EECS 376: Foundations of Computer Science
University of Michigan, Spring 2024

Solutions to 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.**

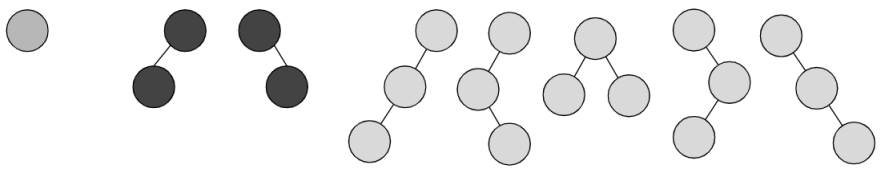
Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this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

(a) Consider the root node in a tree with n nodes. We have n choices for how many nodes l are in the left subtree — any number from 0 to $n-1$. Note that this uniquely determines how many nodes r are in the right subtree, as $r = n-1-l$. So, for any choice of k, there are $T(k)$ configurations possible for the left subtree, and $T(n-1-k)$ configurations possible for the right subtree. Summing over all choices of k, we have

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

Note: This recurrence describes the Catalan numbers.

(b) **Bottom-up**

Algorithm:

- Initialize an array $T[0, 1, \ldots, n]$ and set $T[0] = 1$.
- For $i = 1, \ldots n$,
  - $T[i] = \sum_{k=0}^{i-1} T[k]T[i-k-1]$, which can be computed by a for loop.
- Return $T[n]$.

Runtime Analysis: We iterate over $O(n)$ cells, and each cell takes $O(n)$ time to compute. Thus, the overall runtime is $O(n^2)$.

Top-down

Algorithm:

- Initialize an array $T[0, 1, \ldots, n]$. Set $T[0] = 1$ and $T[i] = \perp$ for $i = 1, \ldots, n$.
- Return Query(T, n),

where the subroutine is defined as follows.

- Query(T, i):
  - If $T[i] \neq \perp$, return $T[i]$.
  - $T[i] = \sum_{k=0}^{i-1} \text{Query}(T, k)\text{Query}(T, i-k-1)$.
  - Return $T[i]$

Runtime Analysis: We have at most $O(n)$ distinct subproblems to calculate, and for each subproblem, it takes $O(n)$ time to iterate over the sum, so altogether the runtime is $O(n^2)$.

(c) Let $T'(n)$ to be the number of rooted binary trees where every node has 0 or 2 children. For the base cases, we have

- $T'(1) = 1$ since we have a single node with no children.
- $T'(2) = 0$ since it is impossible to have a 2-node tree with 0 or 2 children.

Then for $n \geq 3$ we have

$$T'(n) = \sum_{k=1}^{n-2} T'(k)T'(n-k-1),$$

since with 3 or more nodes, each subtree of the root node must be nonempty in order to ensure the root node has 2 children. Then the number of nodes in the left subtree must be at least 1 and at most $n-2$. As before, if there are $T'(k)$ possible left subtrees, there are $T'(n-k-1)$ right subtrees.

Note: We could also eliminate the base case $T'(2)$ and define the recurrence over $n \geq 2$. In this case, $T'(2) = 0$ implicitly since the sum from $k = 1$ to $k = 0$ is empty and therefore 0. This would be an equally valid 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).

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

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

Solution: We'll start by defining our subproblem. If there exists a subset of nuggets that sums to T, then any given meal is either included in the sum or it is not. We define a function $S(i, k)$, which indicates whether elements $M[1, \ldots, i]$ have a subset that sums to k.

The base cases are:

- If $k = 0$, we can always order 0 nuggets by using zero meals so $S(i, k)$ is **true**.
- If $i = 0$ and $k > 0$, we cannot order a positive amount of nuggets with the zero meals available so $S(i, k)$ is **false**.

Then for the recursive case, if we consider the first i meals and total nuggets k, $S(i, k)$ is true if either

- we can make k nuggets by using i meals and $k - A[i]$ of other nuggets.
- we can order k nuggets without using meal i by using only other meals.

