Department of Computer Science
Ashoka University

Design and Analysis of Algorithms

Assignment 2

Collaborators: None **Name: Rushil Gupta**

Problem 1

Define the potential function of a state of the union-find that uses stars to represent subsets as:

$$\Phi = \Pi_{i=1}^{n}(|S_i| - 1)$$

where, S_i is the i-th subset. Notice that in the beging, each element is in its own subset, so $|S_i| = 1$ for all i. Thus, the potential function at the beginning is $\Phi = 0$. Also, since $|S_i| \geq 1$ for all i, we know that $\Phi \geq 0$ for all states. So this is a valid potential function.

Now, consider then we are merging S_j into S_i. Say that $|S_i| = n$ and $|S_j| = m$. Then, the potential functions before and after the merge are:

$$\begin{aligned}\Phi_{pre} &= C_1 \cdot (n-1) \cdot (m-1) \\ \Phi_{post} &= C_1 \cdot (n+m-1)\end{aligned}$$

This gives us the change in potential as:

$$\begin{aligned}\Delta\Phi &= C_1 \cdot (n+m-1) - C_1 \cdot (n-1) \cdot (m-1) \\ &= C_1 \cdot (n+m-1-(n \cdot m - n - m + 1)) \\ &= C_1 \cdot (2n + 2m - 2 - nm)\end{aligned}$$

Since we know we are merging S_j into S_i, we know that $n \geq m$. Thus, we can say that $n = m + k$ for some $k \geq 0$. Substituting this into the above equation, we get:

$$\begin{aligned}\Delta\Phi &= C_1 \cdot (2(m+k) + 2m - 2 - (m+k)m) \\ &= C_1 \cdot \left(2m + 2k + 2m - 2 - m^2 - km\right) \\ &= C_1 \cdot \left(4m + 2k - 2 - m^2 - km\right)\end{aligned}$$

For large values of m, the dominant terms are $-m^2$ and $-km$, both negative. This means that as sets grow larger, merging them results in a decrease in potential. The actual cost of performing a union is $O(m)$ since we need to update the parent pointer of all m elements of S_j.

$$\begin{aligned}\text{amortized cost} &= \text{actual cost} + \Delta\Phi \\ &= O(m) + C_1 \cdot \left(4m + 2k - 2 - m^2 - km\right) \\ &= cm + C_1 \cdot \left(4m + 2k - 2 - m^2 - km\right)\end{aligned}$$

For sufficiently large values of C_1, and since $m^2 + km > 4m + 2k - 2$ for large enough m and k, the change in potential is negative. This negative change in potential effectively "borrows" from the potential function to pay for the actual cost of the operation. Now, also see that in the begining, when we merge 2 singletons, the change in potential is 0, or extremely positive. This means that the potential function is effectively "pays" a high cost early on to "save" on costs later.

Now, each group can only be joined with larger groups $\log n$ times. This is because the size of the group doubles each time it is joined. Thus, the total cost of joining all groups is $O(n \log n)$. Since the potential function is always non-negative, the amortized cost of each UNION operation is $O(\log n)$.

Note: I don't even know what this was. A pretty bad argument that hopes that when we sum all the amortized costs, we get $O(n \log n)$ [no clue how :( ].

Problem 2

The Underlying Data Structure

We will be using multiple trees (or a 'forest') to implement the disjoint-set data structure. Initially, each element will be represented as a singleton tree. Each node has access to its parent. Also, each node has an optional field to store the size of the group. Also, a root will have its parent to itself, and any node that is not a root will have a parent that is not itself.

Note: The details mentioned above are critical as they act as invariants for our data structure.

Find Operation

The FIND operation locates the representative (root) of the set containing a given element. It does so by following parent pointers until it reaches a node that is its own parent. This procedure takes time proportional to the height of the tree. For now, say the time taken is $O(h)$, where h is the height of the tree.

