

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 5

Сортировка, поиск

Содержание

1. Сортировка	3
1.1. Сортировка пузырьком	6
1.2. Сортировка Шелла (Shell Sort)	11
1.3. Сортировка слиянием	17
1.4. Быстрая сортировка (Quick Sort)	21
1.5. Сравнение алгоритмов сортировки.....	28
1.5.1. Сортировка Шелла	37
1.5.2. Сортировка слиянием.....	41
1.5.3. Быстрая сортировка.....	45
2. Поиск.....	50
2.1. Линейный поиск	50
2.2. Бинарный поиск	54

1. Сортировка

Сортировка — это процесс упорядочения (размещения в определенном порядке) данных в некотором наборе.

Наиболее распространенные способы такого упорядочения для числовых данных:

- **сортировка по возрастанию** (меньшее число идет первым в наборе, а большее число после, правее меньшего числа);
- **сортировка по убыванию** (от наибольшего числа в наборе до наименьшего, при этом числа с большими значениями располагаются левее своих меньших «соседей»).

Например:

Исходный набор чисел (в произвольном порядке)

4	-2	1	-5	8
---	----	---	----	---

Набор, отсортированный по возрастанию

-5	-2	1	4	8
----	----	---	---	---

Набор, отсортированный по убыванию

8	4	1	-2	-5
---	---	---	----	----

Для текстовых данных используются сортировки в алфавитном порядке (лексикографическом) и в порядке обратном алфавитному:

Исходный набор текстовых данных (например, список имен клиентов)

Bob	Anna	Joe	Nick
-----	------	-----	------

Набор, отсортированный *в алфавитном порядке*

Anna	Bob	Joe	Nick
------	-----	-----	------

Набор, отсортированный *в порядке обратном алфавитному*

Nick	Joe	Bob	Anna
------	-----	-----	------

Важность сортировки заключается в том, что поиск данных оптимизируется (упрощается, ускоряется), а также данные представляются в более удобочитаемом (понятном) формате.

В примере выше мы отсортировали список клиентов, также частой задачей является отсортировать, например, список товаров, чтобы пользователю удобнее было просматривать, что есть в ассортименте. А список цен товаров традиционно сортируется от самых дешевых до самых дорогих или наоборот, т. е. нужна реализация сортировки по возрастанию и по убыванию цены.

Ранее мы уже использовали встроенные механизмы языка Python для сортировки: метод `sort()` для сортировки списков и функцию `sorted()` для всех остальных типов коллекций. Однако, если в наборе очень большое количество элементов, то сортировка встроенными механизмами может занять очень много времени.

Сегодня существует большое количество разных алгоритмов сортировки: одни являются очень простыми по реализации, другие — ориентированы на ускорение процесса сортировки, третьи — оптимизированы по используемым объемам памяти и т. д. Также постоянно разрабатываются новые методы сортировок, которые

чаще всего являются вариациями (усовершенствованием) существующих алгоритмов и возникают с целью улучшить их показатели.

Наиболее распространёнными на практике являются:

- сортировка пузырьком;
- сортировка слиянием;
- сортировка Шелла;
- быстрая сортировка.

Когда мы говорим об **оптимальности сортировки**, то имеется ввиду ее эффективность по объему используемой памяти (во время работы алгоритма) и по времени работы. Когда алгоритм оценивается по количеству памяти, затрачиваемой на выполнение алгоритма, то считается как вспомогательная память, используемая вычислительными процессами, так и память для хранения данных.

Временная сложность (также известна, как вычислительная сложность) — это время, за которое алгоритм выполнит задачу сортировки с учетом входных данных. Можно сказать, что это некоторая функция, которая описывает время работы алгоритма в зависимости от размера входных данных.

Чаще всего вычислительную сложность описывают с помощью заглавной буквы «O». «O» — это количество элементарных операций (простое присваивание, индексация элемента массива, арифметические операции, операции сравнения, логические операции), которые должен выполнить алгоритм сортировки (ведь каждая такая операция занимает определенное время). Такой подход дает нам возможность оценить, как будет изменяться

время выполнения алгоритма в зависимости от количества входных данных.

Когда мы говорим о вычислительной сложности то может быть три ситуации:

- наихудший случай (когда набор элементов отсортирован в противоположном порядке) — тогда каждый элемент находится не на своем месте и его нужно переместить, т. е. выполнить максимальное количество элементарных операций;
- средний случай (когда исходный порядок сортируемых входных данных произволен);
- наилучший случай (набор входных данных обеспечивает минимально возможное число элементарных операций, например, список уже отсортирован).

Выбирая алгоритм, мы часто фактически выполняем выбор между объёмом памяти и скоростью сортировки. Задачу сортировки можно решить быстро, используя большой объём памяти (например, задействуя дополнительные списки для промежуточного хранения), или медленнее, занимая меньший объём памяти, но при этом выполнив большее количество элементарных операций. Поэтому сравнивая различные алгоритмы, важно знать, как зависит объём используемых вычислительных ресурсов (объём памяти и время выполнения) от объёма входных данных.

1.1. Сортировка пузырьком

Bubble Sort (*Пузырьковая сортировка*) — один из самых простых для понимания и реализации алгоритмов сортировки. Его название происходит от особенностей

работы алгоритма: при каждом новом проходе элемент с самым большим (или маленьким, в зависимости от порядка сортировки: по возрастанию или по убыванию) значением в списке «поднимается вверх» — «всплывает» в конец списка.

Пары соседних элементов сравниваются между собой последовательно и, если порядок в паре нарушен, то меняются друг с другом местами.

Рассмотрим на примере сортировки списка из пяти элементов по убыванию. Индексация элементов на рисунке 1 идет снизу вверх, т. е. самый верхний элемент на рисунке — последний в списке. Желтым цветом на каждом шаге прохода выделяется сравниваемая пара элементов, а красным шрифтом — наименьший элемент в паре (т. к. сортировка по убыванию, то «всплывать» должны меньшие «пузырьки»).

6		6		6		6	обмен	2
7		7		7	обмен	2		6
4		4	обмен	2		7		7
2	нет обмена	2		4		4		4
5		5		5		5		5

Рисунок 1

За один (первый проход) у нас «всплыл» только один «пузырек» — наименьшее число — 2 после проверки четырех пар, т. е. количество пар на единицу меньше количества неотсортированных элементов. Мы прошли от 0-го элемента до $N-1$ -го элемента.

На следующем проходе нам нужно работать уже не со всем массивом (число 2, выделенное на рисунке 2 зеленым цветом, уже находится в правильной позиции).

2		2		2		2
6		6		6	обмен	4
7		7	обмен	4		6
4	нет обмена	4		7		7
5		5		5		5

Рисунок 2

На втором проходе мы проанализировали три пары, начиная от 0-го элемента до $N-2$ -го элемента. Уже на этом этапе можно заметить, что диапазон индексов неотсортированных элементов изменяется от 0 до $N-i$, где i — номер прохода.

Третий проход (рис. 3):

2		2		2
4		4		4
6		6	обмен	5
7	обмен	5		6
5		7		7

Рисунок 3

Четвертый проход (рис. 4):

2	
4	
5	
6	нет обмена
7	

Рисунок 4

Таким образом, мы отсортировали наш список за 4 прохода ($N-1$), на каждом из которых происходило

постепенное смещение элементов с меньшим значением в конец списка (вверх).

Пузырьковая сортировка состоит из нескольких проходов по списку, на каждом из которых запускается процесс анализа пар неотсортированной части списка. Сравнивается каждая пара соседних элементов и элементы меняются местами, если они расположены не в нужном порядке.

