# ps3

221300066Ji Qiankun

June 12, 2024

## 1

**(a)** Inspired by the binary search algorithm,my idea is to find the pivot element, which is the element where the array is rotated.
The algorithm to find k is as fellow:
1. Initialize two pointers, low and high, to the first and last indices of the array respectively.
2. While low is less than high, do the following steps repeatedly:
a. Calculate the middle index as mid = (low + high) / 2.
b. If the middle element is greater than the next element, then the next element is the smallest element and mid + 1 is the value of k. Return mid + 1.
c. If the middle element is smaller than the previous element, then the middle element is the smallest element and mid is the value of k. Return mid.
d. If the middle element is greater than the first element, then the pivot is in the right half of the array. Set low = mid + 1.
e. Otherwise, the pivot is in the left half of the array. Set high = mid - 1.
3. If the loop terminates without finding the pivot, it means the array is not rotated. Return 0.
Runtime complexity of this algorithm is O(log n).
Initialization step takes constant time, O(1).Since it uses a modified binary search approach to find the pivot element,$T(n) = 2T(n/2) + o(1)$.(In each iteration, the array is divided in half by calculating the middle index.)According to the main theorem, the overall runtime complexity is O(log n).
Correctness:
- At each step, I compare the middle element with its adjacent elements to determine if it is the pivot.
- If the middle element is greater than the next element, it means the next element is the smallest element and the pivot is at mid + 1.
- If the middle element is smaller than the previous element, it means the middle element is the smallest element and the pivot is at mid.
- By comparing the middle element with the first element, we can determine which half of the array contains the pivot.
- The algorithm terminates when the pivot is found or when the search space is exhausted, returning 0 if the array is not rotated.

**(b)** The algorithm to search for x:
1. Initialize two pointers, low and high, to the first and last indices of the array respectively.
2. While low is less than or equal to high, do the following steps repeatedly:
a. Calculate the middle index as mid = (low + high) / 2.
b. If the middle element is equal to x, return True.
c. If the first half of the array (from low to mid) is sorted in increasing order, check if x is within that range. If it is, set high = mid - 1. Otherwise, set low = mid + 1.
d. If the second half of the array (from mid to high) is sorted in increasing order, check if x is within that range. If it is, set low = mid + 1. Otherwise, set high = mid - 1.
3. If the loop terminates without finding x, return False.
Runtime complexity of this algorithm is O(log n) since it uses a modified binary search approach to find x(same as (a)).
Correctness:
- At each step, I compare the middle element with x to determine if it is equal.
- If the first half of the array is sorted, I will check if x is within that range. If it is, I will update the search space to the first half.
- If the second half of the array is sorted, I will do the similar thing like last step.
- By dividing the search space in half at each step, I narrow down the range where x can be found.
- The algorithm terminates when x is found or when the search space is exhausted, returning False if x is not found.

# 2

**(a)** The k-th largest element can reside at levels k, k+1, k+2, ..., [n/2].

**(b)** Use mathematical induction to prove.
Base case:
For a heap with only one node (n=1), there are no children, so the statement holds true.
Inductive step:
Assume the statement holds true for a heap with k nodes, where k¿=1. We need to prove that it also holds true for a heap with k+1 nodes.
Let's consider a heap with k+1 nodes. The root of the heap will have two children, which are roots of two subtrees. Let m be the number of nodes in the left subtree and n-k-1-m be the number of nodes in the right subtree (at this time,$k + 1 = 1 + m + (n - k - 1 - m) = n$).
According to our assumption, the left subtree with m nodes contains at most (2m/3) nodes, and the right subtree with n-k-1-m nodes contains at most (2(n-k-1-m)/3) nodes.
The total number of nodes in the left and right subtrees is m + (n-k-1-m) =

n-k-1. To prove that each child of the root is the root of a subtree containing at most 2n/3 nodes, we need to show that:
$(2m/3) + (2(n-k-1-m)/3) + 1 \leq (2(k+1)/3)$
Simplifying this inequality, we get:
$2m + 2(n-k-1-m) + 3 \leq 2k + 2$
$2n - 2k + 1 \leq 2k + 2$
$n \leq 2k + 1/2$
Since k is an integer, the maximum value of 2k will be even.Therefore, (2k + 1/2) is always greater than 2k. Hence, we can conclude that each child of the root of an n-node heap is the root of a subtree containing at most 2n/3 nodes.

To find the smallest constant $\alpha$ such that each subtree has at most $\alpha$n nodes, we need to consider the worst-case scenario for the number of nodes in a subtree.

