Поиск и сортировка

Поиск

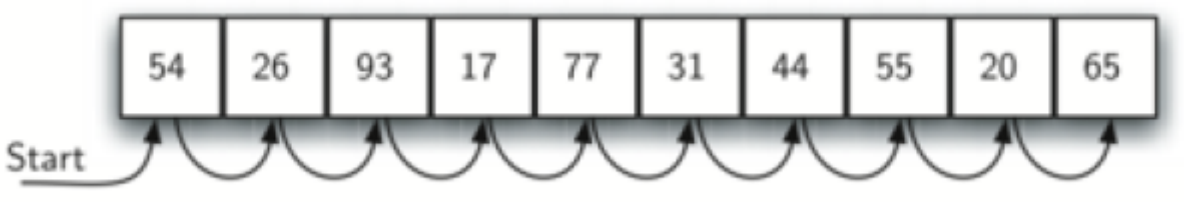
In [1]:
```
# Проверка вхождения в список:
15 in [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
```

Out[1]: False

In [2]:
```
17 in [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
```

Out[2]: True

Пример проведения посика в списке целых чисел:



In [3]:
```
def sequential_search(a_list, item):
    for current_item in a_list:
        if current_item == item:
            return True
    else:
        return False
```

In [4]:
```
def sequential_search_2(a_list, item):
    pos = 0
    found = False

    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            return True
        else:
            pos += 1

    return False
```

In [5]:
```
test_list = [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
# test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]

print(sequential_search(test_list, 3))
print(sequential_search(test_list, 17))
```

```
False
True
```

Сравнение количества сравнений, используемых при **последовательном** поиске в **несортированном** списке:

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

In [8]:
```
def ordered_sequential_search(a_list, item):
    for current_item in a_list:
        if current_item >= item:
            if current_item == item:
                return True
            else:
                return False
    else:
        return False
```

Сравнение количества сравнений, используемых при **последовательном** поиске в **отсортированном** списке:

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

In [6]:
```
ordered_test_list = sorted(test_list)
ordered_test_list
```

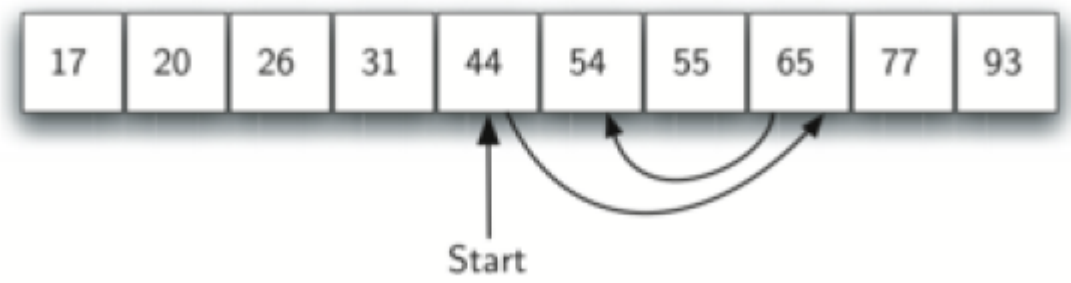
Out[6]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

In [9]:
```
print(ordered_sequential_search(ordered_test_list, 53))
print(ordered_sequential_search(ordered_test_list, 54))
```

```
False
True
```

Бинарный поиск

Пример проведения двоичного посика в отсортированном списке целых чисел:



In [10]:
```
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1

    while first <= last:
        midpoint = (first + last) // 2
        if a_list[midpoint] == item:
            return True
        else:
            if item < a_list[midpoint]:
                last = midpoint - 1
            else:
                first = midpoint + 1
    return False
```

In [11]:
```
print(binary_search(ordered_test_list, 54))
```

```
True
```

In [12]:
```
print(binary_search(ordered_test_list, 53))
```

```
False
```

Сравнение количества оставшихся для рассмотрения элементов в зависимости от количества выполненных операций сравнения:

Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

Бинарный поиск в отсортированном списке имеет сложность O(ln n).

Поиск строк

Сортировка

Формулировка задачи сортировки: имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (ключ сортировки). Тип данных ключа должен включать операции сравнения ("==","<", и производных от них ">", ">=", "<="). Требуется преобразовать исходную последовательность в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа.

Принято различать два типа сортировки:

- внутренняя сортировка, в которой предполагается, что данные находятся в оперативной памяти, и важно оптимизировать число действий программы (для методов, основанных на сравнении, число сравнений, обменов элементов и пр.)

- внешняя, в которой данные хранятся на внешнем устройстве с медленным доступом (магнитные лента, барабан, диск) и прежде всего надо снизить число обращений к этому устройству.

