Sorting Algorithms with Python 3: All You Need to Know

Li Yin[1]

November 30, 2019

[1]www.liyinscience.com

Sorting is the most basic building block for many other algorithms and is often considered as the very first step that eases and reduces the original problems to easier ones.

0.1 Introduction

Sorting In computer science, a *sorting algorithm* is designed to rearrange items of a given array in a certain order based on each item's *key*. The most frequently used orders are *numerical order* and *lexicographical order*. For example, given an array of size n, sort items in increasing order of its numerical values:

```
Array = [9, 10, 2, 8, 9, 3, 7]
sorted = [2, 3, 7, 8, 9, 9, 10]
```

Selection *Selection algorithm* is used to find the k-th smallest number in a given array; such a number is called the k-th *order statistic*. For example, given the above array, find the 3-th smallest number.

```
Array = [9, 10, 2, 8, 9, 3, 7], k = 3
Result: 7
```

Sorting and Selection often go hand in hand; either we first execute sorting and then select the desired order through indexing or we derive a selection algorithm from a corresponding sorting algorithm. Due to such relation, this chapter is mainly about introducing sorting algorithms and occasionally we introduce their corresponding selection algorithms by the side.

Lexicographical Order For a list of strings, sorting them will make them in lexicographical order. The order is decided by a comparison function, which compares corresponding characters of the two strings from left to right. In the process, the first pair of characters that differ from each other determines the ordering: the string that has smaller alphabet from the pair is smaller than the other string.

Characters are compared using the Unicode character set. All uppercase letters come before lower case letters. If two letters are the same case, then alphabetic order is used to compare them. For example:

```
'ab' < 'bc' (differs at i = 0)
'abc' < 'abd' (differs at i = 2)
```

Special cases appears when two strings are of different length and the shorter one s is a prefix of the the longer one t, then it is considered that $s < t$. For example:

```
'ab' < 'abab' ('ab' is a prefix of 'abab')
```

How to Learn Sorting Algorithms? We list a few terminologies that are commonly seen to describe the properties of a certain sorting algorithm:

- **In-place Sorting**: In-place sorting algorithm only uses a constant number of extra spaces to assist its implementation. If a sorting algorithm is not in-place, it is called out-of-place sorting instead.
- **Stable Sorting**: Stable sorting algorithm maintain the relative order of items with equal keys. For example, two different tasks that come with same priority in the priority queue should be scheduled in the relative pending ordering.
- **Comparison-based Sorting**: This kind of sorting technique determines the sorted order of an input array by comparing pairs of items and moving them around based on the results of comparison. And it has a lower bound of $\Omega(n \log n)$ comparison.

Sorting Algorithms in Coding Interviews As the fundamental Sorting and selection algorithms can still be potentially met in interviews where we might be asked to implement and analyze any sorting algorithm you like. Therefore, it is necessary for us to understand the most commonly known sorting algorithms. Also, Python provides us built-in sorting algorithms to use directly and we shall mater the syntax too.

The Applications of Sorting The importance of sorting techniques is decided by its multiple fields of application:

1. Sorting can organize information in a human-friendly way. For example, the lexicographical order are used in dictionary and inside of library systems to help users locate wanted words or books in a quick way.
2. Sorting algorithms often be used as a key subroutine to other algorithms. As we have shown before, binary search, sliding window algorithms, or cyclic shifts of suffix array need the data to be in sorted order to carry on the next step. When ordering will not incur wrong solution to the problems, sorting beforehand should always be atop on our mind for sorting first might ease our problem later.

Organization We organize the content mainly based on the worst case time complexity. Section 0.3 - 0.4 focuses on comparison-based sorting algorithms, and Section 0.5.2-**??** introduce classical non-comparison-based sorting algorithms.

- Naive Sorting (Section 0.3): Bubble Sort, Insertion Sort, Selection Sort;

- Asymptotically Best Sorting (Section 0.4) Sorting: merge sort, quick sort, and Quick Select;
- Linear Sorting (Section 0.5.2): Counting Sort, where k is the range of the very first and last key.
- Python Built-in Sort (Section 0.6):

0.2 Python Comparison Operators and Built-in Functions

Comparison Operators Python offers 7 comparison operators shown in Table. 1 to compare values. It either returns `True` or `False` according to the condition.

Table 1: Comparison operators in Python

>	Greater than - `True` if left operand is greater than the right
<	Less that - `True` if left operand is less than the right
==	Equal to - `True` if both operands are equal
!=	Not equal to - `True` if operands are not equal
>=	Greater than or equal to - `True` if left operand is greater than or equal to the right
<=	Less than or equal to - True if left operand is less than or equal to the right

For example, compare two numerical values:

```
c1 = 2 < 3
c2 = 2.5 > 3
```

The printout is:

```
(True, False)
```

Also, compare two strings follows the lexicographical orders:

```
c1 = 'ab' < 'bc'
c2 = 'abc' > 'abd'
c3 = 'ab' < 'abab'
c4 = 'abc' != 'abc'
```

The printout is:

```
(True, False, True, False)
```

What's more, it can compare other types of sequences such as `list` and `tuple` using lexicographical orders too:

```
c1 = [1, 2, 3] < [2, 3]
c2 = (1, 2) > (1, 2, 3)
c3 = [1, 2] == [1, 2]
```

The printout is:

```
(True, False, True)
```

However, mostly Python 3 does not support comparison between different types of sequence, nor does it supports comparison for `dictionary`. For `dictionary` data structures, in default, it uses its key as the key to compare with. For example, comparison between `list` and `tuple` will raise `TypeError`:

```
[1, 2, 3] < (2, 3)
```

The error is shown as:

```
----> 1 [1, 2, 3] < (2, 3)
TypeError: '<' not supported between instances of 'list' and '
    tuple'
```

Comparison between dictionary as follows will raise the same error:

```
{1: 'a', 2:'b'} < {1: 'a', 2:'b', 3:'c'}
```

Comparison Functions Python built-in functions `max()` and `min()` support two forms of syntax: `max(iterable, *[, key, default])` and `max(arg1, arg2, *args[, key])`. If one positional argument is provided, it should be an iterable. And then it returns the largest item in the iterable based on its key. It also accepts two or more positional arguments, and these arguments can be numerical or sequential. When there are two or more positional argument, the function returns the largest.

For example, with one iterable and it returns 20:

```
max([4, 8, 9, 20, 3])
```

With two positional arguments –either numerical or sequential:

```
m1 = max(24, 15)
m2 = max([4, 8, 9, 20, 3], [6, 2, 8])
m3 = max('abc', 'ba')
```

The printout of these results is:

```
(24, [6, 2, 8], 'ba')
```

With `dictionary`:

```
dict1 = {'a': 5, 'b': 8, 'c': 3}
k1 = max(dict1)
k2 = max(dict1, key=dict1.get)
k3 = max(dict1, key =lambda x: dict1[x])
```

The printout is:

```
('c', 'b', 'b')
```

When the sequence is empty, we need to set an default value:

```
max([], default=0)
```

Rich Comparison To compare a self-defined `class`, in Python 2.X, `__cmp__(self, other)` special method is used to implement comparison between two objects. `__cmp__(self, other)` returns negative value if `self < other`, positive if `self > other`, and zero if they were equal. However, in Python 3, this `cmp` style of comparisons is dropped, and `rich comparison` is introduced, which assign a special method to each operator as shown in Table. 2: To avoid the hassle of providing all six functions, we can only implement `__eq__`, `__ne__`, and only one of the ordering operators, and use the `functools.total_ordering()` decorator to fill in the rest. For example, write a class `Person`:

Table 2: Operator and its special method

==	`__eq__`
!=	`__ne__`
<	`__lt__`
<=	`__le__`
>	`__gt__`
>=	`__ge__`

```python
from functools import total_ordering
@total_ordering
class Person(object):
    def __init__(self, firstname, lastname):
        self.first = firstname
        self.last = lastname

    def __eq__(self, other):
        return ((self.last, self.first) == (other.last, other.
    first))

    def __ne__(self, other):
        return not (self == other)

    def __lt__(self, other):
        return ((self.last, self.first) < (other.last, other.
    first))

    def __repr__(self):
        return "%s %s" % (self.first, self.last)
```

Then, we would be able to use any of the above comparison operator on our class:

```python
p1 = Person('Li', 'Yin')
p2 = Person('Bella', 'Smith')
p1 > p2
```

It outputs `True` because last name “Yin” is larger than “Smith”.

0.3 Naive Sorting

As the most naive and intuitive group of comparison-based sorting methods, this group takes $O(n^2)$ time and usually consists of two nested for loops. In this section, we learn three different sorting algorithms "quickly" due to their simplicity: insertion sort, bubble sort,and selection sort.

0.3.1 Insertion Sort

Insertion sort is one of the most intuitive sorting algorithms for humans. For humans, given an array of n items to process, we divide it into two regions: **sorted and unrestricted region**. Each time we take one item "out" of the unrestricted region to sorted region by inserting it at a proper position.

