Homework 2

This homework has 7 questions, for a total of 100 points and 0 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up.*

(0 pts) 0. **Before you start; before you submit.**

(a) Carefully review Sections 1.2-1.3 (Induction for Reasoning about Algorithms) of Handout 1 before starting this assignment, and apply it to the solutions you submit.

(b) If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

(10 pts) 1. **Self assessment.**

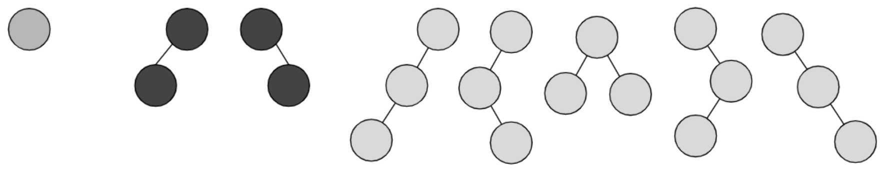
Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

Solution:

2. **Binary Tree Configuration Count.**

Let $T(n)$ be the number of (rooted) binary tree configurations with exactly n nodes. Examples of the first few values of T are shown below – two trees are distinct if a single node changes position, with right and left children considered distinguishable.



$$T(0) = 1, T(1) = 1, T(2) = 2, T(3) = 5$$

$T(n)$ can be described by the following recurrence:

$$T(n) = T(0)T(n-1) + T(1)T(n-2) + ... + T(n-1)T(0).$$

(5 pts) (a) Give a combinatorial argument for why this recurrence correctly counts the number of trees with n nodes.

(5 pts) (b) Write down pseudocode to compute $T(n)$, using dynamic programming. Your implementation should have runtime $O(n^2)$; briefly justify why this is true.

(5 pts) (c) Now consider a variation on this problem: write down a recurrence to count the number of *non-empty* binary trees where every node has 0 or 2 children. Briefly explain why your recurrence is correct.

Solution:

(15 pts) 3. **Counting chicken McNuggets.**

We have a collection M of chicken McNuggets meals; these meals are displayed to you in a menu, represented as an array $M[1..n]$, with the number of McNuggets per meal. Your goal is to determine, for a given positive integer t, whether it is possible to consume *exactly* t McNuggets *using at most one instance of each meal*[1]. For example, for $M = [1, 2, 5, 5]$ and $t = 8$, it is possible with $M[1] + M[2] + M[3] = 8$; however, for the same M and $t = 4$, it is not possible.

Give a recurrence relation (including base cases), that is suitable for a dynamic programming solution to solve this problem in $O(nT)$ time, where $T = \sum_{i=1}^{n} M[i]$ is the total number of available McNuggets. Your solution should include an explanation of why the recurrence is correct. Finally, briefly comment on whether a bottom-up implementation of the recurrence is an "efficient" algorithm, in the sense of how we define "efficiency" in this class (i.e. polynomial with respect to the input size).

Hint: A bottom-up implementation would use a table of roughly $n \times T$ (depending on your base cases) *boolean* values.

Solution:

(20 pts) 4. **Edit distance.**

Imagine that you are building a spellchecker for a word processor. When the spellchecker encounters an unknown word, you want it to suggest a word in its dictionary that the user might have meant (perhaps they made a typo). One way to generate this suggestion is to measure how "close" the typed word A is to a particular word B from the dictionary, and suggest the closest of all dictionary words. There are many ways to measure closeness; in this problem we will consider a measure known as the *edit distance*, denoted $\text{EDIT-DIST}(A, B)$.

In more detail, given strings A and B, consider transforming A into B from start to end, via character *insertions* (i), *deletions* (d), and *substitutions* (s). For example, if $A =$ `ALGORITHM` and $B =$ `ALTRUISTIC`, then one way of transforming A into B is via the following operations:

[1]Somewhat related video: `https://www.youtube.com/watch?v=vNTSugyS038`. Note that this problem is different from the problem in the video.

A	L	G	O	R		I		T	H	M
A	L	T		R	U	I	S	T	I	C
		s	d		i		i		s	s

EDIT-DIST(A, B) **is the minimum cost of transforming string** A **to string** B, given the following three numbers:

- c_i, the cost to insert a character;
- c_d, the cost to delete a character;
- c_s, the cost to substitute a character.

Devise and analyze an efficient dynamic programming algorithm that, on input these three costs and two strings A, B, computes EDIT-DIST(A, B). Be sure to include a recurrence relation for edit distance and justify its correctness, and analyze the running time of your algorithm. You are not required to give explicit pseudocode (though you may if you wish), but at least describe the order in which the table should be filled.

Hint: the recurrence relation for LCS is a good place to look for inspiration.

Solution:

5. **Scheduling classes.**

Consider the following scheduling problem: for a certain lecture room, we are given the start and end times of a set of classes that could be assigned to the room. We wish to create a schedule that *maximizes the number of classes assigned to the room*, so that *none of those classes "overlap" in time.* (The remaining classes will be assigned to other rooms.)

Note that classes whose times intersect only at their boundaries (start/finish times) do *not* overlap, and that there may be more than one optimal schedule.