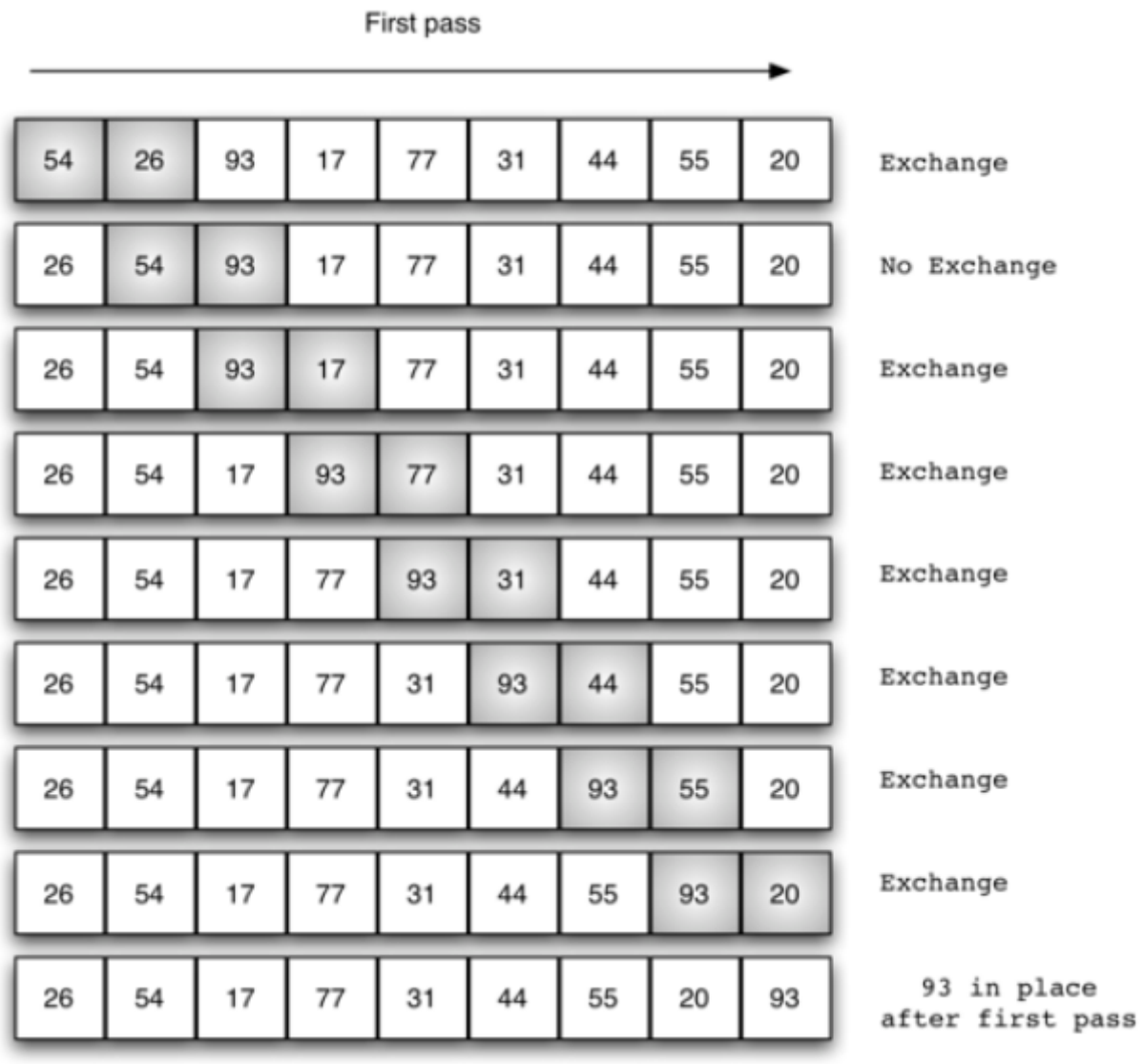
Классификация основных методов сортировки:

Простые методы сортировки

Обменные сортировки (Bubble Sort)

Алгоритм прямого обмена основывается на сравнении и смене позиций пары соседних элементов. Процесс продолжается до тех пор, пока не будут упорядочены все элементы.

Пример первого прохода алгоритма сортировки пузырьком:



In [13]:
```
# Сортировка пузырьком:
def bubble_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp
    return a_list
```

In [14]:
```
bubble_sort(list(test_list))
```

Out[14]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

In [15]:
```
# Сортировка пузырьком:
def bubble_sort_2(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        flag = False
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp
                flag = True
        if not flag:
            return a_list
```

In [16]:
```
bubble_sort_2(list(test_list))
```

Out[16]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

Алгоритм имеет среднюю и максимальную временные сложности $O(n^2)$ (два вложенных цикла, зависящих от n линейно). Введение переменной Flag и прерывание работы в случае отсортированного массива позволяет свести минимальную временную сложность к $O(n)$. Отметим одну особенность приведенного алгоритма: легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются с минимальной скоростью: один шаг за итерацию.

Сортировка выбором (извлечением) (Selection Sort)

Массив делится на уже отсортированную часть:

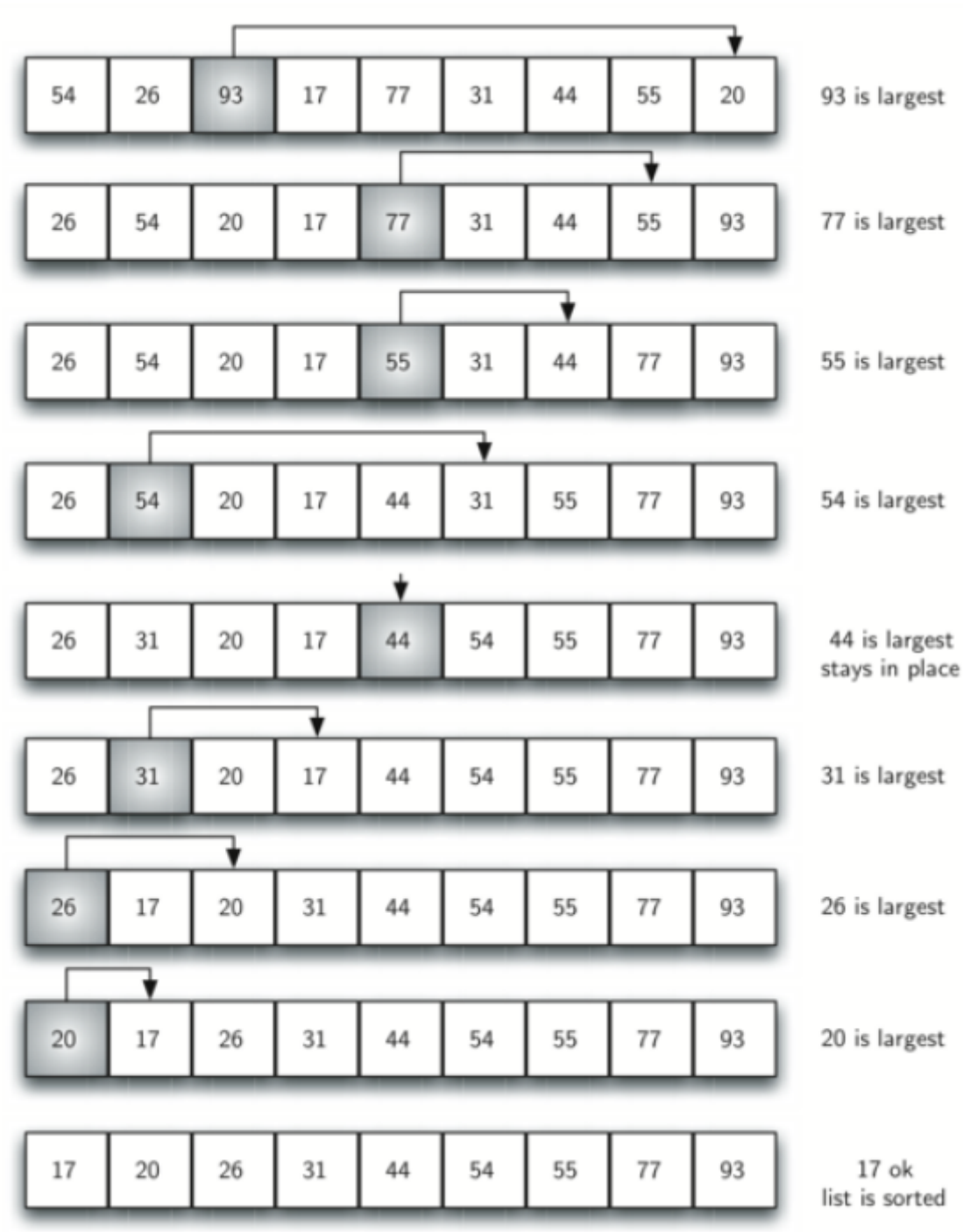
$A_{i+1}, A_{i+2} \dots \mathrm{A}_n$

и неотсортированную:

$A_1, A_2, \dots, A_i.$

На каждом шаге извлекается максимальный элемент из неотсортированной части и ставится в начало отсортированной части. Оптимизация по сравнению с сортировкой пузырьком происходит за счет выполнения только одного обмена за каждый проход через список. В результате каждого прохода находится наибольшее значение в неотсортированной части и устанавливается в корректное место отсортированного фрагмента массива.

