Algorithm Experiment
Sorting Algorithm

李胜锐 2017012066

2020 年 3 月 21 日

Date Performed:　2020 年 3 月 21 日

1 Experiment Environment

Operating System: macOS 10.15.2
Processor: 2.6 GHz 6-Core Intel Core i7
Memory: 16GB
Language: python 3.7
IDE: PyCharm

2 Algorithm Analysis

2.1 Insertion Sort

```
def insert_sort(arr):
    for i in range(1, len(arr)):
        x = arr[i]
        t = 0
        for j in range(i - 1, -1, -1):
            if x < arr[j]:
                arr[j + 1] = arr[j]
            else:
                t = j + 1
                break
        arr[t] = x

    return arr
```

Insertion sort is one of the most naive sorting algorithm. It first constructs a sorted array in front of the whole array and add other unsorted elements into the sorted part pne by one.
Hence, the time complexity of this algorithm is:

$$T(n) = \Theta(n^2)$$

2.2 Shell Sort

```
def shell_sort(arr):
    def shell_insert(arr, d):
        for i in range(d, len(arr)):
            x = arr[i]
            t = (i - d) % d
```

```
            for j in range(i - d, -1, -d):
                if x < arr[j]:
                    arr[j + d] = arr[j]
                else:
                    t = j + d
                    break
            arr[t] = x

    length = len(arr)

    while length >= 1:
        shell_insert(arr, length)
        length = int(length / 2)

    return arr
```

This method includes two phase.

The first phase is Shell insert. It divides a whole array into N groups and all elements in one group have the same value of $i\%N$, where i is the subscript of the element. Then all elements in the same group will be sorted by insert sort mentioned above.

The second phase is to use different N and repeat the process of Shell insert. The list of N must be a monotone decreasing list and the last N is 1.

When n is large enough, it can be estimated that:

$$T(n) = \Theta(n^{1.3})$$

2.3 Quick Sort

```
def quick_sort(arr, p, r):
    def partition(arr, p, r):
        x = arr[r]
        i = p - 1
        for j in range(p, r):
            if arr[j] <= x:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[r] = arr[r], arr[i + 1]
        return i + 1

    def random_partition(arr, p, r):
        i = random.randint(p, r)
        arr[r], arr[i] = arr[i], arr[r]
```

```
        return partition(arr, p, r)

    if p < r:
        q = random_partition(arr, p, r)
        quick_sort(arr, p, q - 1)
        quick_sort(arr, q + 1, r)
```

Quick sort is a recursive sorting algorithm. It finds a pivot and makes elements on the left less than or equal to the pivot and makes the elements on the right greater than it.
Here, my algorithm is the randomized version that will prevent the presence of the worst case that makes the time complexity degenerate to $\Theta(n^2)$.
The expected time complexity of the randomized quick sort is:

$$T(n) = \Theta(nlogn)$$

2.4 Merge Sort

```
def merge_sort(arr, p, r):
    if p == r:
        return
    pivot = int((p + r) / 2)
    merge_sort(arr, p, pivot)
    merge_sort(arr, pivot + 1, r)
    tmp = []
    i = p
    j = pivot + 1
    while i <= pivot and j <= r:
        if arr[i] <= arr[j]:
            tmp.append(arr[i])
            i += 1
        else:
            tmp.append(arr[j])
            j += 1
    while i <= pivot:
        tmp.append(arr[i])
        i += 1
    while j <= r:
        tmp.append(arr[j])
        j += 1
    for t, number in enumerate(tmp):
        arr[p + t] = number
```

Merge sort is also a recursive sorting algorithm. It makes use of additional storage space and two sorted array. Elements from two sorted array are added to the additional storage space by order.
The expected time complexity of merge sort is:

$$T(n) = \Theta(nlogn)$$

2.5 Radix Sort

```
def radix_sort(arr, r):
    def key_of_number(number, begin, end):  # from right to left
        return (number >> begin) & (2 ** (end - begin) - 1)

    def count_sort(des, src, begin, end):
        k = 2 ** (end - begin)
        c = [0 for i in range(k)]
        for j in range(len(src)):
            c[key_of_number(src[j], begin, end)] += 1
        for i in range(1, k):
            c[i] = c[i] + c[i - 1]
        for j in range(len(src) - 1, -1, -1):
            des[c[key_of_number(src[j], begin, end)] - 1] = src[j]
            c[key_of_number(src[j], begin, end)] -= 1

    for begin in range(0, 32, r):
        end = begin + r
        des = [0 for i in range(len(arr))]
        count_sort(des, arr, begin, end)
        arr = des
    return arr
```