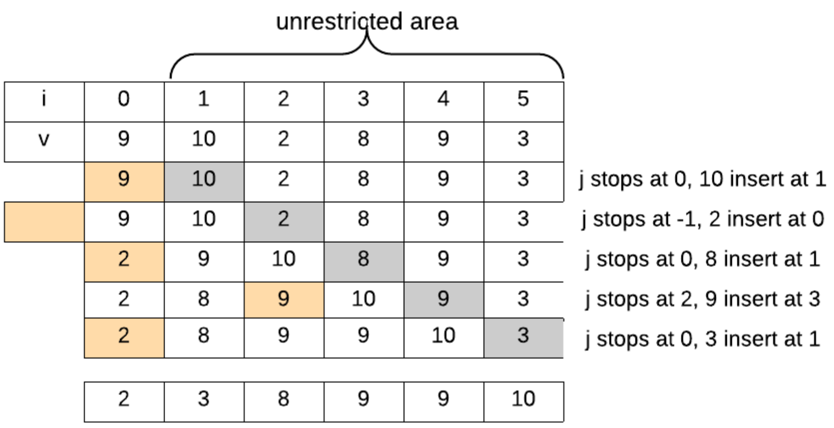

unrestricted area
i
v
j stops at 0, 10 insert at 1
j stops at -1, 2 insert at 0
j stops at 0, 8 insert at 1
j stops at 2, 9 insert at 3
j stops at 0, 3 insert at 1

Figure 1: The whole process for insertion sort: Gray marks the item to be processed, and yellow marks the position after which the gray item is to be inserted into the sorted region.

In-place Insertion The logic behind this algorithm is simple, we can do it easily by setting up another sorted array. However, here we want to focus on the in-place insertion. Given array of size n, we use index 0 and i to point to the start position of sorted and the unrestricted region, respectively. And $i = 1$ at the beginning, indicates that the sorted region will naturely has one item. We have sorted region in $[0, i-1]$, and the unrestricted region in $[i, n-1]$. We scan item in the unrestricted region from left to right, and insert each item $a[i]$ into the sorted sublist.

The key step is to find a proper position of $a[i]$ in the region $[0, i-1]$ to insert into. There are two different ways for iteration over unsorted region: forward and backward. We use pointer j in the sorted region.

- Forward: j will iterate in range $[0, i-1]$. We compare $a[j]$ with $a[i]$, and stop at the first place that $a[j] > a[i]$ (to keep it stable). All items elements $a[j : i-1]$ will be shifted backward for one position, and $a[i]$ will be placed at index j. Here we need i times of comparison and swaps.

- Backward: j iterates in range $[i-1, 0]$. We compare $a[j]$ with $a[i]$, and stop at the first place that $a[j] <= a[i]$ (to keep it stable). In this process, we can do the shifting simultaneously: if $a[j] > a[i]$, we shift $a[j]$ with $a[j+1]$.

In forward, the shifting process still requires us to reverse the range, therefore the backward iteration makes better sense.

For example, given an array $a = [9, 10, 2, 8, 9, 3]$. First, 9 itself is sorted array. we demonstrate the backward iteration process. At first, 10 is compared with 9, and it stays at where it is. At the second pass, 2 is compared with 10, 9, and then it is put at the first position. The whole whole process of this example is demonstrated in Fig. 1.

With Extra Space Implementation The Python `list.insert()` function handles the insert and shifting at the same time. We need to pay attention when the item is larger than all items in the sorted list, we have to insert it at the end.

```python
def insertionSort(a):
  if not a or len(a) == 1:
    return a
  n = len(a)
  sl = [a[0]] # sorted list
  for i in range(1, n):
    for j in range(i):
      if sl[j] > a[i]:
        sl.insert(j, a[i])
        break
    if len(sl) != i + 1: # not inserted yet
      sl.insert(i, a[i])
  return sl
```

Backward In-place Implementation We use a `while` loop to handle the backward iteration: whenever the target is smaller than the item in the sorted region, we shift the item backward. When the `while` loop stops, it is either $j = -1$ or when $t >= a[j]$.

- When $j = -1$, that means we need to insert the target at the first position which should be $j + 1$.

- When $t >= a[j]$, we need to insert the target one position behind j, which is $j + 1$.

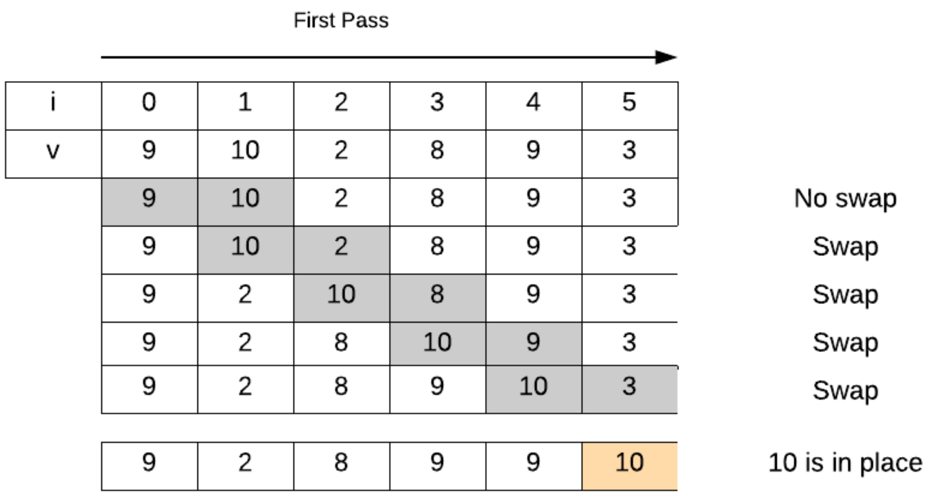
The code is shown as:

```python
def insertionSort(a):
  if not a or len(a) == 1:
    return a
  n = len(a)
  for i in range(1, n):
    t = a[i]
    j = i - 1
    while j >= 0 and t < a[j]:
      a[j+1] = a[j] # Move item backward
      j -= 1
    a[j+1] = t
  return
```

0.3.2 Bubble Sort and Selection Sort

Bubble Sort

Bubble sort compares each pair of adjacent items in an array and swaps them if they are out of order. Given an array of size n: in a single pass, there are $n-1$ pairs for comparison, and at the end of the pass, one item will be put in place.



i	0	1	2	3	4	5	
v	9	10	2	8	9	3	
	9	10	2	8	9	3	No swap
	9	10	2	8	9	3	Swap
	9	2	10	8	9	3	Swap
	9	2	8	10	9	3	Swap
	9	2	8	9	10	3	Swap
	9	2	8	9	9	10	10 is in place

Figure 2: One pass for bubble sort

Passes For example, Fig. 2 shows the first pass for sorting array [9, 10, 2, 8, 9, 3]. When comparing a pair (a_i, a_{i+1}), if $a_i > a_{i+1}$, we swap these two items. We can clearly see after one pass, the largest item 10 is in place. For the next pass, it only compare pairs within the unrestricted window $[0, 4]$. This is what"bubble" means in the name: after a pass, the largest item in the unrestricted window bubble up to the end of the window and become in place.

Implementation With the understanding of the valid window of each pass, we can implement "bubble" sort with two nested `for` loops in Python. The first `for` loop to enumerate the number of passes, say i, which is $n-1$ in total. The second `for` loop to is to scan pairs in the unrestricted window $[0, n-i-1]$ from left to right. thus index j points to the first item in the pair, making it in range of $[0, n-i-2]$.

```python
def bubbleSort(a):
  if not a or len(a) == 1:
    return
  n = len(a)
  for i in range(n - 1): #n-1 passes
    for j in range(n - i -1):
      # Swap
      if a[j] > a[j + 1]:
        a[j], a[j + 1] = a[j + 1], a[j]
  return
```

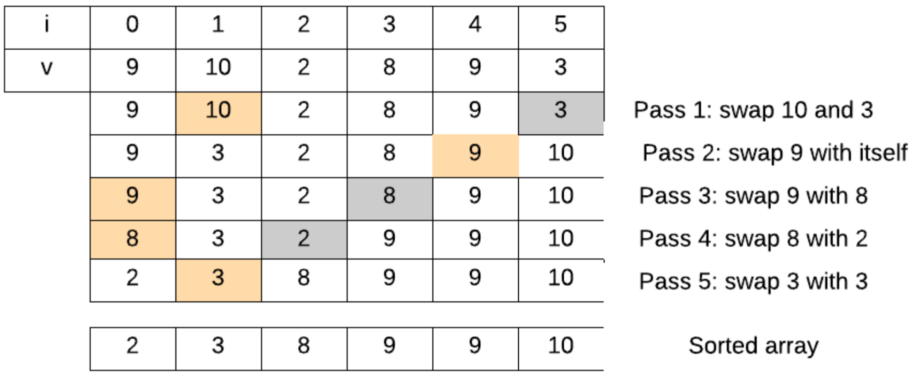
When the pair has equal values, we do not need to swap them. The advantage of doing so is (1) to save unnecessary swaps and (2) keep the original order of items with same keys. This makes bubble sort a **stable sort**. Also, in the implementation no extra space is assigned either which makes bubble sort **in-place sort**.

Complexity Analysis and Optimization In i-th pass, the item number in the valid window is $n-i$ with $n-i-1$ maximum of comparison and swap, and we need a total of $n-1$ passes. The total time will be $T = \sum_{i=0}^{n-i}(n-i-1) = n-1+(n-2)+...+2+1 = n(n-1)/2 = O(n^2)$. The above implementation runs $O(n^2)$ even if the array is sorted. We can optimize the inner `for` loop by stopping the whole program if no swap is detected in a single pass. When the input is nearly sorted, this strategy can get us $O(n)$ time complexity.