Пример работы алгоритма сортировки выбором:



In [17]:
```
def selection_sort(a_list):
    for fill_slot in range(len(a_list) - 1, 0, -1):
        pos_of_max = 0
        for location in range(1, fill_slot + 1):
            if a_list[location] > a_list[pos_of_max]:
                pos_of_max = location

        temp = a_list[fill_slot]
        a_list[fill_slot] = a_list[pos_of_max]
        a_list[pos_of_max] = temp
        # Реализацию можно улучшить!
    return a_list
```

In [18]:
```
test_list
```

Out[18]:
```
[54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
```

In [19]:
```
selection_sort(list(test_list))
```

Out[19]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

$$(n-1)+(n-2)+(n-3)+\ldots+1 = 1/2(n-1)(n-1+1) = 1/2(n^2-n) = O(n^2).$$

По сравнению с обменной сортировкой:

- (+) сущенственно меньше перестановок элементов $O(n)$ по сравнению $O(n^2)$
- (-) нет возможности быстро отсортировать почти отсортированный массив

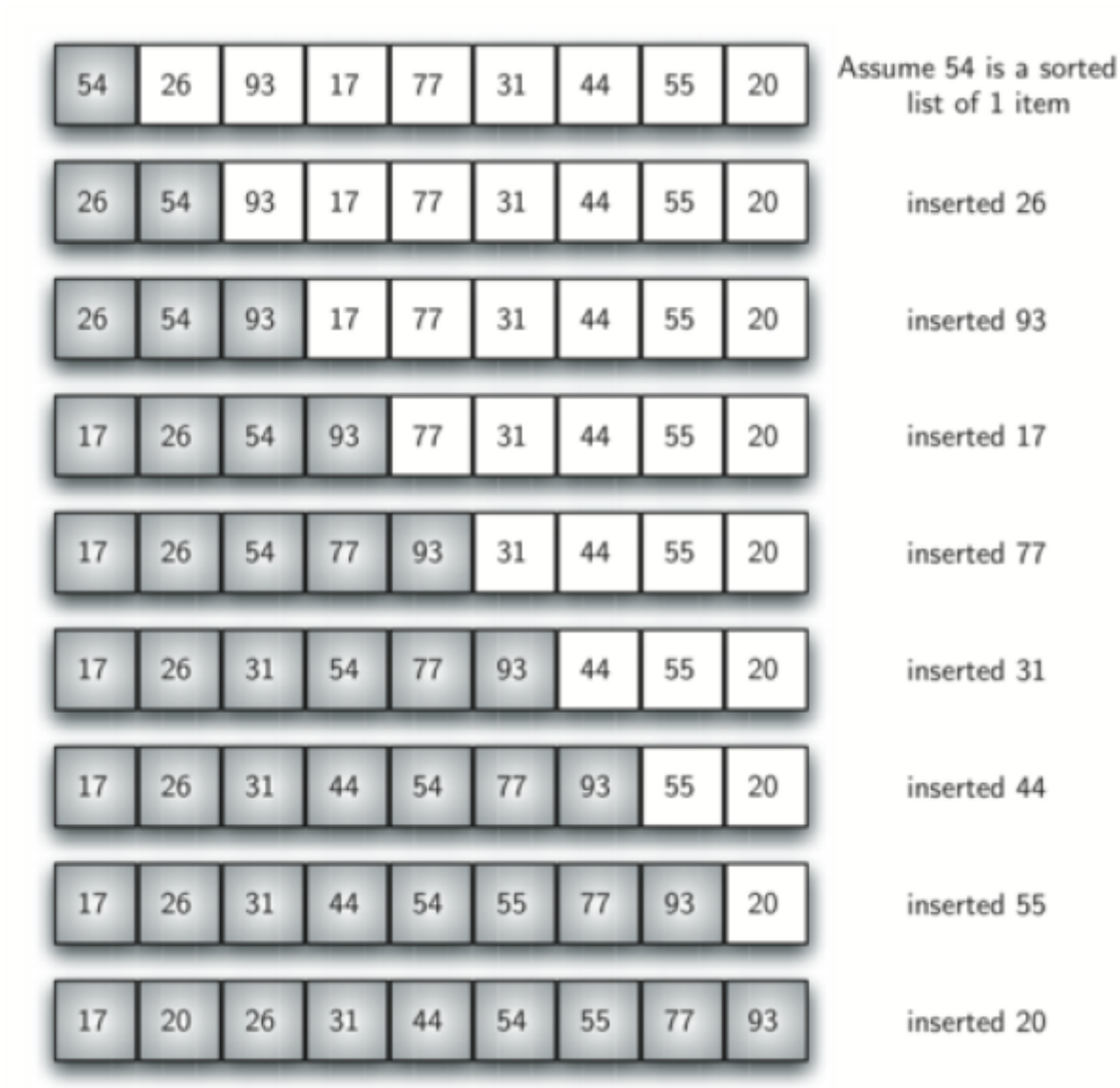
Естественной идеей улучшения алгоритма выбором является идея использования информации, полученной при сравнении элементов при поиске максимального (минимального) элемента на предыдущих шагах.

В общем случае, если n – точный квадрат, можно разделить массив на $\sqrt{n}$ групп по $\sqrt{n}$ элементов и находить максимальный элемент в каждой подгруппе. Любой выбор, кроме первого, требует не более чем $\sqrt{n-2}$ сравнений внутри группы ранее выбранного элемента плюс $\sqrt{n-1}$ сравнений среди "лидеров групп“. Этот метод получил название квадратичный выбор общее время его работы составляет порядка $O(n\sqrt{n})$ что существенно лучше, чем $O(n^2)$.

