Сортировка

Введение

Сортировка – это перестановка элементов массива в заданном порядке.

Порядок сортировки может быть любым; для чисел обычно рассматривают сортировку по возрастанию (или убыванию) значений. Для массивов, в которых есть одинаковые элементы, используются понятия «сортировка по неубыванию» и «сортировка по невозрастанию».

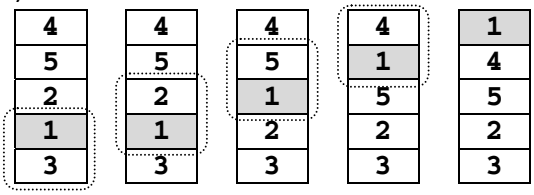
Возникает естественный вопрос: «зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту облегчает поиск нужной информации.

Программисты изобрели множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие (на больших массивах) и 2) сложные, но быстрые. Мы изучим два классических метода из первой группы и один метод из второй – знаменитую «быструю сортировку», предложенную Ч. Хоаром.

Метод пузырька (сортировка обменами)

Название этого метода произошло от известного физического явления – пузырёк воздуха в воде поднимается вверх. Если говорить о сортировке массива, сначала поднимается «наверх» (к началу массива) самый «лёгкий» (минимальный) элемент, затем следующий и т.д.

Сначала сравниваем последний элемент с предпоследним. Если они стоят неправильно (меньший элемент «ниже»), то меняем их местами. Далее так же рассматриваем следующую пару элементов и т.д. (см. рисунок).



Когда мы обработали пару (**A[0]**, **A[1]**), минимальный элемент стоит на месте **A[0]**. Это значит, что на следующих этапах его можно не рассматривать. Первый цикл, устанавливающий на свое место первый (минимальный) элемент, можно на псевдокоде записать так:

```
для j от N-2 до 0 шаг -1
  if A[j+1]<A[j]:
    поменять местами A[j] и A[j+1]
```

Заголовок цикла тоже написан на псевдокоде. Обратите внимание, что на очередном шаге сравниваются элементы **A[j]** и **A[j+1]**, поэтому цикл начинается с **j=N-2**. Если начать с **j=N-1**, то на первом же шаге получаем выход за границы массива – обращение к элементу **A[N]**. Поскольку цикл **for** в Python «не доходит» до конечного значения, мы ставим его равным «–1», а не 0:

```
for j in range(N-2,-1,-1):
  if A[j+1]<A[j]:
    поменять местами A[j] и A[j+1]
```

То есть, в записи **range(N-2,-1,-1)** первое число «–1» – это ограничитель (число. следующее за конечным значением), а второе число «–1» - шаг изменения переменной цикла.

За один проход такой цикл ставит на место один элемент. Чтобы «подтянуть» второй элемент, нужно написать еще один почти такой же цикл, который будет отличаться только конечным значением **j** в заголовке цикла. Так как верхний элемент уже стоит на месте, его не нужно трогать:

```
for j in range(N-2,0,-1):
  if A[j+1]<A[j]:
```

```
    поменять местами A[j] и A[j+1]
```

При установке следующего (3-го по счёту) элемента конечное при вызове функции **range** будет равно 1 и т.д.

Таких циклов нужно сделать **N-1** – на 1 меньше, чем количество элементов массива. Почему не **N**? Дело в том, что если **N-1** элементов поставлены на свои места, то оставшийся автоматически встает на своё место – другого места нет. Поэтому полный алгоритм сортировки представляет собой такой вложенный цикл:

```
for i in range(N-1):
  for j in range(N-2,i-1,-1):
    if A[j+1] < A[j]:
      A[j], A[j+1] = A[j+1], A[j]
```

Записать полную программу вы можете самостоятельно.