Здесь нам понадобятся два цикла. Внешний цикл будет выполнять проходы в количестве $N-1$, как в примере выше, который мы разобрали.

Внутренний цикл будет осуществлять сравнение пар в неотсортированной части списка (число итераций $N-i$). Если i -ый элемент меньше правого соседа ($i+1$ -го элемента), то они меняются местами, таким образом самый маленький элемент будет в правой крайней части списка.

Реализуем алгоритм пузырьковой сортировки в виде отдельной функции. Также создадим функцию для вывода элементов списка и вызовем ее для вывода исходного списка и отсортированного (после вызова функции сортировки).

```
numbers = [5,2,4,7,6]
```

```
def myBubbleSort(myList):  
  
    for i in range(len(myList)-1):  
        for j in range(len(myList)-i-1):  
            if myList[j]<myList[j+1]:  
                temp=myList[j]  
                myList[j]=myList[j+1]  
                myList[j+1]=temp
```

```
def printList(myList):
    for index, elem in enumerate(myList):
        print("element {}: {}".format(index+1, elem))
```

```
print("Original list:")
printList(numbers)
myBubbleSort(numbers)
print("Sorted list:")
printList(numbers)
```

Результат (рис. 5):

```
Original list:
element 1: 5
element 2: 2
element 3: 4
element 4: 7
element 5: 6
Sorted list:
element 1: 7
element 2: 6
element 3: 5
element 4: 4
element 5: 2
```

Рисунок 5

Однако, в случае, когда список уже отсортирован (например, изначально мы имеем 7, 6, 5, 4, 2), то наш алгоритм все равно выполнит сравнения всех пар на всех проходах.

Усовершенствуем его работу такой модификацией: если во время текущего прохода не было перестановок

(а это значит, что все элементы уже отсортированы), то прервем процесс сортировки (внешний цикл):

```
def myBubbleSort_v1(myList):
    for i in range(len(myList)-1):
        sortedFlag=True
        for j in range(len(myList)-i-1):
            if myList[j]<myList[j+1]:
                temp=myList[j]
                myList[j]=myList[j+1]
                myList[j+1]=temp
                sortedFlag=False
        if sortedFlag:
            break
```

1.2. Сортировка Шелла (Shell Sort)

Алгоритм сортировки **Шелла** — это усовершенствованная версия сортировки вставками. Поэтому вначале рассмотрим базовый алгоритм — сортировку вставками.

Как и пузырьковая сортировка, алгоритм сортировки вставками очень прост в реализации и легок в понимании. Основная идея — это виртуальное разделение списка на отсортированную часть (в начале списка) и неотсортированную. На каждом шаге алгоритм берет элемент из неотсортированной части и вставляет его в правильную позицию в отсортированной части списка.

Рассмотрим процесс сортировки списка из N элементов по возрастанию.

На первом шаге предполагается, что отсортированная часть списка состоит только из первого элемента (с индексом 0).

Далее идем по списку от `list[1]` до `list[N-1]` и сравниваем каждый элемент с его «предшественниками», расположенными левее.

Если текущий элемент меньше левого соседа, то сравниваем его с предыдущим левому соседу элементом. И так до тех пор, пока не найдем его правильную позицию в отсортированной левой части списка. Также нужно перемещать (сдвигать) большие элементы на одну позицию вверх (после каждого сравнения с результатом «текущий меньше левого соседа»), чтобы освободить место для вставки (рис. 6).

	0	1	2	3	4
1	4	3	2	10	12
	1 сдвиг				
2	3	4	2	10	12
	2 сдвига				
3	2	3	4	10	12
	остается				
4	2	3	4	10	12
	остается				

Рисунок 6

Зеленым цветом выделена отсортированная часть массива, которая изменяется (дополняется) на каждом шаге.

Красным цветом на рисунке показан текущий элемент, для которого мы на текущем шаге определяем правильную позицию (выделена желтым цветом).

Алгоритм Шелла улучшает среднюю временную сложность сортировки вставками, при которой элементы могут перемещаться только на одну позицию. В случае,

если нужная позиция в отсортированной части находится далеко от текущей позиции элемента, нужно много перемещений (сдвигов), которые увеличивают время выполнения сортировки.

Алгоритм Шелла позволяет перемещать и менять местами удаленные элементы. Этот алгоритм сначала сортирует элементы, которые находятся далеко друг от друга, а затем элементы, расположенные на более близком расстоянии. Это расстояние, в пределах которого проводится сортировка, называется интервалом.

В качестве интервалов для сортировки списка длиной N обычно используется такая последовательность: $N/2$, $N/4$, ..., 1.

Фактически на каждом шаге мы определяем набор списков, которые включают в себя элементы исходного списка, находящиеся на указанном интервале друг от друга. И проводим сортировку вставками внутри этих списков. Далее при сокращении интервала мы сокращаем количество этих списков (они становятся длиннее и более отсортированными). На последнем шаге (когда интервал равен 1) также используется сортировка вставками, но уже всего списка, большая часть которого уже отсортирована.

Рассмотрим на примере.

На первом шаге N равно 8 (число элементов в списке), поэтому элементы лежат друг от друга на интервале 4 ($N/2 = 4$) (рис. 7).

N=8	0	1	2	3	4	5	6	7
1	33	31	40	8	12	17	25	42
N/2=4								

Рисунок 7

Мы получили 4 списка: желтый, оранжевый, голубой и зеленый. Элементы внутри каждого из этих списков будут сравниваться и меняться местами, если они не в порядке.

Здесь в первом цикле элемент на 0-й позиции будет сравниваться с элементом на 4-й позиции. 0-й элемент больше, поэтому он будет заменен элементом на 4-й позиции. В противном случае (например, при сравнении 3-го и 7-го элементов) порядок остается прежним. Этот процесс продолжится для остальных элементов каждого из образованных списков.

Результат после первого прохода (рис. 8):

0	1	2	3	4	5	6	7
12	17	25	8	33	31	40	42

Рисунок 8

Далее во втором цикле элементы одного подсписка лежат на интервале 2 ($N/4 = 2$). Поэтому получим только два подсписка: желтый и голубой (рис. 9).

2	0	1	2	3	4	5	6	7
N/4=2	12	17	25	8	33	31	40	42

Рисунок 9

Каждый из этих подсписков сортируется алгоритмом вставки.

Результат после второго прохода (рис. 10):

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42

Рисунок 10

На третьем проходе элементы лежат друг от друга на интервале 1, т.е мы получаем один список длиной N, который сортируется также методом вставок (рис. 11):

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
сдвиг							
8	12	25	17	33	31	40	42
остаётся							
8	12	25	17	33	31	40	42
сдвиг							
8	12	17	25	33	31	40	42
остаётся							
8	12	17	25	33	31	40	42
сдвиг							
8	12	17	25	33	31	40	42
остаётся							
8	12	17	25	33	31	40	42
остаётся							

Рисунок 11

Как можно заметить сдвиги при сортировке вставками происходят на небольшое расстояние (на 1 элемент в нашем примере).