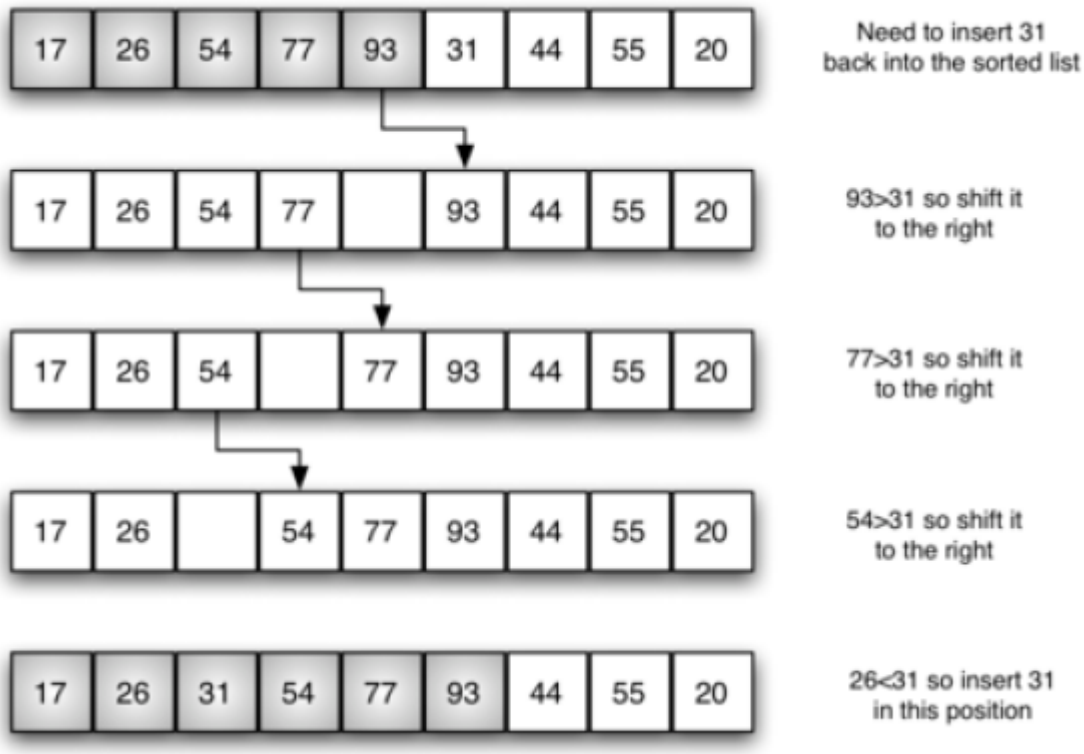
Сортировка включением (вставками) (Insertion Sort)

Массив делится на 2 части: отсортированную и неотсортированную. На каждом шаге берется очередной элемент из неотсортированной части и включается в отсортированную. Простое включение предполагает, что отсортировано начало массива $A_1, A_2, \ldots, A_{i-1}$, остаток массива $A_i, \ldots, A_n$ – неотсортирован. На очередном шаге Ai включается в отсортированную часть на соответствующее место.

Пример работы алгоритма сортировки включением:



Пример работы 5го прохода алгоритма сортировки включением:



In [20]:
```
def insertion_sort(a_list):
    for index in range(1, len(a_list)):

        current_value = a_list[index]
        position = index

        while position > 0 and a_list[position - 1] > current_value:
            a_list[position] = a_list[position - 1]
            position -= 1

        a_list[position] = current_value
    return a_list
```

In [21]:
```
test_list
```

Out[21]:
```
[54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
```

In [22]:
```
insertion_sort(list(test_list))
```

Out[22]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

Алгоритм имеет сложность $O(n^2)$, но в случае исходно отсортированного массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет в этом случае временную сложность $O(n)$.

- (+) является эффективным алгоритмом для маленьких наборов данных
- (+) на практике более эффективен чем остальные простые кавадратичные сортировки

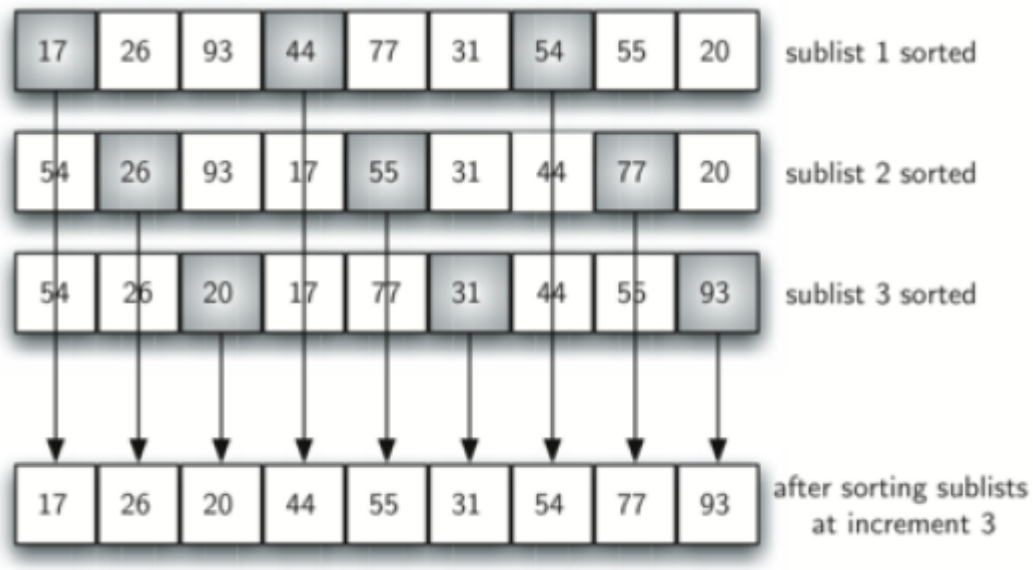
Эффективные методы сортировки

Сортировка Шелла (Shell Sort)

