Department of Computer Science
Ashoka University

Design and Analysis of Algorithms: CS-3210-1

Assignment 2

Name: Dhruman Gupta

---

1. We have the union-find problem. We have n elements in a set S, which are each part of k disjoint sets S_i, such that $\cup_{i=1}^{k} S_i = S$.

   We are using "stars" to represent the sets S_i. Each element in S_i is connected to the root of S_i directly. Each `find` operation simply returns the parent of the element. A `union` operation takes two sets S_i and S_j and takes all elements of one set and puts them under the root of the other set. This maintains the invariant that each element is connected to the root of its set.

   Now, we saw in class that if we have m find operations and N union operations, the amortized time complexity of each operation is $O(m + N \log n)$. Note that $N \leq n - 1$ because after that no more unions are possible.

   Now, we want to do amortized analysis of the union-find problem using a potential function.

   TODO: Add analysis here

2. We want to describe a tree based algorithm for the above problem. The parameterization of the problem is the same as above. We have m find operations and $N \leq n-1$ union operations. However, we are using a tree based algorithm. Below are the implementation details of the algorithm. The problem setup is as follows:

   Each set is represented by a tree. The root of the tree is the representative of the set. Each element is connected to its parent. When we do a union, we put one of the root of the trees under the root of the other tree. When we do a find, we return the root of the tree.

   Union Algorithm: Given two trees T_a and T_b, we want to union them.

   1. Let R_a and R_b be the ranks of T_a and T_b respectively (the heights of the trees).
   2. WLOG, assume $R_a \leq R_b$.
   3. Now, attach the root of T_a to the root of T_b.
   4. We have a new tree with root R_b, which represents the new set.

   Find Algorithm:

   1. Start from the given element.
   2. The parent of this element is the set S_i that the element was originally part of.
   3. While there is a parent, move to the parent.
   4. The root of the tree is pointer to the representative of the set.

   Time Complexity Analysis:
   Each `union` operation takes a constant time, as it is simply adding a pointer. So, each union takes $O(1)$ time. Now, we want to analyze the time taken by each `find` operation. For this, we will first do analysis on the height of the tree.

   Claim: Each tree T_i with rank $= k$, the algorithm deals with, has at least 2^k elements in it. I will prove this using induction.

   Base Case: When $k = 0$, the tree has only one element, which is the root. So, the claim holds.

   Inductive Hypothesis: Assume that the claim is true for all trees with rank k. Now, we need to show that the claim is true for all trees with rank $k+1$.

   Inductive Step: Consider a tree T_i with rank $k+1$. By the union algorithm above, the tree with the smaller rank is attached to the root of the tree with the larger rank. Now, note the following:

   - If the rank of the smaller tree is not equal to the rank of the larger tree, then the rank of the new tree is the same as the rank of the larger tree.

   So, in fact, if we have a tree with rank $k+1$, it means that it is constructed by doing a union on two trees with rank k. By the inductive hypothesis, each of these trees had at least 2^k elements in it. Therefore, the new tree has at least $2 * 2^k = 2^{k+1}$ elements in it.

   Thus, the Number of elements in the tree $T_i \geq 2^{k+1}$.

   So, by the principle of induction, the claim is true for all trees with rank k. What this tells us is that any tree T_i at any state in the algorithm has at least $2^{\text{rank of } T_i}$ elements in it. ■

   Now, for a tree T with rank r, let N be the number of elements in the tree. We have $N \geq 2^r$. From this, we get that $r \leq \log N$. N is at most n, so $r \leq \log n$.

   This means that the maximum height of any tree is bounded by $O(\log n)$. Let's quickly analyze the time taken by each `find` operation.

   $T(h) = T(h-1) + O(1)$, where h is the height of the tree.

   This is a telescoping series, so we get that $T(h) = O(h)$. Since the maximum height is $O(\log n)$, we get that each `find` operation takes $O(\log n)$ time. With m find operations, and N union operations, the total time taken is $O(m \log n + N \log n) = O(m \log n + N)$.

Proof of Correctness:
The `find` operation is correct because it returns the root of the tree, which by definition is the representative of the set.

