Classes of Computational Complexity

Step 1

Now that we have the ability to determine algorithm time complexity in our arsenal of skills, we have found a way to take an arbitrary algorithm and describe its performance as we scale its input size. Note that, however, algorithms are simply *solutions* to *computational problems*. A single computational problem can have numerous algorithms as solutions. As a simple example, say our computational problem is the following: "Given a vector containing *n* integers, return the largest element". Below is one algorithmic solution:

```
int getMax1(vector<int> vec) {
    int max = vec[0];
    for(int i : vec) {
        if(i > max) {
            max = i;
        }
    }
    return max;
}
```

Below is a second algorithmic solution that is equally correct, but that is clearly less efficient (and more convoluted):

```
int getMax2(vector<int> vec) {
    for(int i : vec) {
        bool best = true;
        for(int j : vec) {
            if(i < j) {
                best = False;
                break;
            }
        }
        if(best) {
            return i;
        }
    }
}
```

As we can deduce, the first algorithm has a time complexity of O(n), whereas the second algorithm has a time complexity of O(n^2). We were able to describe the two *algorithms* (getMax1 and getMax2), but is there some way we can describe the *computational problem* itself?

In this section, we will be discussing the main classes of computational problems: **P**, **NP**, **NP-Hard**, and **NP-Complete**.

Step 2

First, we will discuss the **P** class of computational problems. Any computational problem that can be *solved* in *polynomial time* (or better, of course) is considered a member of **P**. In other words, given a computational problem, if there exists a polynomial time algorithm that solves the problem—where we would need to formally prove that the algorithm does indeed always provide an optimal solution to the problem—we would classify the problem as class **P**.

For example, let's define the "Oldest Person Problem" as follows:

- **Input:** A list of people *population*, represented as (Name, Age) pairs
- **Output:** The name of the oldest person in *population* (if there is a tie, return the names of all such people)

Somewhat intuitively, we can come up with the following algorithm to solve the "Oldest Person Problem":

```
OldestPerson(population):
    oldestAge = -1
    oldestNames = empty list of strings
    for each person in population:
        if person.age > oldestAge:
            oldestAge = person.age
            clear oldestNames
            add person.name to oldestNames
        else if person.age == oldestAge:
            add person.name to oldestNames
    return oldestNames
```

As you hopefully inferred, if *population* has *n* individuals, this algorithm has a worst-case time complexity of O(*n*), which is polynomial time. Therefore, since we have found a polynomial-time solution to the "Oldest Person Problem", we can say that the "Oldest Person Problem" belongs in problem class **P**.

Step 3

Next, we will discuss the **NP** (***N***ondeterministic ***P***olynomial time) class of computational problems. Any computational problem where, given a proposed answer to the problem, one can *verify* the answer for correctness in polynomial time is considered a member of **NP**. Note, however, that although we need to be able to *verify* an answer in polynomial time, we do not necessarily need to be able to *compute* a correct answer in polynomial time (i.e., it is not necessary for there to exist a polynomial-time algorithm that solves the problem optimally).

We mentioned in the previous step that **P** is the class of computational problems that can be *solved* in polynomial time. If **NP** is the class of computational problems that can be *verified* in polynomial time (whether or not you can solve them in polynomial time), it should be clear that **P is a subset of NP**: if we can *solve* a problem in polynomial time, then we can certainly *verify* a proposed answer to the problem in polynomial time (we can just solve it in polynomial time and then compare the proposed answer to our answer).

Let's discuss a problem that is a member of **NP** (i.e., a proposed answer can be *verified* in polynomial time) but is *not* a member of **P** (i.e., no polynomial-time solution exists at the moment). The "Subset Sum Problem" is defined as follows:

- **Input:** A set of integers *set*
- **Output:** A non-empty subset of *set* whose sum is 0

You can try to think of a polynomial-time algorithm that solves the "Subset Sum Problem", but we can assure that your efforts will likely be futile (because if you *did* find such an algorithm, you would have solved one of the Millenium Prize Problems and would receive a $1,000,000 prize). However, if we were to give you a set of integers *set* and a proposed answer *subset*, you would be able to check the correctness of *subset* in polynomial time:

```
CheckSubset(set, subset):
    // verify that subset is actually a valid non-empty subset of set
    if subset is empty:
        return False
    for each element of subset:
        if element does not appear in set:
            return False

    // verify that the elements of subset add up to 0
    sum = 0
    for each element of subset:
        sum = sum + subset
    if sum == 0:
        return True
    else:
        return False
```

Step 4

EXERCISE BREAK: Which of the following statements are true? (Check all that apply)

To solve this problem please visit https://stepik.org/lesson/30026/step/4

Step 5

The third class of computational problems we will discuss is **NP-Hard** (***N***ondeterministic ***P***olynomial-time ***hard***). Describing **NP-Hard** problems is a bit trickier because we want to avoid going too deep in the topic; but basically, a problem can be considered **NP-Hard** if it is *at least* as hard as the hardest problems in **NP**. More precisely, a problem *H* is **NP-Hard** when every problem *L* in **NP** can be "reduced", or transformed, to problem *H* in polynomial time. As a result, if someone were to find a polynomial-time algorithm to solve any **NP-Hard** problem, this would give polynomial-time algorithms for all problems in **NP**.