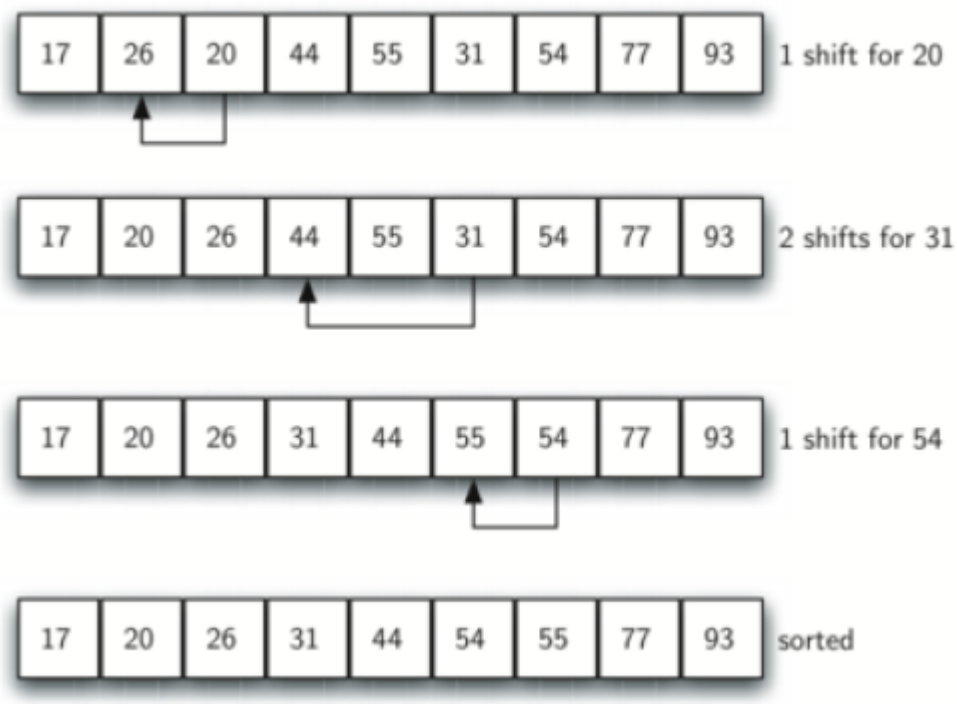
Сортировка Шелла (Donald Shell, 1959г.) является модификацией алгоритма сортировки включением, которая состоит в следующем: вместо включения A[i] в подмассив предшествующих ему элементов, его включают в подсписок, содержащий элементы A[i-h], A[i-2h], A[i-3h] и тд, где h – положительная константа. Таким образом, формируется массив, в котором «h-серии» элементов, отстоящих друг от друга на h, сортируются отдельно. Процесс возобновляется с новым значением h, меньшим предыдущего. И так до тех пор, пока не будет достигнуто значение h=1.

Пример работы сортировки Шелла (разбиение исходго массива с шагом 3):

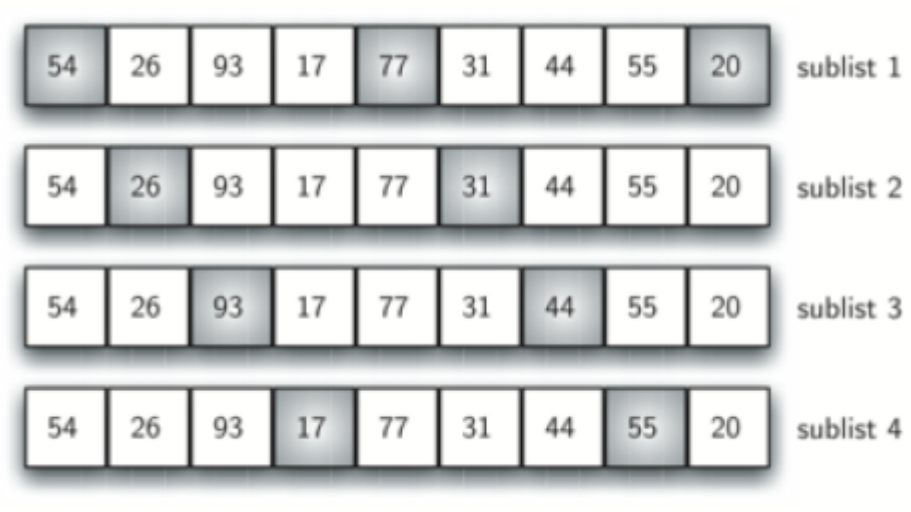
Пример работы сортировки Шелла с шагом 3 (после сортировки каждого подсписка):



Пример работы последней стадии сортировки Шелла с шагом 1 (сортировка вставкой):



Пример работы сортировки Шелла (разбиение исходго массива с шагом 3):



In [25]:
```
# Модификация сортировки вставкой для подмассива с шагом gap и смещением start:
def gap_insertion_sort(a_list, start, gap):
    for i in range(start + gap, len(a_list), gap):
        current_value = a_list[i]
        position = i

        while position >= gap and a_list[position - gap] > current_value:
            a_list[position] = a_list[position - gap]
            position = position - gap

        a_list[position] = current_value
```

In [28]:
```
def shell_sort(a_list):
    sublist_count = len(a_list) // 2
    while sublist_count > 0:
        for start_position in range(sublist_count):
            gap_insertion_sort(a_list, start_position, sublist_count)

        print("After inc. of size", sublist_count, "Lst:", a_list)

        sublist_count = sublist_count // 2
```