The union algorithm is correct because the union attached the root of one tree under the root of another. It suffices to say that once the union is done, all elements of both the sets will return the same root when a `find` operation is done.

This proof is not written formally, but the formulation of the algorithm and the way it models the problem makes the proof trivial.

3. To prove that the ammortized cost of insertion is $O(\log n)$, I will first show the following. Given the following sorted sets:

$$A_0, A_1, \ldots, A_{k-1}$$

Where $|A_i| \leq 2^i$, we can merge all of them into a sorted array of size $N = \sum_{i=0}^{k-1} |A_i|$ in $O(N)$ time. Before any of this, let's show that $O(N) = 2^k - 1$.

$|A_i| = 2^i \implies N = \sum_{i=0}^{k-1} 2^i = 2^k - 1 = O(2^k)$.

Algorithm:

1. Let $i = 0$.
2. First merge A_i and A_{i+1} to get a sorted array B_{i+1}. This can be done by comparing the first elements of the two arrays each insertion.
3. Now, merge B_{i+1} and A_{i+2} to get B_{i+2}.
4. Continue this until we have merged all the arrays.
5. When the algorithm terminates, B_k is the merged array, with $L_k = N$.

Proof of Correctness:
Invariant: $0 \leq i \leq k$, and B_i is a sorted array consisting of all the elements of $A_0,\ A_1, \ldots, A_{i-1}$ and $|B_i| = \sum_{j=0}^{i-1} |A_j|$.

Base Case: When $i = 0$, B_0 is (by convention) an empty array. This trivially satisfies the invariant, as B_0 is the merged array of all the arrays in the set ϕ.

Inductive Hypothesis: Assume that the invariant is true for some i.

Inductive Step: We need to show that the invariant is true for $i + 1$.

Consider the array B_i. It is a sorted array consisting of all the elements of $A_0,\ A_1, \ldots, A_{i-1}$. Now, we merge B_i and A_i to get B_{i+1}. We look at By the inductive hypothesis, B_i is a sorted array. Since A_i is also a sorted array, B_{i+1} is also a sorted array.

Also, B_{i+1} consists of all the elements of $A_0,\ A_1, \ldots, A_{i-1}, A_i$ (the elements of A_i are just added).

So, B_{i+1} is a sorted array consisting of all the elements of $A_0,\ A_1, \ldots, A_{i-1}, A_i$, and $|B_{i+1}| = |B_i| + |A_i| = \sum_{j=0}^{i-1} |A_j| + |A_i| = \sum_{j=0}^{i} |A_j|$.
Thus, the invariant is true for $i + 1$. By the principle of induction, the invariant is true for all i. ■

At **termination**, $i = k$, and B_k is a sorted array consisting of all the elements of $A_0,\ A_1, \ldots, A_{k-1}$. This is the merged array of all the arrays in the set. So the algorithm is correct.

Now, let's see how long the algorithm takes. Merging two sorted arrays of length a, b takes $O(a + b)$ time. Let c_i denote the cost at the i-th iteration. We have:

$$\begin{aligned}
c_0 &= 1 \\
c_1 &= |A_0| + |A_1| = 1 + 2 \\
c_2 &= |A_0| + |A_1| + |A_2| = 1 + 2 + 4 \\
&\vdots \\
c_{k-1} &= |A_0| + |A_1| + |A_2| + \cdots + |A_{k-1}| = \sum_{i=0}^{k-1} 2^i
\end{aligned}$$

Each $c_i = \sum_{j=0}^{i} 2^j$. This is a geometric series, so we get that $c_i = 2^{i+1} - 1$. The total cost is $\sum_{i=0}^{k-1} c_i = \sum_{i=0}^{k-1} (2^{i+1} - 1) = \sum_{i=i}^{k} 2^i - k - 1$.

This is also a geometric series, so we get that $\sum_{i=i}^{k} 2^i - k - 1 = 2^k - k - 3$. This is $O(2^k)$, which is $O(N)$ (as shown above). So, we can merge in $O(N)$ time. ■

Now, we can use this to show that the amortized cost of insertion is $O(\log n)$.

