Department of Computer Science
Ashoka University

Design and Analysis of Algorithms

Assignment 1

Collaborators: None **Name: Rushil Gupta**

Problem 1

To run merge sort with k partitions, where k can be arbitrary, we can use the following algorithm:

Algorithm 1 Merge Sort with k partitions

```
1:  function MergeSortK(A, k)
2:      n ← length(A)
3:      if n ≤ 1 then                                          ▷ Base case
4:          return A
5:      end if
6:      m ← ⌈n/k⌉
7:      B ← new array of size k
8:      for i ← 0 to k − 1 do
9:          B[i] ← A[i · m : min((i + 1) · m, n)]
10:         B[i] ← MergeSortK(B[i], k)
11:     end for
12:     minHeap = MinHeap()                  ▷ We will use a min-heap to merge the k partitions
13:     for i ← 0 to k − 1 do
14:         if length(B[i]) > 0 then
15:             minHeap.insert(B[i][0], i, 0)   ▷ Put the first indices of each partition in the min-heap
16:         end if
17:     end for
18:     C ← new array of size n
19:     for i ← 0 to n − 1 do
20:         (val, l, j) ← minHeap.pop()
21:         C[i] ← val
22:         if (j < length(B[l]) − 1) and (B[l][j + 1] ≠ null) then
23:             minHeap.insert(B[l][j + 1], l, j + 1)
24:         end if
25:     end for
26:     return C
27: end function
```

Proof of Correctness

We prove by induction on the length n of the input array A that the algorithm MergeSortK returns a sorted version of A.

Base Case: If $n \leq 1$, the algorithm immediately returns A. An array with zero or one element is trivially sorted, so the algorithm is correct in this case.

Inductive Hypothesis: Assume that for every array A' with length less than n, the algorithm correctly returns a sorted array.

Inductive Step: Now, consider an array A of length $n > 1$. The algorithm divides A into k subarrays, each of length at most $\lceil n/k \rceil$. By the inductive hypothesis, each recursive call

$$B[i] \leftarrow \text{MergeSortK}(B[i], k)$$

returns a sorted subarray.

After sorting the subarrays, the algorithm merges them using a min-heap. Initially, the first element of each non-empty subarray is inserted into the heap. At every step, the algorithm pops the minimum element from the heap and puts it to the output array C. Then, it inserts the next element from the same subarray (if any) to the heap. This process ensures that at each step the smallest available element among the subarrays is selected, maintaining the overall sorted order (this can be interpreted as keeping a track of "k" pointers, one for each subarray).

Thus, the merge operation correctly combines the k sorted subarrays into one sorted array.

By induction, MergeSortK correctly sorts any array A.

Time Complexity

Since this is a divide and conquer algorithm, we will first analyze the cost of splitting and merging the array.

Splitting: The algorithm splits the array into k subarrays, each of length at most $\lceil n/k \rceil$. This operation takes $O(n)$ time, since each subarray is created by copying a contiguous part of the original array.

Merging: The algorithm merges the k subarrays using a min-heap. The min-heap has at most k elements at any time, and the heap operations take $O(\log k)$ time. Since there are n elements in total, the total time complexity of merging is $O(n \log k)$.

This gives us the following recurrence relation for the time complexity of the algorithm:

$$\begin{aligned} T(n) &= kT\left(\frac{n}{k}\right) + O(n \log k) \quad (O(n) \text{ is dominated by } O(n \log k)) \\ T(1) &= O(1) \end{aligned}$$

We will solve this recurrence relation by expanding it:

$$\begin{aligned} T(n) &= k\,T\left(\frac{n}{k}\right) + O(n \log k) \\ &= k\left(k\,T\left(\frac{n}{k^2}\right) + O\left(\frac{n}{k}\log k\right)\right) + O(n \log k) \\ &= k^2\,T\left(\frac{n}{k^2}\right) + 2\,O(n \log k) \\ &\vdots \\ &= k^i\,T\left(\frac{n}{k^i}\right) + i \cdot O(n \log k) \end{aligned}$$

Here, i is the number of recursion levels. The recursion terminates when

$$\frac{n}{k^i} = 1 \quad \Longrightarrow \quad i = \log_k n.$$

Substituting $i = \log_k n$ into the recurrence, we get:

$$T(n) = k^{\log_k n} + O(n \log k) \cdot \log_k n = O(n) + O\left(n \cdot \log_k n \cdot \log k\right).$$

Noting that

$$\log_k n = \frac{\log n}{\log k},$$

we simplify the expression:

$$T(n) = O\left(n \cdot \frac{\log n}{\log k} \cdot \log k\right) = O(n \log n).$$

Problem 5

Let $N(h)$ be the number of nodes in an AVL tree of height h. Then, we derive the following inequality:

$$\begin{aligned}
&N(h) \geq 1 + N(h-1) + N(h-2) \quad \text{(since the height of the tree is at most } h\text{)} \\
&N(0) = 1 \quad \text{(base case)} \\
&N(1) = 2 \quad \text{(base case)}
\end{aligned}$$

Now, we will look at the worst case, where the heights of the subtrees of each node differ by 1 (bounded by 1 by the AVL property). This gives us the following:

$$N(h) = 1 + N(h-1) + N(h-2) \quad \text{for } h \geq 2$$

Comparing this with the Fibonacci sequence, we **claim** that $N(h) \geq F(h)$.

