Algorithm Design and Analysis Assignment 1
Divide and Conquer

Xinmiao Zhang
202028013229129

October 29, 2020

1 Divide and Conquer

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., [0,1,2,4,5,6,7] is an ascending array, then it might be rotated and become [4,5,6,7,0,1,2].) How to find the minimum of a rotated sorted array?
(*Hint*:All elements in the array are distinct.)

For example, the minimum of the rotated sorted array [4,5,6,0,1,2] is 0.

Please give an algorithm with $O(\log n)$ complexity, prove the correctness and analyze the complexity.

1.1 Algorithm Description

1.1.1 Method

Given a *rotated sorted array* A and its sub-array $A[l\,..\,r]$, which starts from the lth elements of A and ends with the rth element. If we know that the minimum of A lies in $A[l\,..\,r]$, we do the following operations:

1. If $l \geq r-1$, then return $min\{A[l], A[r]\}$;
2. Let $m = \lfloor \frac{l+r}{2} \rfloor$, judge whether m is the minimum value of A, If so, return $A[m]$;
3. Otherwise, if $A[l] > A[m]$, then recursively process $A[l\,..\,m]$ and return its process result; else recursively process $A[m\,..\,r]$ and return its result.

At the very beginning, set $l = 0$ and $r = n-1$.

1.1.2 Pseudo code

First of all, we design Find-Min-Rotated-Array(A, l, r) as the recursive algorithm operating on the subarray $A[l\,..\,r]$.

As the described before, our inputs are a *Rotated Sorted Array* A, and left/right index of subarray l and r. The algorithm is shown as below.

```
FIND-MIN-ROTATED-ARRAY(A, l, r)
if l ≥ r − 1
    return min{A[l], A[r]}
m = ⌊(l+r)/2⌋
if A[m − 1] > A[m]
    return A[m]
if A[l] > A[m]
    return FIND-MIN-ROTATED-ARRAY(A, l, m)
else
    return FIND-MIN-ROTATED-ARRAY(A, m, r)
```

In normal cases, we simply call FIND-MIN-ROTATED-ARRAY$(A, 0, n-1)$ where $n = A.\mathit{length}$. However, when the rotation pixel is at position 0, A is an ascending array, which is not suitable to use our bisection-based method.

Fortunately, this situation is easy to be settled by adding a simple $if-else$ logic. And the main function FIND-MIN-MAIN(A) is shown as below:

```
FIND-MIN-MAIN(A)
n = A.length
if A[0] < A[n − 1] // the array is rotated at pivot 0.
    return A[0]
else
    FIND-MIN-ROTATED-ARRAY(A, 0, n − 1)
```

Until now, I have given my algorithm description in both ways. In the following parts I will show you Subproblem Reduction Graph, correctness and time-consuming complexity of this algorithm in detail.

1.2 Subproblem Reduction Graph

Clearly, the algorithm uses a classical *Divide-and-Conquer* framework to find solution to original problem. In this section, I want to utilize a toy example to display how the algorithm runs and how it reduces problem space step-by-step.

Considering about the effect of showing, we don't judge whether $A[m]$ is the minimum value in the recursive process. In other words we let algorithm end up naturally. (However, this in-process judgement has good boosting effect.)

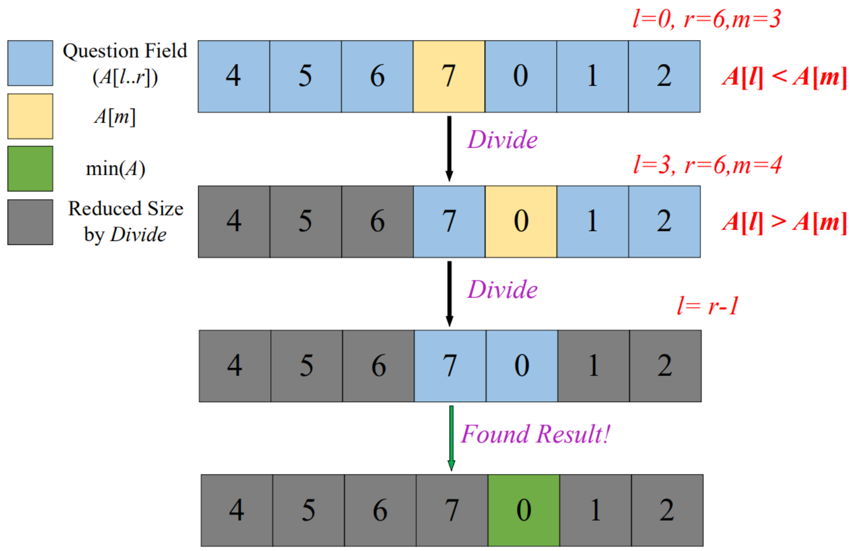

Figure 1: Subproblem Reduction Graph of Problem 1

In Figure 1, we can see clearly that by taking a *Divide* step, the problem size is reduced to half.

