2.6 Algorithm steps

1. Let X be the value we wish to search for.
2. We use 2 variables 'start' and 'end' to mark beginning index and ending index
3. Repeat the following steps while start <= end
   (a) We calculate 2 mids, mid1 and mid2
   mid1 = (end-start)/3 + start
   mid2 = 2*(end-start)/3 + start
   The array is now divided into 3-parts
   → [start, mid1]
   → [mid1, mid2]
   → [mid2, end]

   (b) If X is equal to arr[mid1] or arr[mid2], return index of x and terminate the program.

   (c) If x lies in interval [start, mid1] set end = mid - 1

   (d) If x lies in the interval (mid1, mid2) set start = mid + 1 and end = mid2 - 1

   (e) Else x lies in the interval [mid2, end], set start = mid2 + 1

4. If x is not present in array, return -1.

Time complexity

Let N be the size of array. After the kth iteration, size of array = $N/3^k$.

$N/3^k = 1$, the program will terminate.

$N/3^k = 1$

$N = 3^k$

$K = \log_3(N)$ since the worse case requires k iterations, hence the time complexity is $O(\log_3 N)$

Space Complexity

The space complexity for this algorithm is $O(N)$

2.7 Algorithm Steps

Let 'low' be starting index and 'high' be ending index

[function FindMax]

```
if low = high then
    return arr[low]
else if low = high-1 then
    return max(A[low], A[high]
else
    set mid ← (low + high)/2
    left ← Recur FindMax(A, low, mid)
    right ← recur FindMax(A, mid+1, high)
    return max(left, right)
```

Time Complexity

Worst case - O(N)
Base case - O(N)

Space Complexity

The space complexity is o(log N) for recursion call stack.

2-13.

step 11 : [function merge]

Set i ← low, j ← m1, k ← m2, l ← low

while i < m1 and j < m2 and k < high repeat

if arr[i] < arr[j] then

if arr[i] < arr[k] then

set final[l] ← arr[i]

Increase l and i by 1

else

set final[l] ← arr[k]

Increase l and k by 1

[end of 'if']

else

if arr[j] < arr[k] then

set final[l] ← arr[j]

Increase l and j by 1

else

set final[l] ← arr[k]

Increase l and k by 1

[End of 'if']

[End of 'while'].

while i < m1 and j < m2 repeat

if arr[i] < arr[j] then

Set final[l] ← arr[i]

Increase l and i by 1

else

Set final[l] ← arr[j]

Increase l and j by 1

[End of 'while']

while j < m2 and k < high repeat

if arr[j] < arr[k] then

Set final[l] ← arr[j]

Increase l and j by 1

else

Set final[l] ← arr[k]

Increase l and k by 1

[End of 'while']

while i < m1 and k < high repeat
    if arr[i] < arr[k] then
        Set final[l] ← arr[i]
        Increase l and i by 1
    else
        Set final[l] ← arr[k]
        Increase l and k by 1

[End of 'while']

while i < m1 repeat
    Set final[l] ← arr[i]
    Increase l and i by 1

[end of 'while']

while j < m2 repeat
    Set final[l] ← arr[j]
    Increase l and j by 1

[End of 'while']

while k < high repeat
    Set final[l] ← arr[k]
    Increase l and k by 1

[End of 'while']

Step 2: [function sort_recursive]
    if high - low < 2 then
        return
    Set m1 ← low + (high - low)/3
    Set m2 ← low + 2 * (high - low)/3 + 1
    Call 'sort_recursive' for index 'low' to 'm1'
    Call 'sort_recursive' for index 'm1' to 'm2'
    Call sort_recursive' for index m2 to high
    Call function merge to perform 3 way merge sort

[End of function 'sort_recursive']

Step 3 : [function 'sort']
if n = 0 then
return

for i = 0 to n-1 repeat
Set duplicate[i] ← arr[i]
[End of 'for']

Call the function 'sort_recursive' from indexRto n-1
for i = 0 to n-1 repeat
Set arr[i] ← duplicate[i]
[End of 'for']

[End of function 'sort']

Algorithm Analysis

$T(n) = 3T(n) + O(n)$

$T(n) = O(n \log_3 n)$

Space complexity

- $O(n)$