As a result, we have:

$$S(i, k) = S(i - 1, k - A[i]) \vee S(i - 1, k)$$

This solution is not efficient, as the size of the memo depends on T, which is a sum of input *values* rather than input *sizes*. Since the memo is not of polynomial size, filling it out cannot be 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: Letting $m = |A|$ and $n = |B|$, we can represent the strings as character arrays: $A = A[1, \ldots, m]$ and $B = B[1, \ldots, n]$.

Let us define $\mathbf{ED}(i, j)$ to be Edit-Dist$(A[1, \ldots, i], B[1, \ldots, j])$ in order to make explicit which subproblems we will solve. That is, the subproblems correspond to the pairs of *prefixes* of A and B (substrings of A and B starting from their first characters). The value we ultimately want to compute is $\mathbf{ED}(m, n)$, because it equals Edit-Dist(A, B).

We give a recurrence relation for **ED**. There are two **base cases**:

- If $i = 0$, then the length-i prefix of A is the empty string, so we must insert j characters into it in order to transform it into the length-j prefix of B. Hence, the edit distance is $\mathbf{ED}(0, j) = c_i \times j$.
- If $j = 0$, then the length-j prefix of B is the empty string, so we must delete all i characters from the length-i prefix of A in order to transform it into B. Hence, the edit distance is $\mathbf{ED}(i, 0) = c_d \times i$.

For the recurrence relation, we need to compare the ith character of A to the jth character of B. If $A[i] = B[j]$, then there are *three* possibilities for how an optimal transform may look. Note that it is not immediately obvious that leaving $A[i]$ unchanged (after optimally transforming the prior characters) is part of an overall optimal strategy, so instead we consider all the possibilities of how an optimal transform of $A[1, \ldots, i]$ to $B[1, \ldots, j]$ may look:

1. (optimally) transform $A[1, \ldots, i]$ to $B[1, \ldots, j-1]$, then *insert* $B[j]$ at the end of A;
2. (optimally) transform $A[1, \ldots, i-1]$ to $B[1, \ldots, j]$, then *delete* $A[i]$;
3. (optimally) transform $A[1, \ldots, i-1]$ to $B[1, \ldots, j-1]$, then leave $A[i]$ unchanged (since it matches $B[j]$). Note that this option is no worse than substituting (which has some cost), so we do not need to consider substitution here.

This yields the recurrence (which holds if $A[i] = B[j]$):

$$\mathbf{ED}(i,j) = \min \begin{cases} c_i \ + \ \mathbf{ED}(i, j-1), \\ c_d \ + \ \mathbf{ED}(i-1, j), \\ \mathbf{ED}(i-1, j-1). \end{cases}$$

(It is possible, but tedious and subtle, to rigorously prove that leaving $A[i]$ unchanged is a best option. This allows the recurrence to be simplified to a single case, but does not change the asymptotic runtime of the final algorithm.)

Now we consider the case where $A[i] \neq B[j]$. Observe that when (optimally) transforming $A[1, \ldots, i]$ to $B[1, \ldots, j]$, we *must* perform some operation that changes the final character of A, so there is no option to leave it unchanged as there was above. Instead, an optimal transform must look like one of the following three options (corresponding to the insert, delete, and substitute operations):

1. (optimally) transform $A[1, \ldots, i]$ to $B[1, \ldots, j-1]$, then *insert* $B[j]$ at the end of A;
2. (optimally) transform $A[1, \ldots, i-1]$ to $B[1, \ldots, j]$, then *delete* $A[i]$;
3. (optimally) transform $A[1, \ldots, i-1]$ to $B[1, \ldots, j-1]$, then *substitute* $A[i] \to B[j]$.

