(Sol) Midterm 1 practice questions

1. (True/False) For each statement below, write T if the statement is true and F otherwise.
   a. Dynamic programming is only useful when there is a repeated substructure. T
   b. If one greedy algorithm fails to give an optimal solution for a given problem, then all greedy algorithm would fail to do so. F
   c. If $f(n) = f(n-1) + 3$, then $f(n) \in O(n^2)$. T
   d. If $f(n) = n^2/2$, then $f(n) \in O(2^n)$. T
2. (Divide and Conquer) Consider an array A of n distinct integers (assume the index starts from 1). We call (i, j) with $A[i] < A[j]$ a **nice pair** ($1 \leq i < j \leq n$). Example: A = [10, 5, 20] with n = 3. We have two nice pairs: (1, 3) and (2, 3). Describe how we can calculate the number of nice pairs given A, in $O(n \lg n)$ time. For your reference, counting inversion algorithm sketch is given below.

SORT-AND-COUNT(L)

IF (list L has one element)
  RETURN $(0, L)$.

Divide the list into two halves A and B.
$(r_A,\ A) \leftarrow$ SORT-AND-COUNT(A).
$(r_B,\ B) \leftarrow$ SORT-AND-COUNT(B).
$(r_{AB},\ L) \leftarrow$ MERGE-AND-COUNT(A, B).

RETURN $(r_A + r_B + r_{AB},\ L)$.

Answer 1: Every pair is either nice or inverted, so the number of nice pairs is the number of total pairs minus the number of inversion pairs. Use counting inversion

algorithm from the textbook, and then subtract the number of inversions from $n(n-1)/2$. Counting inversion algorithm takes $O(n \lg n)$, and subtraction takes $O(1)$, so overall it takes $O(n \lg n)$.

Answer 2: Modify counting inversion algorithm to count the nice pairs instead of the inversions. We only need to flip the inequality equations from the algorithm. Counting inversion algorithm takes $O(n \lg n)$, and the modified version doesn't have more steps than counting inversion algorithm, so it takes $O(n \lg n)$.

3. (Greedy) Consider a variation of the interval scheduling problem below.
Input: n jobs with deadlines. It is ok to schedule a job at any time as long as it finishes before its own deadline.
Output: the schedule with the maximum number of jobs.
Example: Given 4 jobs $(1,3,5),(2,5,10),(3,2,10),(4,1,3)$ in (index, duration, deadline) format, $\{1,2,3\}$ is a valid schedule as the job 1 runs from time 0 to 3, and job 2 runs from time 3 to 8, and job 3 runs from time 8 to 10. In fact this is one of the optimal solutions scheduling 3 jobs.

Give an algorithm that finds a schedule with the maximum number of jobs. (A clear English description is enough.) For your reference, interval scheduling algorithm sketch is given below.

```
EARLIEST-FINISH-TIME-FIRST (n, s1, s2, …, sn , f1, f2, …, fn)

SORT jobs by finish times and renumber so that f1 ≤ f2 ≤ … ≤ fn.
S ← ∅.   ← set of jobs selected
FOR j = 1 TO n
    IF (job j is compatible with S)
        S ← S ∪ { j }.
RETURN S.
```

Answer: Modify the earliest-finish-time-first algorithm to use the deadline, instead of the finish time. Start time is computed as (deadline - duration). When sorting, if two jobs' deadlines are the same, use the start time to sort. The example 4 jobs would be sorted to $(1,1,3),(2,3,5),(3,5,10),(4,2,10)$.

```
//di is the duration of job i, fi is the deadline of job i
Earliest-Finish-Time-First(n, d1, d2, ..., dn, f1, f2, ..., fn) {
    sort jobs by deadlines and renumber so that f1 <= f2 <= ... <= fn
    S <- {}
    fs = 0;
    for j = 1 to n {
        if (fs+dj <= fj) {
            S <- S U {j}
            fs = fs+dj
        }
    }
    return S
}
```

4. (Dynamic programming) The longest common subsequence (LCS) between " best" and "pet" is "et." Input strings are $X =< x_1, x_2, \cdots, x_m >$ and $Y =< y_1, y_2, \cdots, y_n >$, i.e. $X =< b, e, s, t >$ and $Y =< p, e, t >$. Answer the following subquestions to find the dynamic programming solution of the LCS problem.

    a. `LCS(X,Y)` returns the length of the longest common subsequence between X and Y. Fill in the blank in the code below.

```
 1: LCS(X, Y) {
 2:  //base case
 3:  if (X.length == 1) {
 4:    if x_1 appears in Y {
 5:     return 1;
 6:    }
 7:    else {
 8:      return 0;
 9:    }
10:  }
11:  if (Y.length == 1) {
12:    if y_1 appears in X {
13:      return 1;
14:    }
15:    else {
16:      return 0;
17:    }
18:  }
19:  //recursive case
20:  X' = <x_1, x_2, ... , x_{m-1}>
21:  Y' = <y_1, y_2, ... , y_{n-1}>
22:  if (x_m == y_n) {
```

```
23:    return LCS(X',Y')+1;
24:  else {
25:    return max(LCS(X, Y'), LCS(X', Y));
26:  }
27:}
```

b. Let a 2D array M of size m by n. $M[i,j]$ is the length of the LCS between $< x_1, x_2, \cdots, x_i >$ and $< y_1, y_2, \cdots, y_j >$ and Y. Assume that M is properly initialized with base cases. Modify the code above to use the memoization. You only need to re-write the modified with line numbers. If you wish to add a line, specify where the new line goes using the line numbers. e.g. between line 7 and line 8.

```
//rewrite line 23
if (M[m-1][n-1]<0) {
  M[m-1][n-1] = LCS(X', Y');
}
return M[m-1][n-1]+1;
```

```
//rewrite line 25 to
if (M[m-1][n]<0) {
  M[m-1][n] = LCS(X', Y);
}
if (M[m][n-1]<0) {
  M[m][n-1] = LCS(X, Y');
}
return max(M[m-1][n], M[m][n-1]);
```

c. What is the time complexity of this algorithm with the memoization? A clear English explanation is enough.

Answer: Each element in $m \times n$ OPT array is filled only once, and each element needs at most one comparison and one assignment to fill. So each element takes $O(1)$, and there are $(m+1)(n+1)$ elements, so the time complexity is $O(mn)$.

5. (Extra credit) Consider the following algorithm for Kendall's Tau distance: Given A and B, we first create a 2-D array C such that C[i][0] := A[i] and C[i][1] := B[i]. Then, we sort C by the first column, and count the number of inversions in C[·][1].

For instance, suppose A = [1,5,2] and B = [1,2,5]. According to the statement, C = [[1,1],[5,2],[2,5]] before sorting. C = [[1, 1], [2, 5], [5, 2]] would be after sorting. The number of inversions in C[·][1] (= [1, 5, 2]) would be 1. Indeed, the number of disagreeing pairs, Kendall's Tau between A and B is 1.

This algorithm's main idea is "pairing up" A and B, and then sorting A to "normalize" the order within A so that we can utilize our algorithm for the counting inversions problem.

Find a small counterexample ($n \leq 5$) to show that this algorithm is incorrect.

Answer: consider A = [1, 3, 2, 4] & B = [2, 4, 1, 3]. Kendall's Tau distance between A and B is 4. 2D array C looks like below, and after sorting by the first column, and the number of inversions in the second column is only 2, (2,1), (4,3).