Assignment 1 is due March 20 (Sunday), 23:30.

Submission A pdf copy of your own solutions to Problems 1 and 2 should be submitted at SUCourse+.

Grading Full credit will be given to correct solutions that are described clearly.

Problem 1 (Recurrences (10 points)) Give an asymptotic tight bound for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. No explanation is needed.

(a) $T(n) = 2T(n/2) + n^3$

(b) $T(n) = 7T(n/2) + n^2$

(c) $T(n) = 2T(n/4) + \sqrt{n}$

(d) $T(n) = T(n-1) + n$

Problem 2 (Longest Common Subsequence - Python) Consider the two algorithms for the Longest Common Subsequence problem, shown in Figures 1 and 2. Each algorithm takes two strings, `X` and `Y`, and two natural numbers, `i` and `j`, and returns the length of the longest common subsequence (LCS) between the prefixes `X[0..i]` and `Y[0..j]` of the given strings. The algorithm shown in Figure 1 is a naive recursive algorithm whereas the algorithm shown in Figure 2 is a slight variation with memoization. We can compute the length of the LCS of two given strings, using these two algorithms, as illustrated in Figure 3.

In the following, m is the length of the string `X`, and n is the length of the string `Y`.

(a) **(20 points)** According to the cost model of Python, the cost of computing the length of a string using the function `len` is $O(1)$, and the cost of finding the maximum of a list of k numbers using the function `max` is $O(k)$. Based on this cost model:

(i) What is the best asymptotic worst-case running time of the naive recursive algorithm shown in Figure 1? Please explain.

(ii) What is the best asymptotic worst-case running time of the recursive algorithm with memoization, shown in Figure 2? Please explain.

```
def lcs(X,Y,i,j):
    if (i == 0 or j == 0):
       return 0
    elif X[i-1] == Y[j-1]:
       return 1 + lcs(X,Y,i-1,j-1)
    else:
       return max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))
```

Figure 1: A recursive algorithm to compute the LCS of two strings

```
def lcs(X,Y,i,j):
    if c[i][j] >= 0:
       return c[i][j]

    if (i == 0 or j == 0):
       c[i][j] = 0
    elif X[i-1] == Y[j-1]:
       c[i][j] = 1 + lcs(X,Y,i-1,j-1)
    else:
       c[i][j] = max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))
    return c[i][j]
```

Figure 2: A recursive algorithm to compute the LCS of two strings, with memoization

```
X = "acggacgggatctgggtccg"
Y = "tcccacatggtgcttccccg"

lX = len(X)
lY = len(Y)

#uncomment the next line to initialize c (for memoization)
#c = [[-1 for k in range(lY+1)] for l in range(lX+1)]

print "Length of LCS is ", lcs(X,Y,lX,lY)
```

Figure 3: An example: Computing the length of the LCS of two given DNA sequences

(b) **(30 points)** Implement these two algorithms using Python. For each algorithm, determine its scalability experimentally by running it with different lengths of strings, in the worst case.

(i) Fill in following table with the running times in seconds.

Algorithm	$m = n = 5$	$m = n = 10$	$m = n = 15$	$m = n = 20$	$m = n = 25$
Naive					
Memoziation					

Specify the properties of your machine (e.g., CPU, RAM, OS) where you run your programs.

(ii) Plot these experimental results in a graph.

(iii) Discuss the scalability of the algorithms with respect to these experimental results. Do the experimental results confirm the theoretical results you found in (a)?

(c) **(40 points)** For each algorithm, determine its average running time experimentally by running it with randomly generated DNA sequences of length $m = n$. For each length $5, 10, 15, 20, 25$, you can randomly generate 30 pairs of DNA sequences, using Sequence Manipulation Suite.[1]

(i) Fill in following table with the average running times in seconds (μ), and the standard deviation (σ).

Algorithm	$m = n = 5$		$m = n = 10$		$m = n = 15$		$m = n = 20$		$m = n = 25$	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive										
Memoization										

(ii) Plot these experimental results in a graph.

(iii) Discuss how the average running times observed in your experiments grow, compared to the worst case running times observed in (b).

[1] `https://www.bioinformatics.org/sms2/random_dna.html`.