CMPS102 HW1

Jinxuan Jiang

jjiang17@ucsc.edu

Grading option: Homework Heavyweight

Question 1 “Maximum Profit”:

For an array A, the length of A is n.

We wish to find Max(A[k] - A[j]) for $1 \le j < k \le n$.

Solution:

If we divide the array into two subarrays, the maximum difference is equal to the maximum of right subarray - minimum of left subarray.

Then we have three cases:

Case 1: The smallest element and the largest element are both in the left subarray,

Case 2: The smallest element and the largest element are both in the right subarray.

Case 3: The smallest element is in the left subarray and the largest element is in right subarray.

```
Take (A, i, j) as input:
If j ≥ k
        Return
If j < k
    1. Divide the array into two subarrays left and right
    2. Find the smallest number of the left array (linear time)
    3. Find the largest number of the right array (linear time)
    4. Result1 = (largest number in right array - smallest number in left array)
```

5. Call the left subarray by recursion and get result2
6. Call the right subarray by recursion and get result3
7. Compare Result1, Result2 and Result3
8. Return the largest result

T(n) = 2T(n/2) (recursion) + O(n) (linear comparison and allocation)

By master theorem, we know that the algorithm will take **O(n log n)**

Proof of correctness:

Claim 1: This algorithm will terminate.

Proof:

If size of array = 1, it will return for sure by base case

If size of array > 1, the function will recursively split the array by n/2 until the base case is reached which will always happen because n/2 is finally getting smaller and smaller.

Claim 2: This algorithm will return the maximum profit of the array A

Proof:

If there exists an (A[j] - A[i]) which is greater than what is returned by the algorithm above, it will go back to the three cases of algorithm, choosing the minimum and maximum again. Since it will return the maximum and minimum, this algorithm will definitely return the maximum profit of the array A.

Question 2 "To play or not to play"

In this question, we need to give a divide and conquer algorithm that takes a list L as input and computes the probability of winning.

Solution:

In order to solve this problem, we need to divide the list into two subarrays first, and then figure out the number of odd and even subarrays of each part. Then multiplying the number of odd and the number of even. Call the function recursively.

Take list L as input:

1. Sum the list L linearly and determine whether the sum is even
2. Divide the list L into two subarrays and count the number of odd elements
3. In each subarray,
   a. If the number of element ==1, determine the odevity of this element, if it is odd, return 1
   b. If the number of element > 1, check the number of odd and even elements of each part.
4. Multiplying the number of odd elements by the number of even elements
5. Using support method to call each part.
6. Total odd number = all the results from recursion call + calculated results.
7. Return the total odd number.
8. Total cases = n(n+1)/2, total even number = total cases - total odd number

   Probability = # of total even number / total cases.

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

By master theorem, we know that the algorithm will take **O(n log n)**

Question 3 "Yet another search problem":

1. **Prove that there exist an index *j* such that $| A[j] - j | \leq$ (k+1) / 2**

   Solution:

   Let's prove it by contradiction:

   Assume that there always exist an index j such that $| A[j] - j | > (k+1) / 2$

   Maximum (A[j]) should be the largest element in the array,

   so, maximum A[j] = n

   j is the index of A[j],

   so the possible maximum of j is (n+1) and the minimum of j is 0.

   Then, the minimum of $| A[j] - j | = 0$, the maximum of $| A[j] - j | = n$

   For $0 \leq K < n$, the maximum of $(k+1)/2 = n/2$

   When $| A[j] - j | = 0$ and $(k+1)/2 = n/2$, $| A[j] - j | > (k+1) / 2$ doesn't exist.

   So, there exist an index *j* such that $| A[j] - j | \leq (k+1) / 2$

2. **Given the number k, find an O(log n) Divide and Conquer Algorithm that finds such an index.**

   Solution:

   Take (A, j) as input

   1. Find the initial mid point $j = (n-1)/2$ and if $-(k+1)/2 \leq A[j] \leq (k+1)/2$, return j
   2. If $A[j] - j < -(k+1)/2$, we need to decrease j and increase the value of (A[j] - j) in the left part by recursion from 0 to midpoint
   3. If $A[j] - j > (k+1)/2$, we need to increase j and decrease the value of (A[j] - j) in the left part by recursion from midpoint+1 to n-1
   4. Repeat the recursion above until find the index j.

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

By master theorem, we know that the algorithm will take **O(log n)**

Question 4 “Fool’s Gold”

Solution:

The key to this problem is to use a modified binary search to find local maximum in the array.

Take array A as input:

1. Find the midpoint q of the array, and compare the value with A[q -1] and A[q+1]
2. If the $A[q-1] \leq A[q] \leq A[q+1]$, return q.
3. If either mid point is larger than it then call the modified binary search to that half of the array and repeat. We have:
    a. If A[midpoint -1] < A[midpoint] < A[midpoint+1], return midpoint
    b. If A[midpoint - 1] > A[midpoint], call the function again on the left half of the array and solve the problem by recursion
    c. If A[midpoint + 1] > A[midpoint], call the function again on the right half of the array and solve the problem by recursion.
4. It terminates when it gets all the local maximum point in the array

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

By master theorem, we know that the algorithm will take **O(log n)**

Proof of correctness:

Claim: There exists a local maximum in the array.

Proof:

Assume there doesn’t exist a local maximum in the subarray.

Since the first and the last element in the array are 0, every other element between the first and last element should be greater than 0.

So, there always exists a local maximum in the array.