Department of Computer Science
Ashoka University

Design and Analysis of Algorithms: CS-3210-1

Assignment 1

Name: Dhruman Gupta

---

1. We want to write a k-way merge sort algorithm.
   This can be done by the following algorithm:

   Input: A list A of size n and an integer k.

   1. First, we remove the trivial case, where $n = 1$. In this case, we return the array as it is.
   2. Otherwise, we divide the list into k parts. Solve each of these parts recursively.
   3. Now, we have k sorted lists. We now want to merge them efficiently.
   4. We can do this by using a min-heap. We can parameterize the heap as a heap which's elements is an list, and the key of the heap is the first element of the list.
   5. Insert the k sorted lists into the heap.
   6. While the heap is not empty, do the following:
      - Remove the minimum element from the heap. This is the array with the smallest first element.
      - Add the element to the result list.
      - Remove the element from the list it was from, and insert the remaining tail into the heap. If the tail is empty, do not insert it.
   7. Return the result list.

   Now, note that we have the following invariants:
   1. Each list is sorted. i.e, the smallest element is at the beginning of the list.
   2. The min element of the heap is the smallest first element of all the lists.

   By this, we get the following conclusion:
   1. The min element of the heap is the smallest element in all k lists.
   2. The first element of the tail (after removing the first element of the list) is still the smallest element in that list (thus the invariant is maintained).

   Proof of correctness:
   We will prove using strong induction on the size of the list.

   Base case: When $n = 1$, the list is already sorted, and the result is the same as the input.

   Inductive Hypothesis: $\exists k \in \mathbb{N}, \forall n \leq k$ k-way merge sort returns a sorted list.

   Inductive Step: We need to show that if the algorithm is correct for all $n \leq k$, then it is correct for $n = k + 1$.

   We first split the list into k parts. So, each part has at most $\lceil \frac{n}{k} \rceil$ elements. We sort these, which will be correctly sorted by the inductive hypothesis. Now, we construct the min-heap. Following the invairants listed above, we can see that the minimum element of the heap is the smallest element in all k lists. Now, we remove the first element from the min element of the heap heap, and insert the remaining tail into the heap. This will maintain the invariant that the heap is a min-heap, and that all k lists in the heap are sorted. We repeat this until the heap is empty. At this point, we have merged all the lists, and the result is a sorted list.

   Thus, if the algorithm is correct for all $n \leq k$, then it is correct for $n = k + 1$.

   By the principle of strong induction, we have shown that the algorithm is correct for all $n \in \mathbb{N}$, i.e for any arbitrary size of the list, the algorithm will return a sorted list.

Time Complexity Analysis:
$T(n)$ = Time taken to split + Time taken to recursively sort each part + Time taken to merge.

Now, partitioning into k lists takes $O(n)$ time. This can be done in a single pass.
Recursively sorting each part takes $T(\lceil \frac{n}{k} \rceil)$ time by the inductive hypothesis.
Now, we need to merge the k sorted lists. Say this takes $R(n, k)$ time.

So, $T(n) = O(n) + k \times T(\lceil \frac{n}{k} \rceil) + R(n, k)$

Let's solve for $R(n, k)$.

$R(n, k)$ = Time to build min-heap + Time to extract and insert elements from the heap.

It is known that time to build a min-heap is $O(k)$, as there are k elements.
There are a total of n elements in all the lists. We will extract each element exactly only once. Each extraction will take $O(\log k)$ time, as we are extracting from a heap of size k.

So, $R(n, k) = O(k) + n \times O(\log k) = O(k + n \log k) = O(n \log k)$ (under the assumption that k is no more than order of n).

So, $T(n) = O(n) + k \times T(\lceil \frac{n}{k} \rceil) + O(n \log k) = kT(\frac{n}{k}) + O(n \log k)$.
Also, note that $T(1) = O(1)$. Let's solve this.

$$\begin{aligned}
T(n) &= kT(\frac{n}{k}) + O(n \log k) \\
&= k(kT(\frac{n}{k^2}) + O(\frac{n}{k} \log k)) + O(n \log k) \\
&= k^2 T(\frac{n}{k^2}) + 2O(n \log k) \\
&= k^2(kT(\frac{n}{k^3}) + O(\frac{n}{k^2} \log k)) + 2O(n \log k) \\
&= k^3 T(\frac{n}{k^3}) + 3O(n \log k) \\
&= \dots \\
&= k^i T(\frac{n}{k^i}) + iO(n \log k)
\end{aligned}$$

Now, assume for some i, $n = k^i$. Then, $i = \log_k n = \frac{\log n}{\log k}$.

$$\begin{aligned}
T(n) &= k^{\log_k n} T(1) + \log_k n \times O(n \log k) \\
&= O(n \frac{\log k \log n}{\log k}) \\
&= O(n \log n)
\end{aligned}$$

Thus, $T(n) = O(n \log n)\ \forall n, k \in \mathbb{N}$.

