References for Today’s Lecture

• Required reading
  – Miller and Boxer, Chapter 1

• References
  – AMO Sections 3.2
  – CLRS Sections 1.1–1.3
Algorithms

• algorithm\(^1\)
  1. any systematic method of solving a certain kind of problem
  2. a predetermined set of instructions for solving a specific problem in a limited number of steps
• The concept of an algorithm is not new but formal study of efficiency is relatively new.

\(^1\text{Wester's New World Dictionary}\)
Introduction to Computational Complexity

• What is the goal of computational complexity theory?
  – To provide a method of comparing the difficulty of two different problems.
  – To provide a method of comparing the efficiency of two different algorithms for the same problem.

• 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.

• This topic is addressed in IE 407.
Elementary Operations

• In order to analyze the number of steps necessary to execute an algorithm, we have to say what we mean by a “step.”

• To define this precisely is tedious and beyond the scope of this course.

• A precise definition depends on the exact hardware being used.

• Our analysis will assume a very simple model of a computer in which the following operations take one step.
  
  – arithmetic (addition, subtraction, multiplication, division)
  – data movement (read from memory, store in memory, copy)
  – comparison
  – control (function calls, goto commands)

• This is a very idealized model, but it works in practice.

• We will sometimes need to simplify the model even further.
Problems, Instances, and Algorithms

• A problem $P$ is a mapping of a set of inputs to specified outputs.

• An instance is a problem along with a particular input.

• An algorithm is a procedure for computing the output expected from a given input.

• An algorithm solves a problem $P$ if that algorithm produces the expected output for any input.

• Example: Traveling Salesman Problem

  – Given an undirected graph $G = (N, A)$ and non-negative arc lengths $d_{ij}$ for all $(i, j) \in A$, find a cycle that visits all nodes exactly once and is of minimum total length.
  – How do we specify an instance?
Computational Complexity: What is the Objective?

- Complexity analysis is aimed at answering two types of questions.
  - How hard is a given problem?
  - How efficient is a given algorithm for a given problem?

- Our measure of efficiency will be *running time*, defined as either
  - The actual wall clock time required to execute the algorithm on a computer (problematic) or
  - the number of elementary operations required (more on this later).

- The running time may differ by instance, algorithm, and computing platform.

- How should we measure the performance so that we can select the “best” algorithm from among several?
What Do We Measure?

Three methods of analysis:

- **Empirical analysis**
  - Try to determine how algorithms behave in practice on real computational platforms under load in real-world conditions.

- **Average-case analysis**
  - Try to determine the expected running time an algorithm will take analytically.

- **Worst-case analysis**
  - Provide an upper bound on the running time of an algorithm for any instance in a given set.
## Drawbacks of Three Approaches

<table>
<thead>
<tr>
<th>Method</th>
<th>Drawbacks</th>
</tr>
</thead>
</table>
| **Empirical**   | 1. Depends on programming language, compiler, etc.  
|                 | 2. Time consuming and expensive                                           |
|                 | 3. Often inconclusive                                                     |
| **Average-Case**| 1. Depends on probability distribution                                     |
|                 | 2. Difficult to determine appropriate distribution                        |
|                 | 3. Intricate mathematical analysis                                        |
|                 | 4. No information on distribution of outcomes                             |
| **Worst-Case**  | 1. Influenced by pathological instances                                   |
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.
- The most easily identifiable 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 linear program, 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 an Instance

• Formally, we consider the size of the input to be the amount of memory it takes to store a complete description of the instance in memory.

• 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 Traveling Salesman Problem.

• The input data for an instance of the TSP may be either
  – an explicit a *vector of costs* for traveling between pairs of locations or
  – explicit coordinates of each location, with the costs being implicitly defined as Euclidean distances.

• Hence, the size of an instance may be either the number of locations or the number of costs specified between pairs of locations.