Часто используют более простой вариант этого метода, который можно назвать «методом камня» – самый «тяжёлый» элемент опускается в конец массива.

```
for i in range(N):
  for j in range(N-i-1):
    if A[j+1] < A[j]:
      A[j], A[j+1] = A[j+1], A[j]
```

Метод выбора

Еще один популярный простой метод сортировки – метод выбора, при котором на каждом этапе выбирается минимальный элемент (из оставшихся) и ставится на свое место. Алгоритм в общем виде можно записать так:

```
for i in range(N-1):
  найти номер nMin минимального элемента среди A[i]..A[N-1]
  if i != nMin:
    поменять местами A[i] и A[nMin]
```

Здесь перестановка происходит только тогда, когда найденный минимальный элемент стоит не на своём месте, то есть **i != nMin**. Поскольку поиск номера минимального элемента выполняется в цикле, этот алгоритм сортировки также представляет собой вложенный цикл:

```
for i in range(N-1):
  nMin = i
  for j in range(i+1,N):
    if A[j] < A[nMin]:
      nMin = j
  if i!= nMin:
     A[i], A[nMin] = A[nMin], A[i]
```

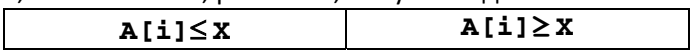
«Быстрая сортировка»

Ч.Э. Хоар (p. 1934)
(en.*wikipedia.org*)

Методы сортировки, описанные в предыдущем параграфе, работают медленно для больших массивов данных (более 1000 элементов). Поэтому в середине XX века математики и программисты серьезно занимались разработкой более эффективных алгоритмов сортировки. Один из самых популярных «быстрых» алгоритмов, разработанный в 1960 году английским учёным Ч. Хоаром, так и называется – «быстрая сортировка» (англ. *quicksort*).

Будем исходить из того, что сначала лучше делать перестановки элементов массива на большом расстоянии. Предположим, что у нас есть **N** элементов и известно, что они уже отсортированы в обратном порядке. Тогда за **N/2** обменов можно отсортировать их как нужно – сначала поменять местами первый и последний, а затем последовательно двигаться с двух сторон к центру. Хотя это справедливо только тогда, когда порядок элементов обратный, подобная идея положена в основу алгоритма *Quicksort*.

Пусть дан массив **A** из **N** элементов. Выберем сначала наугад любой элемент массива (назовем его **X**). На первом этапе мы расставим элементы так, что слева от некоторой границы (в первой группе) находятся все числа, меньшие или равные **X**, а справа (во второй группе) – большие или равные **X**. Заметим, что элементы, равные **X**, могут находиться в обеих частях.

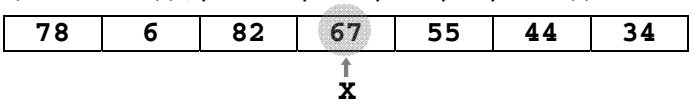


Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй и наоборот. Поэтому далее достаточно отсортировать *отдельно* каждую часть массива. Такой подход называют «разделяй и властвуй» (англ. *divide and conquer*).

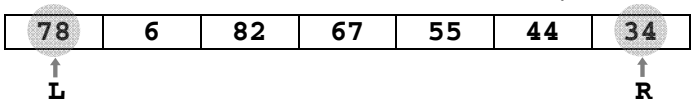
Лучше всего выбирать **X** так, чтобы в обеих частях было равное количество элементов. Такое значение **X** называется медианой массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив[17], то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве **X** выбирают средний элемент массива или элемент со случайным номером.

Сначала будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **X** (и, следовательно, должен стоять справа от **X**). Затем просматриваем массив справа до тех пор, пока не обнаружим элемент меньше **X** (он должен стоять слева от **X**). Теперь поменяем местами эти два элемента и продолжим просмотр до тех пор, пока два «просмотра» не встретятся где-то в середине массива. В результате массив окажется разбитым на 2 части: левую со значениями меньшими или равными **X**, и правую со значениями большими или равными **X**. На этом первый этап («разделение») закончен. Затем такая же процедура применяется к обеим частям массива до тех пор, пока в каждой части не останется один элемент (и таким образом, массив будет отсортирован).

