Problems marked with **E** are graded on effort, which means that they are graded subjectively on the perceived effort you put into them, rather than on correctness. For bonus questions, we will not provide any insight during office hours or 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 names. You must write your own solution and may not look at any other student's write-up.

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

Solution:

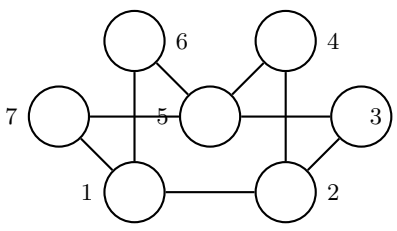
1. The *graph coloring problem* is the task of determining whether it is possible to color all the vertices in a graph such that no adjacent vertices have the same color, while using the minimum number of distinct colors.

   A *greedy coloring algorithm* is an algorithm for coloring a graph that iterates linearly through the graph's vertices in numeric order and assigns them the first available color. A color is considered available if no adjacent vertices have that color. The below pseudocode describes such an algorithm; in it, k represents the number of colors used, adj(i) gives a list of all vertices adjacent to vertex i, c(i) gives the color of the vertex i represented as a positive integer.

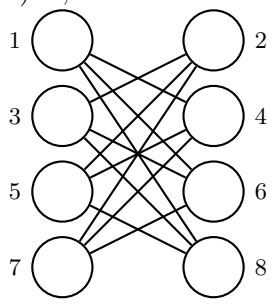
```
1: function GREEDY-GRAPH-COLOR(G = (V, E))
2:     Assign a number i, 1 ≤ i ≤ |V|, to each vertex of G
3:     k ← 0
4:     for i = 1 to |V| do
5:         let c(i) be the smallest positive integer such that c(i) ∉ {c(j) | j < i, j ∈ adj(i)}
6:         if c(i) > k then
7:             k ← c(i)
8:     return k
```

   (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.

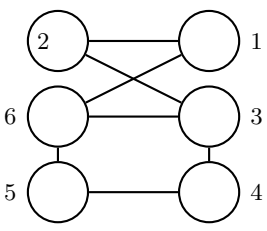
i.



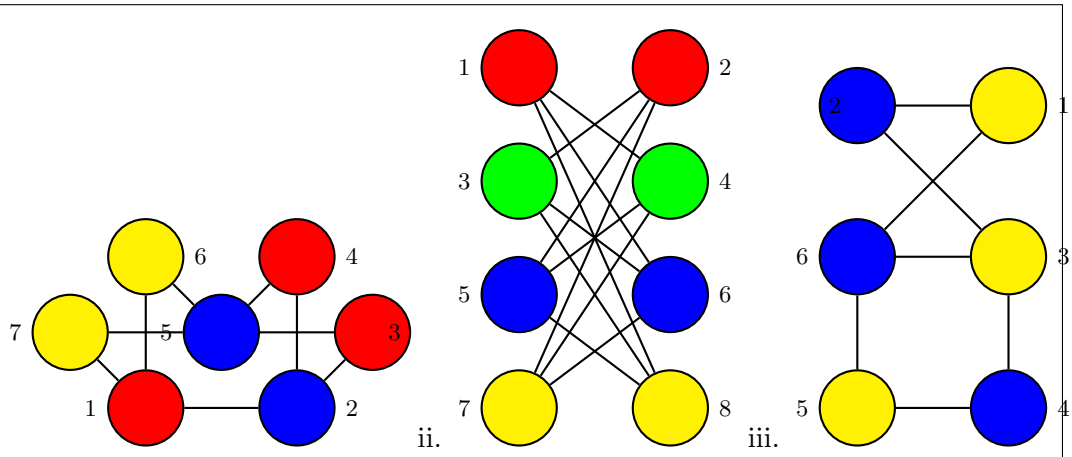
ii.



iii.



Solution:



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

(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. $k^* = 3$ using the color scheme the greedy algorithm produced. So the greedy algorithm was 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 was not optimal for this graph.