In a binary tree, the maximum number of nodes in a subtree occurs when the tree is perfectly balanced, and each level has the maximum number of nodes possible.

Obviously,$\alpha = 1/2$.To prove that for an n-node heap, each subtree has at most n/2 nodes, we need to use a proof by induction:
Base Case:
For a heap with only one node (n=1), the subtree rooted at that node will have no node, just at most $n/2 = 1/2$ nodes, which is true.
Inductive Hypothesis:
Assume that for a heap with k nodes, each subtree has at most k/2 nodes.
Inductive Step:
Consider a heap with k+1 nodes. We know that a heap is a complete binary tree, meaning that all levels of the tree are fully filled except possibly for the last level, which is filled from left to right.
In a heap, the root node is the maximum element. When we remove the root node, we replace it with the last element in the heap and then perform a heapify operation to maintain the heap property.
After removing the root node, we are left with a heap of k nodes. By the inductive hypothesis, each subtree in this heap has at most k/2 nodes.
Since the last level of the heap is filled from left to right, the subtree rooted at the last element can have at most k/2 nodes. This is because the last element is either a leaf node or has only one child.
Therefore, for a heap with k+1 nodes, each subtree has at most $(k/2) + (k/2) + 1 = k/2 + 1$nodes.
By induction, we have proven that for an n-node heap, each subtree has at most n/2 nodes.So the constant $\alpha = 1/2$.

**(c)** Steps:
1. Update the value of the element at index i to val.
2. Compare the updated element with its parent.
3. If the updated element is smaller than its parent, swap them.
4. Repeat steps 2 and 3 until the updated element is in its correct position or

becomes the root of the heap.

Here is the implementation in pseudocode:

```
Function heap_update(heap, i, val):
    if i < 0 or i ≥ length(heap) then
        return heap ;      // Invalid index, return the original
         heap
    end
    heap[i] = val ;                  // Update the value at index i
    while i > 0 and heap[i] < heap[(i − 1)//2] do
        swap(heap[i], heap[(i-1)//2]) ;   // Swap the updated element
         with its parent if it is smaller
        i = (i − 1)//2 ;             // Move up to the parent index
    end
    return heap
```

The runtime complexity of this implementation is O(log n) because in the worst case, the updated element may need to be moved up to the root of the heap, which requires traversing the height of the heap (log n levels).

# 3

**(a)** $\begin{bmatrix} 2 & 3 & 4 & \infty \\ 5 & 8 & 9 & \infty \\ 12 & 14 & 16 & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$

**(b)** ExtractMin Algorithm:
(1) Record A[1, 1] then replace it with $\infty$. Let i = 1 and j = 1.
(2) If $i + 1 > n$ or $A[i, j + 1] < A[i + 1, j]$ goto (2.1), else goto (2.2).
(2.1) Swap A[i, j] with $A[i, j + 1], i \leftarrow i + 1$. Goto (3).
(2.2) Swap A[i, j] with $A[i + 1, j], j \leftarrow j + 1$. Goto (3).
(3) If $i == n$ and $j == m$ then return A[1, 1]old, else goto (2).
Correctness:
– Loop invariant: Before step (2), A[1 : i, 1 : j] is an Young tableau.
– Initialization: There are only one element, A[1,1].
– Maintain:
* If (2.1) run, j-th column is sorted;
* else (2.2) run, i-th row is sorted.
– Termination: When $i = n$ and $j = m$, all rows and column are sorted. Therefore A[1 : n, 1 :m] is an Young tableau.
Especially, the updated tableau will still be an Young tableau.

**(c)** To insert a new element:
1. Start at the top-right corner of the tableau (i.e., the element at position (1,

n)).
2. Compare the new element with the current element.
3. If the new element is smaller than the current element, swap them.
4. Move one position to the left (decrease the column index) if possible, or move one position down (increase the row index) if moving left is not possible.
5. Repeat steps 2-4 until the new element is in its correct position or until you reach the bottom-left corner of the tableau (i.e., the element at position (m, 1)).
Since we are moving either left or down at each step, the maximum number of steps required will be the number of rows plus the number of columns in the tableau(m + n), resulting in an O(m + n) time complexity.

# 4

**(a)** To prove by induction that Cruel correctly sorts any input array, two things need to be explained: (1) If Cruel is called on a subarray, it correctly sorts that subarray(2)The role of the Unusual algorithm in the Cruel algorithm is to ensure that the ordering of the subproblems is correct. It guarantees the ordering of the middle part by using recursion and swapping elements. The Unusual algorithm first sorts the two subproblems and then recursively sorts the middle part. This combination of recursion and swapping ensures that the overall ordering of the Cruel algorithm is correct.

