- Problems marked with **P** are graded on progress, which means that they are graded subjectively on the perceived progress toward a solution, rather than **solely** on correctness.
- Problems marked with a **G** are group problems. A group of 1-3 students may work on these together and turn in one assignment for the entire group. Each member listed should have made real contributions. The group problems will be turned in separately on gradescope.

For bonus and most extra credit questions, we will not provide any insight during office hours or 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 look at any other student's write-up.

---

1. **Recurring Recurrences.** (10) Give recurrence relations (including base cases) that are suitable for dynamic programming solutions to the following problems. You do not need to prove their correctness.

   Given a string S, determine the length of a longest subsequence of S (not necessarily a substring) that is a palindrome. Recall that a palindrome is a string which is the same forwards and backwards (e.g., "racecar").

Solution: Define a function LS(i,j) to return the length of a longest sub sequence in string S that is a palindrome, where i and j are indices of characters in string S. Given that the first call to LS(i,j) has i as the first character and j as the last character the recurrence would be as following:

$$LS(i,j) = \begin{cases} 1 & \text{if } i = j \text{ (they are the same index)} \\ 2 & \text{if } i+1 = j \& S[i] = S[j] \\ max\{LS(i+1,j), LS(i,j-1)\} & \text{if } S[i] \neq S[j] \\ LS(i+1,j-1)+2 & \text{if } S[i] = S[j] \& i+1 \neq j \end{cases}$$

G 2. **Balancing Weights**. (25) You have $n \geq 1$ positive *integer* weights, $w_1, \ldots, w_n$. The goal is to find a subset of indices $A \subseteq \{1, \ldots, n\}$ so that the difference $|\sum_{i \in A} w_i - \sum_{i \notin A} w_i|$ is *minimized* and to return this difference. (Treat an empty sum as 0.) For example, if $n = 4$ and $(w_1, w_2, w_3, w_4) = (1, 3, 2, 4)$, then $A = \{1, 4\}$ minimizes the aforementioned difference and 0 should be returned.

   (a) The naïve algorithm is given by: "try all possible subsets for A: for each such set, compute the sums, compute the difference, and update the best choice of A if need be". Express the complexity of the naïve algorithm with big-Θ notation.

Solution: The cost to compute a sum is O(n) because we must loop through the elements to add them up.The cost to computer a difference is O(n) because we must loop through the elements not in the set to add them up and then the actual calculation of the difference is constant. To update A is also constant time. This must be done 2^n times because there are 2^n possible subsets for a set of size n. Therefore, the complexity of the naive algorithm is $\Theta(2^n n^2)$.

(b) Provide a bottom-up solution to this problem using dynamic programming. Aim for a total running time of $O(nW)$, where $W = \sum_{i=1}^{n} w_i$ is the sum of the weights.
Hint: This problem is similar to the 0-1 knapsack problem.

Solution: function MinimumDifference(W[1,2,..,n])
int sum=0;
for(i=0; i<n;i++){
sum=sum+W[i];
}
bool memo[n+1][sum+1];
for(int i=0; i≤n;i++){
memo[i][0]=true;
}
for(int i=1; i≤sum;i++){
memo[0][i]=false;
}
for(int i=1; i≤n; i++){
for(int j=1; i≤sum; j++){
memo[i][j]=memo[i-1][j];
if W[i-1]≤j then memo[i][j]=memo[i][j] || memo[i-1][j-W[i-1];
}
}
int diff=MAXINT;
for(int i=sum/2; i≥0;i–){
if memo[n][i]=true then diff=sum-2*i;break;
}
return diff;

(c) Modify your algorithm from the previous part to return a set of indices A that minimizes the difference. Your algorithm should continue to run in time $O(nW)$.

Solution: function MinimumDifference(W[1,2,..,n])
int sum=0;
for(i=0; i<n;i++){
sum=sum+W[i];
}
int memo[n+1][sum+1][n];

```
for(int i=0; i≤n;i++){
memo[i][0][0]=1;
}
for(int i=1; i≤sum;i++){
memo[0][i][0]=0;
}
for(int i=1; i≤n; i++){
for(int j=1; i≤sum; j++){
memo[i][j][0]=memo[i-1][j];
if W[i-1]≤j then memo[i][j][0]=memo[i][j][0] || memo[i-1][j-W[i-1][0]; memo[i][j][memo[i][j].size]=i;
}
}
int arr[n];
for(int i=sum/2; i≥0;i–){
if memo[n][i]=true then arr=memo[n][i][1,...,n];break; Arr contains all elements in
memo at that row and column besides the first value.
}
return arr;
```

(d) In what case would you expect the naïve algorithm to run faster than the DP algorithm from the previous part? List at least 1 case in which this could happen, assuming that you run both algorithms on the same machine.