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

(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.

(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 $k \leq \max\{d(1), \ldots, d(n)\} + 1$.

2. Design an algorithm that given a positive integer n, where $n = 2^k$ is a power of 2 (k is a non-negative integer), computes the n^{th} Fibonacci number F_n, using $O(k)$ floating-point operations (multiplications or additions). Prove that the complexity $O(k)$ holds for the number of floating-point operations used in the algorithm. You **do not** need to prove the overall running time of the algorithm.

   Remarks:

   - The algorithm is **not** given k as an input.
   - If you would like to use any mathematical function, explain how to implement it.

Solution:

Input: integer n s.t. $n = 2^k$ for some non-negative integer k
Output: F_n

```
1: function FIB(n)
2:     return 1/√5 (POW((1+√5)/2, n) − POW((1−√5)/2, n))
```

Input: real number x and integer y s.t. $y = 2^m$ for some non-negative integer m
Output: x^y

```
1: function POW(x, y)
2:     if y > 1 then
3:         return (POW(x, y/2))^2
4:     else
5:         return x
```

This solution leverages the fact that Fibonacci numbers can be expressed in a closed form with the expression $F_n = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$. The first function, **FIB**, calls the helper function, **POW**, to compute $(\frac{1+\sqrt{5}}{2})^n$ and $(\frac{1-\sqrt{5}}{2})^n$. Since the **FIB** algorithm does not perform any direct multiplication, all the multiplications performed in the **FIB** algorithm come from the multiplications of the **POW** algorithm.

The **POW** algorithm will recursively compute $x^{y/2}$ and simply square the result until it reaches the base case where $y = 1$ (and thus $x^1 = x$). At every recursive iteration, there is one floating-point multiplication (squaring), so there is an overhead of $O(1)$ floating-point multiplications. The complexity (in terms of floating-point multiplications) can be represented in the following recurrence relation: POW-X(y) = POW-X$(y/2)$+$O(1)$. Applying the Master Theorem, we have $k = 1$, $b = 2$, and $d = 0$; we therefore have the second case of the Master Theorem, yielding the complexity $O(y^d \log y) = O(y^0 \log y) = O(\log y)$.

Since $y = n = 2^k$ for each of the two calls to **POW**, the complexity can be restated as $2 \cdot O(\log(2^k)) = O(k)$; thus, this algorithm uses $O(k)$ floating-point multiplications.

3. Give recurrence relations (including base cases) that are suitable for dynamic programming solutions to the following problems. You do not need to prove your correctness, but provide justification for each.

   (a) HIKING-TRIP($A[1 \dots n]$): You are going for a long hiking trip and will start at trail marker 0. There are n campgrounds along the way at miles $a_1 < a_2 < \cdots < a_n$, where each a_i is measured with respect to the starting point. You may only stop at these specified campgrounds, but are able to choose which grounds you stop at. Additionally, you must end your hiking trip at the last campground, located at a_n.

   Ideally, you would like to hike 15 miles per day, however, due to the spacing between campgrounds, this may not always be achievable. If you hike x miles on a given day, you will face a penalty of $(15 - x)^2$ for that day. Return the minimum total penalty, where the total penalty is defined as the sum of all daily penalties.

   For example, if $A = [5, 25, 27]$, then:
   $1, 2, 3 \rightarrow (15-5)^2 + (15-20)^2 + (15-2)^2 = 294$
   $12, 3 \rightarrow (15-25)^2 + (15-2)^2 = 269$
   $1, 23 \rightarrow (15-5)^2 + (15-22)^2 = 149$
   $123 \rightarrow (15-27)^2 = 144$
   Therefore, the the minimum total penalty is 144.

   Solution: Let us define $p(j)$ to be the function returning the minimum total trip penalty starting at mile 0 and ending at campground j. As a base case, we set $p(1) = (15 - a_1)^2$ because this is the penalty incurred if we end our hike at the first campground. The recurrence relation we should use is:

   $$p(j) = \min(\min_{i<j}(p(i) + (15 - (a_j - a_i))^2), (15 - a_j)^2).$$

   There are two cases for ending at campground j: either we started our day from some previous campground, or we are in the first day and started at mile 0.