Proof: We will prove this using induction.

Base Case: $N(0) = 1 \geq 0 = F(0)$

Inductive Hypothesis: Assume $N(k) \geq F(k)$ for all $k \leq h$ for some h.
Inductive Step: We want to show that $N(h+1) \geq F(h+1)$.

$$\begin{aligned}
N(h+1) &= 1 + N(h) + N(h-1) \\
&\geq 1 + F(h) + F(h-1) \quad \text{(by the inductive hypothesis)} \\
&= 1 + F(h+1) \\
&\geq F(h+1) \quad \text{(since } 1 > 0\text{)}
\end{aligned}$$

Thus, $N(h) \geq F(h)$ for all $h \geq 0$. $\square$

Now, we **claim** that $F(h) \geq \frac{\phi^{h-1}}{\sqrt{5}} - 1$.

Proof: We will prove this using strong induction.

Base Cases: $F(0) = 0 \geq \frac{\phi^{-1}}{\sqrt{5}} - 1$, $F(1) = 1 \geq \frac{\phi^{0}}{\sqrt{5}} - 1$
Inductive Hypothesis: Assume $F(k) \geq \frac{\phi^{k-1}}{\sqrt{5}} - 1$ for all $k \leq h$ for some h.
Inductive Step: We want to show that $F(h+1) \geq \frac{\phi^{h}}{\sqrt{5}} - 1$.

$$\begin{aligned}
F(h+1) &= F(h) + F(h-1) \\
&\geq \frac{\phi^{h-1}}{\sqrt{5}} + \frac{\phi^{h-2}}{\sqrt{5}} - 2 \quad \text{(by the inductive hypothesis)} \\
&= \frac{\phi^{h-2}}{\sqrt{5}} \cdot (\phi + 1) - 2
\end{aligned}$$

By the definition of ϕ, we know that $\phi^2 = \phi + 1$. Thus, we can say that:

$$\begin{aligned}
F(h+1) &\geq \frac{\phi^{h-2}}{\sqrt{5}} \cdot \phi^2 - 2 \\
&\geq \frac{\phi^{h}}{\sqrt{5}} - 1
\end{aligned}$$

Thus, $F(h) \geq \frac{\phi^{h-1}}{\sqrt{5}} - 1$ for all $h \geq 0$. $\square$

Now, we say that $n = N(h) \geq F(h) \geq \frac{\phi^{h-1}}{\sqrt{5}} - 1 \implies \log_\phi\left(\sqrt{5}(n+1)\right) \geq h - 1 \implies h \leq \log_\phi\left(\sqrt{5}(n+1)\right) + 1$.

This is by definition saying that $h = O(\log n)$.

Problem 6

We will try to adapt the Karatsuba algorithm for this problem of polynomial multiplication. We have:

$$P_A(n) = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$$
$$P_B(n) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$$

See that:

$$P_A(n) \cdot P_B(n) = (a_{n-1}x^{n-1} + \cdots + a_1x + a_0) \cdot (b_{n-1}x^{n-1} + \cdots + b_1x + b_0)$$

So, if we break the polynomials into two halves like the Karatsuba algorithm, we get:

$$P_A(n) = A_1x^{\frac{n}{2}} + A_0 = (a_{n-1}x^{\frac{n}{2}-1} + \cdots + a_{\frac{n}{2}})x^{\frac{n}{2}} + (a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + a_0)$$
$$P_B(n) = B_1x^{\frac{n}{2}} + B_0 = (b_{n-1}x^{\frac{n}{2}-1} + \cdots + b_{\frac{n}{2}})x^{\frac{n}{2}} + (b_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + b_0)$$

If n is odd here, we can thing of the polynomials as having $n+1$ terms, with the coefficient of x^n being 0. Now, using the same trick as Karatsuba, we can write:

$$P_A(n) \cdot P_B(n) = A_1B_1x^n + (A_1B_0 + A_0B_1)x^{\frac{n}{2}} + A_0B_0$$
$$(A_1 + A_0) \cdot (B_1 + B_0) = A_1B_1 + (A_1B_0 + A_0B_1) + A_0B_0$$

We can compute $A_1 + A_0$ and $B_1 + B_0$ in $O(n)$ time and the following multiplications can be done recursively:

- A_1B_1
- A_0B_0
- $(A_1 + A_0) \cdot (B_1 + B_0)$

We can compute $A_1B_0 + A_0B_1$ from the earlier formula. Then, we just need to multiply the terms by the appropriate powers of x and add them up. This will take $O(n)$ time (addition and subtraction of n numebrs takes $O(n)$ time, and multiplying with x^n is just shifting the array by n).

Lastly, the explicitly state the base case, where $n = 1$. In this case, the multiplication can be done in $O(1)$ time, since we are only multiplying two numbers.