Solution: The naive algorithm would run faster than the DP algorithm in the case that the sum of the weights of the items are extremely high compared to the number of them and the DP algorithm would spend a lot of time computing values for all of the possible weights. For example in the case W: 40,50,70,20, the naive algorithm runs in $O(2^4 * 4^2) = 256$ and the dp algorithm runs in $O(4 * 180) = 720$.

3. **Decidability**. (20) Show that each of the following binary languages are decidable on a "normal" computer by describing (no code needed) a decision program written for such a computer, i.e. given a binary string $x \in \{0,1\}^*$, your program should return "true" or "false" based on whether or not x is in the language. For each part, there is some fixed encoding function, $\langle\cdot\rangle$, that allows one to represent the objects of interest (strings, a pair of numbers, graphs, etc) as a finite binary string (in $\{0,1\}^*$). You may assume that, given a binary string $x \in \{0,1\}^*$, it is possible to check if x represents an object of interest and to "decode" x if so (e.g., convert x into a graph).

   For example, one possible decision program for the binary language $\{\langle$"Brehob"$\rangle$, $\langle$"Stout"$\rangle$, $\langle$"Zhu"$\rangle\}$ would be to check if a given binary string $x \in \{0,1\}^*$ is equal to one of $\langle$"Brehob"$\rangle$, $\langle$"Stout"$\rangle$, or $\langle$"Zhu"$\rangle$ (these are binary strings, thanks to the encoding function $\langle\cdot\rangle$). If so, the program returns "true"; otherwise, it returns "false".

   Hint: Do not aim to be efficient. Just sketch what the program does.

   (a) $\{\langle(p,q)\rangle \mid p,q \in \mathbb{Z}^+ \text{ and } \gcd(p,q) = 2\}$.

> **Solution:** Given a binary string $x \in \{0,1\}^*$ a normal computer can decide if is in the binary language. Using the encoding function $\langle\cdot\rangle$ on the binary stirng to represent a pair of positive integers. Given these integers from the original binary string the program could calculate the greatest common denominator of the two by running the Euclidean algorithm and if it equals 2 the program would return true and if not it would return false.

(b) $\{\langle G\rangle \mid G$ is a graph that can be 3-colored$\}$. Recall that a graph is *3-colorable* if there is a way to color its vertices using at most 3 colors so that no two adjacent vertices have the same color.

> **Solution:** Given a binary string a program can decide if it is in the language by using the encoding function $\langle\cdot\rangle$ to have the binary string represent a graph. Then by using a brute-force approach of trying every possible 3-color arrangement on the graph to see if any of them yield a graph that has no connected nodes as the same color, we can return true if the approach was able to find a possible 3-color arrangement on the binary encoded graph where no connected nodes had the same color and false if not.

(c) $\{\langle(f,u,n)\rangle \mid f$ is a C++ function that takes in an integer argument and u,n are positive integers such that, on input u, $f(u)$ terminates after at most n steps$\}$.

> **Solution:** Given a binary string a program can decide if it is in the language by using the encoding function $\langle\cdot\rangle$ to represent the binary string as a function and two integers. Using this the progrm can run the function with an input value of the first integer and count how many steps it took for the program to terminate. Once the number of steps is calculated the program can return true if the number of steps is less than or equal to the second integer and return false if it is greater than the second integer.

(d) $\{\langle(P,n)\rangle \mid P$ is a proposition in predicate logic and n is a positive integer such that P has a natural deduction proof that uses at most n characters$\}$.

> **Solution:** Given a binary string a program can decide if it is in the language by using the encoding function $\langle\cdot\rangle$ to convert the binary string into representing a proposition and a positive integer. Then the program could call some function written by a very smart person that can write the shortest natural deduction proof possible for that proposition. Then the program could use a function to count how many characters are in this proof and if the number of characters is less than or equal to the integer from the binary string the program returns true, otherwise the program returns false.