A bold 376 classmate claims to have devised a greedy algorithm that always produces an optimal schedule, which is shown below.

```
function SCHEDULE($X$)
    $Y \leftarrow$ empty list
    for each $c$ in $X$, in ascending order by start time (breaking ties arbitrarily) do
        if $c$ does not overlap with any class in $Y$ then
            append $c$ to $Y$
    return $Y$
```

Consider the following set of potential classes for the room:

EECS 376 10:30A–12:00P	EECS 281 11:30A–1:00P	EECS 370 1:30P–3:00P	ASTRO 106 1:00P–2:00P	EARTH 103 2:15P–4:15P
EECS 482 11:00A–5:00P	UC 371 12:00P–1:00P	THTREMUS 285 2:00P–3:15P	CRUMHORN 400 4:00P–4:30P	HISTORY 220 5:00P–6:00P

(5 pts) (a) What schedule will the above algorithm return when given the above list of classes? Is this schedule optimal? Explain why or why not.

Solution:

(5 pts) (b) Provide a set of class times for which the above algorithm returns a *suboptimal* schedule, and give an optimal schedule for comparison.

Solution:

(10 pts) (c) Let's modify the above algorithm so that it instead considers the classes in order by their *finish* times, still in ascending order.

Let $Y = [c_1, c_2, \ldots, c_k]$ denote the output of the modified algorithm, and let $S = [s_1, s_2, \ldots, s_m]$ be some arbitrary *optimal* schedule (in ascending order by time). Note that $k \leq m$ because both Y and S are valid schedules, and S is an optimal one.

Prove by induction that for every $i \leq k$, class c_i finishes *no later than* s_i finishes. In other words, prove that $c_i.f \leq s_i.f$, where the $.f$ suffix denotes the finishing time of a class. (Your proof may also use suffix $.s$ for a class's starting time, and you may assume that every class c has nonzero length, i.e., $c.s < c.f$.)

Finally, use this to prove that $k = m$, i.e., schedule Y is optimal (because it has the same number of classes as an optimal schedule).

Solution:

6. **Coloring vertices in a graph.**

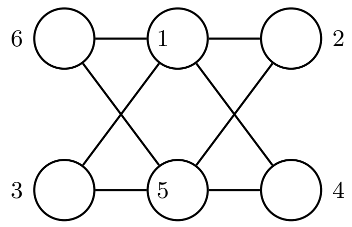
The *graph coloring problem* requires one to color the vertices in a graph so that no two adjacent vertices have the same color. The goal is to minimize the number of colors used.

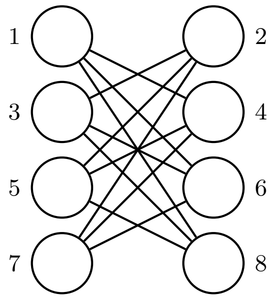
In the *greedy* coloring algorithm below, each vertex is numbered from 1 to $|V|$ and each color is represented by a *positive* integer. The algorithm tries to minimize the number of colors used, k, by greedily coloring each vertex i with the "smallest" color, $c(i)$, that hasn't been used by any of its neighbors, $N(i)$, thus far. Sometimes, this increases the number of colors used. After coloring each vertex, it returns k.

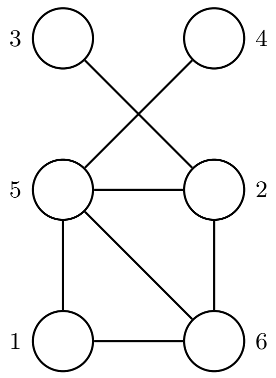
```
function GREEDY-GRAPH-COLOR($G = (V, E)$)
    number the vertices of $G$ from 1 to $|V|$
    initialize $k \leftarrow 0$ and $c(i) \leftarrow 0$ for each $i$
    for $i = 1$ to $|V|$ do
        $c(i) \leftarrow \min\{c \in \mathbb{Z}^+ \mid \forall j \in N(i), c \neq c(j)\}$
        if $c(i) > k$ then
            $k \leftarrow c(i)$
    return $k$
```

(5 pts) (a) For each of the following graphs, run the above greedy coloring algorithm and give 1) the resulting colored graph and 2) k, the number of colors used for that graph.

Note: To change the color in LaTeX, you may change "white" in the code below to the color of your choice. (e.g. blue, green, yellow, ...)

i.


ii.


iii.


Solution:

(5 pts) (b) For each graph in (a), find k^*, the optimal number of colors necessary to color the graph. Use this to state whether the greedy algorithm was optimal or not for each graph.

Solution:

(5 pts) (c) Does the greedy coloring algorithm always use the same number of colors k for a graph, no matter how its vertices are numbered? Justify your answer.

Solution:

(5 pts) (d) The upper bound for k on a particular graph is related to the degrees of that graph's vertices. Let $d(i)$ represent the degree of the i-th vertex of graph G, with vertices numbered $1, 2, \ldots, n$. Prove that, regardless of the ordering of the vertices, k will be at most $\max\{d(1), \ldots, d(n)\} + 1$.

Solution: