CSE 331: Introduction to Algorithm Analysis and Design

Divide and Conquer

1 Sorting

1.1 Problem

Input: An array a containing $a_1, a_2, \ldots, a_n$ where each $a_i \in \mathbb{R}$.

Output: An array containing $b_1, b_2, \ldots, b_n$ where the values are the same as the input but are sorted in non-decreasing order.

1.2 Merge-Sort

mergeSort(a, n):

```
1. if $n == 1$: return $a_1$
2. if $n == 2$: return $(min(a_1, a_2), max(a_1, a_2))$
3. $a_L = a_1, \ldots, a_{n/2}$
4. $a_R = a_{n/2+1}, \ldots, a_n$
5. return $merge(mergeSort(a_L, n/2), mergeSort(a_R, n/2))$
```

Assume a and b are both sorted arrays
merge(a, b):

```
1. $i = 1$
2. $k = 1$
3. $c = []$
4. while $i <= |a|$ and $k <= |b|$
    (a) if $a_i < b_k$:
        i. add $a_i$ to $c$
        ii. $i++$
    (b) else:
        i. add $b_k$ to $c$
        ii. $k++$
5. if $i > |a|$:
    (a) add $b_k, \dots, b_{|b|}$ to $c$
6. if $k > |b|$:
    (a) add $a_i, \dots, a_{|a|}$ to $c$
7. return $c$
```

1.3 Correctness

Proof. We will prove the correctness by induction following the structure of the recursion

Base case: For $n = 1$ there is only one element with only one ordering so it must be sorted. The algorithm correctly returns the only order. For $n = 2$ the algorithm checks both possible orderings and manually chooses the correct order by return the min then the max of the two elements.

Induction step: We will assume that the algorithm is correct for all n such that $1 \leq n \leq k$ and show that it is also correct for $n = k + 1$. For a call of *mergeSort* on an input of size $k + 1$ that is not covered by a base case the algorithm calls *merge* on the return of two calls of *mergeSort* each with half of the array. These two calls to *mergeSort* are assumed to be correct by our inductive hypothesis so we only have to prove that *merge* is correct when the inputs are both sorted. For this, the algorithm is starting with the smallest element in both and adding the min of them to the result. This must then min the overall minimum value in both arrays. The algorithm then advances the appropriate points and takes the min of the two min values that have not yet been added to the output array which must be the min element of the remaining elements. Once the min remaining elements of one of the arrays is greater than the last (max) elements in the other array we can copy the rest of the elements from that list to the output since all those elements are larger than every element in the other list and the entirety of the other list is already in the output. (Note: For more formality *merge* could also be proven by induction)

Since the algorithm is correct for any size input then it must be correct for any input of size n. □

1.4 Runtime

For the runtime analysis we need to observe the recurrence relation of the algorithm. **Base case:** In

the base case there is always a constant amount of work performed since none of the computations depend on the original input size n so the base case takes $O(1)$ time.

Recursive step: In the inductive step we perform $T(n) = O(n) + 2T(n/2)$ where the $O(n)$ term is the time taken to call *merge* and $2T(n/2)$ is the time to make 2 recursive calls of size $n/2$

Since the input is being cut in half after each recursive call we can say that after i levels in the recursion we have 2^i calls of size $\frac{n}{2^i}$. We will reach the base case when $2^i = n$ or $i = \log n$ so we have $\log n$ layers of recursion each doing $2^i \cdot \frac{O(n)}{2^i} = O(n)$ work for an overall runtime of $O(n \cdot \log n)$.