In [30]:
```
shell_sort(list(test_list))
```

```
After inc. of size 5 Lst: [31, 26, 55, 17, 65, 54, 44, 93, 20, 77]
After inc. of size 2 Lst: [20, 17, 31, 26, 44, 54, 55, 77, 65, 93]
After inc. of size 1 Lst: [17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

Для достаточно больших массивов рекомендуемой считается такая последовательность, что

$$h_{i+1} = 3h_i + 1, h_1 = 1$$

.

получается последовательность : 1, 4, 13, 40, 121…(hi+1=3hi+1) .

Начинается процесс с hm-2, такого, что:

$$h_{m-2} \geq [n/9]$$

.

Временная сложность для алгоритма Шелла – $O(n^{4/3})$ и $\Theta(n^{7/6})$, среднее число перемещений ~ $1,66n^{1,25}$.

Количество перестановок элементов по результатам экспериментов со случайным массивом иллюстрируется следующей таблицей.

Размерность	n = 25	n = 1000	n = 100000
Сортировка Шелла	50	7700	2 100 000
Сортировка простыми вставками	150	240 000	2.5 млрд.

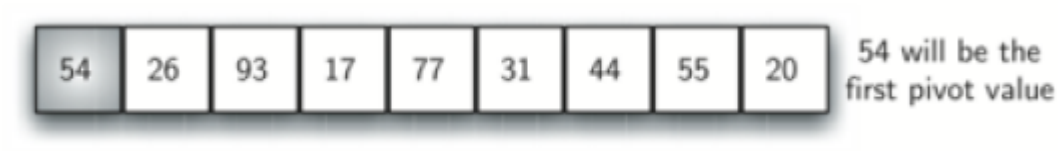
Быстрая сортировка (Quick Sort)

Быстрая сортировка – это алгоритм сортировки, время работы которого для входного массива из n чисел в наихудшем случае равно $O(n^2)$. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм на практике зачастую оказывается оптимальным благодаря тому, что в среднем время его работы намного лучше: $O(n \ln n)$. Кроме того, постоянные множители, не учтенные в выражении $O(n \ln n)$, достаточно малы по величине. Алгоритм обладает также тем преимуществом, что сортировка в нем выполняется без использования дополнительной памяти, поэтому он хорошо работает даже в средах с виртуальной памятью.

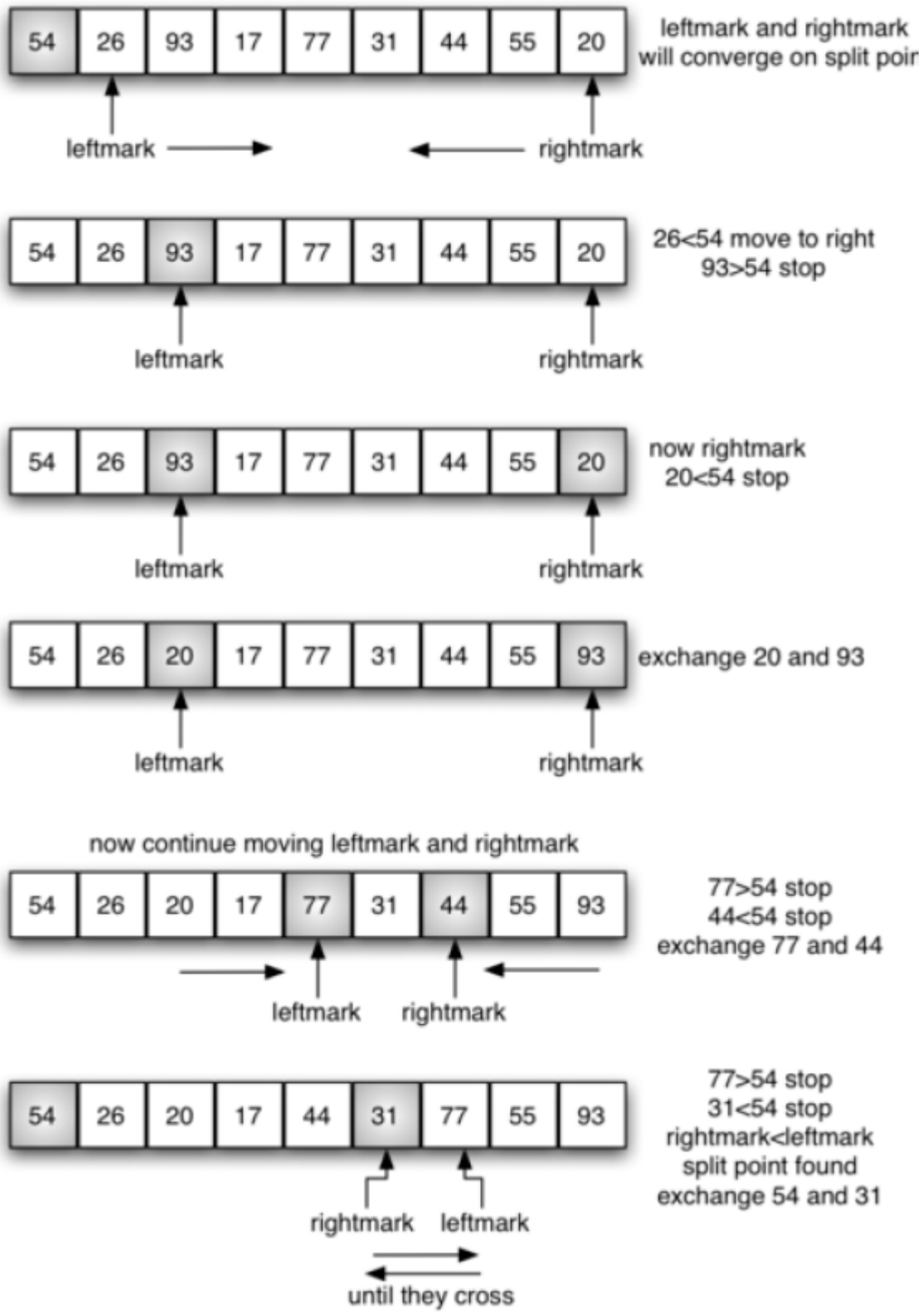
Алгоритм быстрой сортировки является реализацией парадигмы «разделяй и властвуй». Разделение исходного массива осуществляется по следующему принципу:

1. Выбрать наугад какой-либо элемент массива – x
2. Просмотреть массив слева направо, пока не обнаружим элемент Ai > x
3. Просмотреть массив справа налево, пока не встретим Ai < x
4. Поменять местами эти два элемента
5. Процесс просмотра и обмена продолжается, пока указатели обоих просмотров не встретятся

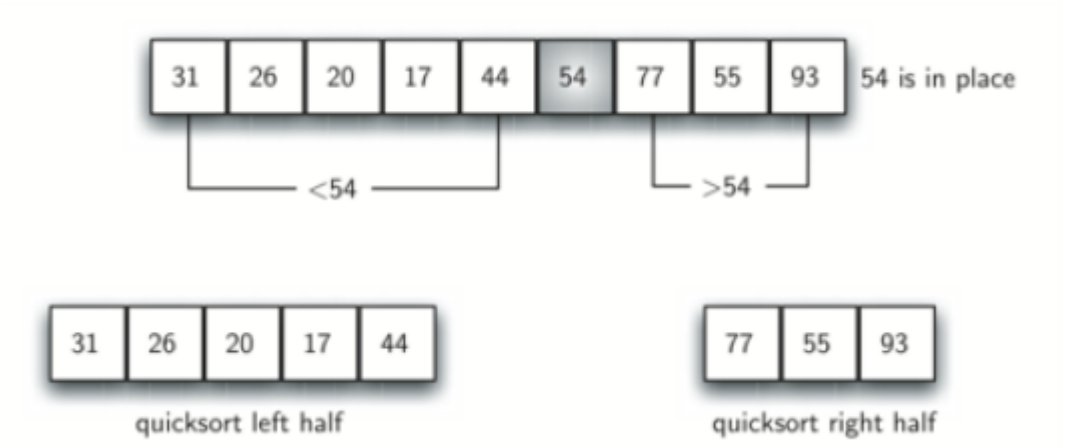
Пример работы быстрой сортировки (выбор элемента для разделения массива):



Пример работы быстрой сортировки (разделения массива на две части: со значениями меньшими и большими чем у разделяющего элемента):



Пример работы быстрой сортировки (подготовка к рекурсивному вызову сортировки двух подмассивов):



In [31]:
```
def quick_sort(a_list):
    quick_sort_helper(a_list, 0, len(a_list) - 1)
    return a_list

