Solutions to Homework 5

This homework has 7 questions, for a total of 100 points and 0 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on 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 name(s). You must *write your own solution* for all problems and *may not use any other student's write-up*.

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't do last week's homework, choose a problem from it that looks challenging to you, and in a few sentences, explain the key ideas behind its solution in your own words.

Solution:

My solution to 4.c. of the previous homework was as follows:

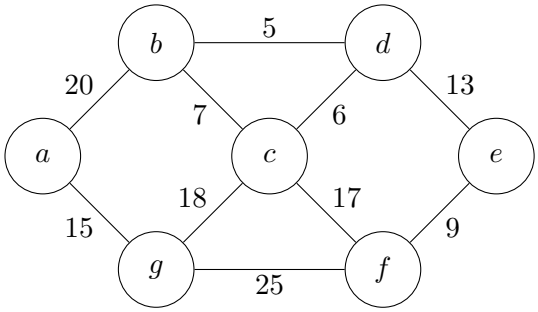
Input: Array $D[1 \ldots n]$ where $D[i]$ is the number of ducks in the ith pile.
Output: Maximum number of ducks Lambda can guarantee.

```
1: function ComputeLambdaDucks(D[1...n])
2:     Let memo[1...n][1...n] be an n x n 2D array table for memoization.
3:     for i = 1 to n do
4:         memo[i][i] = D[i] // base case
5:     for length = 2 to n do
6:         for i = 1 to n − length + 1 do
7:             j = i + length − 1
8:             memo[i][j] = max{ D[i] − memo[i + 1][j], D[j] − memo[i][j − 1] }
9:     total = D[1] + ··· + D[n]
10:    return (total + memo[1][n]) / 2
```

This was a good solution, but I did not include a correctness proof or runtime analysis. A correct correctness proof would have outlined how the algorithm fills the values of the DP table corrctly to land at $\frac{total+memo[1][n]}{2}$. A runtime analysis would look at the nested loop in the algorithm and find $O(n^2)$ runtime.

2. **MST mashup.**

(2 pts) (a) Suppose Kruskal's algorithm is run on the following edge-weighted graph.



List the *edge weights* of the edges chosen by Kruskal's algorithm, in the order chosen. Then, determine the weight of the minimum spanning tree of the graph.

Solution: Edges sorted by weight:

$$5,\ 6,\ 7,\ 9,\ 13,\ 15,\ 17,\ 18,\ 20,\ 25.$$

Kruskal's algorithm: ascending order, choosing non-cycle-forming edges. By weight order, the edges chosen are:

$$5,\ 6,\ 9,\ 13,\ 15,\ 18.$$

7 vertices = exactly 6 edges for the spanning tree. The total weight of the MST is:

$$5 + 6 + 9 + 13 + 15 + 18 \ = \ 66.$$

(4 pts) (b) Consider the following claim: if the graph's edge weights are distinct (all different), then the minimum spanning tree is unique.

Professor Υ presents you with the following bogus "proof" of this claim: *because edge weights are distinct, all choices in Kruskal's algorithm are completely determined—there are no "ties" between edge weights that the algorithm can choose how to break. So, the algorithm has only one possible output. Because Kruskal's algorithm is correct for the MST problem, its unique output must be the only minimum spanning tree in the graph.*

Briefly explain what is wrong with this "proof."

Solution: Although distinct edge weights means we can list edges in a unique order, this does not force a single MST. Even when the sorted order is fixed, there can be multiple edges that are safe to add at a given step, that do not form a cycle. Thus, the absence of ties in edge weights does not guarantee a single minimum spanning tree.

(8 pts) (c) Give a correct proof for the claim in the previous part.

Hint: Consider two distinct spanning trees T_1, T_2 and prove via an exchange argument that one of them does not have minimum weight.