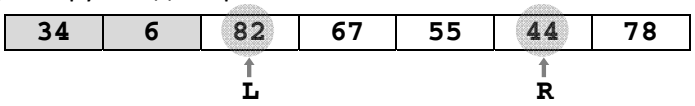
Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив



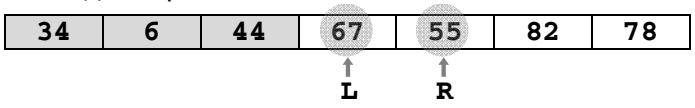
Выберем в качестве **X** средний элемент массива, то есть 67. Найдем первый слева элемент массива, который больше или равен **X** и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через **L**. Теперь находим самый правый элемент, который меньше **X** и должен стоять в первой части. Это число 34. Обозначим его индекс через **R**.



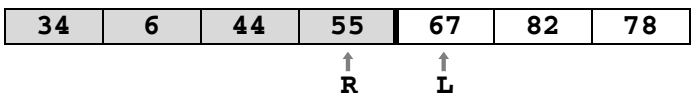
Теперь поменяем местами два этих элемента. Сдвигая переменную **L** вправо, а **R** – влево, находим следующую пару, которую надо переставить. Это числа 82 и 44.



Следующая пара элементов для перестановки – числа 67 и 55.



После этой перестановки дальнейший поиск приводит к тому, что переменная **L** становится больше **R**, то есть массив разбит на две части. В результате все элементы массива, расположенные левее **A[L]**, меньше или равны **X**, а все правее **A[R]** – больше или равны **X**.



Таким образом, сортировка исходного массива свелась к двум сортировкам частей массива, то есть к двум задачам того же типа, но меньшего размера. Теперь нужно применить тот же алго-

[17] Хотя есть и другие, тоже достаточно непростые алгоритмы.

ритм к двум полученным частям массива: первая часть – с 1-ого до **R**-ого элемента, вторая часть – с **L**-ого до последнего элемента. Как вы знаете, такой прием называется *рекурсией*.

Процедура сортировки принимает три параметра: сам массив (список) и значения индексов, ограничивающие её «рабочую зону»: **nStart** – номер первого элемента, и **nEnd** – номер последнего элемента. Если **nStart = nEnd**, то в «рабочей зоне» один элемент и сортировка не требуется, то есть нужно выйти из процедуры. В этом случае рекурсивные вызовы заканчиваются.

Приведем полностью процедуру быстрой сортировки на Python:

```
def qSort ( A, nStart, nEnd ):
  if nStart >= nEnd: return
  L = nStart; R = nEnd
  X = A[(L+R)//2]
  while L <= R:
    while A[L] < X: L += 1   # разделение
    while A[R] > X: R -= 1
    if L <= R:
      A[L], A[R] = A[R], A[L]
      L += 1; R -= 1
  qSort ( A, nStart, R )   # рекурсивные вызовы
  qSort ( A, L, nEnd )
```

Для того, чтобы отсортировать весь массив, нужно вызвать эту процедуру так:

```
qSort ( A, 0, N-1 )
```

Скорость работы быстрой сортировки зависит от того, насколько удачно выбирается вспомогательный элемент **X**. Самый лучший случай – когда на каждом этапе массив делится на две равные части. Худший случай – когда в одной части оказывается только один элемент, а в другой – все остальные. При этом глубина рекурсии достигает **N**, что может привести к переполнению стека (нехватке стековой памяти).

Для того, чтобы уменьшить вероятность худшего случая, в алгоритм вводят случайность: в качестве X на каждом шаге выбирают не середину рабочей части массива, а элемент со случайным номером:

```
X = A[randint(L,R)]
```

Напомним, что функцию **randint** нужно импортировать из модуля **random**.