def quick_sort_helper(a_list, first, last):
    if first < last:

        split_point = partition(a_list, first, last)

        quick_sort_helper(a_list, first, split_point - 1)
        quick_sort_helper(a_list, split_point + 1, last)

def partition(a_list, first, last):
    pivot_value = a_list[first]

    left_mark = first + 1
    right_mark = last

    done = False
    while not done:

        while left_mark <= right_mark and a_list[left_mark] <= pivot_value:
            left_mark = left_mark + 1

        while a_list[right_mark] >= pivot_value and right_mark >= left_mark:
            right_mark = right_mark - 1

        if right_mark < left_mark:
            done = True
        else:
            temp = a_list[left_mark]
            a_list[left_mark] = a_list[right_mark]
            a_list[right_mark] = temp

    temp = a_list[first]
    a_list[first] = a_list[right_mark]
    a_list[right_mark] = temp

    return right_mark
```

In [32]:
```
quick_sort(list(test_list))
```

Out[32]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

Ожидаемое число обменов в быстром алгоритме – $(n-1)/6$, общее число сравнений $n \ln n$. Наихудший случай – в качестве элемента для разбиения x выбирается наибольшее из всех значений в указанной области, т.е. левая часть состоит из $n-1$ элементов, а правая из 1, тогда временная сложность становится пропорциональна n^2.

Сортировка слиянием (Merge Sort)

Многие полезные алгоритмы имеют рекурсивную структуру: для решения данной задачи они рекурсивно вызывают сами себя один или несколько раз, чтобы решить вспомогательную задачу, имеющую непосредственное отношение к поставленной задаче. Такие алгоритмы зачастую разрабатываются с помощью метода декомпозиции, или разбиения: сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем; далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получить решение исходной задачи.

Парадигма, лежащая в основе метода декомпозиции «разделяй и властвуй», на каждом уровне рекурсии включает в себя три этапа:

1. Разделение задачи на несколько подзадач.
2. Покорение – рекурсивное решение этих подзадач. Когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно.
3. Комбинирование решения исходной задачи из решений вспомогательных задач.

Алгоритм сортировки слиянием (merge sort) в большой степени соответствует парадигме метода разбиения. На интуитивном уровне его работу можно описать таким образом.

Разделение: сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит n/2 элементов.

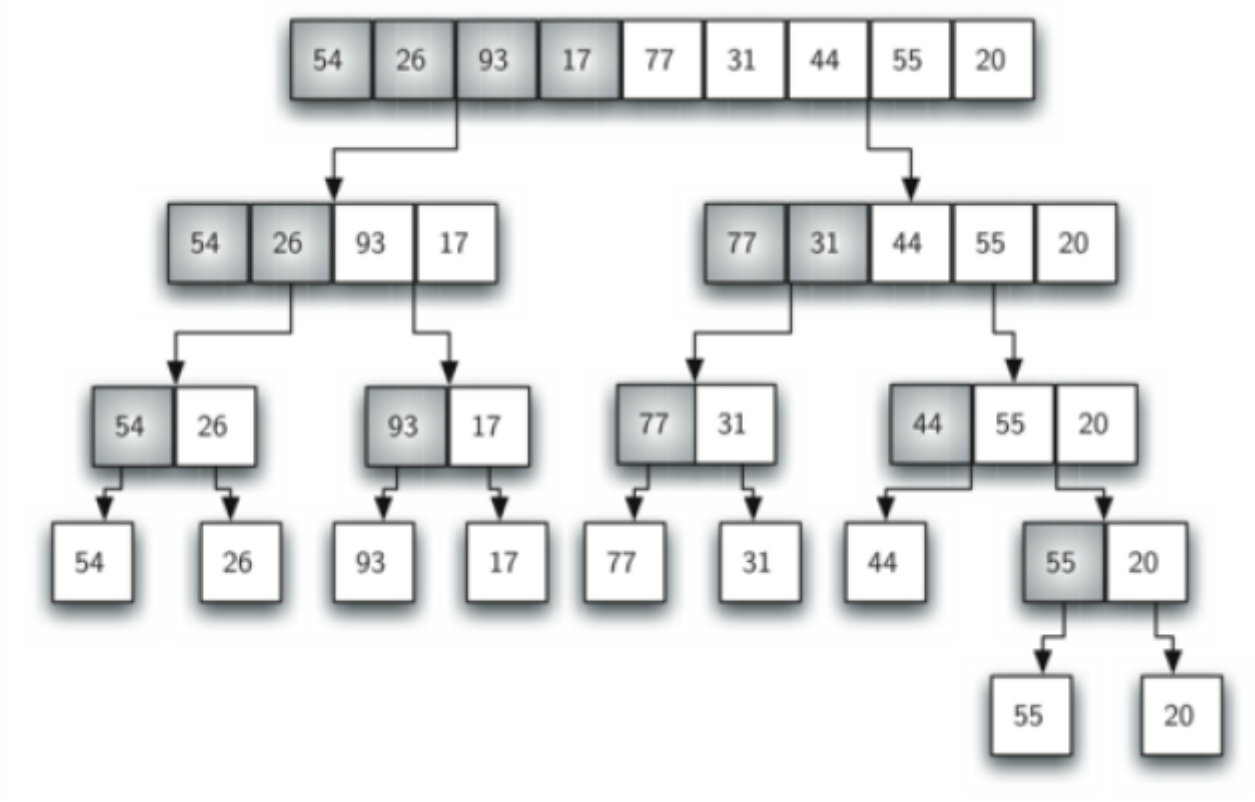
Покорение: сортировка обеих вспомогательных последовательностей методом слияния.

Комбинирование: слияние двух отсортированных последовательностей для получения окончательного результата.

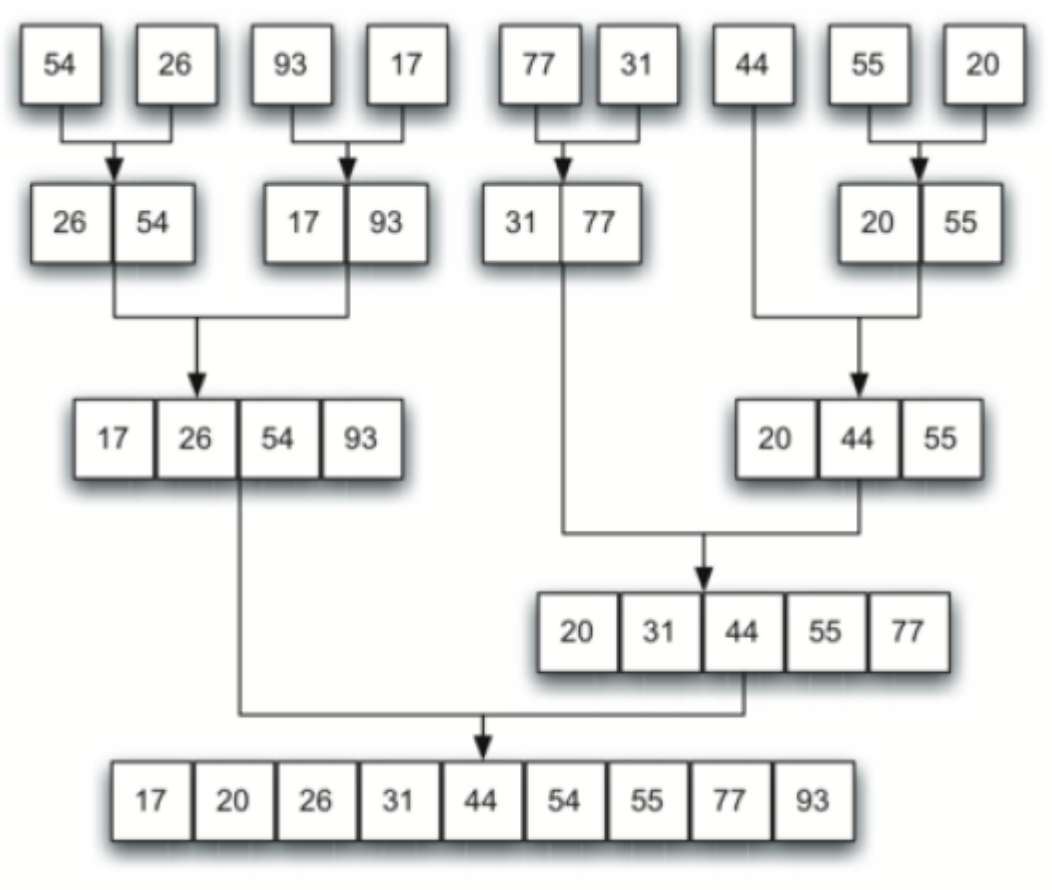
Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной. Основная операция, которая производится в процессе сортировки по методу слияний, – это объединение двух отсортированных последовательностей в ходе комбинирования (последний этап). Это делается с помощью вспомогательной процедуры слияния. В этой процедуре предполагается, что элементы подмассивов упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы. Для выполнения этой процедуры требуется время в O(n), где n – количество подлежащих слиянию элементов.

Временная сложность алгоритма сортировки слиянием можно определить как O(n ln n).

Разбиение исходного массива в сортировке слиянием:



Слияние разбитого массива в сортировке слиянием:



In [35]:
```
def merge_sort(a_list):
    print("Splitting ", a_list)
    if len(a_list) > 1:
        mid = len(a_list) // 2
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = 0
        j = 0
        k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                a_list[k] = left_half[i]
                i += 1
            else:
                a_list[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            a_list[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            a_list[k] = right_half[j]
            j += 1
            k += 1