The "Subset Sum Problem" described previously is an example of an **NP-Hard** problem. There's no "clean" way to come to this realization on your own without taking a complexity theory course, so take our word that it is (an explanation would go well out of the scope of this text).

Step 6

The last class of computational problems we will discuss is **NP-Complete**, which is simply the intersection between **NP** and **NP-Hard**. In other words, an **NP-Hard** problem is considered **NP-Complete** if it can be verified in polynomial time (i.e., it is also in **NP**).

One interesting **NP-Complete** problem is the "Boolean Satisfiability Problem", which is the basis of modern encryption. When we encrypt sensitive data, we feel safe because, without knowing the encryption key, a malicious person would need to solve the "Boolean Satisfiability Problem" to be able to forcefully decrypt our data, which is unfeasible because of the hardness of the problem.

STOP and Think: The "Boolean Satisfiability Problem" is the basis of modern encryption, and it asks the following question: "Given some arbitrary Boolean formula, does there exist a set of values for the variables in the formula such that the formula is satisfied?" For example, if we were given the Boolean formula `x AND NOT y`, does there exist a set of values for `x` and `y` that satisfies the formula? In this small example, we can see there does indeed a solution that satisfies this formula (`x = TRUE` and `y = FALSE`), but is there way of algorithmically determining if a solution exists for some arbitrary Boolean formula?

The "Boolean Satisfiability Problem" is **NP-Complete**, meaning it is both **NP-Hard** *and* **NP**. If someone were to find a polynomial-time solution to *any* **NP-Hard** problem (not necessarily the "Boolean Satisfiability Problem"), what would be the repercussions to data encryption, if any?

Step 7

EXERCISE BREAK: Which of the following statements are true? (Check all that apply)

