CSC373H1: Assignment 1

Junming Zhang: 1003988982
Yuchen Tong: 1003534669
Yuchen Fan: 1003800265

Due: October 14th, 2019 before 11:59 p.m.

Instructions

1. Be sure to include your name and student number with your assignment. Typed assignments are preferred (e.g., PDFs created using LaTeX or Word), especially if your handwriting is possibly illegible or if you do not have access to a good quality scanner. Please submit a single PDF on MarkUS at `https://markus.teach.cs.toronto.edu/csc373-2019-09`
2. You will receive 20% of the points for any (sub)problem for which you write "I do not know how to approach this problem." (you will receive 10% if you leave the question blank and do not write this or a similar statement).
3. You may receive partial credit for the work that is clearly on the right track. But if your answer is largely irrelevant, you will receive 0 points.
4. This assignment has 4 questions. The total number of marks is 80.

Note: The theme of this assignment is the circus!

Q1 [25 Points] Raising the Bar

There are n clowns in the circus numbered $1, 2, ..., n$. The height of clown i is h_i. All clowns have equal-sized heads of height f. The clowns form a queue whose ordering is defined by an array of integers L, where $L[j]$ is the index of the j^{th} clown in the queue.

The first clown $L[1]$ walks under a metal bar cranked-up from the ground by the strong-man of the circus, just high enough so the clown can pass under it. Then $L[2]$ stands on $L[1]$'s shoulders and they both walk under the bar, again raised from the ground just high enough so they both can pass. This continues until a tower of clowns standing on each other's shoulders, with $L[1]$ at the bottom and $L[n]$ at the top, walk under the bar. Specifically, when the tower of clowns $L[1], ..., L[k]$ pass under the bar, the strong-man raises the bar to the height $H_k = (\sum_{j=1}^{k} h_{[L_j]}) - (k-1) * f$ The effort made by the strong-man is proportional to the cumulative height $\sum_{k=1}^{n} H_k$ to which he must crank the bar.

(a) [3 Points] Design a greedy algorithm to compute the optimal order of clowns in the queue L that will minimize the strong-man's effort.

Solution.
Algorithm:
Sort the clowns with increasing order by the height of clown, such that $h_1 \leq h_2 \leq ... \leq h_n$ and form a queue L such that L[1] is the clown with lowest height and so on. Always pick the clown with lowest height from remaining queue to stand on the previous clown's shoulder.

(b) [5 Points] Prove that your greedy algorithm from (a) will always yield a minimum-effort solution.

Solution.
Correctness:
Let C_i be effort made in the greedy algorithm after first i clowns pass through mental bar. We prove by induction on $i = 0, ..., n$ that there exists an optimal solution OPT_i that extends C_i. When $i = n$ there are no more clowns to join the tower so C_n must be optimal.
The base is $i = 0$ and $C_0 = 0$, so clearly there exits an optimal OPT_0 that extends G_0. Assume the

inductive hypothesis holds for $i-1$. We prove it for i.

- $H_i = (\sum_{j=1}^{i} h_{[L_j]}) - (i-1) * f$
  $C_i = \sum_{k=1}^{i} H_k = C_{i-1} + H_i$
- Since H_i is the minimum height for first i clowns need to passing the bar together. And these first i clowns have lowest i heights.
  We form OPT_i from $OPT_{i-1} + H_i$, so OPT_i must be optimal because OPT_{i-1} is optimal, Clearly OPT_i extends C_i,

(c) [2 Points] What is the worst-case time complexity of this algorithm?

Solution.
Worst-case time complexity:
We sort the clowns by heap sort. Therefore, the worst-case time complexity will be: $O(n \log(n))$

(d) [4 Points] Suppose now that the strong-man can only crank the bar up to a maximum height of H. We again want successive towers of clowns to pass under the bar. Unlike part (a), note that a tower consisting of all clowns may no longer be feasible given the maximum height of H. Our goal is now to construct a sequence of towers of clowns such that (a) the final tower in our solution is as tall as possible (we want the finale to be impressive!), and (b) among all such sequences, our solution minimizes the strong-man's effort. Design a greedy algorithm for this problem. Bizarrely, you are told that the shoulder heights hi f of the clowns are all unique powers of 2 (that is, each $h_i f$ is a power of 2, and $h_i f \neq h_j - f$ for $i \neq j$).