Algorithm 1 Find Operation

```
function FIND($x$)
    if $x.parent \neq x$ then
        return FIND($x.parent$)
    else
        return $x$
    end if
end function
```

Union Operation

The UNION operation merges the subsets containing two elements, x and y. To ensure $O(1)$ time, we will assume that x and y are the roots of the trees containing the elements. The method will compare the sizes of the two trees and attach the smaller tree to the larger tree (choose arbitrarily if the sizes are equal).

Algorithm 2 Union Operation

```
function UNION($x, y$)
    ▷ Assume $x$ and $y$ are roots.
    if $x = y$ then
        return                                ▷ Already in the same set.
    end if
    if $x.size < y.size$ then                 ▷ We know $x.size$ and $y.size$ are non-null
        $x.parent \leftarrow y$
        $y.size \leftarrow y.size + x.size$
    else
        $y.parent \leftarrow x$
        $x.size \leftarrow x.size + y.size$
    end if
end function
```

Analysis of the Operations

Bounding the Cost of Find

Notice that a FIND operation involves following parent pointers from a node up to the root. We claim that for any tree of height h, the tree contains at least 2^h elements. We can prove this by induction:

Base Case: For a tree of height 0, the tree has exactly one node and $2^0 = 1$.

Inductive Hypothesis: Assume that every tree of height i has at least 2^i nodes, for $i = 0, 1, ..., h$ for some h.

Inductive Step: When two trees are united using the UNION operation, the smaller tree is attached to the larger tree. Let h_1, h_2 be the heights of the two trees. Let h be the height of the new tree. WLOG assume $h_1 \geq h_2$. Now, we have 2 cases:

Case 1: $h_1 > h_2$. Then, since h_1 and h_2 are integers, we know $h_1 \geq h_2 + 1$. Since we are attaching the tree with height h_2 to h_1, we know $h = h_1$. So, by our induction hypothesis, any tree of height h has at least 2^h nodes. Since we have only added more nodes to it, we are correct.

Case 2: $h_1 = h_2$. Then, the height of the new tree is $h_1 + 1$. By our induction hypothesis, we know that the tree of height h has at least 2^h nodes. So, our new tree has atleast $2^h + 2^h = 2^{h+1}$ nodes. So, this case is also correct.

Thus, we have shown that for any tree of height h, the tree contains at least 2^h elements. This implies that the height of the tree is at most $\log n$, where n is the number of elements in the tree. Thus, the time taken for a FIND operation is $O(\log n)$.

Bounding the Cost of Union

Observe that we assume that we are given the roots of the 2 trees we want to unite. So, just assinging a parent to the root of a node and updating a fixed amount of fields takes $O(1)$ time. Thus, the time taken for a UNION operation is $O(1)$.

Overall Cost

Performing N UNION operations and m FIND operations with n elements will require a total time of:

$$O\big(m \cdot \log n + N\big)$$

Proof of Correctness

By the construction of the algorithm, the FIND operation will always return the root of the tree containing the element, so it is correct. The UNION operation will always attach the smaller tree to the larger tree, and will maintain the properties of the data structure. Thus, the algorithm is correct.

Note: This is a hand-wavy proof of correctness, but it suffices to say that the algorithm is correct by construction.

Problem 3

We will first show how to lists S_0, S_1, ..., S_{k-1} in $O(\sum_{i=0}^{k-1} |S_i|)$ time, where S_j contains 2^j elements.

Algorithm 3 Merge Lists

```
1: function MERGE($S_0, S_1, ..., S_{k-1}$)
2:     $S \leftarrow S_0$
3:     for $i = 1$ to $k - 1$ do
4:         $S \leftarrow$ MERGE2($S, S_i$)
5:     end for
6:     $S \leftarrow S[1:]$                    ▷ We know that $-\infty$ is always in the start
7:     return $S$
8: end function
```

Here, MERGE2(S_1, S_2) is a function that merges 2 lists S_1 and S_2 in $O(|S_1|+|S_2|)$ time. This is our standard merge function, that we use in methods like mergesort.