Создадим отдельную функцию для реализации сортировки вставками `InsertionSort()` для элементов, находящихся на расстоянии, равном интервалу, друг от друга и вызовем ее в цикле функции сортировки Шелла `ShellSort()`:

```
def ShellSort(myList):
    subListN = len(myList)//2
    step=1
    while subListN > 0:
        for startInd in range(subListN):
```

```

        InsertionSort(myList, startInd, subListN)
    print("Interval={}. After step {}: {}".format(subListN, step, myList))

    subListN = subListN // 2

def InsertionSort(myList, startInd, gapValue):
    for i in range(startInd+gapValue,
                    len(myList), gapValue):

        currentElem = myList[i]
        currentInd = i

        while currentInd >= gapValue and
              myList[currentInd-gapValue] >
              currentElem:
            myList[currentInd] =
                myList[currentInd-gapValue]
            currentInd = currentInd-gapValue

        myList[currentInd]=currentElem

numbers=[33,31,40,8,12,17,25,42]
print("Original list: {}".format(numbers))

ShellSort(numbers)
print("Sorted list: {}".format(numbers))

```

Результат (рис. 12):

```

Original list: [33, 31, 40, 8, 12, 17, 25, 42]
Interval=4. After step 1: [12, 17, 25, 8, 33, 31, 40, 42]
Interval=2. After step 1: [12, 8, 25, 17, 33, 31, 40, 42]
Interval=1. After step 1: [8, 12, 17, 25, 31, 33, 40, 42]
Sorted list: [8, 12, 17, 25, 31, 33, 40, 42]

```

Рисунок 12

1.3. Сортировка слиянием

Теперь мы рассмотрим алгоритм, который относится к категории быстрых алгоритмов сортировки. Эти алгоритмы предназначены для работы с большими наборами данных и обладают лучшими временными характеристиками, по сравнению с рассмотренными ранее алгоритмами.

В основе этой сортировки слиянием находится принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего, уже упорядоченные части, сливаются воедино. Данная процедура деления списка повторяется рекурсивно.

Если список пустой или содержит только один элемент, то он отсортирован (базовый случай — остановка рекурсивного разбиения). Иначе происходит рекурсивный вызов сортировки слиянием для каждой из половин отдельно.

Далее отсортированные фрагменты (которые находятся в отдельных списках) нужно собрать в правильном порядке — т. е. выполнить слияние. При этом на каждом шаге слияния из каждого из двух списков выбирается меньший элемент пары (если сортировка по возрастанию) и записывается в свободную левую ячейку дополнительного списка. В случае если один из списков закончится раньше, то элементы другого будут записаны (добавлены в конец) в результирующий список.

Опишем общие шаги рекурсивной сортировки, на вход которой подается список (или его части на последующих шагах рекурсии):

Если длина списка больше 1:

1. Найти точку для разбиения списка на 2 половины:
 $m = (\text{текущая длина списка})/2$,
2. Вызвать сортировку для первой (левой) половины списка (от начала до m -го элемента).
3. Вызвать сортировку для второй (правой) половины списка (от m -го элемента до последнего элемента списка).
4. Выполнить слияние двух отсортированных списков (которые были возвращены рекурсивными вызовами после шага 2 и шага 3):

4.1. Пока есть элементы в левом списке и в правом списке:

- Если текущий элемент левого списка меньше, чем текущий элемент из правого, то в результирующий список вставляется элемент левого списка, иначе — из правого списка.
- После вставки элемента из списка перейти к следующему его элементу, а для другого списка (из которого не было вставки) текущий элемент не меняется.

4.2. Проверяем, если в каком-то списке остался элемент, то добавляем его конец результирующего списка.

Рассмотрим на примере. На рисунке 13 показаны все 12 шагов сортировки слиянием: номера шагов, выделенные красным цветом шрифта, соответствуют шагам разбиения списка, а номера синего цвета — шагам слияния. Красным цветом выделены точки разбиения списка, а зеленым — точки останова рекурсии и начала «подъема» — слияния.

	0	1	2	3	4	5	6						
	35	22	41	4	10	80	11						
			1 m=7/2=3										
	0	1	2		0	1	2	3					
	35	22	41		4	10	80	11					
	2 m=3/2=1					6 m=4/2=2							
		0	1			0	1			0	1		
	35		22	41		4	10			80	11		
			3 m=2/2=1			7 m=2/2=1				9 m=2/2=1			
		22			41			10		80		11	
						4							
						8	4	10					
			4	22	41					10	11	80	
	5	22	35	41				11	4	10	11	80	
		12	4	10	11	22	35	41	80				

Рисунок 13

Результаты промежуточных и финального слияний выделены желтым цветом.

Реализуем нашу функцию сортировки слиянием.

```
def MergeSort(myList):
    if len(myList) > 1:

        # Finding the middle of the list
        m = len(myList)//2
        print("m: {}".format(m))

        # Splitting a list into left and right parts
        leftPart = myList[:m]
        rightPart = myList[m:]
        print("left: {}".format(leftPart))
        print("right: {}".format(rightPart))

        # Sorting the left part
        MergeSort(leftPart)
```

```

# Sorting the right part
MergeSort(rightPart)

#merge steps
i = j = k = 0

# Copy data to sorted list from leftPart-list
and rightPart-list
while i < len(leftPart) and j < len(rightPart):
    if leftPart[i] < rightPart[j]:
        myList[k] = leftPart[i]
        i += 1
    else:
        myList[k] = rightPart[j]
        j += 1
    k += 1

# Checking if any element was left in
leftPart-list
while i < len(leftPart):
    myList[k] = leftPart[i]
    i += 1
    k += 1

# Checking if any element was left in
rightPart-list
while j < len(rightPart):
    myList[k] = rightPart[j]
    j += 1
    k += 1

print("temp merge: {}".format(myList))

numbers = [35,22,41,4,10,80,11]
print("Original list: {}".format(numbers))

MergeSort(numbers)
print("Sorted list: {}".format(numbers))

```

Результаты тестирования (рис. 14):

```
Original list: [35, 22, 41, 4, 10, 80, 11]
m: 3
left: [35, 22, 41]
right: [4, 10, 80, 11]
m: 1
left: [35]
right: [22, 41]
m: 1
left: [22]
right: [41]
temp merge: [22, 41]
temp merge: [22, 35, 41]
m: 2
left: [4, 10]
right: [80, 11]
m: 1
left: [4]
right: [10]
temp merge: [4, 10]
m: 1
left: [80]
right: [11]
temp merge: [11, 80]
temp merge: [4, 10, 11, 80]
temp merge: [4, 10, 11, 22, 35, 41, 80]
Sorted list: [4, 10, 11, 22, 35, 41, 80]
```

Рисунок 14

1.4. Быстрая сортировка (Quick Sort)

Как и рассмотренная ранее сортировка слиянием, быстрая сортировка относится к категории алгоритмов, который используют стратегию «разделяй и властвуй».

Quick Sort — один из самых популярных методов сортировки. Большинство готовых библиотек и методов сортировки в разных языках программирования используют этот алгоритм как основу.

Основная идея состоит в разделении набора данных на две части. Элемент, который находится в точке разделения, называется опорным (**pivot**). Далее происходит перемещение элементов набора (например, последовательность нужно упорядочить по возрастанию): элементы, меньшие, чем опорный, перемещаются влево от него, а большие элементы — в правую часть набора.

Эта процедура повторяется рекурсивно отдельно для левой и правой части, пока весь набор не будет отсортирован, т. е. алгоритм дойдет до такой ситуации, что в каждой части останется один элемент. Помним, что пустой список и список, состоящий из одного элемента, является отсортированным.

Существует много разных версий, как выбирать опорную точку: можно выбрать первый элемент в качестве опорного, последний элемент, случайный элемент набора или медиану (элемент, расположенный посередине набора).

Наиболее удачным (и потому рекомендуемым) является выбор медианы, как опорного элемента: тогда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов.

Рассмотрим работу алгоритма на по шагам (при сортировке по возрастанию).

1. Введем две переменные для хранения индексов первого (**firstInd**) и последнего (**lastInd**) элементов сортируемого набора (на первом шаге это весь набор **myList**) и переменную для хранения индекса (**midInd**) опорного элемента.