Solution.
Algorithm:

- The maximum height we need for shoulder height will be $H_s = H - f$.
- Sort the clowns with increasing order by the shoulder height S of clown, such that $S_1 \leq S_2 \leq ... \leq S_n$. for i from S_n down to S_1, if $H_s - S_i \geq 0$, add clowns with shoulder height S_i to queue and update H_s to $H_s - S_i$. And for each S_i will only be used once.
- if there is no more $H_s - S_i \geq 0$ satisfied, Those clowns who are added into queue will consist the tower. And when we insert clowns into the queue, it follows by decreasing order. Therefore we pop the clowns by increasing order, always pick the clowns with lowest height from the remaining queue and stand on the shoulder of the previous popped clowns.

(e) [5 Points] Prove that the your greedy algorithm from (d) will always yield an optimal solution.

Solution.
Correctness:

(a) Firstly, I need to prove the final tower is as tall as possible.
Let T_i is height of tower after first i clowns has been inserted into queue. Also $T_i \leq H_s$, We prove by induction by $i = 0, ...n$ that there exits an optimal OPT_i that extends T_i.
The base is $i = 0$ and $T_i = 0$, there are no clowns in the tower, so clearly there exists an optimal OPT_0 that extend G_0. Assume the inductive hypothesis holds for $i-1$, we prove it for i.

- $T_i = T_{i-1} + S_i$
- Since we need to satisfy $T_{i-1} + S_i \leq H_s$ and All S is unique power of 2, S_i is the largest increment in the remaining combinations of rest of shoulder heights in clowns. Because $\sum_0^{n-1} 2^i = 2^n - 1$. S_i must in the optimal solution.
  We form OPT_i from $OPT_{i-1} + S_i$, so OPT_i must be optimal because OPT_{i-1} is optimal, clearly OPT_i extends T_i.

(b) Secondly, after proving the maximum height we can reach, we need to minimize the effort. From Part(a), if we always pick the clowns with lowest height from remaining queue to the tower, we have proved that it will yield a minimum-effort solution.

(c) Overall, the greedy algorithm will always yield an optimal solution.

(f) [3 Points] What is the worst-case time complexity of your algorithm from (d)?

Solution.
Worst-case time complexity:
Firstly, we have to sort the clowns with heap sort by shoulder height of each clowns. it takes $n\log(n)$. And we need to find out the clowns who will be in the tower, it takes constant time for each clowns to compare current height of tower with maximum height, the total time is $O(n)$. Therefore worst-case complexity is $O(n\log(n)) + O(n) = O(n\log(n))$

(g) [3 Points] Let us go back to part (a) without any maximum height restriction. Suppose now that the clowns also have girths, where the girth of clown i is g_i. To allow a tower of clowns $L[1], ..., L[k]$ to pass, the strong-man now needs to both raise the bar to the height Hk and spread two vertical posts by width $W_k = max(g_{L[1]}, ..., g_{L[k]})$, which is just wide enough for the clowns to pass. The effort made by the strong-man is now proportional to the cumulative area $\sum_{k=1}^{n} W_k \cdot H_k$. Is your greedy algorithm of (a) still guaranteed to provide a solution that minimizes the strong-man's effort? Either prove that it does, or provide a counter-example.

Solution.
No, We cannot guarantee it provides a solution that minimizes the effort. if the clowns with lowest height has the extremely large width W_1. The cumulative effort will be $C = W_1 \cdot \sum_{k=1}^{n} H_k$ and it will maybe much larger than the solution which we put the clowns with widest girth at the end of queue. $C = W_1 \cdot H_k + \sum_{k=1}^{n-1} W_k \cdot H_k$

Q2 [25 Points] Circus Arena

The circus handyman tasked with creating a roped off arena in the circus arbitrarily lays n stakes into the ground at points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ while sleepwalking. In the morning, a rope is placed loosely surrounding all the stakes and then tightened to mark the arena.

(a) [7 Points] Design a divide and conquer algorithm to determine the clockwise circular sequence of stakes that anchor the rope. Formally, your algorithm should output an array $P[1, ..., k]$ such that $P[1]$ is some anchor touching the rope, $P[2]$ is the next anchor touching the rope in the clockwise order, $P[3]$ is the next anchor touching the rope in the clockwise order, and so on. The divide step should reduce the problem into approximately equal-sized smaller instances of the problem. The combine or merge step should then use the circular sequences of stakes that anchor sub-arenas to compute the solution for the single larger arena.