Now to show that the amortized cost of insertion is $O(\log n)$. Let's first recall the argument done in class for the bit counter. Say you increment a bit counter from 0 to n one-by-one. This bit counter will have $\log n$ bits in it. Each bit b_i changes at most $n/2^i$ times. So, the total cost of incrementing the bit counter is $\sum_{i=0}^{\log n} \frac{n}{2^i} = O(n)$.

Making the same argument as above, we not have the case that insertion in the arrays takes $O(2^j)$ time instead of $O(1)$, where j is the array A_j where the merging process terminates. This time bound is proven from the argument above.

So, our amortized cost here is just each term multiplied by $O(2^j)$:

$$\begin{aligned} &= \sum_{i=0}^{\log n} O(\frac{n}{2^j} \cdot 2^j) \\ &= O(\sum_{i=0}^{\log n} n) \\ &= O(n \log n) \end{aligned}$$

So, inserting n elements takes $O(n \log n)$ time. So, the amortized cost of insertion for each element is $O(\log n)$. ■

4. We are given a set of n elements, and a set of weights $\omega = w_1, w_2, \ldots, w_n$, s.t $\sum_{i=1}^{n} w_i = 1$. The weighted median x_k is defined as:

$$\sum_{i|x_i \leq x_k} w_i \geq \frac{1}{2} \text{ and } \sum_{i|x_i \geq x_k, i \neq k} w_i \geq \frac{1}{2}$$

We want to find the weighted median in $O(n)$ time. The algorithm is as follows:

1. Initialize the parameter w_l, w_r as 0.
2. Find the unweighted median of the set, in $O(n)$ time. This can be done by using an algorithm like the Median-of-Medians algorithm. Call this x_m.
3. This effectively partitions the set into two, say $P_<, P_>$. Calculate the sum of the weights of the elements in each partition, call them $s_<, s_>$.
4. Now, if $w_l + s_< \leq \frac{1}{2}$ and $w_r + s_> \leq \frac{1}{2}$, then return the median.
5. If $w_r + s_> \geq \frac{1}{2}$, then set $w_l = w_l + s_<$ and recursively find the weighted median of the set $P_<$.
6. If $w_l + s_< \geq \frac{1}{2}$, then set $w_r = w_r + s_>$ and recursively find the weighted median of the set $P_>$.

Time Complexity Analysis: The median of medians has the property that it will give an approximate median. A standard bound is $\frac{3n}{10} \leq r \leq \frac{7n}{10}$. So the rank of the median will be between this range.

$$\begin{aligned} T(n) &\leq T(7n/10) + O(n) \\ &\leq T((\frac{7}{10})^2 n) + O(7n/10) + O(n) \\ &= O(n) \end{aligned}$$

So, the time complexity of the algorithm is $O(n)$.

Proof of Correctness:
Let's prove the correctness using invariants. At each iteration of the algorithm:

Invariant 1: At every recursive step, the candidate set C that we continue processing contains the weighted median x^* by the inductive hypothesis.

Invariant 2: $w_l, w_r \leq \frac{1}{2}$. $w_l = \sum_{i=1}^{j} w_i$, $w_r = \sum_{i=k+1}^{n} w_i$, where j, k is the range of the indices of the elements in the candidate set C.

Base Case: When initialized, the candidate set C contains all the elements, so the invariant is trivially true. $w_r, w_l = 0$, and $j = 1$, $k = n$. So, the invariant is true as there are no weights to add.

Inductive Hypothesis: Assume that the invariant is true at some step i.

Inductive Step: We have a w_l, w_r, and a candidate set C. We know that $w_l < \frac{1}{2}$, $w_r < \frac{1}{2}$, and C contains the weighted median x^*.

We partition C into $P_<, P_>$ by using the unweighted median x_m of C. Call the sums of the weights of the elements in $P_<, P_>$ as $s_<, s_>$ respectively.

Case 1: $w_l + s_< \leq \frac{1}{2}$ and $w_r + s_> \leq \frac{1}{2}$. In this case, we return x_m as the weighted median. This is indeed the weighted median, by definition.