Base Case:
When n = 1, the array is already sorted, so Cruel trivially sorts the array.

Inductive Hypothesis:
Assume that Cruel correctly sorts any input array of size k, where k ¡ n.

Inductive Step:
We need to show that Cruel correctly sorts an input array of size n.

According to the pseudocode, Cruel first calls itself recursively on the first half of the array (A1...(n/2)]) and then on the second half of the array (A[(n/2 + 1)...n]). By the inductive hypothesis, these recursive calls correctly sort the respective subarrays.

After that, Cruel calls Unusual on the entire array. We need to show that Unusual correctly sorts the subarray.

Unusual first checks if n = 2. If so, it swaps the elements if necessary to ensure the correct order. This step correctly sorts the array of size 2.

If $n > 2$, Unusual performs a series of swaps to enforce the correct order. It swaps elements in the second and third quarters of the array, and then recursively calls itself on the first half, second half, and middle half of the array. By the inductive hypothesis, these recursive calls correctly sort the respective subarrays.

Therefore, by induction, we can conclude that Cruel correctly sorts any input array.

**(b)** The for loop is responsible for swapping the elements in the second and third quarters of the array. Without this loop, the algorithm would not ensure

the correct order between these quarters, leading to an incorrect sorting result.

**(c)** The last two lines are responsible for recursively calling Unusual on the first half, second half, and middle half of the array. By swapping these lines, the algorithm would not perform the correct recursive calls, leading to an incorrect sorting result.

**(d)** The running time of Unusual is O(n). This is because Unusual performs a constant number of operations for each element in the array, resulting in a linear time complexity.
The running time of Cruel can be expressed as a recurrence relation. According to the pseudocode, Cruel makes two recursive calls on subarrays of size n/2 and then calls Unusual on the entire array. Therefore, I can express the running time as:
$T(n) = 2T(n/2) + O(n)$

Using the Master Theorem, I can determine that the running time of Cruel is O(n log n).

# 5

Using the Dutch National Flag algorithm, also known as the 3-way partitioning algorithm, this algorithm was originally designed by Edsger Dijkstra and is efficient for sorting arrays with multiple values.
Here is the algorithm for red-white-blue sorting:
1. Initialize three pointers: low, mid, and high.
- low points to the first element of the array.
- mid points to the first element of the array.
- high points to the last element of the array.
2. While mid ¡= high:
- If A[mid] is red:
- Swap A[low] and A[mid].
- Increment low and mid pointers by 1.
- If A[mid] is white:
- Increment mid pointer by 1.
- If A[mid] is blue:
- Swap A[mid] and A[high].
- Decrement the high pointer by 1.
3. Repeat step 2 until mid ¡= high.
At the end of this algorithm, all the array will be sorted in the desired order.
The time complexity of this algorithm is O(n), where n is the number of elements in the array. This is because each element is examined and swapped at most once, resulting in a linear time complexity.

# 6

**(a)** Borrow from bubble sort, a possible algorithm is as fellow:
1. Initialize a variable sorted to False.
2. While sorted is False:
- Set sorted to True.
- Iterate through the permutation p from the first element to the second-to-last element:
- If $p[i] > p[i+1]$:
- Reverse the subsequence p[i] to p[i+1].
- Set sorted to False.
4. If sorted is True, the permutation is sorted.
Correctness:
In each iteration, we check if adjacent elements are in the correct order. If not, we reverse the subsequence to bring the larger element towards the end. By repeating this process, we gradually move the larger elements towards the end of the permutation until it is sorted.
Runtime:
The number of reversals required to sort the permutation is at most n-1, as each reversal brings at least one element into its correct position. Therefore, the algorithm performs O(n) reversals.

**(b)** No, it is not possible to sort any permutation using o(n) reversals.
To justify it, we need to consider the number of possible permutations of n elements. There are n! (n factorial) possible permutations of n elements.
If we have o(n) reversals, the maximum number of possible outcomes is $2^{o(n)}$. However,$2^{o(n)}$ is significantly smaller than n!, which means there are more permutations than possible outcomes of the reversals. Therefore, there must exist some permutations that cannot be sorted using o(n) reversals.
In other words, the number of possible permutations grows much faster than the number of possible outcomes of o(n) reversals. Hence, it is not possible to sort any permutation using o(n) reversals.