To solve this problem please visit https://stepik.org/lesson/30026/step/7

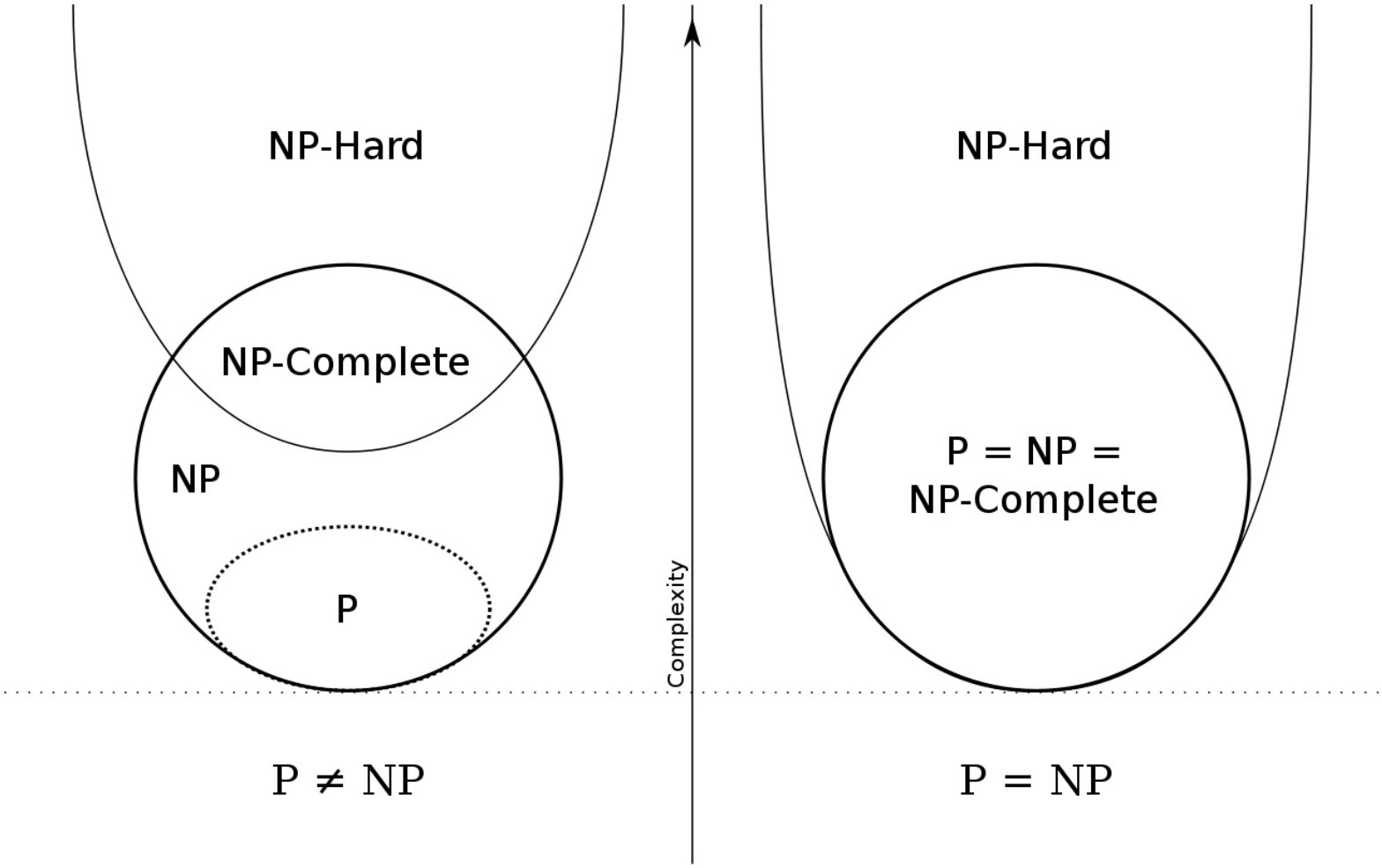
Step 8

When we introduced **P** and **NP**, we mentioned that **P** is a subset of **NP**. What if, however, **P** wasn't just a subset of **NP**? What if **P** and **NP** were actually equal sets? This question of whether or not **P** equals **NP** is known as the "**P** vs. **NP** Problem", which is a major unsolved problem in the field of computer science. Informally, it asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.

Formally, if it can be mathematically proven that **P = NP**, then it would imply *all* problems that can be verified in polynomial time can also be solved in polynomial time. Thus, if there is no known algorithmic solution to a problem that can be verified in polynomial time, we can conclude that a solution *must* exist, and humans just have not found the solution yet.

On the other side, if it can be mathematically proven that **P ≠ NP**, then it would imply that there indeed exists a set of problems that can be verified in polynomial time but that cannot be solved in polynomial time. If we run into a problem in **NP** for which we cannot find a polynomial time solution, it very well could be that no such solution exists.

Below is a diagram representing the relationship between **P**, **NP**, **NP-Hard**, and **NP-Complete** under both possible scenarios regarding the equality (or lack thereof) of **P** and **NP**.



Step 9

EXERCISE BREAK: Which of the following statements are known to be true? (Check all that apply)

To solve this problem please visit https://stepik.org/lesson/30026/step/9

Step 10

As computer scientists, our entire lives are focused on solving computational problems. Even if you choose to tackle problems in other fields (e.g. Biology, Economics, Neuroscience, etc.), you will take the original non-computational problem and model it as a formal computational problem formulation. Your goal will then be to solve the computational problem, and the solution you receive will help give you insight about the original non-computational problem.

As such, it is important to be aware of how to classify computational problems, and understanding the different classes of computational problems will help you predict how fruitful it would be to try to solve the problems you encounter. For example, say you work as an analyst for a sales company, and your boss hands you a list of cities and the distances between each pair of cities, and he asks you to find the shortest possible route that visits each city exactly once and returns to the origin city. You might spend hours, days, or even years trying to find a polynomial-time solution to the problem, but had you taken a step back and thought about the problem, you would have realized that this problem is exactly the "Traveling Salesman Problem", which is **NP-Complete**. Thus, you would have immediately realized that, although possible, it is extremely unlikely that you would be able to find a polynomial-time solution to the problem because of its **NP-Complete** status.

In general, when you face a computational problem, if you can deduce that the problem is not in class **P** (and you are unable to simplify the problem to make it part of class **P**), you typically are forced to choose between one of two options:

- If the input size you are dealing with is small enough, a non-polynomial-time solution may work fine for you
- If the input size is too large for a non-polynomial-time solution, you can try to create a polynomial-time "heuristic" (i.e., an algorithm that isn't guaranteed to give the globally optimal solution, but that does a pretty good job coming close to optimality, hopefully "good enough" for your purposes)

We hope that, now that you have learned about the classes of computational problems, you will be able to think about the problem before blindly jumping in to try to solve it to hopefully save yourself some time and effort.