Значение **midInd** вычислим: $\text{midInd} = (\text{firstInd} + \text{lastInd}) / 2$ или это половина набора (**число элементов / 2**)

2. Разбиваем набор и перемещаем элементы левее или правее опорного элемента: элементы из правой части, меньшие, чем опорный, перемещаются влево от него, а большие элементы из левой части — в правую часть набора.

2.1. Перемещение происходит путем обмена двух элементов: увеличиваем значение `firstInd` на 1, пока `myList[firstInd] < myList[midInd]`, таким образом обнаруживая индекс первого кандидата для переноса в правую часть.

2.2. Таким же способом (сдвигая `lastInd` влево, уменьшая его значение на 1) обнаруживаем позицию элемента-кандидата для переноса левее опорного.

2.3. Найденные слева и справа от опорного «кандидаты» меняются местами.

Шаги 2.1-2.3 повторяются, пока `firstInd < lastInd`.

3. Проверяем, если в левой части более одного элемента, то повторяем рекурсивное упорядочивание этой части (шаги 1-2). Аналогичную проверку и вызов сортировки выполняем потом и для правой части набора.

Рассмотрим на примере сортировки по возрастанию такого списка (рис. 15):

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42

Рисунок 15

При первом вызове нашей функции-сортировки мы работаем со все массивом.

Определяем позиции `firstInd`, `lastInd` и `midInd` (рис. 16).

1	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	firstInd			midInd				lastInd

Рисунок 16

Далее начинаем передвигать **firstInd** вправо и **lastInd** влево, чтобы найти возможных кандидатов на обмен (рис. 17).

	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
на месте	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	на месте	0	1	2	3	4	5	6
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	0	1	обмен	2	3	4	5	6
	12	8	25	17	33	31	40	42
	0	1	обмен	2	3	4	5	на месте
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	0	1	2	3	4	5	6	7
	12	8	17	25	33	31	40	42

Рисунок 17

Как мы видим первый кандидат на обмен в левой части — это число **25** (2-рой элемент). А вот в правой

части все элементы оказались больше опорного (т. е. в правильной части списка). Таким образом, `lastInd` дошел до позиции опорного элемента и т. к. опорный оказался меньше кандидата слева, то произошел обмен.

Так как и в левой и с правой части количество элементов больше 1, то описанная процедура сортировки будет запущена для каждой части отдельно.

Рассмотрим пример дальнейшей обработки левой части списка (рис. 18).

2	0	1	2	3
	12	8	17	25
обмен				
	0	1	2	3
	12	8	17	25
обмен				
	0	1	2	3
	12	8	17	25
обмен				
	0	1	2	3
	12	8	17	25
обмен				
	0	1	2	3
	12	8	17	25
обмен обмен				
	0	1	2	3
	8	12	17	25

Рисунок 18

Получилась аналогичная ситуация: кандидат с левой части обменялся с опорным элементом местами.

В левой части только один элемент. Поэтому данный фрагмент уже готов — отсортирован. А вот в правой части количество элементов больше одного.

Выполняем процедуру сортировки отдельно для правой части (рис. 19).

3	2	3	
	17	25	
	firstInd		
		на месте	

Рисунок 19

Как мы видим, **firstInd** и **midInd** совпали, и кандидатов для перестановок в правой части также нет. При этом количество элементов правой и левой части равно 1, т. е. на данном этапе получаем готовый отсортированный фрагмент. Так как при использовании алгоритма быстрой сортировки не создаются никакие копии списка или его частей, то мы уже получили на текущем этапе половину отсортированного списка (рис. 20):

0	1	2	3
8	12	17	25

Рисунок 20

После выполнении аналогичных шагов для правой части получим полностью отсортированный набор (рис. 21):

0	1	2	3	4	5	6	7
8	12	17	25	31	33	40	42

Рисунок 21

Реализуем нашу функцию быстрой сортировки.

```
def QuickSort(myList, firstInd, lastInd):
    global C
    global E
```

```

i = firstInd
j = lastInd

# pivot element in the middle
pivotElem = myList[firstInd + (lastInd - firstInd) // 2]
while i <= j:
    while myList[i] < pivotElem:
        i += 1
        C+=1
    while myList[j] > pivotElem:
        j -= 1
        C+=1
    C+=1
    if i <= j: # swap
        E+=1
        myList[i], myList[j] = myList[j], myList[i]
        i += 1
        j -= 1

if firstInd < j: # sort left part
    QuickSort(myList, firstInd, j)
if i < lastInd: # sort right part
    QuickSort(myList, i, lastInd)

print("C={}, E={}".format(C,E))

return myList

```

Результаты тестирования (рис. 22, 23):

Original list: [12, 8, 25, 17, 33, 31, 40, 42]

Рисунок 22

Sorted list: [8, 12, 17, 25, 31, 33, 40, 42]

Рисунок 23

1.5. Сравнение алгоритмов сортировки

Каждый из рассмотренных нами алгоритмов сортировки имеет и преимущества, и недостатки. И теперь, когда мы изучили особенности наиболее распространенных алгоритмов сортировки, давайте проведем их сравнительный анализ.

Почему и когда это важно? Давайте представим, что мы перешли от небольших размеров наборов данных в количестве до 1000 элементов до наборов, содержащих 100000 или даже более миллиона элементов (вполне реальная цифра даже для обычного интернет-магазина). В этих ситуациях эффективность алгоритма сортировки (которая является лишь одной операцией среди одновременно запущенных функций программного обеспечения) может серьезно повлиять на работу всего приложения.

Часто оценку сложности алгоритмов проводят по времени выполнения или по используемой в процессе сортировки памяти. Однако время выполнения (скорость работы алгоритма) не является хорошей мерой оценки алгоритма, поскольку оно зависит от параметров конкретного компьютера, типов данных, используемых алгоритмом, особенностей языка программирования и т. д.

Более надежным показателем оптимальности сортировки является вычислительная сложность, т. е. количество элементарных операций (простое присваивание, индексация элемента массива, арифметические операции, операции сравнения, логические операции), которые должен выполнить алгоритм сортировки (ведь каждая такая операция занимает определенное время).

Но и этот показатель, основанный на количестве выполненных операторов, не является идеальной мерой оптимальности, поскольку количество операторов в реализации алгоритма зависит как от используемого языка программирования, так и от стиля разработчика.

Наилучшим решением будет представить время работы алгоритма в виде функции $f(n)$, которая зависит только от размера входных данных (n элементов в наборе). Такой вид сравнения алгоритмов не зависит ни от машинного времени, ни от используемых технологий и стиля программирования.

Для обозначения такой временной сложности используется заглавная буква **O-Big-O Notation**.

Можно сказать, что нотация **Big-O** — это способ отслеживания того, насколько быстро растет время выполнения по отношению к размеру входных данных.

Нотация **Big-O** может выражать лучшее, худшее и среднее время работы алгоритма. Большинство обсуждений **Big-O** для разных алгоритмов сосредоточено на «верхней границе» времени выполнения алгоритма, которое часто называют наихудшим случаем. Об особенностях этих трех случаев тестирования алгоритмов поговорим немного позже.

А сейчас давайте рассмотрим возможные значения оценки сложности в нотации **Big-O** на примерах.

O(1) — время работы алгоритма — константа (т. е. постоянно) независимо от размера входных данных.

Предположим, что у нас есть список оценок студентов. Если нам нужно узнать оценку определенного студента (мы четко знаем его номер в списке группы, т. е. индекс

элемента известен), то мы получим его оценку из списка по индексу за одно и тоже время как для списка из 10 студентов, так и для списка из 1000. Таким образом, получаем $O(1)$.

