1. Suppose your boss hands you a target sheet with n circles drawn on it. Describe and analyze an efficient algorithm to find a properly nested subset of these circles that maximizes the sum of the circle areas. You cannot *move* the circles; you must keep them exactly where your boss has drawn them.

   The input to your algorithm consists of three arrays $R[1..n]$, $X[1..n]$, and $Y[1..n]$, specifying the radius of each circle and the x- and y-coordinates of its center. The output is the sum of the circle areas in the best target.

Solution: We reduce the problem to finding the longest weighted path in a dag $G = (V, E)$ defined as follows:

- V contains a vertex for each circle, identified by an integer between 1 and n, and two artificial vertices s and t.
- E contains three types of weighted directed edges:
  - a *source* edge $s \to j$ with weight equal to the area of circle j, for every circle index i.
  - a *containment* edge $i \to j$ with weight equal to the area of circle j, for every pair of indices i and j such that circle i contains circle j,
  - a *sink* edge $i \to t$ with weight 0, for every circle index i.

Altogether G has n vertices and at most $\binom{n}{2} = O(n^2)$ edges. We can construct G by brute force in $O(n^2)$ time, by testing every pair of circles.

Every directed path in G from s to t corresponds to a properly nested subset of the given circles; moreover, the sum of the weights of the edges on the path is equal to the value of the corresponding target. Thus, we are looking for the longest weighted path in G from s to t. This problem can be solved using the LongestPath algorithm in the textbook (section 6.4) in $O(V + E) = \boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 10 points: Standard graph reduction rubric (with implicit brute force construction). This is not the only correct solution; this is not the only correct presentation of this algorithm.

Max 8 points for a correct $O(n^3)$-time algorithm (for example, separately computing a longest path from s to each index j); max 6 points for a correct $O(n^4)$-time solution (for example, separately computing a longest path from each index i to each index j); scale partial credit.

This is not the fastest algorithm for this problem, but it is the fastest algorithm I can explain in less than ten pages. In particular, this problem can be solved in $O(n^{4/3} \operatorname{polylog} n)$ time using sophisticated geometric data structures. For details, ask anyone currently taking Timothy Chan's section of CS 598; the same problem appears in their Homework 3.

2. Describe and analyze an algorithm to compute the maximum amount of candy that a single child can obtain in a walk through Cheery Hells, starting at the entrance node s. The input to your algorithm is the directed graph G, along with a non-negative integer $c(v)$ for each vertex describing the amount of candy that house gives each first-time visitor.

[Hint: Think about two special cases first: (1) Cheery Hells is strongly connected, and (2) Cheery Hells is acyclic. Solving only these two special cases is worth half credit.]

Solution: If G is strongly connected, then we can collect candy from *every* house in Cheery Hells, so our algorithm should report $\sum_v c(v)$.

If G is a directed acyclic graph, then we are looking for a *path* in G, from s to some other vertex, that maximizes the sum of $c(v)$ over all nodes v in the path. Add an artificial sink t, with edges $v \to t$ from every house v, and then give every edge $v \to w$ a weight by setting $\ell(v \to w) = c(v)$. With this change, the length of any path is the sum of its edge weights, so we are looking for the longest path from s to t in the modified graph. We can compute this longest path in $\boldsymbol{O(V+E)}$ ***time*** using the LongestPath algorithm in section 6.4 of the textbook.

Now let's consider the general setting. As in the previous special case, modify the input graph G by adding an artificial sink t and edges $v \to t$ from every house v. We are now looking for a maximum-value walk in G from s to t. If an optimal walk enters a particular strong component of G, it can (and therefore must) visits *every* vertex in that strong component.

Let $H = (V', E')$ denote the strong component graph of G; we can compute H in $O(V+E)$ time using either of the linear-time algorithms in section 6.6 of the textbook. Let $[v]$ denote the strong component of any vertex v in G; each set $[v]$ is a vertex of H. Define the weight of each edge $[v] \to [w]$ of H to be the sum of $c(x)$ over all vertices $x \in [v]$ (that is, all vertices x that are strongly connected to v).

Now we need to compute the length of the longest path in H from $[s]$ to $[t]$. We can compute this longest path in $O(V'+E') = \boldsymbol{O(V+E)}$ ***time*** using the LongestPath algorithm in section 6.4 of the textbook. ■

Rubric: 10 points: standard graph reduction rubric, including 5 points for construction of the strong components graph. This is not the only correct presentation of this algorithm.

3. Rick has given Morty a detailed map of the Clackspire Labyrinth, which consists of a directed graph $G = (V, E)$ with non-negative edge weights (indicating distance in flinks), along with two disjoint subsets $P \subset V$ and $F \subset V$, indicating the plumbus storage units and fleebholes, respectively. Morty needs to identify a start vertex s, a plumbus storage unit $p \in P$, and a fleebhole $f \in F$, such that the shortest-path distance from p to f is at most 137 flinks long, and the length of the shortest walk $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$ is as short as possible.

Describe and analyze an algo(burp)rithm to so(burp)olve Morty's problem. You can assume that it is in fact possible for Morty to succeed.

Solution: The length of a closed walk does not depend on the choice of starting vertex. Thus, we can assume without loss of generality that Morty will teleport into the Labyrinth at a plumbus storage unit p, walk along a shortest path to a fleebhole f, and then walk back to p to teleport out. In other words, we can assume that $s = p$.

We start by computing the shortest-path distances between every pair of vertices in G using the Floyd-Warshall algorithm. Then we consider each plumbus storage unit $p \in P$ and fleebhole $f \in F$ by brute force. If the distance from p to f is at most 137 flinks, then we compare the shortest walk $p \rightsquigarrow f \rightsquigarrow p$ to the best legal walk found so far.

```
InterdimensionalCable(G, P, F):
  dist[·,·] ← FloydWarshall(G)
  best ← ∞
  for all vertices p ∈ P
      for all vertices f ∈ F
          if dist[p, f] ≤ 137
              best ← min{best, dist[p, f] + dist[f, p]}
  return best
```

The overall algorithm runs in $\boldsymbol{O(V^3)}$ ***time***; the running time is dominated by the initial all-pairs shortest-path computation. ■

Rubric: 10 points: 5 for reduction to all-pairs shortest paths (or lots of single-source shortest paths) + 3 for other algorithm details + 2 for running time. This is not the only correct solution. No penalty for time bounds like $O(VE \log V + V^2)$ that are near-cubic in the worst case.

If the input graph is sparse, and/or there are relatively few plumbus storage units and fleebholes, we can reduce the time to $O((P + F)E \log V + PF) = O(VE \log V + V^2)$ by computing shortest paths from each source vertex in $P \cup F$ using Dijkstra's algorithm, instead of running Floyd-Warshall. (We can remove the final V^2 term from the running time by considering each strong component of G separately, because Morty's walk stays entirely within one strong component.) But we can't avoid the all-pairs shortest path computation, so for dense graphs with lots of plumbuses and fleebs, $O(V^3)$ is (essentially) the best we can hope for.