CSC373 Fall'19
Assignment 1

Due Date: October 13, 2019, by 11:59pm

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. Please attempt them all.

Note: The theme of this assignment is the circus!

Q1 [25 Points] Raising the Bar

There are n clowns in the circus numbered $1, 2, \ldots, n$. 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 $L[1], \ldots, L[k]$ pass under the bar, the strong-man raises the bar to the height $H_k = \left(\sum_{j=1}^{k} h_{L[j]}\right) - (k-1) * f$. 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: Sort the clowns in order of ascending height, and output that order.

(b) [5 Points] Prove that your greedy algorithm from (a) will always yield a minimum-effort solution.
solution: The total effort is $\sum_{k=1}^{n} H_k = \left(\sum_{j=1}^{n}(n+1-j) \cdot h_{L[j]}\right) - (n-1) \cdot n \cdot f/2$. If $i < j$ and $h_{L[i]} > h_{L[j]}$, then swapping i and j decreases the effort by $(j-i)(h_{L[i]} - h_{L[j]}) > 0$.

(c) [2 Points] What is the worst-case time complexity of this algorithm?
solution: Sorting takes $O(n \log n)$ time.

(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 $h_i - f$ 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

Calibrate the problem to shoulder heights by letting $h' = h - f$, and $H' = H - f$. Sort the clowns in an array M by descending height. Keep a boolean array $C[1..n]$ where $C[i]$ indicates if clown $M[i]$ is chosen to walk under the bar.

```
$ht = 0$;
initialize $C[1..n] = false$;
for i=1..n { if ($h'_{M[i]} + ht \leq H'$) { $ht \leftarrow ht + h'_{M[i]}$; $C[i] = true$;}}
for i=n..1 { if ($C[i]$) append $M[i]$ to $L$ }
```

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

solution

First we show that the greedy solution defined by C produces the tallest possible tower of clowns of height $\leq H$. Assume there exists an index k such that the greedy algorithm adds the clown $M[k]$ to the tower, but the optimal solution does not. Fix k to be the first such index. Let x be the total height of all previous clowns in the greedy solution. Then the greedy solution gives a tower of height at least $M[k] + x$, but the optimal solution gives a tower of height at most $M[k] + x/2 + x/4 + x/8 + \ldots + 1 < M[k] + x$, which is a contradiction.

Now, given the subset of clowns that can pass under the bar, queuing them in ascending order of height minimizes effort as shown in part (b).

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

solution: Sorting the clowns by height is $O(n \log n)$, the greedy selection of clowns is $O(n)$, thus overall $O(n \log n)$.

(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 $L[1], \ldots, L[k]$ to pass, the strong-man now needs to both raise the bar to the height H_k *and* spread two vertical posts by width $W_k = \max(g_{L[1]}, \ldots, g_{L[k]})$, 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: counterexample: Say we have two clowns (height, girth) = {(1,100),(10,1)}. The greedy solution of (b) requires effort = 1*100+11*100 = 1200, whereas the effort with the opposite order is 10*1+ 11*100= 1110.

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 $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ 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 $P[1, \ldots, k]$ 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

See https://iq.opengenus.org/divide-and-conquer-convex-hull/ for a helpful visual. Assume for simplicity that no three points are colinear.

1. Sort the points in order of increasing x-coordinate. This step is done only once globally, not once per recursive call. Let us say the points $Q[1..n]$ are in increasing sorted order of x-coordinate.

2. P =convexhull$(1, n)$; // returns points on hull in clockwise order with leftmost point as P[1].

3.
   ```
   convexhull($a, b$) {
   // base cases:
   if $b == a$ return Q[a]; // one point
   if $b == a + 1$ return (Q[a], Q[b]); // 2 points
   A=convexhull(a,floor((a+b)/2)); //left half convex hull
   B=convexhull(floor((a+b)/2)+1,b); // right half convex hull
   return mergehulls(A,B);
   }
   ```

