Learning Objectives

- Understand what computation and algorithm are
- Be able to compare alternative algorithms for a problem w. r. t their complexity
- Given an algorithm, be able to describe its big O complexity
- Be able to explain NP-complete
The “Writer” Automaton

http://www.youtube.com/watch?v=FUa7oBsaSDk8

Computational Complexity

- Why complexity?
- Primarily for evaluating difficulty in scaling up a problem
  - How will the problem grow as resources increase?
- Knowing if a claimed solution to a problem is optimal (best)
- Optimal (best) in what sense?
Why do we have to deal with this?

- Moore’s law
- Hwang’s law
- Growth of information and information resources
- Storing stuff
- Finding stuff

Moore’s law: Empirical observation that the transistor density of integrated circuits (IC) doubles every 24 (or 18) months

Hwang’s law: doubles every year

Impact

- The efficiency of algorithms/methods
- The inherent "difficulty" of problems of practical and/or theoretical importance

A major discovery in the science was that computational problems can vary tremendously in the effort required to solve them precisely. The technical term for a hard problem is "_________________" which essentially means: "abandon all hope of finding an efficient algorithm for the exact (and sometimes approximate) solution of this problem".
Optimality

- A solution to a problem is sometimes stated as “optimal”
- Optimal in what sense?
  - Empirically?
  - Theoretically? (the only real definition)

You’re given a problem

- Data informatics: How will our system scale if we store everybody’s email every day and we ask everyone to send a picture of themselves and to everyone else. We will also hire a new employee each week for a year

- Social informatics: We want to know how the number of social connections in a growing community increase over time.
We will use algorithms

- **Algorithm**: a sequence of finite instructions for doing some task
  - unambiguous and simple to follow
- **Eg, Euclid’s algorithm to determine the maximum common divisor of two integers greater than 1**
  - Subtract the smaller number from the larger one
  - Repeat until you get 0 or 1

Eg. Sorting Algorithms

- **Problem**: Sort a set of numbers in ascending order
  - [1 7 3 5 9]
- **Many variations of sorting algorithms**
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - …
Scenarios

- I’ve got two algorithms that accomplish the same task
  - Which is better?
- I want to store some data
  - How do my storage needs scale as more data is stored
- Given an algorithm, can I determine how long it will take to run?
  - Input is unknown
  - Don’t want to trace all possible paths of execution
- For different input, can I determine how system cost will change?

Measuring the Growth of Work

While it is possible to measure the work done by an algorithm for a given set of input, we need a way to:

- Measure the rate of growth of an algorithm or problem based upon the size of the input
- Compare algorithms or methods to determine which is better for the situation
Basic unit of the growth of work

- Input N
- Describing the relationship as a function $f(n)$
  - $f(N)$: the number of steps required by an algorithm

Why Input Size N?

- What are some of the factors that affect computation time?
  - Speed of the CPU
  - Choice of programming languages
  - Size of the input
- A test case shows a specific mapping from between these factors and the computation time
- Since an algorithm can be implemented in different machines, and different programming languages, we are interested in comparing algorithms using factors that are not affected by implementations.
- Hence, we are interested in how “size of input” affects computation time.
Comparing the growth of work

Time vs. Space

Very often, we can trade space for time:

For example: maintain a collection of students’ with SSN information.

- Use an array of a billion elements and have immediate access (better time)
- Use an array of number of students and have to search (better space)
Introducing Big O Notation

- Used in a sense to put algorithms and problems into families
  - Will allow us to evaluate algorithms and growth of problems.
  - Has precise mathematical definition
- Q: How long does it take from S.C. to Chicago if you drive?
  - Less than 11 months
  - More than 10 minutes

Size of Input (measure of work)

- In analyzing rate of growth based upon size of input, we’ll use a variable
- Why?
  - For each factor in the size, use a new variable
  - $N$ is most common…

Examples:
- A linked list of $N$ elements
- A 2D array of $N \times M$ elements
- A Binary Search Tree of $P$ elements
**Formal Definition of Big-O**

For a given function \( g(n) \), \( O(g(n)) \) is defined to be the set of functions

\[
O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}
\]

