VO TRAN QUANG TUAN
18127248
18127248@student.hcmus.edu.vn

Homework Assignment 3
Introduction To Design And Analysis Of Algorithms, Fall 2020

2020-12-02

Contents

1 Problem 1

Find both the smallest and largest values in an array of size n.

1.1 Problem 1.a

Analysis the complexity of the divide-and-conquer algorithm presented in class without using the Master theorem. Assuming that the comparisons are the basic operations.

Algorithm Find both smallest and largest values in an array of size n

procedure FINDMAXMIN($A[1 \dots N], left, right, min, max$) ▷ Find both max and min values of array A
  if $left \geq right - 1$ **then**
    if $A[left] < A[right]$ **then**
      $min \leftarrow A[left]$
      $max \leftarrow A[right]$
    else
      $min \leftarrow A[right]$
      $max \leftarrow A[left]$
    end if
  else
    $mid \leftarrow \left\lfloor \frac{left+right}{2} \right\rfloor$
    FINDMAXMIN($A, left, mid, minL, maxL$)
    FINDMAXMIN($A, mid + 1, minR, maxR$)
    $min \leftarrow (minL < minR)\ ?\ minL : minR$
    $max \leftarrow (maxL > maxR)\ ?\ maxL : maxR$
  end if
end procedure

To begin with the analysis of this algorithm, we must specify that:

- Size of input: array of n elements.
- Basic operations: the comparison between $a[l]$ and $a[r]$ in if condition, between $minL$ and $minR$, between $maxL$ and $maxR$.
- The distribution of data doesn't effect to the time complexity of this algorithm.

Let $C(n)$ is the time complexity of this algorithm, we get the following recurrence:

$$C(n) = C(\left\lfloor \frac{n}{2} \right\rfloor) + C(n - \left\lfloor \frac{n}{2} \right\rfloor) + 2, n > 2$$
$$C(n) = 1, n \leq 2$$

Assume $n = 2^k$, use the Backward substitution to solve this recurrence:

$$\begin{aligned} C(2^k) &= 2 \cdot C(2^{k-1}) + 2 \\ &= 2 \cdot [2 \cdot C(2^{k-2}) + 2] + 2 = 2^2 \cdot C(2^{k-2}) + 2^2 + 2^1 \\ &= 2^2 \cdot [2 \cdot C(2^{k-3}) + 2] + 2^2 + 2^1 = 2^3 \cdot C(2^{k-3}) + 2^3 + 2^2 + 2^1 \\ &= \dots = 2^i \cdot C(2^{k-i}) + 2^{i+1} - 2 \end{aligned}$$

Base on the initial condition $C(n) = 1$ with $n \leq 2$, we choose $n = 1$. Let's set $2^{k-i} = 1$ so we have $k = i$. Therefore, we see that: $C(2^k) = 2^k + 2^{k+1} - 2 = 3 \cdot 2^k - 2$. Come back with $n = 2^k$, the time complexity $C(n) = 3 \cdot n - 2 \in \Theta(n)$

1.2 Problem 1.b

Design an equivalent loop algorithm to solve this problem.

Description: Assign the min value and max value is the first element of this array. After that, iterate array at the second element and compare these values with every element.

Pseudo code:
To begin with the analysis of this algorithm, we must specify that:

Algorithm Find both smallest and largest values in an array of size n with loop and time complexity is $\Theta(n)$

```
procedure FINDMAXMINLOOP(A[1 ... N])
    min ← A[1]
    max ← A[1]
    for i = 2, ..., N do
        if A[i] > max then
            max ← A[i]                          ▷ Update max
        else
            if A[i] < min then
                min ← A[i]                      ▷ Update min
            end if
        end if
    end for
    return max, min
end procedure
```

- Size of input: array of n elements.
- Basic operations: the compare operations in two if conditions in the algorithm. They are $A[i] > max$ and $A[i] < min$.
- The arrangement of data affects to the time complexity of this algorithm, so we must analyze in three cases: best case, worst case and average case.

Let $f(n)$ is the time complexity of this algorithm.

Firstly, with the best case is the array was sort in ascending order, so only the first if condition $A[i] > max$ is performed. With remaining $n-1$ elements, this condition is performed $n-1$ times. So $f(n) = n - 1 \in \Theta(n)$.