2. We want to prove the $O(\log n)$ height property for an AVL tree. We are given that the different between the heights of the left and right subtrees is at most 1.

   Let $N(h)$ be the minimum number of nodes in an AVL tree of height h. Now, we know that in the worst case, the tree is unbalanced at every level. But, pair of siblings can only differ by 1. We can use this to formulate the following constraint:

   $N(h) = N(h-1) + N(h-2) + 1$. We can keep the base cases as $N(0) = 0$, and $N(1) = 1$. Now, we can actually show that:

   $$N(h) \geq F_{n+1} - 1 \ \forall h \in \mathbb{N}$$

   Where F_n is the nth Fibonacci number ($F_0 = 0, F_1 = 1$). We can prove this by induction.

   Base case:
   When $h = 0$, $N(0) = 0 = F_1 - 1 = 0$. When $h = 1$, $N(1) = 1 = F_2 - 1 = 0$. So, the base case holds.

   Inductive Hypothesis:
   $\exists k \in \mathbb{N}, \forall n \leq k \ N(n) \geq F_{n+1} - 1$.

   Inductive Step:
   We need to show that if the statement is true for all $n \leq k$, then it is true for $n = k+1$.

   By the inductive hypothesis, we know that:

   $$N(k-1) \geq F_k - 1$$
   $$N(k-2) \geq F_{k-1} - 1$$

   $N(k) = N(k-1) + N(k-2) + 1 \geq (F_k - 1) + (F_{k-1} - 1) + 1 = F_{k+1} - 1$.
   Thus, by the principle of induction, we have shown that $N(h) \geq F_{h+1} - 1 \ \forall h \in \mathbb{N}$.
   Now, we know that the n^{th} Fibonacci number is given by the following formula:

   $$F_n = \Omega(\phi^n)$$

   Where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Now, we know that $N(h) \geq F_{h+1} - 1$.

   $$N(h) = \Omega(\phi^{h+1} + 1) = O(\phi^{h+1})$$

   Now, if an AVL tree of height h has n nodes, we know that:

   $$n \geq N(h) \geq \Omega(\phi^h)$$

   Taking logarithms on both sides, we get:

   $$\log n \geq \log \Omega(\phi^h) = h \log \phi$$
   $$\implies h = O(\log n)$$

   So, thus, the height of an AVL tree is $O(\log n)$, where n is the number of nodes in the tree.

3. Given 2 polynomials $P_A(n) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$ and $P_B(n) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \dots + b_1x + b_0$, we want to multiply them in sub $O(n^2)$ time. We will use the karatsuba algorithm to do this.

First, divide the polynomials into two halves. Let n be the degree of the polynomial. Then, we can write:

$$\begin{aligned}
P_{A1} &= a_{n-1}x^{\lfloor \frac{n}{2} \rfloor - 1} + a_{n-2}x^{\lfloor \frac{n}{2} \rfloor - 2} + \dots + a_{\lceil \frac{n}{2} \rceil}x^0 \\
P_{A2} &= a_{\lceil \frac{n}{2} \rceil - 1}x^{\lceil \frac{n}{2} \rceil - 1} + a_{\lceil \frac{n}{2} \rceil - 2}x^{\lceil \frac{n}{2} \rceil - 2} + \dots + a_0x^0 \\
P_{B1} &= b_{n-1}x^{\lfloor \frac{n}{2} \rfloor - 1} + b_{n-2}x^{\lfloor \frac{n}{2} \rfloor - 2} + \dots + b_{\lceil \frac{n}{2} \rceil}x^0 \\
P_{B2} &= b_{\lceil \frac{n}{2} \rceil - 1}x^{\lceil \frac{n}{2} \rceil - 1} + b_{\lceil \frac{n}{2} \rceil - 2}x^{\lceil \frac{n}{2} \rceil - 2} + \dots + b_0x^0
\end{aligned}$$

Now, we can write:

$$\begin{aligned}
P_A(n) &= P_{A1}x^{\lceil \frac{n}{2} \rceil} + P_{A2} \\
P_B(n) &= P_{B1}x^{\lceil \frac{n}{2} \rceil} + P_{B2}
\end{aligned}$$