A best choice among these three options corresponds to the actual edit distance. So, this yields the recurrence (for the case $A[i] \neq B[j]$):

$$\mathbf{ED}(i,j) = \min \begin{cases} c_i \ + \ \mathbf{ED}(i, j-1), \\ c_d \ + \ \mathbf{ED}(i-1, j), \\ c_s \ + \ \mathbf{ED}(i-1, j-1). \end{cases}$$

We can combine the above two recurrences more succinctly:

$$\mathbf{ED}(i,j) = \min \begin{cases} c_i \ + \ \mathbf{ED}(i, j-1), \\ c_d \ + \ \mathbf{ED}(i-1, j), \\ \begin{cases} \mathbf{ED}(i-1, j-1) & \text{if } A[i] = B[j], \\ c_s \ + \ \mathbf{ED}(i-1, j-1) & \text{if } A[i] \neq B[j]. \end{cases} \end{cases}$$

To compute an answer in a bottom-up fashion using this recurrence, we construct a $(m+1) \times (n+1)$ table $M[0, \ldots, m][0, \ldots, n]$, where $M[i][j]$ will hold the value of $\mathbf{ED}(i,j)$. One possible way to fill the table follows:

1. Fill in $M[i][0]$ and $M[0][j]$ as specified by the base cases.
2. For $i = 1, \ldots, m$
   (a) For $j = 1, \ldots, n$, fill in $M[i][j]$ as specified by the above recurrence.

Observe that filling in the table in this order ensures that for each cell $M[i][j]$, its "dependencies" are calculated and filled in before it. That is, $M[i][j-1], M[i-1][j], M[i-1][j-1]$ are set before $M[i][j]$ is to be set.

(Note that in line (b), it does not matter that the outer loop corresponds to i and the inner loop corresponds to j. We could swap the inner and outer loops, and the algorithm would still be correct.)

After the table M is completely filled, we finally return the entry

$$M[m][n] = \textbf{ED}(A[1,\ldots,m][1,\ldots,n]) = \textbf{ED}(A,B)\,.$$

There are $O(nm)$ entries to fill in the table, and each entry can be computed in $O(1)$ time, so the entire table (and $M[m][n]$ in particular) can be computed in $O(nm)$ time.

We provide a bottom-up pseudocode implementation for reference (though this is not required for full credit).

Input: words A and B for which EDIT-DISTANCE needs to be computed
Output: **EDIT-DISTANCE**(A,B)

```
1:  function EDIT-DISTANCE(A, B)
2:      Initialize an (m + 1) × (n + 1) table M
3:      for i = 0, ..., m do                                   ▷ Base cases
4:          M[i][0] = c_d × i
5:      for j = 0, ..., n do
6:          M[0][j] = c_i × j
7:      for i = 1, ..., m do          ▷ As noted above, lines 7 and 8 can be interchanged
8:          for j = 1, ..., n do
9:              if A[i] = B[j] then
10:                 M[i][j] ← min{M[i − 1][j − 1], M[i][j − 1] + c_i, M[i − 1][j] + c_d}
11:             else
12:                 M[i][j] ← min{M[i − 1][j − 1] + c_s, M[i][j − 1] + c_i, M[i − 1][j] + c_d}
13:     return M[m][n]
```

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: The algorithm yields the following schedule: EECS 376, UC 371, ASTRO 106, THTREMUS 285, CRUMHORN 400, HISTORY 220.

This schedule is optimal, since the (correct) algorithm described in part (c) gives a schedule with the same number of classes.

Alternatively, we can reason directly about the given classes, to show that there is no (legal) schedule with *seven* classes:

1. Observe that HISTORY 220 does not overlap with any other class, so it can safely be included in any schedule (without ruling out anything else from the schedule).
2. Then notice that EECS 482 conflicts with too many other classes, so it cannot be part of any seven-class schedule, so we ignore it from now on.
3. Next, there are only two other classes that go past 4PM: EARTH 103 conflicts with three other classes, so it cannot be part of any seven-class schedule, so we remove it from consideration. Then CRUMHORN 400 does not conflict with any remaining classes under consideration, so we can safely include it.
4. This leaves six classes under consideration, of which we would need to take five more to get a schedule of seven classes.
5. But notice that EECS 376 and 281 conflict with each other, and EECS 370 and ASTRO 160 conflict with each other, so we can include at most one class from each pair in the schedule. This means it is impossible to take five of the remaining six classes.