Secondly, with the worst case is the array was sort in descending order, so two if conditions $A[i] > max$ and $A[i] < min$ are performed. With remaining $n-1$ elements, every condition is performed $n-1$ times. So $f(n) = 2 \cdot (n-1) \in \Theta(n)$.

Finally, with the average case, assume that 50% the first if condition is true ($A[i] > max$), we have the following time complexity:

$$f(n) = \left\lceil \frac{n-1}{2} \right\rceil + 2[(n-1) - \left\lceil \frac{n-1}{2} \right\rceil]$$

With n is odd, $f(n) = \frac{n-1}{2} + 2(n-1) - (n-1) = \frac{3 \cdot (n-1)}{2} \in \Theta(n)$.

With n is even, $f(n) \leq \frac{n-1}{2} + 2(n-1) - (n-1) = \frac{3 \cdot (n-1)}{2} \in \Omega(n)$.

2 Problem 2

Analysis the complexity of Merge Sort algorithm without using the Master theorem in two cases. To begin with the analysis, we must focus on the pseudo code of the algorithm.

Algorithm Merge Sort

```
procedure MERGESORT(a[1...n], low, high)
    if low < high then
        mid ← ⌊(low+high)/2⌋
        MERGESORT(a, low, mid)
        MERGESORT(a, mid + 1, high)
        MERGE(a, low, mid, high)
    end if
end procedure

procedure MERGE(a[1...n], low, mid, high)
    i ← low
    j ← mid + 1
    k ← low
    while i ≤ mid and j ≤ high do
        if a[i] ≤ a[j] then
            buf[k++] ← a[i++]
        else
            buf[k++] ← a[j++]
        end if
    end while
    if i > mid then
        buf[k...high] ← a[j...high]
    else
        buf[k...high] ← a[i...mid]
    end if
    a[low...high] ← buf[low...high]
end procedure
```

2.1 Problem 2.a

Case 1: Consider the comparison to be the algorithm's basic operation.

Let $T(n)$ is the time complexity of this algorithm. Generally, assuming that we need to sort an array of n numbers in ascending order, we can specify some important information for the analysis:

- The size of input: an array with n numbers needs to be sorted.
- The basic operation: the comparison in if condition inner the while loop $a[i] \leq a[j]$.
- The distribution of the input array effects on the time complexity of this algorithm, so we must analyze in two cases: best case, worst case. Can't determine the time complexity in average case.

To begin with the analysis of time complexity, we assume that $n = 2^k$.

Firstly, let's analyze in the best case. Clearly, we can see that, the best case happens when the array was sorted in desired order. Because the basic operation is in merge algorithm, when merging two sub array we must comparing two pair of element in those sub array, with the sorted array, number of time compare operation is performed is length of the shorter array. So we get the following recurrence for the running time $T(n)$:

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + \left\lfloor \frac{n}{2} \right\rfloor, n > 1$$
$$T(1) = 0$$

With $n = 2^k$, using the Backward substitution to solve this recurrence:

$$\begin{aligned}
T(2^k) &= 2 \cdot T(2^{k-1}) + 2^{k-1} \\
&= 2 \cdot [2 \cdot T(2^{k-2}) + 2^{k-2}] + 2^{k-1} = 2^2 \cdot T(2^{k-2}) + 2 \cdot 2^{k-1} \\
&= 2^2 \cdot [2 \cdot T(2^{k-3}) + 2^{k-3}] + 2 \cdot 2^{k-1} = 2^3 \cdot T(2^{k-3}) + 3 \cdot 2^{k-1} \\
&= \ldots = 2^i \cdot T(2^{k-i}) + i \cdot 2^{k-1}
\end{aligned}$$

Base on the initial condition: $T(1) = 0$, let's set $2^{k-i} = 1$ and we have $k = i$. Therefore, we see that: $T(2^k) = k \cdot 2^{k-1} = \frac{k \cdot 2^k}{2}$.

We also have $k = \log_2 n$, so $T(n) = \frac{n \cdot \log_2 n}{2} \in \Theta(n \cdot \log n)$.