Solution.

Algorithm:

- Sort the list of coordinates by x coordinates using merge sort.
- For a input list L, divide it into left half of list A and right half B.
- Recursively determine the convex hull for A and B.
  If the the size of the list is less than 5. Compute the convex hull for both A list and B list using brute force: Let the left most coordinate be p_0, sort the rest of the points p_i by the angle between vector (0,1) and $(p_i - p_0)$. Start from p_1 to find a pair (p_1, p_t) such that every other coordinates fall in one side of the line determined by these two coordinates. If p_t was found, then repeat the process start from p_t to another coordinate, until some pair (p_k, p_0) satisfies. Thus we get a convex hull with coordinates stored in clockwise.
  (We determine p_z is on which side of a line by calculating the equation of the line using 2 coordinates, substitute the y value of p_z to get a x_0, if the x value of p_z is greater than x_0, then p_z is on the right side of the line, otherwise it is on the left side.)
- Merge two convex hull using the algorithm as follows:
- Suppose the middle coordinate is M. Choose the coordinate that is the left most closest to M as C_1. Choose the coordinate that is the right most closest t to M as C_2. define the vertical line pass

through M as L_1, the line pass through C_1 and C_2 as L_2, the intersection of L_1 and L_2 as I_1. The distance from M to I_1 is recorded as D_1.

- Find the upper tangential coordinates Up_A, Up_B: First move C_2 to the next coordinate clockwisely. If D_1 decreases, move C_2 back, else update C_2 as the current coordinate. Then move C_1 to next coordinate counterclockwisely. If D_1 decreases, move C_1 back, else update C_1 as the current coordinate. Repeat the procedure above until D_1 hits the maximum. Thus, the coordinate of C_1 and C_2 are the upper tangential coordinates.
- Find the lower tangential coordinates Low_A, Low_B: Same as finding the upper tangential coordinates except C_1 moving clockwisely while C_2 moving counterclockwisely to minimize D_1.
- Replace the coordinates located at the right side of the line passes through Up_A and Low_A in the left convex hull with the list of coordinates located at the right side of the line passes through Up_B and Low_B in right convex hull. The remaining coordinates forms the new larger convex hull stored in clockwise direction.

(b) [7 Points] Prove the correctness of your algorithm in (a).

Solution.

Proof for brute force algorithm for finding convex hull:

- By induction,
- Base Case: Consider when the size of the list is 3 (coordinates non-colinear), we start form the left most coordinate p_1 and sort the other 2 by the angle between vector (0,1) and $(p_i - p_1)$. The sorted list is a clockwise convex hull. Thus the base case holds.
- Induction step: Suppose the size of the list is less than or equal to n, then the elements being chosen forms the clockwise convex hull.
- Want to show that the hypothesis holds for list of size n+1. Since the hypothesis holds for list of size n. Then the convex hull with list of size n+1 is same as the clockwise convex hull with list of size n plus some new elements chosen from the pair $(x_i, x_n + 1), i = 1, 2, 3...n$. If there is a pair that makes every other points fall in the same side. replace the new elements with the old elements in the n coordinates convex hull.
- Therefore, the brute force method for finding convex hull is correct.

Proof for the merge algorithm:

- Since both hull A on the left and hull B on the right are convex hulls. Based on the convexity and direction of the coordinates being stored, if there is a line L passes through one coordinate from A to another coordinate from B that has a maximum distance from the middle point M to the intersection of the vertical line passes through M and the line L. Then this line is the tangential line to both convex hull. The coordinates that is on the line are on the newer merged convex hull.

Proof for the main algorithm:

- By induction,
- Base case: Consider when size of the list is less than 5, then the brute force algorithm for finding convex hull successfully finds the correct convex hull based on the previous proof.
- Induction steps: Suppose the size of the list is less than or equal to n, then the algorithm produces the correct convex hull for n coordinates in the list.
- Want to show that the hypothesis holds for the list of n+1 coordinates. The main algorithm divides the list into A and B sub list. Based on the hypothesis, the algorithm will produce the correct convex hulls CA and CB from list A and list B respectively since the size of list A and B are smaller than n. Using the merge algorithm, a larger convex hull will be correctly created based on the correctness of the merge algorithm from above.
- Therefore, the main algorithm correctly produce a convex hull from input lists.