Case 2: $w_r + s_> \geq \frac{1}{2}$. In this case, we set $w_l = w_l + s_<$ and recursively find the weighted median of the set $P_<$. Since the sum of the weights is 1, $w_l + s_< \leq \frac{1}{2}$. Also, since $w_r + s_> \geq \frac{1}{2}$, we know that the weighted median must lie in $P_>$.

We can show this quickly using contradiction. Say the weighted median x^* lies in $P_<$. Then, $w_r + s_< + \epsilon \leq \frac{1}{2}$ where ϵ is the pending elements to consider. This implies that $w_r + s_< < \frac{1}{2}$. This contradicts the fact that $w_r + s_> \geq \frac{1}{2}$. So, indeed the weighted median must lie in $P_>$.

Case 3: $w_l + s_< \geq \frac{1}{2}$. In this case, we set $w_r = w_r + s_>$ and recursively find the weighted median of the set $P_>$. The argument is similar to the above case.

So, by the principle of induction, the invariant is true for all steps.

Termination: The algorithm terminates when $w_l + s_< \leq \frac{1}{2}$ and $w_r + s_> \leq \frac{1}{2}$. At this point, we return the median. This is indeed the weighted median. ■

5. We are solving the odd-even mergesort problem.

   (a) Prove that the smallest element of S^O is the smallest element in the entire set.

   S^O is the result of interspersing the sequences S_1^O and S_2^O. Now, S_i^O has the odd-indexed elements of S_i.

   Claim: S_i^O and S_i^E are sorted.
   Proof: S_i is sorted. S_i^O takes the odd-indexed elements of S_i. S_i^E takes the even-indexed elements of S_i. Since S_i is sorted, the odd-indexed elements are also sorted, and the even-indexed elements are also sorted. This is because eevry next odd-numbered element has a bigger index than before and thus is a bigger element. The same goes for the even-indexed elements.

   Now, we know S_i^O and S_i^E are sorted. So, the first element of S_i^O is the smallest element of S_i. S^O has the elements of both S_1^O and S_2^O. So, the first two elements of S^O are the smallest elements of S_1^O and S_2^O. Thus, the smallest element of S^O is smallest element of S_1 and S_2 (the entire set).

   (b) If we show that $\forall i, \alpha_{2i+1} \leq \alpha_{2i+2}$, then we can show that the sequence only needs to be sorted in pairs at α_{2i} and α_{2i+1}.

   Claim: Let $S^O = [a_1, a_2, \ldots, a_n]$ and $S^E = [b_1, b_2, \ldots, b_n]$. Then, $\alpha_{2i-1} \leq \alpha_{2i}$ for all i. This is saying $a_i \leq b_i \forall i$.

   Proof: We will prove this using induction.

   Base Case: $n = 1$. We have the sets $S^O[1...2]$ and $S^E[1...1]$. These are just the singleton sets. The claim is trivially true here, as shown in part (a).

   Inductive Hypothesis: Assume that the claim is true for some k, i.e there exists $S^O[1...k]$ and $S^E[1...k]$ s.t $a_i \leq b_i \forall i \leq k$.

   Inductive Step: Consider the set $S^O = [a_1, a_2, \ldots, a_{k+1}]$ and $S^E = [b_1, b_2, \ldots, b_{k+1}]$. By the induction hypothesis, we know that $a_i \leq b_i \forall i \leq k$. Now, we need to show that $a_{k+1} \leq b_{k+1}$.
   We have two cases here.

   Case 1: a_{k+1} and b_{k+1} come from the same set S_i. If they do, then, there exists a p s.t. $a_{k+1} = S_{i,2k-1}$ and $b_{k+1} = S_{i,2k}$. Since S_i is sorted, we have that $a_{k+1} \leq b_{k+1}$.

   Case 2: They come from different sets. WLOG assume a_{k+1} comes from S_1 and b_{k+1} comes from S_2. Now, for some k, $a_{k+1} = S_{1,2k-1}$. Since S_1 is sorted, we have that $a_{k+1} \leq S_{1,2k}$.
   Now, b_{k+1} is the $k+1$th smallest element of all the even-indexed elements. In particular, we get $b_{k+1} \geq \min\{S_{1,2k}, S_{2,2k}, \ldots, S_{2,2n}\}$. Now, both S_1 and S_2 are sorted. $b_{k+1} \geq S_{1,2k} \geq a_{k+1}$.
   This gives us that $a_{k+1} \leq b_{k+1}$.

