Алгоритмы сортировки

Артамонов Ю.Н.

Международный университет
природы, общества и человека "Дубна"
филиал Котельники

5 апреля 2016 г.

Содержание

Общая постановка задачи

- Исходные данные - поледовательность чисел $a_1, a_2, \dots, a_n$.
- Результат сортировки - последовательность тех же чисел,идущих в неубывающем порядке $a'_1, a'_2, \dots, a'_n, \forall i, j : i < j \Rightarrow a'_i \leq a'_j$.
- В качестве объектов для сортировки обычно выступают массивы, но возможны и другие объекты (списки, строки и т.д.).
- Алгоритмы сортировки можно разделить на два вида: устойчивые и неустойчивые. К устойчивым алгоритмам сортировки относятся такие алгоритмы, которые при наличии в наборе данных нескольких равных элементов в отсортированном наборе оставляют их в том же порядке, в котором эти элементы были в исходном наборе.
- Любой алгоритм сортировки, основанный на сравнении, принадлежит к классу $O(n \cdot log(n))$.

Пузырьковая сортировка - описание

- Процесс начинается с самого левого элемента a_1, его сравнивают с ближайшим соседом a_2, стоящим справа;
- Если $a_1 > a_2$, элементы меняют местами;
- Затем берут текущий второй элемент a'_2 и сравнивают его с третьим a_3, если $a'_2 > a_3$ - их меняют местами;
- После прохождения всего списка, самым правым окажется самый максимальный элемент $a'_n = max(a_1, a_2, , a_n)$. Он стоит на своем месте, поэтому предыдущие шаги повторяют для подсписка $a_1, a_2, \ldots, a_{n-1}$;
- На каждом проходе подсписок сокращается на один элемент. Алгоритм останавливается, когда от подсписка остается один элемент. В этом случае весь список отсортирован.

Реализация пузырьковой сортировки

```
#Без возможных улучшений
def bubble_sort(lst):
    for i in range(len(lst)):
        for j in range(len(lst)-i-1):
            if lst[j]>=lst[j+1]:
                lst[j],lst[j+1]=lst[j+1],lst[j]
    return lst
```

Реализация пузырьковой сортировки для демонстрации

```
#Без возможных улучшений
#s=[4,5,2,7,0]
def bubble_sort(lst):
    #Количество операций сравнения
    compare=0
    #Количество перестановок
    permulation=0
    for i in range(len(lst)):
        for j in range(len(lst)-i-1):
            compare += 1
            if lst[j]>=lst[j+1]:
                lst[j],lst[j+1]=lst[j+1],lst[j]
                permulation +=1
            print(lst)
    return [compare,permulation]
```

Демонстрация работы

s = [4,5,2,7,0]
bubble_sort(s)
[4, 5, 2, 7, 0]
[4, 2, 5, 7, 0] - 1 перестановка
[4, 2, 5, 7, 0]
[4, 2, 5, 0, 7] - закончилась первая итерация внешнего цикла, 2 перестановка
[2, 4, 5, 0, 7] - 3 перестановка
[2, 4, 5, 0, 7]
[2, 4, 0, 5, 7] - закончилась вторая итерация внешнего цикла, 4 перестановка
[2, 4, 0, 5, 7]
[2, 0, 4, 5, 7] - закончилась третья итерация внешнего цикла, 5 перестановка
[0, 2, 4, 5, 7] - закончилась четвертая итерация внешнего цикла, 6 перестановка
[10, 6]

Пузырьковая сортировка - улучшения

- Пузырьковую сортировку можно улучшить, если при выходе из внутреннего цикла продолжать внешний цикл только в случае, если была хотя бы одна перестановка, в противном случае список уже отсортирован.

```
def bubble_best_sort(lst):
    compare = 0; permulation = 0; pr = True; i = 0
    while pr & (i<len(lst)):
        pr=False
        for j in range(len(lst)-i-1):
            compare += 1
            if lst[j]>=lst[j+1]:
                lst[j],lst[j+1]=lst[j+1],lst[j]
                permulation += 1; pr = True
            print(lst)
        i += 1
    return [compare,permulation]
```

Улучшение пузырьковой сортировки - шейкер-сортировка

- Идея состоит в том, что вместо прохода списка всегда слева направо, в шейкер-сортировке предлагается проходить список попеременно - сначала слева направо (справа фиксируется наибольший элемент), потом справа налево (слева фиксируется наименьший элемент) и т.д., пока список не будет отсортирован. Практика показывает, что такая сортировка немного быстрее.