Selection Sort

In the bubble sort, each pass we get the largest element in the valid window in place by a series of swapping operations. While, selection sort makes a slight optimization via searching for the largest item in the current unrestricted window and swap it directly with the last item in the region. This avoids the constant swaps as occurred in the bubble sort. The whole sorting process for the same array is shown in Fig 3.

Implementation Similar to the implementation of Bubble Sort, we have the concept of number of passes at the outer `for` loop, and the concept



i	0	1	2	3	4	5	
v	9	10	2	8	9	3	
	9	10	2	8	9	3	Pass 1: swap 10 and 3
	9	3	2	8	9	10	Pass 2: swap 9 with itself
	9	3	2	8	9	10	Pass 3: swap 9 with 8
	8	3	2	9	9	10	Pass 4: swap 8 with 2
	2	3	8	9	9	10	Pass 5: swap 3 with 3
	2	3	8	9	9	10	Sorted array

Figure 3: The whole process for Selection sort

of unrestricted at the inner `for` loop. We use variables `ti` and `li` for the position of the largest item to be and being, respectively.

```python
def selectSort(a):
  n = len(a)
  for i in range(n - 1): #n-1 passes
    ti = n - 1 - i
    li = 0 # The index of the largest item
    for j in range(n - i):
      if a[j] >= a[li]:
        li = j
    # swap li and ti
    a[ti], a[li] = a[li], a[ti]
  return
```

Like bubble sort, selection sort is **in-place**. In the comparison, we used `if a[j] >= a[li]:`, which is able to keep the relative order of equal keys. For example, in our example, there is equal key 9. Therefore, selection sort is stable sort too.

Complexity Analysis Same as of bubble sort, selection sort has a worst and average time complexity of $O(n^2)$ but more efficient when the input is not as near as sorted.

0.4 Asymptotically Best Sorting

We have learned a few comparison-based sorting algorithms and they all have an upper bound of n^2 in time complexity due to the number of comparisons must be executed. Can we do better than $O(n^2)$ and how?

Comparison-based Lower Bounds for Sorting Given an input of size n, there are $n!$ different possible permutations on the input, indicating that our sorting algorithms must find the one and only one permutation by comparing pairs of items. So, how many times of comparison do we need to reach to the answer? Let's try the case when $n = 3$, and all possible permutations using the indexes will be: $(1, 2, 3), (1, 3, 2), (3, 1, 2), (2, 1, 3), (2, 3, 1), (3, 2, 1)$. First we compare pair $(1, 2)$, if $a_1 < a_2$, our candidates set is thus narrowed down to $\{(1, 2, 3), (1, 3, 2), (3, 1, 2)\}$.

We draw a decision-tree, which is a full binary tree with $n!$ leaves–the $n!$ permutations, and each branch represents one decision made on the comparison result. The cost of any comparison-based algorithm is abstracted as the length of the path from the root of the decision tree to its final sorted permutation. The longest path represents the worst-case number of comparisons.

Using h to denote the height of the binary tree, and l for the number of leaves. First, a binary tree will have at most 2^h leaves, we get $l \leq 2^h$. Second, it will have at least $n!$ leaves to represent all possible orderings, we have $l \geq n!$. Therefore we get the lower bound time complexity for the worst case:

$$
\begin{align}
n! \leq l \leq 2^h \tag{1}\\
2^h \geq n! \tag{2}\\
h \geq \log(n!) \tag{3}\\
h = \Omega(n \log n) \tag{4}
\end{align}
$$

In this section, we will introduce three classical sorting algorithms that has $O(n \log n)$ time complexity: Merge Sort and Quick Sort both utilize the Divide-and-conquer method, and Heap Sort uses the max/min heap data structures.

0.4.1 Merge Sort

As we know there are two main steps: "divide" and "merge" in merge sort and we have already seen the illustration of the "divide" process in Chapter. **??**.

Divide In the divide stage, the original problem $a[s...e]$, where s, e is the start and end index of the subarray, respectively. The divide process divides its parent problem into two halves from the middle index $m = (s + e)//2$: $a[s...m]$, and $a[m + 1, e]$. This recursive call keeps moving downward till the size of the subproblem becomes one when $s = e$, which is the base case for a list of size 1 is naturally sorted. The process of divide is shown in Fig. 4.

Merge When we obtained two sorted sublists from the left and right side, the result of current subproblem is to merge the two sorted list into one. The merge process is done through two pointer method: We assign a new list and

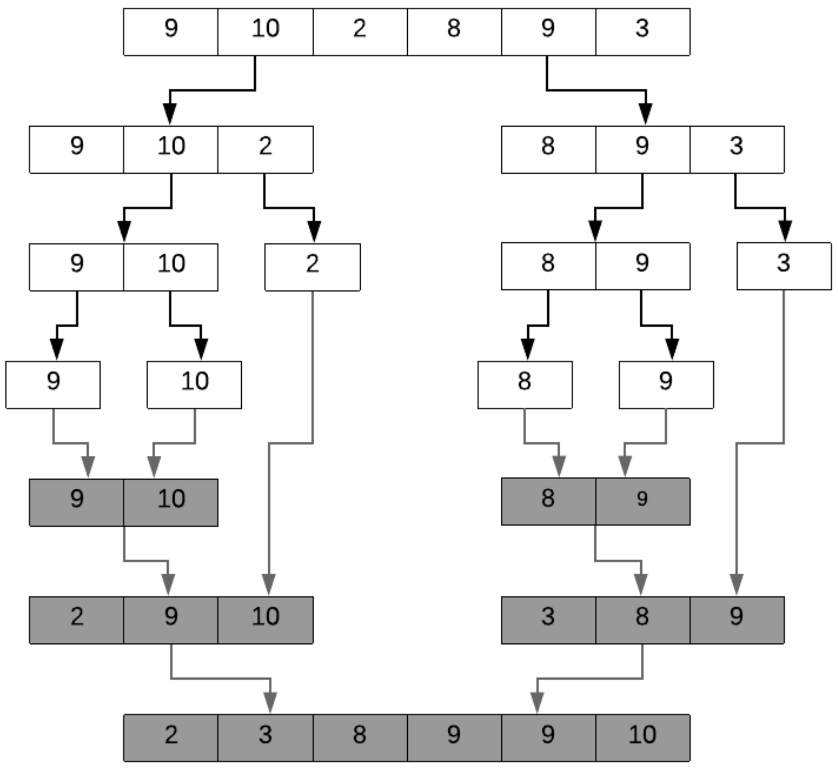


Figure 4: Merge Sort: The dividing process is marked with dark arrows and the merging process is with gray arrows with the merge list marked in gray color too.

put two pointers at the start of the two sublists, and each time we choose the smaller item to append into the new list between the items indicated by the two pointers. Once a smaller item is chosen, we move its corresponding pointer to the next item in that sublist. We continue this process until any pointer reaches to the end. Then, the sublist where the pointer does not reach to the end yet is coped to the end of the new generated list. The subprocess is shown in Fig. 4 and its implementation is as follows:

```python
def merge(l, r):
  ans = []
  # Two pointers each points at l and r
  i = j = 0
  n, m = len(l), len(r)

  while i < n and j < m:
    if l[i] <= r[j]:
      ans.append(l[i])
      i += 1
    else:
      ans.append(r[j])
      j += 1

  ans += l[i:]
  ans += r[j:]
  return ans
```

In the code, we use $l[i] <= r[j]$ instead of $l[i] < r[j]$ is because when the left and right sublist contains items of equal keys, we put the ones in the left first in the merged list, so that the sorting can be **stable**. However, we used a temporary space as $O(n)$ to save the merged result a, making merge sort an **out-of-place** sorting algorithm.

Implementation The whole implementation is straightforward.

```
def mergeSort(a, s, e):
  if s == e:
    return [a[s]]

  m = (s + e) // 2

  l = mergeSort(a, s , m)
  r = mergeSort(a, m+1, e)
  return merge(l, r)
```

Complexity Analysis Because for each divide process we need to take $O(n)$ time to merge the two sublists back to a list, the recurrent relation of the complexity function can be deducted as follows:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= 2 * 2T(n/4) + O(n) + O(n) \\ &= O(n \log n) \end{aligned} \tag{5}$$

Thus, we get $O(n \log n)$ as the upper bound for merge sort, which is asymptotically optimal within the comparison-based sorting.

0.4.2 HeapSort

To sort the given array in increasing order, we can use min-heap. We first `heapify` the given array. To get a sorted list, we can simply pop out items till the heap is empty. And the popped out items will be in sorted order.

Implementation We can implement heap sort easily with built-in module `heapq` through the `heapify()` and `heappop()` functions :

```
from heapq import heapify, heappop
def heapsort(a):
  heapify(a)
  return [heappop(a) for i in range(len(a))]
```

Complexity Analysis The `heapify` takes $O(n)$, and the later process takes $O(\log n + \log(n-1) + ... + 0) = \log(n!)$ which has an upper bound of $O(n \log n)$.

