ICS 311 Topic #1: Introduction to Algorithms

Outline

1. Syllabus
2. Algorithms and Programs
3. Algorithm Design & Analysis
4. Computational Complexity
5. Abstract Data Types

Syllabus

1. General Course Information
2. Topic Overview
3. Format
4. Podcasts
5. Assessment (Grading)
6. Assignments
7. Policies

Algorithms and Programs

By now you have a working idea of what a "program" is because you have written many. Programs are particular instructions that work on specific machines.

In this course we work with an abstraction of programs: algorithms.

Algorithms

Informally, an algorithm is a well-defined computational procedure that takes some value(s) as input and produces some value(s) as output.

Somewhat more formally, an algorithm is a finite sequence of instructions chosen from a finite, fixed set of instructions, where the sequence of instructions satisfies the following criteria:

- **Input:** It has zero or more input parameters.
- **Output:** It produces at least one output parameter.
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** For every input, it executes only a finite number of instructions (it eventually halts).
- **Effectiveness:** Every instruction must be sufficiently basic so that a machine can execute the instruction.

Discussion: What is the difference between an algorithm and a program?

Discussion: What kinds of algorithms do you think are needed when you use your smartphone (or any mobile phone for that matter?)

Algorithm Design & Analysis

Here is how algorithm design is situated within the phases of problem solving in software development:

Phase 1: **Formulation of the Problem** (Requirements Specification)
To understand a real problem, **model it mathematically**, and specify input and output of the problem clearly.

Phase 2: **Design and Analysis of an Algorithm** for the Problem (our focus)

- Step 1: **Specification** of an Algorithm - **what is it?**
- Step 2: **Verification** of the Algorithm - **is it correct?**
- Step 3: **Analysis** of the Algorithms - **what is its time and space complexity?**

Phase 3: **Implementation of the Algorithm**
Design data structures and realize the algorithm as executable code for a targeted platform (lower level abstraction).

Phase 4: **Performance Evaluation of the Implementation** (Testing)
The predicted performance of the algorithm can be evaluated/verified by empirical testing.

Algorithms are often specified in **pseudocode**, a mixture of programming language control structures, mathematical notation, and natural language written to be as unambiguous as possible. Examples will be given in the next lecture. In this course, we will use mostly the notation used in the book.

Why not just test programs?

Why not just run experimental studies on programs? We can implement the algorithms of interest, run them on a modern computer on various input sizes, and compare the results. Why bother with all this math in the book?

I find the math painful too, but there are three major limitations to experimental studies:

- To run experiments, you have to implement and run the algorithm. Implementation takes time, and some of these runs may take a long time.
- Experiments can only be done on a limited set of test inputs. Are you sure your results generalize to all possible inputs?
- It is difficult to compare the efficency of tests run on one hardware and software environment to what will happen on others. Are you sure that your results generalize across platforms?

Formal analysis of algorithms:

- Can be performed on a high-level description of the algorithm without implementation.
- Takes into account all possible inputs.
- Allows comparisons of algorithms independently of hardware and software.

So, there is a good reason ICS 311 is THE central course of the ICS curriculum! Stick with us.

Computational Complexity

Input Size

The computational complexity of an algorithm generally depends on the amount of information given as input to the algorithm.

This amount can be formally defined as the **number of bits** needed to represent the input information with a reasonable, non-redundant coding scheme.

To simplify things, we often analyze algorithms in terms of larger constant-sized **data units** (e.g., signed integer, floating point number, string of bounded length, or data record).

These units are a *constant factor* larger than a single bit, and are operated on as a unit, so the result of the analysis is the same.

Measures of Complexity

The choice of algorithms and data structures has a critical impact on the following, both of which are used as measures of computational complexity:

- Run **time** to solve a problem of a given input size
- Storage **space** for data, including auxiliary structures

Example (preview of next lecture)

For example, suppose you have an input size of n elements, such as n strings to be sorted in lexicographic order. Suppose further that you have two algorithms at your disposal (these algorithms will be examined in detail in the next lecture):

Insertion sort:

1. start with an empty list
2. take each item to be sorted and insert it in its proper location

Merge sort:

1. if the list has only one item, return it
2. otherwise, split the list in half, sort each half with this procedure, and then merge the results

We will see that given *n* items to be sorted (it does not matter what they are as long as they are bounded by a constant size and can be compared by an < operator),

- *Insertion sort* takes time proportional to c_1n^2 steps, where c_1 is a constant depending on the implementation, and requires space proportional to *n*.

- *Merge sort* takes c_2n lg(*n*) steps, where c_2 is another constant depending on the merge sort implementation, and requires space proportional to 2*n*.

Exercise: This would be a good place for you to pause and do excercise 1.2-2, page 14:

> Suppose we are comparing implementations of insertion sort and merge sort on the same machine, where c_1=8 and c_2=16. For which values of n does insertion sort beat merge sort?

Constants matter for small input sizes, but since constants don't grow we ignore them when concerned with the time complexity of large inputs: it is the growth in terms of *n* that matters.

In the example above, ignoring the constants and factoring out the common n in each term shows that the difference in growth rate is *n* versus lg(*n*). For one million items to sort, this would be a time factor of one million for insertion sort, but about 20 for merge sort.