$O(n)$ — линейная сложность: время выполнения алгоритма будет увеличиваться с той же скоростью, что и количество входных данных.

Например, в нашей группе 50 студентов и у каждого нужно спросить, желает ли он записаться на дополнительное занятие по программированию (каждый отвечает только «да» или «нет»). Допустим, что время ответа одного студента составляет одну минуту. Тогда на опрос группы из 50 студентов уйдет 50 минут. А зная, что нам предстоит опросить 200 студентов, мы сразу же выделим в своем графике 200 минут.

В программировании одной из наиболее распространенных задач с линейной сложностью является обход массива. Например, вывод всех элементов массива или изменение значения каждого из элементов. Таким образом, сложность фрагмента кода с циклом, который выполняется n раз, — $O(n)$.

Алгоритм линейного поиска в наборе, при котором мы проверяем каждый элемент на равенство ключу, также имеет линейную сложность.

$O(n^2)$ — квадратичная сложность означает, что вычисление выполняется за квадратичное время, т. е. равное квадрату размера входных данных.

Вернемся к нашему примеру опроса 10 студентов. Мы спрашиваем у первого студента, записывается ли он на дополнительное занятие. После этого спрашиваем его же

о мнении каждого студента из всех 9 оставшихся: как он считает, запишется ли второй студент, третий и т. д. Этот процесс опроса о себе и о других повторим с каждым студентом из группы. Общее число раз задания вопроса будет 10×10 , т. е. 10^2 . Так как код, реализующий процесс опроса таким способом будет содержать два цикла: один цикл (опрос студента обо всех) будет вложенным в другой (повторить такой опрос о себе и обо всех для каждого студента группы)

В программировании наличие двух вложенных циклов часто является признаком того, что этот фрагмент кода имеет сложность $O(n^2)$.

$O(\log n)$ — логарифмическая сложность означает, что время выполнения растет пропорционально логарифму размера входных данных, т. е. время выполнения очень мало увеличивается при увеличении количества входных данных.

Примером является бинарный поиск, где на каждом шаге мы сокращаем пространство поиска в два раза. Мы сравниваем центральный элемент набора с ключом поиска, если элемент больше ключа, то исключаем из поиска все элементы, правее центрального, иначе — все элементы левее центрального и т. д., пока не найдем ключ или левая и правая границы не сомкнутся. Теперь мы можем определить оценку сложности в нотации Big-O для каждого из изученных алгоритмов сортировки.

Также дополнительно давайте проведем сравнение по количеству элементарных операций, которые должен выполнить каждый алгоритм. В качестве таких операций (число которых мы посчитаем при выполнении каждого

алгоритма) будет использовать операцию сравнения (С) и операцию обмена (Е).

Однако, когда мы говорим о вычислительной сложности, то помним о возможных трех ситуациях:

- наихудший случай (когда набор элементов отсортирован в противоположном порядке) — тогда каждый элемент находится не на своем месте и его нужно переместить, т. е. выполнить максимальное количество элементарных операций;
- средний случай (когда исходный порядок сортируемых входных данных произволен);
- наилучший случай (набор входных данных обеспечивает минимально возможное число элементарных операций, например, список уже отсортирован).

Таким образом, тестирование проведем на трех наборах данных (по 20 элементов в каждом):

- набор со случайно сгенерированными числовыми значениями (средний случай);
- набор, в котором числовые значения отсортированы в порядке обратном порядку сортировки (наихудший случай);
- набор, в котором половина значений уже отсортирована в правильном порядке (наилучший случай).

Внесем расчет количества сравнений и перестановок элементов в наш код алгоритмов сортировки (согласно его особенностей).

Также нам нужно создать функции для генерации набора среднего случая и наилучшего.


```

from random import randint

def printList(myList):
    for index, elem in enumerate(myList):
        print("element {}: {}".format(index+1, elem))

def generateNum(n):
    nList=[]
    for i in range(n):
        nList.append(randint(11,100))
    return nList

```

Для наихудшего случая будем брать данные после сортировки в режиме «среднего» случая, развернув их наоборот с помощью функции `sorted()`.

Начнем с сортировки пузырьком. Мы видим в коде два вложенных цикла, с помощью которых алгоритм два раза проходит по всему списку. Таким образом имеем здесь сложность $O(n^2)$.

```

def myBubbleSort(myList):
    global C
    global E
    print("C={}, E={}".format(C,E))

    for i in range(len(myList)-1):
        sortedFlag=True
        for j in range(len(myList)-i-1):
            C+=1
            if myList[j]<myList[j+1]:
                E+=1
                temp=myList[j]
                myList[j]=myList[j+1]
                myList[j+1]=temp
                sortedFlag=False

```

```

        if sortedFlag:
            break
    print("C={}, E={}".format(C,E))

```

```

C=0
E=0
numbers1=generateNum(20)

print("Original list1:")
printList(numbers1)
myBubbleSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1)
print("Original list2:")
printList(numbers2)
myBubbleSort(numbers2)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3 = generateNum(10) + [10,9,8,7,6,5,4,3,2,1]

print("Original list3:")
printList(numbers3)
myBubbleSort(numbers3)
print("Sorted list3:")
printList(numbers3)

```

Результаты тестирования алгоритма сортировки «пузырьком» (таб. 1).

Таблица 1

Исходные данные	Отсортированные
Средний случай	
original list1: element 1: 31 element 2: 82 element 3: 100 element 4: 89 element 5: 62 element 6: 56 element 7: 69 element 8: 94 element 9: 59 element 10: 98 element 11: 29 element 12: 47 element 13: 38 element 14: 34 element 15: 51 element 16: 91 element 17: 71 element 18: 43 element 19: 33 element 20: 77	C=175, E=78 Sorted list1: element 1: 100 element 2: 98 element 3: 94 element 4: 91 element 5: 89 element 6: 82 element 7: 77 element 8: 71 element 9: 69 element 10: 62 element 11: 59 element 12: 56 element 13: 51 element 14: 47 element 15: 43 element 16: 38 element 17: 34 element 18: 33 element 19: 31 element 20: 29
Наихудший случай	
Original list2: element 1: 29 element 2: 31 element 3: 33 element 4: 34 element 5: 38 element 6: 43	C=190, E=190 Sorted list2: element 1: 100 element 2: 98 element 3: 94 element 4: 91 element 5: 89 element 6: 82

Исходные данные	Отсортированные
element 7: 47 element 8: 51 element 9: 56 element 10: 59 element 11: 62 element 12: 69 element 13: 71 element 14: 77 element 15: 82 element 16: 89 element 17: 91 element 18: 94 element 19: 98 element 20: 100	element 7: 77 element 8: 71 element 9: 69 element 10: 62 element 11: 59 element 12: 56 element 13: 51 element 14: 47 element 15: 43 element 16: 38 element 17: 34 element 18: 33 element 19: 31 element 20: 29
Наилучший случай	
Original list3: element 1: 70 element 2: 44 element 3: 28 element 4: 65 element 5: 22 element 6: 16 element 7: 29 element 8: 65 element 9: 42 element 10: 46 element 11: 10 element 12: 9 element 13: 8 element 14: 7 element 15: 6 element 16: 5 element 17: 4 element 18: 3 element 19: 2 element 20: 1	C=112, E=20 Sorted list3: element 1: 70 element 2: 65 element 3: 65 element 4: 46 element 5: 44 element 6: 42 element 7: 29 element 8: 28 element 9: 22 element 10: 16 element 11: 10 element 12: 9 element 13: 8 element 14: 7 element 15: 6 element 16: 5 element 17: 4 element 18: 3 element 19: 2 element 20: 1

1.5.1. Сортировка Шелла

Использует алгоритм сортировки вставкой, который также в худшем случае (данные отсортированы в обратном порядке) имеет $O(n^2)$.