0.4.3 Quick Sort and Quick Select

Like merge sort, quick sort applies divide and conquer method and is mainly implemented with recursion. Unlike merge sort, the conquering step the sorting process- *partition* happens before "dividing" the problem into subproblems through recursive calls.

Partition and Pivot In the partition, quick sort chooses a *pivot* item from the subarray, either randomly or intentionally. Given a subarray of $A[s, e]$, the pivot can either be located at s or e, or a random position in range $[s, e]$. Then it partitions the subarray $A[s, e]$ into three parts according to the value of the pivot: $A[s, p-1], A[p]$, and $A[p+1...e]$, where p is where the pivot is placed at. The left and right part of the pivot satisfies the following conditions:

- $A[i] \leq A[p], i \in [s, p-1]$,
- and $A[i] > A[p], i \in [p+1, e]$.

If we are allowed with linear space, this partition process will be trivial to implement. However, we should strive for better and learn an in-place partition methods–*Lomuto's* Partition, which only uses constant space.

Conquer After the partition, one item–the pivot $A[p]$ is placed in the right place. Next, we only need to handle two subproblems: sorting $A[s, p-1]$ and $A[p+1, e]$ by recursively call the quicksort function. We can write down the main steps of quick sort as:

```python
def quickSort(a, s, e):
  # Base case
  if s >= e:
    return
  p = partition(a, s, e)

  # Conquer
  quickSort(a, s , p-1)
  quickSort(a, p+1, e)
  return
```

At the next two subsection, we will talk about partition algorithm. And the requirement for this step is to do it **in-place** just through a series of swapping operations.

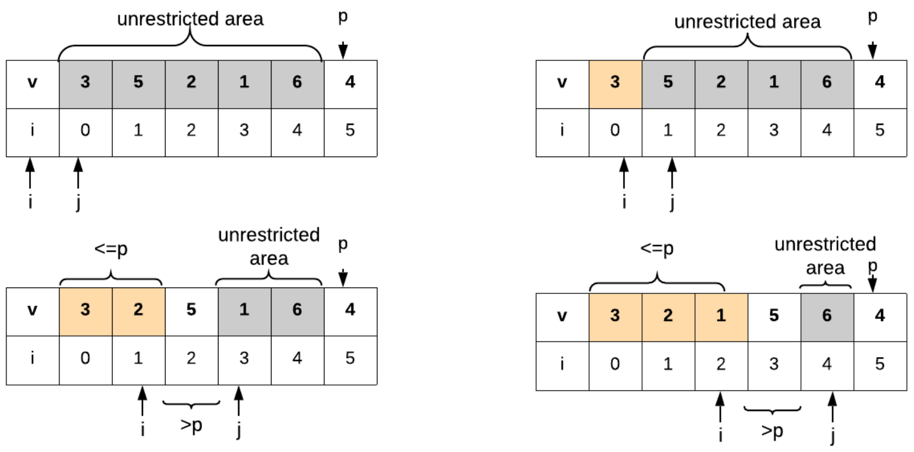


Figure 5: Lomuto's Partition. Yellow, while, and gray marks as region (1), (2) and (3), respectively.

Lomuto's Partition

We use example $A = [3, 5, 2, 1, 6, 4]$ to demonstrate this partition method. Assume our given range for partition is $[s, e]$, and $p = A[e]$ is chosen as pivot. We would use two pointer technique i, j to maintain three regions in subarray $A[s, e]$: (1) region $[s, i]$ with items smaller than or equal to p; (2) $[i+1, j-1]$ region with item larger than p; (3) unrestricted region $[j, e-1]$. These three areas and the partition process on the example is shown in Fig. 5.

- At first, $i = s - 1, j = s$. Such initialization guarantees that region (1) and (2) are both empty, and region (3) is the full range other than the pivot.
- Then, scan scan items in the unrestricted area using pointer j:
  - If the current item $A[j]$ belongs to region (2), that is to say $A[j] > p$, we just increment pointer j;
  - Otherwise when $A[j] <= p$, this item should goes to region (1). We accomplish this by swapping this item with the first item in region (2) at $i + 1$. And now region (1) increments by one and region (2) shifts one position backward.
- After the for loop, we need to put our pivot at the first place of region (2) by swapping. And now, the whole subarray is successfully parititioned into three regions as we needed, and return where the index of where the pivot is at–$i + 1$–as the partition index.

We The implementation of as follows:

```
def partition(a, s, e):
  p = a[e]
  i = s - 1
  # Scan unresticted area
  for j in range(s, e):
    # Swap
    if a[j] <= p:
      i += 1
      a[i], a[j] = a[j], a[i]
  a[i+1], a[e] = a[e], a[i+1]
  return i+1
```

Complexity Analysis The worst case of the partition appears when the input array is already sorted or is reversed from the sorted array. In this case, it will partition a problem with size n into one subproblem with size $n-1$ and the other subproblem is just empty. The recurrence function is $T(n) = T(n-1)+O(n)$, and it has a time complexity of $O(n^2)$. And the best case appears when a subprocess is divided into half and half as in the merge sort, where the time complexity is $O(n \log n)$. Randomly picking the pivot from $A[s, e]$ and swap it with $A[e]$ can help us achieve a stable performance with average $O(n \log n)$ time complexity.

Stablity of Quick Sort

Quick sort is *not stable*, because there are cases items can be swapped no matter what: (1) as the first item in the region (2), it can be swapped to the end of region (2). (2) as the pivot, it is swapped with the first item in the region (2) too. Therefore, it is hard to guarantee the stability among equal keys. We can try experiment with $A = [(2, 1), (2, 2), (1, 1)]$, and use the first element in the tuple as key. This will sort it as $A = [(1, 1), (2, 2), (2, 1)]$.

However, we can still make quick sort stable if we get rid of the swaps by using two extra lists: one for saving the smaller and equivalent items and the other for the larger items.

Quick Select

Quick Select is a variant of quick sort, and it is used to find the k-th smallest item in a list in linear time. In quicksort, it recurs both sides of the partition index, while in quick select, only the side that contains the k-th smallest item will be recurred. This is similar to the binary search, the comparison of k and partition index p results three cases:

- If $p = k$, we find the k-th smallest item, return.
- If $p > k$, then we recur on the right side.
- If $p < k$, then we recur on the left side.

Based on the structure, quick select has the following recurrence time complexity function:

$$T(n) = T(n/2) + O(n) \tag{6}$$

$$T(n) = O(n) \texttt{ (with master theorem)} \tag{7}$$

Implementation We first set k in range of $[s, e]$. When $s = e$, there is only one item in the list, which means we no longer can divide it. This is our end condition and is also the case when our original list has only one item, then we have to return this item as the 0-th smallest item.

```python
def quickSelect(a, s, e, k, partition=partition):
  if s >= e:
    return a[s]

  p = partition(a, s, e)
  if p == k:
    return a[p]
  if k > p:
    return quickSelect(a, p+1, e, k, partition)
  else:
    return quickSelect(a, s, p-1, k, partition)
```

0.5 Linear Sorting

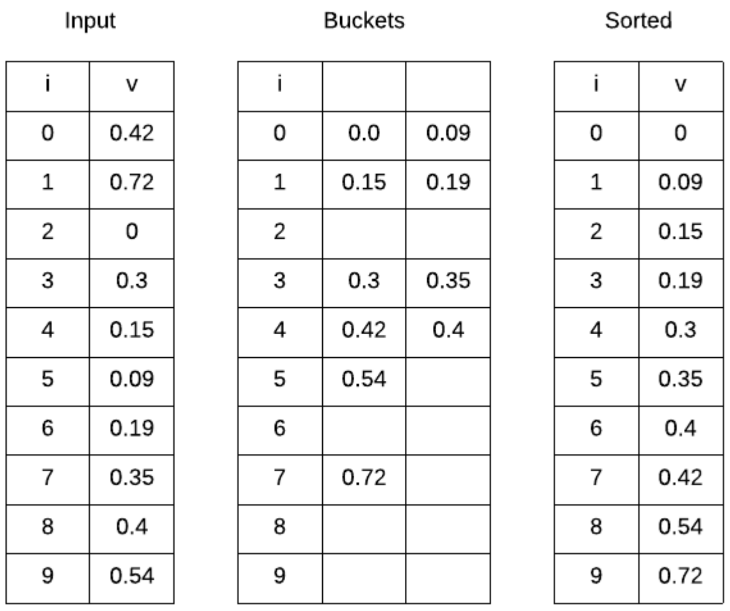
Sorting without basing upon comparisons is possible, creative, and even faster, proved by the three non-comparative sorting algorithms we are about to introduce: Bucket Sort, Counting Sort, and Radix Sort. For these algorithms the theoretic lower bound $O(n \log n)$ of comparison-based sorting is not likewise a lower bound any more; they all work in linear time. However, there are limitations to the input data, as these sorting techniques rely on certain assumptions concerning the data to be sorted to be able to work.

Although the three algorithms we see in this section come in different forms and rely on different assumptions to the input data, we see one thing in common: They all use the divide and conquer algorithm design paradigm. Let's explore their unique tricks and the restrictive applications!

