Analysis of Algorithms: Homework 1 Corrections

Owen Bezick

Due date: March 25th, 23:55

1 Master Method Proof

- **What are the general roles of Sections 3.1, 3.2, 3.3?**
  The general roles of these sections are to provide us with a proof showing that the variable part of our algorithm, meaning the part of the algorithm that is greatly effected by the size of the problem at each level and the number of sub problems at each level, has only 3 possible outcomes for it's run-time depending on the run-time of our constant rebuilding function $f(n)$. By generalizing an infinite amount of $T(n)$'s to only 3 possible outcomes, we have gained the power to analyze nearly any algorithm given to us. Each section proves one of the 3 outcomes. Although there are 4 cases, case 2 is simply a special condition of case 4. These sections give us the confidence to use the Master Method for any $T(n)$. Section 3.1 gives us a proof for Case 1. Section 3.2 gives us a proof for Case 4 which also proves the special case for Case 2 where k = 0. Section 3.3 gives us a proof for Case 3.

- **Why does section 3.2 need two subsections? What are each of them doing?**
  The first half of section 3.2 is providing an upper bound for the variable part $V(n)$ in our $T(n)$ function. The proof begins with the expression that we obtained from Tree Analysis of $T(n)$. After a few algebraic steps the proof splits into two parts. In 3.2.1, the proof continues to provide an upper bound by say that our summation is greater than or equal to our variable part $V(n)$. Section 3.2.2 picks up from Section 3.2.1 before we attempted the upper bound so that we can attempt a lower bound. By finding an expression $g(n)$ such that $T(n) = O(g(n))$, and then reaching the same conclusion when looking for a lower bound, we can say that $T(n) = \Theta(g(n))$ where $g(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. The fact that we can start from the same point, and then

diverge on two separate paths under different assumptions (finding lower bound and finding upper bound), then we can be confident that the expression we find correctly represents our $T(n)$'s growth.

- **How do we go from steps 14 to 15, 18 to 19, 23 to 24, and 35 to 36?**
  In step 14, we have an expression $\log^k(n/b^i)$. Using our log rules, we know that this expression is the same as $\log^k n - \log^k b^i$. In step 14, we are equating $V(n)$ to our summation. But once we reach step 15, we turn it into an inequality saying that $V(n)$ is less than or equal to our summation. When we make this transition, we drop the term $\log^k b^i$. As i approaches infinity, $\log^k b^i$ will become more and more positive. By removing this subtracting term, we are acknowledging that our summation will be bigger than what it would be with this term still in the summation. Since we are searching for an upper bound, this is the correct assumption to make because we are treating the summation as a value greater than what it would normally be.
  Step 18 starts from step 14, the last step where we had an equality rather than an inequality. In line 19 we make the decision to sum up only half of our tree for the variable part $V(n)$. **We also replace the i with the value $(\log_b n - 1)/2$. This value is the size of the last level in our top-half tree. By using this value rather than summing through i, we are accounting for the largest part of our half-tree so we can be sure that it is less than $V(n)$.** From 14 to 15, we made the decision to estimate our summation higher that what it actually is so we could find a value higher than $V(n)$. From 18 to 19, we are limiting our summation to only halfway through the tree so that we can find a value lower than $V(n)$. The goal of this is to have our higher value and our lower value to have the same growth rate so we can sandwich in $V(n)$ and find its growth rate.
  In line 22, we simplified our expression by using some arbitrary constant d with the stipulation that $d = b^{1/2} > 1$. With this simplifying constant we can make the approximation that $\log^k dn^{1/2} \geq \log^k n^{1/2}$. Using our log rules, we can assess the transition from 23 to 24. The expression,

$$\log^k n^{1/2} = (\log n^{1/2})^k = (\frac{1}{2} * \log n)^k$$

  Putting our base to the power k, we arrive at:

$$\frac{1^k}{2^k} * \log^k n = \frac{1}{2^k} * \log^k n$$

  Which is the expression we find in step 24.

On line 35, the cap for i is a finite value. This value is acquired from our Tree Analysis of $T(n) = aT(n/b) + f(n)$. We calculate that the problem size at each level is $\frac{n}{b^i}$ where i is the ith level and that size of the final level is 1. For our final level x,

$$\frac{n}{b^x} = 1 \Rightarrow x = \log_b n$$

Meaning that there are $log_b n$ levels in our tree. No matter the size of n, the number of levels in our tree will always be finite. We then make a simplifying assumption based on the identity that we proved in Quiz 1. Since our tree has finite levels, our summation has a finite cap for i. Naturally, if our summation had an infinite cap for i, the summation with a finite cap would be less than the summation with an infinite cap. By creating an inequality with a value greater than ours, we are establishing an upper bound. The infinite summation allows us to use our identity from Quiz 1, simplifying the expression:

$$f(n) * \sum_{i=0}^{\infty} c^i \Rightarrow f(n) * \left(\frac{1}{1-c}\right)$$

This upper bound allows us to come to the conclusion that $V(n) = O(f(n))$.