This allows us to re-write the product of the two polynomials as:

$$\begin{aligned}
P_A(n)P_B(n) &= (P_{A1}x^{\lceil \frac{n}{2} \rceil} + P_{A2})(P_{B1}x^{\lceil \frac{n}{2} \rceil} + P_{B2}) \\
&= P_{A1}P_{B1}x^n + (P_{A1}P_{B2} + P_{A2}P_{B1})x^{\lceil \frac{n}{2} \rceil} + P_{A2}P_{B2}
\end{aligned}$$

Now, note that $P_{A1}, P_{A2}, P_{B1}, P_{B2}$ are of at most degree $\lceil \frac{n}{2} \rceil$. We need to calculate the 4 products mentioned above, and then do the 3 additions. Cleverly, we can note that:

$$(P_{A2} + P_{A1})(P_{B2} + P_{B1}) - P_{A1}P_{B1} - P_{B2}P_{A2} = P_{A1}P_{B2} + P_{A2}P_{B1}$$

Now, we can do 2 additions, and 2 subtractions with $n/2$ bit numbers. Then, we have to do 2 multiplications with $n/2$ bit numbers, and a multiplication with a set of $n+1$ bit numbers. This reduces the number of multiplications to 3. With this, we can clearly write the following algorithm:
1. If $n = 1$, return a_0b_0
2. Partition $P_A(n)$ and $P_B(n)$ into $P_{A1}, P_{A2}, P_{B1}, P_{B2}$ as defined above. This is just splitting the coefficients of the polynomials.
3. Now, calculate $(P_{A2} + P_{A1})(P_{B2} + P_{B1})$, $P_{A1}P_{B1}$, and $P_{A2}P_{B2}$, recursively.
4. Using the forumla above, we attain the value of $P_{A1}P_{B2} + P_{A2}P_{B1}$.
5. Now, we can calculate the product of the two polynomials using the formula given above.
6. Return $P_{A1}P_{B1}x^n + (P_{A1}P_{B2} + P_{A2}P_{B1})x^{\lceil \frac{n}{2} \rceil} + P_{A2}P_{B2}$

Now, let's analyze the time complexity of this algorithm. $T(n)$ = time to partition the polynomials + time to calculate the 3 products + time to do the 3 additions and subtractions + time to do the bit shifts for the final answer.

- Time to partition the polynomials: $O(n)$
- Time to calculate the each of the 3 products: $T(\lceil \frac{n}{2} \rceil)$ (for one of the products it takes 1 more bit of multiplcation, but when we do big O analysis we can omit that)
- Time to do the 3 additions and subtractions: $O(n)$
- Time to do the bit shifts for the final answer: $O(n)$

So, $T(n) = O(n) + 3T(\lceil \frac{n}{2} \rceil) + O(n) = O(n) + 3T(\lceil \frac{n}{2} \rceil)$

$$\begin{aligned}
T(n) &= O(n) + 3T(\lceil \frac{n}{2} \rceil) \\
&= O(n) + 3O(\lceil \frac{n}{2} \rceil) + 9T(\lceil \frac{n}{4} \rceil) \\
&= \dots \\
&= \sum_{i=0}^{k} 3^i O(\lceil \frac{n}{2^i} \rceil) + 3^{k+1}T(\lceil \frac{n}{2^{k+1}} \rceil)
\end{aligned}$$

Now, assume for some i, $n = 2^i$. Then, $i = \log_2 n$. Also, $T(1) = O(1)$ (just 1 multiplication).

$$T(n) = \sum_{j=0}^{\log_2 n} 3^j O(\lceil \frac{n}{2^j} \rceil) + 3^{\log_2 n} O(1) = O(n + 3n/2 + 9n/4 + \cdots + (\frac{3}{2})^{\log_2 n} n)$$
$$= O(n \frac{(3/2)^{\log_2 n} - 1}{3/2 - 1}) = O(n(n^{\log_2 3 - 1})) = O(n^{\log_2 3})$$

Now, $\log_2 3$ 1.585. So, $n^{\log_2 3} = n^{1.58}$ is sub-quadratic. Thus we are able to multiply 2 polynomials in sub-quadratic time.

Proof of correctness:
The proof of correctness is given by the discussion above. We can do this by induction.

Base case: When $n = 1$, we have $a_0 b_0$, which is the product of the two constants.

Inductive Hypothesis: $\exists k \in \mathbb{N}, \forall n \leq k$ the algorithm is correct.

Inductive Step: We need to show that if the algorithm is correct for all $n \leq k$, then it is correct for $n = k + 1$.

Let's look at the 3 products we calculate. We know that the algorithm is correct for these products by the inductive hypothesis. Now, we know that the final answer is a linear combination of these products. The mathematical correctness of this is given by the discussion above.

