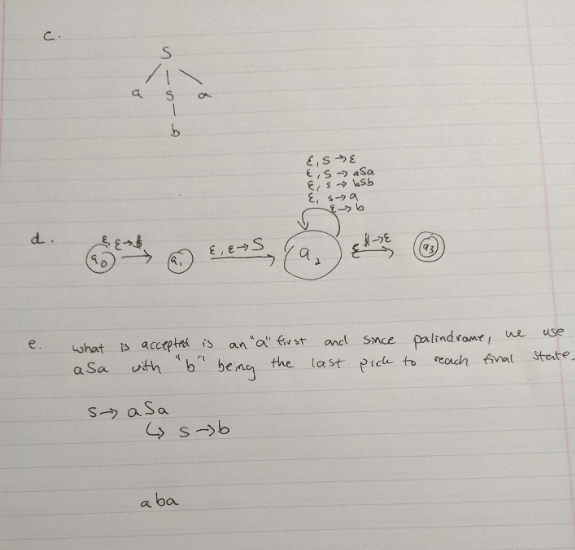
1. Push Down Automata
   1. Aba,abba,abbba
   2. S 🡪 aSa | bSb | a | b | Epsilon



1. Turing acceptance, decider
   1. An example would be a Turing decider that accepts numbers but a Turing accepter that only accepts odd or even numbers. For example, if x is odd then

x = 2x + 1

Otherwise, if x is even

x = x/2

* 1. **Turing Acceptable** : A language L is acceptable if it is recognizing L, it will only accept strings that are in the language L and nothing else. For instance, it can be decided if we can accept “1” but not the letter “a”. The shorter definition of Turing acceptability would be that it MAY OR MAY NOT halt when strings are not in the language decided by the decider

**Turing Decidable** : L is Turing decidable if there is some decider “x” that decides L. Its purpose is to reject strings not in the language. The shorter version of this is just that

* 1. Computer scientists should be interested because these unsolvable problems are relatively simple problems and reasonable, but you just can’t solve it. Furthermore, it lets us know that there are algorithms that don’t yet exist for the problem which that in itself is significant. Instead of giving a binary solution like if an answer is solvable or unsolvable, we are not trying to guess a possible algorithm for a solution for optimization.

1. Universal Turing machine
   1. A Universal Turing machine is a Turing machine that is capable of running other Turing machines in the programs. Basically, the input of a Universal Turing Machine is a Turing machine and an input for the Turing machine and the Universal Turing machine will simulate that Turing machine input. It is an interpreter of Turing machines
   2. We consider this concept because it is literally our computers. Our computer is just a big Universal Turing Machine capable of running other Turing machines. All Universal Turing Machines is, is a way to simulate all the other Turing machines, compiling them all together.
2. PCP
   1. PCP is used to test if a question is undecidable or not and its done where you are given two lists X and Y where for some concatenation of the elements in X and the concatenation of the elements in Y result in X = Y such that the # of elements in both X and Y are equal. (x1…xn = y1 … yn)

For example

X = (a, c b) and Y = (a, b c) [elements are x1,yn … xn,yn]

x1x3x2 = y1y2y3 = ‘abc’

* 1. PCP is useful in proving in a CFG is undecidable or not because CFG’s can be very ambiguous such that there is a left parse tree that can output the same output as a right parse tree. Thinking of this as doing things in different order. The PCP can help test to see if the CFG is decidable or not because they will either be equal to each other after concatenation or undecidable if they are not equal.

7. Zero Knowledge Proof

**a.** ZKP is the proof in which there are two sides to a proof and one side can prove to the other side that something is true without revealing any information regarding the situation other than that the “something” is true. This is very important in cryptography because no key or password will be shared and it is extremely secure, it is used in blockchain.

**b.** an example of ZKP would be to say, if Alice had two coins, one in the left hand and one in the right hand both with different faces. Bob needs to determine if she switched then and so he does when Alice reveals her hands a second time after showing Bob her hands the first time.

8. P, NP

**a.**

**P** is something that can be done in polynomial time. In our class, P would be some language L in a DETERMINISTIC Turing machine that runs in polynomial time. P problem can be something like counting the pieces of broken dish, easy to do.

**NP** is some problem that can be solved in polynomial time using a NON-DETERMINISTIC Turing machine, akin to a choice function. Also, NP problems are problems seem difficult to do, like trying to piece together a broken dish.

**NP Complete** is some problem that is NP and can be tested that the problem can be solved in polynomial time using some algorithm. However, NP complete could also mean that it can be an NP Hard problem which implies that there is a very likely chance that there is no polynomial-time to find an optimal solution.

**b.** The Hamiltonian Cycle problem is an example of an NP complete problem such that for some graph we can find the cycle of it in some time T, but what if we were to add a vertex or edge to the graph? The graph may or may not be solvable now or even a take some time T that is much larger than without the extra vertex or edge

**c.** It is the controversy of whether or not if a problem whose problem can be certified quickly, can be solved quickly. If proved, it would show that problems that seem to be extremely difficult (NP) and indeed problems that can be solved easily (P). This is huge because it will change the way we solve problems by applying the same methods of solving them to other NP-complete problems

9. TSP

**a.** TSP is you are given a list of destinations say [A-Z] and you have to find the shortest route to connect A…Z. You can solve the problem sure, but how do you know if it is indeed the shortest route? What is the algorithm used to find the shortest path? There is none or not yet discovered. This is an NP-complete problem because it’s easy to explain but is difficult to compute, with varying complexity dependent on destinations.

**b.** To develop an Approx. Algo, we’d need to know what the optimal cost is when traversing from points A to Z and when we know the costs, we could use the minimum spanning tree algorithm to determine which path seems to be the most optimal