0.5.1 Bucket Sort

Bucket Sort assumes that the input data satisfying a uniform distribution. The uniform distribution is usually assumed to be in interval $[0, 1)$. However, it can be extended to any uniform distribution with simple modification. Bucket sort applies a one time divide and conquer trick–it divides the input data into n independent segments, n the size of the input, just as what we have seen in merge sort, and then insertion sort is applied on each segment, and finally each sorted segmented is combined to get the result.

Bucket sort manages the dividing process by assigning n empty **buckets**, and then distribute the input data $a[i]$ to bucket index `int(a[i]*n)`. For example, if $n = 10$, and $a[i] = 0.15$, the bucket that the number goes to is the one with index 1. We use example a = [0.42, 0.72, 0. , 0.3 , 0.15, 0.09, 0.19, 0.35, 0.4 , 0.54], and visualize the process in Fig. 6.



Input

i	v
0	0.42
1	0.72
2	0
3	0.3
4	0.15
5	0.09
6	0.19
7	0.35
8	0.4
9	0.54

Buckets

i		
0	0.0	0.09
1	0.15	0.19
2		
3	0.3	0.35
4	0.42	0.4
5	0.54	
6		
7	0.72	
8		
9		

Sorted

i	v
0	0
1	0.09
2	0.15
3	0.19
4	0.3
5	0.35
6	0.4
7	0.42
8	0.54
9	0.72

Figure 6: Bucket Sort

Implementation First, we prepare the input data with `random.uniform` from `numpy` library. For simplicity and the reconstruction of the same input, we used random seed and rounded the float number to only two decimals.

```
import numpy as np
np.random.seed(1)
a = np.random.uniform(0,  1, 10)
a = np.round(a, decimals=2)
```

Now, the code for the bucket sort is straightforward as:

```
from functools import reduce
def bucketSort(a):
  n = len(a)
  buckets = [[] for _ in range(n)]
  # Divide numbers into buckets
  for v in a:
    buckets[int(v*n)].append(v)
  # Apply insertion sort within each bucket
  for i in range(n):
    insertionSort(buckets[i])
```

```
# Combine sorted buckets
return reduce(lambda a, b: a + b, buckets)
```

Complexity Analysis

Extension To extend to uniform distribution in any range, we first find the minimum and maximum value, $minV, maxV$, and compute the bucket index i for number $a[i]$ with formula:

$$i = n\frac{a[i] - minV}{maxV - minV} \tag{8}$$

0.5.2 Counting Sort

Counting sort is an algorithm that sorts items according to their corresponding keys that are small integers. It works by counting the occurrences of each distinct key value, and using arithmetic–prefix sum–on those counts to determine the position of each key value in the sorted sequence. Counting sort no longer fits into the comparison-based sorting paradigm because it uses the keys as indexing to assist the sorting instead of comparing them directly to decide relative positions. For input that comes with size n and the difference between the maximum and minimum integer keys k, counting sort has a time complexity $O(n + k)$.

Premise: Prefix Sum

Before we introduce counting sort, first let us see what is prefix sum. Prefix sum, a.k.a cumulative sum, inclusive scan, or simply scan of a sequence of numbers $x_i, i \in [0, n-1]$ is second sequence of numbers $y_i, i \in [0, n-1]$, and y_i is the sums of prefixes of the input sequence, with equation:

$$y_i = \sum_{j=0}^{i} x_j \tag{9}$$

For instance, the prefix sums of on the following array is:

```
Index: 0  1  2  3  4  5
x:     1  2  3  4  5  6
y:     1  3  6  10 15 21
```

Prefix sums are trivial to compute with the following simple recurrence relation in $O(n)$ complexity.

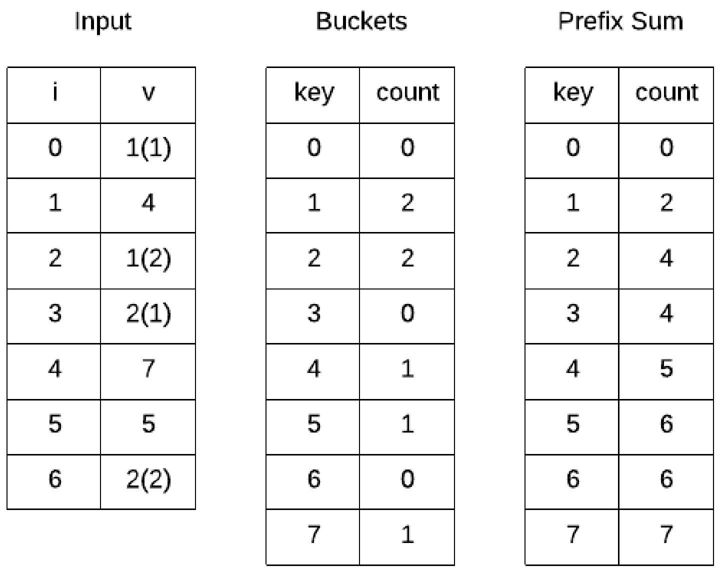
$$y_i = y_{i-1} + x_i, i \geq 1 \tag{10}$$

Despite the ease of computation, prefix sum is a useful primitive in certain algorithms such as counting sort and Kadane's Algorithm as you shall see through this book.

Counting Sort

Given an input array $[1, 4, 1, 2, 7, 5, 2]$, let's see how exactly counting sort works by explaining it in three steps. Because our input array comes with duplicates, we distinguish the duplicates by their relative order shown in the parenthesises. Ideally, for this input, we want it to be sorted as:

```
Index: 0      1     2      3     4  5  6
Key:    1(1)  4    1(2)   2(1)   7  5  2(2)
Sorted:1(1)  1(2) 2(1)   2(2)   4  5  7
```



Input

i	v
0	1(1)
1	4
2	1(2)
3	2(1)
4	7
5	5
6	2(2)

Buckets

key	count
0	0
1	2
2	2
3	0
4	1
5	1
6	0
7	1

Prefix Sum

key	count
0	0
1	2
2	4
3	4
4	5
5	6
6	6
7	7

Figure 7: Counting Sort: The process of counting occurrence and compute the prefix sum.

1. **Count Occurrences:** We assign a `count` array C_i, and assign a size 8, which has index in range $[0, 7]$ and will be able to contain our keys whose range is $[1, 7]$. which has the same size of the key range k. We loop over each key in the input array, and use key as index to count each key's occurrence. Doing so will get the following result. And it means in the input array, we have two 1's, two 2's, one 4, one 5, and one 7. The process is shown in Fig. 7.

   Counting sort is indeed a subtype of bucket sort, where the number of buckets is k, and each bucket stores keys implicitly by using keys as indexes and the occurrence to track the total number of the same keys.

2. **Prefix Sum on Count Array:** We compute the prefix sum for `count` array, which is shown as:

```
Index:          0  1  2  3  4  5  6  7
Count:          0  2  2  0  1  1  0  1
Prefix Sum:     0  2  4  4  5  6  6  7
```

Denote the prefix sum array as ps. For key i, ps_{i-1} tells us the number of items that is less or equals to ($\leq$) key i. This information can be used to place key i directly into its correct position. For example, for key 2, summing over its previous keys' occurrences (ps_1) gives us 2, indicating that we can put key 2 to position 2. However, key 2 appears two times, and the last position of key 2 is indicated by $ps_2 - 1$, which is 3. Therefore, for any key i, its locations in the sorted array is in range $[ps_{i-1}, ps_i)$. We could have just scan the prefix sum array, and use the prefix sum as locations for key indicated by index of prefix sum array. However, this method is only limited to situations where the input array is integers. Moreover, it is unable to keep the relative ordering of the items of the same key.

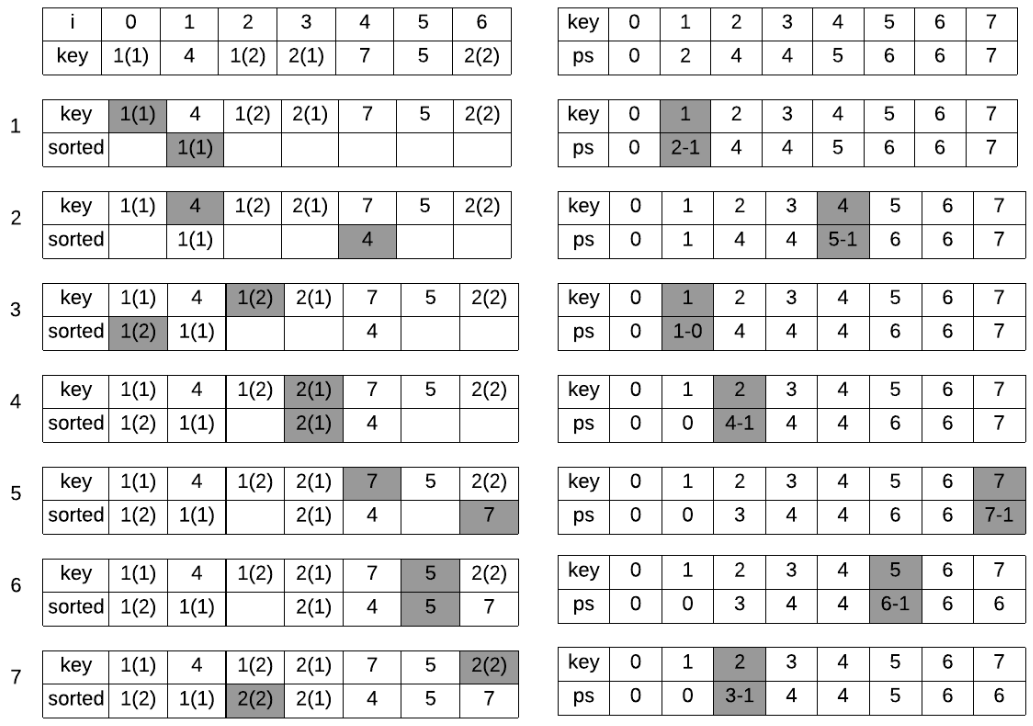
3. **Sort Keys with Prefix Sum Array:** First, let us loop over the input keys from position 0 to $n-1$. For key_i, we decrease the prefix sum by one, $ps_{key_i} = ps_{key_i} - 1$ to get the last position that we can assign this key in the sorted array. The whole process is shown in Fig. 8. We saw that items of same keys are sorted in reverse order. Looping over keys in the input in reverse order is able to correct this and thus making the counting sort a stable sorting algorithm.