4.
   ```
   mergehulls($A, B$) {
   reverse the order of points to counter-clockwise in $A$;
   find the right most point $A[k]$, and shift-rotate the array $k - 1$ points to the left so that A[k] is at A[1];
   // NOTE: Treat arrays as circular (wraparound) when incrementing/decrementing indices

   // find points $A[u_A]$ in $A$ and $B[u_B]$ in $B$ that define the upper tangent as follows:

   $u_A = 1$ and $u_B = 1$; // start with rightmost point in $A$ and leftmost point in $B$
   while true {
   if (size($A$)> 1 and $A[u_A + 1]$) is above LineThrough($A[u_A], B[u_B]$) then increment $u_A$.
   else if (size($B$)> 1 and $B[u_B + 1]$) is above LineThrough($A[u_A], B[u_B]$) then increment $u_B$.
   else break;
   }
   ```

```
// The upper tangent bounding convex shapes from above is LineThrough($A[u_A], B[u_B]$).


// similarly find points $A[l_A]$ in $A$ and $B[l_B]$ in $B$ that define the lower tangent as follows:

$l_A = 1$ and $l_B = 1$; // start with rightmost point in A and leftmost point in B
while true {
if (size($A$)> 1 and $A[l_A - 1]$) is below LineThrough($A[l_A], B[l_B]$) then decrement $l_A$.
else if (size($B$)> 1 and $B[l_B - 1]$) is below LineThrough($A[l_A], B[l_B]$) then decrement $l_B$.
else break;
}
// The lower tangent bounding convex shapes from below is LineThrough($A[l_A], B[l_B]$).

set the merged hull $C$ to be
$[A[l_A], A[l_A - 1], \ldots, A[u_A], B[u_B], B[u_B + 1], \ldots, B[l_B]]$ ("$\ldots$" wraps around the arrays);
find the left most point $C[m]$, and shift-rotate the array $m - 1$ points to the left so $C[m]$ is
$C[1]$;
return $C$;
}
```

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

solution:
By induction on the number of points n.
The base cases of one and two points clearly return convex hulls, a point and line respectively.
Computing the convex hull of n points sorts the points in x and then calls convexhull for $n/2$ points on the left and right. By induction we can assume that the convex hull's for $n/2$ points is correct and the points are contained within the convex shapes A and B.
Given that we can find a vertical dividing line between A and B, one can geometrically see that the overall convex hull is bounded above and below by upper and lower tangents to the shapes. It is also geometrically clear that u_A, u_B defines the upper tangent when the iterative loop terminates (and l_A, l_B the lower tangent).
It is less obvious that the loop does not infinitely cycle around the points of A and B. For this assignment it is enough to observe that the iteration of u_A can not iterate past the left most point of A (imagine convex hulls where B is much higher vertically than A and the upper tangent is almost a vertical line touching A at the left most point) and the u_B past the right most point of B. These two points bound the loop from iterating forever. A symmetric argument can be made for the lower tangent.

(c) [3 Points] What is the worst-case time complexity of the algorithm?
solution: Sorting takes $O(n \log n)$ time. The runtime of the rest of the algorithm is described by the recurrence $T(n) = 2T(n/2) + O(n)$, of which the solution is $T(n) = O(n \log n)$. So overall, the runtime is $O(n \log n)$.

(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: Three points are enough to anchor the rope (think of a large triangle that enclose all the other points). We can thus have n/3, or $O(n)$ nested arenas, and each recursive call takes $O(n \log n)$ time, so in total this could take $O(n^2 \log n)$ time.

(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

```
point-in-convex-polygon(point $x$, polygon $P = [v_0, ...v_k]$)
{
if $k = 3$ return point-in-triangle($x$,$P$);
else if $x$ and $v_1$ are on the same side of LineThrough($v_0, v_{\lceil k/2 \rceil}$)
then return point-in-convex-polygon($x, [v_0, v_1, \ldots, v_{\lceil k/2 \rceil}]$)
else return point-in-convex-polygon($x, [v_{\lceil k/2 \rceil}, v_{\lceil k/2 \rceil+1}, \ldots, v_k, v_0]$).
}
```

Q3 [10 Points] Lions and Tamers

There are m lions $L_1, \ldots, L_m$ and $n - m$ tamers $T_{m+1}, \ldots, T_n$ on pedestals connected by planks. The plank connecting lion/tamer i with lion/tamer j has length $l(i, j)$, for each $i, j \in \{1, \ldots, n\}$. 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 exactly 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: Construct an MST (using e.g. Kruskal's algorithm) of the subgraph induced by the tamers. Then for each lion i add the minimum length plank (i, j), where j is a tamer. The resulting tree of planks is the solution.

