Материалы занятия

Курс: Разработка Web-приложений на Python, с применением Фреймворка Django
Дисциплина: Основы программирования на Python

Тема занятия №8: Сортировка, поиск, регулярные выражения

Введение

Сортировка – это расстановка элементов массива в заданном порядке. Варианты сортировки: по возрастанию, убыванию, последней цифре, сумме делителей, по алфавиту.

Алгоритмы:

- простые и понятные, но неэффективные для больших массивов
  - метод пузырька (Bubble sort)
- сложные, но эффективные
  - быстрая сортировка (QuickSort)
  - сортировка Шелла (Shell Sort)
  - сортировка слиянием (MergeSort)

Функция SORTED

Функция sorted возвращает новый отсортированный список, который получен из итерируемого объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.

Первый аспект, на который важно обратить внимание - sorted всегда возвращает список.

Если сортировать список элементов, то возвращается новый список:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']
In [2]: sorted(list_of_words)
Out[2]: ['', 'dict', 'list', 'one', 'two']
```

При сортировке кортежа также возвращается список:

```
In [3]: tuple_of_words = ('one', 'two', 'list', '', 'dict')
In [4]: sorted(tuple_of_words)
Out[4]: ['', 'dict', 'list', 'one', 'two']
```

Сортировка множества:

```
In [5]: set_of_words = {'one', 'two', 'list', '', 'dict'}
In [6]: sorted(set_of_words)
Out[6]: ['', 'dict', 'list', 'one', 'two']
```

Сортировка строки:

```
In [7]: string_to_sort = 'long string'
In [8]: sorted(string_to_sort)
Out[8]: [' ', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 's', 't']
```

Если передать sorted словарь, функция вернет отсортированный список ключей:

```
In [9]: dict_for_sort = {'id': 1, 'name': 'London', 'IT_VLAN': 320, 'User_VLAN': 1010,
'Mngmt_VLAN': 99, 'to_name': None, 'to_id': None, 'port': 'G1/0/11', }

In [10]: sorted(dict_for_sort)
Out[10]:
['IT_VLAN', 'Mngmt_VLAN', 'User_VLAN', 'id', 'name', 'port', 'to_id', 'to_name']
```

Метод LIST.SORT()

Метод списков list.sort(), который изменяет исходный список (и возвращает None во избежание путаницы). Обычно это не так удобно, как использование sorted(), но если вам не нужен исходный список, то так будет немного эффективнее:

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

REVERSE

Флаг reverse позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.

Указав флаг reverse, можно поменять порядок:

```
In [11]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [12]: sorted(list_of_words)
Out[12]: ['', 'dict', 'list', 'one', 'two']

In [13]: sorted(list_of_words, reverse=True)
Out[13]: ['two', 'one', 'list', 'dict', '']
```

KEY

С помощью параметра key можно указывать, как именно выполнять сортировку. Параметр key ожидает функцию, с помощью которой должно быть выполнено сравнение.

Например, таким образом можно отсортировать список строк по длине строки:

```
In [14]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [15]: sorted(list_of_words, key=len)
Out[15]: ['', 'one', 'two', 'list', 'dict']
```

Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
In [16]: dict_for_sort = { 'id': 1, 'name':'London', 'IT_VLAN':320, 'User_VLAN':1010,
'Mngmt_VLAN':99, 'to_name': None, 'to_id': None, 'port':'G1/0/11'}
```

```
In [17]: sorted(dict_for_sort, key=str.lower)
Out[17]:
['id',
 'IT_VLAN',
 'Mngmt_VLAN',
 'name',
 'port',
 'to_id',
 'to_name',
 'User_VLAN']
```

Параметру key можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию lambda.

С помощью параметра key можно сортировать объекты не по первому элементу, а по любому другому. Но для этого надо использовать или функцию lambda, или специальные функции из модуля operator.

Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
In [18]: from operator import itemgetter

In [19]: list_of_tuples = [('IT_VLAN', 320), ('Mngmt_VLAN', 99), ('User_VLAN', 1010),
   ('DB_VLAN', 11)]

In [20]: sorted(list_of_tuples, key=itemgetter(1))
Out[20]: [('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320), ('User_VLAN', 1010)]
```

Сортировка пузырьком

Сортировка пузырьком — это метод сортировки массивов и списков 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.

В сортировке методом пузырька количество итераций внешнего цикла определяется длинной списка минус единица, так как когда второй элемент становится на свое место, то первый уже однозначно минимальный и находится на своем месте.

Количество итераций внутреннего цикла зависит от номера итерации внешнего цикла, так как конец списка уже отсортирован, и выполнять проход по этим элементам смысла нет.

Рассмотрим практический пример работы сортировки.

Пусть имеется список [6, 12, 4, 3, 8].

За первую итерацию внешнего цикла число 12 переместится в конец. Для этого потребуется 4 сравнения во внутреннем цикле:

6 > 12? Нет
12 > 4? Да. Меняем местами
12 > 3? Да. Меняем местами
12 > 8? Да. Меняем местами
Результат: [6, 4, 3, 8, 12]

За вторую итерацию внешнего цикла число 8 переместиться на предпоследнее место. Для этого потребуется 3 сравнения:

6 > 4? Да. Меняем местами
6 > 3? Да. Меняем местами
6 > 8? Нет
Результат: [4, 3, 6, 8, 12]

На третьей итерации внешнего цикла исключаются два последних элемента. Количество итераций внутреннего цикла равно двум:

4 > 3? Да. Меняем местами
4 > 6? Нет
Результат: [3, 4, 6, 8, 12]

На четвертой итерации внешнего цикла осталось сравнить только первые два элемента, поэтому количество итераций внутреннего равно единице:

3 > 4? Нет
Результат: [3, 4, 6, 8, 12]

Программное представление вышеприведенного практического примера.

Реализация сортировки пузырьком с помощью циклов for

