1 Sorting Problem

Problem 1. With input: Array $A$ of $n$ integers, come up with output: Array $B$ with same entries as $A$, but in increasing order.

There are many typical methods for sorting, for example selection sort, bubble sort, insertion sort etc. Here, we briefly review one of these algorithms.

Definition 2. Selection sort finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. The algorithm can be implemented as follows

\[
\text{for } i = 0 \text{ to } n - 2 \\
\qquad \text{for } j = i + 1 \text{ to } n - 1 \\
\qquad \quad \text{if } A[j] < A[i] \\
\qquad \qquad \text{swap}(A[i], A[j])
\]

The number of operations can be counted:

<table>
<thead>
<tr>
<th>$i$</th>
<th>#swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>1</td>
<td>$n - 2$</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>$i$</td>
<td>$n - i - 1$</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>$n - 2$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Number of swaps in each step

with $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, it can be summarized as $O(n^2)$. 

2 Merge Sort

**Definition 3.** Merge sort is a divide and conquer algorithm. It splits the disordered array into two halves, recursively sort each half and merge the two sorted subarrays.

The algorithm can be represented as:

```java
mergesort(A[0, ⋯, n − 1])
if n > 1
    A1 = mergesort(A[0, ⋯, ⌊n/2⌋])
    A2 = mergesort(A[⌈n/2⌉, ⋯, n − 1])
    B = merge(A1, A2)
    return B
else
    return A
```

The pseudocode of merge part is:

```java
merge(A[0, ⋯, n − 1], B[0, ⋯, m − 1])
Let C = zeros[m + n]
Let p = q = 0
while p < n or q < m
    if A[p] > B[q]
        C[p + q] = B[q ++]
    else
        C[p + q] = A[p ++]
    if p ≥ n
        while q < m
            C[p + q] = B[q ++]
    else
        while p < n
            C[p + q] = A[p ++]
    return C
```

The merge procedure does a constant amount of work per recursive call, for a total running time of $O(n + m)$ in the above case. Assume

$$T(n) = \text{time required to sort an array of length n using mergesort}$$

$$T(1) = O(1)$$

then, according to the algorithm, for $n > 1$

$$T(n) = 2T\left(n/2\right) + O(n)$$

$$= 2\left(2T\left(n/4\right) + O\left(n/2\right)\right) + O(n) = 4T\left(n/4\right) + O(n) + O(n)$$

$$= 4\left(2T\left(n/8\right) + O\left(n/4\right)\right) + O(n) + O(n) = 8T\left(n/8\right) + O(n) + O(n) + O(n)$$
Make a guess about the solution:

\[ T(n) = 2^k T\left(\frac{n}{2^k}\right) + kO(n) \]

and prove it by mathematical induction

\[
T(n) = 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (k - 1) O(n)
= 2^{k-1} \left(2^k T\left(\frac{n}{2^k}\right) + O\left(\frac{n}{2^{k-1}}\right)\right) + (k - 1) O(n)
= 2^k T\left(\frac{n}{2^k}\right) + O(n) + (k - 1) O(n)
= 2^k T\left(\frac{n}{2^k}\right) + kO(n)
\]

conclusion is proved.

Assume \( n = 2^k \) which is equivalent to \( k = \log n \), then

\[ T(n) = 2^k T\left(\frac{n}{2^k}\right) + kO(n) = nO(1) + \log nO(n) = O(n \log n) \]

the operation time of mergesort algorithm is \( O(n \log n) \).

This result can also be obtained directly from following graph

![Diagram](image)

Figure 2.1: Each problem of size \( n \) is divided into a subproblem of size \( n/b \)

In mergesort case, we have \( a = b = 2 \). There are \( \log n \) levels in total, and there are \( 2^k O\left(\frac{n}{2^k}\right) = O(n) \) operations in the \( k \)th level. Thus, the operation time is \( O(n \log n) \).
3 Quick Sort

Definition 4. Quick sort also applies divide and conquer method. It pick an element of the array as pivot and rearrange the array such that elements in left part of pivot is less than pivot and these from right part is larger than pivot. Then, use the same method to recursively sort both left and right part. In this process, pivot always locates in the right position.

The following procedure implements quicksort:

\[
\text{quicksort} \left( A[0, \cdots, n-1] \right) \\
\text{if } n > 1 \\
\quad p = \text{Choose pivot from } A[0, \cdots, n-1] \\
\quad \text{partition} \left( A[0, \cdots, n-1], p \right) \\
\quad \text{recursively quicksort left part} \\
\quad \text{recursively quicksort right part} \\
\quad \text{return } A \\
\text{else} \\
\quad \text{return } A
\]

The key idea of quick sort is partition array around pivot element. The partition algorithm is

\[
\text{partition} \left( A[0, \cdots, n-1], p \right) \\
l = 0, r = n - 1 \\
\text{while } l < r \\
\quad \text{if } A[l] \geq p \text{ and } A[r] \leq p \\
\quad \quad \text{swap} \left( A[l+1], A[r-1] \right) \\
\quad \text{else if } A[l] \geq p \text{ and } A[r] \geq p \\
\quad \quad r -- \\
\quad \text{else if } A[l] \leq p \text{ and } A[r] \leq p \\
\quad \quad l ++ \\
\quad \text{else} \\
\quad \quad l ++, r ++
\]
Appendix:

We introduce another algorithm to do partition. In this partition algorithm, we just pick the first element in the array as pivot and use \( l \) to denote the location of last element.

\[
\text{partition}(A[0, \ldots, n - 1])
\]
\[
p = A[0]
\]
\[
i = 1
\]
\[
\text{for } j = 1 \text{ to } n - 1
\]
\[
\text{if } A[j] < p
\]
\[
\text{swap}(A[j], A[i])
\]
\[
i++
\]
\[
\text{swap}(A[0], A[i-1])
\]

Figure 3.1: The partition process