Radix sort is based on counting sort. It divides a binary number into few parts and conducts counting sort for each of the part from right to left. There is a hyperparameter r, representing the length of each part. It can be estimated that when $r = lgn$, the algorithm cost the least time:

$$T(n, b) = \Theta(bn/\lg n)$$

where b is the length of the binary number.

3 Result Analysis

3.1 Comparision of All 5 Methods

lg(n)	insert sort	shell sort	quick sort	merge sort	radix sort
1	0.0000	0.0000	0.0000	0.0000	0.0007
2	0.0003	0.0003	0.0003	0.0003	0.0028
3	0.0337	0.0046	0.0024	0.0031	0.0147
4	2.9791	0.0757	0.0316	0.0394	0.1092

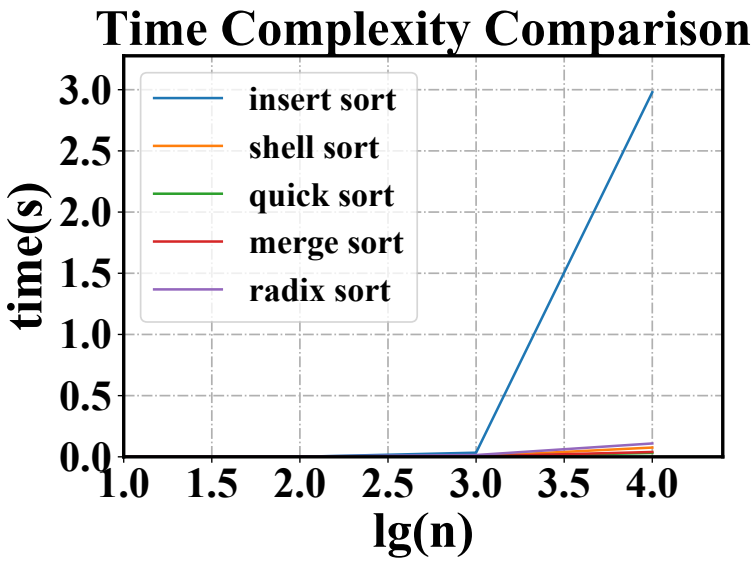

图 1: Comparision of all 5 methods

1) Insert sort spends much more time than the other algorithms. Even when n is just 10000, it will spend nearly 3 seconds. So there it is a waste of time to compare insert sort with others when n is larger.

3.2 Comparision of 4 Methods

1) When n is between 10^4 to 10^6, quick sort is the fastest.

lg(n)	shell sort	quick sort	merge sort	radix sort
1	0.0000	0.0000	0.0000	0.0007
2	0.0003	0.0003	0.0003	0.0029
3	0.0058	0.0043	0.0056	0.0238
4	0.0981	0.0706	0.0643	0.1279
5	1.0585	0.3562	0.4796	0.9844
6	17.7935	4.8960	6.5673	10.1497

