A Deep Survey in Sorting Algorithms

Runlin Hou

April 7, 2020

Contents

1 Introduction

The sorting algorithm is an essential area in computer science. The purpose of sorting algorithms is simple, to put elements of a list in a certain order according to the values.

And sorting algorithms are widely used in many different areas, like the file system in the Windows operation system, we can choose to arrange the files in order of time or size and so on. Also, all the ranking list is based on the sorting algorithms. As a matter of fact, we can say that the sorting algorithm is one of the most critical basement algorithms of the modern computer program.

So I think it would be meaningful to dig deep into sorting algorithms. Also, sorting algorithms are helpful to practice so-called 'programming paradigms,' which basically means to understand what your computer is doing and how it achieves its job.

For this survey, I would research in several classic sorting algorithms to compare the time consumption and space requirement. Also, we will discuss how the algorithms are achieved.

2 Algorithms

I decided to research on five classic algorithms in this section.

- Insertion Sort
- Bubble Sort
- Quick Sort
- Counting Sort
- Bucket Sort

All these algorithms would be analyzed from serveral aspects like: implementation logic, time consumption, space requirement and these attributes would be cmopared in detail in the next section.

2.1 Insertion Sort

The basic logic of insertion sort is simple. The array will be divided into two parts by a pointer, who has the following properties. All elements in the left part should be sorted and all elements in the right part haven't been moved.

The operation to achieve insertion sort is also easy to understand. That can be simplely described like this: pick an element from the original array and find the position where it belongs to.

The pseudocode is as following:

pesudocode

```
set pointer from head
arr = input_unsorted_array

while pointer < len(arr):
    pos = pointer
    while arr[pos] < arr[pos - 1]:
        exchange(arr[pos], arr[pos-1])
        pos move backward
    endwhile
    pointer move forward
endwhile
```

The time consumption of this algorithm is $O(n^2)$. As we can see, in each loop the pointer will move to next element. And to find the position of each element, we need to trace back to find where it belongs to. Clearly, it requires additional comparision operation to find the right position. So, we use loop nesting to implement the program. Then the time consumption would be $n*n$, which represented by big oh as $O(n^2)$. For space requirement, insertion sort only need an additional space for the pointer and no more space is needed, so the space complexity would be $O(1)$.

2.2 Bubble Sort

Bubble sort as what its name described is just like a bubble keeps rising in a cup of water. It works by repeatedly swapping the adjacent elements if they are in the wrong order. In each loop, what we do is to move the largest

element to the tail, meanwhile, since it will try to move the larger element close to tail as possible, the result in practice will be a little bit more sorted each loop.

To implement Bubble Sort, what we need is also just one pointer. In each loop, we will compare if the element been pointed is larger than the next element. If it does, we will swap the two elements and move the pointer forward. If not, we just move the pointer forward.

pesudocode

```
arr = input_unsorted_array
n = len(arr)

for i in range(n):
    for j in range(0, n-i-1):
        if arr[j] < arr[j+1]:
            swap(arr[j], arr[j+1])
        endif
    endfor
endfor
```

For Bubble Sort, it is also clear that we use loop nesting to achieve the goal. Since we can only set one element to correct position in one loop and meanwhile we need n comparison operations to achieve that. So the time consumption would be $O(n^2)$. Space complexity would also be $O(1)$ for the extra pointer.

2.3 Quick Sort

The basic logic of Quick Sort is to pick an element as a criterion and divide the array into two parts, where all elements on the left are smaller than the criterion, and all elements on the right are greater than the criterion. Then recursively do the same to the two parts.

The implementation of Quick Sort can be achieved by two functions. The partition function picks the criterion and separates the array. Then we will recursively call function `partition()` for each part as a new unsorted array.

pesudocode

```
def partition(arr, low, high):
    arr = input_unsorted_array
    pivot = arr[high]

    while haven't reach the last element greater than pivot:
        find an element larger than pivot from head
        find an element greater than pivot from tail
        swap the two elements
    endwhile

    swap the last element greater than pivot with pivot
    return pivot position

def quicksort(arr, low, high):
    if low == high:
        return
    endif

    pos = partition(arr, low, high)

    partition(arr, low, pos-1)
    partition(arr, pos+1, high)
```

The analysis of quicksort would be considered both the worst case and the best case since it is really important to know the differences. The best case would be every pick of pivot happened to be the middle of the input array. So that we only have to make the comparison of the whole array from only $logn$ times, since every time we separate the array into two parts, and by $logn$ times would be separated into one element. Then the time consumption would be $O(nlogn)$. The worst case would be we pick the smallest or the largest element as pivot every time. So that we would be forced to make comparisons between pivot with the whole array every time and meanwhile since we can only set one element to head or tail once, it would take n loop to set the whole array right. The time consumption of this condition would be $O(n^2)$. The space complexity would be $O(1)$.