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

solution

First I argue that in the optimal solution, for every lion, the (unique) plank connecting that lion to some tamer must be the plank of minimum length between that lion and some tamer. For let T be a solution satisfying the given conditions, but not necessarily of minimum total length, such that there exists a lion i and tamers j, j' such that i is connected to j in T but $l(i, j') < l(i, j)$. Define

T' by replacing plank (i, j) with plank (i, j'). Then it's easy to see that T' continues to satisfy all three conditions; for the second one, the new path from i to j consists of plank (i, j') followed by the preexisting path from j' to j.

All that remains is to determine which tamers are connected to each other. By a similar argument, given any candidate solution satisfying the above conditions, we can decrease the total length (without violating any of the conditions) by replacing the induced subgraph of the candidate solution on the tamers with the MST on the tamers.

Running Kruskal's algorithm takes $O((n-m)^2 \log(n-m))$ time, and connecting lions to tamers takes $O(m(n-m))$ time, so the overall runtime is $O((n-m)^2 \log(n-m) + m(n-m))$.

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 $i_k \in \{\text{holder,hanger}\}$ 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) [7 Points] Design a divide and conquer algorithm to determine if the rope is safe given $f_1, \ldots, f_{m+n}$ and $i_1, \ldots, i_{m+n}$.

solution

In the solutions I sent out for the tutorial, I misread the question to mean that the force should be in the range $[-F, F]$, when actually it just needs to be at most F. First I describe a dynamic programming solution, and then a slightly less efficient divide-and-conquer solution, either of which should get full credit. Assume the hanger forces are positive and the holder forces are negative, and that the acrobats are already sorted from left to right.

dynamic programming

For an acrobat k let $s_k = \max_{j \in [k]}(f_j + f_{j+1} + \ldots + f_k)$, i.e. the maximum sum of the forces of any contiguous subsequence of acrobats ending with acrobat k. Also let $s_0 = 0$ (assume acrobats are numbered starting at 1). Observe that for $k \geq 1$,

$$s_k = \max\left(f_k, \max_{j \in [k-1]}(f_j + \ldots + f_{k-1} + f_k)\right) = f_k + \max(s_{k-1}, 0).$$

The algorithm is as follows: for k from 1 to $m + n$, compute s_k using the recurrence $s_k = f_k + \max(s_{k-1}, 0)$, and then return whether $\max_k s_k \leq F$.

greedy

Recursively check whether the left half of the rope contains any segments of force greater than F. Do the same for the right half of the rope. Now it suffices to check the safety of intervals that cross the midpoint *mdpt* of the rope. The maximum force on any interval that crosses *mdpt* is the sum of a) the maximum force on an interval that ends at $mdpt - 1$, and b) the maximum force on any interval that starts at *mdpt*. Since the computations of (a) and (b) are symmetric, it suffices to describe how to compute (b): for j from *mdpt* to $m+n$, compute the sum t_j of forces from *mdpt* to j using the recurrence $t_j = t_{j-1} + f_j$, and output $\max_j t_j$ for the result of this sub-computation.

(b) [7 Points] Prove that your algorithm is correct, and analyze its worst-case time complexity.

solution: The proof of correctness is mixed in with the above discussion. The dynamic programming solution takes $O(m+n)$ time. The greedy solution takes $O((m+n)\log(m+n))$ time, by the recurrence $T(m+n) = 2 \cdot T((m+n)/2) + O(m+n)$.

(c) [4 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

Consider the graph in which acrobats are vertices, and two acrobats are adjacent if they know each other. Let $\deg(a)$ be the degree of an acrobat a in this graph (i.e. the number of acrobats a knows), and let $d = \max_a \deg(a)$.

The algorithm is as follows. Let C be an initially empty set of colors. For each acrobat a (in some order), if there is a color in C that hasn't already been assigned to one of a's neighbors, then assign one such color to a. Else, add a new color to C, and assign that color to a.

Clearly this algorithm produces a valid coloring. Let x be the last new color created, and let a be the first acrobat to which the color x is assigned. By the nature of the algorithm, this implies that each color created before x had already been assigned to one of a's neighbors. Therefore there are at most as many colors preceding x as there are neighbors of a. Since x is by definition the last new color, this implies there are at most $\deg(a) + 1 \leq d + 1$ total colors used.

(d) [2 Points] Provide a counterexample to show that the algorithm from (f) does not always produce costume assignments with the smallest possible number of colors.
solution: https://cs.stackexchange.com/questions/42973/counter-example-to-graph-coloring-heuristic-using-bfs