        print("Merging ", a_list)
    return a_list
```

In [36]:
```
merge_sort(list(test_list))
```

```
Splitting  [54, 26, 93, 17, 77, 31, 44, 55, 20, 65]
Splitting  [54, 26, 93, 17, 77]
Splitting  [54, 26]
Splitting  [54]
Splitting  [26]
Merging  [26, 54]
Splitting  [93, 17, 77]
Splitting  [93]
Splitting  [17, 77]
Splitting  [17]
Splitting  [77]
Merging  [17, 77]
Merging  [17, 77, 93]
Merging  [17, 26, 54, 77, 93]
Splitting  [31, 44, 55, 20, 65]
Splitting  [31, 44]
Splitting  [31]
Splitting  [44]
Merging  [31, 44]
```

Итоги

Сравнение сортировок:

http://javarevisited.blogspot.ru/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html (http://javarevisited.blogspot.ru/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html)

http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html (http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html)

https://en.wikipedia.org/wiki/Sorting_algorithm (https://en.wikipedia.org/wiki/Sorting_algorithm)

Сортировка в Python

In [65]:
```
colors = ['red', 'green', 'blue', 'cyan', 'magenta', 'yellow']
```

In [66]:
```
colors
```

Out[66]:
```
['red', 'green', 'blue', 'cyan', 'magenta', 'yellow']
```

In [69]:
```
# Встроенный метод для list (сортирует на месте):
colors.sort()
# This method has the effect of reordering the elements of the list into order,
# as defined by the natural meaning of the < operator for those elements
```

In [68]:
```
colors
```

Out[68]:
```
['blue', 'cyan', 'green', 'magenta', 'red', 'yellow']
```

In [70]:
```
colors = ['red', 'green', 'blue', 'cyan', 'magenta', 'yellow']
```

In [71]:
```
sorted(colors) # сортирует в новом списке
```

Out[71]:
```
['blue', 'cyan', 'green', 'magenta', 'red', 'yellow']
```

In [72]:
```
colors
```

Out[72]:
```
['red', 'green', 'blue', 'cyan', 'magenta', 'yellow']
```

In [73]:
```
# задание функции для определения ключа для сортировки:
sorted(colors, key=len)
```

Out[73]:
```
['red', 'blue', 'cyan', 'green', 'yellow', 'magenta']
```

https://docs.python.org/3/howto/sorting.html#sortinghowto (https://docs.python.org/3/howto/sorting.html#sortinghowto)

Эффективный поиск в отсортированном списке

Библиотека bisect: https://docs.python.org/3/library/bisect.html#module-bisect (https://docs.python.org/3/library/bisect.html#module-bisect)

In [92]:
```
from bisect import bisect_left

def binary_search(a, x, lo=0, hi=None):  # can't use a to specify default for hi
    hi = hi if hi is not None else len(a)  # hi defaults to len(a)
    pos = bisect_left(a, x, lo, hi)  # find insertion position
    return (pos if pos != hi and a[pos] == x else None)  # don't walk off the end
```

In [93]:
```
test_list_s = sorted(test_list)
test_list_s
```

Out[93]:
```
[17, 20, 26, 31, 44, 54, 55, 65, 77, 93]
```

In [94]:
```
binary_search(test_list_s, 44)
```

Out[94]:
```
4
```

In [97]:
```
print(binary_search(test_list_s, 45))
```

```
None
```

In [ ]:
```

```