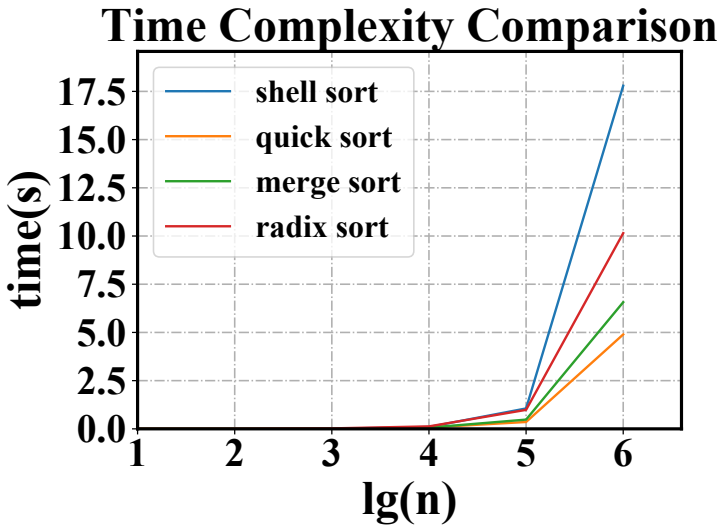

图 2: Comparision of 4 Methods

2) Radix sort is slower than both merge sort and quick sort whose time complexity is $\Theta(nlogn)$ when $n < 10^6$.

3) Shell sort has already been the slowest when $n > 10^5$. Obviously, when n continues to increase, Shell sort will still be the slowest among the four algorithm.

3.3 Comparision of Quick/Merge/Radix Sort($r = \log_{10} n$)

1) When $n <= 10^7$, quick sort is faster than radix sort. But when $n = 10^8$ and $n = 2 \times 10^8$, radix sort is faster than quick sort.

lg(n)	quick sort	merge sort	radix sort
4	0.0381	0.0511	0.1359
5	0.5756	0.5349	1.1161
6	4.2409	6.5725	10.7834
7	61.0320	85.7265	91.3984
8	861.1079	1060.8313	764.5378

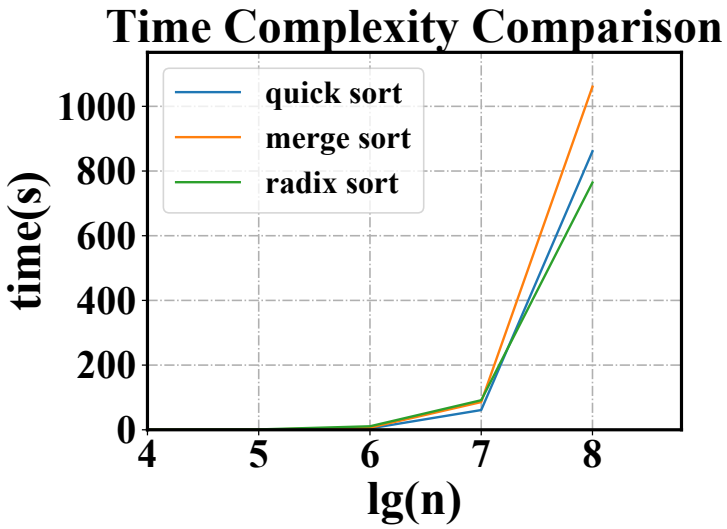

图 3: Comparision of Quick/ Merge/ Radix Sort($r = \log_{10} n$)

2) Quick sort is faster than merge sort in any scale.

lg(n)	quick sort	merge sort	radix sort
6.0	4.3200	6.4579	10.1635
7.0	60.4169	79.5469	90.1600
8.0	747.1955	1012.2965	736.8886
8.3	1763.9016	2236.5468	1560.0444

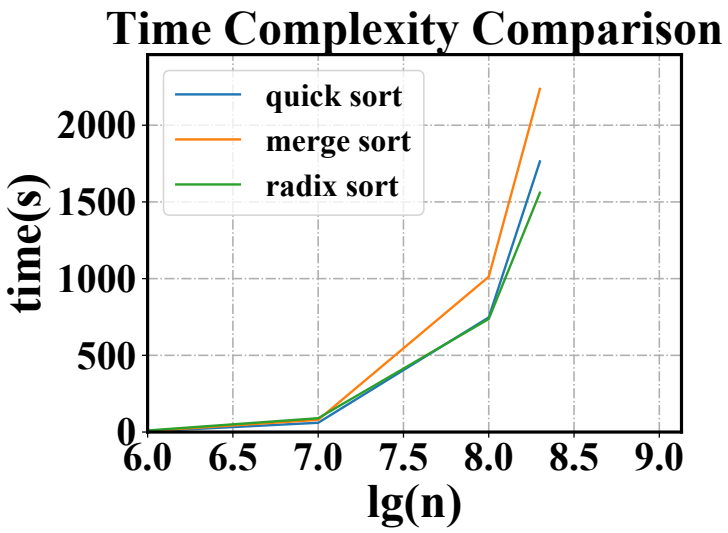

图 4: Comparision of Quick/ Merge/ Radix Sort($r = \log_{10} n$)