We now want to show that we can merge k such lists in $O(2^k)$ time. We will prove this by induction.

Base Case: For $k = 2$, we have 2 lists S_0 and S_1. We can merge them in $O(2^2)$ time trivially (2 comparisions, and 3 insertions).

Inductive Hypothesis: Assume that we can merge k such lists in $O(2^k)$ time into a list of size $2^k - 1$.

Inductive Step: We want to merge $k + 1$ lists S_0, S_1, ..., S_k. We can merge the first k lists in $O(2^k)$ time by our inductive hypothesis.

Using this, we get another list of size $2^k - 1$. The problem is now reduced to merging 2 lists of size 2^k and $2^k - 1$. By our standard merge algoithm (like the one in mergesort), this can be done in $O(2^{k+1})$ time (since we have a total of $2^{k+1} - 1$ elements).

Thus, by induction, we have shown that we can merge k lists in $O(2^k)$ time, which is $O(\sum_{i=0}^{k-1} |S_i|)$.

Now, we want to show that we can insert an element in $O(\log n)$ on average when inserting n elements into an empty semi-dynamic list. Think of this problem as a bit counter problem, as we did for binomial heaps. So, a bit-string of length $\log n$ bits represents whether A_i exists if the i-th bit is 1. For example 1101 represents that A_0, A_1, and A_3 exist, while A_2 does not. Since we have $\log n$ bits, we can store upto $n-1$ elements.

Consider the insertion of a single element. We can think of this as adding 1 to a bit string which has t consecutive 1's on its right. The resulting number is a 1 on the $(t+1)$-th bit from the right, and 0 on all the first t bits. In our scenario, this can be thought of merging t lists of sizes 2^0, 2^1, ..., 2^{t-1}, and then adding the new element. As we saw earlier, this can be done in $O(2^t)$ time.

Now, when incrementing the counter from 0 to $n - 1$, each bit i is flipped a maximum of $n/2^i$ times. Thus, the total time taken to insert n elements is:

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \cdot 2^i = \sum_{i=0}^{\log n} n = n \log n$$

This shows that we can insert n elements in $O(n \log n)$ time in our semi-dynamic list. Thus, the average time taken to insert an element is $O(\log n)$. However, it is important to remember that each operation **does not** take $O(\log n)$ time. We know there will be insertions that take $O(n)$ time, but on **average**, the time taken is $O(\log n)$.

Problem 4

We will modify MoM to a new functon called WeightedMoM which takes in 2 inputs: the orginial array and a weights array. It is described below:

Algorithm 4 Weighted Median

```
1: function WeightedMoM($A, W, w_l = 0, w_r = 0$)
2:     $n \leftarrow |A|$
3:     if $n = 1$ then
4:         return $A[0]$
5:     end if
6:     $m \leftarrow$ MoM($A$)
7:     $L \leftarrow \{x \in A \mid x < m\}$
8:     $R \leftarrow \{x \in A \mid x > m\}$
9:     $w_{l2} = \sum_{x \in L} W[x]$
10:    $w_{r2} = \sum_{x \in R} W[x]$
11:    if $w_{l2} + w_l \leq 0.5$ and $w_{r2} + w_r \leq 0.5$ then
12:        return $m$
13:    end if
14:    if $w_{l2} + w_l > 0.5$ then
15:        return WeightedMoM($L, W, w_l, w_r + w_{r2} + W[m]$)
16:    else
17:        return WeightedMoM($R, W, w_l + w_{l2} + W[m], w_r$)
18:    end if
19: end function
```

Note: This assumes that all elements are distinct. The MoM is our standard median of medians algorithm.

Now, let's look at what the algorithm does. We first find the median of our original array. Then, we create 2 partitions of equal size: L and R. We then calculate the weights of the elements in L and R. If the sum of the weights of the elements in L and R is less than or equal to 0.5, then we return the median. Otherwise, we recurse on the partition with the higher weight. However, while doing this, we also want to keep a track of the weights of the partitions we have not considered. This is done using the w_l and w_r variables.

