Reading for This Lecture

• N&W Sections 1.5.1 and 1.5.2

• Wolsey Chapter 6
Introduction to Computational Complexity

• What is the goal of computational complexity theory?
  – To provide a method of quantifying problem difficulty in an absolute sense.
  – To provide a method comparing the relative difficulty of two different problems.

• We would like to be able to rigorously define the meaning of efficient algorithm.

• Complexity theory is built on a basic set assumptions called the model of computation.

• We will not concern ourselves too much with the details of a particular model here.

• To deal with this topic in full rigor would require a full semester course.
Problems and Instances

• What is the difference between a *problem* and a *problem instance*?

• To define these terms rigorously takes a great deal of mathematical machinery.

• We do so here in the context of mathematical programming.
  – Loosely, a *problem* or *model* is an infinite family of *instances* whose objective function and constraints have a specific structure.
  – An instance is obtained by specifying values for the various problem parameters.

• This is similar to the distinction between *model* and *data* in a modeling language.
Turing Machines

- The complexity framework we use today is based on the concept of a Turing machine proposed by A.M. Turing.

- A Turing machine is what we would think of today as algorithm or, more specifically, a computer program.

- Corresponding to any given algorithm, there is a Turing machine that takes an input and produces an output through a sequence of steps.

- The specific sequence might be different for different inputs, i.e., we have a notion of conditional branching.

- Turing later conceived of something like what we think of as a computer, which was able to load a “program” into its memory.

- This became known as a universal Turing machine.

- These concepts heavily influenced the inventions that lead to modern computer architectures.
Measuring the Difficulty of an Instance

- Loosely speaking, the difficulty of a problem instance is easy to measure.
- We solve the problem instance and see how long it takes (the running time).
- Note that this inherently depends on the algorithm and the computing platform.
- We want a measure independent of both these variables.
  - We will assume a well-specified algorithm is used.
  - We will measure execution time in terms of the total number of elementary operations executed on a given Turing machine.
Measuring the Difficulty of a Problem

• On the previous slide, we discussed how to measure the difficulty of an instance.

• The difficulty of a problem is harder to define.

• We must consider all possible algorithms for solving the problem.

• How do we evaluate the efficiency of an algorithms?
  – Best case running time
  – Average case running time
  – Worst case running time

• **Best case** doesn’t give us any guarantee about the difficulty of a given instance.

• **Average case** is difficult to analyze and depends on specifying a probability distribution on the instances.

• **Worst case** addresses these problems and is usually easier to analyze.

• These measures are defined with respect to a particular algorithm.

• It is still not exactly clear how to compare different algorithms.
The Size of a Problem

- Obviously, the time needed to solve a problem instance with a given algorithm depends on certain properties of the instance.
- One such property is the size of the instance.
- However, it is again problematic to define what we mean by “size”.
- In many cases, the size of an instance can be taken to be the number of input parameters.
- For a general MIP, this would be roughly determined by the number of variables and constraints.
- The running time of certain algorithms, however, depends explicitly on the magnitude of the input data.
Measuring the Size of a Problem

- We will define the *size* of an instance to be the amount of information required to represent the instance.

- This is still not a clear definition because it depends on our representation of the data (the *alphabet*).

- Because computers store numbers in binary format, we use the size of a *binary* encoding (a two symbol alphabet) as our standard measure.

- In other words, the size of a number $l$ is the number of bits required to represent it in binary, i.e., $\log_2 l$.

- As long as the magnitude of the input data is *bounded*, this is equivalent to considering the number of input parameters.

- In practice, the magnitude of the input data is *usually*, but not always, bounded.
More on the Size of a Problem

• Note that many combinatorial problems are defined *implicitly*, i.e., independent of a particular formulation.

• An example of this is the Euclidean Traveling Salesman Problem.

• The input data for an instance of the TSP is simply the coordinates of each customer location.

• Hence, the size of an instance is determined by the number of locations (assuming the magnitude of the coordinates is bounded).

• If we formulate the TSP as an integer program, the size of the input to the solver will be larger.

• This is because it includes a distance matrix that can be computed from the coordinates.
Measuring Running Time

• **Running time** is the measure we will use to judge the efficiency of an algorithm.

• We will consider the worst case running time over all instances as a function of the size of the instance.

• In most cases, worst case running time depends primarily on the size of the instance.

• However, we still need a measure of running time that is architecture independent.

• To make the running time architecture independent, we will simply count the number of **elementary operations** required (on a given Turing machine).
Elementary Operations

- *Elementary operations* are very loosely defined to be additions, subtractions, multiplications, comparisons, etc.

- In most cases, we will assume that each of these can be performed in constant time.

- Again, this is a good assumption as long as the size of the numbers remains “small” as the calculation progresses.