Однако, в лучшем случае (когда массив изначально отсортирован правильно) этот способ обеспечит $O(n)$.

```
def InsertionSort(myList, startInd, gapValue):

    global C
    global E

    for i in range(startInd+gapValue, len(myList),
                    gapValue):
        currentElem = myList[i]
        currentInd = i

        C+=1
        while currentInd >= gapValue and
              myList[currentInd-gapValue] >
                currentElem:
            myList[currentInd] = myList[currentInd-
                                         gapValue]
            currentInd = currentInd-gapValue
            E+=1

        myList[currentInd]=currentElem
        E+=1

def ShellSort(myList):
    global C
    global E
    print("C={}, E={}".format(C,E))

    subListN = len(myList)//2
    step=1
```

```

while subListN > 0:
    for startInd in range(subListN):
        InsertionSort(myList, startInd, subListN)

    #print("Interval={}. After step {}: {}".format(subListN, step, myList))
    subListN = subListN // 2
print("C={}, E={}".format(C, E))

```

```

C=0
E=0
numbers1=generateNum(20)

print("Original list1:")
printList(numbers1)
ShellSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1, reverse=True)
print("Original list2:")
printList(numbers2)
ShellSort(numbers2)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3 = [1,2,3,4,5,6,7,8,9,10] + generateNum(10)

print("Original list3:")
printList(numbers3)

```

```
ShellSort(numbers3)
print("Sorted list3:")
printList(numbers3)
```

Результаты тестирования алгоритма сортировки Шелла (таб. 2).

Таблица 2

Исходные данные	Отсортированные
Средний случай	
Original list1: element 1: 11 element 2: 45 element 3: 70 element 4: 35 element 5: 82 element 6: 12 element 7: 22 element 8: 50 element 9: 32 element 10: 67 element 11: 74 element 12: 33 element 13: 76 element 14: 70 element 15: 63 element 16: 21 element 17: 78 element 18: 47 element 19: 39 element 20: 73	C=62, E=95 Sorted list1: element 1: 11 element 2: 12 element 3: 21 element 4: 22 element 5: 32 element 6: 33 element 7: 35 element 8: 39 element 9: 45 element 10: 47 element 11: 50 element 12: 63 element 13: 67 element 14: 70 element 15: 70 element 16: 73 element 17: 74 element 18: 76 element 19: 78 element 20: 82
Наихудший случай	
Original list2: element 1: 82	C=62, E=98 Sorted list2: element 1: 11

Исходные данные	Отсортированные
element 2: 78	element 2: 12
element 3: 76	element 3: 21
element 4: 74	element 4: 22
element 5: 73	element 5: 32
element 6: 70	element 6: 33
element 7: 70	element 7: 35
element 8: 67	element 8: 39
element 9: 63	element 9: 45
element 10: 50	element 10: 47
element 11: 47	element 11: 50
element 12: 45	element 12: 63
element 13: 39	element 13: 67
element 14: 35	element 14: 70
element 15: 33	element 15: 70
element 16: 32	element 16: 73
element 17: 22	element 17: 74
element 18: 21	element 18: 76
element 19: 12	element 19: 78
element 20: 11	element 20: 82

Наилучший случай

Original list3:	C=62, E=76
element 1: 1	Sorted list3:
element 2: 2	element 1: 1
element 3: 3	element 2: 2
element 4: 4	element 3: 3
element 5: 5	element 4: 4
element 6: 6	element 5: 5
element 7: 7	element 6: 6
element 8: 8	element 7: 7
element 9: 9	element 8: 8
element 10: 10	element 9: 9
element 11: 81	element 10: 10
element 12: 86	element 11: 20
element 13: 87	element 12: 32
element 14: 41	element 13: 39
element 15: 57	element 14: 41
	element 15: 57

Исходные данные	Отсортированные
element 16: 20	element 16: 74
element 17: 77	element 17: 77
element 18: 39	element 18: 81
element 19: 74	element 19: 86
element 20: 32	element 20: 87

1.5.2. Сортировка слиянием

Разбивает сортируемый набор на каждом шаге на две части, таким образом, оценка сложности — $O(\log n)$.

```
def MergeSort(myList):
    global C
    global E
    if len(myList) > 1:

        # Finding the middle of the list
        m = len(myList)//2
        print("m: {}".format(m))

        # Splitting a list into left and right parts
        leftPart = myList[:m]
        rightPart = myList[m:]

        print("left: {}".format(leftPart))
        print("right: {}".format(rightPart))

        # Sorting the left part
        MergeSort(leftPart)

        # Sorting the right part
        MergeSort(rightPart)

    #merge steps
    i = j = k = 0
```

```

# Copy data to sorted list from leftPart-list
  and rightPart-list
while i < len(leftPart) and j < len(rightPart):
    C+=1
    if leftPart[i] < rightPart[j]:
        myList[k] = leftPart[i]
        i += 1
        E+=1
    else:
        myList[k] = rightPart[j]
        j += 1
        E+=1

    k += 1

# Checking if any element was left in
  leftPart-list
while i < len(leftPart):
    myList[k] = leftPart[i]
    i += 1
    k += 1
    E+=1

# Checking if any element was left in
  rightPart-list
while j < len(rightPart):
    myList[k] = rightPart[j]
    j += 1
    k += 1
    E+=1

#print("temp merge: {}".format(myList))
print("C={}, E={}".format(C,E))

```

```

C=0
E=0
numbers1=generateNum(20)

```

```

print("Original list1:")
printList(numbers1)
MergeSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1,reverse=True)
print("Original list2:")
printList(numbers2)
MergeSort(numbers2)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3 = [1,2,3,4,5,6,7,8,9,10] + generateNum(10)
print("Original list3:")
printList(numbers3)
MergeSort(numbers3)
print("Sorted list3:")
printList(numbers3)

```

Результаты тестирования алгоритма сортировки слиянием (таб. 3).

Таблица 3

Исходные данные	Отсортированные
Средний случай	
Original list1: element 1: 57 element 2: 25 element 3: 65	C=61, E=88 Sorted list1: element 1: 14 element 2: 15 element 3: 19

Исходные данные	Отсортированные
element 4: 57	element 4: 23
element 5: 63	element 5: 35
element 6: 79	element 6: 38
element 7: 27	element 7: 40
element 8: 14	element 8: 42
element 9: 83	element 9: 47
element 10: 71	element 10: 56
element 11: 45	element 11: 61
element 12: 30	element 12: 67
element 13: 40	element 13: 72
element 14: 55	element 14: 74
element 15: 59	element 15: 76
element 16: 94	element 16: 81
element 17: 72	element 17: 87
element 18: 78	element 18: 89
element 19: 15	element 19: 93
element 20: 38	element 20: 96

Наихудший случай

	C=48, E=88
Original list2:	Sorted list2:
element 1: 96	element 1: 14
element 2: 93	element 2: 15
element 3: 89	element 3: 19
element 4: 87	element 4: 23
element 5: 81	element 5: 35
element 6: 76	element 6: 38
element 7: 74	element 7: 40
element 8: 72	element 8: 42
element 9: 67	element 9: 47
element 10: 61	element 10: 56
element 11: 56	element 11: 61
element 12: 47	element 12: 67
element 13: 42	element 13: 72
element 14: 40	element 14: 74
element 15: 38	element 15: 76
element 16: 35	element 16: 81
element 17: 23	element 17: 87
element 18: 19	element 18: 89

