EECS 376: Foundations of Computer Science **Due 8:00pm, May 23**
University of Michigan, Spring 2024 **Homework 2** **(accepted until 9:59 pm, no credit after)**

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: None.

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

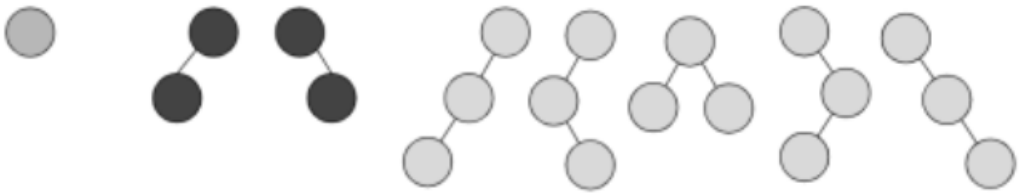
Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this 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: I checked the solution and gladly found that my solution was correct and well-organized for most parts, except in question 2 where I did not consider that $T_Z(n) = O(n^2 \log n)$ which uses the big-O notation which is the asymptotical upper bound while the $T_Y(n) = \Theta(n^2)$ uses the big-Theta notation which is the asymptotically tight bound. So this time I should be more careful in reading the questions and reasoning.

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:

(a)

$$T(n) = \sum_{k=0}^{n-1} T(k)T(n-k-1)$$

$T(n)$ is the number of binary trees with n nodes. There must be a root node which is always in the same position. So we we can view $T(n)$ as the number of ways to arrange for the left and right subtrees of the root by $n-1$ nodes.

If we put k nodes in the left subtree of the root, then there are $n-k-1$ nodes in the right subtree. So the number of ways to arrange the left subtree with k nodes and the right subtree with $n-k-1$ nodes is $T(k)T(n-k-1)$.

All the possible ways to arrange the subtrees are the sum of all such products, from $k=0$ to $k=n-1$. So the recurrence relation is correct.

(b) input: integer n

```
1: function BinaryTreeCount(n)
2:     initialize array T with size n + 1
3:     initialize T(0) ← 1 and T(1) ← 1 and all other T(i) ← 0 for i ∈ {2, 3, ..., n}
4:     for i = 2 to n do
5:         for j = 0 to i − 1 do
6:             T(i) ← T(i) + T(j) * T(i − 1 − j)
7:     return T(n)
```

The time complexity of the above algorithm is $O(n^2)$ because there are two nested loops, the inside loop compute $O(n)$ times and the outside loop perform $O(n)$ times of inside loop.

By bottom-up, we compute the value of $T(2)$ to $T(n)$ in order, while storing the answer of previous subproblems in the array T, acc. to the combinatorial argument. So the output is true.

(c)

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

$$T(n) = \sum_{k=1}^{n-2} T(k)T(n-k-1)$$

Now that a node can only have 0 or 2 children, the root node can have 0 children or 2 children. So the case that we assign 0 node to a subtree is not valid because then we would make the root has only 1 child. So we start from $k = 1$ to $k = n-2$, considering from the case where we assign 1 node to the left subtree to the node and $n-2$ nodes to the right subtree, to the case where we assign $n-2$ nodes to the left subtree and 1 node to the right subtree of the root.

And now our base case starts from $T(1) = 1$ and $T(2) = 0$ because the root node must have 0 or 2 children so there is no way to have such tree with only 2 nodes. And we do not need to consider $T(0)$ because we never need to assign 0 nodes to a subtree.

Then by the same induction logic, the recurrence relation is correct.

