Analysis of Algorithms: Homework 2

Owen Bezick and Morgan Dunnigan

Due date: February 27, 23:55

1 Reading

- We both completed the readings of chapters 6 and 15.

2 Edit Distance

2.1 Optimal Sub-problem Structure

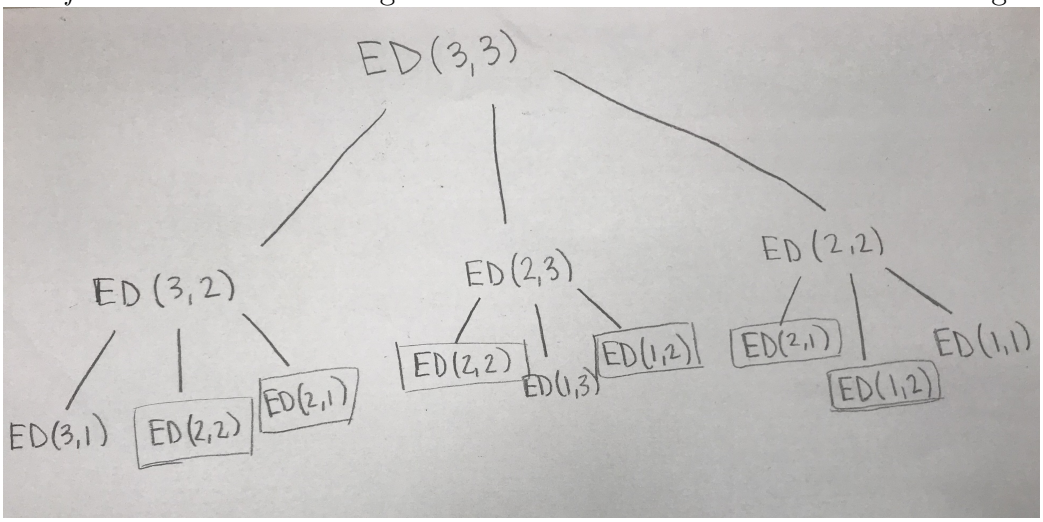
Proof. To illustrate that this problem has optimal sub-structure, we will use proof by contradiction. If the problem P = a * n it is assumed to be optimal where a is the transformation you just completed on the input strings and n is the part of the input strings that remains. This means that a and n must both be optimal. For contradiction, we assume that n is not optimal. If this is the case then there must exist a better P value than the current P. If n is not optimal, then that means that P is not optimal. ↯ □

2.2 Recursive Formulation

Python code for the recursive formulation is in the .zip file

2.3 Overlapping Sub-problems

Proof. Observe the following visual of the recursive calls for the naive algorithm



Note that all the function calls in boxes are calculated more than once. Thus, the recursive algorithm results in repeated sub-problems.

Observe that $T(n) = 2T(n-1) + C$ where 2 is the number of sub-problems and each sub-problem has size $n-1$. Thus, the number of sub-problems goes up by a factor of two for each recursive call in the worst case, so the run time is $\Omega(2^n)$. □

2.4 Dynamic Programming

Python code for the dynamic programming method is in the .zip file

3 Problems from CLRS

- Exercise 1

  Insertion sort works by dividing the input, n, into two parts, a sorted array we will call s, and an unsorted array we will call u. Initially, the sub-array s is the first element of array n. The sub-array u contains the rest of the elements in array n and is $n-1$ elements.

  The outer loop will run $n-1$ times and is a fixed cost. The "worst, worst case" big(O) puts the inner loops as a fixed costs and simply multiplies the size of the two loops; however, this is not the actual worst case, as the inner loop is not a fixed cost. The

inner loop can then be computed as:

$$\sum_{i=0}^{i-1} i + 1$$

Where for each outer loop, the inner loop is the sum of the number of elements the element that is being sorted has to go to get to its sorted position. Thus, insertion sort is $\Theta(n^2)$.

- Exercise 6.4-3
  The running time of heapsort on an array that is already sorted in increasing order is $n \log n$. The running time of heapsort on an array that is already sorted in decreasing order $n \log n$. It costs like this because the tree structure of the heap has height $\log n$ and you iterate over all n elements to sort them.

- Exercise 6.5-9
  Consider k sorted lists where n is the total number of elements in all the lists. We will merge and sort the lists using the following algorithm. Place the smallest element from each list into a min-heap and utilize the min-heapify method to sort the elements. Then add the smallest remaining element from each of the three lists to the heap and once again use min-heapify to sort them. Continue this process until all the elements are in the heap. This algorithm will run in $O(n \log k)$ time since each time you add k elements to the heap, that heapsort will run in $O(k \log k)$ time and we will do this $\frac{n}{k}$ times (note that $\frac{n}{k} * k \log k = n \log k$).