   - If we started the day from a previous campground, the total trip penalty ending at campground j will consist of (1) all the total trip penalty up to but not including the last day's penalty plus (2) the penalty from the last day of the trip. We want to minimize the total of these two components over all possible prior campgrounds, giving us $\min_{i<j}(p(i) + (15 - (a_j - a_i))^2)$.
   - If we started the day from mile 0, then the total penalty is just $(15 - a_j)^2$.

   We choose the minimum over these two cases.

Alternate solution: We can simply the above by defining $a_0 = 0$, a phantom campground at trail marker 0. We define $p(j)$ as before, and the recurrence is:

$$p(j) = \begin{cases} 0 & \text{if j = 0} \\ \min\limits_{i<j}(p(i) + (15 - (a_j - a_i))^2 & \text{otherwise} \end{cases} \quad (1)$$

The reasoning is the same as above, with starting the day from a_0 equivalent to starting from mile 0.

(b) CLIMB-STAIRS(n): There are n steps to climb. Return the number of ways there are for a person standing at the bottom to reach the top if they can climb either 1, 2 or 3 steps at a time.

Solution: For an input number n, define the function CS[n] to be the number of ways there are to reach the n-th step from the bottom if only 1, 2, or 3 steps can be climbed at a time. For DP problems, it is often helpful to think about base cases. This usually involves thinking about what the algorithm would do when given the simplest or smallest inputs possible. First of all, if there are 0 steps, there is exactly 1 way to get to the top because the top is the bottom and we are already there. If there is just 1 step, there is exactly 1 way to get to the top, which is by climbing up one step from the bottom. If threre are 2 steps, we can get to the top by taking 1 step twice or by taking 2 steps once. Next, it is helpful to think of subproblems within the larger problem. When n $\geq$ 3, we can only get to the n-th stair in the next move by either stepping up from the $(n-1)$-th step, the $(n-2)$-th step, or the $(n-3)$-th step. This is because we are only allowed to climb either 1, 2, or 3 steps at a time. So the number of ways to get to the n-th step would simply be CS[$n-1$] + CS[$n-2$] + CS[$n-3$].
This yields the recurrence

$$\text{CS}[n] = \begin{cases} 1 & \text{if } n \leq 1, \\ 2 & \text{if } n = 2, \\ \text{CS}[n-1] + \text{CS}[n-2] + \text{CS}[n-3] & \text{otherwise.} \end{cases}$$

(c) *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 a strings A and B, consider transforming A into B 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 minimal cost of transforming string** A **to string** B, parameterized by 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.

Write base case(s) and a recurrence relation for computing EDIT-DIST(A, B). *Hint: this should resemble the LCS recurrence relation from lecture.*

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 restrict and make explicit which subproblems we will solve (that is, all subproblems correspond to prefixes of A and B, starting at their *first* characters).
There are two **base cases**:

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

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 the cost of transforming A into B is equal to the cost of transforming $A[1 \ldots i-1]$ into $B[1 \ldots j-1]$, because the shared final character is already in place and never needs to be touched. (It is possible, but tedious, to prove that there is an optimal transformation having this form.) Thus, in this case we have:

$$\mathbf{ED}(i, j) = \mathbf{ED}(i-1, j-1)$$

In the other case where $A[i] \neq B[j]$, we have three options:

i. insert the character $B[j]$ at the end of A to make $A[i{+}1] = B[j]$ (then optimally transform $A[1 \ldots i]$ to $B[1 \ldots j-1]$);

ii. delete the ith character from A (then optimally transform $A[1 \ldots i-1]$ to $B[1 \ldots j]$);

iii. substitute the ith character of A with the jth character from B, to make $A[i] = B[j]$ (then optimally transform $A[1 \ldots i-1]$ to $B[1 \ldots j-1]$).

