Lab02-Sorting and Searching

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Li Ma, Autumn 2019

∗ Please upload your assignment to website. Contact webmaster for any questions.

∗ Name: Sun Yiwen Student ID: 517370910213 Email: sunyw99@sjtu.edu.cn

1. **Cocktail Sort.** Consider the pseudo code of a sorting algorithm shown in Alg. 1, which is called *Cocktail Sort*, then answer the following questions.

   (a) What is the minimum number of element comparisons performed by the algorithm? When is this minimum achieved?

   (b) What is the maximum number of element comparisons performed by the algorithm? When is this maximum achieved?

   (c) Express the running time of the algorithm in terms of the O notation.

   (d) Can the running time of the algorithm be expressed in terms of the Θ notation? Explain.

Alg. 1: CocktailSort($a[\cdot]$, n)

```
Input: an array a, the length of array n
for i = 0; i < n − 1; i++ do
    bFlag ← true;
    for j = i; j < n − i − 1; j++ do
        if a[j] > a[j + 1] then
            swap(a[j], a[j + 1]);
            bFlag ← false;
    if bFlag then
        break;
    bFlag ← true;
    for j = n − i − 1; j > i; j−− do
        if a[j] < a[j − 1] then
            swap(a[j], a[j − 1]);
            bFlag ← false;
    if bFlag then
        break;
```

Solution. (a) The minimum number of element comparisons performed by the algorithm is $n-1$ times. This is achieved when the input array is already sorted in ascending order.

(b) The maximum number of element comparisons performed by the algorithm is $\frac{n(n-1)}{2}$ times. This is achieved when the input array is sorted in descending order.

(c) Suppose $T(n) = \frac{n(n-1)}{2}$, then if we pick constants c and n_0 so that for any $n > n_0$, $T(n) \leq c \cdot n^2$, then we can prove $T(n) = O(n^2)$. We choose $c = \frac{1}{2}$ and $n_0 = 1$, then for any $n > 1$, $T(n) = \frac{n^2}{2} - \frac{n}{2} < \frac{n^2}{2}$. Therefore, the worst-case running time of this algorithm is $O(n^2)$.
For average case time complexity, we know that the above cocktail sort is different from the bubble sort. Instead of going through the whole array no matter what, the cocktail sort will exit the program as soon as the array is in expected order. However, each for loop is still dependent on the size of $n * n$. Therefore, the average-case time complexity is also $O(n^2)$.

(d) As explained in (a), the minimum number of element comparisons performed by the algorithm is $n-1$ times. Therefore, the worst-case running time of this algorithm is also $\Omega(n)$. O notation and Ω notation don't coincide, so no, the running time of the algorithm can not be expressed in terms of the Θ notation.

□

2. **In-Place.** In place means an algorithm requires $O(1)$ additional memory, including the stack space used in recursive calls. Frankly speaking, even for a same algorithm, different implementation methods bring different in-place characteristics. Taking *Binary Search* as an example, we give two kinds of implementation pseudo codes shown in Alg. 2 and Alg. 3. Please analyze whether they are in place.

   Next, please give one similar example regarding other algorithms you know to illustrate such phenomenon.

3. **Master Theorem**.

   Definition 1 (Matrix Multiplication)**.** *The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry*

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

   Z_{ij} is the dot product of the ith row of X with jth column of Y. The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication.

Alg. 2: BinSearch($a[\cdot]$, x, low, $high$)

Input : a sorted array a of n elements, an integer x, first index low, last index $high$
Output: first index of key x in a, -1 if not found

```
1 if high < low then
2     return -1;
3 mid ← low + ((high − low)/2);
4 if a[mid] > x then
5     mid ← BinSearch(a, x, low, mid − 1);
6 else if a[mid] < x then
7     mid ← BinSearch(a, x, mid + 1, high);
8 else
9     return mid;
```

Alg. 3: BinSearch($a[\cdot]$, x, low, $high$)

input : a sorted array a of n elements, an integer x, first index low, last index $high$
output: first index of key x in a, -1 if not found

```
1 while low ≤ high do
2     mid ← low + ((high − low)/2);
3     if a[mid] > x then
4         high ← mid − 1;
5     else if a[mid] < x then
6         low ← mid + 1;
7     else
8         return mid;
9 return -1;
```

Solution. Algorithm 2 is recursive. We have to consider stack space, which has the space complexity of $O(\log n)$. Therefore, Alg. 2 is not in place.
Algorithm 3 uses *mid*, *high*, *low* three variables to store data no matter the input size. Its space complexity is $O(1)$. Therefore, Alg. 3 is in place.
A similar example would be the algorithm which calculates the factorial of a non-negative integer n. We can implement the algorithm in an recursive or a non-recursive style. The corresponding pseudo codes are shown in Alg. 4 and Alg. 5.
Algorithm 4 is recursive. We have to consider stack space and thus the space complexity is $O(n)$. Therefore, Alg. 4 is not in place.
Algotirhm 5 uses one variable *res* to store data no matter the input size. Its space complexity is $O(1)$. Therefore, Alg. 5 is in place.

□

Alg. 4: Factorial(n)

Input : a non-negative integer n
Output: $n!$

```
1 if n ≤ 1 then
2 |  return 1;
3 else
4 |  return n * Factorial(n − 1);
```

Alg. 5: Factorial(n)

input : a non-negative integer n
output: $n!$

```
1 if n ≤ 1 then
2 |  return 1;
3 res ← 1;
4 for i from 1 to n do
5 |  res ← res * i;
6 return res;
```

In 1969, the German mathematician Volker Strassen announced a siginificantly more efficient algorithm, based upon divide-and-conquer. Matrix Multiplication can be performed blockwise. To see what this means, carve X into four $\frac{n}{2} \times \frac{n}{2}$ blocks, and also Y:

$$X = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right), \quad Y = \left(\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right).$$

Then their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements.

$$XY = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)\left(\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right) = \left(\begin{array}{c|c} AE+BG & AF+BH \\ \hline CE+DG & CF+DH \end{array}\right).$$

To compute the size-n product XY, recursively compute eight size-$\frac{n}{2}$ products AE, BG, AF, BH, CE, DG, CF, DH and then do a few additions.

(a) Write down the recurrence function of the above method and compute its running time by Master Theorem.

Solution. The recurrence function is $T(n) = 8T(\frac{n}{2})+O(n^2)$. Using the Master Theorem, we have $a = 8$, $b = 2$, $d = 2$, and $a > b^d$. Therefore, $T(n) = O(n^{\log_b a}) = O(n^3)$. □

(b) The efficiency can be further improved. It turns out XY can be computed from just seven $\frac{n}{2} \times \frac{n}{2}$ subproblems.

$$XY = \left(\begin{array}{c|c} P_5+P_4-P_2+P_6 & P_1+P_2 \\ \hline P_3+P_4 & P_1+P_5-P_3-P_7 \end{array}\right),$$

where

$$P_1 = A(F-H), \qquad P_2 = (A+B)H, \qquad P_3 = (C+D)E, \qquad P_4 = D(G-E),$$
$$P_5 = (A+D)(E+H), \qquad P_6 = (B-D)(G+H), \qquad P_7 = (A-C)(E+H).$$

Write the corresponding recurrence function and compute the new running time.

Solution. The recurrence function is $T(n) = 7T(\frac{n}{2})+O(n^2)$. Using the Master Theorem, we have $a = 7$, $b = 2$, $d = 2$, and $a > b^d$. Therefore, $T(n) = O(n^{\log_b a}) = O(n^{\log_2 7})$. □