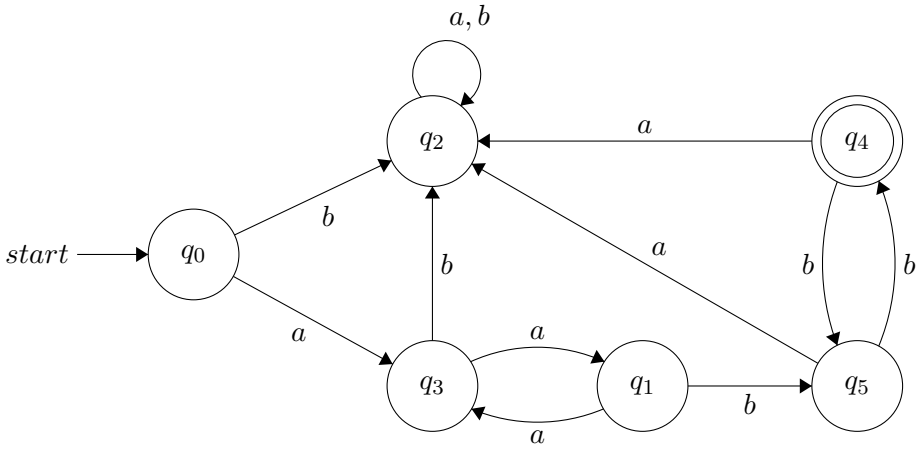
4. **DFAs**. (15) The following questions concern languages over $\Sigma = \{a,b\}$. If you're using LaTeX, you might find https://www.cs.unc.edu/~otternes/comp455/fsm_designer/ useful.

(a) Draw a DFA to decide the language $\{abb^*a\}$.

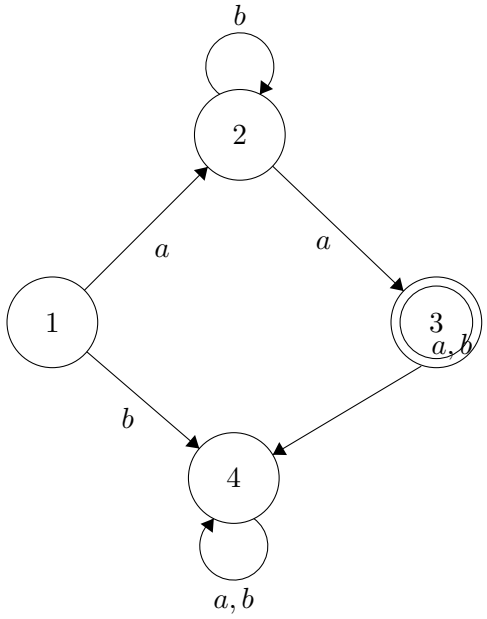
(b) Draw a DFA to decide the language $\{x \in \{a,b\}^* \mid \text{the last character of } x \text{ is } b\}$.

(c) Draw a DFA to decide the language $\{x \in \{a,b\}^* \mid x = ab^* \text{ or } x = bba^*\}$.
Hint: Your answer should have less than 8 states.

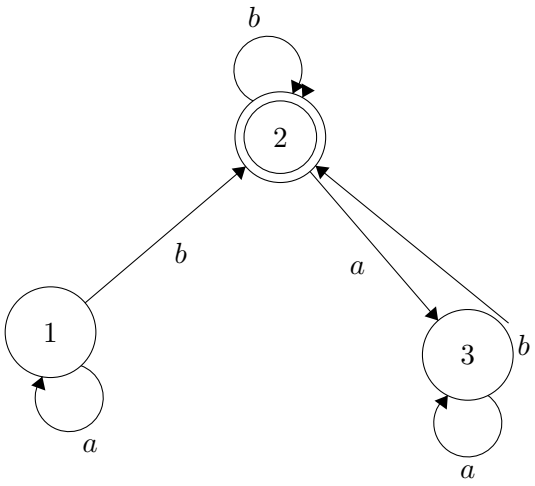
(d) What is the language decided by the following DFA?