Implementation In our implementation, we first find the range of the input data, say it is $[minK, maxK]$, making our range $k = maxK - minK$. And we recast the key as $key - minK$ for two purposes:

- To save space for `count` array.
- To be able to handle negative keys.

The implementation of the main three steps are nearly the same as what we have discussed other than the recast of the key. In the process, we used two auxiliary arrays: `count` array for counting and accumulating the occurrence of keys with $O(k)$ space and `order` array for storing the sorted array with $O(n)$ space, giving us the space complexity $O(n+k)$ in our implementation. The Python code is shown as:

```python
def countSort(a):
  minK, maxK = min(a), max(a)
  k = maxK - minK + 1
  count = [0] * (maxK - minK + 1)
  n = len(a)
  order = [0] * n
  # Get occurrence
  for key in a:
```



i	0	1	2	3	4	5	6
key	1(1)	4	1(2)	2(1)	7	5	2(2)

key	0	1	2	3	4	5	6	7
ps	0	2	4	4	5	6	6	7

1

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted		1(1)					

key	0	1	2	3	4	5	6	7
ps	0	2-1	4	4	5	6	6	7

2

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted		1(1)			4		

key	0	1	2	3	4	5	6	7
ps	0	1	4	4	5-1	6	6	7

3

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted	1(2)	1(1)			4		

key	0	1	2	3	4	5	6	7
ps	0	1-0	4	4	4	6	6	7

4

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted	1(2)	1(1)		2(1)	4		

key	0	1	2	3	4	5	6	7
ps	0	0	4-1	4	4	6	6	7

5

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted	1(2)	1(1)		2(1)	4		7

key	0	1	2	3	4	5	6	7
ps	0	0	3	4	4	6	6	7-1

6

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted	1(2)	1(1)		2(1)	4	5	7

key	0	1	2	3	4	5	6	7
ps	0	0	3	4	4	6-1	6	6

7

key	1(1)	4	1(2)	2(1)	7	5	2(2)
sorted	1(2)	1(1)	2(2)	2(1)	4	5	7

key	0	1	2	3	4	5	6	7
ps	0	0	3-1	4	4	5	6	6

Figure 8: Counting sort: Sort keys according to prefix sum.

```
        count[key - minK] += 1

    # Get prefix sum
    for i in range(1, k):
        count[i] += count[i-1]

    # Put key in position
    for i in range(n-1, -1, -1):
        key = a[i] - minK
        count[key] -= 1 # to get the index as position
        order[count[key]] = a[i]
    return order
```

Properties Counting sort is **out-of-place** for the auxiliary `count` and `order` array. Counting sort is **stable** given that we iterate keys in the input array in reversed order. Counting sort is likely to have $O(n + k)$ for both the space and time complexity.

Applications Due to the special character that counting sort sorts by using key as index, and the range of keys decides the time and space complexity, counting sort's applications are limited. We list the most common applications:

- Because the time complexity depends on the size of k, in practice counting sort is usually used when $k = O(n)$, in which case it makes the time complexity $O(n)$.
- Counting sort is often used as a sub-routine. For example, it is a part of other sorting algorithms such as radix sort, which is a linear sorting algorithm. We will also see some examples in string matching chapter.

0.5.3 Radix Sort

The word "Radix" is a mathematical term for the *base* of a number. For example, decimal and hexadecimal number has a radix of 10 and 16, respectively. For strings of alphabets has a radix of 26 given there are 26 letters of alphabet. Radix sort is a non-comparative sorting methods that utilize the concept of radix or base to order a list of integers digit by digit or a list of strings letter by letter. The sorting of integers or strings of alphabets is different based on the different concepts of ordering–number ordering and the lexicographical order as we have introduced. We show one example for list of integers and strings and their sorted order or lexicographical order:

```
Integers: 170, 45, 75, 90, 802, 24
Sorted:    24, 45, 75, 90, 170, 802
```

```
Strings: apple, pear, berry, peach, apricot
Sorted:  apple, apricot, berry, peach, pear
```

And we see how that the integers are ordered by the length of digits, whereas in the sorted strings, the length of strings does not usually decide the ordering.

Within Radix sorting, it is usually either the bucket sort or counting sort that is doing the sorting using one radix as key at a time. Based upon the sorting order of the digit, we have two types of radix sorting: *Most Significant Digit (MSD) radix sort* which starts from the left-most radix first and goes all the way the right-most radix, and *Least Significant Digit (LSD) radix sort* vice versa. We should address the details of the two forms of radix sort – MSD and LSD using our two examples.

LSD Radix Sorting Integers

LSD radix sort is often used to sort list of integers. It sorts the entire numbers one digit/radix at a time from the least-significant to the most-significant digit. For a list of positive integers where the maximum of them has m digits, LSD radix sort takes a total of m passes to finish sorting.

Here, we demonstrate the process of LSD radix sorting on our exemplary list of integers with counting sort as a subroutine to sort items by using each radix as key. $m = 3$ in our example:

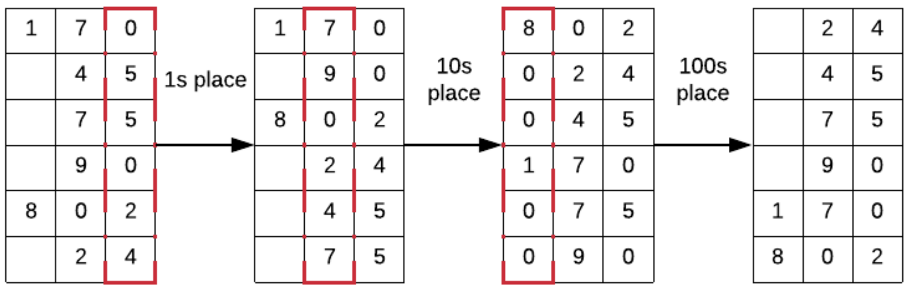


Figure 9: Radix Sort: LSD sorting integers in iteration

- As shown in Fig. 9, in the first pass, the least significant digit (1st place) is used as key to sort. After this pass, the ordering of numbers of unit digits is in-place.
- In the second pass, the 10s place digit is used. After this pass, we see that numbers that has less than or equals to two digits comprising $24, 45, 75, 90$ in our example is in ordering.
- At the last and third pass, the 100s place digit is used. For numbers that are short of 100s place digit, 0 is placed. Afterwards, the entire numbers are in ordering.

We have to notice that the sorting will not work unless the sorting subroutine we apply is stable. For example, in our last pass, there exists four zeros, indicating that they share the same key value. If the relative ordering of them is not kept, the previously sorting effort will be wasted.

Implementation To implement the code with Python, we first need to know how to get each digit out of an integer. With number 178 as an example:

- The least significant digit 8 is the reminder of 178%10.
- The second least-significant digit 7 is the reminder of 17%10.
- And the most-significant digit 1 is the reminder of 1%10.

As we see for digit 8, we need to have 178, for digit 7, we need to have 17, and for digit 1, we only need 1. $178, 17, 1$ are the prefix till the digit we need. We can obtain these prefixes via a base *exp*.

```
exp = 1, (178 // exp ) = 178, 178 % 10 = 8
exp = 10, (178 // exp ) = 17, 17 % 10 = 7
exp = 100, (178 // exp ) = 1, 1 % 10 = 1
```

We can also get the prefix by looping and each time we divide our number by 10. For example, the following code will output [8, 7, 1].

```
a = 178
digits = []
while a > 0:
    digits.append(a%10)
    a = a // 10
```

Now, we know the number of loops we need is decided by the maximum positive integer in our input array. On the code basis, we use a `while` loop to obtain the prefix and making sure that it is larger than 0. At each pass, we call `count_sort` subroutine to sort the input list. The code is shown as:

```
def radixSort(a):
  maxInt = max(a)
  exp = 1
  while maxInt // exp > 0:
    a = count_sort(a, exp)
    exp *= 10
  return a
```

For subroutine `count_sort` subroutine, it is highly similar to our previously implemented counting sort but two minor differences:

- Because we sort by digits, therefore, we have to use a formula: $key = (key//exp)\%10$ to covert the key to digit.
- Because for decimal there are in total only 10 digits, we only arrange 10 total space for the `count` array.