```
from random import randint

N = 10
a = []
for i in range(N):
   a.append(randint(1, 99))
print(a)


for i in range(N-1):
   for j in range(N-i-1):
      if a[j] > a[j+1]:
         a[j], a[j+1] = a[j+1], a[j]

print(a)
```

Пример выполнения кода:

```
[63, 80, 62, 69, 71, 37, 12, 90, 19, 67]
[12, 19, 37, 62, 63, 67, 69, 71, 80, 90]
```

С помощью циклов while

```
from random import randint

N = 10
```

```
a = []
for i in range(N):
   a.append(randint(1, 99))
print(a)

i = 0
while i < N - 1:
   j = 0
   while j < N - 1 - i:
      if a[j] > a[j+1]:
         a[j], a[j+1] = a[j+1], a[j]
      j += 1
   i += 1

print(a)
```

Сортировка шелла (SHELL SORT)

Сортировку Шелла иногда называют “сортировкой с уменьшением инкремента”. Она улучшает сортировку вставками, разбивая первоначальный список на несколько подсписков, каждый из которых сортируется по отдельности. Оригинальный способ выбора таких подсписков - ключевая идея сортировки Шелла. Вместо того, чтобы выделять подсписки из стоящих рядом элементов, сортировка Шелла использует инкремент i (приращение), чтобы создавать подсписки из значений, отстоящих на расстоянии i друг от друга.

Это можно увидеть на рисунке 8.1 Изображённый на нём список состоит из девяти элементов. Если мы используем тройку в качестве инкремента, то получим три подсписка, каждый из которых можно отсортировать вставками. После завершения этих сортировок мы получим список с рисунка 8.2. Хотя он ещё не отсортирован до конца, случилось нечто весьма интересное. Сортируя эти подсписки, мы переместили элементы ближе друг к другу относительно того, где они были раньше.

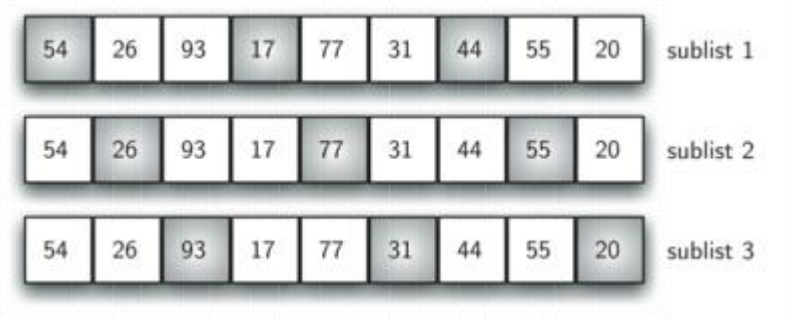


Рисунок 8.1. Сортировка Шелла с инкрементом 3

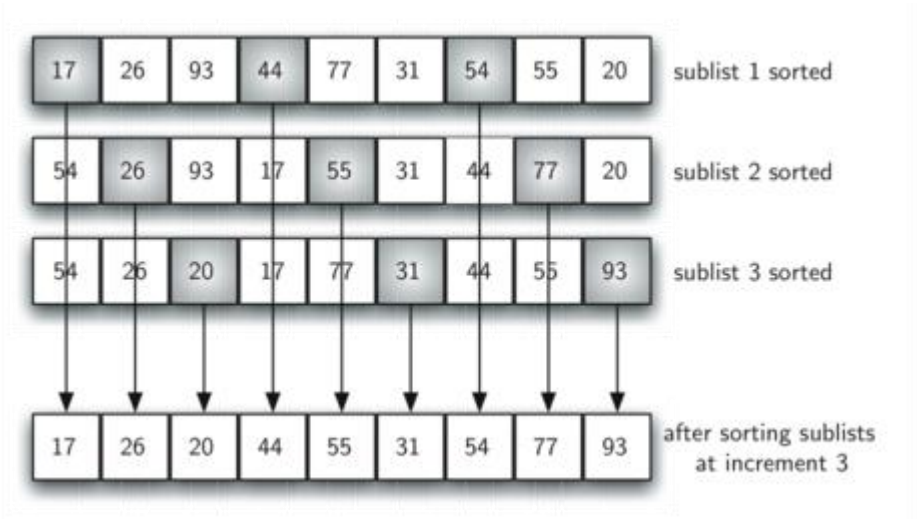


Рисунок 8.2. Сортировка Шелла после сортировки каждого из подсписков