(c) [3 Points] What is the worst-case time complexity of the algorithm?

Solution.

Worst-case time complexity:

- The merge sort at the very beginning is O($nlogn$).
- The merge operation for the convex hulls is O(n) since suppose there are both n coordinates in left and right convex hulls, we at most traverse 2n times for determining the upper or lower tangential coordinates.
- Hence the recurrence of the algorithm is: $T(n) = 2T(n/2) + O(n)$
- Therefore, by master theorem, the worst-case time complexity is O($nlogn$).

(d) [3 Points] Suppose now that a number of nested arenas are created by recursively tightening a rope around stakes left inside the outer arena (i.e. stakes not touching the outer rope). What is the worst-case time complexity of recursively running your algorithm from part (a) until all nested arenas are created so that each stake is touching a rope?

Solution.

Worst-case time complexity for finding all nested arenas:

- Worst-case time complexity for finding all nested arenas:
- Finding one layer using algorithm in a) is O($nlogn$).
- There are at most n/k layers.
- Thus the worst-case time complexity is O(n^2logn).

(e) [5 Points] Suppose lions start parachuting into this arena. Given the output from (a), design an recursive $O(\log k)$ time algorithm to check if a lion touching down at point (u, v) will land inside the arena (here, k is the length of the array found in (a)).

Solution.

Algorithm:

- For input list L (known that the coordinates are stored in clockwise order) and a coordinate p_z, pick 3 coordinates that are non-colinear, find the centroid c_0 of these 3 coordinates.
- Consider c_0 as the origin, binary search the wedge in which p_z lies. If p_z is on the right side of the line passing through c_0, p_k while it is on the left side of the line passing through c_0, p_{k+1}, then p_z lies in wedge $(p_k c_0 p_{k+1})$. We determine p_z is on which side of a line by calculating the equation of the line using 2 coordinates, substitute the y value of p_z to get a x_0, if the x value of p_z is greater than x_0, then p_z is on the right side of the line, otherwise it is on the left side.
- If p_k and p_{k+1} are found, if p_z is on the right side of the line passing through p_k, p_{k+1} then it is outside the arena, otherwise it is inside the arena

Q3 [10 Points] Lions and Tamers

There are m lions $L_1, ..., L_m$ and nm tamers $T_{m+1}, ..., T_n$ on pedestals connected by planks. The plank connecting lion/tamer i with lion/tamer j has length $l(i, j)$, for each $i, j \in \{1, ..., n\}$. You want to select a subset of planks that satisfy the following conditions:

- No lions are directly connected to each other by a plank.
- Every lion is directly connected by a plank to at most one tamer.
- There is a path of planks connecting any two tamers.

(a) [5 Points] Describe an algorithm that, subject to the above conditions, selects the subset of planks minimizing the total length. (You can directly refer to any algorithm mentioned in the lecture slides; you do not need to describe such an algorithm in detail.)

Solution.

Algorithm:

- Create a complete graph G_0 with all the vertices are connected together, the vertices are tamers $T_{m+1}, ..., T_n$ while the edges connects tamer i and tamer j with given length $l(i, j)$, for each $i, j \in \{1, ..., n\}$.
- Run Prim's MST algorithm on G_0. Store all the edges from the MST as (i, j) in list L_r.
- For each lion L_i from $L_1, ..., L_m$ find the minimum length L_{min} from L_i to every tamer $T_{m+1}, ..., T_n$, it the minimum length is $l(i, k)$, append the minimum edge (i, k) to L_r.

(b) [5 Points] Prove that correctness of your algorithm and analyze worst-case time complexity.

Solution.

Proof of correctness:

- By running prim's MST algorithm on a complete graph of tamers, we get the minimum spanning tree of the tamers, by the definition of MST, all the vertices are connected without circles and has the minimum edge wight. Hence it satisfies there is a path of planks connecting any two tamers and also the path has minimum length.
- For each lion find a minimum path to a certain tamer among all tamers guarantees that no lions is connected to each other since we only find the path from lions to tamers. Also, find the exact minimum path satisfies the constraint that every lion is directly connected by a plank to exact one tamer.
- The output of the algorithm satisfies all the constraints.
  Now want to show that the total path is minimum. Since all tamers are connected with a minimum path created by prim's MST algorithm and all the paths that every lion connect to a tamer are minimum. Union both two minimum paths yields the final minimum path.

