Homework 8

1. Modify the *lcs*() algorithm to return a longest common subsequence of the two input strings (not just the length).

A1:

```
void lcsPrint (string s, string t)

{

        Int n = s.size(), m = t.size();

        Int S[n + 1][m + 1];


        For (int j = 0; j <= m; ++j)

        {

                S[0][j] = 0;

        }

        For (int i = 0; i <= n; ++i)

        {

                S[i][0] = 0;

        }

        For (int i = 1; i <= n; ++i)

        {

                For (int j = 1; j <= m; ++j)

                {

                        If (s[i - 1] == t[j - 1])

                        {

                                S[i][j] = 1 + S[i - 1][j - 1];

                        } else
```

```
                {
                        S[i][j] = max(S[i - 1][j], S[i][j - 1]);
                }
        }
}
//s [a b c b d a b],
//T [b d c a b a]
//Create auxiliary array, representing a subsequence of s, length S[n][m]
Char substringS[S[n][m] + 1];
//Add terminating character to the end
substringS[S[n][m]] = '\0';
//Index variable to know where we are when adding characters, starts at last slot
Int subIndex = S[n][m];
//Initialize string to aa..aa
For (int i = 0; i <= S[n][m] - 1; ++i)
{
        substringS[i] = a;
}
//Now that we have the max length, we know how much to put into the answer
//Create index variables corresponding to s and t positions
Int sIndex = n;
Int tIndex = m;
//We will use the 2D array and traverse backwards from the bottom right corner going
top left
while (sIndex > 0 && tIndex > 0)
```

```
    {
    //If the current character in s and t are same, then add it to the answer
    if (s[sIndex - 1] == t[tIndex - 1])
    {
        substringS[subIndex - 1] = s[sIndex - 1];    // Put current character in result
        sIndex -= 1;  // Decrement the 2D array cubbies
        tIndex -= 1;
        subIndex -= 1; // Go back a character in the answer array
    } else if (S[sIndex - 1][tIndex] > S[sIndex][tIndex - 1])      // Otherwise, choose one of
the characters
    {
        sIndex -= 1;
    }
    else
    {
        tIndex -= 1;
    }
    //For (int i = 1; i <= n; ++i)
    //{
        //If we are ever back here, then the previous answer didn't work, reset answer arr
        //?
        //For (int j = 1; j <= m; ++j)
        //{
            //This is the case where the character matches on both sides
            //If (s[i - 1] == t[j - 1])
```

```
                //{
                        //Add the character to the answer array, as long as we have space
to add characters
                        //If (subIndex <= substringS.size() - 1)
                        //{
                                //substringS[subIndex]= s[i - 1];
                        //}
                        //subIndex += 1;
                //}
        //}
//}
//For this case, I will just print the longest subsequence
Cout << “The longest subsequence is: “;
For (int i = 0; i <= substringS.size() - 1; ++i)
{
        Cout << subStringS << “ “;
}
Cout << endl;
return;
//Comparison between substring of S, substringS, and portions of string t
//For (int i = 0; s =- 0; ++i)
//{
        //For (int i = 0; i <= S[n][m] - 1; ++i)
        //{
                //substringS[i] = a;
```

```
        //}
    //}
    //Return S[n][m];
}
```

2. Design and analyze a **dynamic-programming** algorithm for the rod-cutting problem (Exercise 8.1.6, page 291).

   6. Rod-cutting problem Design a dynamic programming algorithm for the following problem. Find the maximum total sale price that can be obtained by cutting a rod of n units long into integer-length pieces if the sale price of a piece i units long is pi for i = 1, 2, . . . , n. What are the time and space efficiencies of your algorithm?

A2: Assume that you are given the length n of the original rod, and array of prices at each length i = 1 → n

```
//Individual sale prices are integers, so total is integer
Int maxPrice(int n, arr[])
{
    //Create an array signifying the total prices depending on how many pieces you cut
    int maxArray[n + 1];
    maxArray[0] = 0;        // Initialize zero pieces as having no answer, 0
    //Check for all pieces of i divisions
    for (int i = 1; i <= n; ++i)
    {
        int tempMax = INT_MIN;        // Temporary max variable, initialized to be beaten
        //Compare current max with a max obtained with i pieces
        for (int j = 0; j <= i - 1; ++j)
        {
```

```
                    //Use the subsolution of the previous instance, current piece j with last
        subsolution
                    tempMax = max(tempMax, arr[j] + maxArray[i - j - 1]);        // max function
                }
                maxArray[i] = tempMax;
        }
        //Return the last cubby of the answer array, built off of previous cubbies
        return maxArray[n];

}
```

Analysis: I will analyze the number of addition operations

$M(n) = 1 + \sum_{i=1}^{n} ((1) + \sum_{j=0}^{i-1} (2)) + \Theta(n)$

$= 1 + \sum_{i=1}^{n} (1) + \sum_{i=1}^{n} \sum_{j=0}^{i-1} (2) + \Theta(n)$

$= 1 + (n - 1 + 1) + \sum_{i=1}^{n} (2)(i - 1 - 0 + 1) + \Theta(n)$

$= 1 + n + 2 (\sum_{i=1}^{n} (i)) + \Theta(n)$

$= 1 + n + 2 (n(n + 1)) / 2 + \Theta(n) = 1 + n + n(n + 1) + \Theta(n) = n^2 + 2n + 1$

$M(n) \in \Theta(n^2)$

Space efficiency: $\Theta(n)$

3. Design and analyze a **transform-conquer** algorithm for the following problem:
    - input: an (unsorted) array of integers *A[1..n];*
    - output: a longest **nondecreasing** subsequence of A.

A3: Assuming that nondecreasing means increasing

```
int* incSubsequence (int A[])

{
```

```
	//May be unnecessary, Θ(n lg n)

	mergeSort(A);

	//Create longest subsequence array, first will be decreasing

	Int B[A.size()];

	//Get the solution from a function that gives a longest decreasing subsequence of A,
assuming that decSubsequence() returns an array

	B = decSubsequence(A);	// Actually a pointer operation, C(n)

	//The solution is a longest subsequence, though it is decreasing, should be same
solution the other way

	//Make a third array, to store everything in reverse order, signifying nondecreasing
instead of decreasing

	int C[B.size()];

	Int j = B.size() - 1;	// Reverse index of i

	For (int i = 0; i <= B.size() - 1; ++i)

	{

		C[j] = B[i];

		J -= 1;

	}

	//return this modified array from the decreasingSubsequence variant

	Return C;

}
```

Analysis: I will be analyzing the number of addition operations

$M(n) = \Theta(n \lg n) + C(n) + {}_{i=0}\Sigma^{n-1} (1)$

$= \Theta(n \lg n) + C(n) + (n - 1 - 0 + 1)$

$= \Theta(n \lg n) + C(n) + n$

→ Could be $\in \Theta(n \lg n)$, depending on $C(n)$

4. (Practice problems; do not turn in)
    - Exercise 8.1.4 (page 290).
    - Exercise 8.1.9 (page 292).
    - Exercise 8.1.10.a (page 292).