• The magnitude of the costs may also affect the size (if this is not bounded).
The Running Time of an Algorithm

- **Running time** is a measure of efficiency for an algorithm.
- For a given instance of a problem, we can determine (roughly) the time required to solve it with a given implementation on a given computing platform.
- Worst-case running time with respect to a given set of instances is the maximum time required over all instances.
- In most cases, worst case running time depends primarily on the size of the instances, as we have defined it.
- Therefore, our measure will typically be the worst-case running time over all instances of a given size.
- However, we still need a measure of running time that is architecture independent.
- We will simply count the number of elementary operations required to perform the algorithm.
More on Elementary Operations

- *Elementary operations* are the basic operation defined earlier.

- 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.

- In some cases, 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.

- We will allow some “sloppiness” and ignore constants and low order terms.

- Because of our many simplifying assumptions, the low order terms may not be accurate anyway.
Growth of Functions

• **Question**: Why are we *really* interested in the theoretical running times of algorithms?

• **Answer**: To compare different algorithm for solving the same problem.

• We are interested in performance for large input sizes.

• For this purpose, we need only compare the *asymptotic growth rates* of the running times.

  – Consider algorithm $A$ with running time given by $f$ and algorithm $B$ with running time given by $g$.
  
  – We are interested in knowing

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

  – What are the four possibilities?
Θ Notation

• We now define the set

\[ \Theta(g) = \{ f : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0 \} \]

• If \( f \in \Theta(g) \), then we say that \( f \) and \( g \) grow at the same rate or that they are of the same order.

• Note that

\[ f \in \Theta(g) \iff g \in \Theta(f) \]

• We also know that if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \) for some constant \( c \), then \( f \in \Theta(g) \).

• If the limit doesn’t exist, we don’t know.
Big-O Notation

- We can similarly define the set

\[ O(g) = \{ f : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0 \} \]

- If \( f \in O(g) \), then we say that “\( f \) is big-O of \( g \)” or that \( g \) grows at least as fast as \( f \).

- Note that if \( f \in O(g) \), then either \( f \in \Theta(g) \) or \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)

- Some other notation
  - \( f \in \Omega(g) \iff g \in O(f) \).
  - \( f \in o(g) \iff f \in O(g) \setminus \Theta(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0. \)
  - \( f \in \omega(g) \iff g \in o(f) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty. \)
Example

Let’s show that if \( f(n) = \frac{1}{2}(n^2 + 3n) \), then \( f \in \Theta(n^2) \).
Comparing Functions

• The notation we have just defines gives us a way of ordering functions.

• We can can interpret

  – \( f \in O(g) \) as “\( f \leq g \),”
  – \( f \in \Omega(g) \) as “\( f \geq g \),”
  – \( f \in o(g) \) as “\( f < g \),”
  – \( f \in \omega(g) \) as “\( f > g \),” and
  – \( f \in \Theta(g) \) as “\( f = g \).”

• This gives us a method for comparing algorithms based on their running times.

• Note that most of the relational properties of real numbers (transitivity, reflexivity, symmetry) work here also.

• However, not every pair of functions is comparable.
Commonly Occurring Functions

• **Polynomials**: All polynomials \( f \) of degree \( k \) are in \( \Theta(n^k) \).

• **Exponentials**
  
  – A function in which \( n \) appears as an exponent on a constant is an exponential function, i.e., \( 2^n \).
  
  – For all positive constants \( a \) and \( b \), \( \lim_{n \to \infty} \frac{n^b}{b^a} = 0 \).
  
  – This means that exponential functions always grow faster than polynomials.

• **Logarithms**
  
  – Logarithms of different bases differ only by a constant multiple, so they all grow at the same rate.
  
  – A polylogarithmic function is a function in \( O(lg^k) \).
  
  – Polylogarithmic functions always grow more slowly than polynomials.

• **Factorials**: Factorial functions grow more quickly than exponentials, but are in \( o(n^n) \).
Problem Difficulty

- The *difficulty* of a problem can be judged by the (worst-case) running time of the best-known algorithm.

