# 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:Jin Zhejian    Student ID: 517370910167    Email: jinzhejian@outlook.com

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 $\Theta$ 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 $n-1$. If the original array is in ascending order, the number of element comparisons is the minimum one. It only need to go through the inner **for loop**, which needs $n-1$ comparisons, and **bFlag** is true after the for loop, then the **outer for loop** breaks and the Cocktail Sort is done.

(b) The maximum number of element comparisons is $\lfloor \frac{n^2}{2} \rfloor$, which means $\frac{n^2-1}{2}$ when n is odd, $\frac{n^2}{2}$ when n is even. This maximum achieves when the original array is in decreasing order. In this case, when n is odd, for each i, $n-2i-1$ times of comparisons are needed. So, in total, there needs $(n-1)+(n-1)+(n-3)+(n-3)+......+(2)+(2) = 2\cdot\frac{(n-1+2)\frac{n-1}{2}}{2} = \frac{(n+1)(n-1)}{2} = \frac{n^2-1}{2}$. When n is even, similarly, there needs $(n-1)+(n-1)+......+(3)+(3)+(1)+(1) = \frac{n^2}{2}$.

(c) For the worst case, there needs $\lfloor \frac{n^2}{2} \rfloor$ element comparisons, and swap function don't need element comparisons. Therefore, the time complexity of Cocktail Sort is $O(n^2)$.

(d) No. Because the time complexity for the best case is $\Omega(n)$, which is not of the same order with $n^2$. Therefore, the running time cannot be expressed in terms of the $\Theta$ 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.

**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**
  $mid \leftarrow \text{BinSearch}(a, x, low, mid - 1)$;
**else if** $a[mid] < x$ **then**
  $mid \leftarrow \text{BinSearch}(a, x, mid + 1, high)$;
**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** takes recursion method. In each recursion, the algorithm needs to create an int space to store ***mid*** , and it needs around $\log n$ times of recursion. So, **Alg.2** is not in place, its space complexity is $O(\log n)$.

**Alg.3** takes traversal method. It only needs to require $O(1)$ additional memory to save ***mid***, and in every loop, ***mid*** is replaced by new ***mid***. So, **Alg.3** is in place.

Similar example:

**Alg. 4:** SumArray(a[·], n)

**Input:** an array with n numbers
**Output:** get the sum of each numbers in the array

**if** n = 0 **then**
  return 0;
**else**
  **return** SumArray(a[·], n-1) + a[n -1];
**end if**

**Alg. 5:** SumArray(a[·], n)

**Input:** an array with n numbers
**Output:** get the sum of each numbers in the array

sum ← 0;
**for** each $i \in [0, n-1]$ **do**
  sum ← sum + a[i];
**end for**
**return** sum;

**Alg.4** uses recursive method, its space complexity is $O(n)$ ,while **Alg.5** uses traversal method, which is in place.

□

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 $i$th row of $X$ with $j$th 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 each blockwise, we need to calculate 8 times of multiplications with size-$\frac{n}{2}$ and 4 additions of size-$\frac{n}{2}$ matrices. Therefore, the recurrence function is:

$$T(n) = 8 \cdot T(\frac{n}{2}) + 4 \cdot (\frac{n}{2})^2 = 8 \cdot T(\frac{n}{2}) + n^2$$

$a = 8, b = 2, d = 2, a > b^d$, so by Master Theorem:

$$T(n) = O(n^{\log_b a}) = O(n^3)$$

□

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

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

where

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

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

**Solution.**

For each blockwise, we need to calculate 7 times of multiplications with size-$\frac{n}{2}$ and 10 additions of size-$\frac{n}{2}$ matrices. Therefore, the recurrence function is:

$$T(n) = 7 \cdot T(\frac{n}{2}) + 10 \cdot (\frac{n}{2})^2 = 7 \cdot T(\frac{n}{2}) + \frac{5}{2} \cdot n^2$$

$a = 7, b = 2, d = 2, a > b^d$, so by Master Theorem:

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 7})$$

$\square$