(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:
Base case:

$$T[i][0] = 1 \text{ for all } i \in \{0, 1, \ldots, n\}$$

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

$$T[0][j] = 0 \text{ for all } j \in \{1, 2, \ldots, t\}$$

Recurrence:

$$T[i][j] = \begin{cases} T[i-1][j] & \text{if } j < M[i] \\ max(T[i-1][j], T[i-1][j-M[i]]) & \text{otherwise} \end{cases}$$

This recurrence is correct which can be verified by induction.
Base case: When $j = 0$, it is always possible to consume 0 McNuggets by not choosing any meal. So $T[i][0] = 1$ for all $i \in \{0, 1, \ldots, n\}$. When $i = 0$, it is impossible to consume any(nonzero) McNuggets by choosing from 0 meals. So $T[0][j] = 0$ for all $j \in \{1, 2, \ldots, t\}$.

Inductive Step: Assume we have determined whether it is possible to consume 0 to j McNuggets by choosing from the first $i - 1$ meals.

When $j < M[i]$, it is impossible to consume j McNuggets by choosing the i-th meal. So we look at the earlier case: $T[i][j] = T[i-1][j]$.
When $j \geq M[i]$, we have two choices: 1) not choosing the i-th meal, which means $T[i][j] = T[i-1][j]$; 2) choosing the i-th meal, which means $T[i][j] = T[i-1][j-M[i]]$. So we pick the maximum of the two choices to see whether any of the two cases work.
So by induction, the recurrence is correct, i.e. we can always figure out whehther it is possible to consume exactly t McNuggets by the algorithm.

Running time analysis: The initialization of the table takes $O(n + T)$ time. The recurrence takes $O(nT)$ time to fill the table since it is a 2-layer nested loop with each operation O(1) time. So the total time complexity is $O(nT)$, which is polynomial with respect to the input size.
So the bottom-up implementation of the recurrence is efficient.

(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 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:

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:
We use a 2D table $dis[|A|+1][|B|+1]$ to store the edit distance between the substring of A and B. $dis[m][n]$ stores the edit distance between $A[0:m-1]$ and $B[0:n-1]$.
We initialize the array as follows:

$$dis[m][0] = m \cdot c_d \text{ for all } A[0:m-1]$$

$$dis[0][n] = n \cdot c_i \text{ for all } B[0:n-1]$$

Then the recurrence relationship is for all $m \in \{1, 2, \ldots, |A|\}$ and $n \in \{1, 2, \ldots, |B|\}$:

$$dis[m][n] = \min \begin{cases} dis[m-1][n-1] & \text{if } A[m] = B[n] \\ min \begin{cases} dis[m-1][n-1] + c_s \\ dis[m-1][n] + c_d \\ dis[m][n-1] + c_i \end{cases} & \text{if } A[m] \neq B[n] \end{cases}$$

We can justify the correctness of the recurrence relationship by induction.
Base case: When $m = 0$, the edit distance between the substring of A and B is the cost of inserting all n characters in the substring $B[0:n-1]$. When $n = 0$, the edit distance between the substring of A and B is the cost of deleting all m characters in the substring $A[0:m-1]$. These are optimal for sure.

Inductive step: Assume that for all $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$, $dis[m][n]$ is the optimal edit distance between the substring of $A[0:m-1]$ and $B[0:n-1]$.
So there are two cases for $dis[m+1][n+1]$.
Case 1: $A[m+1] = B[n+1]$, then the char need not to be changed. So $dis[m+1][n+1] = dis[m][n]$.
Case 2: $A[m+1] \neq B[n+1]$, then three choices may produce the optimal edit distance for this problem: 1) change the char in A to the char in B, which costs c_s; 2) delete the char in A, which costs c_d; 3) insert the char in B to A, which costs c_i. So we pick

the optimal one: $dis[m+1][n+1] = \min(dis[m][n]+c_s, dis[m+1][n]+c_d, dis[m][n+1]+c_i)$.

Therefore induction shows that the recurrence relationship is correct.

Running time analysis The running time of the algorithm is $O(|A| \cdot |B|)$. The initialization takes $O(|A| + |B|)$ time which is linear.
Since we need to fill the table $dis[|A|+1][|B|+1]$ with $|A| \cdot |B|$ cells in a bottom-up manner, starting from $dis[0][0]$ to $dis[|A|][|B|]$ and every step takes at most four calculations which is O(1), the recurrence takes $O(|A| \cdot |B|)$ time.
Therefore the total running time is $O(|A| \cdot |B|)$.