   By the principle of induction, we have that $\alpha_{2i-1} \leq \alpha_{2i}\ \forall i$.

   Now, we know that for the sequence α, we have that $\alpha_{2i+1} \leq \alpha_{2i+2}\ \forall i$. We also know that $\alpha_{i-2} \leq \alpha_i \leq \alpha_{i+2}\ \forall i$. This is derived from the fact that $\alpha i - 2, \alpha i, \alpha i + 2$ lie in the same set and in the same order and are thus sorted.

   Combining these two, we get that:

$$\alpha_i \leq \alpha_{i+2} \leq \alpha_{i+4}\ \forall i$$
$$\alpha_{i+1} \leq \alpha_{i+2}\ \forall i = 2k$$

   Next page.

Applying this to k and $k+1$, we get that (assume k is even):

$$\begin{aligned}\alpha_k &\leq \alpha_{k+2} \leq \alpha_{k+4}\\ \alpha_{k+1} &\leq \alpha_{k+2}\\ \alpha_{k+3} &\leq \alpha_{k+4}\\ \alpha_{k+1} \leq \alpha_{k+3} &\leq \alpha_{k+5}\end{aligned}$$

So, all we have to do is show that $\alpha_{2i} \leq \alpha_{2i+1}\ \forall i$ (1), and it will follow that α is sorted. For this case, we will have:

$\alpha_k \leq \alpha_{k+1} \leq \alpha_{k+2} \leq \alpha_{k+3} \leq \alpha_{k+4} \leq \alpha_{k+5}$ which is the definition of a sorted sequence.

Now, all the other inequalities already hold. We just need to ensure that (1) holds. This is exactly what the question is asking us to show. ■

(c) To implement a odd-even mergesort, we can use the following algorithm:

1. Given a set S, if $|S| = 1$, return S.
2. Otherwise, divide it into S_1 and S_2, with equal (or +1) number of elements.
3. Sort S_1 and S_2 using odd-even mergesort.
4. Merge the two sorted arrays using the odd-even merge algorithm:
   A. First partition into S_1^E, S_1^O and S_2^E, S_2^O.
   B. Recursively merge S_1^E, S_2^E to get S^E and S_1^O, S_2^O to get S^O.
   C. Intersperse S^E and S^O to get a sequence α.
   D. Check every pair $\alpha_{2i} \leq \alpha_{2i+1}\ \forall i$. Swap them if $\alpha_{2i} > \alpha_{2i+1}$.
   E. Return α.

Proof of Correctness:
Base Case: When $|S| = 1$, the set is already sorted.

Inductive Hypothesis: Assume that the algorithm is correct for all sets of size less than n.

Inductive Step: Consider a set S of size n. We partition it into S_1 and S_2, with equal (or +1) number of elements. By the inductive hypothesis, we know that S_1 and S_2 can be correctly sorted recursively. Now, we need to show that the merge step works.

With the analysis in part (b), we know that α is sorted after the $n/2$ comparisons. So, the merge step works.

This completes the proof of correctness. ■

Time Complexity Analysis:

Time taken = Time taken to sort S_1 + Time taken to sort S_2 + Time taken to merge S_1 and S_2 + Time taken to intersperse S^E and S^O + Time taken to check every pair $\alpha_{2i} \leq \alpha_{2i+1}\ \forall i$.

(a) Time taken to sort S_1 and S_2 is $T(n/2)$ each.

(b) Time taken to merge S_1 and S_2 is $O(n)$ as it is n steps with a constant time comparison at each step.

(c) Time taken to intersperse S^E and S^O is $O(n)$ as it is n steps with a constant time insertion at each step.

(d) Time taken to check every pair $\alpha_{2i} \leq \alpha_{2i+1}\ \forall i$ is $O(n)$ as it is $n/2$ steps with a constant time comparison at each step.

So, the total time taken is $T(n) = 2T(n/2) + O(n) + O(n) + O(n) = 2T(n/2) + O(n) = O(n \log n)$. ■