# Course Description

**Weekly Overview**

This week presents, in light of previous material, the first real limitation of a computer. This is a practical limitation, not a theoretical limitation. The idea is that some things *can* be coded, but their best-known algorithms are too slow to be useful. Students will learn about the categories of P, NP, and NP Complete, and understand that this is the number one unsolved problem in computer science. This looks ahead to heuristics (what do you do if you cannot write an algorithm because either (a) you don’t have the information, or (b) it will take too long). Finally, students will program an NP Complete problem to demonstrate that (1) the algorithm is not difficult, but (2) it takes way too long to run for large cases.

# Institutional Learning Outcomes

**Main Objectives**

* Describe the categories of P, NP, and NP Complete
* Program an NP Complete Problem.
* Extrapolate how long it would take to compute the solution to the NP Complete problem for large cases.

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Introduction to P/NP

Day 2: NP Complete

Day 3: NP Complete Lab

Day 4: NP Complete Lab

Day 5: Test

**Included Resources**

Lecture Notes: Introduction to P/NP

Lecture Notes: NP Complete Problems

Lab Assignment: Programming an NP Complete Problem

Java Class: Timer.java

**Lecture Notes: Introduction to P and NP**

**Bell Work (5 minutes)**

What is the *O*(*n*) of the following code?

for(int i = 0; i < n; i++){

System.out.println(i);

i = i / 2;

}

***Answer:*** O(*n*) = log2(*n*)

**Main Lecture: Part 1 (10 minutes)**

*N.B. Much of this material is taken from* [*https://introcs.cs.princeton.edu/java/55intractability/*](https://introcs.cs.princeton.edu/java/55intractability/)*.*

We have been working with computational complexity to understand the “speed” of algorithms. There are a few important things to remember about complexity analysis:

1. We are really only concerned about large values of *n*. For example, while a *O*(*n*) =2*n*2 algorithm might be preferable to a *O*(*n*) = 9000*n* algorithm for small values of *n*, because of the speed at which processors operate, for these small values, the time is virtually indistinguishable for most applications. For very large values of *n*, the *O*(*n*) =2*n*2 algorithm is much slower.
2. There is something particularly devious about a algorithm that runs in exponential time. The values of 2*n* grow so astoundingly fast for large values of *n* that processor speed *cannot* actually keep up with it. This places an inherent limit on the power of computation. The problem may be theoretically “solvable”, but in practice, the solution would take thousands of years to compute.

Perhaps the biggest unsolved problems in computer science deals with the relationship between problems that have known “slow” solutions and problems that have “reasonably” fat solutions.

**Definition:** Any algorithm whose running time is *bounded* by a polynomial in the input size (e.g., *n*log(*n*) or *n*2) as a ***polynomial-time* *algorithm***.

**Definition:** A problem is ***intractable***if there is no polynomial-time algorithm for the problem.

As programmers became more experienced with the way in which hardware increases in speed and the amount of data in “real world” problems, it became clear that problems that can be solved in polynomial time are useful, whereas exponential solutions are not useful.

Some problems are inherently and obviously exponential problems. For example, to print out all numbers 1 through 2*n* will take *O*(*n*) = 2*n* time. There is no way to speed this up.

Other problems have exponential solutions, but it is not obvious that this *must* be the case. Many of these fall into the category of *Search Problems*. This is not the same are our “searching algorithms” from a previous unit. A *search problem* is one in which we are looking for a solution form a large set of possible solutions.

*Example:* Find the greatest common division of two integers *a* and *b*. This amounts to finding the solution among the set of all common divisors.

We are now ready to define the two most important classes of problems on computer science.

**Definition:** The complexity class ***P*** is the set of all search problems solvable in polynomial-time (on a deterministic Turing machine).

Examples of ***P*** problems include the following:

1. Find the greatest common divisor of two numbers.
2. Find a permutation of numbers that puts the numbers in order, i.e. sort the numbers.
3. Find a solution to a system of linear equations.

While the algorithms may in some cases be difficult to solve, these problems are all “easy”, or rather algorithms exist to solve the problems in a “reasonable” (polynomial) amount of time.

The more interesting category is ***NP***.