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.

```
1: function SCHEDULE(X)
2:     Y ← empty list
3:     for each c in X, in ascending order by start time (breaking ties arbitrarily) do
4:         if c does not overlap with any class in Y then
5:             append c to Y
6:     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: {EECS 376, UC 371, ASTRO 106, THTREMUS 285, CRUMHORN 400, HISTORY 220.} This schedule is optimal because the number of classes it produces is 6, the same as what our correct greedy algorithm produces. We already proved a correct greedy algorithm in class(earliest ending time), and by this algorithm, we can

get the same number of classes({EECS 376, UC 371, ASTRO 106, THTREMUS 285, CRUMHORN 400, HISTORY 220}, which is also 6 classes and even the same ones).

(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.

EECS 376 11:30A–1:00P	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

Solution: I modified the start time of EECS 376. Now by the algorithm above, the output will be {EECS 482, HISTORY 220}, which is suboptimal.
By one correct greedy algorithm(earliest ending time), the output will be {EECS 376, ASTRO 106, THTREMUS 285, CRUMHORN 400, HISTORY 220}, which is optimal.

(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:

We prove by induction that for every $i \leq k$, class c_i finishes *no later than* s_i finishes, and therefore we can always exchange s_i by c_i for any $1 \leq i \leq m$ in the optimal solution S without breaking optimal, which indicates that Y is also an optimal solution.

Base case: For $i = 1$, since by the algorithm, c_1 is the class that finishes earliest, so $c_1.f \leq s_1.f$. Therefore s_1 can be exchanged by c_1, keeping the optimal schedule of S.

Inductive step: Assume that for some $i \leq n < m$, $c_i.f \leq s_i.f$ (which means that we can always exchange all s_i by c_i for $1 \leq i \leq n < m$ in the optimal solution S without breaking optimal).
By our assumption, $c_n.f \leq s_n.f$, which indicates that $c_n.f <= s_{n+1}.s$. Therefore the algorithm must have considered s_{n+1}, which means that $s_{n+1}.f \leq c_{n+1}.f$. So we can exchange s_{n+1} by c_{n+1} in the optimal solution S without breaking optimal.

Therefore by induction, we have shown that for every $i \leq m$, $c_i.f \leq s_i.f$, and therefore we can exchange $\{s_1, s_2, ..., s_m\}$ by $\{c_1, c_2, ..., c_m\}$. Since S is optimal, this indicates that $m = k$, and therefore Y is also an optimal solution.

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

