Problem #1 - Exercise 17-2 (page 405)

The palindrome problem is a derivative of the longest common subsequence (LCS) problem; however, in contrast to the standard LCS problem which found the LCS for two independent strings/subsequences, *X* and *Y*, the palindrome problem finds the LCS for two dependent strings/subsequences *X* and X_R (where X_R is the reverse of string *X*). This approach has a slight issue. A given sequence and its reverse may have multiple LCS. For example the sequence:

{ 1, 3, 2, 1, 3 }

and its reverse

{ 3, 1, 2, 3, 1 }

share many longest palindrome subsequences of length three (e.g. 131, 121, 323, 313). However, an additional longest common subsequence for this sequence and its reverse is also 123 (***1***3***2***1***3*** and 3***123***1), but this subsequence is not a palindrome.

I have not been able to come up with a counterexample where the standard LCS algorithm on a string and its reverse did not always return a true palindrome. However, it is possible I missed one; in case a counterexample exists that I have not yet thought of, this problem can be rectified quite easily without changing the algorithm's asymptotic runtime. The procedure is below:

1. Find the longest common subsequence using the standard LCS algorithm in the textbook.

2. If the LCS is already a palindrome, print the result.

3. If the string is not a palindrome and its length, *n*, is even, make a copy of the first half of the returned string (i.e. first $\left\lfloor \frac{n}{2} \right\rfloor$ characters), reverse the string, and then concatenate the reversed string after the first half (i.e. first $\left\lceil \frac{n}{2} \right\rceil$ characters) of the original string. This new string is a longest palindrome subsequence.[1]

4. If the string is not a palindrome and its length, *n*, is odd, make a copy of the first $\left\lfloor \frac{n}{2} \right\rfloor$ characters, reverse the string, and then concatenate the reversed string after the first $\left\lceil \frac{n}{2} \right\rceil$ characters of the original string. This new string is a longest palindrome subsequence.

Below is an example of this modified algorithm using the exception case above (e.g. 123) where the number of characters is odd. The even case follows by extension from this example with the exception that it does not require special handling of the middle, pivot character ("2" in this case).

> **Example:** S = {123}. The first $\left\lfloor \frac{n}{2} \right\rfloor$ characters is "1" while the first $\left\lceil \frac{n}{2} \right\rceil$ characters is "12". Reverse the original string "1" (it is only a single character so no reverse needed) and append it onto "12" forming the palindrome "121". **Note**: this is one of the original longest palindrome subsequences mentioned previously.

[1] Note for an even string of length *n*, $\left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$. This is important when describing the correctness of this approach.

The correctness of this modified approach is guaranteed because of the nature of palindrome and the dependent strings/sequences. The LCS is determined on a string and its reverse. The first $\left\lfloor \frac{n}{2} \right\rfloor$ characters of this LCS are generated from some portion of the string and its reverse. As the LCS algorithm further examines the string and its reverse, these $\left\lfloor \frac{n}{2} \right\rfloor$ characters will appear again in reverse order because the beginning of the reversed string is the end of the original string and vice versa. Therefore, this modified approach is guaranteed to return a palindrome even if the original algorithm did not, which it usually (if not always) will.

Algorithm Runtime Analysis

This algorithm has four distinct steps.

Step #1 – Given a string *X*, find its reverse. This involves iterating through a string from the last character to the first and generating the reverse in that fashion. The running time of this algorithm is $\theta(n)$. Below is a C# implementation of the reverse algorithm.

```
static int[] HW5_Q1_Reverse_String(int[] input_array)
{
    int n = input_array.Length;
    int i;
    int[] output_array = new int[n];

    for(i = 1; i < n; i++){
        output_array[i] = input_array[n - i];
    }
    return output_array;

}
```

Step #2 – Run the standard LCS algorithm on *X* and X_R; this is identical to the algorithm "`LCS-Length`" in the textbook. The running time of this algorithm is:

$$\theta(n \cdot n) = \theta(n^2)$$

Step #3 – Use the modified LCS print algorithm described previously. Run the standard "`Print-LCS`" in the textbook.

$$\theta(n + n) = \theta(n)$$

Step #4 – Check if the string from the "Print-LCS" algorithm is a palindrome. If it is not a palindrome, perform the string reconstruction described earlier.

$$O(n + n) = O(n)$$

Therefore, the total running time of this algorithm is:

$$T(n) = \theta(Step1) + \theta(Step2) + \theta(Step3) + O(Step4)$$

$$T(n) = \theta(n) + \theta(n^2) + \theta(n) + O(n)$$

$$T(n) = \theta(n^2)$$

Problem #2 - Exercise 17-2 (page 473)

Part #A: Describe how to perform a *Search* operation for this data structure.

Array#	Array Data			
A_0	7			
A_1	10	12		
A_2	1	1	3	8

Table 1 – Example Data Structure for Dynamic Binary Search Data Structure with *n*=7

Table 1 is an example of a data structure with 7 elements (i.e. *n*=7) that can be used for dynamic binary search. The worst case for search is when an element is searched for in the set of arrays but is never found. For example, the worst case for the data structure in Table 1 would be to search for any number (e.g. 0) other than {1, 3, 7, 8, 10, 12}. To do this would require performing a binary search on each of the individual arrays (e.g. A_0 to A_2) in the data structure. As such, the worst case total running time is shown below. Note *k* is defined as $k = \lceil \lg(n+1) \rceil$. Define b_i as the value of the i^{th} bit (i.e. 0 or 1) in *n*, where *i* is in the range of 0 to *k*-1.

$$T(n) = \sum_{i=0}^{\lceil \lg(n+1) \rceil - 1} b_i \cdot \lg(|A_i|) + 1 = \sum_{i=0}^{k-1} b_i \cdot \lg(|A_i|) + 1$$