Observe that this is like the binary search algorithm on the array, and thus, the time complexity is given by:

$$\begin{aligned}
T(n) &= T(\frac{n}{c}) + O(n) \\
&= O(n) + (\frac{n}{c}) + O(\frac{n}{c^2}) + \cdots + T(1) \\
&= O(n) + (\frac{n}{c}) + O(\frac{n}{c^2}) + \cdots + O(1) \\
&\leq \left(\frac{1}{c-1}\right) \cdot O(n) = O(n)
\end{aligned}$$

Here, $c \approx 2$ (since MoM gives us the approximate median, but this recurrence holds for any fixed constant $c > 1$). So, our algorithm runs in $O(n)$ time.

Proof of Correctness

To show the correctness of the algorithm, we will use invariants. First, look at the trivial case: when $n = 1$. In this case, the median is the only element in the array, and thus, the algorithm returns the correct answer.

Now, consider the following invariants:

1. Let A_i be the partition we focus on during the i-th iteration (A at iteration i). Let b_i be the sum of the weights of the elements in A_i. Then, $b_i + w_l + w_r = 1$.
2. w_l is the weight of all the elements that are less than all elements in A_i.
3. w_r is the weight of all the elements that are greater than all elements in A_i.

Now we need to show that these invariants hold. An initialization, $w_l = w_r = 0$ which is correct since we haven't discarded any elements. Also, A contains all the initial elements, so their weights sum to 1. Thus, the invariant holds initially.

Now, I'll prove that these invariants are maintained through each recursive call. At the beginning of each iteration, we partition A_i into three parts: L (elements less than m), $\{m\}$ (the median), and R (elements greater than m). We denote:

- w_{l2} = sum of weights of elements in L
- w_{r2} = sum of weights of elements in R
- w_m = weight of the median element m

By our invariant assumption, $w_l + w_r + (w_{l2} + w_m + w_{r2}) = 1$ since $w_{l2} + w_m + w_{r2} = b_i$ (the sum of all weights in our current partition A_i).

We now have three possible cases to consider:

Case 1: If $w_{l2} + w_l \leq 0.5$ and $w_{r2} + w_r \leq 0.5$, then we have found the correct median since we know that w_l and w_r are correctly computed. So, we **terminate** here.

Case 2: If $w_{l2} + w_l > 0.5$, then we know that the current m cannot be the correct weighted median since elements that are less that m have more than 0.5 weight to them. So, we will recursively find the weighted median in L.

- Here, we set $w_r \leftarrow w_r + w_{r2} + W[m]$.
- We know that $w_l + w_r + w_{l2} + w_{r2} + W[m] = 1$, and we need $w_l + w_r + w_{r2} + W[m] + b_{i+1} = 1$.
- Since b_{i+1} is defined as the sum of the weights of all the elements in A at step $i + 1$, it is the sum of all elements of L. This means $b_{i+1} = w_{l2}$.
- So, all our invariants hold.

Case 3: If $w_{r2} + w_r > 0.5$. This is a symmetric argument to the previous case, and we will recurse on R. Clearly, our invariants hold here as well.

Finally, we want to show that our algorithm terminates. We know that w_r and w_l are increasing in each iteration. This is because we either do not change them, or add a positive number to them. We also know that they are bounded by 0.5 by the definition of our terminating case.

Moreover, our search space is appropriately halved in each iteration since we split using our approximate median (the larger idea is to know that the search space is reducing strictly). Thus, our algorithm will terminate by hitting a case and we can guarantee that $w_l \leq 0.5$ and $w_r \leq 0.5$.

Thus, we have shown that our algorithm is correct.

Problem 5

Problem 5.1

We know that S_1 and S_2 are sorted sequences. Let S_1^O and S_2^O be the odd numbered elements of S_1 and S_2 respectively. We know S^O is produced by merging S_1^O and S_2^O. We want to show that the smallest element of S^O is the smallest element in the entire set.

