DAA Project: Closest Pair Points & Karatsuba Multiplication

Team Members:
23K-0583 Taaha Khan
23K-0526 Abdul Basit
23K-0756 Alishba

Q1 a) Find Min Max Algorithm

1. Pseudocode

The algorithm `FindMinMax(Arr, l, h)` takes an array `Arr` with index l and h and outputs a pair $(\min, \max)$.

Base Cases:

- If $l == h$, return $(\text{Arr}[l], \text{Arr}[l])$.
- Else if $h == l + 1$:
  - If $\text{Arr}[l] < \text{Arr}[h]$, return $(\text{Arr}[l], \text{Arr}[h])$.
  - Else, return $(\text{Arr}[h], \text{Arr}[l])$.

Recursive Step:

1. Calculate $mid = (l + h)//2$.
2. Recursively call:
$$(\min_1, \max_1) = \text{FindMinMax}(\text{Arr}, l, mid)$$
$$(\min_2, \max_2) = \text{FindMinMax}(\text{Arr}, mid + 1, h)$$
3. Compare results:
$$\text{finalmin} = \min(\min_1, \min_2)$$
$$\text{finalmax} = \max(\max_1, \max_2)$$
4. Return $(\text{finalmin}, \text{finalmax})$.

2. Complexity Analysis

$T(n)$ represents the number of comparisons made for an array of size n.

Base cases:

- $n = 1 \to 0$ comparisons.
- $n = 2 \to 1$ comparison.

Recursive case $(n > 2)$**:**

- Divide into two subarrays of size $n/2$.
- $T(n) = T(n/2) + T(n/2) + 2$ (comparisons for two minimums and two maximums).
- Simplified recurrence: $T(n) = 2T(n/2) + 2$.

Solving the Recurrence: Using substitution:

$$T(n) = 2T(n/2) + 2$$
$$T(n) = 2[2T(n/4) + 2] + 2 = 4T(n/4) + 4 + 2$$
$$T(n) = 2^k T(n/2^k) + 2^{k-1} + \cdots + 2$$

Assuming $n = 2^k$, then $k = \log n$ and $T(n/2^k) = T(1) = 0$. The series sums to a geometric progression resulting in:

$$T(n) = 1.5n - 2$$

3. Brute-Force Algorithm

- Initialize min = Arr[0] and max = Arr[0].
- Iterate i from 1 to $n - 1$.
- If Arr[i] $<$ min, update min.
- If Arr[i] $>$ max, update max.

Complexity: The loop runs $n - 1$ times with 2 comparisons per iteration, totaling $T(n) = 2(n - 1)$.

Conclusion

The Divide and Conquer algorithm is more efficient than brute force because $1.5n - 2$ comparisons are less than $2n - 2$ comparisons.

Q1 b) Power Algorithm

1. Pseudocode

The algorithm `Power(a, n)` takes base a and exponent n ($n > 0$) to output a^n.

- If $n == 0$, return 1.
- If $n == 1$, return a.
- If $n\%2 == 0$ (even):
  - $half = \text{Power}(a, n/2)$.
  - Return $half \times half$.
- Else (odd):
  - $half = \text{Power}(a, (n-1)/2)$.
  - Return $a \times half \times half$.

2. Complexity Analysis

$T(n)$ is the number of multiplications.
Recurrence:

- If n is even: $T(n) = T(n/2) + 1$.
- If n is odd: $T(n) = T((n-1)/2) + 2$.

Solving ($n = 2^k$):

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

Solving leads to $T(n) = T(1) + \log n = 0 + \log n = \log_2(n)$ multiplications.

3. Brute Force Algorithm

- Iterate from $i = 1$ to n, multiplying the result by a each time.
- **Complexity:** $T(n) = n$ multiplications.

Conclusion

Divide and conquer is more effective since $\log n$ multiplications is less than n multiplications required by the brute force algorithm.

Q1 c) Count Inversion Algorithm

1. Pseudocode

The algorithm `CountInversion(A[0...n-1])` counts inversions in array A.

- If $n \leq 1$, return 0.
- Set $mid = [n/2]$.
- Define `LeftHalf` as $A[0 \ldots mid-1]$ and `RightHalf` as $A[mid \ldots n-1]$.
- Recursive calls:
  - $\text{LeftInv} = \text{CountInversion}(\text{LeftHalf})$
  - $\text{RightInv} = \text{CountInversion}(\text{RightHalf})$
  - $\text{SplitInv} = \text{MergeAndCount}(A, \text{LeftHalf}, \text{RightHalf})$
- Return $\text{LeftInv} + \text{RightInv} + \text{SplitInv}$.

Merge and Count Procedure

Inputs two sorted subarrays $\text{Left}[0 \dots p-1]$ and $\text{Right}[0 \dots q-1]$.

- Initialize $i = 0, j = 0, k = 0$ and $\text{invCount} = 0$.
- While $i < p$ and $j < q$:
  - If $\text{Left}[i] < \text{Right}[j]$:
    * $A[k] = \text{Left}[i]$
    * $i = i + 1$, $k = k + 1$
  - Else:
    * $A[k] = \text{Right}[j]$
    * $j = j + 1$, $k = k + 1$
    * $\text{invCount} = \text{invCount} + (p - i)$
- Copy remaining elements from `Left` and `Right` into A.
- Return `invCount`.

Q1 d) Quick Sort Analysis

1. Best Case

The recurrence is $T(n) = 2T(n/2) + O(n)$. Using Master Theorem where $a = 2, b = 2, k = 1$:

$$\log_b a = \log_2 2 = 1$$

This matches Case II, resulting in $O(n^1 \log n) = O(n \log n)$.

2. Worst Case

Occurs when the pivot is the smallest or largest element. Recurrence: $T(n) = T(n-1) + n$.

Derivation:

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

Substituting leads to $T(n) = T(1) + 2 + 3 + \cdots + n$. Summation formula:

$$T(n) = \frac{n(n+1)}{2} = O(n^2)$$

Q1 e) Closest Pair Algorithm

1. Pseudocode

- Sort A in increasing order.
- Call `Closest(A, 1, n)`.

Procedure Closest(A, low, high):

- If $(high - low) == 1$, return $|A[high] - A[low]|$.
- $mid = (low + high)/2$.
- $d1 = \text{Closest}(A, low, mid)$.
- $d2 = \text{Closest}(A, mid + 1, high)$.
- $d3 = |A[mid + 1] - A[mid]|$.
- Return $\min(d1, d2, d3)$.

2. Time Complexity

Recurrence: $T(n) = 2T(n/2) + c$ where $c = \Theta(1)$. Master Theorem parameters: $a = 2, b = 2, f(n) = \Theta(1)$.

$$\log_b a = \log_2 2 = 1, \text{ so } n^{\log_b a} = n^1 = n$$

Since $f(n) = \Theta(1)$ is smaller than n, $T(n) = O(n)$.

Total Complexity: Sorting step takes $O(n \log n)$.

$$T_{total} = O(n \log n) + O(n) = O(n \log n)$$

3. Conclusion

Yes, it is a good algorithm for this problem. It is efficient, correct, and asymptotically optimal for finding the closest pair in one dimension.