(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: The key idea is to have a class that starts earliest but lasts a long time, so that it conflicts with some shorter classes that do not conflict with each other. Here is a minimal example of this idea:

EECS 376	EECS 281	EECS 370
9:00A-12:00P	9:30A-11:00P	11:30A-12:30P

The above algorithm would select EECS 376 since it is the first class by start time, but it would be unable to select either of the other two classes since both conflict. An optimal schedule would be EECS 281 and EECS 370, since this is two classes, and we obviously cannot take all three.

It should be clear that we could modify this example so that the algorithm's output is arbitrarily "bad" compared to an optimal schedule, i.e., the output has only one class, whereas an optimal schedule has many.

(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: Base Case $i = 1$. By definition, the first class considered by the algorithm finishes no later than any of the other classes, and it is included in the schedule as c_1 (because it does not conflict with the initial, empty schedule). Therefore, $c_1.f \leq s.f$ for *every* class s, including for $s = s_1$ (whatever it may be).

Inductive Case. The inductive hypothesis is that $c_i.f \leq s_i.f$, for some $1 \leq i < k$. We want to show that $c_{i+1}.f \leq s_{i+1}.f$.

We know that $c_i.f \leq s_i.f$ (this is the inductive hypothesis), that $s_i.f \leq s_{i+1}.s$ (because S is a valid schedule that is sorted in ascending order), and that $s_{i+1}.s < s_{i+1}.f$ (because all classes have nonzero length). Combining all these, we get that

$$c_i.f \leq s_{i+1}.s < s_{i+1}.f.$$

So, s_{i+1} does not overlap with c_i, and the algorithm must have considered class s_{i+1} *after* it considered class c_i. (Notice that this conclusion relies on the *strict* inequality $c_i.f < s_{i+1}.f$. Without this, we would not be able to conclude that the algorithm considered s_{i+1} after c_i, because their finishing times might be equal, and ties can be broken arbitrarily.)

Now observe that by definition of the algorithm, among all classes that it considered after c_i and that do not overlap with c_i, class c_{i+1} is one with *minimal* finishing time.

Since we have shown that s_{i+1} was considered after c_i and does not overlap with c_i, we conclude that $c_{i+1}.f \leq s_{i+1}.f$, as needed.

Finally, we show that $k = m$. We claim that S cannot have any class after s_k, because otherwise the algorithm would have added another class to Y after c_k. Specifically, by the same logic as above, any hypothetical s_{k+1} would have been considered by the algorithm after c_k, and does not overlap with c_k. So, the algorithm would have added it (or some other class considered even before s_{k+1}, and after c_k) to the schedule as c_{k+1}. Since there is no such c_{k+1} in the schedule Y output by the algorithm, no such s_{k+1} can exist. This shows that $k \geq m$, and since $k \leq m$ (because S is optimal), we must have $k = m$, as desired.

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

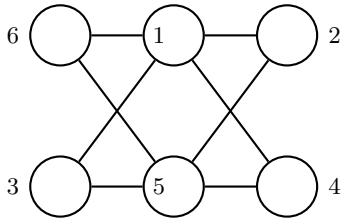
The *graph coloring problem* requires one to color the vertices in a graph so that no two adjacent vertices have the same color. The goal is to minimize the number of colors used.