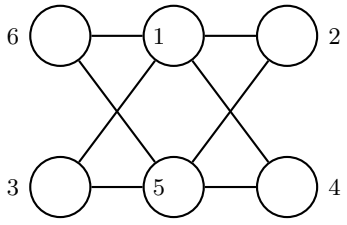
The *graph coloring problem* requires one to color the vertices in a graph so that no two adjacent vertices have the same color. The goal is to minimize the number of colors used.

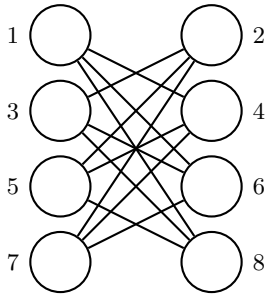
In the *greedy* coloring algorithm below, each vertex is numbered from 1 to $|V|$ and each color is represented by a *positive* integer. The algorithm tries to minimize the number of colors used, k, by greedily coloring each vertex i with the "smallest" color, $c(i)$, that hasn't been used by any of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

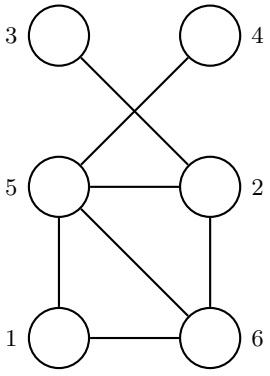
```
1: function GREEDY-GRAPH-COLOR(G = (V, E))
2:     number the vertices of G from 1 to |V|
3:     initialize k ← 0 and c(i) ← 0 for each i
4:     for i = 1 to |V| do
5:         c(i) ← min{c ∈ ℤ⁺ | ∀j ∈ N(i), c ≠ c(j)}
6:         if c(i) > k then
7:             k ← c(i)
8:     return k
```

(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, ...)

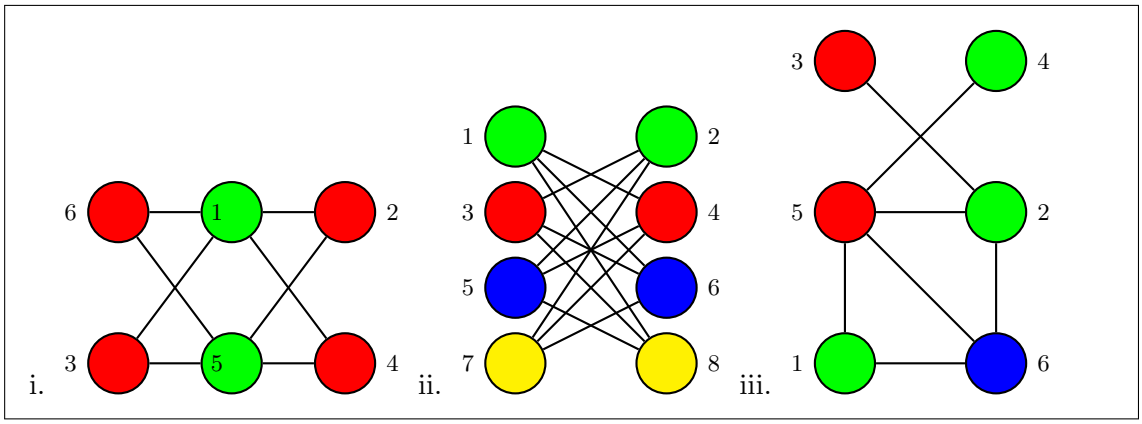
i. 

ii. 

iii. 

Solution: Graphs:

i. k = 2
ii. k = 4
iii. k = 3

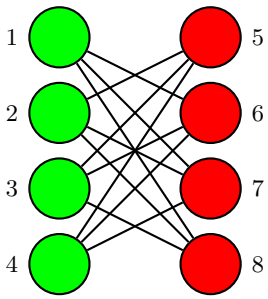


(5 pts) (b) For each graph in (a), find k^*, the optimal number of colors necessary to color the graph. Use this to state whether the greedy algorithm was optimal or not for each graph.

Solution:
i. k* = 2, k = 2. (This is trivial because as long as there are two connected nodes, we must use at least 2 colors.)
ii. k* = 2, k = 4. (The optimal solution is that 1,3,5,7 in one color and 2,4,6,8 in an other color)
iii. k* = 3, k = 3. (6, 1 and 5 form a 3-cycle, which means that we need at least three colors)
So claim: The greedy algorithm is not optimal for each graph.

(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: No. Take ii as a counterexample.



I relabelled the nodes, and it produces a different output of $k = 2$.

(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:
Let N be the node with the largest degree in the graph.
So $d(N) = max\{d(1), d(2), ..., d(n)\}$ Suppose that N and its $d(N)$ neighbors are colored in at most $d(N) + 1$ colors since each node can only be in one color.
Assume for sake of contradiction that there is a Node M such that its color is different from all colors of N and its neighbors.
Then the algorithm must has checked all neighbors of M and used up all colors from 1 to $d(N)$, which means that M has at least $d(N) + 1$ neighbors. So $d(M) > d(N)$. But this contradicts the fact that N has the largest degree. So we have proved the claim by contradiction.