1 или 0, то он уже отсортирован и сортировать его не надо. Если же длина списка больше 1, то разобьем его на две части равной (или почти равной, если длина исходного списка — нечетная). Отсортируем обе эти части, затем сольем их вместе при помощи функции merge.

```
def MergeSort(A):
    if len(A) <= 1:
        return A
    else:
        L = A[:len(A) // 2]
        R = A[len(A) // 2:]
    return merge(MergeSort(L), MergeSort(R))
```

Оценим сложность этого алгоритма. Пусть массив содержит n элементов. Тогда за O(n) его можно разделить на две части и после сортировки слить их вместе. Каждая из

этих двух частей имеет размер n/2, и за O(n) шагов каждую из них можно поделить на две части размером n/4 и затем после сортировки слить их вместе. Аналогично, четыре части размером n/4 за суммарное O(n) шагов делятся на части размером n/8 и сливаются вместе. Этот процесс «в глубину» продолжается столько раз, сколько раз можно число n делить на 2, до тех пор, пока размер части не станет равен 1, то есть $\log_2 n$. Итого, общая сложность этого алгоритма равна O($n \log_2 n$).

Одним из недостатков сортировки слиянием является тот факт, что он требует много вспомогательной памяти (столько же, каков размер исходного массива) для реализации.

Быстрая сортировка

В попытках улучшить такой алгоритм появилась «быстрая сортировка» (или, как ещё ее называют, сортировка Хоара).

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.

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(n^2), 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(n^2).

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($n \log n$).

```
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)
```

Линейные сортировки

Иногда сортируемые элементы имеют небольшой диапазон возможных значений. Например, если необходимо отсортировать список натуральных чисел, каждое из которых не более трех-пяти разрядов в длину. Однако количество сортируемых чисел настолько велико, что нужно обработать список за линейное время. В этом случае разумно использовать сортировку подсчётом.

Сортировка подсчётом

Исходная последовательность чисел длины n, а в конце отсортированная, хранится в массиве A. Также используется вспомогательный массив C с индексами от 0 до k-1 изначально заполняемый нулями.

- Последовательно пройдём по массиву A и запишем в C[i] количество чисел, равных i.
- Теперь достаточно пройти по массиву C и для каждого числа number из диапазона допустимых значений последовательно записать в массив A число number C[number] раз.

```
def SimpleCountingSort(A):
    scope = max(A) + 1
    C = [0] * scope
    for x in A:
        C[x] += 1
    A[:] = []
    for number in range(scope):
        A += [number] * C[number]
```

Можно ли лучше? Был придуман также интересный способ сортировки – поразрядная сортировка.

Поразрядная сортировка

Применение поразрядной сортировки имеет одно ограничение: перед началом сортировки необходимо знать

length – максимальное количество разрядов в сортируемых величинах (например, при сортировке слов необходимо знать максимальное количество букв в слове),

rang – количество возможных значений одного разряда (при сортировке слов – количество букв в алфавите).

Для сортировки чисел второе число заранее известно: количество цифр равно основанию системы счисления. Поскольку в большинстве случаев сортируемые числа состоят из небольшого количества разрядов, поразрядная сортировка мало того, что применима, но и работает сравнительно быстро.

1. Создаем пустые списки, количество которых равно числу rang.

2. Распределяем исходные числа по этим спискам в зависимости от величины младшего разряда (по возрастанию).

3. Собираем числа в той последовательности, в которой они находятся после распределения по спискам.

4. Повторяем пункты 2 и 3 для всех более старших разрядов поочередно.

Объединив числа в список, получим отсортированный массив.

```
A = [12, 5, 664, 63, 5, 73, 93, 127, 432, 64, 34]
length = len(str(max(A)))
rang = 10
for i in range(length):
    B = [[] for k in range(rang)]
    for x in A:
        figure = x // 10**i % 10
        B[figure].append(x)
    A = []
    for k in range(rang):
        A = A + B[k]
print(A)
```