Эта процедура сортирует массив «на месте», без создания нового списка. Можно также оформить алгоритм в виде функции, которая возвращает новый список, не затрагивая существующий:

```
import random
def qSort ( A ):
  if len(A) <= 1: return A          # (1)
  X = random.choice(A)              # (2)
  B1 = [b for b in A if b < X]      # (3)
  BX = [b for b in A if b == X]     # (4)
  B2 = [b for b in A if b > X]      # (5)
  return qSort(B1)+BX+qSort(B2)     # (6)
```

Дадим некоторые пояснения к этой функции. Если длина массива не больше 1, то ответ – это сам массив, сортировать тут нечего (строка 1). Иначе выбираем в качестве разделителя («стержневого элемента», англ. *pivot*) **X** случайный элемент массива (строка 2), для этого используется функция **choice** (в переводе с англ. – выбор) из модуля **random**.. В строках 3-5 с помощью генераторов списков создаем три вспомогательных массива: в массив **B1** войдут все элементы, меньшие **X**, в массив **BX** – все элементы, равные **X**, а в массив **B2** – все элементы, большие **X**. Теперь остается отсортировать списки **B1** и **B2** (вызвав функцию **qSort** рекурсивно) и построить окончательный список-результат, который «складывается» из отсортированного списка **B1**, списка **BX**, и отсортированного списка **B2**.

Для того, чтобы построить отсортированный массив, нужно вызвать эту функцию так:

```
Asorted = qSort(A)
```

В таблице сравнивается время сортировки (в секундах) массивов разного размера, заполненных случайными значениями, с использованием трёх изученных алгоритмов.

N	метод пузырька	метод выбора	быстрая сортировка
1000	0,09 с	0,05 с	0,002 с
5000	2,4 с	1,2 с	0,014 с
15000	22 с	11 с	0,046 с

Как показывают эти данные, преимущество быстрой сортировки становится подавляющим при увеличении **N**.

Сортировка в Python

В отличие от многих популярных языков программирования, в Python есть встроенная функция для сортировки массивов (списков), которая называется **sorted**. Она использует алгоритм *Timsort*[18]. Вот так можно построить новый массив **B**, который совпадает с отсортированным в порядке возрастания массивом **A**:

```
B = sorted ( A )
```

По умолчанию выполняется сортировка по возрастанию (неубыванию). Для того, чтобы отсортировать массив по убыванию (невозрастанию), нужно задать дополнительный параметр **reverse** (от англ. «обратный»), равный **True**:

```
B = sorted ( A, reverse = True )
```

Иногда требуется особый, нестандартный порядок сортировки, который отличается от сортировок «по возрастанию» и «по убыванию». В этом случае используют еще один именованный параметр функции **sorted**, который называется **key** (от англ. «ключ»). В этот параметр нужно записать название функции (фактически – ссылку на объект-функцию), которая возвращает число (или символ), используемое при сравнении двух элементов массива.

Предположим, что нам нужно отсортировать числа по возрастанию последней цифры (поставить сначала все числа, оканчивающиеся на 0, затем – все, оканчивающиеся на 1 и т.д.). В этом случае ключ сортировки – это последняя цифра числа, поэтому напишем функцию, которая выделяет эту последнюю цифру:

```
def lastDigit ( n ):
  return n % 10
```

Тогда сортировка массива A по возрастанию последней цифры запишется так:

```
B = sorted ( A, key = lastDigit )
```

Функция **lastDigit** получилась очень короткая, и часто не хочется создавать лишнюю функцию с помощью оператора **def**, особенно тогда, когда она нужна всего один раз. В таких случаях удобно использовать функции без имени – так называемые «лямбда-функции», которые записываются прямо при вызове функции в параметре **key**:

```
B = sorted ( A, key = lambda x: x % 10 )
```

В рамку выделена лямбда-функция, которая для переданного ей значения **x** возвращает результат **x % 10**, то есть последнюю цифру десятичной записи числа.

Функция **sorted** не изменяет исходный массив и возвращает его отсортированную копию. Если нужно отсортировать массив «на месте», лучше использовать метод **sort**, который определён для списков:

```
A.sort ( key = lambda x: x % 10, reverse = True)
```

Он имеет те же именованные параметры, что и функция **sorted**, но изменяет исходный список. В приведённом примере элементы массива **A** будут отсортированы по убыванию последней цифры.

[18] http://ru.wikipedia.org/wiki/Timsort