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: Jintian Ge   Student ID: 517021911142   Email: gejintian@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 comparisons is $n-1$.

It is achieved when the input array has already been sorted.

(b) The worst running time happens when the for loop is fully executed. When n is odd, the number of element comparisons should be: $2(n-1)+2(n-3)+2(n-5)+\cdots+2\times 2+0=\frac{1}{2}(n^2-1)$. When n is even, the number of element comparisons should be:$2(n-1)+2(n-3)+\cdots+2\times 3+1=\frac{1}{2}n^2$

It is achieved when the input array is in reverse order.

(c) The worst case is the upper bound of the running time of the algorithm.

Since from (b) we know that the running time of the worst case is $\frac{1}{2}(n^2-1)$ for odd n and $\frac{1}{2}n^2$ for even n, the running time of this algorithm is $O(n^2)$.

(d) If it can be expressed in terms of the Θ notation, we need to prove that the best case can be represented as $\Omega(n^2)$.

It is clear that $cn^2 \geq n$ for all $n > c$ regardless how large c is. Therefore, the best case cannot be represented as $\Omega(n^2)$. Hence, the running time cannot be expressed in terms of the Θ notation

$\square$

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. 5. 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 ← low + ((high − low)/2);
if a[mid] > x then
    mid ← BinSearch(a, x, low, mid − 1);
else if a[mid] < x then
    mid ← 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 ≤ high do
    mid ← low + ((high − low)/2);
    if a[mid] > x then
        high ← mid − 1;
    else if a[mid] < x then
        low ← mid + 1;
    else
        return mid;
return -1;
```

Solution. Alg.2 is not in place while Alg.3 is in place.

For Alg.2, it calls stacks as many time as comparisons. So the space complexity is equal to the time complexity, which is $O(\log n)$. Therefore, it is not in place.

For Alg.3, it doesn't call any extra stacks as well as any extra arrays. So, the space complexity of this algorithm is $O(1)$, which indicates that it is in place.

The following is an example based on the dichotomy which finds a zero point of a quadratic function inside a section. Assume the function is $f(x)$. The two different implementations are:

Alg. 4: Dicho($f(x)$, $left$, $right$)

Input : target function $f(x)$, the left side of the section $left$, the right side of the section $right$

Output: one zero point of this function, NaN if not found

if $f(right) == 0$ **then**
    return $right$;
else if $f(left) == 0$ **then**
    return $left$;
else if $f(right) * f(\frac{right+left}{2}) \leq 0$ **then**
    Dicho($f(x)$, $\frac{right+left}{2}$, $right$);
else if $f(left) * f(\frac{right+left}{2})$ **then**
    Dicho($f(x), left, \frac{right+left}{2}) \leq 0$;
else
    return NaN;

Alg. 5: Dicho($f(x)$, $left$, $right$)

input : target function $f(x)$, the left side of the section $left$, the right side of the section $right$

output: one zero point of this function, NaN if not found

while $left \neq right$ **do**
    $mid \leftarrow \frac{right+left}{2}$;
    if $f(mid)f(left) \leq 0$ **then**
        $right \leftarrow mid$;
    else if $f(mid)f(right) \leq 0$ **then**
        $left \leftarrow mid$;
    else if $f(right) == 0$ **then**
        return $right$;
    else if $f(left) == 0$ **then**
        return $left$;
return NaN;

Like binary search, this algorithm break the section into two parts. So, the time complexity is just $O(\log n)$. For the first implementation in Alg.4, it needs space same as time. So its space complexity is $O(\log n)$, which is not in place. For the second implementation is Alg.5, it doesn't call any stack and thus has space complexity $O(1)$, which 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. The algorithm of the function is shown as following:

```
// REQUIRES: two matrices A and B
// EFFECTS: return the result of A multiply with B
mat Matmul(mat A, mat B){
    if(length(A) == 1) return [A*B].
    //If A has only one element, return its multiple result
        with B as a matrix with one element
```

```
    [A1, A2, A3, A4] = SplitMat(A); //SplitMat is used to split A into four parts.
    [B1, B2, B3, B4] = SplitMat(B);
    return AppendMat(Matmul(A1,B1) + Matmul(A2,B3),
                     Matmul(A1,B2) + Matmul(A2,B4),
                     Matmul(A3,B1) + Matmul(A4,B3),
                     Matmul(A3,B2) + Matmul(A4,B4));
    //AppendMat is used to append four matrix together.
}
```

We have four matrix additions in one recurrent process, with each matric containing $(\frac{n}{2})^2 = \frac{n^2}{4}$ elements, so the time complexity of addition is just $O(n^2)$, then we have the following function:

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

According to Master Theorem, we have $a = 8, b = 2$ and $d = 2$. Since $8 > 2^2$, we conclude that the time complexity of this algorithm should be

$$O(n^{\log_2 8}) = O(n^3)$$

.

$\square$

(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. For this time, we only have 7 multiplies and 18 additions/subtractions in this algorithm. The time complexity of these additions/subtractions is still $O(n)$. Therefore, corresponding recurrence function should be:

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

Hence, according to Master Theorem, the new running time is

$$O(n^{\log_2 7})$$

$\square$