This means, given $P_A(n)$ and $P_B(n)$, we break then into halves, and do 3 multiplications recursively at each time step. Then, in linear time, we can combine the results to get the final answer, which is correct.

Time Complexity

As mentioned through the algorithm, the final recurrence relation for the time complexity of this algorithm comes out to be:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$
$$T(1) = O(1)$$

We can solve this recurrence relation by expanding this until we reach the base case:

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{2}\right) + O(n) \\
&= 3^kT\left(\frac{n}{2^k}\right) + O(n)\sum_{i=0}^{k-1}\left(\frac{3}{2}\right)^i \quad \text{General form for the } k^{th} \text{ depth} \\
&= T(1) \cdot 3^{\log_2 n} + O(n)\sum_{i=0}^{\log_2 n - 1}\left(\frac{3}{2}\right)^i \quad (\text{with } k = \log_2 n) \\
&= O\left(n^{\log_2 3}\right) + O(n) \cdot \frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} \\
&= O\left(n^{\log_2 3}\right) = O(n^{1.58})
\end{aligned}$$

So, this algorithm runs in $O(n^{1.58})$ time.

Correctness

This algorithm mirrors the Karatsuba algorithm, and the correctness of the Karatsuba algorithm is well-known. The algorithm breaks the polynomials into two halves, computes the products of the halves, and then combines them to get the final product. The correctness of the algorithm follows from the correctness of the Karatsuba algorithm.

Problem 7

Overview of the Algorithm

The algorithm works by sweeping the points in order of descending x-coordinate. During the sweep, we maintain a data structure ("staircase") which represents the maximal set of points among those processed so far. Denote this set by S_i after scanning the first i points.

For each new point p, since the sweep guarantees that any point processed later has a smaller x-coordinate, it suffices to check whether p is dominated in the y and z coordinates by some point in the current staircase. If p is maximal, we must update the staircase by removing any points that become dominated by p.

The Staircase Data Structure

We implement the staircase as a balanced binary search tree keyed by the y-coordinate. Each node in the BST stores a point's (y, z) pair. The key observation is that the staircase is monotonic: when the points are sorted by y, their corresponding z values are in strictly decreasing order ("as y increases, z falls"). This allows us to do the following:

(a) **Size:** Returns the number of nodes in the BST.

(b) **Successor:** For a new point $p = (x, y, z)$, search in the BST based on y for a point with the lowest y-value not less than $p.y$ and return this point. If no such point exists, return the point with the highest y in the BST. Let $q = (y_q, z_q)$ be this point. If $z_q < p.z$, then p is maximal.

(c) **Predecessor:** Symmetric to the successor operation, but searches for the highest y-value not greater than $p.y$. If no such node exists, return **null**. The goal here is to use this method to delete points that are dominated by a new point p. So, if $q = (y_q, z_q)$ is the predecessor of p and $z_q \leq p.z$, then q is dominated by p and should be removed.

Pseudocode

Algorithm 2 Maximal Points via Plane Sweep

```
1:  function FindMaximalPoints(S)
2:      Input: A set S of n points (x, y, z).
3:      Output: The maximal set M ⊆ S.
4:      Sort S in descending order of x.
5:      Initialize an empty BST, Staircase
6:      Initialize M ← ∅.
7:      for each point p = (x, y, z) in sorted S do
8:          if BST_Size(Staircase) > 0 then
9:              node ← BST_Successor(Staircase, (y, z))
10:             if node.z > z then                    ▷ p is dominated in y-z; skip p.
11:                 continue
12:             end if
13:         end if
14:         Add p to M.                               ▷ If we are here, p is maximal.
15:         BST_Insert(Staircase, (y, z)).            ▷ Insert (y, z) into the BST.
16:         while true do
17:             node ← BST_Predecessor(Staircase, (y, z))
18:             if node = null or node.z > z then
19:                 break
20:             end if
21:             BST_Delete(Staircase, node)
22:         end while
23:     end for
24:     return M
25: end function
```

Correctness

Correctness: Since the points are processed in descending order of x, any point added to M is guaranteed not to be dominated in the x-coordinate by any later point. The staircase maintained in the BST always represents the maximal points in the y-z plane among those with larger x-values. Therefore, a new point is declared maximal if and only if it is not dominated in both y and z, ensuring that M is indeed the set of maximal points in 3D.

Time Complexity

Let n be the number of points in S.

- **Sorting:** Sorting the points in descending order of x takes $O(n \log n)$ time.
- **Processing Each Point:** For each of the n points, we perform several operations on the BST:
  - A **successor** search to check if the point is dominated in the y-z plane, which takes $O(\log n)$.
  - A **BST insertion** when the point is determined to be maximal, taking $O(\log n)$.
  - A **predecessor** search followed by potentially multiple **BST deletions** in the while-loop. Although the inner loop might perform several deletions for a single point, note that each point is inserted exactly once and can be deleted at most once.

  Hence, across all iterations, the total cost for deletions is $O(n \log n)$.

Thus, combining the sorting and the per-point processing, the overall time complexity of the algorithm is:

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