1.3 Correctness Proving

Before proving, we made some appointments in this problem. By rotation, A is divided into two sub-arrays. We call the subarray which has smaller index as **head** array, and other elements form **tail** array. Set k as the index of minimum element in A. It's obvious that $A[k]$, A_{head} and A_{tail} have some special features.

1. Unique feature of $A[k]$: $A[k-1] < A[k]$
2. $\forall A[i]$ in A_{head} and $\forall A[j]$ in A_{tail}, $A[i] > A[j]$
3. When $A[k] \in A[l \mathinner{..} r]$ (assuming $l \neq k$), we have $A[l] \in A_{head}$ and $A[r] \in A_{tail}$, which means $A[l] > A[r]$. And if a subarray $A[l \mathinner{..} r](l \neq k)$ includes $A[k]$, it is necessary that $A[l] > A[r]$.

The first two features seems trivial. For the last feature's necessacity, if $A[l] < A[r]$, then according to feature(2) they must in the same set of array, which do not include $A[k]$.

Lemma 1. FIND-MIN-ROTATED-ARRAY(A, l, r) *always guarantees that* $A[k] \in A[l \mathinner{..} r]$.

Proof. We use the mathematical induction methods to prove.

1. At the very beginning, $A[k] \in A[0 \,..\, n-1]$.
2. Assuming that $A[k] \in A[l \,..\, r]$, set $m = \frac{l+r}{2}$.
3. If $A[m] = A[k]$ which can be recognized according to feature(1), then simply return the result.
4. Otherwise, if $A[l] < A[m]$, then according to feature(3), $A[l \,..\, m]$ do not include $A[k]$. And we can know from feature(2), $A[m] \in A_{head}$. And because $A[r] \in A_{tail}$, according to feature(2), $A[m] > A[r]$. So $A[k] \in A[m \,..\, r]$ according to feature(3). Then we will operate on $A[m \,..\, r]$ if $A[l] > A[m]$, then according to feature(3), $A[l \,..\, m]$ includes $A[k]$. Then we will operate on the $A[l \,..\, m]$.

Notice: We always have $A[l] \in A_{head}$. □

Then when we recursively call this function, finally we will reduce the question to compare no more than two elements size which is trivial.

1.4 Algorithm Analysis

We can easily write the recursion of algorithm:

$$T(n) = T(n/2) + 1$$

The recursion tree has $\log_2 n$ levels, on every level there is only one node, and time-complexity of this node is $O(1)$. So it's easy to calculate out $T(n)$ as follows:

$$T(n) = O(\log n)$$

2 Divide and Conquer

Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v.You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a local minumum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v.

Show how to find a local minumum of T using only $O(\log n)$ probes to nodes of T.

2.1 Algorithm Description

It's trivial to see that this binary tree is an *Full Binary Tree*.

2.1.1 Method

Given a n-node *complete binary tree* T. ($n = 2^d - 1$) and the nodes in T are indicated as $v_1, v_2, .., v_n$, where the left child of v_i is v_{2i} and its right child is v_{2i+1}. We define the *sub-complete-binary* tree T_k as one of sub-complete-binary-trees of T which roots at v_k. (Clearly, $T = T_1$). For T_k, assuming that we have known x_k, then we do following operations:

1. If $|T_k| = 1$, then return x_k.
2. Let $x_l = \text{PROBE}(v_{2k})$ and $x_r = \text{PROBE}(v_{2k+1})$, if $x_l > x_k$ and $x_r > x_k$, then return x_k.
3. If $x_l \le x_k$, then recursively process T_{2k}, else recursively process T_{2k+1}.

At the very beginning, we will probe x_1 and process T_1.

2.1.2 Pseudo Code

We use a set X to save the value we have probed, T is the original tree which has n nodes, and k is the root of T_k. The algorithm Pseudo-Code is shown below:

FIND-LOCAL-MIN-TREE(T, k)

$x_k = X[k]$
if $n < 2k$
    return x_k
$X[2k] = \text{PROBE}(v_{2k})$
$X[2k+1] = \text{PROBE}(v_{2k+1})$
if $x_k \le X[2k]$
    FIND-LOCAL-MIN-TREE$(T, 2k)$
else if $x_k \le X[2k+1]$
    FIND-LOCAL-MIN-TREE$(T, 2k+1)$
else
    return x_k

And the main function will call this recursive function as following algorithm:

FIND-LOCAL-MIN-MAIN(T)

$X[1] = \text{PROBE}(v_1)$
FIND-LOCAL-MIN-TREE$(T, 1)$

2.2 Subproblem Reduction Graph

It is not difficulty to find that FIND-LOCAL-MIN-TREE(T, k) fits the *Divide-and-Conquer* framework. So how does the algorithm divide problem to subproblems so as to scale-down the difficulty?