- Problems for which there is an algorithm with polynomial running time (or better) are called *polynomially solvable*.

- Generally, these problems are considered to be *easy*.

- There are many interesting problems for which it is not known if there is a polynomial-time algorithm.

- These problems are generally considered *difficult*.

- One of the great open questions in mathematics is whether these problems really are difficult or if we just haven’t discovered the right algorithm yet.

- If you answer this question, you can win a *million dollars*.

- In this course, we will stick mostly to the easy problems.
Example

for $i = 1 \cdots p$ do

    for $j = 1 \cdots q$ do

        $c_{ij} = a_{ij} + b_{ij}$

How many elementary operations?
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 $\Theta(n^2)$. 
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.

• This will be important in some cases.
Polynomial Time Algorithms

• An algorithm is said to be *polynomial-time* if its worst-case complexity is bounded by a polynomial function of the input.

• For network problems
  – A *strongly polynomial* algorithm is bounded by a polynomial function that involves only $n$ and $m$.
  – A *weakly polynomial* has a running time that is a function of the size of the whole input, including capacities, etc.

• An algorithm is said to be *exponential-time* if it worst-case complexity grows as a function that cannot be bounded by a polynomial function.

• An algorithm is *pseudopolynomial-time* if its running time is bounded by a polynomial function of the actual values of the inputs parameters, such as the largest arc capacity.
Example: Finding a Simple Path

- How easy is it to determine if there is a path connecting a given pair of vertices in a graph?
- For now, let us consider undirected graphs.
- Using the operations in the Graph class, we can answer this question easily using a recursive algorithm.

```python
def SPath(G, v, w):
    if v == w:
        return True
    for n in G.get_node_list():
        G.set_node_attr(n, 'color', 'red')
    v.set_node_attr('color') = 'green'
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'color') == 'red':
            G.set_node_attr(n, 'color', 'green')
        if SPath(G, n, w):
            return True
    return False
```
Finding a Hamiltonian Path

• Now let’s consider finding a path connecting a given pair of vertices that also visits every other vertex in between (called a Hamiltonian path).

• We can easily modify our previous algorithm to do this by passing an additional parameter \( d \) to track the path length.

• What is the change in running time?

```python
def HPath(G, v, w, d):
    if v == w: return d == 0
    for n in G.get_node_list():
        G.set_node_attr(n, 'color', 'red')
    G.set_node_attr(v, 'color', 'green')
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'color') == 'red':
            G.set_node_attr(n, 'color', 'green')
        if HPath(G, n, w, d-1):
            return true
    G.set_node_attr(v, 'color', 'red')
    return false
```
Worst-Case Complexity of Algorithms

- Dijkstra’s Algorithm $O(n^2)$
- Dial’s Algorithm $O(m + nC)$
- Floyd-Warshall Algorithm $O(n^3)$
- Shortest Augmenting Path Algorithm $O(n^2m)$
- Out-of-Kilter Algorithm $O(nU)$
- Minimum Mean Cycle-Canceling Algorithm $O(n^2m^3\log n)$
- Kruskal’s Algorithm $O(nm)$
Computational Complexity: Activity!

• Compare the following functions for various values of \( n \).

• Determine which function is larger (according to “big O”) and the approximate value of \( n \) after which it is always larger.

  – \( 1000n^2 \) and \( 2^n/100 \)
  – \( n^{0.001} \) and \( (\log n)^3 \)
  – \( 0.1n^2 \) and \( 10000n \)
Computational Complexity: Summary

• (Theoretical) objective is to develop polynomial-time algorithms with smallest possible growth rate
  – Why?

• Need to consider empirical performance because not all polynomial-time algorithms perform better in practice than exponential-time algorithms
  – Classic example?
  – Explanation?

• Will we always be able to find a polynomial-time algorithm for every combinatorial optimization problem?