3.4 Comparision of Radix Sort with different r

lgn	2	4	6	8	10	12	14	16
2	0.5498	0.2747	0.2265	0.1617	0.2651	0.5419	1.6934	4.2869
3	2.0323	0.9320	0.7451	0.5197	0.5277	0.4760	0.9314	1.8358
4	2.6461	1.3422	0.9557	0.6297	0.6662	0.5117	0.5726	0.6758
5	2.6934	1.3948	1.0081	0.6764	0.7320	0.5399	0.5579	0.3976
6	2.7423	1.3886	1.0265	0.6789	0.7071	0.5255	0.5568	0.3743

1) In fact, if we choose larger r in the above experiment, radix sort will perform even better.

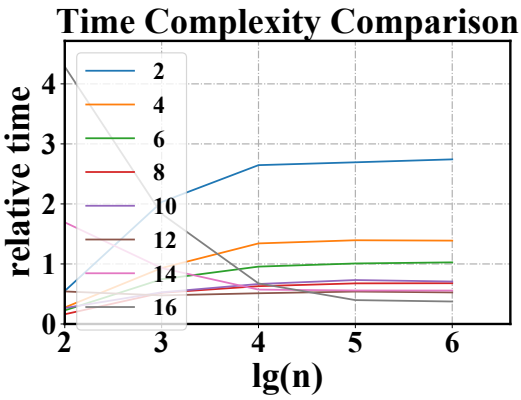

图 5: Comparision of Radix Sort with different r

lgn	radix 2	radix 4	radix 8	radix 16	quick sort
2	0.0024	0.0012	0.0007	0.0179	0.0002
3	0.0235	0.0114	0.0058	0.0218	0.0023
4	0.2381	0.1186	0.0611	0.0533	0.0309
5	2.5924	1.2577	0.6000	0.3379	0.3590
6	30.0230	15.0923	7.4540	3.8458	4.3160

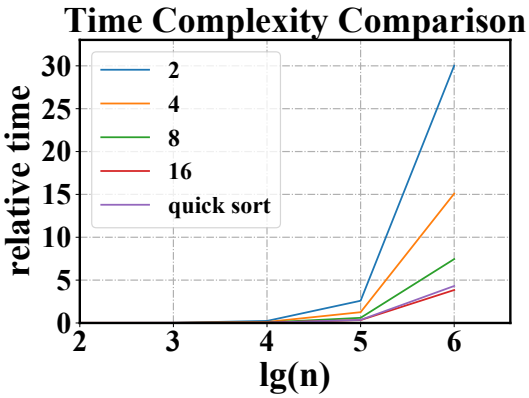

图 6: Comparision between Radix Sort and Quick Sort

2) When n is small, smaller r will lead to faster performance.

3) When $n \geq 10^5$, $r = 16$ will be the fastest among all r.

4) If we can choose optimal r in experiment with different n, radix sort can outperform quick sort only when $n \geq 10^4$.

4 Conclusions

1) The time complexities of the five algorithms are:

$$\text{Insert Sort:} T_{Insert}(n) = \Theta(n^2)$$

$$\text{Shell Sort:} T_{Shell}(n) = \Theta(n^{1.3})$$

$$\text{Quick Sort:} T_{Quick}(n) = \Theta(n \log n)$$

$$\text{Merge Sort:} T_{Merge}(n) = \Theta(n \log n)$$

$$\text{Radix Sort:} T_{Radix}(n, b) = \Theta(bn / \lg n)$$

And the experiment proves the theorem.

2) Insert sort is too slow compare with the others.

3) If we choose r of radix sort as 8. Quick Sort has the best performance when $n \leq 10^7$ among all methods. However, if we choose larger r, such as 16, radix sort will outperform quick sort when $n \geq 10^5$. Futher, If we can choose optimal r in experiment with different n, radix sort can outperform quick sort only when $n \geq 10^4$.