**Definition:** A problem is ***NP*** if a particular solution can be *checked* in polynomial time.

Some things here are very important to note. First, every problem that is ***P*** is also ***NP*.** In other words, ***P*** is a subset of ***NP***. This is obvious, actually. If a solution can be *produced* in polynomial time, it can in fact be checked in polynomial time. We could simply produce the solution and then check it (with a simple if statement) against the particular solution we are questioning.

The much more interesting question is: **are there any problems that are *NP* but not *P*?**

In other words, is there a problem that can be checked in polynomial time, but not solved in polynomial time?

Let’s have an example:

**Example:** Given a set of numbers, split the set into two different subsets such that the sum of the two subsets is the same?

[Give student the set {4, 5, 8, 13, 15, 24, 33}. Ask them to split this into two different sets (using every number exactly once) such that the two sums are the same. Answer: {5, 13, 33} and {4, 8, 15, 24}, with the sum of each being 51.]

**IN CLASS EXERCISE (15 minutes):**

Write the method:

public static boolean isSolution(int[] mainSet, int[] subset\_1, int[] subset\_2){

//first check that the two subsets are a valid partition

//next check that the sums are the same

}

**Main Lecture – Part 2 (15 minutes)**

While the students are working, put the following pseudo code on the board:

public static SetPair findSolution(int[] main\_Set){

Set firstSet, secondSet;

for(int i = 0; i < 2^mainSet.length; i++){

n = binaryRepresentation(i);

// At this point, n looks like 00010011110000, etc.

for(int j = 0; j < length(n); j++){

if(n.charAt(j) == 1)

firstSet.add(mainSet[j]);

}

for(int k = 0; k < main\_Set.length; k++){

if(firstSet does not contain main\_et[k])

secondSet.add(mainSet[k]);

}

if(sum(firstSet) == sum(secondSet)

return true;

}

return false;

}

After they have generates their “isSolution” code, engage the following conversation.

1. What is the *O*(*n*) of your “isSolution” method? (Here, *n* is the size of the mainSet.) [Answer: *n*2 with the checking that the two sets are a valid partition, *n* otherwise]
2. (After talking through the pseudo code above … ) What is the *O*(*n*) of the findSolution? [Answer: 2*n*]

So, this problem of partitioning is clearly in the class ***NP***. Why? (Because a solution can be checked in polynomial time, e.g. isSolution.)

Is the problem in ***P***? This is a major point: ***we don’t know***. We have a solution that runs in exponential time, and we do not have any solution that runs in polynomial time. So we *cannot* say that it *is* in ***P***. However, maybe there is another solution out there that *does* run in polynomial time.

So the task at hand is either:

1. Find a polynomial time solution, or
2. Prove that, by its very nature (like printing out number 1 through 2*n*), this problem can never be solved in polynomial time.

So what is the answer? ***We don’t know***. Computer scientists do no know this answer. They have not found a polynomial time solution, but no one has been able to prove that it is impossible.

In fact, the astonishing this is … we don’t know if there are *any* problems that are ***NP*** but not ***P***. For every single “hard” ***NP*** problem (one for which we know an exponential, or in some cases factorial, solution, but for which we do not know a polynomial solution), computer sciences have not been able to find a polynomial solution, nor have they been able to prove that a polynomial solution is impossible.

This is the ***biggest*** unsolved problem in computer science.

**Unsolved Problem: *Does* P *=* NP*?***

As a point of note, the encryption systems most countries use in some way depend on the inability to factor large numbers efficiently. Factoring *n* is an **NP** problem and not known to be **P**. If **P = NP**, then an algorithm exists (but has not yet been found) to factor these large numbers efficiently, which is bad news for our security systems.

The good news is, while this is an unsolved problem, most computer scientist believe that **P *≠* NP**.