Secondly, let's analyze in the worst case. Clearly, we can see that the worst case happens when the number of comparison is maximum, two sub array were compare together and the index i and j in while loop scan to the end of these sub array, so the number of comparison is $n - 1$. So we get the following recurrence for the running time $T(n)$:

$$\begin{gathered}
T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + (n-1), n > 1 \\
T(1) = 0
\end{gathered}$$

With $n = 2^k$, using the Backward substitution to solve this recurrence:

$$\begin{aligned}
T(2^k) &= 2 \cdot T(2^{k-1}) + 2^k - 1 \\
&= 2 \cdot [2 \cdot T(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 = 2^2 \cdot T(2^{k-2}) + 2 \cdot 2^k - (2^1 + 2^0) \\
&= 2^2 \cdot [T(2^{k-3}) + 2^{k-2} - 1] + 2 \cdot 2^k - (2^1 + 2^0) = 2^3 \cdot T(2^{k-3}) + 3 \cdot 2^k - (2^2 + 2^1 + 2^0) \\
&= \ldots = 2^i \cdot T(2^{k-i}) + i \cdot 2^k - (2^i - 1)
\end{aligned}$$

Base on the initial condition: $T(1) = 0$, let's set $2^{k-i} = 1$ and we have $k = i$. Therefore, we see that: $T(2^k) = k \cdot 2^k - (2^k - 1)$.

We also have $k = \log_2 n$, so $T(n) = n \cdot \log_2 n - n + 1 \in \Theta(n \cdot \log n)$

2.2 Problem 2.b

Case 2: Consider the data movement to be the algorithm's basic operation.

Let $T(n)$ is the time complexity of this algorithm. Generally, assuming that we need to sort an array of n numbers in ascending order, we can specify some important information for the analysis:

- The size of input: an array with n numbers needs to be sorted.
- The basic operation: the data movement (assigning buf and a).
- The distribution of the input array doesn't effect on the time complexity of this algorithm because we must move n numbers.

To begin with the analysis of time complexity, we assume that $n = 2^k$. We have the following recurrence for the running time $M(n)$:

$$\begin{gathered}
M(n) = M(\left\lfloor \frac{n}{2} \right\rfloor) + M(\left\lceil \frac{n}{2} \right\rceil) + n, n > 1 \\
M(1) = 0
\end{gathered}$$

With $n = 2^k$, using the Backward substitution to solve this recurrence:

$$\begin{aligned}
M(2^k) &= 2 \cdot M(2^{k-1}) + 2^k \\
&= 2 \cdot [2 \cdot M(2^{k-2}) + 2^{k-1}] + 2^k = 2^2 \cdot M(2^{k-2}) + 2 \cdot 2^k \\
&= 2^2 \cdot [M(2^{k-3}) + 2^{k-2}] + 2 \cdot 2^k = 2^3 \cdot M(2^{k-3}) + 3 \cdot 2^k \\
&= \ldots = 2^i \cdot M(2^{k-i}) + i \cdot 2^k
\end{aligned}$$

Base on the initial condition: $M(1) = 0$, let's set $2^{k-i} = 1$ and we have $k = i$. Therefore, we see that: $M(2^k) = k \cdot 2^k$.

We also have $k = \log_2 n$, so $M(n) = n \cdot \log_2 n \in \Theta(n \cdot \log n)$

3 Problem 3

Modify the Merge Sort algorithm that run in linear time.

3.1 Problem 3.a

Focus on the pseudo code of merge sort in the lecture, we can specify that:

- The algorithm divide the array despite the arrangement of data input, so if the data input is desired sorted, the time complexity is still $\Theta(n \log n)$.
- We can use some method to handle this case, decrease the time complexity to $\Theta(n)$ if the array is sorted. We call its name Natural Merge Sort.

Idea: this algorithm define the sub array with is sorted, and merge them. So if the array is sorted, the algorithm doesn't divide it. The time complexity for traversing the array to find the sub array is sorted is $\Theta(n)$ with array is presorted.

3.2 Problem 3.b

Algorithm Update Merge Sort can run in linear time

```
procedure NATURALMERGESORT(a[1...n])
    left ← 1
    right ← n
    sorted ← false
    l ← 1
    r ← right
    do
        sorted ← true
        left ← 1
        while left < right do
            l ← left
            while l < right and a[l] ≤ a[l + 1] do
                l++
            end while
            while r == right − 1 or r < right and a[r] ≤ a[r + 1] do
                r++
            end while
            if r ≤ right then
                MERGE(a, left, l, r)
                sorted ← false
            end if
            left ← r + 1
        end while
    while !sorted
end procedure
```