Solution: Let G be a connected graph with distinct edge weights. Suppose, for contradiction, that there are two different minimum spanning trees, T_1 and T_2. Because they differ, there must be at least one edge e that lies in T_2 but not in T_1. Adding e to T_1 forms a cycle C, because T_1 is an MST. Pick a distinct edge in C, and call it f.

If $f \neq e$, then f must lie in T_1. Because f is strictly heavier than e (because e is in T_2 and T_2 is an MST), we can remove f from $T_1 \cup \{e\}$ to obtain another spanning tree whose weight is strictly less than the weight of T_1. This contradicts the definition of MST for T_1.

Therefore, no two distinct minimum spanning trees can exist in a graph with unique edge weights. Hence, the MST is unique.

3. **Greedy hydration stations.**

Kaitlyn loves to go hiking in various state parks across Michigan. Interested in maximizing her hiking experience, she plans her daily water intake carefully for her treks through different trails. Her water bottle can hold enough water to hike for a certain distance before needing a refill. Each trail has a hydration station where her bottle can be refilled: the current water is dumped and replaced with fresh water. (Any remaining water in the bottle will be discarded during refilling.) The goal is to visit the trails in a specified order, while minimizing the number of water refills.

We model this problem as follows. Kaitlyn can hike a distance up to R on a single bottle of water. There are n trails to visit in sequence. For $i = 1, \ldots, n$, the distance from the $(i-1)$st trail (or from the starting point, when $i = 1$) to the ith trail is $d_i \leq R$. Given R and an array $D = [d_1, \ldots, d_n]$ as input, the goal is to find a *smallest* set $S \subseteq \{1, \ldots, n\}$ of trails (specified by their indices) where refilling should occur, so that Kaitlyn can reach all n trails without running out of water at any point. She starts out with a full water bottle, and no refilling is needed at the final trail.

(6 pts) (a) Give a greedy algorithm (including pseudocode) that solves this problem. Defer a correctness argument to the next part, but provide everything else involved in "giving an algorithm" here.

Solution: Starting with a full water bottle, at each trail (except the final one) subtract distance traveled. If the remaining water is not enough to cover the distance to the next trail, we refill at the current trail. This guarantees that we minimize the number of refills while ensuring that each leg of the journey is traversable.

Input: Array $D[1 \dots n]$, where $D[i]$ is the distance from the $(i-1)$th trail (or the starting point for $i = 1$) to the ith trail; number R, the maximum distance Kaitlyn can hike on a full bottle.
Output: Set $S \subseteq \{1, \dots, n-1\}$ of trails at which a refill should occur.

```
1: function GreedyHydration(D[1...n], R)
2:     S = ∅
3:     water = R // start with a full water bottle
4:     for i = 1 to n − 1 do
5:         water = water − D[i]
6:         if water < D[i + 1] then
7:             S = S ∪ {i}
8:             water = R // refill at the ith trail
9:     return S
```

All operations in the loop are $O(1)$, and the loop runs $n-1$ times, so the runtime is $O(n)$.

(12 pts) (b) Suppose OPT is some arbitrary optimal solution and ALG $= \{i_1, \dots, i_k\}$ are the indices of trails chosen by your algorithm. If ALG = OPT, correctness is immediate. Otherwise, let i_{diff} be the first trail in ALG that is not in OPT. Now, complete the following steps:

(i) Describe another optimal solution OPT$'$ that contains all of $i_1, \dots, i_{\text{diff}}$ by modifying OPT, exchanging out some suitable trail j for i_{diff} instead.

(ii) Explain why OPT$'$ remains valid (meets all constraints) and optimal ($|\text{OPT}'| = |\text{OPT}|$).

(iii) Briefly explain why Item (ii) implies that your algorithm is correct.

Solution:

(i) If ALG = OPT, then the algorithm is optimal. Otherwise, let i_{diff} be the first index in ALG that is not in OPT.
Let j be the first refill station in OPT occurring after the last common refill with ALG, ($j = \text{OPT}\{i_{diff}\}$).

$$\text{OPT}' = (\text{OPT} \setminus \{j\}) \cup \{i_{\text{diff}}\}.$$

OPT$'$ contains all indices $i_1, \dots, i_{\text{diff}}$.

(ii) Upon arriving at i_{diff}, the remaining water is insufficient to cover the next segment. Thus, refilling at i_{diff} is necessary. Since any remaining water at j would be without use in the event of a refill, replacing j with i_{diff} still provides a full bottle at the right time. Since this exchange involves only one station and does not change the total number of refills, $|\text{OPT}'| = |\text{OPT}|$, so OPT$'$ is optimal.

(iii) Since any optimal solution OPT can be transformed via ALG exchanges, it must be the case that ALG uses no more refills than OPT. Therefore, the greedy algorithm ALG is correct.

(4 pts) (c) Now, suppose that the price of refilling water may differ from trail to trail. For example, filling a water bottle might cost \$8 at the first trail, \$3 at the second trail, etc. So, the input now also includes an array $C = [c_1, \ldots, c_n]$, where c_i is the cost of refilling at the ith trail. The goal now is to complete the tour while *minimizing the total cost* of the water that is purchased along the way.

Kaitlyn claims that we should use the same greedy algorithm that you proposed above to solve this problem. Determine whether this claim is true or not. If it is, give a proof. Otherwise, provide a small counterexample: give a specific input, the output of your greedy algorithm on that input, and a better solution for that input; also, *very briefly* explain where your proof from the previous part fails for this problem.

Solution: Kaitlyn's claim is false. The greedy algorithm that refills only when necessary does not always minimize the total cost when refill prices vary.

Let $R = 10$, $n = 3$, and set $D = [6, 4, 6]$ with refill costs $C = [3, 8, 5]$

- **Greedy Algorithm:**
  - Start with a full bottle (10 units). Travel to trail 1 uses 6, leaving 4 units.
  - At trail 1, 4 units is exactly enough for the next leg ($D[2] = 4$), so no refill is performed.
  - Travel to trail 2 uses 4, leaving 0 units.
  - At trail 2, since $0 < D[3] = 6$, the algorithm refills at trail 2 at cost \$8.

  Total cost = \$8.

- **Better Alternative:**
  - Start with a full bottle (10 units). Travel to trail 1 uses 6, leaving 4 units.
  - Refill at trail 1 (even though not necessary for reaching trail 2) at cost \$3, resetting water to 10.
  - Travel to trail 2 uses 4, leaving 6 units, which is exactly enough to reach trail 3.

  Total cost = \$3.

The greedy algorithm does not produce an optimal solution.

The exchange argument relies on refilling only when the current water is insufficient to cover the next leg, because we don't factor refill cost. When refill costs vary, it can be advantageous to refill earlier at a cheaper station even if the current water is enough for the immediate next segment. Adding cost invalidates the necessity condition used in the original proof.

4. **Greedy knapsack filling.**

Recall the knapsack problem (sometimes called the "0-1" knapsack problem): you are given n items, where the ith item has weight w_i and value v_i (both non-negative integers), and a non-negative weight capacity W of the knapsack. An optimal solution is a subset $S \subseteq \{1, 2, \ldots, n\}$ of the items having maximum total value

$$\sum_{i \in S} v_i \ ,$$

under the constraint that the total weight is at most the knapsack capacity, i.e.,

$$\sum_{i \in S} w_i \leq W \ .$$

In this problem, we introduce the *fractional* variant of the knapsack problem, where one may take any fraction $p_i \in [0, 1]$ of any item i. Naturally, a p_i-fraction of item i weighs $p_i \cdot w_i$, and has value $p_i \cdot v_i$. The goal is to maximize the total value of the selected fractional items.

(2 pts) (a) Briefly explain why the optimal value for the fractional knapsack problem is *at least as large* as that of the original 0-1 knapsack problem (for the same weights, values, and knapsack capacity).

> **Solution:** Any valid solution to the original 0-1 knapsack problem can be used in the fractional knapsack problem by taking $p_i = 1$ for every item chosen and $p_i = 0$ otherwise. Since the fractional knapsack problem allows taking any fraction of an item, it has a larger set of feasible solutions. Thus, the optimal value for the fractional knapsack is at least as large as that for the 0-1 knapsack.

(4 pts) (b) Consider the following two greedy algorithms for the fractional knapsack problem:

- ValueGreedy: While there is still some unused knapsack capacity, choose an item having the *largest value* (among those not considered yet), and add the largest possible fraction of that item (up to the entire item) that will fit within the remaining knapsack capacity.
- WeightGreedy: While there is still some unused knapsack capacity, choose an item having the *smallest weight* (among those not considered yet), and add the largest possible fraction of that item (up to the entire item) that will fit within the remaining knapsack capacity.

Suppose we run *both* algorithms ValueGreedy and WeightGreedy, and output a best one of their two outputs. Is this a correct algorithm for the fractional knapsack problem? If so, prove it. Otherwise, give an input for which this algorithm's output is not optimal, and show why it is not.

> **Solution:** Neither ValueGreedy nor WeightGreedy (nor the best of the two) always produces an optimal solution for the fractional knapsack problem, so the algorithm is not correct.

Let the knapsack capacity be $W = 10$, and consider four items given by:

Item	w_i	v_i	v_i/w_i
A	6	18	3
B	5	17	3.4
C	3	6	2
D	2	5	2.5

Sorting items by their value-to-weight ratio, the best order is to consider item B first, then item A. An optimal solution is to take item B entirely (weight 5, value 17), and with remaining capacity 5, take $\frac{5}{6}$ of item A ($18 \cdot \frac{5}{6} = 15$). The total value is $17+15 = 32$.

ValueGreedy Algorithm picks the item with the highest value first, item A (value 18, weight 6, remaining capacity: $10 - 6 = 4$), and then, among the remaining items, it chooses the next highest value item, item B, with a fraction of $\frac{4}{5}$, yielding value $17 \cdot \frac{4}{5} = 13.6$, resulting in total value $18 + 13.6 = 31.6$.

WeightGreedy Algorithm picks the item with the smallest weight first, item D (weight 2, value 5, remaining capacity: $10 - 2 = 8$), then picks item C (weight 3, value 6, remaining capacity: $8 - 3 = 5$), and then, picks item B (weight 5, value 17, remaining capacity: 0) with a total value of $5 + 6 + 17 = 28$.

$\textsc{Max}(\textsc{ValueGreedy}, \textsc{WeightGreedy}) = 31.6$, which is less than the optimal value of 32.

Neither algorithm takes advantage of the value-to-weight ratio, which is the key difference in the fractional knapsack problem. ValueGreedy may choose an item with high absolute value even if its ratio is lower, and WeightGreedy may select a light item with a poor ratio. Thus, taking the better result of the two algorithms does not guarantee an optimal solution.

(10 pts) (c) Consider the following greedy algorithm for the fractional knapsack problem:

- RelativelyGreedy: While there is still some unused knapsack capacity, choose an item having the largest *relative value* v_i/w_i (among those not considered yet), and add the largest possible fraction of that item (up to the full item) that will fit within the remaining knapsack capacity.

In this part, you will prove that the RelativelyGreedy algorithm is a correct algorithm for the fractional knapsack problem. This is notable because we used a more complicated dynamic programming algorithm to solve 0-1 Knapsack, but the fractional variant admits a much simpler greedy algorithm!

The setup for your correctness proof is as follows. Without loss of generality, suppose that the items are in sorted order by relative value $r_i = v_i/w_i$, from largest to smallest (the algorithm considers them in this order). Let $\text{ALG} = p_1, p_2, \ldots, p_n$ be the fractions of items $1, 2, \ldots, n$ chosen by the algorithm, respectively. (So, $p_i = 1$ if the algorithm chooses all of item i, and $p_i = 0$ if it doesn't choose any of item i.)

Let OPT be an arbitrary optimal solution. If ALG = OPT, then ALG is optimal, as needed. So suppose that ALG $\neq$ OPT, and consider the smallest index j such that OPT does not have *exactly* a p_j-fraction of item j.

You will construct another optimal solution OPT$'$ whose fractions of the first j items are exactly $p_1, p_2, \ldots, p_j$ by modifying OPT, using an exchange argument.

Describe which fraction(s) of item(s) to exchange, and prove that OPT$'$ meets the stated requirements; this should include a proof that OPT$'$ is a *valid* solution.

Solution: j is the smallest index such that the fraction of item j in OPT differs from p_j, so, say the fraction at OPT$\{j\}$ is q_j. Because the algorithm takes as much as possible from each item until the knapsack is full, we must have $q_j < p_j$.

Because OPT uses the full capacity, there exists some item with index $k > j$ for which $q_k > 0$. Choose the smallest such index k. We now perform an exchange between items j and k to "shift" capacity from item k (lower relative value) to item j (higher relative value).
Say we increase the fractional weight taken from item j. An increase of ϵ in the fraction of item j adds weight $\epsilon\, w_j$. Thus, we decrease the fraction of item k by $\frac{w_j \epsilon}{w_k}$.

Choose $\epsilon = \min\left\{p_j - q_j,\ \frac{w_k}{w_j}\, q_k\right\}$, so that we do not remove more than the available fraction of item k.

Define a new solution OPT$'$ as follows:

$$q'_j = q_j + \epsilon, \quad q'_k = q_k - \frac{w_j \epsilon}{w_k}, \quad \text{and } q'_i = q_i \text{ for all } i \neq j, k.$$

The weight added to item j is $\epsilon\, w_j$,
The weight removed from item k is $\frac{w_j \epsilon}{w_k}\, w_k = \epsilon\, w_j$.
Thus, the total weight remains unchanged, and OPT$'$ is a valid solution.

Increasing item j by ϵ adds value $\epsilon\, v_j = \epsilon\, r_j\, w_j$, and decreasing item k by $\frac{w_j \epsilon}{w_k}$ subtracts value $\frac{w_j \epsilon}{w_k}\, v_k = \epsilon\, r_k\, w_j$.
Since $r_j \geq r_k$, the net change in value is nonnegative. But OPT was optimal, so the value cannot strictly increase. So, the exchange preserves optimality.

Repeating this exchange process until the fraction for item j is increased to p_j, obtain a new optimal solution OPT$'$ that agrees with ALG on item j. Items $1, 2, \ldots, j-1$ already match between OPT and ALG, so OPT$'$ has fractions $p_1, p_2, \ldots, p_j$ for the first j items. RelativelyGreedy produces an optimal solution.

5. **DFAs to languages.**

(4 pts) (a) Consider the DFA over alphabet $\Sigma = \{a, b\}$ with start state q_0 and accept state q_2 defined by the transition functions below:

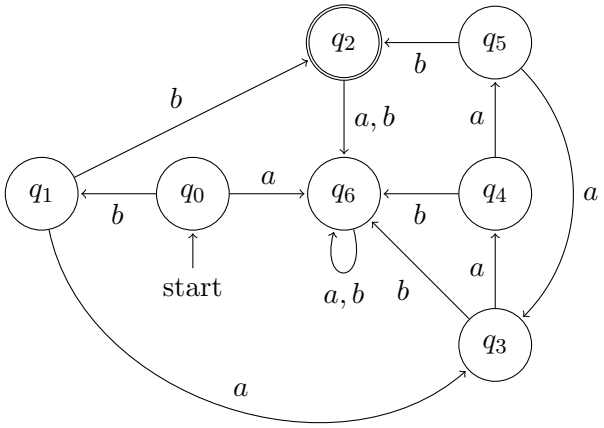
Current state	Input	Next state
q_0	a	q_1
q_0	b	q_0
q_1	a	q_2
q_1	b	q_1
q_2	a	q_2
q_2	b	q_2

Describe the language **both** in English and as a regular expression (using the "string notation" from class) decided by the DFA. No justification is required for either of these.

Solution: All strings over $\{a, b\}$ with at least two occurrences of a.

$$b^* \, a \, b^* \, a \, (a \mid b)^*.$$

(4 pts) (b) Consider the DFA over alphabet $\Sigma = \{a, b\}$ with start state q_0 and accept state q_2 defined in the diagram below:



Describe the language **both** in English and as a regular expression (using the "string notation" from class) decided by the DFA. No justification is required for either of these.

Solution: All strings that begin with b, then have a multiple of 3 sequential copies of a (possibly zero), and then end with a single b.

$$b \, (a^3)^* \, b.$$

6. **Languages to DFAs.**

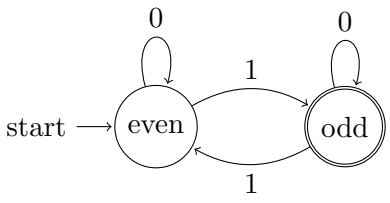
For each of the following languages, give a DFA that decides it. You may represent a DFA as a state-transition function or as a diagram, whichever you prefer. You do not need to justify your answer, just make sure you

- clearly indicate the start state or label it with a "start →",
- clearly indicate the final state(s) or denote it with a double circle, and
- have transition function for *every* state for *every* character in the alphabet as input.

If you're writing your solution in LaTeX, you might want to check out this tool to generate `tikzpicture` script for your DFA.

(3 pts) (a) The language of binary strings (over alphabet $\{0,1\}$) that have an odd number of 1's.

Solution:

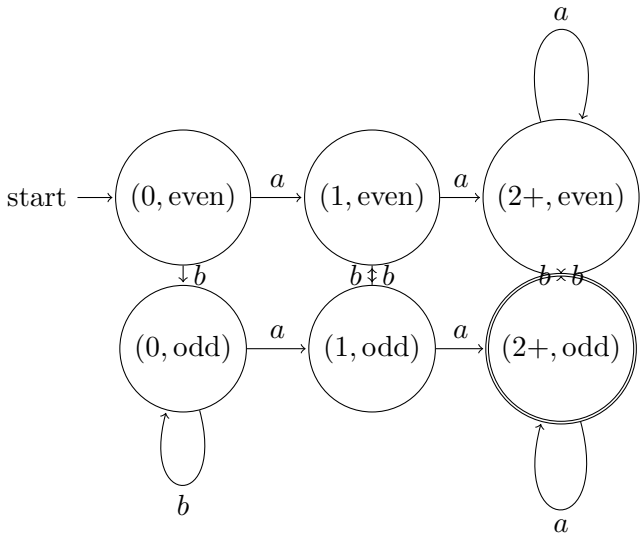


(5 pts) (b) The following language over alphabet $\Sigma = \{a, b\}$:

$$L = \{s \in \Sigma^* : s \text{ contains at least two } a\text{'s } \textbf{and} \text{ an odd number of } b\text{'s.}\}.$$

Hint: Part a of Question 5 and Part a of Question 6 may give you some ideas.

Solution:

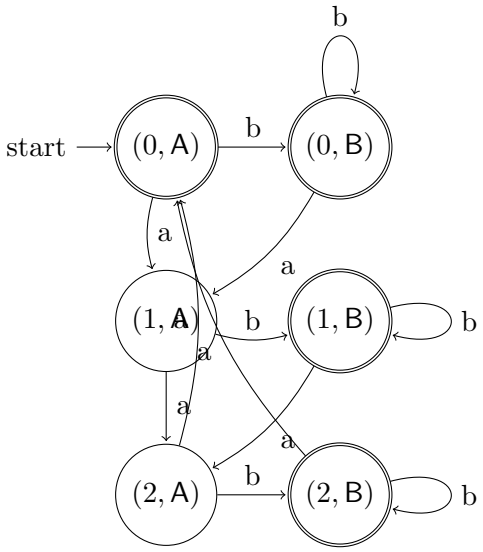


(6 pts) (c) The following language over alphabet $\Sigma = \{a, b\}$:

$$L = \{s \in \Sigma^* : s \text{ contains a multiple of 3 number of } a\text{'s } \textbf{or} \text{ ends with a } b.\}.$$

Hint: A DFA can have multiple accept states. Also, 0 is a multiple of every integer.

Solution:

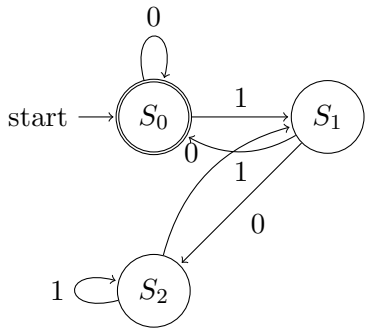


(6 pts) (d) The following language over alphabet $\Sigma = \{0, 1\}$:

$$L = \{b \in \Sigma^* : b \text{ is a binary representation of a nonnegative integer } x \text{ and } x \bmod 3 = 0\}.$$

Hint: Let x' be the value (in decimal) after appending a bit (either 1 or 0) to the right of b. How are x' mod 3 and x mod 3 related?

Solution:



(10 pts) 7. **Balanced Oreo.**

An Oreo is *balanced* if it has an equal number of cookies with the logo facing upwards and downwards, sandwiching the cream filling. We can represent an Oreo as a string over the alphabet $\Sigma = \{\overline{\text{O}}, \text{RE}, \underline{\text{O}}\}$, where $\overline{\text{O}}$ and $\underline{\text{O}}$ denote the cookies with logo facing upwards and downwards, respectively, and RE denote a cream filling layer. In a balanced Oreo, all $\overline{\text{O}}$'s must appear before any $\underline{\text{O}}$'s, and there must be at least one cream filling layer in between the $\overline{\text{O}}$'s and the $\underline{\text{O}}$'s. There could be cream filling layer(s) in between two $\overline{\text{O}}$'s or in between two $\underline{\text{O}}$'s while still being a balanced Oreo. For example, $\overline{\text{O}}\text{RE}\overline{\text{O}}\text{RE}\underline{\text{O}}\underline{\text{O}}$ is a balanced Oreo, but $\overline{\text{O}}\underline{\text{O}}$ and $\overline{\text{O}}\overline{\text{O}}\text{RE}\underline{\text{O}}$ are not.

Consider the language of strings over Σ that represent a balanced Oreo, either design a DFA that decides the language, or prove that no such DFA exists.

Solution: Thank you for the easter egg :-)
No DFA can decide the language of balanced Oreos, because the language is nonregular.

Consider the subset $L' = \{\overline{\text{O}}^n \, \text{RE} \, \underline{\text{O}}^n \mid n \geq 1\}$. Every string in L' is a balanced Oreo because all $\overline{\text{O}}$'s appear before the single RE, followed by $\underline{\text{O}}$'s, and the number of $\overline{\text{O}}$'s equals the number of $\underline{\text{O}}$'s. $L' \subseteq L$, where L is the language of balanced Oreos.

The language $\{a^n b^n \mid n \geq 1\}$ is not regular. Observe that L' is isomorphic to $\{a^n b^n \mid n \geq 1\}$ under:

$$a \mapsto \overline{\text{O}}, \quad b \mapsto \underline{\text{O}},$$

with RE inserted between the two blocks. Thus, L' is nonregular.

Assume for contradiction that L is regular. Then, the intersection:

$$L \cap \left(\overline{\text{O}}^* \, \text{RE} \, \underline{\text{O}}^*\right)$$

would also be regular. However, this intersection is exactly L', which is nonregular. So, L itself is nonregular, and thus, no DFA exists that decides the language of balanced Oreos.