
7205 HW2

Name: Xuebao Zhao NUID: 002108354

Q1:

The program implements the MERGE-SORT and MERGE algorithms. It first asks for an integer n . The n is the amount of the element in the array. Then fill the array using random values. Sort the array using merge algorithms. Then print the contents of the array before and after calling sort on it.

The modify is to compare the $A[p]$ and $A[p+1]$ at the beginning of the function Merge. Because the two subarrays are both orderly, If $A[p]$ which is the greatest one of the precious array is smaller than $A[p+1]$ which is the smallest one of the other subarray, the array A is orderly.

result:

```
Please input the value n: 6
The array before sort:78  98  69  12  2  78
The array after sort:2  12  69  78  78  98
-bash-4.2$
```

```
Please input the value n: 61
Please input the value n between 1 and 51! Try again: 10
The array before sort:36  47  92  52  94  61  6  28  28  55
The array after sort:6  28  28  36  47  52  55  61  92  94
```

Q2:

MERGE-SORT(A, p, r)

 if $p < r$

$q = \lfloor (p + r) / 2 \rfloor$

 MERGE-SORT(A, p, q)

 MERGE-SORT($A, q + 1, r$)

 MERGE(A, p, q, r)

MERGE(A, p, q, r)

 If $A[q] \leq A[q + 1]$

 exit the function

 else

$n1 = q - p + 1$

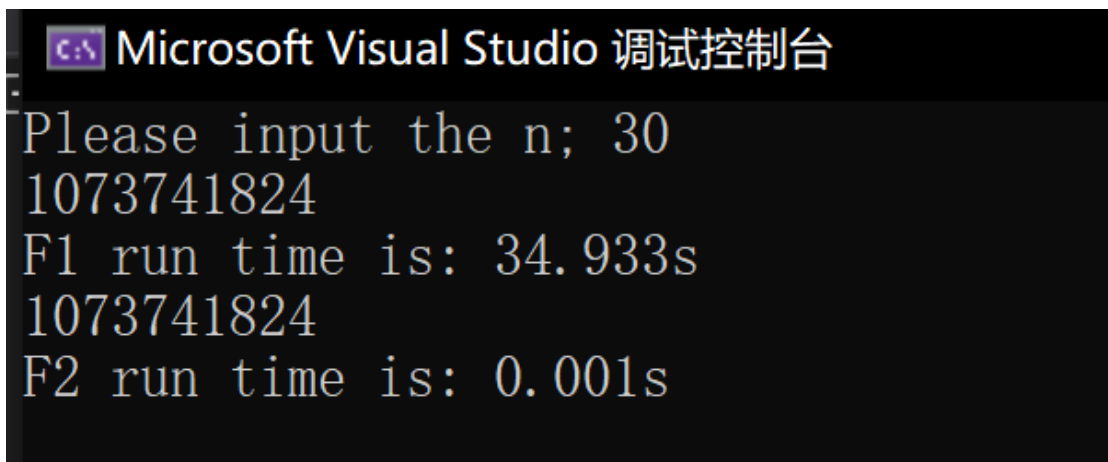
```

n2 = r - q
Let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
i = 1
j = 1
for k = p to r
    if i > n1
        A[k] = R[j++]
    else if j > n2
        A[k] = L[i++]
    else if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1

```

Q3:

- Both functions are the nth power of 2.
- Function F2 is faster, the following image is the time of the two functions when n is 30. F1 used about 34 seconds while F2 used about 0.001 seconds.



```

Microsoft Visual Studio 调试控制台
Please input the n; 30
1073741824
F1 run time is: 34.933s
1073741824
F2 run time is: 0.001s

```

c.

F1:

```

int F1(int n)
{
    if (n == 0) return 1;
    return F1(n-1) + F1(n-1);
}

```

$T(n) = \begin{cases} C & n=0 \\ 2T(n-1) & n>0 \end{cases}$
 $T(n) = 2T(n-1) = 2 \times (2T(n-2)) = 2^2 T(n-2)$
 $= 2^k T(n-k) = \dots = 2^n T(0) = O(2^n)$

F2:

```

int F2(int n)
{
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int result = F2(n/2);
        return result * result;
    }
    else
        return 2 * F2(n-1);
}

```

$T(n) = \begin{cases} C & n=0 \\ T(n/2) + C & n \% 2 = 0 \text{ and } n > 0 \\ T(n-1) + C & n \% 2 = 1 \text{ and } n > 0 \end{cases}$

The worst case: $n = 2^k - 1$

$T(n) = T(2^k - 1)$
 $= T(2^{k-1} - 1) + C$
 $= T(2^{k-2} - 1) + 2C$
 $= T(2^{k-3} - 1) + 3C$
 $= \dots$
 $= T(2^0 - 1) + (2k+1)C$
 $= (2k+1)C = (2\log_2(n+1) + 1) \cdot C$
 $= 2C \log_2(n+1) + C = O(\log n)$

The worst case of function F2 is much better than F1. So, F2 is faster.

Q4:

a) Function procedureX is bubble sort.

It compares the adjacent elements from the end of the array. Exchange the adjacent elements if the previous number is greater than the latter number. So, the first element will be the smallest number. Repeat the above steps to move the second small number to the second position of the array. Repeat until the array is orderly.

b)

worst case: The array is decreasing.

ProcedureX(A)

```

for i = 1 to A.length - 1
    for j = A.length down to i+1
        if A[j] > A[j-1] (It will be true everytime)
            exchange A[j] with A[j-1]

```

$T(n) = [(n-1) + (n-2) + (n-3) + \dots + n - (n-1)] \cdot C + C \times (n-1)$
 $= [(n-1) + (n-2) + (n-3) + \dots + 1] \cdot C + C(n-1)$
 $= C \frac{(n-1+1) \cdot (n-1)}{2} + C(n-1) = \frac{C}{2} n^2 + \frac{C}{2} n - C$
 $= O(n^2)$

Q5:

Insertionsort(A, n)

```

{
    if n > 1

```

Insertionsort(A, n-1)
REC-INSERTION(A, n)

}

$$\begin{aligned} & \text{Insertionsort}(A, n) \\ & \{ \text{ if } n > 1 \\ & \quad \text{Insertionsort}(A, n-1) \quad T(n-1) \\ & \quad \text{REC-INSERTION}(A, n) \quad cn \\ & \} \\ & T(n) = \begin{cases} c & \text{if } n=1 \\ T(n-1) + cn & \text{if } n > 1 \end{cases} \\ & T(n) = T(n-1) + cn \\ & = T(n-2) + [n + (n-1)] \cdot c \\ & = T(n-3) + [n + (n-1) + (n-2)] \cdot c \\ & \vdots \\ & = T(1) + [n + (n-1) + (n-2) + \dots + 2] \cdot c \\ & = [n + (n-1) + (n-2) + \dots + 2 + 1] \cdot c \\ & = \frac{(n+1) \times n}{2} \cdot c = \frac{c}{2} (n^2 + n) = O(n^2) \end{aligned}$$