Figure 2 gives us a direct knowledge of how algorithm works by a simple example. T is a complete full binary tree, which includes 15 nodes $\{v_1, .., v_{15}\}$. Each node v_i has a different value x_i.

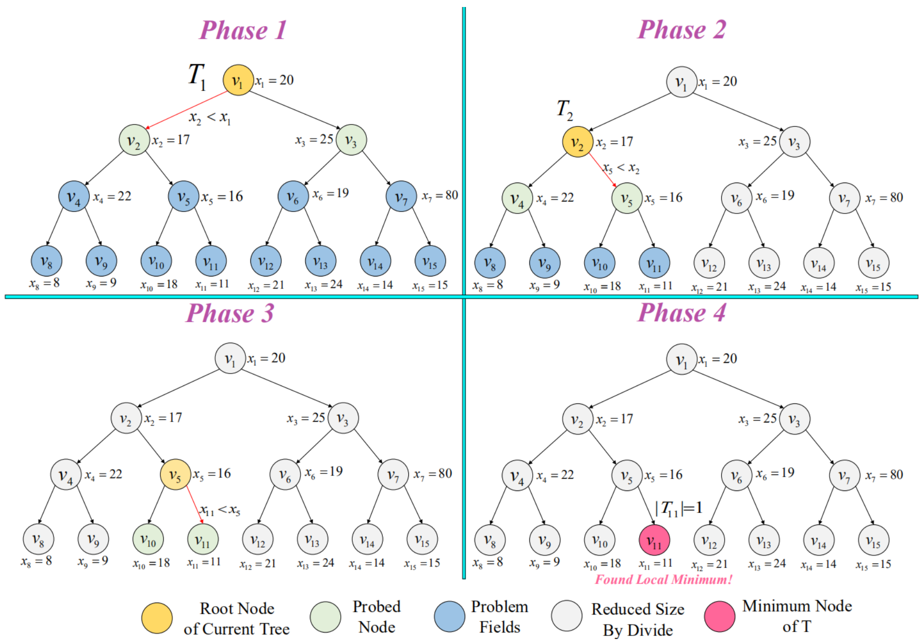

Figure 2: Subproblem Reduction Graph of Problem 2

We can analyze from Figure 2 that Find-Local-Min-Tree can reduce the problem to a subproblem whose question field size is nearly half of the original one.

2.3 Correctness Proving

First of all, we divide all the nodes in T into two disjoint part: *leaf nodes* and *non-leaf nodes*. Obviously there is some difference between them to hold a local minimum value.

For a non-leaf node v_i, if it has a value x_i which is local minimal, then the following conditions are *necessary and sufficient*.

$$\begin{cases} v_{\lfloor i/2 \rfloor} > & v_i, \\ v_{2i} > & v_i, \\ v_{2i+1} > & v_i. \end{cases}$$

For a leaf node v_i, its *necessary and sufficient* condition of becoming local minimum is:

$$v_{\lfloor i/2 \rfloor} > v_i.$$

Our algorithm starts from root node v_1 and goes a *path* p. And all the visited nodes forms a queue $Q_{visited} = \{v_{p_1}, v_{p_2}, .., v_{p_L}\}$, Set L as the length of current p. And $Q_{visited}$ has some features:

Lemma 2. *For* $\forall i(i = 1, 2, .., L-1)$*,* $x_{p_i} \geq x_{p_{i+1}}$*. (i.e.,* $Q_{visited}$ *is monotonically decreasing.*

Proof. If x_{p_i} is visited, $x_{p_{i-1}}$ must bigger than it because of the judge condition in algorithm. If it's not the *minumum* of T. Then according to non-leaf nodes' local minimal condition mentioned before in this section, we have either $x_{2i} \leq v_i$ or $x_{2i+1} \leq v_i$. Without loss of generality, we assume $x_{2i} \leq x_i$, then the algorithm will puts x_{2i} into $Q_{visited}$. In this way we would finally construct $Q_{visited}$, which is *monotonically decreasing*. In other words, Lemma 2 is proved. $\square$

Consider about the worst case in which we finally have to visit leaf node v_d. According to algorithm, we must have $v_{\lfloor d/2 \rfloor} \in Q_{visited}$. So we have $v_d \leq v_{\lfloor d/2 \rfloor}$. (Else $v_{\lfloor d/2 \rfloor}$ must be local minumum) Notice that it perfectly match the *necessary and sufficient* condition where *leaf node* has local minimum of whole tree.