The code is as:

```
def count_sort(a, exp):
  count = [0] * 10 # [0, 9]
  n = len(a)
  order = [0] * n
  # Get occurrence
  for key in a:
    key = (key // exp) % 10
    count[key] += 1

  # Get prefix sum
  for i in range(1, 10):
    count[i] += count[i-1]

  # Put key in position
  for i in range(n-1, -1, -1):
    key = (a[i] // exp) % 10
    count[key] -= 1 # to get the index as position
    order[count[key]] = a[i]
  return order
```

Properties and Complexity Analysis Radix sorting for integers takes m passes with m as the total digits, and each pass takes $O(n+k)$, where $k = 10$ since there is only 10 digits for decimals. This gives out a total of $O(mn)$ time complexity, and m is rather of a constant compared with variable n, thus radix sorting for integers with counting sort as subroutine has a linear time complexity. Due to the usage of counting sort, which is stable, making the radix sorting a stable sorting algorithm too.

With the usage of auxiliary `count` and `order`, it gives a $O(n)$ space complexity, and makes the LSD integer sorting an out-of- place sorting algorithm.

MSD Radix Sorting Strings

In our fruit alphabetization example, it uses MSD radix sorting and groups the strings by a single letter with either bucket sort or counting sort under the hood, starting from the very first letter on the left side all the way to the very last on the right if necessary. MSD radix sorting is usually implemented with recursion.

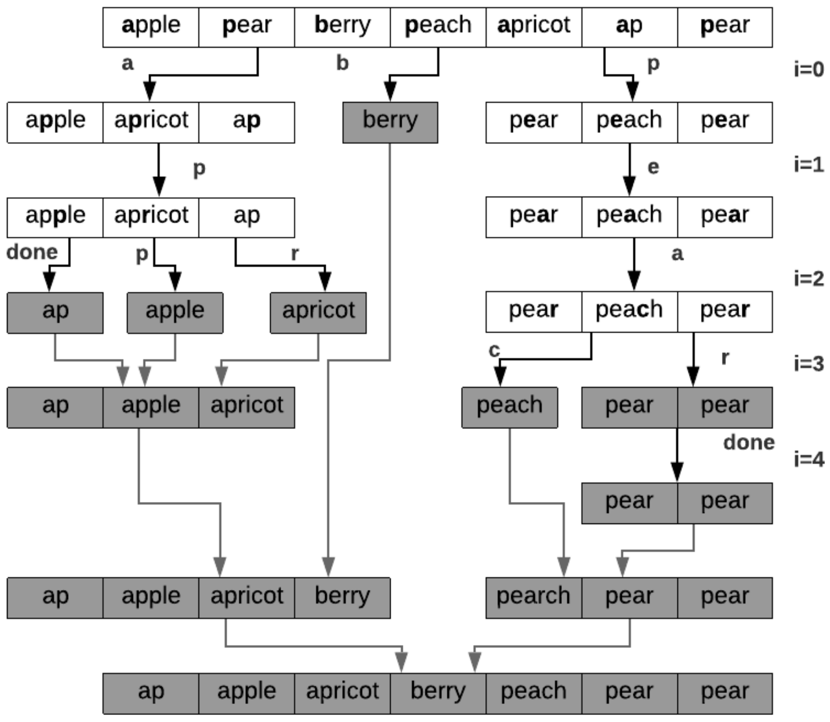


Figure 10: Radix Sort: MSD sorting strings in recursion. The black and grey arrows indicate the forward and backward pass in recursion, respectively.

For better demonstration, we add two more strings: “ap” and “pear”.

String “ap” is for showing what happens when the strings in the same bucket but one is shorter and has no valid letter to compare with. And string “pear” is to showcase how the algorithm handles duplicates. The algorithm indeed applies the recursive divide and conquer design methodology.

Implementation We show bucket sort here, as in lexicographical ordering, the first letter of the strings already decide the ordering of groups of strings bucketed by this letter. This LSD sorting method divide the strings into different buckets indexed by letter, and it then combines the returned results together to get the final sorted strings, which is highly similar to merge sort.

- At first, the recursion handles the first keys of the first letter in the string, as the process where $i = 0$ shown in Fig. 10. There are three buckets with letter ‘a’, ‘b’, and ‘p’.
- At depth 2 when $i = 1$, the resulting buckets from depth 1 is further bucketed by the second letter. Bucket ‘b’ contains only one item, which itself is sorted, thus, the recursion end for this bucket. For the last bucket ‘a’ and ‘p’, they are further bucketed by letter ‘p’ and ‘e’.
- At depth 3 when $i = 2$, for the last bucket ‘p’, it further results two more buckets ‘p’ and ‘r’. However, for string ‘ap’ in the bucket, there is no valid third letter to use as index. And according to the lexicographical order, it puts ‘ap’ in earlier ordering of the resulted buckets.
- In our example, the forward process in the recursion is totally done when $i = 4$. It then enters into the backward phase, which merges buckets that are either composed of a single item or the `done_bucket`.

The code is offered as:

```python
def MSD_radix_string_sort(a, i):
  # End condition: bucket has only one item
  if len(a) <= 1:
    return a

  # Divide
  buckets = [[] for _ in range(26)]
  done_bucket = []
  for s in a:
    if i >= len(s):
      done_bucket.append(s)
    else:
      buckets[ord(s[i]) - ord('a')].append(s)
  # Conquer and chain all buckets
  ans = []
  for b in buckets:
    ans += MSD_radix_string_sort(b, i + 1)
  return done_bucket + ans
```

Properties and Complexity Analysis Because the bucket sort itself is a stable sorting algorithm, making the radix sort for string stable too.

The complexity analysis for the recursive Radix sorting can be accomplished with recursion tree. The tree has nearly n leaves. The worst case occurs when all strings within the input array are the same, thus the recursion tree degrades to linear structure with length n and within each node $O(n)$ is spent to scan items of corresponding letter, making the worst time complexity $O(n^2)$.

For the existence of auxiliary `buckets`, `done_bucket`, and `ans` arrays in sorting of strings, it is an out-of-place sorting. With the same recursion tree analysis for space, we have linear space complexity too.

0.6 Python Built-in Sort

There are two built-in functions to sort `list` and other iterable objects in Python 3, and both of them are stable. In default, they use `<` comparisons between items and sort items in increasing order.

- Built-in method `list.sort(key=None, reverse=False)` of `list` which sorts the items in the list in-place, and returns `None`.
- Built-in function `sorted(iterable, key=None, reverse=False)` works on *any iterable object*, including `list`, `string`, `tuple`, `dict`, and so on. It sorts the items out-of-place; returning another `list` and keeps the original input unmodified.

Basics

To use the above two built-in methods to sort a list of integers is just as simple as:

```
lst = [4, 5, 8, 1, 2, 7]
lst.sort()
```

Printing out `lst` shows that the sorting happens in-place within `lst`.

```
[1, 2, 4, 5, 7, 8]
```

Now, use `sorted()` for the same list:

```
lst = [4, 5, 8, 1, 2, 7]
new_lst = sorted(lst)
```

We print out:

```
new_lst, lst
([1, 2, 4, 5, 7, 8], [4, 5, 8, 1, 2, 7])
```

Let's try to sort other iterable object, and try sort a tuple of strings:

```
fruit = ('apple', 'pear', 'berry', 'peach', 'apricot')
new_fruit = sorted(fruit)
```

Print out `new_fruit`, and we also see that it returned a `list` instead of `tuple`.

```
['apple', 'apricot', 'berry', 'peach', 'pear']
```

Note: For `list`, `list.sort()` is faster than `sorted()` because it doesn't have to create a copy. For any other iterable, we have no choice but to apply `sorted()` instead.

Change Comparison Operator What if we want to redefine the behavior of comparison operator `<`? Other than writing a class and defining `__lt__()`, in Python 2, these two built-in functions has another argument, `cmp`, but it is totally dropped in Python 3. We can use `functools`'s `cmp_to_key` method to convert to key in Python 3. For example, we want to sort [4, 5, 8, 1, 2, 7] in reverse order, we can define a `cmp` function that reverse the order of items to be compared:

```
def cmp(x, y):
  return y - x
```

And then we call this function as:

```
from functools import cmp_to_key
lst.sort(key=cmp_to_key(cmp))
```

The printout of `lst` is:

```
[8, 7, 5, 4, 2, 1]
```

Timsort These two methods both using the same sorting method – *Timsort* and has the same parameters. Timesort is a hybrid stable and in-place sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", January 1993. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered, and uses that knowledge to sort the remainder more efficiently.

Arguments

They both takes two keyword-only arguments: `key` and `reverse`:, and each has `None` and `False` as default value, respectively.

- Argument `key`: ot specifies a function of one argument that is used to extract a comparison key from each list item (for example, `key=str.lower`). If not set, the default value `None` means that the list items are sorted directly.

- Argument `reverse`: a boolean value. If set to `True`, then the list or iterable is sorted as if each comparison were reversed(use `>=` sorted list is in Descending order. The dafault value is False.