Исходные данные	Отсортированные
element 19: 15 element 20: 14	element 19: 93 element 20: 96

Наилучший случай

Original list3:	C=45, E=88
element 1: 1	Sorted list3:
element 2: 2	element 1: 1
element 3: 3	element 2: 2
element 4: 4	element 3: 3
element 5: 5	element 4: 4
element 6: 6	element 5: 5
element 7: 7	element 6: 6
element 8: 8	element 7: 7
element 9: 9	element 8: 8
element 10: 10	element 9: 9
element 11: 78	element 10: 10
element 12: 67	element 11: 17
element 13: 41	element 12: 41
element 14: 51	element 13: 49
element 15: 17	element 14: 51
element 16: 49	element 15: 61
element 17: 61	element 16: 67
element 18: 68	element 17: 68
element 19: 96	element 18: 78
element 20: 98	element 19: 96
	element 20: 98

1.5.3. Быстрая сортировка

Сложность этого алгоритма в среднем и лучшем случаях $O(n \cdot \log n)$, а вот при худшем случае мы получим $O(n^2)$.

```
def QuickSort(myList, firstInd, lastInd):

    global C
    global E
```

```

i = firstInd
j = lastInd
pivotElem = myList[firstInd + (lastInd - firstInd) //2]
# pivot element in the middle
while i <= j:
    while myList[i] < pivotElem:
        i += 1
        C+=1
    while myList[j] > pivotElem:
        j -= 1
        C+=1
    C+=1
    if i <= j: # swap
        E+=1
        myList[i], myList[j] = myList[j], myList[i]
        i += 1
        j -= 1
if firstInd < j: # sort left part
    QuickSort(myList, firstInd, j)
if i < lastInd: # sort right part
    QuickSort(myList, i, lastInd)

print("C={}, E={}".format(C,E))
return myList

```

```

C=0
E=0
numbers1=generateNum(20)
print("Original list1:")
printList(numbers1)
QuickSort(numbers1,0,len(numbers1)-1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

```

```

numbers2=sorted(numbers1,reverse=True)
print("Original list2:")
printList(numbers2)
QuickSort(numbers2,0,len(numbers1)-1)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3 = [1,2,3,4,5,6,7,8,9,10] + generateNum(10)

print("Original list3:")
printList(numbers3)
QuickSort(numbers3, 0, len(numbers1)-1)
print("Sorted list3:")
printList(numbers3)

```

Результаты тестирования алгоритма сортировки слиянием (таб. 4).

Таблица 4

Исходные данные	Отсортированные
Средний случай	
Original list1: element 1: 20 element 2: 74 element 3: 65 element 4: 67 element 5: 31 element 6: 79 element 7: 11 element 8: 63 element 9: 28 element 10: 77	C=70, E=24 Sorted list1: element 1: 11 element 2: 17 element 3: 20 element 4: 21 element 5: 24 element 6: 28 element 7: 29 element 8: 31 element 9: 40 element 10: 46

Исходные данные	Отсортированные
element 11: 46	element 11: 63
element 12: 71	element 12: 65
element 13: 21	element 13: 66
element 14: 99	element 14: 67
element 15: 29	element 15: 71
element 16: 40	element 16: 74
element 17: 66	element 17: 77
element 18: 17	element 18: 79
element 19: 94	element 19: 94
element 20: 24	element 20: 99

Наихудший случай

Original list2:	C=60, E=22
element 1: 99	Sorted list2:
element 2: 94	element 1: 11
element 3: 79	element 2: 17
element 4: 77	element 3: 20
element 5: 74	element 4: 21
element 6: 71	element 5: 24
element 7: 67	element 6: 28
element 8: 66	element 7: 29
element 9: 65	element 8: 31
element 10: 63	element 9: 40
element 11: 46	element 10: 46
element 12: 40	element 11: 63
element 13: 31	element 12: 65
element 14: 29	element 13: 66
element 15: 28	element 14: 67
element 16: 24	element 15: 71
element 17: 21	element 16: 74
element 18: 20	element 17: 77
element 19: 17	element 18: 79
element 20: 11	element 19: 94
	element 20: 99

Наилучший случай

Original list3:	C=70, E=16
element 1: 1	Sorted list3:
	element 1: 1

Исходные данные	Отсортированные
element 2: 2	element 2: 2
element 3: 3	element 3: 3
element 4: 4	element 4: 4
element 5: 5	element 5: 5
element 6: 6	element 6: 6
element 7: 7	element 7: 7
element 8: 8	element 8: 8
element 9: 9	element 9: 9
element 10: 10	element 10: 10
element 11: 69	element 11: 20
element 12: 88	element 12: 25
element 13: 25	element 13: 33
element 14: 42	element 14: 42
element 15: 33	element 15: 69
element 16: 72	element 16: 72
element 17: 94	element 17: 74
element 18: 85	element 18: 85
element 19: 74	element 19: 88
element 20: 20	element 20: 94

Таблица 5. Итоговые результаты сравнения алгоритмов

Алгоритм	Средний случай		Наихудший случай		Наилучший случай	
	С	Е	С	Е	С	Е
Пузырьком	175	78	190	190	112	20
Шелла	62	95	62	98	62	76
Слиянием	61	88	48	88	45	88
Быстрая сортировка	70	24	60	22	70	16

2. Поиск

2.1. Лине́йный поиск

Задача поиска данных является одной из самой распространенной и практически в каждом приложении (независимо от предметной области его использования и средств реализации) есть функция поиска. Давайте вначале определимся, а в чем, собственно, состоит задача поиска.

Поиск — процесс обнаружения в некотором наборе данных таких элементов (объектов), характеристики которых (одна или несколько) соответствуют критериям поиска.

Критерий поиска — это условие (ограничение), накладываемое на значения свойств (характеристик) данных.

Простейший пример: у нас есть список логинов пользователя и нужно узнать, есть ли в этом списке логин «**admin**». Значение критерия поиска часто называют *ключом поиска* (т. е. в нашем примере ключ — это строка «**admin**»). В этом случае критерий поиска: значение элемента равно «**admin**» (т. е. проверяемой характеристикой элемента является его значение).

Другой пример: найти в списке логинов пользователей такие логины, которые содержат меньше 7 символов. Здесь критерием поиска будет количество символов в логине (длина логина).

Сегодня существует множество различных алгоритмов поиска, идеи которых часто зависят от особенностей набора данных, в которых мы проводим поиск.

Наборы данных в самом общем случае можно разделить на неупорядоченные и упорядоченные (отсортированные).

Так как о неупорядоченном наборе данных нам заранее ничего неизвестно, то самым распространённым алгоритмом поиска в таких коллекциях является прямой перебор или линейный алгоритм поиска.

Линейный поиск — это полный последовательный (один за одним) перебор всех элементов набора данных (например, некоторого списка `myList`), при котором каждый элемент набора проверяется на соответствие критерию поиска (в простейших случаях сравнивается с ключом поиска `key`).

Начинается линейный поиск с крайнего, чаще всего, левого элемента набора.

Поиск окончен, если:

- нужный элемент найден (такой, что `myList[i] == key`);
- весь список просмотрен (поиск дошел до конца списка) и нужный элемент не найден.

Рассмотрим на примере.

0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
40==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
62==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
90==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
10==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
17==17? True							
Stop searching							

Рисунок 24

В нашем примере на пятом шаге сравнения текущего элемента (выделен желтым цветом фона) с ключом поиска (значение 17) нужный элемент (с индексом 4) был найден. Поиск был остановлен на этом этапе. Результат поиска — успешен.