- Generally we will want to ensure that the numbers can be encoded in a size polynomial in the size of the input.

- This justifies our assumption about constant time operations.

- We will see later, we may have to be very careful about checking this assumption.
Asymptotic Analysis

- So far, we have determined that our measure of running time will be a function of instance size (a positive integer).
- Determining the exact function is still problematic at best.
- We will only really be interested in approximately how quickly the function grows “in the limit”.
- To determine this, we will use asymptotic analysis.
- Order relations
  \[
  f(n) \in O(g(n)) \iff \exists c \in \mathbb{R}_+, n_0 \in \mathbb{Z}_+ \text{ s.t. } f(n) \leq cg(n) \forall n \geq n_0.
  \]
- In this case, we say \( f \) is order \( g \) or \( f \) is ‘big O’ of \( g \) or also \( f \) and \( g \) are of the same order.
- Using this relation, we can divide functions into classes that are all of the same order.
Order Relations

- For polynomials, the order relation from the previous slide can be used to divide the set of functions into equivalence classes.
- We will only be concerned with what equivalence class the function belongs to.
- Note that class membership is invariant under multiplication by scalars and addition of “low-order” terms.
- For polynomials, the class is determined by the largest exponent on any of the variables.
- For example, all functions of the form $f(n) = an^2 + bn + c$ are $O(n^2)$. 
Running Time and Complexity

• **Running time** is a measure of the efficiency of an algorithm.

• **Computational complexity** is a measure of the difficulty of a problem.

• The computational complexity of a problem is the running time of the best possible algorithm.

• In most cases, we cannot prove that the best known algorithm is the also the best possible algorithm.

• We can therefore only provide an upper bound on the computational complexity in most cases.

• That is why complexity is usually expressed using “big O” notation.

• A case in which we know the exact complexity is comparison-based sorting, but this is unusual.
Aside: Space Complexity

• So far, we have discussed only the amount of computing time required to solve a problem.

• The amount of memory required to execute a given algorithm may also be an issue.

• This is known as space complexity.

• We can analyze space complexity in an analogous manner.
Comparing Algorithms

- So far, we have defined complexity as a tool for comparing the difficulty of two different problems.
- This machinery can also be used to compare two algorithms for the same problem.
- In this way, we can judge whether one algorithm is “better” than another one.
- Note that worst case analysis is far from perfect for this job.
- The simplex algorithm has an exponential worst case running time, but does extremely well in practice.
Polynomial Time Algorithms

- Algorithms whose running time is bounded by a polynomial function are called *polynomial time algorithms*.

- For the purposes of this class, we will call an algorithm *efficient* if it is polynomial time.

- Problems for which a polynomial time algorithm exists are called *polynomially solvable*.

- The class of all problems which are known to be polynomially solvable occupies a special place in optimization theory.

- For most interesting problems, it is not known whether or not there is a polynomial algorithm.

- This is one of the great unsolved problems in mathematics.

- If you can solve it, the American Mathematical Society will give you one million dollars and you will become instantly famous.

- We’ll come back to this.
Problems Solvable in Polynomial Time

- **Shortest path problem with nonnegative weights:** $O(m^2)$.
  - Note that the number of operations is independent of the magnitude of the edge weights.

- **Solving a system of equations:** $O(n^3)$.
  - Note that the magnitude of the numbers that occur is bounded by the largest determinant of any square submatrix of $(A, b)$.
  - Since $\det A$ involves $n! < n^n$ terms, this largest number is bounded by $(n\theta)^n$, where $\theta$ is the largest entry of $(A, b)$.
  - This means that the size of their representation is bounded by a polynomial function of $n$ and $\log \theta$.

- **Minimum weight spanning tree problem:** $O(\min\{m \log n, m + n \log n\})$

- **Assignment Problem:** $O(\min\{n(m + n \log n), n^3\})$
The Case of Linear Programming

- General linear programming is polynomially solvable.
- Note, however, that the simplex algorithm is *not* polynomial time!
- In practice, the expected running time *is* polynomial.
- A polynomial-time algorithm (the ellipsoid method) for LP was not found until 1979!
- Although this algorithm has not had a big practical impact, it’s theoretical impact has been large.
- This is one of the biggest cases against using worst-case analysis.
Pseudopolynomial Time Algorithms

• A pseudopolynomial algorithm is one that is polynomial in the length of the data when encoded in unary.

• Unary means that we are using a one-symbol alphabet.

• Hence, to store an integer $k$, we would need $k$ symbols.

• Example: The Integer Knapsack Problem
  – There is an $O(nb)$ algorithm for this problem, where $n$ is the number of items and $b$ is the size of the knapsack.
  – This is not a polynomial time algorithm in general.
  – If $b$ is bounded by a polynomial function of $n$, then it is.
  – However, it is pseudopolynomial.