To conclude, the algorithm can end up normally, and it can finally find the local minimum of T.

2.4 Algorithm Analysis

Considering about the worst case, when we have to visit *leaf node*. The recursion tree has $\log n$ levels, and in ith level we at most probe two nodes v_{2i} and v_{2i+1}, so we will at most probe $O(\log n)$ in sum.

3 Divide and Conquer

Given an integer array, one or more consecutive integers in the array form a sub-array. Find the maximum value of the sum of all subarrays.

Please give an algorithm with $O(n \log n)$ complexity.

3.1 Algorithm Description

3.1.1 Method

We will recursively process the subarray $A[l\,..\,r]$ in the following steps:

1. if $l = r$, then return two value ($A[l]$, $A[l]$).
2. Set $m = \lfloor \frac{l+r}{2} \rfloor$, recursively process the *left part* of $A[l\,..\,r](A[l\,..\,m])$, gets return values (M_L, M_{L_m}). M_L is the max value of $A[l\,..\,m]$, and M_{L_m} is the max value including $A[m]$ in array $A[l\,..\,m]$. Similarly, we recursively process the *right part* $A[m+1\,..\,r]$ and gets (M_R, M_{R_m}).
3. Let $M = \max\{M_L, M_R, M_{L_m} + M_{R_m}\}$, if $A[L\,..\,R]$ is A itself, then return M and the algorithm ends. If $A[l\,..\,r]$ is a *left part*, then calculate the max value including $A[r]$ in $A[l\,..\,r]$, else calculate the max value including $A[l]$ in $A[l\,..\,r]$, noted as M_m and return (M, M_m).

3.1.2 Pseudo Code

We show the Pseudo Code of processing the subarray $A[l \,..\, r]$. And input *direction* means current subarray $A[l \,..\, r]$ is divided as the *left/right/no* part of its parent array. (Straightforwardly, when there is *no* divide *direction*, then $A[l \,..\, r] = A$.)

```
FIND-MAX-SUBARRAY-SUM(A, l, r, direction)
if l == r
    return (A[l], A[l])
m = ⌊(l+r)/2⌋
[M_L, M_{L_m}] = FIND-MAX-SUBARRAY-SUM(A, l, m, left)
[M_R, M_{R_m}] = FIND-MAX-SUBARRAY-SUM(A, m+1, r, right)
M = max{M_L, M_R, M_{L_m} + M_{R_m}}
M_m = 0, tmp = 0
if direction == no
    return M
else if direction == left
    for i = r to l
        tmp = tmp + A[i]
        if tmp > M_m then M_m = tmp
        i = i - 1
else
    for i = l to r
        tmp = tmp + A[i]
        if tmp > M_m then M_m = tmp
        i = i + 1
return (M, M_m)
```

And our main algorithm will call the FIND-MAX-SUBARRAY-SUM$(A, 0, n-1, no)$, and the main function algorithm is shown as below:

```
FIND-MAX-SUBARRAY-SUM-MAIN(A)
if A.length == 1
    return A[0]
return FIND-MAX-SUBARRAY-SUM(A, 0, n - 1, no)
```

We repair the trivial situation when A only has one element.

3.2 Subproblem Reduction Graph

To better understand how the algorithm works, we use a toy array $A = \{-6, 7, -9, 20, -8, 3, 0, 1\}$ as an example to show how the *Divide-and-Conquer* algorithm divide the question and finally conquer up to the final solution.

Figure 3 shows all the details. From Figure 3 we can deduce that our algorithm can reduce the original question to half in size in every *dividing* steps.

And the *combining* steps combine the answers of subarray to approach to the final solution to problem.