A best choice among these three options corresponds to the actual edit distance (again, this can be proved tediously), so we have:

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

(d) MATRIX-MULTIPLICATION($M[1 \ldots n]$): Given a sequence of matrices to multiply together, determine the minimum number of element-element multiplications required (you may disregard additions).
As an example, suppose the sequence of matrices to multiply is ABC, where A is 2×3, B is 3×4, and C is 4×5.
$(AB)C$ requires $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 64$ multiplications.
$A(BC)$ requires $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 90$ multiplications.
Therefore, the minimum number of multiplications required is 64.

The following is a short example of how to perform matrix multiplication:
If $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ and $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$, then $AB = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$.

Remarks: Matrix multiplication is associative. Knowledge of Linear Algebra is not required for this question. Read more about matrix multiplication here: `https://en.wikipedia.org/wiki/Matrix_multiplication`

Solution: Given two matrices with dimensions $m \times n$ and $n \times p$, we require mnp multiplications because the resultant matrix is of dimension $m \times p$ and each element requires n multiplications. Let us define $L(i,j)$ to be the minimum number of multiplications required to multiply the i^{th} through j^{th} matrices. The base cases are when we only have a single matrix, i.e. $i = j$, in which case no multiplications are required: $L(i,i) = 0$ for each i.
Let each $M[i]$ be of the form (r,c), where r is the number of rows and c is the number of columns in the matrix. Then

$$L(i,j) = \min_{i \le k < j} (L(i,k) + L(k+1,j) + M[i].r \cdot M[k].c \cdot M[j].c).$$

This recurrence will find the minimum by recognizing that each sequence of matrices that are multiplied have a final multiplication, therefore, by checking each final breakpoint, we can find the breakpoint that corresponds to the minimum number of multiplications.

4. Due to high demand, the EECS Department has finally decided to install a vending machine, called the Bob and Betty Beyster Building Beverage Bestower. Like any decent vending machine, the B^6 must provide change to customers after purchases. Because this is a building teeming with computer scientists, the machine should return the smallest number of coins possible for any given amount of change. Due to design constraints, the machine only has space for three types of coins: 1¢, 5¢, and 26¢. The student designing the machine (who did not take EECS 376) decides that it should use a greedy algorithm to dispense change. Specifically, it first returns as many 26¢ coins as it can (without overpaying), then as many 5¢ coins as it can, then the appropriate number of 1¢ coins. The student hopes that this strategy will always return an optimal (smallest) number of coins; in this question, you (an EECS 376 student) will prove that this is indeed the case.

   (a) Let c denote the amount of change and let n_1, n_5 and n_{26} respectively denote the number of 1¢, 5¢, and 26¢ coins returned by the greedy algorithm. Likewise, let n_1^*, n_5^* and n_{26}^* denote the number of 1¢, 5¢, and 26¢ coins in an optimal solution. (That is, the sum $n_1^* + n_5^* + n_{26}^*$ is minimal.)

      i. Show that $n_1^* \leq 4$.

      Solution: We want to show that in an optimal solution, the number of 1¢ coins (n_1^*) is at most four. Suppose, for establishing a contradiction, that $n_1^* \geq 5$. Then we may simply replace five of the 1¢ coins by one 5¢ coin, thereby reducing the total number of coins returned by four, which contradicts the assumed minimality of the original solution.

      ii. Show that $5n_5^* + n_1^* \leq 25$. In other words, the *total amount* given in 1¢ and 5¢ coins at most 25.