Реализация шейкер-сортировки

```
def shaker_sort(lst):
    compare = 0; permulation = 0; pr = True; i = 0
    while pr & (i<(len(lst)//2)):
        pr=False
        for j in range(len(lst)-i-1):
            compare += 1
            if lst[j]>=lst[j+1]:
                lst[j],lst[j+1]=lst[j+1],lst[j]
                permulation += 1; pr = True
            print(lst)
        for j in range(len(lst)-i-3):
            compare += 1
            if lst[len(lst)-i-2-j]<=lst[len(lst)-i-3-j]:
                lst[len(lst)-i-2-j],lst[len(lst)-i-3-j]=
                            lst[len(lst)-i-3-j],lst[len(lst)-i-2-j]
                permulation += 1; pr = True
            print(lst)
        i += 1
    return [compare,permulation]
```

Общие положения по оценке вычислительной сложности алгоритмов

- При анализе вычислительной сложности алгоритма принято рассматривать 3 случая: лучший, худший и средний.
- В теоретическом плане для сравнения разных алгоритмов исходят из самого худшего случая, то есть ориентируются на пессимистический случай без учета вероятности, с которой он может случиться.
- В практическом плане целесообразно оценивать среднее время работы на случайной выборке.
- Вычислительную сложность характеризуют как количество сравнений, так и количество перестановок, однако в теоретическом плане оценивают только количество сравнений, т.к. количество перестановок сильно зависит от исходного несортированного списка.

Теоретический анализ пузырьковой сортировки

- Пузырьковая сортировка относится к классу неустойчивых алгоритмов. Однако если вместо условия «if lst[j]>=lst[j+1]:» использовать строгое неравенство, то сортировка становится устойчивой, однако это не даст существенного выигрыша.
- Самым худшим случаем для пузырьковой сортировки является обратно отсортированный массив
- На первой итерации внешнего цикла выполняется $n-1$ сравнение, на второй итерации выполняется $n-2$ сравнения и т.д., на последней итерации выполняется 1 сравнение:
$$(n-1)+(n-2)+(n-3)+\ldots+3+2+1=\frac{n\cdot(n-1)}{2}$$
Получаемая оценка:
$$O(n^2)$$
- Лучший случай - когда массив уже отсортирован, тогда в случае первого улучшения потребуется $n-1$ операция сравнения, т.е. оценка $O(n)$

Средний случай для пузырьковой сортировки (вспомогательная функция)

```
import random
def gen(n):
    return random.sample(range(2*n),n)
```

Средний случай для пузырьковой сортировки (расчеты)

```
result=[]
n=100
for i in range(n-3):
    comp_bubble=comp_best=comp_shaker=0
    perm_bubble=perm_best=perm_shaker=0
    for j in range(20):
        a=gen(i+3); b=list(a); c=list(a)
        result_bubble=bubble_sort(a);
        result_best=bubble_best_sort(b)
        result_shaker=shaker_sort(c)
        comp_bubble=comp_bubble+result_bubble[0]
        comp_best=comp_best+result_best[0]
        comp_shaker=comp_shaker+result_shaker[0]
        perm_bubble=perm_bubble+result_bubble[1]
        perm_best=perm_best+result_best[1]
        perm_shaker=perm_shaker+result_shaker[1]
    result=result+[[i,comp_bubble/20,perm_bubble/20],
                   [i,comp_best/20,perm_best/20],
                   [i,comp_shaker/20,perm_shaker/20]]
```

Средний случай для пузырьковой сортировки (вывод результата)

```
j=0
for i in range(len(result)//3):
    print(result[j],result[j+1],result[j+2])
    j=j+3
```

Средний случай для пузырьковой сортировки (результат)

[0, 3.0, 1.65] [0, 2.9, 1.65] [0, 2.0, 1.25]
[1, 6.0, 3.3] [1, 5.65, 3.3] [1, 5.9, 3.3]
[2, 10.0, 5.45] [2, 9.55, 5.45] [2, 10.0, 5.45]
[3, 15.0, 6.9] [3, 13.6, 6.9] [3, 17.2, 6.9]
[4, 21.0, 9.85] [4, 19.15, 9.85] [4, 23.1, 9.85]
[5, 28.0, 13.3] [5, 26.3, 13.3] [5, 31.7, 13.3]
[6, 36.0, 17.95] [6, 33.4, 17.95] [6, 40.4, 17.95]
....
[92, 4465.0, 2200.8] [92, 4397.6, 2200.8] [92, 4202.3, 2200.8]
[93, 4560.0, 2295.95] [93, 4490.7, 2295.95] [93, 4254.8, 2295.95]
[94, 4656.0, 2303.8] [94, 4574.9, 2303.8] [94, 4376.3, 2303.8]
[95, 4753.0, 2353.35] [95, 4676.75, 2353.35] [95, 4422.5, 2353.35]
[96, 4851.0, 2418.1] [96, 4787.8, 2418.1] [96, 4610.6, 2418.1]

Сортировка методом выбора - описание

- Находим минимальный элемент $a_1' = min(a_1, a_2, \ldots, a_n)$.
- Меняем a_1' местами с первым элементом.
- Выбираем минимальный элемент из оставшихся элементов $a_2' = min(a_2, \ldots, a_n)$.
- Меняем a_2' местами со вторым элементом.
- Процес продолжается до тех пор, пока не дойдем до последнего элемента.