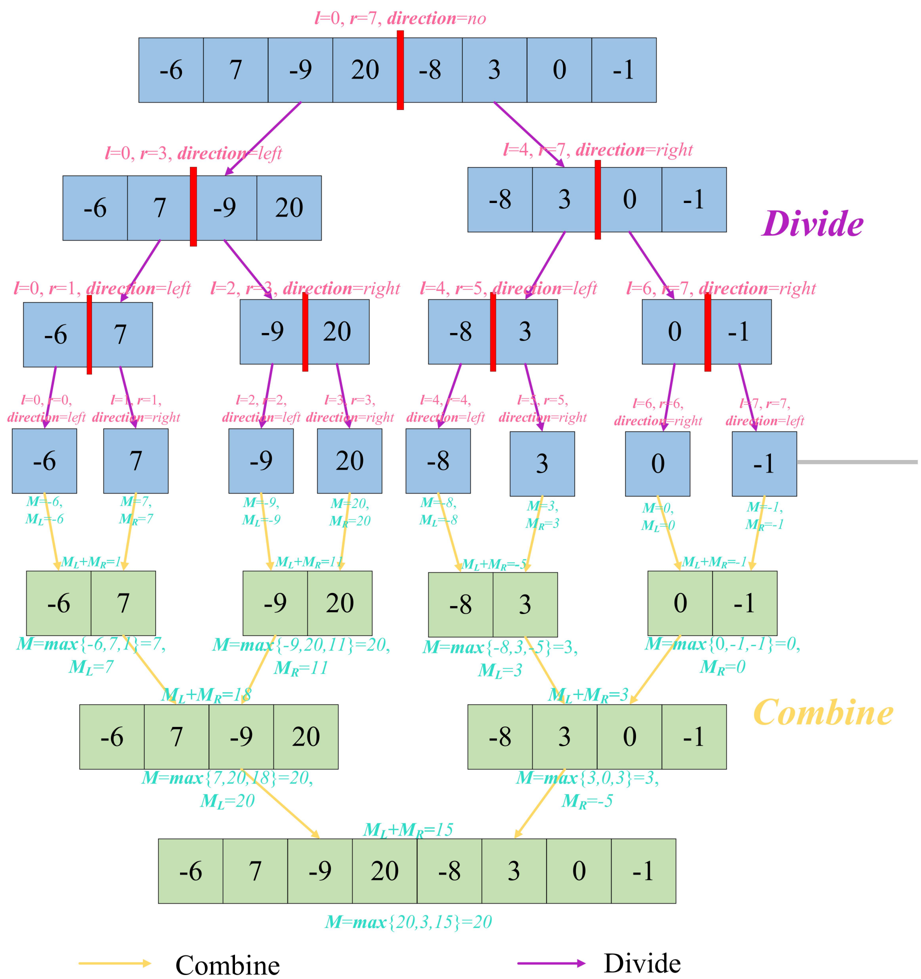

Figure 3: Subproblem Reduction Graph of FIND-MAX-SUBARRAY-SUM

3.3 Correctness Proving

We will prove our $Divide-and-Conquer$ based algorithm by mathematical induction method.

First of all, we have the following observation:

For an array $A[l\,..\,r](r > l)$, if we use $m = \lfloor \frac{l+r}{2} \rfloor$ to divide $A[l\,..\,r]$. A basic fact that if the $A[u, v](l \le u \le v \le r)$ has the maximal sum, then it satisfies one of the following condition:

- $u \le m$ and $v \le m(M_L)$
- $u > m$ and $v > m(M_R)$
- $u \le m$ and $v > m(M_m)$

To sum up, $M = \max\{M_L, M_R, M_m\}$.

For simplity, we define some more variables:

$$M_{L_m} = \max\{\sum_{i=s}^{m} A[i]\}(s = l \dots m)$$

M_{L_m} is the max sum value of subarray including $A[m]$ in $A[l\,..\,m]$. Likewise, we define M_{R_m}:

$$M_{R_m} = \max\{\sum_{i=m+1}^{t} A[i]\}(t = m+1 \dots r)$$

M_{R_m} is the max sum value of subarray including $A[m+1]$ in $A[m+1\,..\,r]$. In like wise, we define M_{R_m}:

Then accoring to the definition of M_m, we have:

$$M_m = M_{L_m} + M_{R_m}$$

.

Here is our proof:

Proof. If $l = r$, then return its max value $A[l]$. Whether it is a left or right part of its parent problem, return another $A[l]$.

Suppose we have already solved out M_L and M_R, M_{L_m} and M_{R_m}.Then we can easily get solution to $A[l\,..\,r]$:

$$M = \max\{M_L, M_R, M_m\}$$

.

If it is the *root problem*, algorithm ends. Otherwise, without loss of generality we suppose current is a left part, using a simple loop can calculate M'_L for its parent.

That means our algorithm can end up normally. □

3.4 Algorithm Analysis

In every recursion function, the combine phase will cost $O(n)$ time. So we can get $T(n)$ recursion as follows:

$$T(n) = 2T(n/2) + O(n)$$

. According to the *Master Theorom*, we have:

$$T(n) = O(n \log n)$$

which fulfills problem bound.