CS/ECE 374 B: Spring 2020
Solution to the Final Version: 2.1

1 (16 PTS.) Regular languages

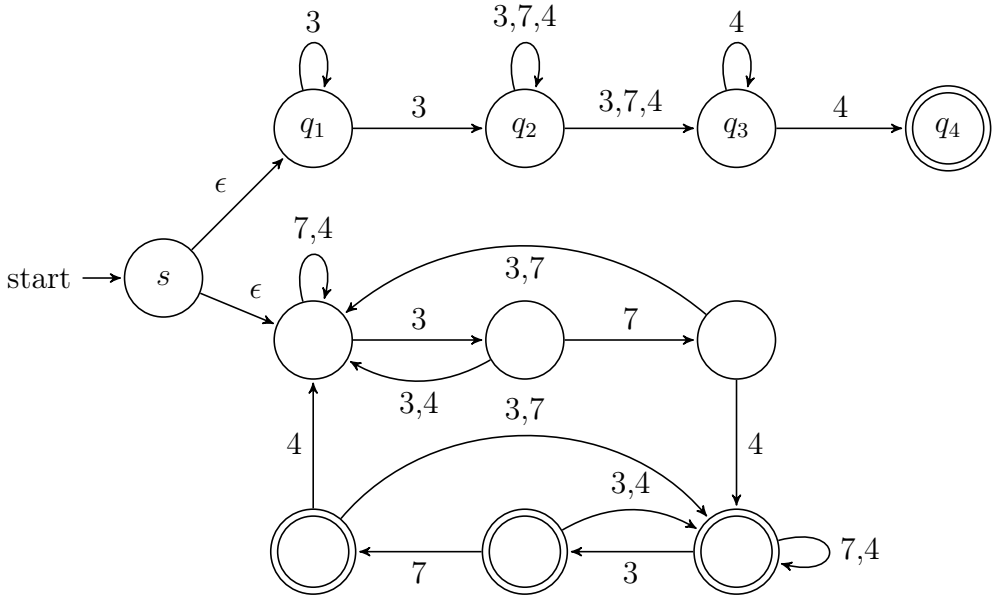
Consider the following languages over the alphabet $\Sigma = \{3, 7, 4\}$:

$L_1 = \{3^m(3|7|4)^n 4^\ell \mid m > 0, n > 0, \ell > 0\}$
$L_2 =$ all strings that contain the substring "374" an odd number of times.
$L_3 =$ an arbitrary regular language recognized by the DFA: $D' = (\Sigma, Q', s', A', \delta')$.

1.A. (8 PTS.) Draw an NFA that accepts the language $L = L_1 \cup L_2$.

Solution:

Using NFA epsilon transitions, we can step to the top portion to accept strings in L_1, and the bottom portion to accept strings in L_2, resulting in an NFA that accepts the union. For L_1, we need one state for 3^+ (at least one 3), a next state for $(3|7|4)^+$ (at least once), and finally a next state for 4^+ (at least one 4). For L_2, we need to keep track of the odd-even parity of how many times we encounter the sequence "374".



Rubric: 8 points =

+ 2 for a formal, complete, and unambiguous description of a DFA or NFA
  No points for the rest of the problem if this is missing.
+ 5 for a correct NFA
  −1 for a single mistake in the description (for example a typo)
+ 1 for a *brief* English justification. We explicitly do *not* want a formal proof of correctness, but we do want one or two sentences explaining how the NFA works.

1.B. (8 PTS.) Construct an NFA that accepts the language $L = L_1 \backslash L_3$.

Solution:

Since D' is a DFA, we can recognize the complement language simply by taking the complement of its accepting states. An NFA to recognize L_1 using states q_1, q_2, q_3, q_4 is already given in the answer to part A above. The idea is to recognize the intersection of $L_1 \cap \overline{L_3}$ by giving a product construction. A formal description is below:

$$Q = \{(q_i, q') \mid i \in [1,2,3,4], q \in Q'\}$$

$$\begin{aligned}
\delta((q_1, q'), 3) &= \{(q_1, \delta'(q', 3)), (q_2, \delta'(q', 3))\} \\
\delta((q_2, q'), 3) &= \{(q_2, \delta'(q', 3)), (q_3, \delta'(q', 3))\} \\
\delta((q_2, q'), 7) &= \{(q_2, \delta'(q', 7)), (q_3, \delta'(q', 7))\} \\
\delta((q_2, q'), 4) &= \{(q_2, \delta'(q', 4)), (q_3, \delta'(q', 4))\} \\
\delta((q_3, q'), 4) &= \{(q_3, \delta'(q', 4)), (q_4, \delta'(q', 4))\}
\end{aligned}$$

(Omitted cases are transitions to empty set)

$$s = (q_1, s')$$

$$A = \{(q_4, q') \mid q' \notin A'\}$$

Rubric: 8 points =

+ 2 for a formal, complete, and unambiguous description of a DFA or NFA
  No points for the rest of the problem if this is missing.
+ 5 for a correct NFA
  −1 for a single mistake in the description (for example a typo)
+ 1 for a *brief* English justification. We explicitly do *not* want a formal proof of correctness, but we do want one or two sentences explaining how the NFA works.

2 (16 PTS.) GREEDY

Consider the following problem: you are given n tasks $T_1, \cdots, T_n$. Each task has a deadline d_i and takes t_i time to complete. For simplicity, assume all deadlines d_i and times t_i are unique, i.e. no two tasks have the same deadline or take the same time. For every task you finish late, you pay a penalty of $f_i - d_i$ where f_i is the time you finish task T_i. For every task you finish early, you get a profit of $d_i - f_i$. Your goal is to maximize your profit, i.e., maximize:

$$\sum_{i}^{n} (d_i - f_i)$$

.

Describe an efficient algorithm to schedule your tasks in order to maximize your profit. Analyze and prove the correctness of your algorithm.

Solution:

This is very similar to the job scheduling algorithm, you might have seen in the class notes, that aims to minimize the maximum latness. In that case, the optimal greedy strategy is Earliest Deadline First (EDF). However, here our goal is to minimize the sum of lateness (penalty) and maximize the sum of earliness (profit) in order to maximize the total profit.

The optimal greedy scheduling startegy here is Shortest Task First. Hence, we sort the tasks increasing order of the time it takes to complete each task and we schedule them in that order one after the other.

Running time. Our algorithm runs in $O(n \log n)$ which it the time it needs to sort the tasks according to t_i, the time to complete each task.

Correctness. We will prove that our greedy strategy is optimal via an exchange (swap) argument. Suppose, $O_1, O_2, \cdots, O_n$ is the optimal scheduling solution. Let $d'_1, \cdots, d'_n$ be the deadlines of these optimally scheduled tasks, and $t'_1, \cdots, t'_n$ be the time it takes to complete the tasks i.e. if $O_i = T_j$, then $d'_i = d_j$ and $t'_i = t_j$.

Assume for the sake of contradiction that there are two tasks O_i and O_j such that $i < j$ and $t'_i > t'_j$ i.e. O_i is scheduled before O_j in the optimal schedule even though it has a longer processing time. Given this assumption, it is easy to prove by contradiction that there are two adjacent tasks in the optimal schedule O_r and O_{r+1} such that $t'_r > t'_{r+1}$ and $i \leq r < j$.

Let s_r be the start time of O_r. The profit or penalty incurred on this task is then $d'_r - s_r - t'_r$. Similary, the profit or penalty incurred on O_{r+1} is $d'_{r+1} - s_r - t'_r - t'_{r+1}$.

We can now create a new schedule by swaping O_r and O_{r+1}. Hence, O_{r+1} ends at $s_r + t'_{r+1}$ and O_r ends at $s_r + t'_{r+1} + t'_r$ in the new schedule. Since this does not change the start time or finish time of any other tasks except for O_r and O_{r+1}, the difference between the new schedule and the optimal schedule is only due to O_r and O_{r+1}.

Optimal Schedule Profit: $P_{opt} = d'_r + d'_{r+1} - 2s_r - 2t'_r - t'_{r+1}$.

New Schedule: $P_{new} = d'_r + d'_{r+1} - 2s_r - t'_r - 2t'_{r+1}$

Since $t'_r > t'_{r+1}$, we have $P_{new} > P_{opt}$. Hence, the optimal schedule is not optimal which is a contradiction. Thus, the optimal schedule must follow a short task first strategy.

Alternative Proof. We can unroll the profit summation.

$$\begin{aligned} P_{opt} &= \sum_{i=1}^{n} (d'_i - f'_i) \\ &= \sum_{i=1}^{n} \left(d'_i - \sum_{j=1}^{i} t'_j \right) \\ &= \sum_{i=1}^{n} d'_i - \sum_{i=1}^{n} \sum_{j=1}^{i} t'_j \\ &= \sum_{i=1}^{n} d'_i - \sum_{i=1}^{n} (n-i+1) t'_j \\ &= \sum_{i=1}^{n} d'_i - nt'_1 - (n-1)t'_2 - (n-2)t'_3, \cdots, t'_n \end{aligned}$$

Clearly, to maximize the profit, we should schedule the shortest task first since the tasks scheduled first carry larger negative weight.

Rubric: 16 points =

- +5 points for correct Greedy Strategy of Shortest Task First
- +3 points for runtime analysis. −1 point for assuming input is already sorted.
- +8 points for proof of correctness.

3 (17 pts.) MST and deletions.

Let $G = (V, E)$ be an undirected graph with n vertices and $m = n\sqrt{n} = n^{1.5}$ edges, all with positive edge weights.

3.A. (5 pts.) Finding the MST: Which of the (Borůvka's, Prim's, Kruskal's) algorithms could correctly be used to find a Minimum Spanning Tree of G? Which would give the best asymptotic running time as a function of n?

Solution:

As the edges are all positive, we could safely use any of these algorithms. Prim's would result in $O(n^{1.5})$ running time using the best data structures, whereas Kruskal's and Borůvka's would give $O(n^{1.5} \log n)$ time, which is worse.

Rubric: 5 points =

- +2 points for saying you can use any.
- +3 points for chosing the correct algorithm.

3.B. (12 pts.) Handling a deletion: After finding a Minimum Spanning Tree T using the algorithm above, your real world application needs have changed. The n'th largest edge, $e = (u, v)$, has now been removed from the graph. That is, the removed edge has a smaller weight than n other edges.

Give an algorithm that computes a new minimum spanning tree for $G' = G \backslash \{e\}$ as fast as possible. Your algorithm can use G, e, and T as input. You may also assume that you have an array of the edges in G in sorted order by weight.

Solution:

Recomputing it directly, using Prim's algorithm, would result again in $O(n^{1.5} + n \log n) = O(n^{1.5})$ time. We can reuse the partial solution T to reduce this to $O(n \log n)$, a solid improvement.

Starting from T, remove any of the edges that are greater than the removed edge (this takes $O(n)$ time, since there are n edges in T).

The resulting forest, F, corresponds to a partial execution of Kruskal's algorithm where all but the last k edges have already been considered. Hence the remaining edges in F are also guaranteed to be present in the final result we are looking for.

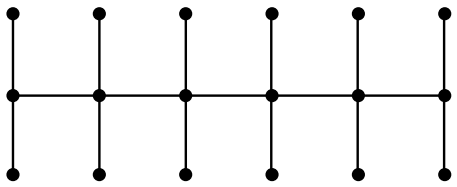
We can construct a new graph F' where the nodes are the connected components of F ($O(n)$ time to compute connected components using DFS), and the vertices are the $\Theta(n)$ edges larger than e. If there are multiple edges connecting the same two nodes in F', take the minimum. We can now run any of Boruvka, Prim, Kruskal's algorithm to find the MST of this smaller graph F'. The result will take $O(n \log n)$ time in any case.

Rubric: 12 points =

- +12 for correct $O(n \log n)$ algorithm with correct explanation
- Max of 6 points for algorithm that processes only the largest n edges with minor errors.
- Max of 2 points for any algorithm that runs in $O(|E|) = O(n^{1.5})$ or that uses the solution from Lab 12b. (Note: You might as well run Prim from scratch and get an $O(n^{1.5})$.

4 (17 PTS.) Centipedes.

A *centipede* is an undirected graph formed by a path of length k with two edges (legs) attached to each node on the path as shown in the below figure. Hence, the centipede graph has $3k$ vertices. The CENTIPEDE problem is the following: given an undirected graph $\mathsf{G} = (V, E)$ and an integer k, does G contain a *centipede* of $3k$ vertices as a subgraph? Prove that CENTIPEDE is NP-COMPLETE.



Solution:

The problem is in NP. We let the certificate be three ordered lists of length k. The first list is the main path and the other two lists form the legs. We can easily verify in polynomial time that the lists form a *centipede* by checking that in the first list any two consecutive vertices have an edge between them in G and checking that a vertex from the second or third list has an edge to a vertex in the first list of the same order (position in the list).

The problem is NP-HARD by reduction from HAMILTONIAN-PATH to CENTIPEDE. Given an instance of HAMILTONIAN-PATH, a graph G with n vertices $v_1, v_2, \cdots, v_n$, create a new graph G' by adding $2n$ vertices: $u_1, u_2, \cdots, u_n$ and $x_1, x_2, \cdots, x_n$. Then, add an edge (u_i, v_i) and (x_i, v_i) for $1 \leq i \leq n$. The reduction is polynomial time since we only added $2n$ of vertices and edges to the graph.

Claim 0.1. G *has a Hamiltonian path* $\iff$ G' *contains a* centipede *of* $3n$ *vertices as a subgraph.*

Proof: $\implies$ If G has a Hamiltonian path, then we can form a *centipede* in G' by taking the Hamiltonian path in G and the new edges and vertices we added by construction.

$\impliedby$ If G' has a $3n$ vertex subgraph that is a *centipede*, then this sub-graph uses all the vertices of G'. The newly added vertices $(u_1, u_2, \cdots, u_n, x_1, x_2, \cdots, x_n)$ can only be in the feet (legs) of the centipede sub-graph since they have degree 1 whereas vertices in the body (center path) of the centipede must have degree at least 3. Hence, any vertex in the body path of the centipede must be in G. Hence, the body of the centipede is a Hamiltonian path in G.

Rubric: 17 points =

- +4 points proving the problem is NP
- +7 points for correct reduction.
- +3 points for $\implies$ proof.
- +3 points for $\impliedby$ proof.

5 (17 pts.) Merchants of Dag'än.

A game of *Merchants of Dag'än* is defined by a map of n towns connected by $O(n)$ one way roads. The roads between the towns form a directed acyclic graph DAG, with a single source and a single sink. Your character starts at the source (Start) and must travel through towns to end at the sink (End). The character has an inventory that can hold up to a maximum of 7 items. There are two kinds of items: Corn and Brick.

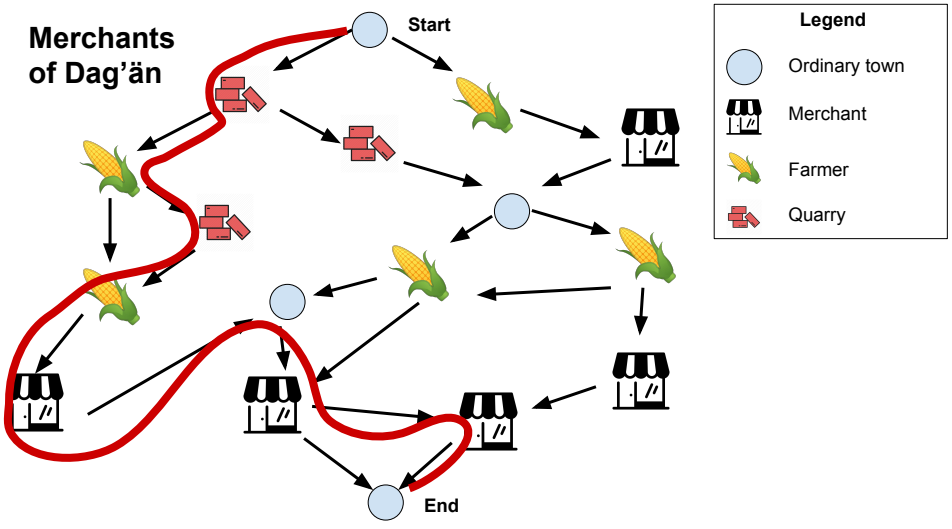
As you visit each town v, you *may* encounter *one* of the following kinds of traders to interact with:

1. $Quarry(v) = 1$: You can add 1 brick to your inventory (unless your inventory is full).
2. $Farmer(v) = 1$: You can trade 2 Brick for 3 Corn (unless this would overflow your inventory)
3. $Merchant(v) = 1$: You can trade a single item in your inventory for a Victory Point (VP). Brick and Corn are worth the same, 1 VP each. VPs do not take up any inventory space.

The traders are mutually exclusive since each town has at most one of them, so if $Farmer(v) = 1$ then $Quarry(v) = 0$, etc. At each town you may make at most a *single* trade before travelling onward to an adjacent town of your choice. If you have anything left over in your inventory after reaching the End town, it is wasted and doesn't count towards VPs. You can assume the Start and End towns are ordinary towns with no traders.

What is the maximum score of Victory Points you can achieve by playing this game for a given map? Give an algorithm, as fast as possible, using dynamic programming to answer this question for any map. Give a clear English description of the function you are trying to evaluate, and how to call your function to get the final answer, then provide a recursive formula for evaluating the function (including base cases) and a correct evaluation order. Analyze the running time as a function of n.

Example: Given the following map, the best achievable score is 3 VPs, achievable by following the red path. The path involves picking up a brick from each of the two quarries, declining to trade at the first farmer (as you'd only have one brick, not enough to trade), trading for 3 corn at the second farmer, and then trading one corn each at the three merchants.



Solution:

There are n nodes (and $O(n)$ total roads). At each town you have several choices: whether to trade or not, what item to trade (in case of merchant), and which outgoing town to visit next. We define the best score with a recursive function where the subproblems are the best achievable score starting at node (town) v with b number of bricks, and c number of corns. To simplify handling all the possible if-conditions, we'll define the score as $-\infty$ for cases where the inventory overflows or goes negative.

$$Score(v,b,c) = \begin{cases} -\infty & & b<0, c<0, b+c>7 \\ 0 & & v = End \\ \max_{v \to u} \begin{cases} \begin{cases} Score(u, b+1, c) \\ Score(u,b,c) \end{cases} & Brick(v) \\ \begin{cases} Score(u, b-2, c+3) \\ Score(u,b,c) \end{cases} & Farmer(v) \\ \begin{cases} 1 + Score(u, b-1, c) \\ 1 + Score(u, b, c-1) \\ Score(u,b,c) \end{cases} & Merchant(v) \\ Score(u,b,c) & Ordinary(v) \end{cases} & otherwise \end{cases}$$

To get the final answer, we call $Score(Start, 0, 0)$.

There are n nodes. If we allow each of b and c to range from -1 to 8 (which captures all the possible underflow and overflow conditions), then there are 100 possible inventory configurations to check - a large constant but a constant nonetheless. Hence there are a total of $O(n)$ subproblems. Although an individual subproblem may take up to $O(n)$ time to check, the total number of edges to consider is only $O(n)$ and each edge $u \to v$ is evaluated only once, hence running time should be only $O(n)$ in total.

The table must be computed in reverse topological ordering, from the End working the way back towards Start. If the graph is not given in topological ordering, we can compute this through a DFS for $O(n)$ time as well.

Rubric: Standard DP problem scaled to 17 points as follows:

- 10 points for a correct recurrence:
  - \+ 2 points for a clear **English** description of the function you are trying to evaluate. **Automatic zero if the English description is missing.**
  - \+ 1 point for stating how to call your function to get the final answer.
  - \+ 2 points for base case(s). -1 for one *minor* bug, like a typo or an off-by-one error.
  - \+ 5 points for recursive case(s). -1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 7 points for details of the dynamic programming algorithm
  - \+ 1 points for describing the memoization data structure
  - \+ 3 points for describing a correct evaluation order;
  - \+ 3 points for time analysis. 0 points for $O(n^2)$ analysis instead of $O(n)$.

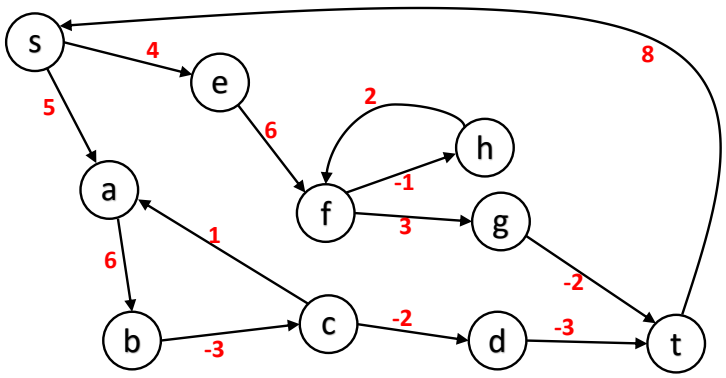
6 (17 pts.) Loopy Walks

Let $G = (V, E)$ be a weighted directed graph. Let $n = |V|$ and $m = |E|$. The weights can be negative but the graph has no negative cycles. Recall that an s-t walk in the graph is a sequence of vertices $v_0, v_1, v_2, ..., v_j$ where $v_0 = s, v_j = t$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < j$; the length of the walk is the sum of the weights of the j edges in the walk. A walk differs from a path in that it allows vertices and edges to be repeated.

An s-t walk is said to be k-*loopy* if along the walk from s to t, it reaches a node v and loops k times around a single simple cycle that starts and ends at v before continuing to t. Assume a cycle must contain more than one node, $k \geq 1$, and the walk contains no other cycles.

For example, in the below graph, the following s-t walks are said be 2-*loopy*:

- $s \to a \to b \to c \to a \to b \to c \to a \to b \to c \to d \to t$
- $s \to e \to f \to h \to f \to h \to f \to g \to t$
- $s \to a \to b \to c \to d \to t \to s \to a \to b \to c \to d \to t \to s \to a \to b \to c \to d \to t$



Given two nodes s and t and a parameter k, describe and analyze an efficient algorithm to find the shortest k-*loopy* walk from s to t in G.

For example, in the above graph, the shortest walk from s to t that is:

- 2-*loopy* is $s \to a \to b \to c \to a \to b \to c \to a \to b \to c \to d \to t$ and has length 11.
- 3-*loopy* is $s \to e \to f \to h \to f \to h \to f \to h \to f \to g \to t$ and has length 14.

Solution:

We start by finding all pairs shortest paths. This requires running Bellman-Ford n times and runs in $O(n^2 m)$. (Note that we can also use the Floyd-Warshall algorithm that runs in $O(n^3)$ for a faster algorithm.)

Let $d(u, v)$ be the length of the shortest path from node u to v.

Then for every vertex v, we find the length of the shortest simple cycle containing v. Let $c(v)$ be the shortest simple cycle containing v and let $\ell_c(v)$ be the length of the this cycle. We have:

$$\ell_c(v) = \min \{d(v, u) + w(u, v) \mid (u, v) \in E\}$$

where $w(u, v)$ is the weight of edge (u, v). This takes $O(n + m)$ since for each vertex, we touch each edge once.

We now can find the shortest *k-loopy* walk from s to t as:

$$\min \{d(s, v) + k \times \ell_c(v) + d(v, t) \mid v \in V\}$$

. This takes $O(n)$ time to check all vertices.

Thus, the overall algorithm runs in $O(n^2 m)$. (or $O(n^3)$ using more efficient algorithm).

<u>*Rubric:*</u> 17 points =

- +13 points correct algorithm
  - +8 points for correctly finding the shortest cycle.
  - +5 points for correctly finding the shortest *k-loopy* walk.
- +4 points for correct runtime analysis.
  - 0 points if algorithm is incorrect.
  - −1 point for assuming finding the shortest simple cycle takes $O(nm)$ rather than $O(m + n)$
- max of 5 points for any algorithms that uses DFS + backward edges to find the shortest cycle (It only finds some cycle, no necessarily the shortest).
- max of 4 points for any algorithm that:
  - uses Dijkstra instead of Bellam-Ford but otherwise is correct.
  - has an extra factor of n beyond $O(n^4)$.
- max of 0 points for any algorithm that:
  - runs in exponential time.
  - tries to find all possible simply cycles (there is an exponential number of simple cycles).
  - has an extra factor of n^2 or m beyond $O(n^4)$.
- proof of correctness is not required.
- no penalty for running BF n times rather than FW algorithm.

7 BONUS (10 PTS.) DECISIONS, DECISIONS.

This problem is **bonus**. You do not need to solve it. **"IDK"** policy does apply to this question. If you do not solve it, you do not get any points and you do not lose any points. This problem is not hard. We made it a bonus to ensure you can finish the exam in 3hrs.

Prove (via reduction) that the following language is undecidable.

$$\mathsf{AcceptOrBust} = \{\langle M\rangle \mid M \text{ does not reject any input}\}.$$

Your reduction *must* involve the SelfHalts problem, which is known to be undecidable.

$$\mathsf{SelfHalts} = \{\langle M\rangle \mid M \text{ halts on input } \langle M\rangle\}.$$

You can not use Rice's Theorem in solving this problem.

Solution:

Suppose there is an algorithm **DecideAcceptOrBust** that correctly decides the language AcceptOrBust. Then we can solve the SelfHalting problem as follows:

```
DECIDESELFHALT(⟨M⟩):
  Encode the following Turing machine M′:
    M′(x):
      run M on input ⟨M⟩
      return FALSE
  if DecideAcceptOrBust(⟨M′⟩)
    return FALSE
  else
    return TRUE
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M is self halting.
Then M' rejects all strings in Σ^*.
So **DecideAcceptOrBust** rejects the encoding $\langle M'\rangle$.
So **DecideSelfHalt** correctly accepts the encoding $\langle M\rangle$.

$\Longleftarrow$ Suppose M is not self halting.
Then M' diverges on *every* input string x.
In particular, M' does not reject any string.
So **DecideAcceptOrBust** accepts the encoding $\langle M'\rangle$.
So **DecideSelfHalt** correctly rejects the encoding $\langle M\rangle$.

In both cases, **DecideSelfHalt** is correct. But that's impossible, because SelfHalt is undecidable. We conclude that the algorithm **DecideAcceptOrBust** does not exist.

Rubric: 10 points =

- +5 for correct reduction.
- +5 for correct proof.
- No "IDK" points.