Worst-case complexity:

- The time complexity for creating a complete graph with n - m vertices is O($(n - m)^2$)
- The time complexity for running prim's MST algorithm on graph with (n - m) * (n - m - 1) edges is O($(n - m)(n - m - 1)log(n - m)$)
- The time complexity for looping through all m lions to find the minimum path to (n - m) tamers is O($m(n - m)$)
- Hence the total time complexity is O($(n - m)^2 + (n - m)(n - m - 1)log(n - m) + m(n - m)$)

Q4 [20 Points] Trapeeze

There are $m + n$ acrobats on a long horizontal circus rope. Among the acrobats, m are "holders" suspended upside-down and pulling the rope up, while the rest n are "hangers" hanging from the rope and pulling the rope down. The force applied by acrobat k from the left end of the rope is f_k, and index $i_k \in$ {holder,hanger} indicates whether acrobat k is a holder or a hanger. Acrobats at the two ends of the rope, i.e. acrobats 1 and $m+n$, are holders.

For the safety of the rope, it is important that the net force pulling down in any segment of the rope (i.e. the total force applied by hangers in that segment minus the total force applied by holders in that segment) does not exceed F.

(a) [5 Points] Design a divide and conquer algorithm to determine if the rope is safe given $f_1, ..., f_{m+n}$ and $i_1, ..., i_{m+n}$.

Solution.
Algorithm:

- If there are more than one force on the rope, divide rope into two equal segments, repeating dividing until there is only one force on the segment.
- compute that force and direction on segment, also check whether this segment is safe, then return the value.
- For the rope which be divided to two parts, get the net forces from two sub-segments, then compute the net force on the rope and check whether the rope is safe.

```
def Net_Force(force_acrobat, index):
    size = len(force_acrobat)
    # there is only one force in this segment
    if size == 1:
        f = force_acrobat[0]
        # each force cannot be greater than F
        if f > F:
            return False
        # find out who apply this force,
        # holder, negate the value of force
        if index[0] == holder:
            f = -f
        return f
    else:
        # there are more than one force in this segment
        half = size / 2
        # separate the rope into two equal segments
        first_f = Net_Force(force-acrobat[:half+1], index[:half+1])
        second_f = Net_Force(force-acrobat[half+1:], index[half+1:])
        # both segments are safe
        # compute the net force on this segment
        if first_f and second_f:
            f = first_f + second_f
            # net force cannot be greater than F
            if abs(f) > F:
                return False
            else:
                return f
        # one of segment or both are not safe
        else
            return False
```

(b) [5 Points] Prove that your algorithm is correct, and analyze its worst-case time complexity.

Solution.

Proof of correctness:

- By induction,
- Base case: Consider when there is only one acrobat, if the force f exceeds F the program terminates and return false indicates it is not safe, otherwise apply a direction to the force f and return it. Therefore, the base case holds.
- Induction steps: Suppose the program returns the correct value when there are less than or equal n+m acrobats on the rope, that is, the program returns false if the rope is not safe otherwise returns the net force of this segment.
- Want to show that the hypothesis holds for the rope with n+m acrobats. The algorithm divides the list into A and B sub list. Based on the hypothesis, the algorithm will produce the correct result since the size of list A and B are smaller than n+m. If either the left sub or the right sub return false, then the algorithm return false to indicate that the rope is not safe otherwise it is safe. Therefore, it holds for the rope with n+m acrobats.
- Therefore, by induction, The algorithm is correct.

Worst-case time complexity:

- The recurrence for the algorithm: $T(m+n) = 2T((m+n)/2) + O(1)$
- By master theorem, the worst-case time complexity is: $O(m+n)$

(c) [5 Points] Among the $m+n$ acrobats, you are now given which acrobats know which other acrobats. You need to assign them costumes of different colors such that no two acrobats who know each other are assigned costumes of the same color. Design a greedy algorithm to assign costumes to acrobats which uses at most one more color than the maximum number of acrobats that any acrobat knows. Prove that your algorithm produces a valid coloring and uses no more than the desired number of colors.