$|A_i|$ is then defined as:

$$|A_i| = 2^i$$

For a given *k*, the worst case is when *n* is:

$$n = 2^k - 1$$

In that case, b_0 to b_{k-1} are all equal to one. As such, the worst case running time is:

$$T(n) = \sum_{i=0}^{\lceil \lg(n+1) \rceil - 1} \left(\lg\left(2^i\right) + 1\right)$$

$$T(n) = \sum_{i=0}^{\lceil \lg(n+1) \rceil - 1} (\mathrm{i} + 1)$$

$$T(n) = \frac{(\lceil \lg(n+1) \rceil - 1)(\lceil \lg(n+1) \rceil)}{2} + \lceil \lg(n+1) \rceil$$

By breaking up the left term, the equation above simplifies to:

$$T(n) = \frac{(\lceil \lg(n+1) \rceil)^2}{2} + \frac{\lceil \lg(n+1) \rceil}{2}$$

For n≥1, $\lceil \lg(n+1) \rceil$ is less than or equal to $\lg(2n)$ so the equations simplify to:

$$T(n) < \frac{(\lg(2n))^2}{2} + \frac{\lg(2n)}{2}$$

By definition of the logarithm function:

$$\lg(2n) = \lg(n) + \lg(2) = \lg(n) + 1$$

$$T(n) < \frac{(\lg(n) + 1)^2}{2} + \frac{\lg(n) + 1}{2}$$

$$T(n) < \frac{\lg^2(n) + 2\lg(n) + 1}{2} + \frac{\lg(n) + 1}{2}$$

$$T(n) < \frac{\lg^2(n) + 3\lg(n) + 2}{2}$$

$$T(n) = O(\lg^2(n)) = O(k^2)$$

Part #B: Describe how to perform the *INSERT* operation. Analyze its worst case running time.

The dynamic binary search data structure trades an increase in the search complexity for a reduction in the insertion complexity. This data structure is very similar to the binary counter example in the book. To perform *n* insertions, the total cost is:

$$T(n) = \sum_{i=0}^{k-1} 2^i \cdot \left\lfloor \frac{n}{2^i} \right\rfloor$$

where *i* corresponds to the i^{th} bit in the binary representation of *n*.

Dynamic binary search differs from the binary counter example because for each i^{th} bit in *n*, the cost to update that array is not $\theta(1)$ as it was for the counter. Instead, it is $\theta(2^i)$ since all 2^i elements in that array will need to be updated. The equation above simplifies to:

$$T(n) < \sum_{i=0}^{k-1} 2^i \cdot \frac{n}{2^i}$$

$$= \sum_{i=0}^{k-1} n$$

$$= kn$$

k was defined as $k = \lceil \lg(n+1) \rceil$ in part A so the above equation can be written as:

$$T(n) < n \cdot \lceil \lg(n+1) \rceil$$

In part A of this question, I explained:

$$\lceil \lg(n+1) \rceil \leq \lg(2n) \mid n \geq 1$$

As such, the total execution time for *n* insertions is:

$$T(n) < n \cdot \lg(2n)$$

$$T(n) < n \cdot (\lg(n) + 1)$$

$$T(n) < n \cdot \lg(n) + n$$

$$T(n) = O(n \cdot \lg(n))$$

The amortized cost per operation is:

$$\frac{O(n \cdot \lg(n))}{n} = O(\lg(n))$$

The worst case running time is $O(n)$. This occurs when *n* is equal to a power of two and all lower order subarrays must be copied into a new array $k = \lg(n)$.

Part C: There are two possible implementations of *DELETE*. The first searches for an element with a particular value (e.g. *d*) then if it exists, it deletes it. The second deletes the j^{th} (where *j* is between 1 and *n*) ordered element in the entire data set. Since the question did not specify which type of *DELETE* to discuss, I will discuss both.

Delete Implementation #1: Unlike the *SEARCH* and *INSERT* operations, this *DELETE* operation requires up to two steps. First, the *SEARCH* operation must be done to find the element of value "*d*" that will be deleted. As shown in part A, this step has running time of $O(\lg^2(n))$. If no element has value *d*, then the *DELETE* operation terminates. If element d is found, then it is found in one of the subarrays A_i (where $0 \le i \le k-1$).

All arrays A_m where $0 \le m \le i$ then need to be reconstructed since the number of elements is now *n*-1. For a given array non-empty A_m, find the first non-empty array A_p (where $0 \le p < m$). If this array exists, then take an element from that array and insert it into its sorted location A_m. If no such array A_p exists, then populate arrays 0 to m-1 with the remaining values in A_m.

Delete Implementation #2: *DELETE* may also need to delete an element based off its sorted precedence. For example, if given 100 elements in the array, delete the i^{th} element (where $1 \le i \le 100$). The simplest implementation would be to copy the dynamic binary search data into a new array, run a selection algorithm on it (e.g. quickselect), then use the deletion implementation #1 described above. The worst case running time of this is $O(n^2)$.

Another option would be to copy all the data from the dynamic binary search data structure into an array then sort it using merge sort, which is $\theta(n \cdot \lg(n))$. Once the element to be deleted as been selected, the dynamic binary search data structure can be rebuilt in two ways:

1. Clear the original data structure and use INSERT operation described in section B, which has an amortized cost of in $O(\lg(n))$ for each element making the cost for *n*-1 elements be $O(n \cdot \lg(n))$.

2. Since the number of elements is known and fixed (i.e. *n*-1), a custom function could be written to use the sorted array to rebuild the dynamic binary search data structure in $O(n)$ time. Note, this approach would not use the INSERT functionality and would build the data structure by dividing the array in sections whose length is based off the binary value of *n*-1.