Рисунок 8.3 демонстрирует окончательную сортировку вставками с использованием приращения 1 - другими словами, стандартную сортировку вставками. Обратите внимание, что, благодаря проведению предварительных сортировок подсписков, сейчас мы сократили общее число операций смещения, необходимых для расположения элементов списка в правильном порядке. В нашем случае требуется всего четыре сдвига, чтобы завершить процесс.

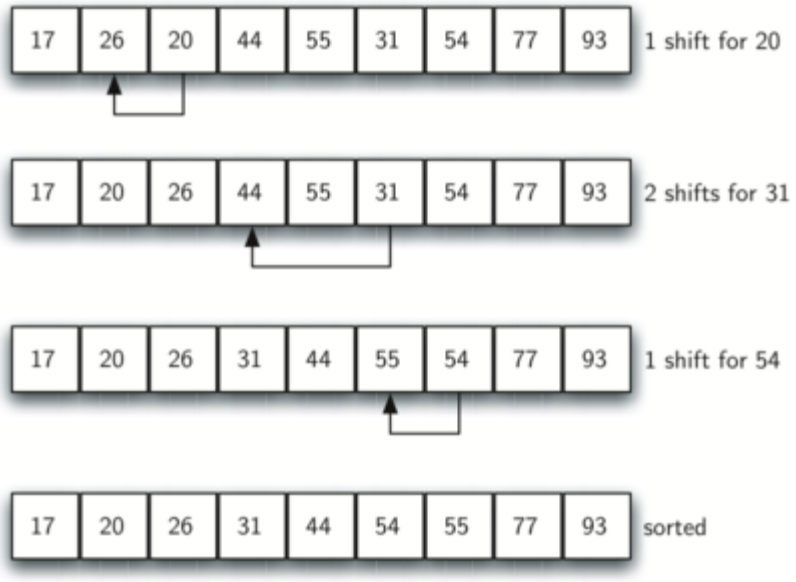


Рисунок 8.3. Сортировка Шелла: итоговая сортировка вставками с инкрементом 1

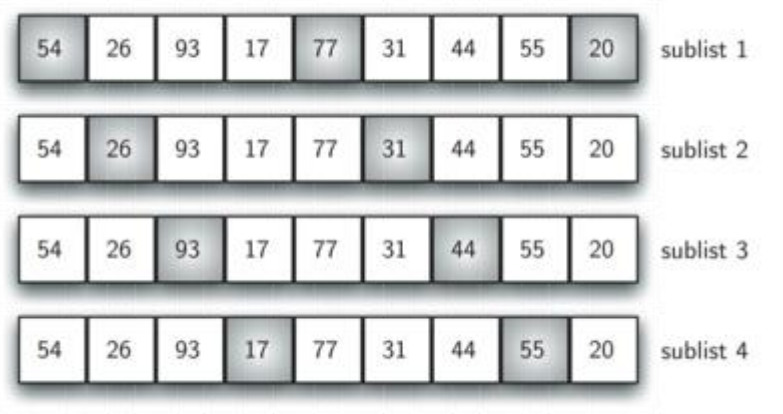


Рисунок 8.4. Начальный подсписки для сортировки Шелла

Ранее мы уже говорили, что способ, которым выбирается инкремент, - уникальное свойство сортировки Шелла. Функция, показанная в практическом примере, использует различный набор инкрементов. Мы начинаем с n2n2 подсписков. На следующем проходе сортируются n4n4 подсписков. Наконец, единственный список сортируется с помощью базовой сортировки вставками. Рисунок 8.4 показывает первые подсписки из нашего примера, использующие этот инкремент.

Следующий вызов функции shellSort демонстрирует списки, частично отсортированные после каждого приращения, и финальную сортировку с инкрементом 1.

```
def shellSort(alist):
   sublistcount = len(alist)//2
   while sublistcount > 0:

     for startposition in range(sublistcount):
      gapInsertionSort(alist,startposition,sublistcount)

     print("After increments of size",sublistcount,
                           "The list is",alist)

     sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
   for i in range(start+gap,len(alist),gap):

       currentvalue = alist[i]
       position = i

       while position>=gap and alist[position-gap]>currentvalue:
          alist[position]=alist[position-gap]
          position = position-gap

       alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

Результат выполнения

```
After increments of size 4 The list is [20, 26, 44, 17, 54, 31, 93, 55, 77]
After increments of size 2 The list is [20, 17, 44, 26, 54, 31, 77, 55, 93]
After increments of size 1 The list is [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]


** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Сортировка

Введение

Сортировка – это расстановка элементов массива в заданном порядке. Варианты сортировки: по возрастанию, убыванию, последней цифре, сумме делителей, по алфавиту.

Алгоритмы:

- сложные, но эффективные
  - быстрая сортировка (QuickSort)
  - сортировка слиянием (MergeSort)

8.1 Быстрая сортировка *(QuickSort)*

Автор английский ученый Чарльз Хоар, 1960 год. Данный метод предполагает деление массива на две части, в одной из которых находятся элементы меньше определённого значения, в другой – больше или равные.

Рассмотрим пошагово то, что мы планируем сделать, это поможет проиллюстрировать весь процесс. Пусть у нас будет следующий список.

29 | 99 (low),27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,21,44 (high)

Выберем первый элемент как опору 29|, а указатель на меньшие элементы (называемый «low») будет следующим элементом, указатель на более крупные элементы (называемый «high») станем последний элемент в списке.

29 | 99 (low),27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,21 (high),44

Мы двигаемся в сторону high то есть влево, пока не найдем значение, которое ниже нашего опорного элемента.

29 | 99 (low),27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,21 (high),44

Теперь, когда наш элемент high указывает на элемент 21, то есть на значение меньше, чем опорное значение, мы хотим найти значение в начале массива, с которым мы можем поменять его местами. Нет смысла менять местами значение, которое меньше, чем опорное значение, поэтому, если low указывает на меньший элемент, мы пытаемся найти тот, который будет больше.

Мы перемещаем переменную low вправо, пока не найдем элемент больше, чем опорное значение. К счастью, low уже имеет значение 89.

Мы меняем местами low и high:

29 | 21 (low),27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,99 (high),44

Сразу после этого мы перемещает high влево и low вправо (поскольку 21 и 89 теперь на своих местах)

Опять же, мы двигаемся high влево, пока не достигнем значения, меньшего, чем опорное значение, и мы сразу находим — 12

Теперь мы ищем значение больше, чем опорное значение, двигая low вправо, и находим такое значение 41

Этот процесс продолжается до тех пор, пока указатели low и high наконец не встретятся в одном элементе:

29 | 21,27,12,19,28 (low/high),44,78,87,66,31,76,58,88,83,97,41,99,44

Мы больше не используем это опорное значение, поэтому остается только поменять опорную точку и high, и мы закончили с этим рекурсивным шагом:

28,21,27,12,19,29,44,78,87,66,31,76,58,88,83,97,41,99,44

Как видите, мы достигли того, что все значения, меньшие 29, теперь слева от 29, а все значения больше 29 справа.

Затем алгоритм делает то же самое для коллекции 28,21,27,12,19 (левая сторона) и 44,78,87,66,31,76,58,88,83,97,41,99,44 (правая сторона). И так далее.

Рассмотрим реализацию быстрой сортировки.

```
import random
def QuickSort(A):
    if len(A) <= 1:
        return A
    else:
        q = random.choice(A)
        L = []
        M = []
        R = []
        for elem in A:
            if elem < q:
                L.append(elem)
            elif elem > q:
                R.append(elem)
            else:
                M.append(elem)
        return QuickSort(L) + M + QuickSort(R)