Реализация сортировки выбором

```
def select_sort(lst):
    for i in range(len(lst)):
        min=s[i]
        index=i
        for j in range(i+1,len(lst)):
            if min>lst[j]:
                min=lst[j]
                index=j
        if index != i:
            a=lst[i]
            lst[i]=lst[index]
            lst[index]=a
    return lst
```

Реализация сортировки выбором для демонстрации

```
def select_sort(lst):
    compare = 0; permulation = 0
    for i in range(len(lst)):
        min=s[i]
        index=i
        for j in range(i+1,len(lst)):
            compare += 1
            if min>lst[j]:
                min=lst[j]
                index=j
        if index != i:
            a=lst[i]
            lst[i]=lst[index]
            lst[index]=a
            permulation += 1
        print(lst)
    return [compare,permulation]
```

Демонстрация работы

```
s = [4,5,2,7,0]
select_sort(s)
[0, 5, 2, 7, 4]
[0, 2, 5, 7, 4]
[0, 2, 4, 7, 5]
[0, 2, 4, 5, 7]
[0, 2, 4, 5, 7]
[10, 4]
```

Теоретический анализ сортировки выбором

- Сортировка выбором относится к классу устойчивых алгоритмов. В отличие от пузырьковой сортировки здесь при каждой итерации внешнего цикла выполняется только одна перестановка. Если стоимость перестановки много больше стоимости сравнения, то такая сортировка оказывается приемлемой.
- На первой итерации внешнего цикла выполняется $n-1$ сравнение, на второй итерации выполняется $n-2$ сравнения и т.д., на последней итерации выполняется 1 сравнение:

$$(n-1)+(n-2)+(n-3)+\ldots+3+2+1=\frac{n\cdot(n-1)}{2}$$

Получаемая оценка:

$$O(n^2)$$

- Лучший случай - когда массив уже отсортирован, тогда не требуется ни одной перестановки.

Задание 1

Сравнить вычислительную сложность пузырьковой сортировки (улучшение шейкер-сортировка) с сортировкой методом выбора для среднего случая.

Сортировка методом вставки - описание

- Сравниваем первые два элемента, при необходимости меняем их местами.
- Смотрим на третий элемент и вставляем его в нужную позицию между первым и вторым.
- Смотрим на четвертый элемент и вставляем его в нужную позицию между первым, вторым и третьим.
- Продолжаем аналогичные действия для остальных элементов, пока не поставим на правильное место последний элемент.

Реализация сортировки вставками

```
def insert_sort(lst):
    for i in range(1,len(lst)):
        j=0
        while (lst[j]<lst[i]) & (j<i):
            j += 1
        if (lst[j]>=lst[i]) & (j<i):
            lst=lst[0:j]+[lst[i]]+[lst[j]]+
                    lst[(j+1):i]+lst[(i+1):len(lst)]
    return lst
```

Реализация сортировки вставками для демонстрации

```
def insert_sort(lst):
    compare = 0; permulation = 0
    for i in range(1,len(lst)):
        j=0
        compare += 1
        while (lst[j]<lst[i]) & (j<i):
            j  += 1
            compare += 1
        if (lst[j]>=lst[i]) & (j<i):
            lst=lst[0:j]+[lst[i]]+[lst[j]]+
                  lst[(j+1):i]+lst[(i+1):len(lst)]
            permulation += 1
        print(lst)
    return [compare,permulation]
```

Демонстрация работы

```
s = [4,5,2,7,0]
insert_sort(s)
[4, 5, 2, 7, 0]
[2, 4, 5, 7, 0]
[2, 4, 5, 7, 0]
[0, 2, 4, 5, 7]
[8, 2]
```

Теоретический анализ сортировки вставкой

- Сортировка вставкой относится к классу устойчивых алгоритмов - она сохраняет относительное положение элементов с равными значениями.
- Самым худшим случаем для сортировки вставкой является обратно отсортированный массив
- На первой итерации внешнего цикла выполняется 1 сравнение, на второй итерации выполняется 2 сравнения и т.д., на последней итерации выполняется $n-1$ сравнение:

$$(n-1)+(n-2)+(n-3)+\ldots+3+2+1=\frac{n\cdot(n-1)}{2}$$

Получаемая оценка:

$$O(n^2)$$

- Лучший случай - когда массив уже отсортирован, тогда потребуется $n-1$ операция сравнения, т.е. оценка $O(n)$

