Algorithms and Data Structures
Searching and Sorting

Nguyễn Gia Phong—BI9-184

December 3, 2019

1 Cocktail Shaker Sort

The code is implemented following the cocktail shaker sort's pseudocode[1] with bubble sort's optimization[2] whose time complexity is analyzed as follows

1.1 Best Case

For the matter of brevity, we consider all operations on the array's n members are in constant time ($\Theta(1)$). If the array is already sorted, after the first `while` loop (line 25), `h` is still `low` and thus the `do`–`while` loop is broken. Since the while loop runs from `low + size` to `high - size` by `size` steps, the running time is `(high - low - size*2)/size + 1` or `nmemb - 1`. Therefore the best case time complexity is $\Omega(n-1) = \Omega(n)$.

1.2 Average Case

Assume the average case is when the array is uniformly shuffled, that is, every permutation has the equal probability to occur.

Given a permutation of an n-element array, consider the positive integer $k \le n$ that exactly the last $n-k$ members are continuously in the correct positions (as in the ascendingly sorted array). It is obvious that for $k = 1$, the array is sorted and the probability of the permutation to appear is $1/n!$. For $1 < k \le n$, if we fix the last $n-k$ members in their right places, out of the $k!$ permutations of the first k elements, $(k-1)!$ ones has the k-th greatest

[1] `https://en.wikipedia.org/wiki/Cocktail_shaker_sort#Pseudocode`
[2] `https://en.wikipedia.org/wiki/Bubble_sort#Optimizing_bubble_sort`

at the correct place. Therefore, let X be the number that exactly $n - X$ last elements are in the right positions, we have

$$p_X(k) = \begin{cases} \dfrac{1}{n!} & \text{if } k = 1 \\ \dfrac{k! - (k-1)!}{n!} & \text{otherwise} \end{cases}$$

Applying this to the first `while` (line 25) with n and $X - 1$ being the number of steps from `low` to `high`, before and after `high = h` respectively, the expectation of X is

$$\begin{aligned} \mathbf{E}[X] &= \sum_{k=1}^{n} k p_X(k) \\ &= \frac{1}{n!} + \sum_{k=2}^{n} \frac{k!k - k!}{n!} \\ &= \frac{1}{n!} + \sum_{k=3}^{n+1} \frac{k!}{n!} - \sum_{k=2}^{n} \frac{k!}{n!} - \sum_{k=2}^{n} \frac{k!}{n!} \\ &= \frac{1}{n!} + \frac{(n+1)!}{n!} - \frac{2!}{n!} - \sum_{k=2}^{n} \frac{k!}{n!} \\ &= n + 1 - \sum_{k=1}^{n} \frac{k!}{n!} \\ &= n - \sum_{k=1}^{n-1} \frac{k!}{n!} \end{aligned}$$

Hence after line 28, the newly sorted length of the array is

$$n - \mathbf{E}[X - 1] = n - \mathbf{E}[X] + 1 = 1 + \sum_{k=1}^{n-1} \frac{k!}{n!} = \Theta(1)$$

Similarly, line 31 to 35 also sort $\Theta(1)$ element(s), thus each iteration of the `do`–`while` loop to sort $\Theta(1)$ members. The overall average-case time complexity is

$$\begin{aligned} T(n) &= \begin{cases} (n - \Theta(1)) + (n - \Theta(1)) + T(n - \Theta(1)) & \text{if } n > 0 \\ \Theta(1) & \text{otherwise} \end{cases} \\ &= \begin{cases} 2n - \Theta(1) + T(n - \Theta(1)) & \text{if } n > 0 \\ \Theta(1) & \text{otherwise} \end{cases} \\ &= \Theta(1) + \sum_{k=1}^{m} (2k - \Theta(1)) = 2\sum_{k=1}^{m} k - \sum_{k=1}^{m} \Theta(1) = m^2 + m - \sum_{k=1}^{m} \Theta(1) \end{aligned}$$

where m satisfies

$$\exists\{f_k \mid k \in 1\,..\,m\} \subset \Theta(1),\ \sum_{k=1}^{m} f_k(n) = n \Longrightarrow \sum_{k=1}^{m} \Theta(1) = \Theta(n) \Longrightarrow m = \Theta(n)$$
$$\Longrightarrow T(n) = \Theta\left(n^2\right) + \Theta(n) - \Theta(n) = \Theta\left(n^1\right)$$

1.3 Worst Case

If the array is reversely sorted, after each first `while` (line 25), `high` is decreased by `size`; and after each second `while` (line 32), `low` is increased by `size`. For `low + size >= high`, it takes `(high-low-size)/size + 1 >> 1` or `nmemb / 2` iterations of the `do`–`while` loop (line 23). The overall complexity would then be

$$\begin{aligned}\sum_{k=1}^{\lfloor n/2\rfloor}(n - 2k + 1 + n - 2k) &= \sum_{k=1}^{\lfloor n/2\rfloor}(2n - 4k + 1)\\ &= n^2 + 2\left\lfloor\frac{n}{2}\right\rfloor\left(\left\lfloor\frac{n}{2}\right\rfloor + 1\right) + \left\lfloor\frac{n}{2}\right\rfloor\\ &= O\left(n^2\right)\end{aligned}$$

2 Merge Sort

As usual, the linked list is implemented using classic Lisp's `cons`-cells. The program is thus compiled by

```
cc construct.c Ex2.c -o Ex2
```

To keep the implementation concise, memory safety as well as stack limit was not considered.

It is trivial that the time complexity of `merge` is $\Theta(n)$ with n being the total length of `left` and `right`. For `msort`, the running time of the `while` loop at line 27 is also $\Theta(n)$, where n is the length of the input `list`. The overall time complexity is

$$T(n) = \begin{cases}\Theta(1) & \text{if } n \le 1\\ \Theta(n) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) & \text{otherwise}\end{cases}$$

The recurrence can be stated as

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

By the master theorem[3],

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta\left(n^{\log_2 2}\right) = \Theta\left(n^{\log_2 2} \lg n\right) = \Theta(n \lg n)$$

3 Copying

[3]Let $a \geq 1$ and $b > 1$ be constants, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^{\log_b a}\right)$$

where n/b is interpreted as either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, then

$$T(n) = \Theta\left(n^{\log_b a} \lg n\right)$$