Solution.
Assume the maximum number of other acrobats one acrobat know is k, such that $0 \le k \le m+n-1$. For the greedy algorithm below, three parameters are needed for coloring:

- Graph: build an undirectional adjacency list graph such that each acrobat is a vertex in this graph, and each edge connects to two vertices that two acrobats represented by know each other. The head of each adjacency list is one acrobat, the the following vertices represent the acrobats this acrobat knows. The are $m+n$ adjacency lists in the graph.
- ColorList: build a list which contains $k+1$ distinct colors.
- NumNodes: number of acrobats, which equals to $m+n$.

The algorithm is shown as below:

```
class acrobat:
    def __init__(self, color=None):
        self.color = color
    def set_color(color):
        self.color = color

def Color_Greedily(Graph, ColorList, NumNodes):
    colorSet = set(colorList)
    for adjacencyList in Graph:
        colorUsed = set()
        #number of acrobats the head acrobat know, including himself
        numKnown = len(adjacencyList)
        acrobatIndex = 0
        while acrobatIndex < numKnown:
            acrobat = adjacencyList[acrobatIndex]
            if acrobat.color is None:
                colorNotUsed = colorSet.difference(colorUsed)
                acrobat.set_color(colorNotUsed.pop())
            colorUsed.add(acrobat.color)
            acrobatIndex += 1
    return Graph
```

Proof:

I. Produces a valid coloring.
In the inner loop of this algorithm, the color of each colored acrobat is put into the set colorUsed. And then if a acrobat not colored is found, color him with the color popped from the set, which is the difference between the colorSet, the set form of colorList that holds all different colors may be used in the algorithm ($m+n = k+1$, k is the maximum number of other acrobats an acrobat know.) and the set colorUsed (colors have not been used yet, this set has at most k before last acrobat being searched since one adjacency list holds at most $k+1$ acrobat because one acrobat knows at most $m+n-1 = k < k+1$ acrobats). Thus an acrobat is always colored differently with the acrobat he knows.

II. Uses no more than the desired number of colors.
Assume the i^{th} acrobat is being searched (i is the index of the adjacency list), $1 \le i$, and assume all $i-1$ acrobats searched have different colors. Since there are at most $m+n$ acrobats know each other, and the maximum number of other acrobats an acrobat know is k, $i \le k < k+1 \le m+n$, thus each adjacency list is at most $m+n$ long. A colorSet holds $k+1$ different colors in this algorithm.

One color can be added to the set colorUsed after searching each acrobat vertex (colored acrobat or coloring an uncolored acrobat), and each acrobat before i^{th} acrobat is colored differently from other colored acrobats, thus the length of colorUsed is less than $k+1$ before the last acrobat is searched, hence less than the length of colorSet. Thus the difference between the colorSet and the set colorUsed always has the length greater than 0 until the inner loop is over. Therefore, no more than the desired number of colors can be used, which is 1 plus the maximum number of the other acrobats an acrobat know in the group $(k+1)$.

(d) [5 Points] Provide a counterexample to show that the algorithm from (c) does not always produce costume assignments with the smallest possible number of colors.

Solution.
Counterexample:

Assume there are $m+n$ acrobats, $(m+n) \mod 2 = 0$ (the group has even number of acrobats and 0 is vacuous, thus consider there are at least 2 acrobats in the group), and there is one bipartite graph, that is, all vertices can be separate to two sets, V_1 and V_2, all vertices in one set do not connect with each other (all acrobats represented by vertices in the same vertex group do not know each other), and a vertex from one set connects to all vertices in another set (an acrobat knows all acrobats in another set). Let each vertices set has $\alpha = \frac{m+n}{2}$ vertices, that means each acrobat knows $(\alpha+1) \leq (m+n)$ acrobats, including himself. Based on the algorithm, $\alpha+1$ colors will be used for coloring all acrobats. However, in the vertex representation of each acrobat, since each acrobat does not know other acrobats who share the same vertex set in a bipartite graph, all acrobats in the same vertex set can share the same color. Since there are two separate sets in a bipartite graph, indeed, 2 different colors are needed to color all acrobats in this situation, and $2 \leq (\frac{2}{2}+1) \leq (\alpha+1)$. Therefore, the algorithm fails to provide the smallest possible number of colors.