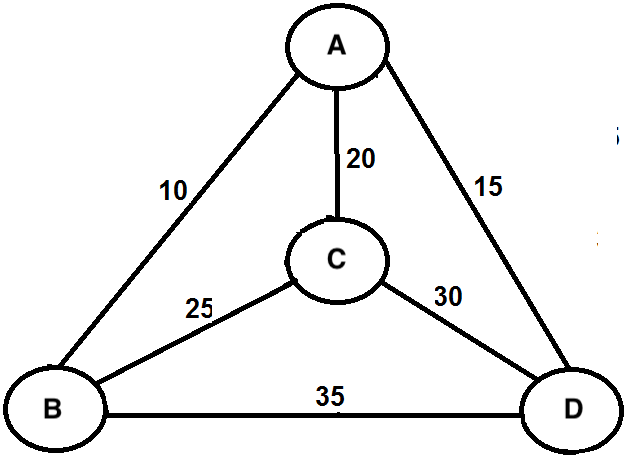
Historical Note: Godel's letter to von Neumann anticipated the **P = NP** question. He recognized that if **P = NP**, it "would have consequences of the greatest importance" since then "the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine." He asked for which combinatorial problems was there a more efficient alternative to exhaustive search.

**Homework:** None

**Lecture Notes: NP Complete Problems**

**Bell Work (10 minutes)**

Draw the following on the board:



Write the following question:

*The letters represent the locations that a salesman needs to visit, and the numbers represent the distance in between these locations. The salesman wants to start and end in the same location, but wants to minimize the total distance. What path should the salesman take?*

**Main Lecture: Part 1 (20 minutes)**

*N.B. Much of this material is taken from* [*https://introcs.cs.princeton.edu/java/55intractability/*](https://introcs.cs.princeton.edu/java/55intractability/)*.*

Yesterday we defined the classes **P** and **NP**, and noted that the biggest open question in computer science is whether or not **P = NP**.

While this is indeed unsolved, the path towards solving it is made quite a bit easier by a rather remarkable fact. There is a subset of **NP** problems known as **NP-complete** problems.

**I**nformally, **NP-complete** problems are the "hardest" problems in **NP**; they are the ones most likely to not be in P.

Formally,

**Definition:** A problem is ***NP-complete***if

1. it is in NP and
2. every problem in **NP** “polynomially” reduces to it.

The second point is the trickiest: it states that we can transform every **NP** problem into the supposed “**NP-complete**” problems, but of course to do so in polynomial time. A non-polynomial reduction is not so useful, but a polynomial reduction is incredibly useful. If such a thing exists, then the ramifications are intense. It means that the entire class of **NP** is reducible to this *one* **NP-complete problem.**

Defining the concept of **NP-completeness** does not mean that such problems exist. In fact, the existence of **NP-complete** problems is an amazing thing. We cannot prove a problem is **NP-complete** by presenting a reduction from each **NP** problem since there are infinitely many of them. It wasn’t until the 1960s that a problem was first proved as actually **NP-complete**.

The remarkable fact is: if we can solve any **NP-complete**, then we can solve any problem in **NP**.

It is even more amazing that there exist "natural" problems that are **NP-complete**.

The impact of **NP-completeness** on the natural sciences has been undeniable. Once the first **NP-complete** problems were discovered, intractability "spread like a shockwave through the space of problems", first in computer science, and then to other scientific disciplines.

There are least 20 diverse scientific disciplines that were coping with internal questions. Ultimately, scientists discovered their inherent complexity after realizing that their core problems were **NP-complete**.

Few scientific theories have had such a breadth and depth of influence.

The opening Bell Work is an example of one of the more famous **NP-complete** problems, known as the “Travelling Salesman Problem.” It can be checked in polynomial time – it can be solved in exponential time, but whether or not a polynomial-time solution exists is unknown. But if one does, then **P = NP**, and the largest and most important open questions in computer science falls overnight.

So, *you* can begin working on whether or not **P = NP** by trying to either find a polynomial time solution for any *one* **NP-complete** problem, or by demonstrating definitively that it cannot be done.

So what are your choices for **NP-complete** problems?