Thus, by the principle of induction, we have shown that the algorithm is correct for all $n \in \mathbb{N}$.

4. We have to extend the problem of maximal points done in class to 3 dimensions.

(i). We want to create a data structure to determine if a given 2D point is below or above the staircase, in $O(\log n)$ time. Deletion of each maximal point should also be $O(\log n)$ time.
For this, choose any Balanced Binary Search Tree, i.e Red-Black trees, or AVL trees. For they key, we can use the first coordinate of the point. We can store the points in the tree as well, and the ordering of the points is imposed by the first coordinate. This tree can now be extended to have a search function for nearest upper (on the first axis), in $O(\log n)$ time.

To find if a point $p = (p_1, p_2)$ is below or above the staircase, we can use the search function to find the nearest upper point on the x-axis, say $q = (q_1, q_2)$. Finding q takes $O(\log n)$ time. If q does not exist then this point is a maximal point by definition, as no point dominates this point on the first axis. Otherwise, we can simply check if $q_2 \geq p_2$. If $q_2 \geq p_2$, then p is above the staircase, otherwise it is below the staircase.

To delete a maximal point, we can simply delete it from the tree. This takes $O(\log n)$ time, due to the properties of the class of trees we are using.

(ii). Implementing the line sweep.

We are given a set of n points, (x_i, y_i, z_i). We want to find the maximal points in this set. First, we sort them by the x-coordinate. This takes $O(n \log n)$ time. Now, we can use the line sweep algorithm to find the maximal points. Also, since it is in 3D, it is a plane sweep.

The algorithm is as follows:

1. Sort the points by the x-coordinate.
2. Create an empty set S to store the points.
3. Initialize a tree as mentioned above. Let it be empty as of now.
4. Traverse the points in a descending order of x-coordinate. We can do this since we sorted the points by the x-coordinate.
5. Given a point $p_i = (x_i, y_i, z_i)$, check if (y_i, z_i) is a maximal point in the current set of points in the tree.
6. If it is, insert it into the set S. In the tree, insert (y_i, z_i). Then, go over each element in the tree and check whether it is dominated by (y_i, z_i). If it is, delete it from the tree. Denote this number of deletions as d_i.
7. Otherwise, if it is not a maximal point, discard it.
8. Return the set S as the maximal points.

Proof of correctness:
Let's first establish the invariants for this algorithm.

1. Before and after traversion of each point, the tree always contains only maximal points in the 2D y-z plane at the current x-coordinate.
2. The set S always contains the maximal points in 3D until the current point is considered.

Initialization: When $n = 0$, the tree is empty, and S is empty. The invariants are trivially true.

Maintenance: We need to show that the invariants are maintained after the i^{th} point is considered. Before the execution, we know that the invariants hold up until the $(i-1)^{th}$ point. Now, we consider the i^{th} point.

Case 1: The i^{th} point is a maximal point.
If this is the case, then since the i^{th} point is already dominated on the x axis by all previous points (due to the descending order of x-coordinates), it must be a maximal point in the 2D y-z plane. This is exactly what the tree checks for. So, the point i will be inserted into S, and the y and z coordinates will be inserted into the tree.

Now what remains is to check if any of the points in the tree are dominated by (y_i, z_i). If they are, then they are deleted from the tree. This is done in a straightforward manner by traversing and comparing the points in the tree. There are d_i deletions at this step. After this process, the tree contains only maximal points.

So, the invariants are maintained.

Termination: When the algorithm terminates, all points have been considered. By the invariants, the set S contains all the maximal points in the 3D space. This is exactly what is required.

Thus, the algorithm is correct.

Time Complexity:
$T(n)$ = time to sort the points + time spent on each iteration of the sweep.
Sorting the points takes $O(n \log n)$ time.

At step i, the time taken is:
1. $O(\log n)$ time to check if the point is a maximal point in the 2D y-z plane.
2. $O(\log n)$ time to insert the point into the tree.
3. $O(d_i \log n)$ time to delete the dominated points from the tree.

So, the total time taken is $O(n \log n) + \sum_{i=1}^{n} O(\log n + \log n + d_i \log n) = O(n \log n + \sum_{i=1}^{n}(2 \log n + d_i \log n)) = O(n \log n + \sum_{i=1}^{n} d_i \log n)$.

d_i is simply the amount of deletions at step i. Now, we know that the number of deletions is at most n (since we can have at most n points to delete to begin with). So, the total time taken is $O(n \log n + n \log n) = O(n \log n)$.

Thus, the algorithm runs in $O(n \log n)$ time, and is able to find the maximal points in the 3D space.