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:Wu Jiayao Student ID:517370910257 Email: jiayaowu1999@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 is $n-1$

$$n-0-1-0=n-1$$

. It is achieved when the array is already sorted.
(b)The maximum number is $\frac{n^2}{2}-(n\%2)\cdot\frac{1}{2}$.

$$\begin{aligned}\sum_{i=0}^{\lfloor\frac{n-1}{2}\rfloor} 2\times(n-i-1-i) &= \sum_{i=0}^{\lfloor\frac{n-1}{2}\rfloor} 2\times(n-2i-1)\\ &= \frac{n^2}{2}-(n\%2)\cdot\frac{1}{2}\end{aligned}$$

It is achieved when the array is sorted from the biggest to the smallest number.
(c) $O(n^2)$
(d) No. The best case is $O(n)$ and the worst case is $O(n^2)$. Hence, neither $\Theta(n)$ nor $\Theta(n^2)$ can express the running time. □

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

if $high < low$ **then**
    return *-1*;
$mid \leftarrow low + ((high - low)/2)$;
if $a[mid] > x$ **then**
    $mid \leftarrow$ BinSearch$(a, x, low, mid - 1)$;
else if $a[mid] < x$ **then**
    $mid \leftarrow$ BinSearch$(a, x, mid + 1, high)$;
else
    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

while $low \leq high$ **do**
    $mid \leftarrow low + ((high - low)/2)$;
    if $a[mid] > x$ **then**
        $high \leftarrow mid - 1$;
    else if $a[mid] < x$ **then**
        $low \leftarrow mid + 1$;
    else
        return mid;
return *-1*;

Solution. Alg. 2 is not in place. For each recursive step, there will be $O(1)$ additional memory for temp variable *mid*. As there are $O(\log n)$ recursive steps, the additional memory is $O(\log n)$.
Alg. 3 is in place. Only $O(1)$ additional memory is required for temp variable *mid*.
Example:
The merge sort we learn in class, which takes $O(n)$ additional memory, is not in place. Here is another implementation of merge sort.

Alg. 4: MergeSort($a[\cdot]$,$left$,$right$)

input : An array a of n elements,the most $left$ and most $right$ index of the part to sort
output: a sorted array a

```
if left ≥ right then
    return
MergeSort(a[·],left,n/2)
MergeSort(a[·],n/2 + 1,right)
i ← left
j ← n/2 + 1
tmp ← j
while i < right do
    while a[i] < a[j] do
        i++
    while a[i] > a[j] do
        j++
    swap a[i:index] with a[index:j]
    i ← i + j − tmp
```

Only 3 additional variable is used. This MergeSort is in place. □

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. $f(n) = 8f(\frac{n}{2}) + 4 = f(n) = 8f(\frac{n}{2}) + O(1)$. The running time is $O(n^{\log_2 8}) = O(n^3)$. □

(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. $f(n) = 7f(\frac{n}{2}) + 8 = f(n) = 7f(\frac{n}{2}) + O(1)$. The running time is $O^{\log_2 7}$. $\square$