2.4 Counting Sort

Counting Sort is different from the previous sorting algorithm. All the previous sorting algorithms depend on the comparison between each pair of elements. For counting sort, we don't decide the position of elements by comparison with each other. Instead, we save the another of each nonredundant elements in an auxiliary array. And then fill the original array according to the auxiliary array.

The implementation is easy. We first find the smallest and largest elements of the array. And then create then auxiliary for saving each elements' amount. So we can achieve the algorithm with three traversals . We can decide the largest and the smallest elements in the first traversal. In the second traversal, we will record the amount of each element in the auxiliary array. In the last traversal, we will fill the original array in order according to the auxiliary array.To save time, we can optimize by finding the largest element and record the amount in the same traversal.

pesudocode

```
arr = input_unsorted_array
max = 0

aux = [0 for i in range(max+1)]

for i in arr:
    if i > max:
        max = i
    endif
    aux[arr[i]] = aux[arr[i]] + 1
endfor

index = 0
value = 0
for i in aux:
    while i > 0:
        arr[index] = value
        index = index + 1
        i = i - 1
```

```
        endwhile
        value = value + 1
    endfor
```

For counting sort, space complexity will be relatively larger than the previous algorithms, since it requires space for the auxiliary array, which is $O(k)$ (where k is the largest element in the array subtract the smallest). As a payback, compared to the previous algorithms, the time consumption will be reduced to $O(n + k)$.

2.5 Bucket Sort

The thought to implement Bucket Sort is to separate the array into ordered buckets. And then sort the elements inside the buckets individually. So the Bucket Sort can be seen as an upgrade of Counting Sort.

To implement Bucket Sort, we should set the number of buckets first and put elements into each bucket. After setting up each bucket, we sort the elements inside each bucket and joint buckets together.

pesudocode

```
arr = input_unsorted_array
bucket_num = set_bucket_number
min = max = arr[0]

for i in arr:
    if min > arr[i]:
        min = arr[i]

    if max < arr[i]:
        max = arr[i]
endfor

//set the sections of each bucket
section = int((max - min)/bucket_num)

//create bucket_num empty buckets from bucket[0]
buckets = [[] for i in range(bucket_num)]
```

```
for i in arr:
    the_bucket_num = i mod section
    put i into the_bucket_num bucket
endfor

for bucket in buckets:
    insertion_sort(bucket)
endfor

res = []
for bucket in buckets:
    for ele in bucket:
        res.append(ele)
    endfor
endfor
```

For the Bucket Sort, since we will have a much shorter and more sorted array inside each bucket, we choose insertion sort for its great performance on this kind of array. Meanwhile, the more buckets have the better performance we will get. So the best case would be $O(n)$ that each element happened to be put in one bucket. But if almost every element was put into one bucket, we hit the worst case that takes time consumption as $O(n^2)$ cause we basically do an insertion sort to the array. And the space complexity would be $O(n)$ since we save all elements to buckets. But the space complexity can be reduced by using the chain table since we just have to save the head node of each bucket. Then the space complexity would be reduced to $O(k)$ where k is the number of buckets.

3 Application Usage

Every algorithm is designed to serve people for a certain purpose. But when facing different situations, we will always make a different requirement. Not only having requirements for time consumption and space complexity but also having requirements for stability.

For example, when we are trying to find a ticket in a certain period of time, we also want tickets to different destinations been sorted by the initial

letter. To achieve this goal, we need the sequence sorted by the time first and then do another sorting to the sequence according to the destination. In this situation, if we use an unstable sorting algorithm like QuickSort, the first sorting would bee meaningless since the element will no longer keep its original order.

The algorithms I mentioned in the last section can be divided into comparison sorting and non-comparison sorting. As we can see, non-comparison sorting always pays space complexity for time consumption. Since comparison sorting has a limitation of time consumption can not be reduced below $O(nlogn)$, which non-comparison does not have, sometimes it might be a good choice to choose non-comparison sorting. When we have enough space, we can use the extra space to trade for time-saving, since the time consumption can reach a level of $O(n)$.