First, observe that the first elements of S_1 and S_2 are the smallest elements of their own sets, and are contained in S_1^O and S_2^O respectively. Thus, the smallest element of S^O is the smallest element of S_1^O and S_2^O. Since S_1^O has the smallest element of S_1 and S_2^O has the smallest element of S_2, the smallest element of S^O is the smallest element in the merged entire set.

Problem 5.2

Consider $\alpha_1, \alpha_2, ..., \alpha_n$. We know that we have started the interspersing with the first element of S^O, so α_1 is the smallest element in the entire set. We want to show that we can obtain a sorted sequence by comparing the pairs α_{2i} and α_{2i+1} independently.

Observe that this is equivalent to saying that $\forall i > 0$, $\alpha_{2i-1} \leq \alpha_{2i}$. This is because we also know $\alpha_1 \leq \alpha_3 \leq \cdots \leq \alpha_{2i-1}$ and $\alpha_2 \leq \alpha_4 \leq \cdots \leq \alpha_{2i}$. So, all we would need to do is compare the pairs α_{2i} and α_{2i+1} independently since those are the pairs we cannot make a conclusion about.

Proof by Induction:

Base Case ($i = 1$):
By the construction of the odd-even merge, the first element

$$\alpha_1 = \min(s_1, t_1),$$

where s_1 and t_1 are the first (smallest) elements of the two sorted sequences S_1 and S_2 respectively. The second element, α_2, is chosen from the even-indexed subsequences (i.e., from either s_2 or t_2). Since in each original sequence we have

$$s_1 \leq s_2 \quad \text{and} \quad t_1 \leq t_2,$$

it follows that

$$\alpha_1 \leq \alpha_2.$$

Thus, the base case holds.

Inductive Step:
Assume that for some $k \geq 1$ the property holds:

$$\alpha_{2k-1} \leq \alpha_{2k}.$$

We must show that

$$\alpha_{2k+1} \leq \alpha_{2k+2}.$$

Consider the way the merged sequence is constructed. At this stage:

- α_{2k+1} is the next element selected from the *odd* subsequences (i.e., from the merge of S_1^O and S_2^O)
- α_{2k+2} is the next element selected from the *even* subsequences (i.e., from the merge of S_1^E and S_2^E)

Since both S_1 and S_2 are sorted, the corresponding odd-even pairs in each sequence satisfy

$$(\text{for } S_1) \quad s_{2i-1} \leq s_{2i} \quad \text{and} \quad (\text{for } S_2) \quad t_{2i-1} \leq t_{2i}.$$

Moreover, the merging process preserves the internal order of the odd and even subsequences. Thus, when choosing the next elements, the algorithm selects:

- α_{2k+1} as the smallest among the remaining odd-indexed elements, and
- α_{2k+2} as the smallest among the remaining even-indexed elements.

Because in each original sequence the odd element is always less than or equal to its corresponding even element, the smallest available odd element (which becomes α_{2k+1}) is no greater than the smallest available even element (which becomes α_{2k+2}). Hence, we have

$$\alpha_{2k+1} \leq \alpha_{2k+2}.$$

So, by induction, we have shown that for all $i > 0$, $\alpha_{2i-1} \leq \alpha_{2i}$.

This means we have the following facts:

1. α_1 is the smallest element in the entire set.
2. $(\alpha_1 \leq \alpha_2)$, $(\alpha_3 \leq \alpha_4)$, $\cdots$, $(\alpha_{2k-1} \leq \alpha_{2k})$.
3. $\alpha_1 \leq \alpha_3 \leq \cdots \leq \alpha_{2k-1}$.
4. $\alpha_2 \leq \alpha_4 \leq \cdots \leq \alpha_{2k}$.

Now, we can conclude that just be comparing the pairs α_{2i} and α_{2i+1} independently, we can obtain a sorted sequence.