\( f(n) = O(g(n)) \) means that \( f(n) \) is **no worse** than the function \( g(n) \) when \( n \) is large. That is, **asymptotic upper bound**

---

**Visual O( ) Meaning**

- **Upper Bound**: \( cg(n) \)
- **Our Algorithm**: \( f(n) \)
- Size of input
- Work done

\( f(n) = O(g(n)) \)
Simplifying O( ) Answers

We say Big O complexity of

\[ 3n^2 + 2 = O(n^2) \quad \text{← drop constants!} \]

because we can show that there is a $n_0$ and a $c$ such that:

\[ 0 \leq 3n^2 + 2 \leq cn^2 \text{ for } n \geq n_0 \]

i.e. $c = 4$ and $n_0 = 2$ yields:

\[ 0 \leq 3n^2 + 2 \leq 4n^2 \text{ for } n \geq 2 \]

Correct but Meaningless

You could say

\[ 3n^2 + 2 = O(n^6) \quad \text{or} \quad 3n^2 + 2 = O(n^7) \]

But this is like answering:

- How long does it take to drive to Chicago?
  - Less than 11 years.
Comparing Algorithms

- Now that we know the formal definition of $O(\ )$ notation (and what it means)...
- If we can determine the $O(\ )$ of algorithms and problems...
  - This establishes the worst they perform.
- Thus now we can compare them and see which has the “better” performance.

Classes of Big O Functions

- $O(1)$: constant
- $O(\log \log n)$: double logarithmic
- $O(\log n)$: logarithmic
- $O((\log n)^c), c>1$: polylogarithmic
- $O(n)$: linear
- $O(n \log n) = O(\log n!)$: quasilinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^c), c>1$: polynomial
- $O(c^n)$: exponential (or geometric)
- $O(n!)$: combinatorial
- $c^{O(c^n)}$: double exponential
Correctly Interpreting $O(\cdot)$

**O(1) or “Order One”**
- Does not mean that it takes only one operation
- Does mean that the work doesn’t change as $N$ changes
- Is notation for “constant work”

**O(N) or “Order N”**
- Does not mean that it takes $N$ operations
- Does mean that the work changes in a way that is proportional to $N$
- Is a notation for “work grows at a linear rate”

Complex/Combined Factors

- Algorithms and problems typically consist of a sequence of logical steps/sections
- We need a way to analyze these more complex algorithms…
- It’s easy – analyze the sections and then combine them!
Eg: Insert into Linked List

- Task: Insert an element into an “ordered” list...
  - Find the right location
  - Do the steps to create the node and add it to the list

```
head → 17 → 38 → 142 → //
```

Step 1: find the location = O(N)

Inserting 75

Step 2: Do the node insertion = O(1)
Combine the Analysis

- Find the right location = $O(N)$
- Insert Node = $O(1)$

- Sequential, so add:
  - $O(N) + O(1) = O(N + 1) = O(N)$

Eg: Search a 2D Array

- Task: Search an “unsorted” 2D array (row, then column)
  - Traverse all rows
  - For each row, examine all the cells (changing columns)
Eg: Search a 2D Array

- Search an unsorted 2D array (row, then column)
  - Traverse all rows
  - For each row, examine all the cells (changing columns)

```
Row
1 2 3 4 5 6 7 8 9 10
```

```
Column
1 2 3 4 5 6 7 8 9 10
```

Combine the Analysis

- Traverse rows = \(O(N)\)
  - Examine all cells in row = \(O(M)\)

- Embedded, so multiply:
  - \(O(N) \times O(M) = O(N \times M)\)
Sequential Steps

- If steps appear sequentially (one after another), then **add** their respective $O()$.  

\[
\begin{align*}
\text{loop} & \quad \{ \text{...} \} & \quad \text{N} \\
\text{endloop} \\
\text{loop} & \quad \{ \text{...} \} & \quad \text{M} \\
\text{endloop} \\
\end{align*}
\]

$O(N + M)$

Embedded Steps

- If steps appear embedded (one inside another), then **multiply** their respective $O()$.  