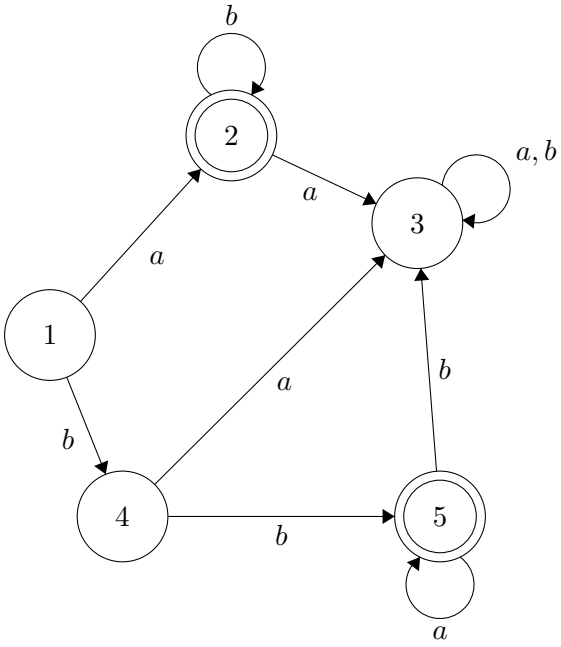
Solution: a)



b)



c)

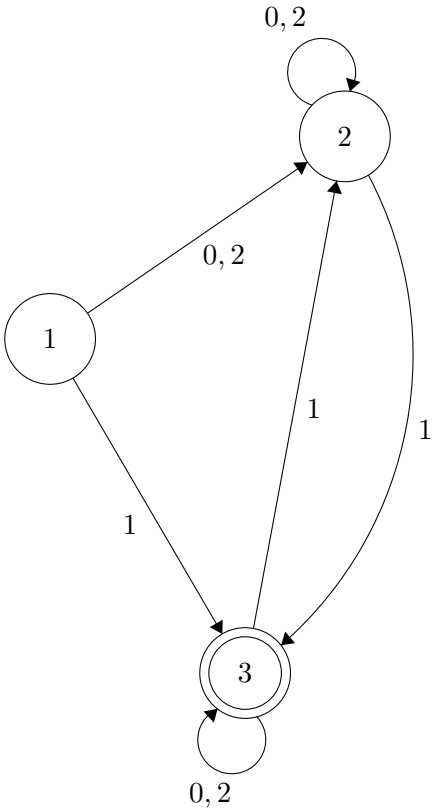


d)
The language where $\{x \in \{a,b\}^* \mid$ where x is any even number of consecutive a's followed by any even number of consecutive b's $\}$

5. **More DFAs**. (20) For each of the following problems, construct a DFA that decides the given language. If you're using LaTeX, you might find https://www.cs.unc.edu/~otternes/comp455/fsm_designer/ useful for making state diagrams.

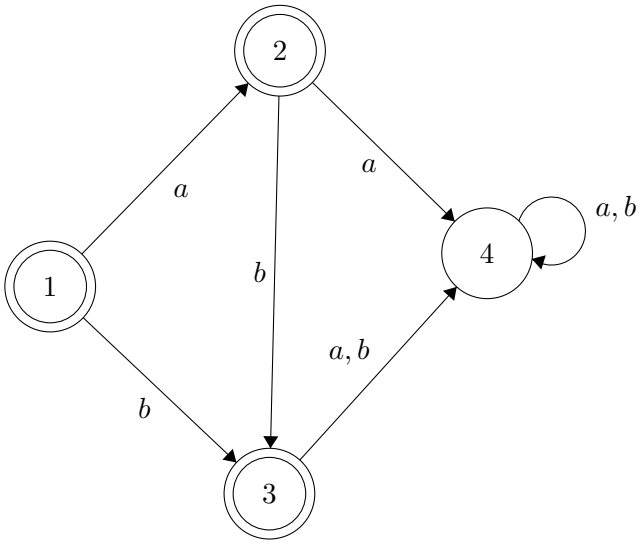
   (a) $L_1 = \{x \in \{0,1,2\}^* \mid x$ represents an odd number in base 3$\}$. For example, 111 represents $1 \cdot 3^2 + 1 \cdot 3^1 + 1 \cdot 3^0 = 13$ in base 3.

Solution:



(b) $L_2 = \{\epsilon, a, b, ab\}$. Treat $\Sigma = \{a, b\}$.

Solution:



(c) You're designing a safe with a digital input for identity verification. The keys consist of all alphanumeric characters (A-Z, a-z, 0-9) as well as 2 special symbols (!, #). It also

includes a NEXT button, but this has a special purpose. In this system, it requires 2 steps of verification consisting of 2 different passwords input into different fields sequentially, both having different requirements. The catch is that when a user first sets their 2 different passwords, they must get it correct in one go. There is no backspace button. **The display input moves on to the next field when the user hits the NEXT button.** Each field has the following requirements:

- **Field 1:** Maximum of 2 characters, must contain at least 1 lowercase alphabet and can contain at MOST 1 digit.
- **Field 2:** ***Minimum*** of 3 characters, can contain at MOST 1 special symbol.

For example, the following strings are valid passwords (although they may not be particularly strong...):

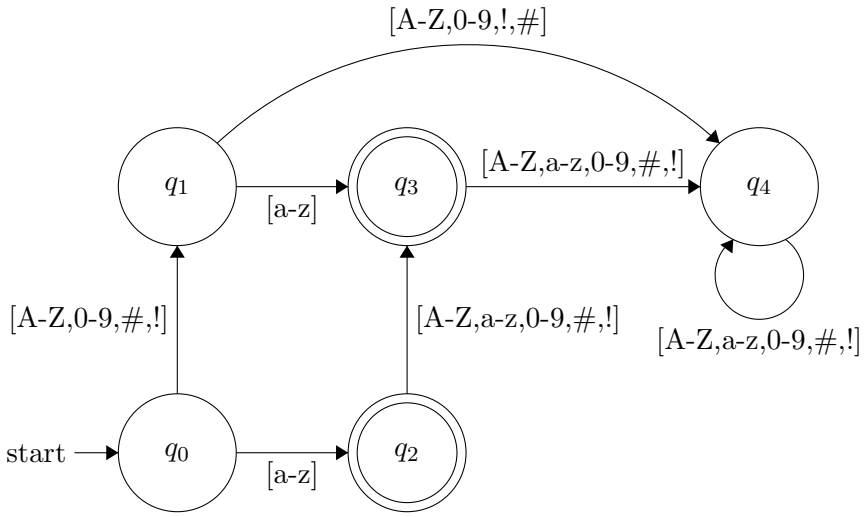
- g0 <NEXT> Blue!
- h4 <NEXT> 1LtoTheVict0rs
- a <NEXT> cde
- @b <NEXT> cde@

The following strings are NOT valid passwords (... indicates the string carries on but we don't care about the rest of it):

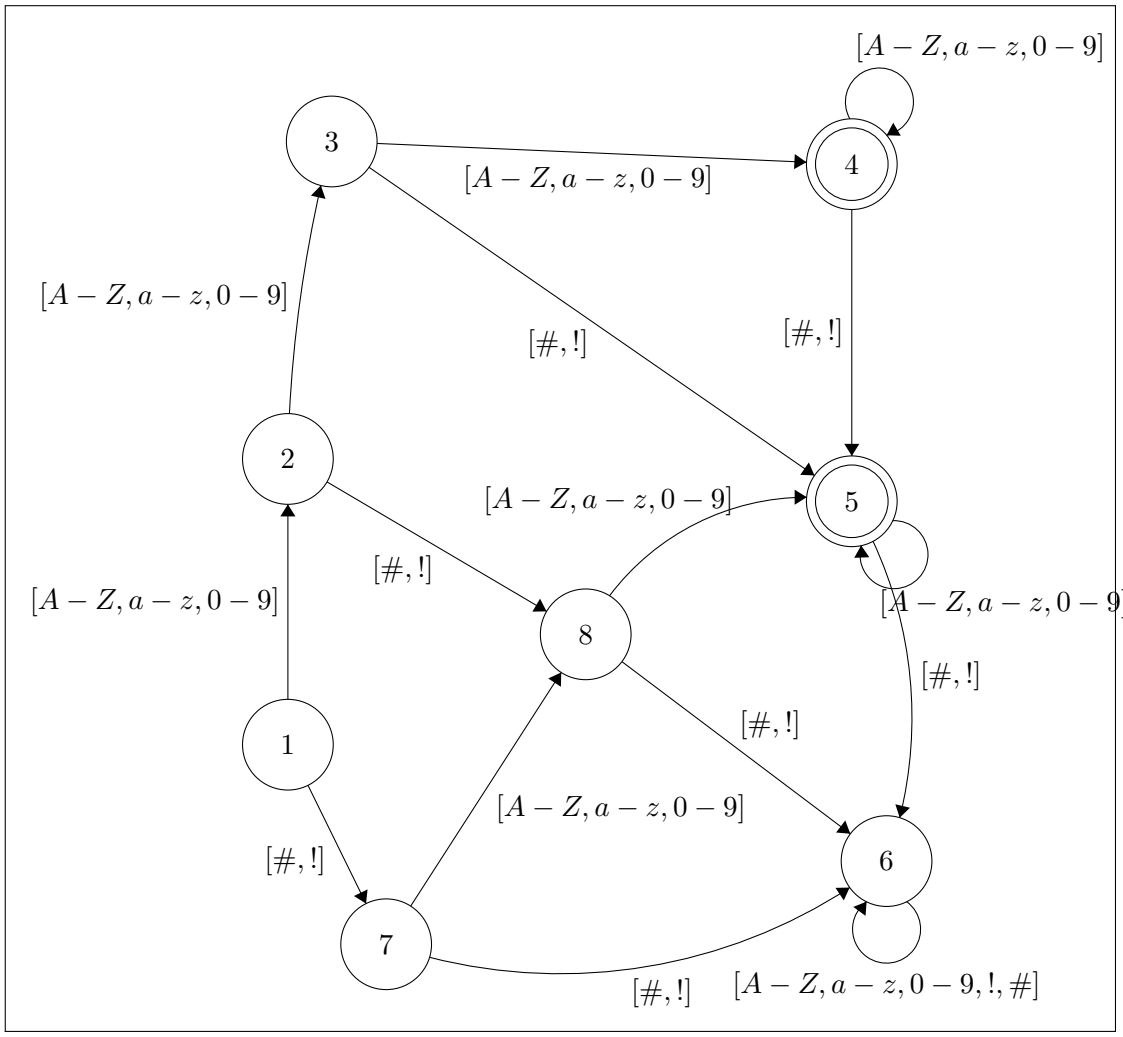
- 11 <NEXT> ... (> 1 digit, no lowercase alphabet in Field 1)
- 1A <NEXT> ... (no lowercase alphabet in Field 1)
- 1aa <NEXT> ... (> 2 characters in Field 1)
- 1z <NEXT> @@@ <NEXT> (> 1 special symbol in Field 2)

Construct a DFA for verifying **Field 2**. **You are permitted to use at most TWO accept states.** Treat the NEXT button as extrinsic to your DFA. In other words, your DFA **does not** need to account for when the NEXT button is hit, it is not part of the alphabet. The machine will know what state the DFA is in and accept or reject accordingly. We have constructed the DFA for Field 1 as an example.

Field 1 DFA



Solution:



6. (10) **Turing Machines**. Complete the following definitions relating to Turing Machines.

   (a) In the definition of a Turing Machine, the _____ alphabet is a subset of the _____ alphabet, and contains all the valid characters or symbols that can be fed into a Turing Machine. In the 7-tuple definition of a Turing Machine, this alphabet is represented with the symbol ___.

   Solution: Blank 1: input
   Blank 2: tape
   Blank 3: Σ

   (b) The _____ alphabet is any character that can be read and/or written by a Turing machine. It must include a symbol not found in the _____ alphabet which we call 'blank' and represent as ___. In the 7-tuple definition of a Turing Machine, this alphabet is represented with the symbol ___.

Solution: Blank 1: tape
Blank 2: input
Blank 3: $\perp$
Blank 4: Γ

(c) A _____ is a set of elements that share a common specified property. These elements are composed of the _____ alphabet.

Solution: Blank 1: language
Blank 2: input

(d) A _____ for language L is a Turing Machine that _____ all inputs that are in L and _____ all inputs not in L. Deciders can never ____.

Solution: Blank 1: decider
Blank 2: accepts
Blank 3: rejects
Blank 4: reach an infinite loop