      Solution: We shall continue with a another proof by contradiction, to show that $5n_5^* + n_1^* \leq 25$. For contradiction suppose that the total value change associated to n_1^* and n_5^* is more than 25 cents, i.e., $5n_5^* + n_1^* > 25$. If $n_1^* = 0$, then $5n_5^* \geq 30$, i.e., $n_5^* \geq 6$. Then we can replace six of the five-cent coins with one 26-cent and four one-cent coins, thereby contradicting optimality. If $n_1^* = 1$, then $5n_5^* \geq 25$, i.e., $n_5^* \geq 5$. Then we can replace five five-cent coins and one one-cent coin with a single 26-cent coin, again contradicting optimality. The same is true if $n_1^* = 2$, $n_1^* = 3$, or $n_1^* = 4$. These are the only cases because $n_1^* \leq 4$ by the previous part.

      iii. Show that $n_{26}^* \geq n_{26}$ and conclude that $n_{26}^* = n_{26}$.

      Solution: We want to show that an optimal solution chooses at least as many 26¢ coins as the greedy solution does. Assume for contradiction that it doesn't for some total amount of change

      $$c = n_1^* + 5n_5^* + 26n_{26}^* = n_1 + 5n_5 + 26n_{26}.$$

      By assumption, $n_{26}^* < n_{26}$, so subtracting $26n_{26}^*$ from both sides gives us

      $$n_1^* + 5n_5^* = n_1 + 5n_5 + 26(n_{26} - n_{26}^*) \geq n_1 + 5n_5 + 26.$$

Because n_1, n_5 are non-negative, this means that $n_1^* + 5n_5^* \geq 26$, but this contradicts the previous part.

iv. Similarly, conclude that $n_5^* = n_5$. **Hint:** Use the previous parts.

Solution: By the above, $n_{26}^* = n_{26}$, so we have $n_1^* + 5n_5^* = n_1 + 5n_5$. To show that $n_5^* = n_5$, we argue similarly to the previous part. If $n_5^* < n_5$, then by subtracting $5n_5^*$ from both sides we have $n_1^* \geq n_1 + 5 \geq 5$. This contradicts the first part.

v. Conclude that $n_1^* = n_1$. That is, the greedy algorithm outputs an optimal solution

Solution: From the previous parts we have $n_{26}^* = n_{26}$ and $n_5^* = n_5$. It follows immediately that we must have $n_1^* = n_1$ as well, so the greedy solution is in fact optimal.

(b) In a parallel universe, the B^6 only has space for 1¢, k¢ and 26¢ coins, where k is an integer strictly between 1 and 26. Find a k such that the greedy approach does **not** return the optimal number of coins for every amount of change. Note that you just proved in (a) that with $k = 5$ the greedy algorithm gives the optimal answer.

Solution: It is important to note that one counter-example will prove that not all amounts of change are optimal. Consider returning 30¢ in change with the denominations 1¢, 10¢ and 26¢ (here, $k = 10$). The greedy algorithm will return change in the following order: 26¢, 1¢, 1¢, 1¢, 1¢ (a total of 5 coins). However, upon observation, we see that an optimal solution is 10¢, 10¢, 10¢ (a total of 3 coins). $k = 10$ is just one such example, but any k between 8 and 25 (inclusive) besides k = 9 and k = 13 will also result in a non-optimal greedy algorithm.

(c) Although the student designing the B^6 is sad that the greedy algorithm will not always work for dispensing change in this universe, you (an EECS 376 student) have learned that you can find the optimal number of coins for any set of coin denominations using dynamic programming. Write base case(s) and the recurrence relation for computing OPT-COIN-CHANGE(C), where C is the amount of change you want to make. You can only use 1¢, k¢ and 26¢ coins (k is the value you found in (b)). Briefly justify why this will be optimal regardless of denominations.

Solution: We define our denominations $d = \{1, k, 26\}$.
First, the base case is when when $C = 0$, OPT-COIN-CHANGE(C) = 0 because there is no dispensing of any change in this scenario.
Next, the recurrence relation is:
OPT-COIN-CHANGE(C) = $\min_{i:d_i \leq C}\{1+\text{OPT-COIN-CHANGE}(C-d_i)\}$. This will always be optimal regardless of denominations because, for any amount of change C, the recurrence relation considers all denominations $\leq C$, and selects the one that will lead to the lowest total number of coins needed.