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:________ Student ID:________ Email: __________

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
1  for i = 0; i < n − 1; i++ do
2      bFlag ← true;
3      for j = i; j < n − i − 1; j++ do
4          if a[j] > a[j+1] then
5              swap(a[j], a[j+1]);
6              bFlag ← false;
7      if bFlag then
8          break;
9      bFlag ← true;
10     for j = n − i − 1; j > i; j−− do
11         if a[j] < a[j−1] then
12             swap(a[j], a[j−1]);
13             bFlag ← false;
14     if bFlag then
15         break;
```

Solution. (a) The minimum number of element comparisons is $n-1$ times.

When the given array is already in ascending order, the algorithm only runs $n-1$ times comparisons in line 4.

(b) The maximum number of element comparisons is achieved when the given array is in descending order.

If n is even, the total number of comparisons is $[(n-1)+(n-3)+...+1]\cdot 2 = \frac{n^2}{2}$.

If n is odd, the total number of comparisons is $[(n-1)+(n-3)+...+2]\cdot 2 = \frac{n^2-1}{2}$.

(c) From (b), we can know that the algorithm could be expressed in $O(n^2)$.

(d) No. From (a) and (b), we can know that the best case is $\Omega(n)$, and the worst case is $O(n^2)$, so the running time of the algorithm cannot be expressed in 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.

Solution. Alg. 2 is not in-place. The space complexity is $O(logn)$.

Alg. 3 is in-place. This method uses auxiliary space to store corresponding variables, independent of the size of the problem. So the space complexity is $O(1)$.

Another example is as follows:

Alg. 4 is a recursive funtion, which will take a space on the stack when it is called. So the space complexity is $O(n)$ and the implementation is not in-place.

Alg. 5 only needs an auxiliary space to store result, so the space complexity is $O(1)$ and the implementation is in-place. □

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*;

Alg. 4: Factorial(n)

Input : A natural number n
Output: The value of $n!$

$result \leftarrow 1$;
if $n = 0$ *or* $n = 1$ **then**
  return *1*;
else
  $result \leftarrow n \cdot Factorial(n - 1)$;
return *result*;

Alg. 5: Factorial(n)

Input : A natural number n
Output: The value of $n!$

$result \leftarrow 1$;
while $n \geq 0$ **do**
  $result \leftarrow result \cdot n$;
  $n \leftarrow n - 1$;
return *result*;

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.

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. For problem with size n, there are totally 8 submatrix multiplication with size $\frac{n}{2} \times \frac{n}{2}$. And then do 4 addition of submatrix with size $\frac{n}{2} \times \frac{n}{2}$, which is $O(n^2)$. Therefore, the recurrence function of this method is written as:

$$T(n) = 8 \times T(\frac{n}{2}) + O(n^2)$$

By master theorem, $a = 8$, $b = 2$, $d = 2$, so the running time is $O(n^3)$. □

(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

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

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

Solution. For problem with size n, there are totally 7 submatrix multiplication with size $\frac{n}{2} \times \frac{n}{2}$. As the subtraction and addition take a contant multiple of $O(n^2)$ times, the recurrence function is written as:

$$T(n) = 7 \times T(\frac{n}{2}) + O(n^2)$$

By master theorem, $a = 7$, $b = 2$, $d = 2$, so the running time is $O(n^{\log_2 7}) \approx O(n^{2.81})$. □