2 Karatsuba

2.1 A first attempt

- If $m = n$, what is the recurrence relation and cost of the algorithm above? The assembling function has to sum 4 times per multiplication. This equates to 4^i summations on the $i - th$ level. The amount of summations is always 4 per sub-problem, and the amount of summations per level has no dependence on n. Therefore, our assembly function is $f(n) = O(1)$. The recurrence relation with floors and ceilings accounted for looks like the following: The pr multiplication takes two ceilings, the ps and qr both take a combination of floors and ceilings, and then qs takes two floors. This is because ceilings partition to the left and floors partition to the right. The multiplication pr takes the two left segments, ps and qr are a mix of left and right segments, and qs are both right segments. Thus, since $n = m$:

$$T(n) = T(\frac{\lceil n/2 \rceil + \lceil n/2 \rceil}{2}) + T(\frac{\lceil n/2 \rceil + \lfloor n/2 \rfloor}{2}) + T(\frac{\lceil n/2 \rceil + \lfloor n/2 \rfloor}{2}) + T(\frac{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor}{2}) + f(n)$$

The floors and ceilings cancel:

$$T(n) = 4 * T(\frac{\lceil n/2 \rceil + \lfloor n/2 \rfloor}{2}) + O(1) = 4 * T(n/2) + O(1)$$

$T(n) = 4T(\frac{n}{2}) + O(1)$. Using the master method, $a = 4$ and $b = 2$. $f(n) = O(n^{\log_2 4 - \epsilon})$, for some $\epsilon > 0$, so the cost is $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

2.2 A better attempt

- If $m = n$, what is the recurrence relation and cost of the algorithm above?

$$T(n) = T(\frac{\lceil n/2 \rceil + \lceil n/2 \rceil}{2}) + T(\frac{\lceil n/2 \rceil + \lfloor n/2 \rfloor}{2}) + T(\frac{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor}{2}) + O(1)$$

The floors and ceilings cancel:

$$T(n) = 3 * T(n/2) + O(1)$$

In the better attempt, there are three sub-problems on each level, instead of 4, and our $f(n) = O(1)$. So the recurrence relation is $T(n) = 3T(\frac{n}{2}) + O(1)$. Using the master method, $a = 3$ and $b = 2$. $O(n) = O(n^{\log_2 3 - \epsilon})$, for some $\epsilon > 0$, so the cost of the algorithm is $\theta(n^{\log_2 3})$. $T(n) \approx \Theta(n^{1.585})$.

2.3 Implementation

3 Problems from CLRS

3.1 Correctness of Bubblesort

- (a) A' is the sorted array of A.
- (b)
  - Invariant:At the start of each iteration of the for loop, the array $A[j - n]$ has the elements originally in $A[n - j]$ and the first element is the smallest one.
  - Initialization: The sub-array only contains the last element $A[n]$.
  - Maintenance: In every step $A[j]$ is compared with $A[j - 1]$ and then $A[j - 1]$ is the smallest. The sub-array length increases by one and the first element is the smallest.

- Termination: The loop terminates when $j = i$.

- (c)
  - Invariant: The sub-array, $A[1 - i - 1]$, consists of the $i - 1$ smallest elements in sorted order.
  - Maintenance: After the execution of the inner loop, $A[i]$ will be the smallest element in the sub-array. The outer loop begins with the sorted elements that are smaller than the elements of $A[i - n]$. After the outer loop, the sub-array will consist of elements that are smaller outer loop array in sorted order.
  - Termination: The loop terminates when $i = A$.

- (d)
  - The ith iteration of the for loop on lines 1-4 will cause $n - 1$ iterations of the for loop on lines 2-4, so the worst case is $\theta(n^2)$ which is the same as insertion sort.

- **Problem 2-2 In the last question of the problem, show why the running times of bubble sort and insertion sort are the ones you gave.**

```
class BubbleSort {
    void bubbleSort(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n; i++)    <— O(n)
            for (int j = n -1; j > i; j--)        <— O(n)
                if (arr[j] < arr[j-1]) {
                    // swap arr[j+1] and arr[i]
                    int temp = arr[j-1];
                    arr[j-1] = arr[j];                    <— 3*O(1)
                    arr[j] = temp;
                }
    }

    /* Prints the array */
    void printArray(int arr[]) {
        int n = arr.length;
        for (int i=0; i<n; ++i){
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Driver method to test above
    Run | Debug
    public static void main(String args[]) {
        BubbleSort ob = new BubbleSort();
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        ob.bubbleSort(arr);
        System.out.println("Sorted array");
        ob.printArray(arr);
    }
}
/* This code is adapted from code contributed by Rajat Mishra on geeksforgeeks.com */
```

The above code for the BubbleSort algorithm, adapted from code found on "geeksforgeeks.com" is an implementation matching the implementation described in the pseudo-code from the chapter problem. The outer for-loop will ultimately run n times, while the inner for loop will run $n - 1 - i$ times for each iteration of the outer loop. Since j will cover all values from the range 0 to n, albeit in descending order, we can represent the loop structure as a summation in the following form.

$$\sum_{i=1}^{n} i = (1 + 2 + 3 + 4 + ... + n)$$

It follows that this summation can be represented as:

$$\frac{n(n+1)}{2}$$

Therefore, the loop structure can be determined to cost $\mathbf{\Theta(n^2)}$.

```java
// Java program for implementation of Insertion Sort
class InsertionSort {
    /*Function to sort array using insertion sort*/
    void sort(int arr[]) {
        int j;
        int n = arr.length;
        for (int i = 1; i < n; i++) {          <— O(n)
            int key = arr[i];          <— O(1)
            for(j = i - 1; j >= 0 && arr[j] > key; j--){          <— O(n)
                arr[j + 1] = arr[j];          <— O(1)
            }
            arr[j+1] = key;          <— O(1)
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n; ++i){
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Driver method
    Run | Debug
    public static void main(String args[]) {
        int arr[] = { 12, 11, 13, 5, 6 };

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
    }
} /* This code is adapted from code contributed by Rajat Mishra on geeksforgeeks.com. */
```

The above code is an implementation of InsertionSort adapted from code found on "geeksforgeeks.com". It is clear that this loop structure follows the same pattern as

Bubblesort's. Therefore, the running time for InsertionSort is $\Theta(n^2)$.