\[
\begin{align*}
\text{loop} \quad \{ \text{loop} \{ \text{...} \} & \quad \text{M} \} & \quad \text{N} \\
\text{endloop} & \quad \} \\
\text{endloop} \\
\end{align*}
\]

$O(N*M)$
Correctly Determining $O(\cdot)$

- Can have multiple factors:
  - $O(N \times M)$
  - $O(\log P + N^2)$
- But keep only the dominant factors:
  - $O(N + N\log N) \rightarrow O(N\log N)$
  - $O(N\times M + P) \rightarrow O(N\times M)$
  - $O(V^2 + V\log V) \rightarrow O(V^2)$
- Drop constants:
  - $O(2N + 3N^2) \rightarrow O(N + N^2) \rightarrow O(N^2)$

Summary

- We use $O()$ notation to discuss the rate at which the work of an algorithm grows with respect to the size of the input.
- $O()$ is an upper bound, so only keep dominant terms and drop constants
Poly-time vs. expo-time

Such algorithms with running times of orders $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ etc. are called *polynomial-time algorithms*.

On the other hand, algorithms with complexities which cannot be bounded by polynomial functions are called *exponential-time algorithms*. These include "exploding-growth" orders which do not contain exponential factors, like $n!$.

---

The Towers of Hanoi

Goal: Move stack of rings to another peg
- Rule 1: May move only 1 ring at a time
- Rule 2: May never have larger ring on top of smaller ring
Towers of Hanoi: Solution

For 3 rings we have 7 operations.

In general, the cost is $2^N - 1 = O(2^N)$

Each time we increment $N$, we double the amount of work.

This grows incredibly fast!
Towers of Hanoi ($2^N$) Runtime

For $N = 64$

$$2^N = 2^{64} = 18,450,000,000,000,000,000$$

If we had a computer that could execute a billion instructions per second…

- It would take 584 years to complete

But it could get worse…

The case of 4 or more pegs: Open Problem

The Traveling Salesman Problem

- A traveling salesman wants to visit a number of cities and then return to the starting point. Of course he wants to save time and energy, so he wants to determine the shortest path for his trip.

- We can represent the cities and the distances between them by a weighted, complete, undirected graph.

- The problem then is to find the round-trip of minimum total weight that visits each vertex exactly once.
The Traveling Salesman Problem

**Example:** What path would the traveling salesman take to visit the following cities?

**Solution:** The shortest path is Boston, New York, Chicago, Toronto, Boston (2,000 miles).

The Traveling Salesman Problem

- Traveling Salesman Problem (TSP): one of the best-known computational problems that is known to be “extremely hard to solve”
  - NP-hard
- A brute force solution takes $O(n!)$
- A dynamic programming solution takes $O(c^n)$, $c>2$
- Harder to solve than Tower of Hanoi problem
Where Does this Leave Us?

- Clearly algorithms have varying runtimes or storage costs.
- We’d like a way to categorize them:
  - Reasonable, so it may be useful
  - Unreasonable, so why bother running

Performance Categories

<table>
<thead>
<tr>
<th>Polynomial</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-linear</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Quasilinear</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(N^2)$</td>
</tr>
</tbody>
</table>

Exponential

- $O(2^N)$
- $O(N!)$
- $O(N^N)$
Two Categories of Algorithms

Reasonable vs. Unreasonable

- **Reasonable** algorithms feature polynomial factors in their $O(\cdot)$ and may be usable depending upon input size.
  - $O(\log N), O(N), O(N^K)$ where $K$ is a constant

- **Unreasonable** algorithms feature exponential factors in their $O(\cdot)$ and have little practical utility
  - $O(2^N), O(N!), O(N^N)$
### Eg: Computational Complexity

Give the Big O complexity in terms of \( n \) of each expression below and order the following as to increasing complexity. (all unspecified terms are to be determined constants)