Sort in Reverse Order Set `reverse=True` will sort a list of integers in decreasing order:

```
lst = [4, 5, 8, 1, 2, 7]
lst.sort(reverse=True)
```

Print out `lst`, we see:

```
[8, 7, 5, 4, 2, 1]
```

This is equivalent to customize a class `Int` and rewrite its `__lt__()` special method as:

```
class Int(int):
  def __init__(self, val):
    self.val = val
  def __lt__(self, other):
    return other.val < self.val
```

Now, sort the same list but without setting `reverse` will get us exactly the same result:

```
lst = [Int(4), Int(5), Int(8), Int(1), Int(2), Int(7)]
lst.sort()
```

Customize `key` We have mainly two options to customize the `key` argument: (1) through `lambda` function, (2) through a pre-defined function. And in either way, the function only takes one argument. For example, to sort the following list of tuples by using the second item in the tuple as key:

```
lst = [(8, 1), (5, 7), (4, 1), (1, 3), (2, 4)]
```

We can write a function, and set `key` argument to this function

```
def get_key(x):
  return x[1]
new_lst = sorted(lst, key = get_key)
```

The sorted result is:

```
[(8, 1), (4, 1), (1, 3), (2, 4), (5, 7)]
```

The same result can be achieved via lambda function which is more convenient:

```
new_lst = sorted(lst, key = lambda x: x[1])
```

Same rule applies to objects with named attributes. For example, we have the following class named `Student` that comes with three attributes: `name`, `grade`, and `age`, and we want to sort a list of `Student` class only through the age.

```python
class Student(object):
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age

    # To support indexing
    def __getitem__(self, key):
        return (self.name, self.grade, self.age)[key]

    def __repr__(self):
        return repr((self.name, self.grade, self.age))
```

We can do it through setting `key` argument still:

```python
students = [Student('john', 'A', 15), Student('jane', 'B', 12),
    Student('dave', 'B', 10)]
sorted(students, key=lambda x: x.age)
```

which outputs the following result:

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()` and `attrgetter()` to get the attributes by index and name, respectively. For the sorting above, we can do it like this:

```python
from operator import attrgetter
sorted(students, key=attrgetter('age'))
```

`attrgetter` can take multiple arguments, for example, we can sort the list first by 'grade' and then by 'age', we can do it as:

```python
sorted(students, key=attrgetter('grade', 'age'))
```

which outputs the following result:

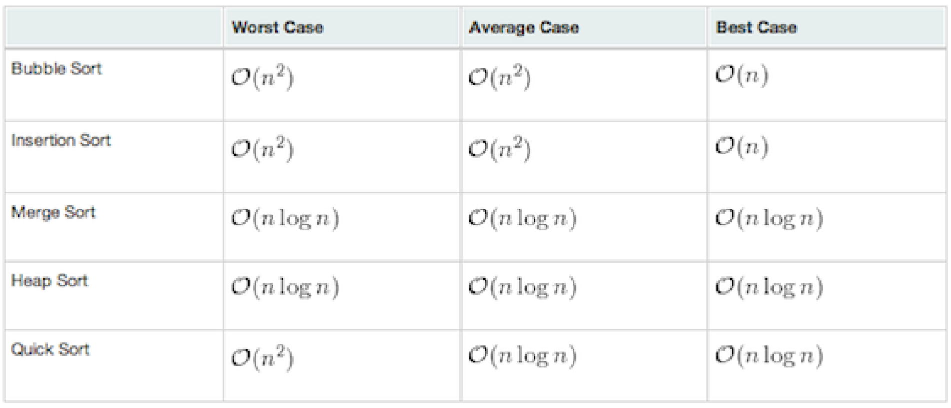
```
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

If our object supports indexing, which is why we defined `__getitem__()` in the class, we can use `itemgetter()` to do the same thing:

```python
from operator import itemgetter
sorted(students, key=itemgetter(2))
```

0.7 Summary and Bonus

Here, we give a comprehensive summary of the time complexity for different sorting algorithms.



	Worst Case	Average Case	Best Case
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Figure 11: The time complexity for common sorting algorithms

0.8 LeetCode Problems

Problems

0.1 **Insertion Sort List (147).** Sort a linked list using insertion sort. A graphical example of insertion sort. The partial sorted list (black) initially contains only the first element in the list. With each iteration one element (red) is removed from the input data and inserted in-place into the sorted list

Algorithm of Insertion Sort: Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

```
Example 1:
Input: 4->2->1->3
Output: 1->2->3->4

Example 2:
Input: -1->5->3->4->0
Output: -1->0->3->4->5
```

0.2 **Merge Intervals (56, medium).** Given a collection of intervals, merge all overlapping intervals.

```
Example 1:
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps,
    merge them into [1,6].
```

```
Example 2:
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered
    overlapping.
```

0.3 **Valid Anagram (242, easy).** Given two strings s and t , write a function to determine if t is an anagram of s.

```
Example 1:
Input: s = "anagram", t = "nagaram"
Output: true

Example 2:
Input: s = "rat", t = "car"
Output: false
```

Note: You may assume the string contains only lowercase alphabets.

Follow up: *What if the inputs contain unicode characters? How would you adapt your solution to such case?*

0.4 **Largest Number (179, medium).**

0.5 **Sort Colors (leetcode: 75).** Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue. Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively. *Note: You are not suppose to use the library's sort function for this problem.*

0.6 **148. Sort List (sort linked list using merge sort or quick sort).**

Solutions

1. Solution: the insertion sort is easy, we need to compare current node with all previous sorted elements. However, to do it in the linked list, we need to know how to iterate elements, how to build a new list. In this algorithm, we need two while loops to iterate: the first loop go through from the second node to the last node, the second loop go through the whole sorted list to compare the value of the current node to the sorted element, which starts from having one element. There are three cases for the comparison: if the comp_node does not move, which means we need to put the current node in front the previous head, and the cur_node become the new head; if the comp_node stops at the back of it, so current node is the end, we set its value to 0, and we save the pre_node in case; if it stops in the middle, we need to put cur_node in between pre_node and cur_node.

```python
def insertionSortList(self, head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if head is None:
        return head
    sorted_head = head
    cur_node = head.next
    head.next = None #sorted list only has one node, a new list
    while cur_node:
        next_node = cur_node.next #save the next node
        cmp_node = head
        #compare node with previous all
        pre_node = None
        while cmp_node and cmp_node.val <= cur_node.val:
            pre_node = cmp_node
            cmp_node = cmp_node.next

        if cmp_node == head: #put in the front
            cur_node.next = head
            head = cur_node
        elif cmp_node == None: #put at the back
            cur_node.next = None #current node is the end, so set it to None
            pre_node.next = cur_node
            #head is not changed
        else: #in the middle, insert
            pre_node.next = cur_node
            cur_node.next = cmp_node
        cur_node = next_node
    return head
```

2. Solution: Merging intervals is a classical case that use sorting. If we do the sorting at first, and keep track our merged intervals in a heap (which itself its sorted too), we just iterate into the sorted intervals, to see if it should be merged in the previous interval or just be added into the heap. Here the code is tested into Python on the Leetcode, however for the python3 it needs to resolve the problem of the heappush with customized class as iterable item.

```python
# Definition for an interval.
# class Interval(object):
#     def __init__(self, s=0, e=0):
#         self.start = s
#         self.end = e
from heapq import heappush, heappop

class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[Interval]
        """
        if not intervals:
            return []
        #sorting the intervals nlogn
        intervals.sort(key=lambda x:(x.start, x.end))
        h = [intervals[0]]
        # iterate the intervals to add
        for i in intervals[1:]:
            s, e = i.start, i.end
            bAdd = False
            for idx, pre_interal in enumerate(h):
                s_before, e_before = pre_interal.start,
    pre_interal.end
                if s <= e_before: #overlap, merge to the
    same interval
                    h[idx].end = max(e, e_before)
                    bAdd = True
                    break
            if not bAdd:
                #no overlap, push to the heap
                heappush(h, i)
        return h
```

3. Solution: there could have so many ways to do it, the most easy one is to sort the letters in each string and see if it is the same. Or we can have an array of 26, and save the count of each letter, and check each letter in the other one string.

```
def isAnagram(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        return ''.join(sorted(list(s))) == ''.join(sorted(
    list(t)))
```

The second solution is to use a fixed number of counter.

```
def isAnagram(self, s, t):
    """
    :type s: str
    :type t: str
    :rtype: bool
    """
    if len(s) != len(t):
        return False
    table = [0]*26
    start = ord('a')
    for c1, c2 in zip(s, t):
        print(c1, c2)
        table[ord(c1)-start] += 1
        table[ord(c2)-start] -= 1
    for n in table:
        if n != 0:
            return False
    return True
```

For the follow up, use a hash table instead of a fixed size counter. Imagine allocating a large size array to fit the entire range of unicode characters, which could go up to more than 1 million. A hash table is a more generic solution and could adapt to any range of characters.

4. Solution: from instinct, we know we need sorting to solve this problem. From the above example, we can see that sorting them by integer is not working, because if we do this, with 30, 3, we get 303, while the right answer is 333. To review the sort built-in function, we need to give a key function and rewrite the function, to see if it is larger, we compare the concatenated value of a and b, if it is larger. The time complexity here is $O(n \log n)$.

```
class LargerNumKey(str):
    def __lt__(x, y):
        return x+y > y+x

class Solution:
    def largestNumber(self, nums):
        largest_num = ''.join(sorted(map(str, nums), key=LargerNumKey))
        return '0' if largest_num[0] == '0' else largest_num
```