m=[4,8,96,-9,-6,2,-7,11]
print(QuickSort(m))
```

В данном примере в списке L собираются элементы, меньшие q, в списке R — большие q, а в списке M — равные q. Разделение на три списка, а не на два используется для того, чтобы алгоритм не зацикливался, например, в случае, когда в списке остались только равные элементы. Барьерный элемент q выбирается случайным образом из списка при помощи функции choice из модуля random.

8.2 Сортировка слиянием *(Mergesort)*

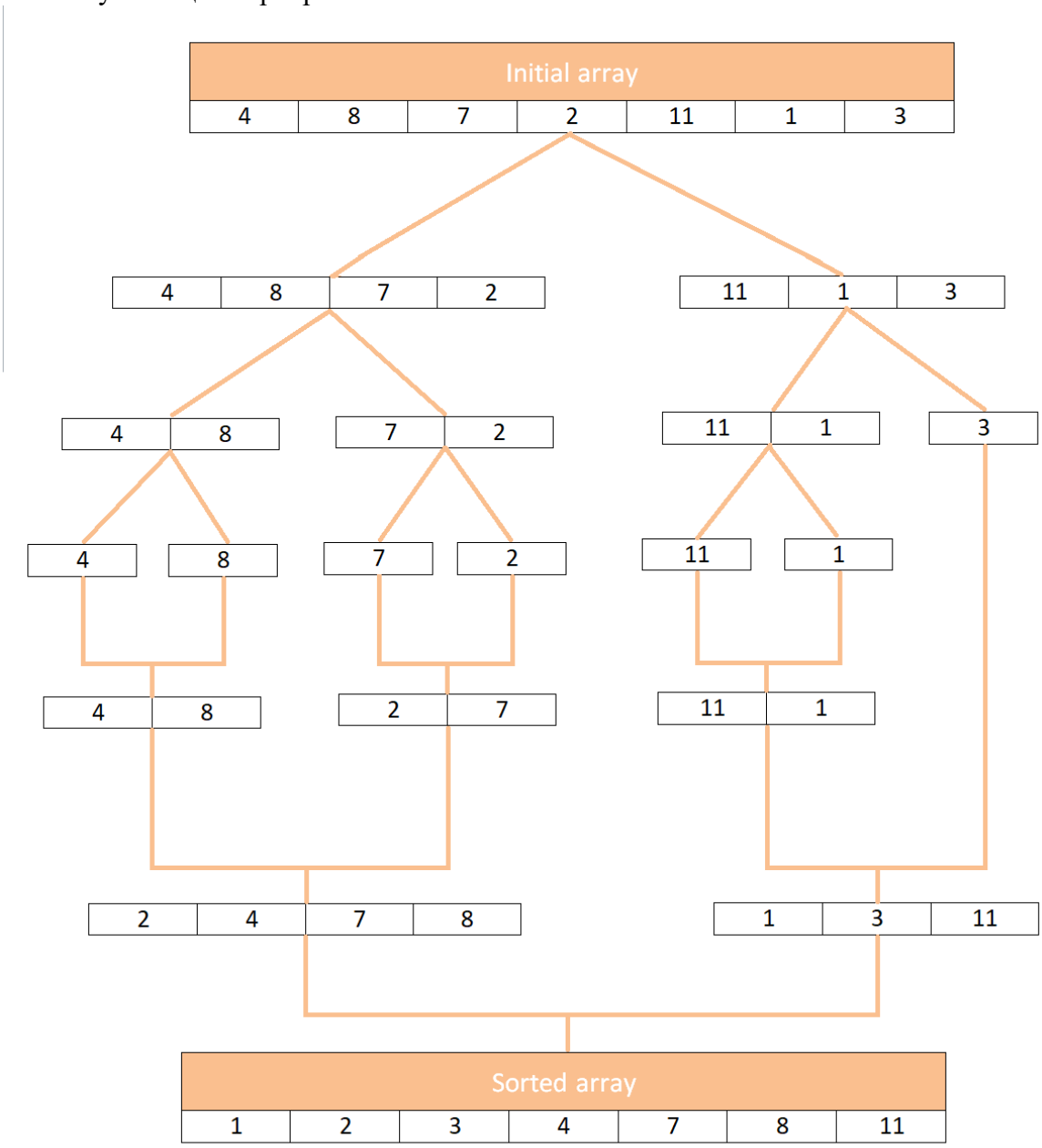
Способ работы сортировки слиянием заключается в следующем:

Исходный массив делится на две примерно равные части. Если массив имеет нечетное число элементов, то одна из этих “половинок” на один элемент больше другой.

Подмассивы снова и снова делятся на половинки, пока вы не получите массивы, каждый из которых имеет только один элемент.

Затем вы объединяете пары одноэлементных массивов в двухэлементные массивы, сортируя их в процессе. Затем эти отсортированные пары объединяются в четырехэлементные массивы и так далее, пока вы не закончите сортировку исходного массива.

Вот визуализация сортировки слиянием:



Как вы можете видеть, тот факт, что массив не может быть разделен на равные половины, не является проблемой. 3 просто “ждет”, пока начнется сортировка.

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

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

Примечание: Подход снизу вверх обеспечивает интересную оптимизацию, которую мы обсудим позже. Мы будем реализовывать подход сверху вниз, поскольку он проще и интуитивно понятен в сочетании с тем фактом, что нет реальной разницы между временной сложностью между ними без конкретной оптимизации.

Основная часть обоих этих подходов заключается в том, как мы объединяем (сливаем) два меньших массива в больший массив. Это делается довольно интуитивно, допустим, мы рассмотрим последний шаг в нашем предыдущем примере. У нас есть массивы:

А: 2 4 7 8

Б: 1 3 11

сортировка: пусто

Первое, что мы делаем, это смотрим на первый элемент обоих массивов. Мы находим тот, который меньше, в нашем случае это 1 , так что это первый элемент нашего отсортированного массива, и мы двигаемся вперед в массиве B :

А: 2 4 7 8

Б: 1 3 11

сортировка: 1

Затем мы рассмотрим следующую пару элементов 2 и 3 ; 2 меньше, поэтому мы помещаем его в наш сортированный массив и двигаемся вперед в массиве A . Конечно, мы не движемся вперед в массиве B и держим наш указатель на 3 для будущих сравнений:

А: 2 4 7 8

Б: 1 3 11

сортировка: 1 2

Используя ту же логику, мы проходим через все остальное и в конечном итоге получаем массив{1, 2, 3, 4, 7, 8, 11}.

Два особых случая, которые могут возникнуть:

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

Мы “выбегаем” из элементов в одном подмассиве. Например, у нас есть массив с {1, 2, 3} и массив с {9, 10, 11}. Ясно, что мы пройдем через все элементы в первом массиве, не двигаясь вперед ни разу во втором. Всякий раз, когда у нас заканчиваются элементы в подмассиве, мы просто добавляем элементы второго один за другим.

Имейте в виду, что мы можем сортировать так, как хотим – этот пример сортирует целые числа в порядке возрастания, но мы можем так же легко сортировать в порядке убывания или сортировать пользовательские объекты.

Реализация сортировки слияния на практическом примере:

```
def merge_sort(alist, start, end):
    '''Sorts the list from indexes start to end - 1 inclusive.'''
    if end - start > 1:
        mid = (start + end)//2
        merge_sort(alist, start, mid)
        merge_sort(alist, mid, end)
        merge_list(alist, start, mid, end)

def merge_list(alist, start, mid, end):
    left = alist[start:mid]
    right = alist[mid:end]
    k = start
    i = 0
    j = 0
    while (start + i < mid and mid + j < end):
        if (left[i] <= right[j]):
            alist[k] = left[i]
            i = i + 1
        else:
            alist[k] = right[j]
            j = j + 1
        k = k + 1
    if start + i < mid:
        while k < end:
            alist[k] = left[i]
            i = i + 1
            k = k + 1
    else:
        while k < end:
            alist[k] = right[j]
            j = j + 1
            k = k + 1

alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
```

merge_sort(alist, 0, len(alist))

print('Sorted list: ', end='')

print(alist)

Результат программы:

Enter the list of numbers:

4 5 6 8 -9 -65 4 12 65 -8

Sorted list: [-65, -9, -8, 4, 4, 5, 6, 8, 12, 65]

**** Process exited - Return Code: 0 ****

Press Enter to exit terminal

Введение

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.

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

9.1 Линейный поиск

Линейный поиск — это один из самых простых и понятных алгоритмов поиска. Мы можем думать о нем как о расширенной версии нашей собственной реализации оператора in в Python.

Суть алгоритма заключается в том, чтобы перебрать массив и вернуть индекс первого вхождения элемента, когда он найден:

```
def LinearSearch(lys, element):
    for i in range (len(lys)):
        if lys[i] == element:
            return i
    return -1
```

Итак, если мы используем функцию для вычисления:

>>> print(LinearSearch([1,2,3,4,5,2,1], 2))

То получим следующий результат:

1

Это индекс первого вхождения искомого элемента, учитывая, что нумерация элементов в Python начинается с нуля.

Временная сложность линейного поиска равна O(n). Это означает, что время, необходимое для выполнения, увеличивается с увеличением количества элементов в нашем входном списке lvs.

Линейный поиск не часто используется на практике, потому что такая же эффективность может быть достигнута с помощью встроенных методов или существующих операторов. К тому же, он не такой быстрый и эффективный, как другие алгоритмы поиска.

Линейный поиск хорошо подходит для тех случаев, когда нам нужно найти первое вхождение элемента в несортированной коллекции. Это связано с тем, что он не требует сортировки коллекции перед поиском (в отличие от большинства других алгоритмов поиска).

9.2 Бинарный поиск

Бинарный поиск работает по принципу «разделяй и властвуй». Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

Предполагая, что мы ищем значение val в отсортированном массиве, алгоритм сравнивает val со значением среднего элемента массива, который мы будем называть mid.

Если mid — это тот элемент, который мы ищем (в лучшем случае), мы возвращаем его индекс.

Если нет, мы определяем, в какой половине массива мы будем искать val дальше, основываясь на том, меньше или больше значение val значения mid, и отбрасываем вторую половину массива.

Затем мы рекурсивно или итеративно выполняем те же шаги, выбирая новое значение для mid, сравнивая его с val и отбрасывая половину массива на каждой итерации алгоритма.

Алгоритм бинарного поиска можно написать, как рекурсивно, так и итеративно.

Поскольку хороший алгоритм поиска должен быть максимально быстрым и точным, давайте рассмотрим итеративную реализацию бинарного поиска:

```
def BinarySearch(lys, val):
    first = 0
    last = len(lys)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if lys[mid] == val:
            index = mid
        else:
            if val<lys[mid]:
                last = mid -1
            else:
                first = mid +1
    return index
```

Если мы используем функцию для вычисления:

```
>>> BinarySearch([10,20,30,40,50], 20)
```

То получим следующий результат, являющийся индексом искомого значения:

```
1
```

На каждой итерации алгоритм выполняет одно из следующих действий:
Возврат индекса текущего элемента.
Поиск в левой половине массива.
Поиск в правой половине массива.

Мы можем выбрать только одно действие на каждой итерации. Также на каждой итерации наш массив делится на две части. Из-за этого временная сложность двоичного поиска равна O(log n).

Одним из недостатков бинарного поиска является то, что если в массиве имеется несколько вхождений элемента, он возвращает индекс не первого элемента, а ближайшего к середине:

>>> print(BinarySearch([4,4,4,4,4], 4))

После выполнения этого фрагмента кода будет возвращен индекс среднего элемента:

2

Для сравнения: выполнение линейного поиска по тому же массиву вернет индекс первого элемента:

0

Однако мы не можем категорически утверждать, что двоичный поиск не работает, если массив содержит дубликаты. Он может работать так же, как линейный поиск, и в некоторых случаях возвращать первое вхождение элемента. Например:

>>> print(BinarySearch([1,2,3,4,4,4,5], 4))
3

Бинарный поиск довольно часто используется на практике, потому что он эффективен и быстр по сравнению с линейным поиском. Однако у него есть некоторые недостатки, такие как зависимость от оператора //.

Регулярные выражения

Введение

Регулярное выражение — это строка, задающая шаблон поиска подстрок в тексте. Одному шаблону может соответствовать много разных строчек. Термин «Регулярные выражения» является переводом английского словосочетания «Regular expressions». Перевод не очень точно отражает смысл, правильнее было бы «шаблонные выражения». Регулярное выражение, или коротко «регулярка», состоит из обычных символов и специальных командных последовательностей. Например, \d задаёт любую цифру, а \d+ — задает любую последовательность из одной или более цифр. Работа с регулярками реализована во всех современных языках программирования.

Основы регулярных выражений

Регулярками называются шаблоны, которые используются для поиска соответствующего фрагмента текста и сопоставления символов.

Грубо говоря, у нас есть input-поле, в которое должен вводиться email-адрес. Но пока мы не зададим проверку валидности введённого email-адреса, в этой строке может оказаться совершенно любой набор символов, а нам это не нужно.

Чтобы выявить ошибку при вводе некорректного адреса электронной почты, можно использовать следующее регулярное выражение:

r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)+$'

По сути, наш шаблон — это набор символов, который проверяет строку на соответствие заданному правилу.

Синтаксис RegEx

Синтаксис у регулярок необычный. Любая строка (в которой нет символов .^$*+?{}[]\|()) сама по себе является регулярным выражением. Так, выражению Хаха будет соответствовать строка “Хаха” и только она. Регулярные выражения являются регистрозависимыми, поэтому строка “хаха” (с маленькой буквы) уже не будет соответствовать выражению выше. Подобно строкам в языке Python, регулярные выражения имеют спецсимволы .^$*+?{}[]\|(), которые в регулярках являются управляющими конструкциями. Для написания их просто как символов требуется их экранировать, для чего нужно поставить перед ними знак \. Так же, как и в питоне, в регулярных выражениях выражение \n соответствует концу строки, а \t — табуляции.

Символы могут быть как буквами или цифрами, так и метасимволами, которые задают шаблон строки:

Также есть дополнительные конструкции, которые позволяют сокращать регулярные выражения:

\d — соответствует любой одной цифре и заменяет собой выражение [0-9];

\D — исключает все цифры и заменяет [^0-9];

\w — заменяет любую цифру, букву, а также знак нижнего подчёркивания;

\W — любой символ кроме латиницы, цифр или нижнего подчёркивания;

\s — соответствует любому пробельному символу;

\S — описывает любой непробельный символ.

Для чего используются регулярные выражения:

- для определения нужного формата, например телефонного номера или email-адреса;
- для разбивки строк на подстроки;
- для поиска, замены и извлечения символов;
- для быстрого выполнения нетривиальных операций.

Синтаксис таких выражений в основном стандартизирован, так что вам следует понять их лишь раз, чтобы использовать в любом языке программирования.

Но не стоит забывать, что регулярные выражения не всегда оптимальны, и для простых операций часто достаточно встроенных в Python функций.

Примеры регулярных выражений

Регулярка Её смысл

simple text В точности текст «simple text»

\d{5} Последовательности из 5 цифр

\d означает любую цифру

{5} — ровно 5 раз

\d\d/\d\d/\d{4} Даты в формате ДД/ММ/ГГГГ

(и прочие куски, на них похожие, например, 98/76/5432)

\b\w{3}\b Слова в точности из трёх букв

\b означает границу слова

(с одной стороны буква, а с другой — нет)

\w — любая буква,

{3} — ровно три раза

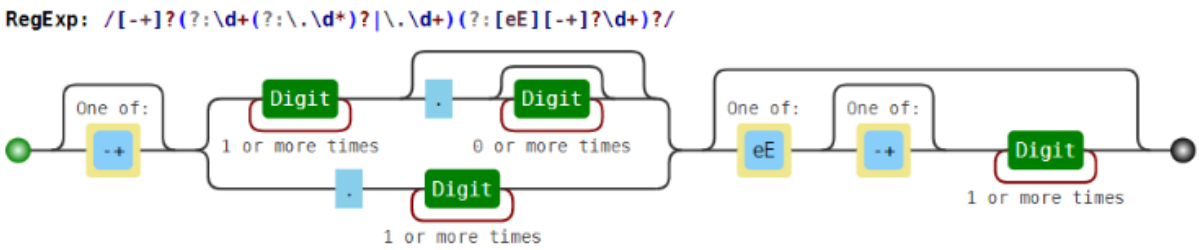
[-+]?\d+ Целое число, например, 7, +17, -42, 0013 (возможны ведущие нули)

[-+]? — либо -, либо +, либо пусто

\d+ — последовательность из 1 или более цифр

[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)? Действительное число, возможно в экспоненциальной записи

Например, 0.2, +5.45, -.4, 6e23, -3.17E-14.



Регулярные выражения в Python

В Python для работы с регулярками есть модуль re. Его нужно просто импортировать:

import re

А вот наиболее популярные методы, которые предоставляет модуль:

re.match()

re.search()

re.findall()

re.split()

re.sub()

re.compile()

Рассмотрим подробнее по отдельности:

e.match(pattern, string)

Этот метод ищет по заданному шаблону в начале строки. Например, если мы вызовем метод match() на строке «AV Analytics AV» с шаблоном «AV», то он завершится успешно. Но если мы будем искать «Analytics», то результат будет отрицательный:

```
import re
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result
```

Результат:

<_sre.SRE_Match object at 0x0000000009BE4370>

Искомая подстрока найдена. Чтобы вывести её содержимое, применим метод group() (мы используем «r» перед строкой шаблона, чтобы показать, что это «сырая» строка в Python):

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result.group(0)
```

Результат:

AV

Теперь попробуем найти «Analytics» в данной строке. Поскольку строка начинается на «AV», метод вернет None:

```
result = re.match(r'Analytics', 'AV Analytics Vidhya AV')
print result
```

Результат:

None

Также есть методы start() и end() для того, чтобы узнать начальную и конечную позицию найденной строки.

result = re.match(r'AV', 'AV Analytics Vidhya AV')
print result.start()
print result.end()

Результат:
0
2

Эти методы иногда очень полезны для работы со строками.

re.search(pattern, string)

Метод похож на match(), но ищет не только в начале строки. В отличие от предыдущего, search() вернёт объект, если мы попытаемся найти «Analytics»:

result = re.search(r'Analytics', 'AV Analytics Vidhya AV')
print result.group(0)
Результат:
Analytics

Метод search() ищет по всей строке, но возвращает только первое найденное совпадение.

re.findall(pattern, string)

Возвращает список всех найденных совпадений. У метода findall() нет ограничений на поиск в начале или конце строки. Если мы будем искать «AV» в нашей строке, он вернет все вхождения «AV». Для поиска рекомендуется использовать именно findall(), так как он может работать и как re.search(), и как re.match().

result = re.findall(r'AV', 'AV Analytics Vidhya AV')
print result
Результат:
['AV', 'AV']

re.split(pattern, string, [maxsplit=0])

Этот метод разделяет строку по заданному шаблону.

result = re.split(r'y', 'Analytics')
print result
Результат:
['Anal', 'tics']

В примере мы разделили слово «Analytics» по букве «y». Метод split() принимает также аргумент maxsplit со значением по умолчанию, равным 0. В данном случае он разделит строку столько раз, сколько возможно, но если указать этот аргумент, то разделение будет произведено не более указанного количества раз. Давайте посмотрим на примеры Python RegEx:

result = re.split(r'i', 'Analytics Vidhya')
print result

Результат:
['Analyt', 'cs V', 'dhya'] # все возможные участки.

result = re.split(r'i', 'Analytics Vidhya',maxsplit=1)
print result

Результат:
['Analyt', 'cs Vidhya']

Мы установили параметр maxsplit равным 1, и в результате строка была разделена на две части вместо трех.

re.sub(pattern, repl, string)

Ищет шаблон в строке и заменяет его на указанную подстроку. Если шаблон не найден, строка остается неизменной.

```
result = re.sub(r'India', 'the World', 'AV is largest Analytics community of India')
print result
```

Результат:

'AV is largest Analytics community of the World'

re.compile(pattern, repl, string)

Мы можем собрать регулярное выражение в отдельный объект, который может быть использован для поиска. Это также избавляет от переписывания одного и того же выражения.

```
pattern = re.compile('AV')
result = pattern.findall('AV Analytics Vidhya AV')
print result
result2 = pattern.findall('AV is largest analytics community of India')
print result2
```

Результат:

['AV', 'AV']

['AV']

До сих пор мы рассматривали поиск определенной последовательности символов. Но что, если у нас нет определенного шаблона, и нам надо вернуть набор символов из строки, отвечающий определенным правилам? Такая задача часто стоит при извлечении информации из строк. Это можно сделать, написав выражение с использованием специальных символов. Вот наиболее часто используемые из них:

Оператор	Описание
.	Один любой символ, кроме новой строки \n.
?	0 или 1 вхождение шаблона слева
+	1 и более вхождений шаблона слева
*	0 и более вхождений шаблона слева
\w	Любая цифра или буква (\W — все, кроме буквы или цифры)
\d	Любая цифра [0-9] (\D — все, кроме цифры)
\s	Любой пробельный символ (\S — любой непробельный символ)
\b	Граница слова
[..]	Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)
\	Экранирование специальных символов (\. означает точку или \+ — знак «плюс»)
^ и $	Начало и конец строки соответственно
{n,m}	От n до m вхождений ({,m} — от 0 до m)
a|b	Соответствует a или b
()	Группирует выражение и возвращает найденный текст
\t, \n, \r	Символ табуляции, новой строки и возврата каретки соответственно