Улучшение сортировки вставкой - метод Шелла

- Заметим, что в методе вставки при небольших расстояниях элементов от своих позиций в финальном отсортированном списке, внутренний цикл, если его начать справа налево будет выполняться малое количество раз. Если расстояние от своей финальной позиции как-то ограничено, например, не превышает константы, то количество сравнений, которые могут потребоваться можно оценить $Q(n)$. Таким образом, прежде чем использовать сортировку вставками целесообразно пересортировать искомый список, чтобы каждый элемент находился недалеко справа или слева от своей финальной позиции.
- В сортировке методом Шелла стремятся выполнить такую предварительную сортировку по следующему алгоритму:
  - формируется подмножество элементов списка, как выборка каждого h-ого элемента, начиная с любой позиции в списке;
  - полученное подмножество сортируется методом вставки и возвращается в исходный список;

Метод Шелла (продолжение)

- \- значение h уменьшается и снова повторяются два предыдущих шага;
  \- как только $h = 1$ выполняется последняя сортировка всего списка методом вставки.
- Шелл в своей статье предложил использовать следующую последовательность значений $h = 1, 2, 4, 8, 16, 32\dots$ (естественно эти значения должны браться в обратном порядке, исходя из длины списка)
- В 1969 году Дональд Кнут предложил последовательность $1, 4, 13, 40, 121, \dots (a_{n+1} = 3 \cdot a_n + 1)$. Для списков средних размеров эта последовательность в среднем дает оценку $O(n^{5/4})$, а для худшего случая $O(n^{3/2})$
- Самая быстрая известная последовательность: $1, 5, 19, 41, 109, \dots$: в среднем имеем оценку $O(n^{7/6})$, в худшем случае она дает $O(n^{4/3})$. Пока неизвестно, существуют ли более быстрые последовательности.

Быстрая сортировка (описание)

- Алгоритм был разработан Хоаром в 1960 году
- Алгоритм широко используется в программировании (например, в C++ функция qsort реализована на базе быстрой сортировки)
- Для общего случая этот алгоритм относится к классу $O(n \cdot \log(n))$, однако в худшем случае быстродействие составляет $O(n^2)$, кроме этого, алгоритм неустойчив.

1. Выбирается базовый элемент списка a_1. Относительно базового элемента формируется два подсписка S_1, S_2, причем $\forall a_i \in S_1 \Rightarrow a_i < a_1$, $\forall a_i \in S_2 \Rightarrow a_i a_1.$. Ответ формируется соединением списков: $S_1 + [a_1] + S_2$.
2. Внутри каждого из списков S_1, S_2 выбирается базовый элемент, и для каждого из этих подсписков процедура пункта 1 вызывается повторно. В результате для каждого из получающихся списков процедура пункта 1 вызывается рекурсивно.

Реализация быстрой сортировки

```
def quick_sort(lst):
    if lst==[]:
        return []
    else:
        s1=[]
        s2=[]
        a=lst[0]
        for i in range(1,len(lst)):
            if lst[i]<a:
                s1=s1+[lst[i]]
            else:
                s2=s2+[lst[i]]
        return quick_sort(s1)+[a]+quick_sort(s2)
```

Реализация быстрой сортировки для демонстрации

```
compare = 0; permutation = 0
def quick_sort(lst):
    global compare,permutation
    if lst==[]:
        return []
    else:
        s1=[]
        s2=[]
        a=lst[0]
        for i in range(1,len(lst)):
            compare += 1
            if lst[i]<a:
                s1=s1+[lst[i]]
            else:
                s2=s2+[lst[i]]
        permutation += 1
        print(s1+[a]+s2)
        return quick_sort(s1)+[a]+quick_sort(s2)
```

Демонстрация работы

```
s = [4,5,2,7,0]
compare = 0; permutation = 0
quick_sort(s)
[2, 0, 4, 5, 7]
[0, 2]
[0]
[5, 7]
[7]
[0, 2, 4, 5, 7]
[compare,permutation]
[6, 5]
```

Сортировка слиянием (описание)

1. Список разбивается на две части S_1, S_2. Предполагаем, что в каждой части все элементы уже отсортированы. Стоит задача - как из уже отсортированных частей собрать полностью отсортированный список.
2. Для этого берут первый элемент из списка S_1 и первый элемент из списка S_2, наименьший из них добавляют к результату и удаляют его из соответствующего S_1, или S_2. Пункт 2 продолжают до тех пор, пока не опустеет хотя бы один из списков, после этого оставшийся непустым список добавляют к результату. Сортировка закончена.
3. Действия пункта 2 называются слиянием (merge). Однако следует вспомнить, что S_1 и S_2 сами неотсортированы, поэтому к ним применяют аналогичную сортировку слиянием (merge_sort) рекурсивно: merge(merge_sort(S_1),merge_sort(S_2)).