Reading for this Lecture

- Aho, Hopcroft, and Ullman, Chapter 1
- Miller and Boxer, Chapters 1 and 5
- Fountain, Chapter 4
Problems and Instances

• Roughly, a *problem* specifies what output is desired for each given input.

• In more rigorous mathematical terms, *solving* a problem can be equated to evaluating a (mathematical) *function*.

• A *problem instance* is to evaluate the given function for a specific input (argument).

• An *algorithm* is a procedure for converting the inputs to an output of the desired form (called a *solution*).
Simple Example

• Consider the following simple problem:

  \textbf{Input}: \( x \in \mathbb{R} \) and \( n \in \mathbb{Z} \).
  
  \textbf{Output}: \( x^n \).

• What is the most straightforward algorithm for solving this problem?

• Is there a \textit{better} algorithm?
A More Efficient Algorithm

Let’s assume that $n = 2^m$ for $m \in \mathbb{Z}$. We can use repeated squaring:

```python
def pow(x, n):
    for i in range(log(n, 2)):
        x *= x
```

Questions:

- Is this algorithm correct?
- How much more efficient is it?
- What do we mean by efficiency?
- Why don’t we call it speed?
- How do we modify it for the general case?
Algorithms

• An algorithm is a procedure for evaluating the function associated with a given problem.

• In general, a problem will have more than one algorithm.

• How do we compare?
  – Correctness (accuracy)
  – Efficiency

• An algorithm that is guaranteed to result in a solution for every instance is said to be correct.
  – In some cases, it is not possible to produce the solution exactly (problems with irrational solutions).
  – In such cases, we generally want know how close the approximate solution is to the true solution.
  – This is the domain of numerical analysis, which we will later in the course.
Example: Log Function in Python

```python
>>> math.log10(1000)
3.0
>>> math.log(1000, 10)
2.9999999999999996
```

- Here, we are using two different algorithms for calculating the logarithm base 10 of 1000.
- One is more accurate than the other.
- We will look at issues of floating point error later in the course.
Analyzing Algorithms

• The goal of analyzing algorithms is to determine the resources required to execute the algorithm in practice.

• Generally speaking, \textit{time} and \textit{space} are the two most important resources to consider.

• A simple approach to evaluating performance in practice would be to do an \textit{empirical analysis}.

• Rigorous empirical analyses are more difficult than they appear because of factors that are difficult to take into account.

  – What instances should we use to do the testing?
  – How should we measure performance (CPU time, wallclock time)?
  – How should we take into account performance variability?
  – What platform should we do our testing on?

• Even more challenges arise in the analysis of parallel algorithms.
Theoretical Analysis

• Formal theoretical analysis of algorithms attempts to abstract away some of these issues.
  – We consider a formal model of a computer called the *model of computation*.
  – We use summary measures that account for performance across all instances.
  – We explicitly account for the dependence of efficiency on important properties of individual instances.

• The result of the most common type of analysis is a *running time* function.

• This function represents execution time as a function of properties of the instance (typically *size*).
Computational Complexity

• What is computational complexity?
  – With respect to a *mathematical problem*, complexity theory provides a rigorous framework for assessing difficulty (absolute and relative).
  – With respect to different *algorithms* for the same problem, complexity theory provides a framework for assessing and comparing efficiency.

• To rigorously define the meaning of *efficient*, we need a basic set assumptions called the *model of computation*. 
Models of Computation

• The model of computation is a conceptual model of how a computer works on which we can base a formal analysis of algorithms.

• We would like our model to come as close as possible to representing the realities discussed in the first four lectures.

• Unfortunately, it is not possible for a useful conceptual model to take into account all practical details.

• The model must be simple enough to allow proof of certain mathematical results based on assumptions of the model.

• Well-known models do not take into account details of the hardware like the memory hierarchy.
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 essentially specifies an algorithm or, more specifically, a program, but the concept was invented before computers existed.

• 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.
The RAM Model of a Computer
Execution of a Program

• In each time step, one instruction is executed.

• The instructions are similar to those in a machine language.

• Each instruction consists of an operation and an address (input to the operation)

• The input to the operation can either be the location of the next instruction, a memory location or a constant operand.

• The input to the RAM program is read sequentially and the output is written sequentially.
Assumptions of the RAM model

• The program is not stored in memory and hence cannot be modified.
• The instance is small enough to fit in the memory.
• All numbers fit into one computer “word” (Important).
• All fundamental operations can be performed in one unit of time.
• Any memory location can be accessed in one unit of time.
• This is what is known as a "unit cost model".
Running Time

• The number of basic steps required for an algorithm to solve a given instance of a problem is called the *running time* for that instance.

• The running time can be different for different models of computation.

• For the types of analysis we’ll do, the details will not usually matter, at least for sequential algorithms.

**Proposition 1.** *Sequential Computation Thesis:* All ”reasonable" models are polynomially equivalent.

• The assumptions of the model allow us to do rigorous analysis.

• It is possible to abuse the assumptions of the model.

• For example, log cost models explicitly take into account the size of the numbers.
Alphabets and Languages

• The simplest kind of RAM program is one that reads an input string and outputs either TRUE or FALSE.

• The associated problem is called a decision problem.

• We say that a given input string is accepted by the program if the output for that string is TRUE.

• The characters that are allowed in forming the input string are called the alphabet.

• The set of strings that are accepted by a program P is called the language accepted by P.
Example: Evaluating a Polynomial

• Consider the problem of evaluating a polynomial.

  **Input**: The coefficients $a_0, \ldots, a_n$ and $x \in \mathbb{R}$.
  **Output**: The value $\sum_{i=0}^{n} a_i x^i$.

• What is the most obvious way of computing the output for a given input?