1. The Traveling Salesman Problem (see bell work)
2. The Subset Sum Problem: Given a set of integers and a number *B*, can you find a subset that adds to *B*?
3. The Partition Problem (from yesterday): Given a set of integers, can you split it into two subsets, each of which has the same sum?
4. The Truth Decision Problem: Given *n* Boolean variables and a logical formula, is there a set of True/False values that will make the formula evaluate to True?
5. The Regular Expression Problem: Give two regular expressions, do they represent the same language?
6. The Battelship Problem: (see <https://en.wikipedia.org/wiki/Battleship_(puzzle>))
7. Sudoku
8. Pancake Sorting: (see https://en.wikipedia.org/wiki/Pancake\_sorting)

**IN CLASS EXERCISE (15 minutes):**

Tomorrow you will be programming and timing either an **NP** or an **NP-complete** problem. It is your choice which to work on.

Option 1: (**NP-complete**) The subset problem

public static boolean hasSubset(int[] mainSet, int b){

//returns True if mainSet has a subset such that the sum of

//the elements in the subset is b, False otherwise

}

Option 2: (**NP, not known to be NP-complete**) The factoring problem

public static int[] factor(int n){

//returns prime factorization of n

//e.g., factor(100) returns [2, 2, 5, 5]

}

Teacher note: The easiest way to generate the set of subsets is to use the binary representation (see previous lecture notes). This first options is considerably more difficult than the factoring problem. In order to help students with the subset, the class java.lang.Integer has methods that will do the binary conversion (<https://docs.oracle.com/javase/7/docs/api/index.html?java/lang/Integer.html>).

**Homework:** None, but potentially finish coding

**NP / NP-Complete Lab Assignment**

1. From yesterday, you wrote either the hasSubset method or the factor method. Write and test this method.
2. Pull down the StopWatch.java code from GitHub. You have seen this code before when you timed your searching and sorting algorithms. Use the class to test how long your method will take on at least 10 large values. (You can use a random number generator to do this.)
3. Plot your data and sketch a graph. Using your graph, come up with a reasonable function, *f*(*n*), that represents the time it takes to factor the number *n*.
4. Use your function to estimate how long it would take to factor a 100 digit long number, what about a 500 digit long number?
5. Submit your code through GitHub, but your graph and calculations on paper.

//Stopwatch.java

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac Stopwatch.java

\* Execution: java Stopwatch n

\* Dependencies: none

\*

\* A utility class to measure the running time (wall clock) of a program.

\*

\* % java8 Stopwatch 100000000

\* 6.666667e+11 0.5820 seconds

\* 6.666667e+11 8.4530 seconds

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* The {@code Stopwatch} data type is for measuring

\* the time that elapses between the start and end of a

\* programming task (wall-clock time).

\*

\* See {@link StopwatchCPU} for a version that measures CPU time.

\*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

public class Stopwatch {

private final long start;

/\*\*

\* Initializes a new stopwatch.

\*/

public Stopwatch() {

start = System.currentTimeMillis();

}

/\*\*

\* Returns the elapsed CPU time (in seconds) since the stopwatch was created.

\*

\* @return elapsed CPU time (in seconds) since the stopwatch was created

\*/

public double elapsedTime() {

long now = System.currentTimeMillis();

return (now - start) / 1000.0;

}

/\*\*

\* Unit tests the {@code Stopwatch} data type.

\* Takes a command-line argument {@code n} and computes the

\* sum of the square roots of the first {@code n} positive integers,

\* first using {@code Math.sqrt()}, then using {@code Math.pow()}.

\* It prints to standard output the sum and the amount of time to

\* compute the sum. Note that the discrete sum can be approximated by

\* an integral - the sum should be approximately 2/3 \* (n^(3/2) - 1).

\*

\* @param args the command-line arguments

\*/

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

// sum of square roots of integers from 1 to n using Math.sqrt(x).

Stopwatch timer1 = new Stopwatch();

double sum1 = 0.0;

for (int i = 1; i <= n; i++) {

sum1 += Math.sqrt(i);

}

double time1 = timer1.elapsedTime();

System.out.println("%e (%.2f seconds)\n" + sum1 + "\n" + time1);

// sum of square roots of integers from 1 to n using Math.pow(x, 0.5).

Stopwatch timer2 = new Stopwatch();

double sum2 = 0.0;

for (int i = 1; i <= n; i++) {

sum2 += Math.pow(i, 0.5);

}

double time2 = timer2.elapsedTime();

System.out.println("%e (%.2f seconds)\n" + sum2 + "\n" + time2);

}

}