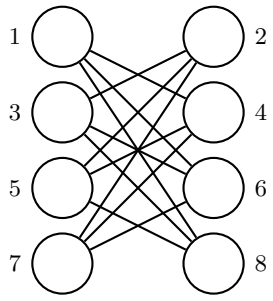
In the *greedy* coloring algorithm below, each vertex is numbered from 1 to $|V|$ and each color is represented by a *positive* integer. The algorithm tries to minimize the number of colors used, k, by greedily coloring each vertex i with the "smallest" color, $c(i)$, that hasn't been used by any of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

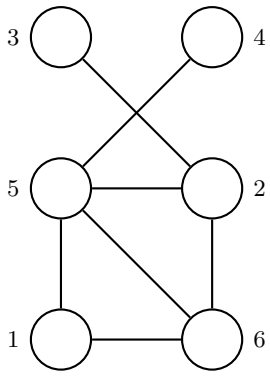
```
1: function GREEDY-GRAPH-COLOR(G = (V, E))
2:     number the vertices of G from 1 to |V|
3:     initialize k ← 0 and c(i) ← 0 for each i
4:     for i = 1 to |V| do
5:         c(i) ← min{c ∈ ℤ⁺ | ∀j ∈ N(i), c ≠ c(j)}
6:         if c(i) > k then
7:             k ← c(i)
8:     return k
```

(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.

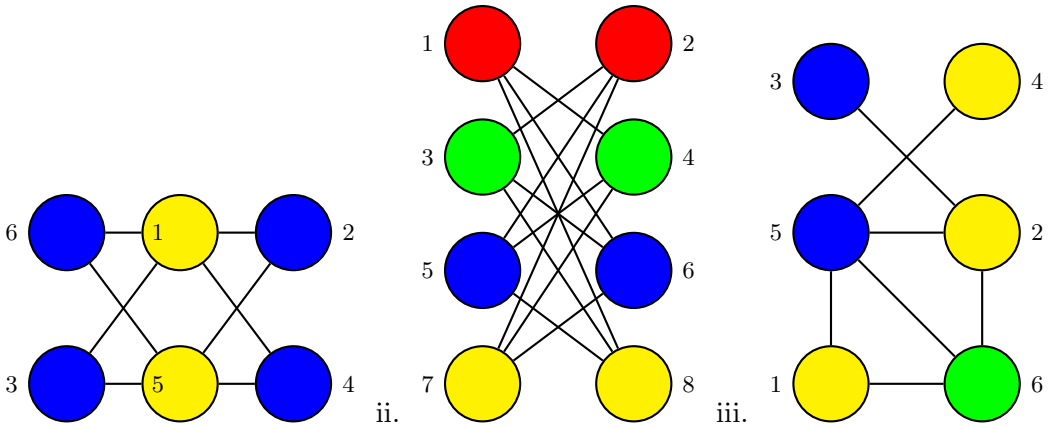
Note: To change the color in LaTeX, you may change "white" in the code below to the color of your choice. (e.g. blue, green, yellow, ...)

i. 

ii. 

iii. 

Solution:



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. $k^* = 2$ using the color scheme the greedy algorithm produced. So the greedy algorithm is optimal for this graph.

ii. $k^* = 2$ if you color the nodes on the left side the same color, and color all the nodes on the right side another color. Therefore, the greedy algorithm is not optimal for this graph.

iii. $k^* = 3$ using the color scheme the greedy algorithm produced. So the greedy algorithm is optimal for this 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. The graph in (ii) can be numbered differently (with 1, 2, 3, 4 on the left, and 5, 6, 7, 8 on the right) and only use 2 colors (k=2) instead of 4 (k=4). Thus, a different ordering of vertices can result in a different number of colors k for a graph.

(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: A new color is only introduced in the following situation: there are k_i used colors so far, and vertex i has $c_i = k_i$ adjacent vertices with unique colors. Note that $c_i \leq d(i)$, so we have $k_i = c_i \leq d(i)$. By the pigeonhole principle, we must introduce

a new color. We then have $k_{i+1} = c_i + 1 \leq d(i) + 1$. In the worst case ordering, the final color will be introduced when $d(i) = \max\{d(1), \ldots, d(n)\}$. Therefore, we have $k \leq \max\{d(1), \ldots, d(n)\} + 1$.