<table>
<thead>
<tr>
<th>Expression</th>
<th>( O(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 1000 + 7 ( n )</td>
<td>n</td>
</tr>
<tr>
<td>b. 6 + .001 log ( n )</td>
<td>log ( n )</td>
</tr>
<tr>
<td>c. 3 ( n^2 ) log ( n ) + 21 ( n^2 )</td>
<td>( n^2 ) log ( n )</td>
</tr>
<tr>
<td>d. ( n ) log ( n ) + .01 ( n^2 )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>e. 8n! + 2(^n)</td>
<td>n!</td>
</tr>
<tr>
<td>f. 10 ( k^n )</td>
<td>( k^n )</td>
</tr>
<tr>
<td>g. a log ( n ) + 3 ( n^3 )</td>
<td>( n^3 )</td>
</tr>
<tr>
<td>h. ( b ) ( 2^n ) + 10(^8) ( n^2 )</td>
<td>( 2^n )</td>
</tr>
<tr>
<td>i. A ( n^n )</td>
<td>( n^n )</td>
</tr>
</tbody>
</table>

Order (from most complex to least):

- \( \#7 \) \( n \)
- \( \#8 \) log \( n \)
- \( \#5 \) \( n^2 \) log \( n \)
- \( \#6 \) \( n^2 \)
- \( \#1 \) n!
- \( \#2 \) or \( \#3 \) (by \( k \)) \( k^n \)
- \( \#4 \) \( n^3 \)
- \( \#2 \) or \( \#3 \) (by \( k \)) \( 2^n \)
- \( \#1 \) \( n^n \)

### Decidable vs. Undecidable

- **Any problem that can be solved by an algorithm is called _Decidable_**.
- Problems that can be solved in polynomial time are called _tractable_ (easy).
- Problems that can be solved, but for which no polynomial time solutions are known are called intractable (hard).
- Problems that can not be solved given any amount of time are called _undecidable_.

- **Decidable**: Any problem that can be solved by an algorithm is called _decidable_.
- **Tractable**: Problems that can be solved in polynomial time are called _tractable_ (easy).
- **Intractable**: Problems that can be solved, but for which no polynomial time solutions are known are called _intractable_ (hard).
- **Undecidable**: Problems that can not be solved given any amount of time are called _undecidable_.

---

- **Decidability**: Any problem that can be solved by an algorithm is called **decidable**.
- **Tractability**: Problems that can be solved in polynomial time are called **tractable** (easy).
- **Intractability**: Problems that can be solved, but for which no polynomial time solutions are known are called **intractable** (hard).
- **Undecidability**: Problems that can not be solved given any amount of time are called **undecidable**.
Undecidable Problems

- No algorithmic solution exists
  - Regardless of cost
  - These problems aren’t computable
  - No answer can be obtained in finite amount of time

Complexity Classes

- Problems have been grouped into classes based on the most efficient algorithms for solving the problems:
  - Class P: those problems that are solvable in polynomial time.
  - Class NP: problems that are “verifiable” in polynomial time (i.e., given the solution, we can verify in polynomial time if the solution is correct or not.)
NP-complete Problems

If problem A is NP-complete, and problem B can be transformed to problem A in a polynomial time algorithm, then problem B is also NP-complete.

Examples: Traveling Sales Problems

The Challenge Remains ...

- If one finds a polynomial time algorithm for ONE NP-complete problem, then all NP-complete problems have polynomial time algorithms.
- A lot of smart computer scientists have tried, non have succeeded.
- So, people believe NP-complete problems are Very Very Very .... Hard problems w/o polynomial time solutions
What is a good algorithm?

If the algorithm has a running time that is a polynomial function of the size of the input, N, otherwise it is a “bad” algorithm.

A problem is considered tractable if it has a polynomial time solution and intractable if it does not.

For many problems we still do not know if they are tractable or not.

What’s this good for anyway?

- Knowing hardness of problems lets us know when an optimal solution can exist.
  - Salesman can’t sell you an optimal solution
- Keeps us from seeking optimal solutions when none exist, use heuristics instead.
  - Some software/solutions used because they scale well.
- Helps us scale up problems as a function of resources.
- Many interesting problems are very hard (NP-hard)!
  - Use heuristic solutions
- Only appropriate when problems have to scale.
Reference

- Many contents in this slide are modified from Prof. J. Yen and L. Giles’ IST 511 slides