Problem 5.3

Now, using this odd-even merge method, we will desing a sorting algorithm. We will call this the OddEvenMergeSort algorithm. It is described below:

Algorithm 5 Odd-Even Merge Sort

```
function OddEvenMergeSort($A$)
    $n \leftarrow |A|$
    if $n = 1$ then
        return $A$
    end if
    $L \leftarrow A[1 : n/2]$
    $R \leftarrow A[n/2 + 1 : n]$
    $L \leftarrow$ OddEvenMergeSort($L$)
    $R \leftarrow$ OddEvenMergeSort($R$)
    return OddEvenMerge($L, R$)
end function
```

Proof of Correctness

We will prove that the method OddEvenMergeSort is correct by induction by showing that an array of length n for all n.

Base Case: For $n = 1$, the array is already sorted, so the algorithm returns the correct answer.

Inductive Hypothesis: Assume that the algorithm works for all $n \leq k$ for some k.

Inductive Step: We want to show that the algorithm works for $n = k + 1$. We first partition our array A into 2 arrays L and R of equal size (their sizes differ by upto 1). Then, we know we can sort arrays of size $\frac{k}{2}$ and $\frac{k}{2} + 1$ using our inductive hypothesis. We then merge the two sorted arrays using the OddEvenMerge algorithm. We know that the OddEvenMerge algorithm works correctly (as seen in part b), and thus, the algorithm works for $n = k + 1$.

Time Complexity

The time complexity of the OddEvenMergeSort algorithm is given by the following recurrence:

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

This is because of the following steps:

1. Partition into 2 arrays in $O(n)$ time
2. Recurse on the 2 arrays in $T\left(\frac{n}{2}\right)$ time each $\implies 2T\left(\frac{n}{2}\right)$ time
3. Merge the 2 arrays in $B(n)$ time using the OddEvenMerge algorithm

So, we will first analyse the time complexity of the OddEvenMerge algorithm. We know that the OddEvenMerge algorithm works by splitting the 2 sorted arrays into odd and even indexed elements, and then merging them. So, the time complexity of the OddEvenMerge algorithm is given by:

$$B(n) = 2B\left(\frac{n}{2}\right) + O(n)$$

This is a standard recurrence and we know this resolves to $B(n) = O(n \log n)$. Thus, the time complexity of the OddEvenMergeSort algorithm is given by:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + O\left(n \log n\right) \\
&= 4T\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\log\frac{n}{2}\right) + O(n \log n) \\
&= 2^k T\left(1\right) + \sum_{i=0}^{k-1} 2^i O\left(\frac{n}{2^i}\log\frac{n}{2^i}\right) = 2^k + \sum_{i=0}^{k-1} O\left(n \log \frac{n}{2^i}\right)
\end{aligned}$$

We know that $k = \log n$, so the time complexity of the OddEvenMergeSort algorithm is:

$$\begin{aligned}
T(n) &= 2^k + \sum_{i=0}^{k-1} O\left(n \log \frac{n}{2^i}\right) \\
&= n + cn \sum_{i=0}^{k-1} \left(\log \frac{n}{2^i}\right) \\
&= n + cn \sum_{i=0}^{k-1} (\log n - i) \\
&= n + cn \left[k \log n - \sum_{i=0}^{k-1} i\right] \\
&= n + cn \left[k \log n - \frac{k(k-1)}{2}\right] \\
&= n + cn \left[\log n \cdot \log n - \frac{\log n \cdot (\log n - 1)}{2}\right] \\
&= n + cn \left[\log^2 n - \frac{\log^2 n - \log n}{2}\right] \\
&= n + cn \left[\frac{\log^2 n + \log n}{2}\right]
\end{aligned}$$

This gives us a time complexity of $O(n \log^2 n)$. Clearly, this is not optimal, since we know that the time complexity of the merge sort algorithm is $O(n \log n)$. Thus, the OddEvenMergeSort algorithm is not optimal.