• Is there a “better” way?
Time and Space Complexity

- The *time complexity* of an algorithm is the number of time steps needed to execute it over a (possibly infinite) set of instances.

- The *space complexity* is the number of registers required to execute the algorithm.

- We need a summary measure in order to compare the complexity of different algorithms.

- Some possibilities are
  - Worst case
  - Average case
  - Best case

- What are the advantages/disadvantages of each of these?
Size of an Instance

• It is clear that 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.”

• The size of an instance is defined to be the amount of memory required to store a description of the instance.

• The size may be different, depending on assumptions of our model of computation.
  – In the basic unit cost model, the size of the instance is the number of distinct input parameters.
  – We may need to explicitly account for the magnitude of the input data at times.

• Complexity of an algorithm is usually expressed as a function $f$ that maps the size $n$ of the input to the worst-case running time.
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.
The Basic PRAM model
Analysis of Parallel Algorithms

• The analysis of parallel algorithms is more difficult.

• The assumptions of the model make a much bigger difference.

• It is no longer true that all reasonable models are polynomially equivalent.
Assumptions of the PRAM model

- This is a synchronous model with shared memory.
- There are a fixed number of processors (bounded).
- All processors execute the same program, but each one can be in a different place.
- At each time step, each processor performs one read, one elementary operation, and one write.
- Memory access is performed in constant time.
- Processors are not linked directly.
- Communication issues are not considered.
Concurrent Memory Access

• What if two processors try to read/write to/from the same memory location in the same time step?

• We have to resolve these conflicts.

• Four possible models:
  – CREW ⇐ We will use this one (most of the time)
  – CRCW
  – EREW
  – ERCW
Assessment of the PRAM Model(s)

- This model is not as "robust" as the RAM model.
- However, it allows us to do rigorous analysis.
- It is a reasonable model of a small parallel machine.
- It is not "scalable."
- It does not model distributed memory or interconnection networks.
- How do we fix it?
Distributed PRAM Model

- Attempt to model the interconnection network.
- Eliminate global memory.
- Each processor can read or write only from its neighbors’ registers.
- This will likely increase the complexity of many algorithms, but is more realistic and scalable.
Aside: Introduction to Graphs

• A graph \( G = (V, E) \) is defined by two sets, a finite, nonempty set \( V \) of vertices (or nodes) and a set \( E \subseteq V \times V \) of edges.

• Example: A road network.

• The edges can be either ordered pairs or unordered pairs.

• If the edges are ordered pairs, then they are usually called arcs and the graph is called a directed graph.

• Otherwise, the graph is called undirected.

• See AHU, Section 2.3
(Undirected) Graph Terms

- Vertices $u$ and $v$ are endpoints of the edge $(u, v)$.
- We say an edge $e = (u, v)$ is incident to its endpoints.
- Two vertices $u$ and $v$ are adjacent if $(u, v) \in E$.
- The degree of a vertex is the number of edges incident to it (equivalently, the number of vertices adjacent to it).
- A path is a sequence of edges $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$
- The length of such a path is $n - 1$.
- Often, we represent a path simply as a sequence of vertices.
Applications of Graph Theory

• Graph theory is a very rich subject area.

• Sample Applications
  – Shortest Path Problem
  – Minimum Spanning Tree
  – Traveling Salesman Problem
What is an interconnection network?

- A graph (directed or undirected)
- The nodes are the processors
- The edges represent direct connections
- Properties and Terms
  - Degree of the Network
  - Communication Diameter
  - Bisection Width
  - Processor Neighborhood
  - Connectivity Matrix
  - Adjacency Matrix
Measures of Goodness

- **Communication diameter**: The maximum shortest path between two processors.
- **Bisection width**: The minimum cut such that the two resulting sets of processors have the same order of magnitude.

- Connectivity Matrix
- Adjacency Matrix
Bottlenecks

• The communication diameter indicates how long it may take to send information from one processor to another.

• Thus, it may be the bottleneck in any algorithm in which the data are initially distributed equally.

• The bisection width is the bottleneck when processors must exchange large amounts of information.

• The bisection width is a lower bound for sorting.
## Connectivity Matrix: Example 1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Connectivity Matrix: Example 2
### 2-step Connectivity Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>![Cell 0,0]</td>
<td>![Cell 0,1]</td>
<td>![Cell 0,2]</td>
<td>![Cell 0,3]</td>
</tr>
<tr>
<td>1</td>
<td>![Cell 1,0]</td>
<td>![Cell 1,1]</td>
<td>![Cell 1,2]</td>
<td>![Cell 1,3]</td>
</tr>
<tr>
<td>2</td>
<td>![Cell 2,0]</td>
<td>![Cell 2,1]</td>
<td>![Cell 2,2]</td>
<td>![Cell 2,3]</td>
</tr>
<tr>
<td>3</td>
<td>![Cell 3,0]</td>
<td>![Cell 3,1]</td>
<td>![Cell 3,2]</td>
<td>![Cell 3,3]</td>
</tr>
</tbody>
</table>
**N-step Connectivity Matrices**

- Indicates the processor pairs that can reach each other in N steps
- Computed using Boolean matrix multiplication
- The corresponding adjacency matrix indicates how many disjoint paths connect each pair.
Linear Array
Ring
Mesh
Tree
Other Schemes

- **Pyramid**: A 4-ary tree where each level is connected as a mesh

- **Hypercube**: Two processors are connected if and only if their ID #'s differ in exactly one bit.
  - Low communications diameter
  - High bisection width
  - Doesn’t have constant degree

- **Perfect Shuffle**: Processor i is connected one-way to processor $2i \mod N - 1$.

- **Others**: Star, De Bruijn, Delta, Omega, Butterfly