Рассмотрим еще один пример. Пусть ключ поиска — это число 29. Перебрав последовательно все элементы нашего списка (выполнив количество операций сравнений равное числу элементов в списке) мы не нашли число 29. В этом случае поиск был остановлен по второму из двух условий — пройден весь список. Результат поиска — не успешен.

Реализуем нашу функцию линейного поиска, которая будет принимать два аргумента: список (в котором будет проходить поиск) и ключ (значение элемента, позицию которого нужно найти).

Предположим, что нам нужно найти первое вхождение ключа в наш список (т. е. в случае совпадения элемента с ключом функция вернет индекс этого элемента и поиск будет остановлен). Если же поиск неуспешен (ключа нет в списке), то функция вернет значение -1.

```
def LinearSearch(myList, keyItem):  
    for i in range(len(myList)):  
        if myList[i] == keyItem:  
            return i  
    return -1  
  
numbers = [40, 62, 90, 10, 17, 11, 80, 25]  
print("Original list: {}".format(numbers))  
  
key1=17
```

```
print("Key element {} is in List in {} position".
      format(key1,LinearSearch(numbers, key1)))

key2=29
print("Key element {} is in List in {} position".
      format(key2,LinearSearch(numbers, key2)))
```

Результаты успешного и неуспешного поиска:

```
Original list: [40, 62, 90, 10, 17, 11, 80, 25]
Key element 17 is in List in 4 position
Key element 29 is in List in -1 position
```

Рисунок 25

Теперь рассмотрим ситуацию, когда в нашем списке находятся много элементов со значениями, равными ключу. И нам нужно знать индексы их всех (все вхождения ключа в список).

Для этого создадим дополнительный список (изначально пустой) и при каждом нахождении ключа в списке будем заносить его индекс в список индексов.

```
def LinearSearchAllKeys(myList, keyItem):
    indList=[]
    for i in range(len(myList)):
        if myList[i] == keyItem:
            indList.append(i)

    return indList

numbers=[40,17,62,90,10,17,11,80,25,17]
key1=17
key2=29
```

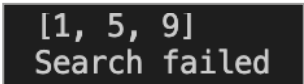
```

indKey1=LinearSearchAllKeys(numbers, key1)
indKey2=LinearSearchAllKeys(numbers, key2)

for indKey in (indKey1, indKey2):
    if len(indKey) != 0:
        print(indKey)
    else:
        print("Search failed")

```

Результаты для успешного поиска (ключ 17) и для неуспешного (ключ 29).



[1, 5, 9]
Search failed

Рисунок 26

У данного алгоритма есть только одно преимущество — простота идеи и реализации. Однако при больших наборах данных — это очень медленный алгоритм. И в худшем случае (когда ключ находится в конце списка) нам нужно перебрать весь список. В случае неотсортированного набора данных — это единственно возможный способ реализации поиска.

2.2. Бинарный поиск

Этот алгоритм поиска может использоваться только в отсортированных наборах данных. Однако несмотря на такие требования к данным (даже в случае их изначальной неупорядоченности, которая потребует выполнения сортировки перед поиском) бинарный поиск считается лучшим и наиболее быстрым алгоритмом.

Алгоритм бинарного поиска использует подход «разделяй и властвуй». Основной шаг при бинарном поиске — это извлечение элемента из середины набора и его проверка на равенство ключу. Далее, в зависимости от результатов сравнения, мы убираем ту или иную половину набора из дальнейшего рассмотрения. Таким образом, пространство поиска сокращается в два раза на каждом шаге алгоритма.

При том использовать бинарный поиск для обнаружения максимального или минимального значения в наборе нет смысла, т. к. набор упорядочен и данные элементы находятся в начале и в конце набора соответственно.

Рассмотрим шаги бинарного поиска детальнее. Пусть наш исходный набор данных отсортирован по возрастанию.

1. Выполняем сравнение среднего элемента набора с ключом.
2. Если ключ и средний элемент оказываются равны, то поиск завершается успешно.
3. Если средний элемент больше ключа, то, значит, нужно продолжать поиск только в левой части набора (до среднего элемента), иначе — поиск будет проходить в правой части, т. е. определяем новые границы пространства поиска.
4. Шаги 1-3 повторяем, пока в пространстве поиска есть элементы.

Когда мы говорим об исключении правой или левой части из пространства поиска, то фактически мы «передвигаем» левую или правую границу пространства вправо или влево соответственно.

Изначально левая граница (L) находится на 0-вом элементе списка, а правая граница (R) — на последнем.

Рассмотрим на примере.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L			$m=(L+R)//2$			key	R
			12==21? 12<21				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
				L		key	R

Рисунок 27

После сравнения среднего (3-го элемента, значение 12) с ключом (значение 21) пространство поиска изменилось передвиганием левой границы (L) в позицию $m+1$ ($3+1=4$), т. е. на следующем шаге мы продолжим поиск в наборе, начиная с индекса 4 по 7 включительно.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
				L		key	R
					$m=(L+R)//2$		
					18==21? 18<21		
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
						key	R
						L	

Рисунок 28

Получаем аналогичную ситуацию (средний элемент меньше ключа) и опять смещаем левую границу. Продолжаем поиск в пространстве с 6-го по 7-мой элемент набора включительно.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
						key	R
						L	
						$m=(L+R)//2$	
						21==21? True	

Рисунок 29

При этой проверке средний (6-той) элемент совпал с ключом. Поиск останавливается. Результат поиска успешен.

Теперь рассмотрим пример того, как будет работать алгоритм, когда ключа нет в наборе (неуспешный поиск).

Пусть набор остается тот же, а ключ равен 7.

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L			$m=(L+R)//2$				R
			12==7? 12>7				

Рисунок 30

Центральный элемент (12) больше ключа, поэтому передвигаем правую границу ($R=m-1=3-1=2$).

На следующем шаге пространство поиска — это часть списка от 0-го до 2-рого элемента включительно.

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L		R					

Рисунок 31

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L		R					
	5==7? 5<7						

Рисунок 32

После этого шага (центральный элемент со значением 5 меньше ключа) мы передвигаем уже левую границу:

		L	key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
		R					

Рисунок 33

Получаем ситуацию, когда правая и левая границы совпали. Проверяем элемент в этой точке — он не равен ключу. Поиск закончен по причине отсутствия пространства поиска. Результат поиска неудачен — ключ в наборе не был найден.

Реализуем нашу функцию бинарного поиска.

```
def BinarySearch(myList, keyItem):
    L=0
    R=len(myList)-1
    keyFound=False

    while (L<=R) and (keyFound==False):
        m=(L+R)//2

        if myList[m]==keyItem:
            keyFound=True
```

```

        elif myList[m]>keyItem:
            R=m-1
        else:
            L=m+1

    if keyFound:
        return m
    else:
        return -1

numbers=[2,5,9,12,17,18,21,32]
key1=17
key2=29
indKey1=BinarySearch(numbers, key1)
indKey2=BinarySearch(numbers, key2)

for indKey in (indKey1,indKey2):
    if indKey!=-1:
        print(indKey)
    else:
        print("Item is not found")

```

Результат:

```

4
Item is not found

```

Рисунок 34



Урок 5

Сортировка, поиск

© STEP IT Academy, www.itstep.org

© Анна Егошина

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии с законодательством о свободном использовании произведения без согласия его автора (или другого лица, имеющего авторское право на данное произведение). Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования. Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника. Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством.