EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 5

This homework has 9 questions, for a total of 100 points and 10 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*.

(0 pts) 0. **Before you start; before you submit.**

If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

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

Carefully read and understand the posted solutions to the previous homework; you may also find the video "walkthroughs" in the Canvas Media Gallery helpful. 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.

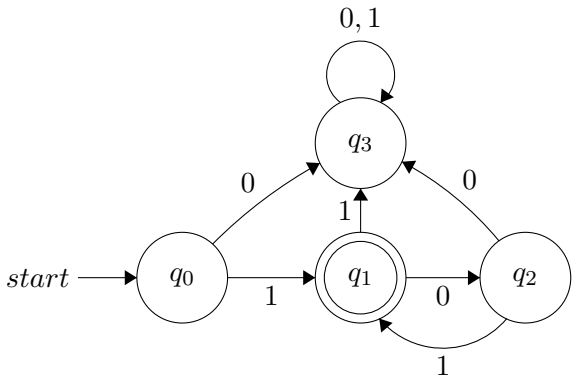
(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't turn in the previous homework, then (1) state that you didn't turn it in, and (2) pick a problem that you think is particularly challenging from the previous homework, and explain the answer in your own words. You may reference the answer key, but your answer should be in your own words.

Solution:

2. **Short answer.**

(4 pts) (a) Write the state-transition function (called δ in the lecture and the notes) for the following DFA, as a table. Also give a regular expression (using the "string notation" from class) for the language this DFA decides. No justification is required for either of these. For reference on formatting the state-transition function, see the lecture notes on Finite Automata.

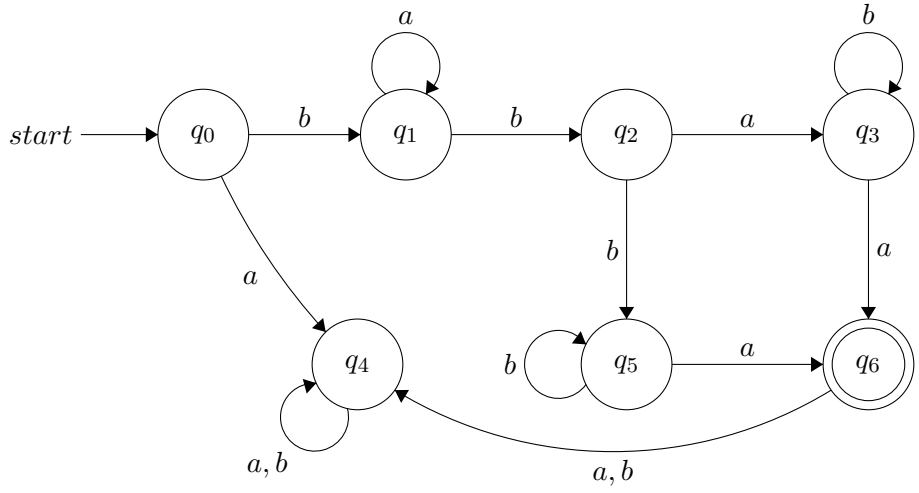


Solution: The state transition function δ is given by this table:

	0	1
q_0	q_3	q_1
q_1	q_2	q_3
q_2	q_3	q_1
q_3	q_3	q_3

A regular expression for the language decided by this DFA is $1(01)^*$.

(4 pts) (b) Give a regular expression (using the "string notation" from class) for the language decided by the following DFA. No justification is required.



Solution: One such regular expression is $ba^*b(ab^*a|bb^*a)$.

(4 pts) (c) Consider the following claim: a minimum spanning tree in a graph is *unique* if the edge weights are *distinct* (all different).

Briefly explain what is wrong 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.*

Solution: The last sentence does not logically follow from the previous ones. We have not ruled out the possibility that there are other MSTs that Kruskal's algorithm cannot output (regardless of what choices it makes). More generally, even if the output of some specific algorithm (Kruskal's algorithm in this case) is uniquely determined, this does not imply that this output is only optimal solution.

As an example to illustrate the previous sentence, consider the "greedy touring" problem below. We'll exhibit an input for which the greedy algorithm must output a specific unique solution, but there is also a different optimal solution. Let the input distances be $d_1 = d_2 = d_3 = 1$, and let $R = 2$. There are no arbitrary choices for the algorithm to make, only greedy ones, and the greedy choice is to refuel at town 2 alone (after traveling a distance of $R = 2$). But this is not the only optimal solution; it is also optimal to refuel at town 1 alone (after traveling distance 1) and then travel the remaining distance of 2 to complete the tour.

(4 pts) (d) Give a correct proof for the claim in the previous part.

Solution: Let T_1 and T_2 be any two (different) spanning trees of the graph. We will show that at least one of them is *not* a *minimum* spanning tree, using an exchange argument very similar to the one from the correctness proof for Kruskal's algorithm. This implies the claim, that there cannot be two different minimum spanning trees.

Let e be the lightest edge appearing only in one of the two spanning trees, which is T_1 without loss of generality. (Such an e must exist because the spanning trees are different, and e is unique because the edge weights are distinct.) As in the proof for Kruskal's algorithm, $T_2 \cup \{e\}$ contains a cycle C; let e' be the heaviest edge in this cycle. Then $T_2' := T_2 \cup \{e\} \setminus \{e'\}$ is a spanning tree. Moreover, T_2' is strictly lighter than T_2, because $w(e') > w(e)$. So T_2 is not a minimum spanning tree, as needed.

3. **Greedy knapsack filling.**

Recall that in the "0-1" knapsack problem, you are given n items, where the ith item has weight w_i and value v_i (both positive), and a non-negative weight capacity C 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.,

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

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.

(3 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:** Every valid choice of items in the 0-1 knapsack problem is also a valid choice in the fractional knapsack problem. So, the optimal total value for the fractional problem is at least as large as the optimal value for the 0-1 problem.

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

- Algorithm A: 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.
- Algorithm B: 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 A and B, 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:** The algorithm is not correct. To construct a counterexample, the basic observation is that Algorithm A chooses high-value items without regard to their weights, and Algorithm B chooses low-weight items without regard to their values. So, the counterexample will have some high-value item with large weight, and some low-weight item with tiny value, whereas an optimal solution will consist of an item that is "in the middle."
>
> Specifically, consider a knapsack with capacity $C = 10$, a 'high' item with value 100 and weight 100, a 'low' item with value 10 and weight 10, and a 'medium' item with

value 80 and weight 20. Algorithm A takes just 1/10 of the 'high' item, and gets value 10 from it. Algorithm B takes the entire 'low' item (and nothing else), and also gets value 10 from it. However, an optimal solution is to take 1/2 of the 'medium' item, for a much better value of 40.

(We can tweak the numbers so that an optimal solution is better than the algorithms' outputs by as large of a factor as we like. So, the algorithm in question produces very poor outputs in general.)

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

- Algorithm C: 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.

Here is the skeleton of an inductive proof that Algorithm C outputs an optimal solution. 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). View any selection of fractional items as a sequence $p_1, p_2, \ldots, p_n$, where p_i is the fraction of item i. Let $g_1, g_2, \ldots, g_n$ be the sequence of fractions that the greedy algorithm selects. We aim to prove the following claim:

> For *every* $k = 0, \ldots, n$, the first k values $g_1, \ldots, g_k$ of this sequence are a prefix of (i.e., are the first k item fractions in) *some* optimal solution.

Given this claim, the full sequence (for $k = n$) is itself an optimal solution, because there are no more items that can be (fractionally) selected.

Prove the claim by induction. Specifically, establish the base case $k = 0$, and then prove the inductive step, which says: for any $k < n$, if $g_1, \ldots, g_k$ is a prefix of *some* optimal solution $OPT = o_1, \ldots, o_n$, then $g_1, \ldots, g_k, g_{k+1}$ is a prefix of *some* optimal solution $OPT' = o'_1, \ldots, o'_n$. Use an exchange argument to modify OPT into a valid selection OPT', without reducing the value (so that it remains optimal).

Solution: We prove that Algorithm C is correct by induction following the structure laid out in the problem statement.

Base Case: When $k = 0$, there is nothing to prove: some optimal solution exists, and the condition on its prefix is vacuous when $k = 0$.

Inductive Step: Assume that for some $k < n$, the sequence $g_1, \ldots, g_k$ is a prefix of some optimal solution $OPT = o_1, o_2, \ldots, o_n$. We will prove that the sequence $g_1, g_2, \ldots, g_k, g_{k+1}$ is also a prefix of *some* optimal solution $OPT' = o'_1, o'_2, \ldots, o'_n$, by exhibiting such an OPT' using OPT and the greedy solution.

First, observe that $o_{k+1} \leq g_{k+1}$ by the design of the greedy algorithm, because $o_i = g_i$ for all $i = 1, \ldots, k$ and the greedy algorithm takes as much of item $k + 1$ as will fit in the remaining knapsack capacity (after taking the first k items). So, OPT cannot have $o_{k+1} > g_{k+1}$ because that would exceed the knapsack capacity (regardless of how much of the remaining items OPT takes).

The basic idea is to define OPT' from OPT by *increasing* the amount of item $k+1$ to match the greedily chosen fraction g_{k+1}, and by *decreasing* the amounts of *arbitrary* later items $j > k+1$ by *enough total weight to stay within the knapsack capacity.* (It is important to show that in OPT, there is enough total weight among those items to make this exchange; the math below does so.) This exchange adds at least as much value as it removes, for two reasons: item $k+1$ has the largest relative value (per unit weight) among all items $j \geq k+1$, and we are adding at least as much weight of this item as we are removing from the others. So, the overall value does not decrease, and OPT' is optimal, as needed.

We now proceed formally. Set $o'_i = g_i = o_i$ for $i = 1, \ldots, k$; set $o'_{k+1} = g_{k+1} \geq o_{k+1}$; and let

$$w := (o'_{k+1} - o_{k+1}) \cdot w_{k+1} \geq 0$$

be the extra weight from item $k+1$ in OPT' versus OPT. Let

$$w' := w + wt(OPT) - C \leq w$$

be the "excess weight" by which OPT would exceed the knapsack capacity with this increase in the amount of item $k+1$. Observe that in OPT, there must be weight at least w' among items $k+2$ through n, because there is weight at most C among items 1 through $k+1$, including the extra weight w of item $k+1$ (since these match the greedy selection, which is at most the knapsack capacity C).

If $w' < 0$, then there is no excess weight, so we set $o'_j = o_j$ for all $j \geq k+2$. Otherwise, we set $o'_{k+2} \leq o_{k+2}, \ldots, o'_n \leq o_n$ *arbitrarily* so that the total weight among the remaining items is reduced by w', i.e., $\sum_{i=k+2}^{n} (o_i - o'_i) w_i = w'$. Since $v_{k+1}/w_{k+1} \geq v_i/w_i$ for all $i \geq k+2$, comparing values, we get that

$$\begin{aligned}
val(OPT') - val(OPT) &= \sum_{i=k+1}^{n} (o'_i - o_i) \cdot v_i \\
&= (o'_{k+1} - o_{k+1}) \cdot v_{k+1} - \sum_{i=k+2}^{n} (o_i - o'_i) \cdot v_i \\
&\geq (v_{k+1}/w_{k+1}) \cdot \Big(w - \sum_{i=k+2}^{n} (o_i - o'_i) \cdot w_i \Big) \\
&\geq (v_{k+1}/w_{k+1}) \cdot (w - w') \geq 0 \,,
\end{aligned}$$

so OPT' is also an optimal solution, as needed.

(8 pts) (d) Now consider a new variant of the fractional knapsack problem, in which there are ℓ knapsacks, where the jth knapsack has capacity C_j, and we aim to maximize the total value across all the knapsacks. Give a greedy algorithm that solves this variant.

Solution: We start with a key observation. Consider two instances: (1) one with ℓ knapsacks of capacities $C_1, \ldots, C_\ell$, and (2) one with a single knapsack of capacity $C^* = \sum_{j=1}^{\ell} C_j$ (and the same item weights and values). Observe that a solution for one instance can be converted into a solution for the other instance with the same total value simply by splitting/merging fractions of items (the splitting can be done arbitrarily). Therefore, an optimal solution for one gives us an optimal solution for another.

This leads us to the following algorithm. We simply compute an optimal solution for the instance with a single knapsack of capacity C^*, using Algorithm C from the previous part. Then, we split fractions of items into ℓ knapsacks while preserving the total value. (More precisely, we arbitrarily put fractions of chosen items into the ith knapsack so that the total capacity does not exceed C_i.) This must be an optimal solution by the observation in the previous paragraph.

The running time is as follows. In $O(n \log n)$ time we can sort the items by their relative values, and sum the knapsack capacities in $O(\ell)$ time. Then Algorithm C solves the single-knapsack instance in $O(n)$ time, by making a single pass over the items. Finally, we can split the solution into ℓ solutions by making a single pass over the solution and allocating fractions to an appropriate knapsack, in $O(n + \ell)$ time. Altogether, the running time is $O(n \log n + \ell)$ time. (If the items are already presented in sorted order by relative value, then the running time is just $O(n + \ell)$.)

4. **Greedy touring.**

PETROBLUE, in support of the M Bus Formula One Team, has innovated a modular fuel pack for a bus that will transport EECS 376 students on a tour through various towns in Michigan. This fuel pack allows the vehicle to travel a certain distance before refueling. Each town has a service station where a fuel pack can be reloaded: the current pack is simply swapped out for a fresh full one. (Any remaining fuel in the removed pack will not be used by the bus.) The goal is to visit the towns in a specified order, while minimizing the number of fuel pack reloads.

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

(8 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: Consider the starting point as town $i = 0$. For convenience in the pseudocode, also define the 'sentinel' value $D[n+1] = \infty$.

The algorithm repeats the following: from the current town i, find the furthest town k such that the distance $\sum_{j=i+1}^{k} D[j]$ from town i to town k is at most R. It refuels at town k (adding that town to the initially empty list of towns) and repeats the process from town k.

```
1:  function GREEDY(D[1, ..., n], R)
2:      i ← 0, S ← []                          ▷ S is the list of towns where we refuel
3:      while i < n do
4:          j ← i, t ← 0                        ▷ t is distance from town i to j
5:          while t ≤ R do
6:              j ← j + 1
7:              t ← t + D[j]                    ▷ will be ∞ for j = n + 1
8:          S ← append(S, j − 1)
9:          i ← j − 1
10:     return S
```

Since no town is farther than R from the previous one, we strictly increase i in each iteration of the outer loop, ensuring that we access each element of the array at most twice. Hence, there are at most n iterations of the outer loop, and at most $2n$ iterations *in total* of the inner loop. Therefore, the overall running time of the algorithm is $O(n)$.

(8 pts) (b) Give an inductive proof using an exchange argument (like the ones presented in lecture and in the previous problem) that your algorithm from the previous part is correct.

Solution: Let S be the solution given by our greedy algorithm, and $|S|$ denote its number of fuel stops. Below we prove by induction for any $0 \le k \le |S|$, there exists some optimal solution `OPT` that has at least k fuel stops, and whose first k fuel stops are the same as those in S. Given this claim, S must be an optimal solution, because for $k = |S|$ the claim implies that S has no more fuel stops than some optimal solution.

For the base case when $k = 0$, there is nothing to prove. For the inductive step with $k < |S|$, the inductive hypothesis is that there exists an optimal solution `OPT` that has the same first k fuel stops as S. We want to construct an optimal solution `OPT`$'$ where the first $k+1$ fuel stops are the same as those of S.

Let S_{k+1} and O_{k+1} be the indices (town numbers) of the $(k+1)$st fuel stops in S and in `OPT`, respectively. (We know that `OPT` makes at least $k+1$ fuel stops, because it is not possible for the final fuel stop to be at S_k, since the greedy algorithm makes another stop after S_k.) If $S_{k+1} = O_{k+1}$, then we are done. If $S_{k+1} \neq O_{k+1}$, then we know that $O_{k+1} < S_{k+1}$ by the design of the greedy algorithm, because the kth fuel

stops in both S and OPT are the same (i.e., $S_k = O_k$), and the greedy algorithm goes as far as it can from the kth stop without exceeding distance R.

We construct OPT′ via an exchange as follows: let OPT′ and OPT have the same stops, except we simply replace O_{k+1} with S_{k+1}, as needed. This is a valid schedule of stop, because S_{k+1} is within distance R of the previous stop by the design of the greedy algorithm, and O_{k+2} is within distance R of O_{k+1}, which means O_{k+2} is less than distance R from $S_{k+1} > O_{k+1}$. Since OPT′ has the same number of fuel stops as OPT, it is also an optimal solution, as desired. This completes the proof.

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

PETROBLUE 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: The claim is not true. Let $D = [1, 1, 1]$, $R = 2$, and $C = [1, 2000, 1]$. The greedy algorithm will refuel (only) at town 2, at a cost of \$2000, but we could have refueled at town 1 and finished at the final town (without any more refueling) at a total cost of \$1.

The proof in the previous part fails because the exchange step does not work. More specifically, consider the step in the proof where we create another solution OPT′ from OPT by replacing the refueling at town 1 with refueling at town 2. Although this still creates a *legal* solution, it might be far from being optimal; the total cost will increase if refueling at S_{k+1} is more expensive than O_{k+1} (which it is in this example).

(3 EC pts) (d) *Optional extra credit.* Give an efficient dynamic programming algorithm (with recurrence and bottom-up implementation) that solves the problem from the previous part.

Solution: Let $OPT(i)$ be the optimal "cost" (number of refueling stops) for traveling from the starting point to town i, **including** refueling at town i. To account for the fact that we do not need to refuel at town n, we can set the refuel cost at n to be 0, i.e., $C[n] = 0$ without affecting correctness. Therefore, the optimal cost is exactly $OPT(n)$ from this setup, because it is optimal for traveling from the starting point to town n and refueling (for free) at town n, which is the same as not requiring refueling at the final town n.

For $i \geq 0$, let $p_i = \sum_{j=1}^{i} D[i]$ denote the total distance from the starting point to town i. So $p_0 = 0$. The recurrence for $OPT(i)$ is

$$OPT(i) = \min_{j:0 \leq j < i \wedge p_i - p_j \leq R} C[i] + OPT(j)$$

for all $i > 1$. The base case is $OPT(0) = 0$ because there is no refueling cost at the starting point. For $i > 0$, to travel to town i and refuel at town i, we must have first traveled to some town j **and** refueled there most recently, for some $0 \leq j < i$ and $p_i - p_j \leq R$. Finally, we refuel at town i. Thus, this gives the total cost of $C[i] + OPT(j)$. Therefore, $OPT(i)$ is obtained by minimizing over the choice of valid towns j described above.

This recurrence leads to the following code:

```
1:  function OptFuelCost(D[1,...,n], C[1,...,n])
2:      C[n] ← 0
3:      allocate memo[1,...,n] of 0s
4:      for all i ≥ 0, p_i ← Σ_{j=1}^{n} D[i]
5:      for i = 1 to n do
6:          minCost ← ∞
7:          for j = i − 1 to 0 do
8:              if p_i − p_j > R then break
9:              minCost ← min{minCost, C[i] + memo[j]}
10:         memo[i] ← minCost
11:     return memo[n]
```

Note that the algorithm breaks on line 8 as soon as p_j become so far away from p_i, that it would be impossible to travel town j to town i without refueling. The time complexity is $O(n^2)$ because there are two nested loops, and each loop runs at most n iterations.

Constructing an optimal sequence of towns can be done via standard backtracking, where for each i we additionally store some value of j that gave us the value of memo$[i]$. From this information we can construct an optimal sequence in $O(n)$ time.

5. **Selecting suitable skis.**

Suppose there are n skiers of various heights, and n pairs of skis having various lengths. (Both skis in a pair have the same length.) We want to match each skier with a pair of skis so as to *minimize the total difference* between each skier's height and the length of their skis.

Formally, we are given a *sorted* array $P[1, \ldots, n]$ of the skiers' heights (where $P[i] \leq P[i+1]$ for all $i < n$), and a (not necessarily sorted) array $S[1, \ldots, n]$ of the skis' lengths. We want to return a rearrangement (permutation) $S'[1, \ldots, n]$ of S that minimizes

$$\sum_{i=1}^{n} \big|P[i] - S'[i]\big|.$$

(3 pts) (a) Give an $O(1)$-time algorithm for the case $n = 2$.

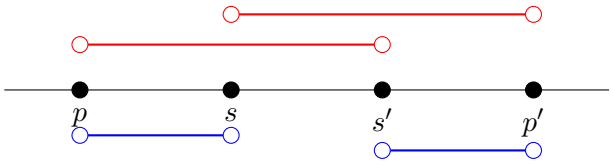
Solution: Just let S' be S in sorted order. Since there are only two entries, sorting takes $O(1)$ time.

To see correctness, let us simplify the notation: let $p \leq p'$ be the heights of the skiers, and let $s \leq s'$ be the length of the skis. We want to prove that the total difference is optimal when p goes with s and p' goes with s':

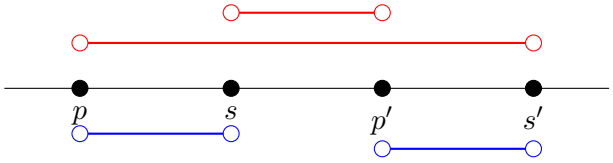
$$\big|p - s'\big| + \big|p' - s\big| \geq |p - s| + \big|p' - s'\big|.$$

Let us assume that p is smallest among the four numbers. The proof is symmetric if s is smallest. There are three cases, and the inequality can be verified via the corresponding diagram:

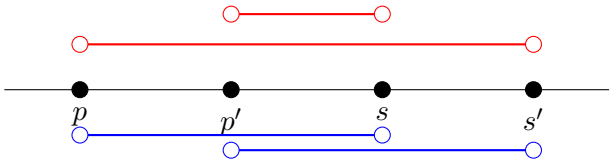
1. $s, s' \leq p'$.



2. $s \leq p' \leq s'$,



3. $p' \leq s, s'$.



(7 pts) (b) For general n, consider an algorithm that just returns S in sorted order. We can view this as a greedy algorithm that just repeatedly selects a pair of skis of shortest length (among those that remain). So, it makes exactly n selections $s'_1, s'_2, \ldots, s'_n$ in sorted order.
Give an inductive proof using an exchange argument (like the ones presented in lecture and in the previous problems) that this algorithm is correct.
Hint: Use the previous part as an ingredient of your proof.

Solution: We will prove by induction that this algorithm is correct. The inductive hypothesis is that there exists an optimal sequence OPT in which the first k skis have lengths $s'_1, s'_2, \ldots, s'_k$. For the base case when $k = 0$, there is nothing to prove. The inductive step is to show that, assuming the existence of OPT, there also exists an optimal sequence OPT' in which the first $k+1$ skis have lengths $s'_1, \ldots, s'_{k+1}$.
If OPT already has the needed form, then let $OPT' = OPT$, and we are done. If not, then the $(k+1)$st ski length in OPT is some $\ell > s'_{k+1}$ (any the shorter skis are in the first k of OPT, because OPT matches the sorted greedy selection on the first k), and s'_{k+1} appears somewhere later in OPT's ordering. We get OPT' from OPT simply by exchanging ℓ and s'_{k+1}. This cannot increase the total height-length difference, by the argument in part (a): the only height-length differences that this change can affect are those of the two skiers whose skis were exchanged. Therefore, OPT' is also an optimum, and it has the required form as needed.

6. **Decision problems and languages.**

Recall that the goal of a *decision problem* is to determine whether a given input "object" has a certain "property", e.g., determine whether a given integer is prime, or whether a given string is a palindrome. In class, we said that any decision problem is equivalent to a *membership problem* for a corresponding *language*. That is, determining whether a given object has the property is equivalent to determining whether its encoding (as a string) is a member of the language.

For each of the following decision problems, (i) define a reasonable (finite) alphabet Σ, (ii) give the encoding (over the alphabet) of a representative input object, (iii) analyze the length of the encoding in terms of the value of the input, and (iv) define a language L for the corresponding decision problem. Use the notation $\langle X \rangle$ for the encoding of object X as a string over the alphabet. As examples, we have provided solutions to the first two parts.

(0 pts) (a) "Does a given non-negative integer k have 3 as its last digit, when written in base 10?"

Solution:

(i) A reasonable alphabet consists of the decimal digits, $\Sigma = \{\texttt{0}, \ldots, \texttt{9}\}$. Alternatively, we could use the binary digits $\{0, 1\}$, or hexadecimal digits (0 through 9 and A through F), or the digits from any other base (including unary), or some entirely different-looking alphabet like $\{\star, \otimes, \perp\}$ (though the latter would not be very human-friendly). No matter the choice of (nonempty, finite) alphabet, there is a way to unambiguously encode (represent) non-negative integers as strings of characters from that alphabet.

Note that it would *not* be valid to include all of the non-negative integers in the alphabet, because an alphabet must be a *finite* set, and there are infinitely many non-negative integers.

(ii) Encoding: for an integer k, write it in base 10 in the usual way, as a string of digits. For example, if k is the integer forty-seven, then $\langle k \rangle = \texttt{47}$. We stress that this is a *string* of the characters `4` and `7`, which *represents* the number forty-seven.

(iii) The length of this encoding would be the number of digits in the value, which is $\Theta(\log k)$.

(iv) A corresponding language would be $L_{EndsWith3} = \{\langle k \rangle : k \bmod 10 = 3\}$. It would also be acceptable to copy the phrasing of the decision problem, i.e., $L_{EndsWith3} = \{\langle k \rangle : k$ written in base 10 has 3 as the last digit$\}$

Alternatively, we could also encode integers in binary, using the alphabet $\Sigma = \{0, 1\}$. For example, if $k = 5$, then $\langle k \rangle = \texttt{101}$. The length of this encoding is still $\Theta(\log k)$. The above two definitions of the language hold without modification, because they both describe membership only in terms of the *value* of k, not its encoding.

(0 pts) (b) "Is a given array of non-negative integers sorted?"

Solution:

(i) A reasonable alphabet Σ is the set of ASCII characters, or more selectively, the decimal digits along with some separator characters: $\Sigma = \{\texttt{0}, \ldots, \texttt{9}, \texttt{[}, \texttt{,}, \texttt{]}\}$. (Notice that a comma is one of these characters.)

Note that we can't just use the decimal digits alone if we also want some special symbols to indicate the start/end of the array and to separate the elements.

(ii) Example encoding: For an array A, encode its elements as in the previous part, and list those encodings with appropriate separators. For example, the array A with entries one, two, three, and four would have $\langle A \rangle = \texttt{[1,2,3,4]}$.

(iii) The length of this encoding is $\Theta(n + e)$, where n is the number of elements in the array, and e is the total length of the encodings of the elements.

(iv) The corresponding language is

$$L_{\text{sorted}} = \{\langle A \rangle : A \text{ is a sorted array of non-negative integers.}\}.$$

(3 pts) (c) "Given an array S of integers, are all its elements distinct?"

Solution:

(i) A reasonable choice is $\Sigma = \{\texttt{-}, \texttt{0}, \ldots, \texttt{9}, \texttt{[}, \texttt{,}, \texttt{]}\}$, the same alphabet as in the

previous part, but with a minus sign `-` added.

(ii) We'll adapt the encoding from the previous parts. The only difference is that in this part the array can contain *negative* integer values. For any negative value, just include a negative sign in the usual way. For example, if $S = [6, 342, -47]$, then $\langle S \rangle =$ `[6,342,-47]`.

(iii) As before, the length of this encoding is $\Theta(|S| + \ell)$, where ℓ is the total length of the encodings of all the elements. This is because there are at most $|S|$ negative signs, which does not change the length asymptotically.

(iv) We could express this language as $L_{\text{Distinct}} = \{\langle A \rangle : A[i] \neq A[j] \text{ for all } i \neq j\}$.

(3 pts) (d) "Is a given undirected graph complete?" *Hint*: Consider the adjacency matrix.

Solution: Without loss of generality we can name the vertices $1, 2, \ldots, |V|$, and we can represent the edges by an adjacency matrix. To encode this matrix as a string, we can represent each row of the matrix as a 0/1 string, and separate the rows by a special character.

(i) We define $\Sigma = \{$`0`, `1`, `,`$\}$.

(ii) We encode each row of the graph's adjacency matrix as a bit string, and separate the rows by commas. For example, consider an undirected graph $G = (V, E)$ with $V = \{1, 2, 3\}$ and a single edge $(1, 2)$. Then, the string encoding would be $\langle G \rangle =$ `010,100,000`.

(iii) The length of this encoding is $\Theta(|V|^2)$.

(iv) We could express the language as

$$L_{\text{ConnectedGraph}} = \{\langle G \rangle : G \text{ is a complete undirected graph}\} .$$

(3 pts) (e) "Does a given C++ program halt when run on a given input?"

Solution: We need to encode both a program and its input. For uniformity, we can use the ASCII character set for both. Then, we just need some special character to separate the two.

(i) We define $\Sigma = \text{ASCII} \cup \{\Delta\}$, where $\Delta \notin \text{ASCII}$ denotes some additional non-ASCII character.

(ii) Suppose we are given the C++ program

```
int main() {
    string m =
    "bool M(string x) {
```

```
                return True;
            }"
        }
```

and input $x =$ `Hello world`. The encoding of M, x would be

$\langle M, x\rangle =$ `int main() {\n\tstring m =\n\t"bool M(string x) {\n\t\t return True;\n\t}"\n}ΔHello world`.

(Here, `\n` denotes the ASCII newline character, and `\t` denotes the ASCII tab character).

(iii) The length of this encoding is $\Theta(|M| + |x|)$.

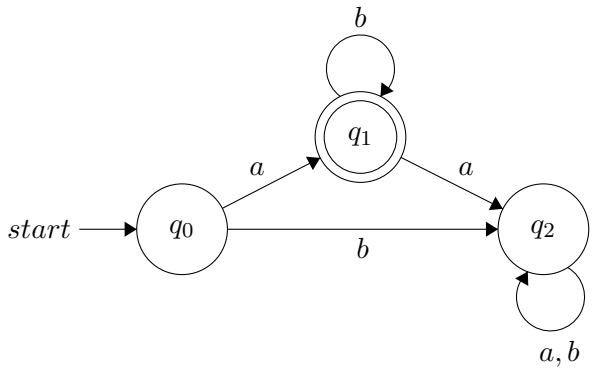
(iv) We can express the language as $L_{\text{Halt}} = \{\langle M, x\rangle : M \text{ halts when run on } x\}$.

7. **Deciding languages with 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.

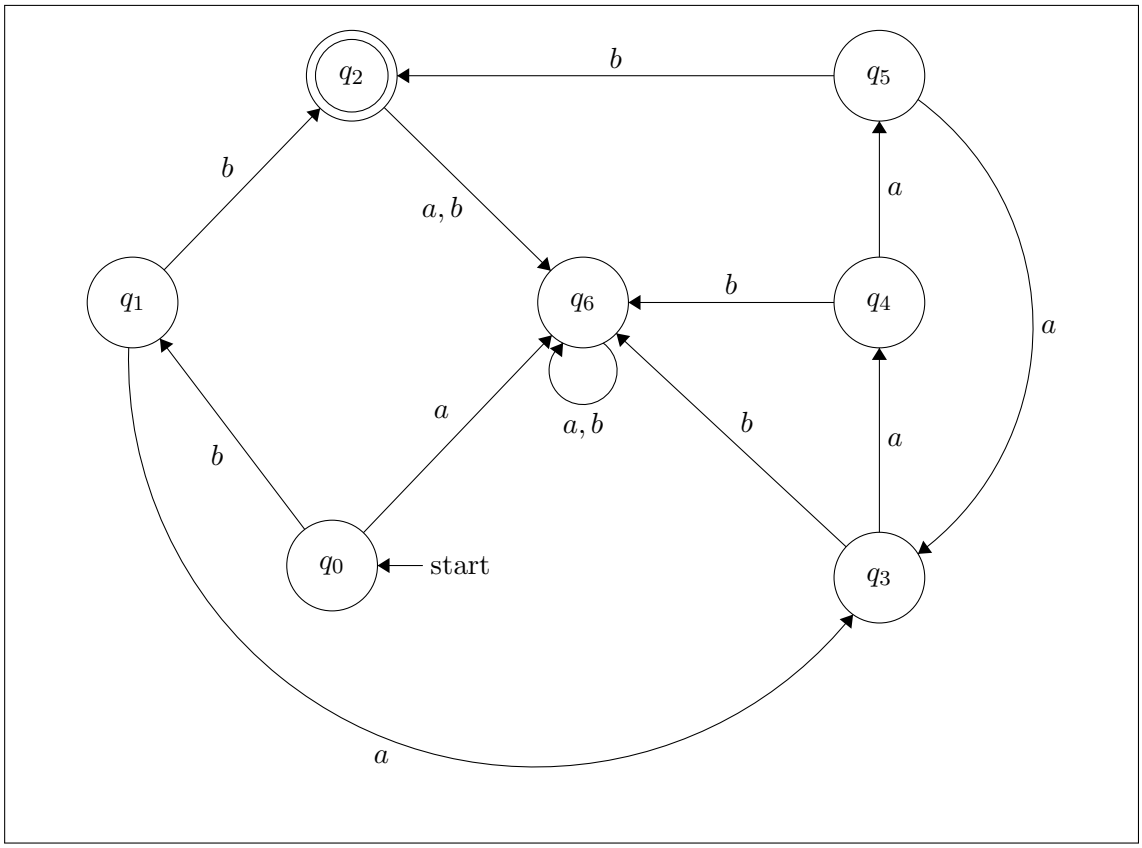
(3 pts) (a) The language (over alphabet $\{a, b\}$) corresponding to the regular expression ab^* .

Solution: The DFA should accept if and only if it first sees one a, and then zero or more bs. On any character that does not fit this pattern, the DFA goes to its "sink" state q_2.



(5 pts) (b) The language (over alphabet $\{a, b\}$) corresponding to the regular expression $b(aaa)^*b$.

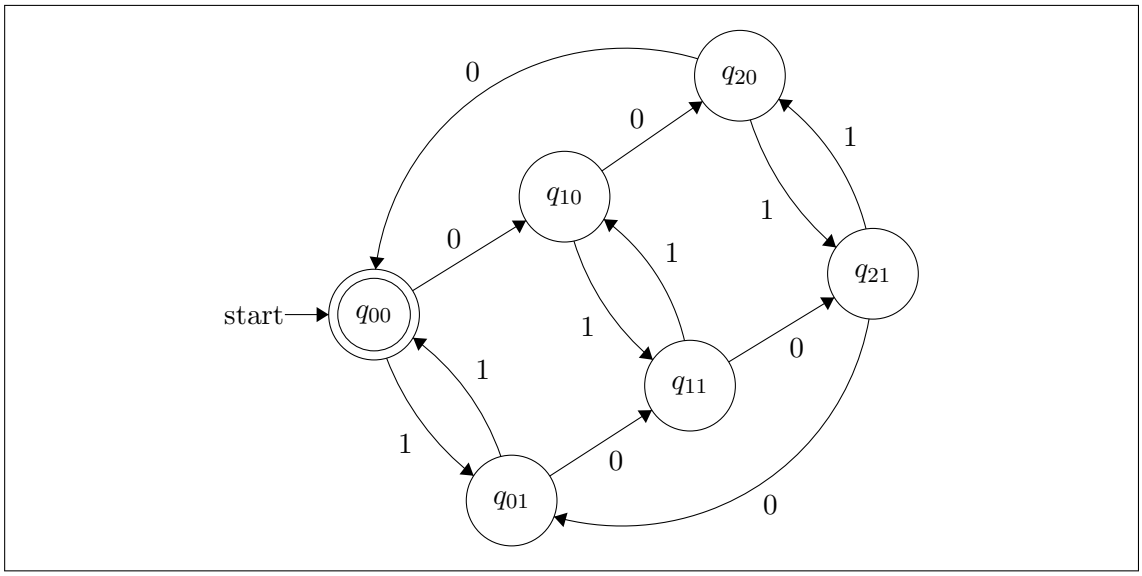
Solution: Here q_1 and q_5 are the states from which we want read zero or more aaa sequences followed by b (alternatively, q_1 and q_5 can be merged); states q_3, q_4, q_5 are used to check that the as are coming in groups of 3; and q_6 is the "sink" state that is reached if the input ever diverges from the regexp pattern.



(5 pts) (c) The language of binary strings in which the number of 0s is divisible by 3 *and* the number of 1s is even.

For example, the strings ε, 000, 01010, and 11000 are in this language, while 1, 01, 001, 0011, and 000111 are not.

Solution: We have six states q_{ij} for $i \in \{0, 1, 2\}$ and $j \in \{0, 1\}$, where q_{ij} is the state where the number of 0s (modulo 3) read so far is i, and the number of 1s (modulo 2) read so far is j. We have q_{00} as the start state and the only accepting state.



(4 EC pts) (d) *Optional extra credit.*

Consider the following language:

$$L = \{x \in \{0,1\}^* : [x] \bmod 3 = 0\}.$$

where $[x] := \sum_{i=0}^{n} 2^i x_i$ for $x = x_0 x_1 \dots x_n$. E.g., $[\varepsilon] = 0$ and $[01100000] = [011] = 6$. Note that the bits are given in 'reverse' order from what is typical (e.g., 6 in binary is usually written as 110).

Give a DFA that decides the language L.

Hint: Relate $[x_0 x_1 \dots x_k]$ to $[x_0 x_1 \dots x_{k-1}]$, x_k, and k. You can use the fact (which is easy to prove) that $2^{2m} \bmod 3 = 1$ and $2^{2m+1} \bmod 3 = 2$ for any non-negative integer m.

Solution:

We start with an elegant solution that does not even use the provided hint, and results in a DFA having only three states. It is based on a recursive equation for $[x]$, and a focus on the string that *remains* for the DFA to process, rather than what the DFA has *already* processed.

First observe that for $x \neq \varepsilon$, we can write $x = x_0 x_r$ for an initial bit $x_0 \in \{0,1\}$ and the remaining string $x_r \in \{0,1\}^*$. From the definition of the $[\cdot]$ notation, we can see that $[x] = x_0 + 2 \cdot [x_r]$. Since $2 \equiv -1 \pmod 3$, we get that

$$[x] \equiv x_0 - [x_r] \pmod 3 .$$

This inspires the following approach to designing a DFA: instead of having states corresponding to "what integer value (mod 3) have we read *so far*," let's make the states correspond to "what integer value (mod 3) does the *remainder of the string need to be* in order for the *entire string* (from the start) to be in the language?"

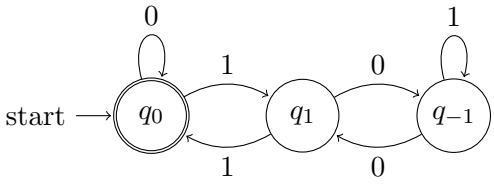
At any particular state corresponding to a needed value, reading the next bit determines what value the remainder of the string needs to be. The general rule is that if

we need n and we read x_0, then we need the n' for which $n \equiv x_0 - n' \pmod 3$, i.e., $n' \equiv x_0 - n \pmod 3$. Specifically:

- At the "need 0" state: if we read 0, then we need the rest of the string to be 0 (mod 3); if we read 1, then we need the rest to be 1 (mod 3).
- At "need 1": if we read 0, we need -1; if we read 1, we need 0.
- At "need -1": if we read 0, we need 1; if we read 1, we need -1 (because $1 - (-1) \equiv -1 \pmod 3$).

Finally, as a base case, when there is no more string to read, we should accept if and only if we need 0; this means that the "need 0" state should be the only accepting state. This corresponds to the fact that $[\varepsilon] \in L$.

Putting all this together, we get the following simple DFA.



Now we give a more complicated analysis and DFA construction that more directly follows the definition of $[x]$ and the hint, but has six states (until we finally notice that they can be folded into three).

First observe that for any $k > 0$, by definition, $[x_0 x_1 \dots x_k] = [x_0 x_1 \dots x_{k-1}] + 2^k x_k$.

If x_k is 0, we add nothing, so the result is the same as the previous result. If x_k is 1, we add 2^k, so we should determine the value of ($2^k \bmod 3$) to see how it changes the previous result.

We have two cases to consider: k even or k odd.

- If k is even, $k = 2m$ for some m, and we can prove by induction on m that $2^{2m} \equiv 1 \pmod 3$.
- If k is odd, $k = 2m + 1$ for some m, and we can prove by induction on m that $2^{2m+1} \equiv 2 \pmod 3$.

Putting all this together, for a nonempty string $x_0 \dots x_k$,

- If $x_k = 0$, $[x_0 x_1 \dots x_k] \equiv [x_0 x_1 \dots x_{k-1}] \pmod 3$.
- If $x_k = 1$, we consider two sub-cases:
  - If k is even, $[x_0 x_1 \dots x_k] \equiv [x_0 x_1 \dots x_{k-1}] + 1 \pmod 3$.
  - If k is odd, $[x_0 x_1 \dots x_k] \equiv [x_0 x_1 \dots x_{k-1}] + 2 \pmod 3$.

Therefore, we design a DFA to track two things:

1. The running value (mod 3) of the first k bits of the input:
$$[x_0 x_1 \dots x_{k-1}] \bmod 3.$$
2. The parity k mod 2 of the next position k that will be read.

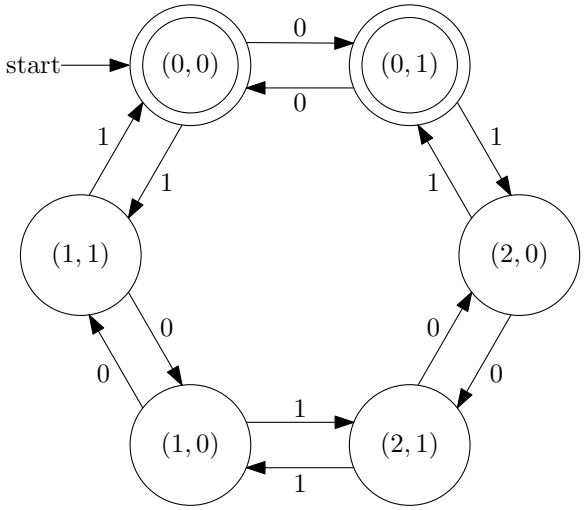
With this observation, we construct a DFA as follows.

- Its states are indexed by pairs (i, j), where if the DFA has read k bits of the input so far, it should be in the state corresponding to $i = [x_0 x_1 \dots x_{k-1}] \bmod 3$ and $j = k \bmod 2$.
- The accepting states are the two for which $i = 0$ (at the end of the string, it does not matter what j is).
- The start state is $(0, 0)$, because we have not read anything yet (i.e., $k = 0$).

For the state transitions, we track the parity of k by alternating between $j = 0$ and $j = 1$ with each bit that is read. We also transition the value of i by the observations above: if $x_k = 0$, the first entry remains the same, and if $x_k = 1$, the first entry advances by 1 or 2 (mod 3) depending on j (the parity of k). These rules give us the following transition table:

state	read 0	read 1	
(0,0)	(0,1)	(1,1)	(start state)
(1,0)	(1,1)	(2,1)	
(2,0)	(2,1)	(0,1)	
(0,1)	(0,0)	(2,0)	
(1,1)	(1,0)	(0,0)	
(2,1)	(2,0)	(1,0)	

One may also draw the DFA diagram.



Finally, if we notice the symmetry between the states and the transition arrows, we can "fold" the DFA over the vertical axis and merge the overlapping states, which results in exactly the same DFA from our first solution!

(3 EC pts) 8. *Optional extra credit.*

Consider the alphabet $\Sigma = \{\texttt{(}, \texttt{)}\}$, which consists of the open- and close-parenthesis characters `(` and `)`. Say that a string $x \in \Sigma^*$ is *balanced* if x has an equal number of `(` and `)` characters, and every *prefix* of x has at least as many `(` characters as `)` ones.

For example, the empty string ε, `(()())`, and `((()))` are balanced, but `())(()` is not, because it fails the second condition (but not the first one).

Either give a DFA that decides the language of balanced strings over Σ, or rigorously prove that no such DFA exists.

Solution: No such DFA exists. Let B be some arbitrary DFA. We will show that B does *not* decide the language in question, by exhibiting a string on which B produces the wrong output (i.e., it rejects a balanced string, or accepts an unbalanced one).

Let n be the number of states of B. Let q_i be the state of B after it reads the string $\texttt{(}^i$. By the pigeonhole principle, there must be indices $0 \leq i < j \leq n$ such that $q_i = q_j$. Now consider the strings $\texttt{(}^i\texttt{)}^i$ and $\texttt{(}^j\texttt{)}^i$. The DFA must output the *same answer* on both strings, because it is in state $q_i = q_j$ after reading the `(` characters in either string, and then it reads $\texttt{)}^i$ in both cases. However, the first string is balanced, and the second one is not (because $i \neq j$), so B produces the wrong output on one of them, as claimed.