Models of Computation

Rather than bother with determining the constant factors for any given implementation or computer, algorithms for a problem are analyzed by using an abstract machine called a **model of computation**.

Many models of computation have been proposed, but they are essentially equivalent to each other (Church-Turing Thesis) as long as computation executed on them are *deterministic* and *sequential*. Commonly used models are *Turing Machines* and *Random Access Machines* (see Section 2.2 of the textbook).

Run Times for Different Complexities

In general, suppose that you have a computer of speed 10^7 steps per second. The running time of algorithms of the given complexity (rows) as a function of *n* would be:

size n	10	20	30	50	100	1000	10000
n	0.001ms	0.002ms	0.003ms	0.005ms	0.01ms	0.1ms	1ms
n lg n	0.003ms	0.008ms	0.015ms	0.03ms	0.07ms	1ms	13ms
n^2	0.01ms	0.04ms	0.09ms	0.25ms	1ms	100ms	10s
n^3	0.1ms	0.8ms	2.7ms	12.5ms	100ms	100s	28h
...	...	...	...	...	...	...	...
2^n	0.1ms	0.1s	100s	3yr	3x10^13c	inf	inf

Discussion: What is the difference between analysis of an algorithm and analysis of an implementation (a program)?

Discussion: What is the relationship between the efficiency of an algorithm and the difficulty of the problem to be solved by that algorithm? (We return to this in the last two topics of the semester, but see also below.)

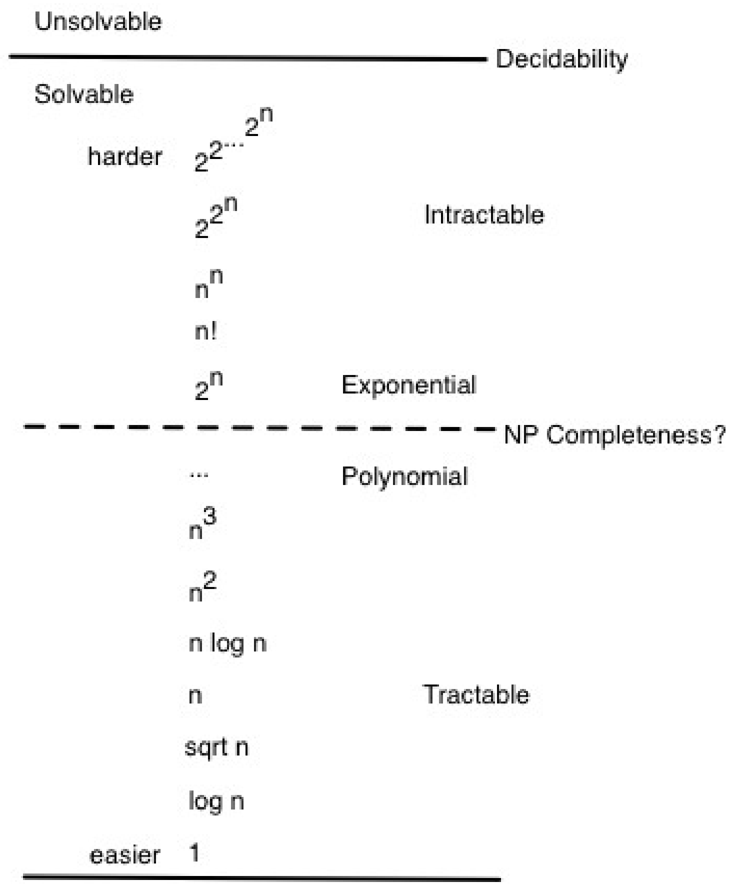
Consider the example above: the problem of sorting a list of items. We saw two algorithms for solving the problem, one more efficient than the other. Is it possible to make a statement about the time efficiency of *any possible* algorithm for the problem of sorting? (We address this question in Topic #10.)

Easy vs Hard Problems

Theoretical computer science has made substantial progress on understanding the intrinsic difficulty of *problems* (across all possible algorithms), although there are still significant open questions (one in particular).

First of all, there are problems that we cannot solve, i.e., problems for which there does not exist any algorithm. Those problems are called **unsolvable** (or **undecidable** or **incomputable**), and include the *Halting Problem* (refer to Section 3.1 pp. 176-177 of the textbook for ICS141 & 241).

Within the problems that can be solved, there is a hierarchy of **complexity classes** according to how difficult they are. Difficulty is based on proofs of the minimum complexity of *any* algorithm that solves the problem, and on proofs of equivalences between problems (translating one into another). Here is a graphic:



(Although algorithms can be ranked by this hierarchy, the above figure refers to problem classes, not algorithms.)

Sometimes small differences in a problem specification can make a big difference in complexity.

For example, suppose you use a graph of vertices representing cities and weighted edges between the vertices representing the distance via the best road traveling directly between the cities.

- The **Single Pair Shortest Paths** problem: what is the shortest path between a single pair of vertices (from one start vertex to one destination vertex) in the weighted graph?
- The **Shortest Paths** problem: what is the shortest path from one vertex to all of the other vertices in the weighted graph?
- The **All Pairs Shortest Paths** problem: what is the shortest path between every pair of vertices in the weighted graph?
- The **Traveling Salesman** problem: what is the shortest path that starts at given vertex in a weighted graph and visits all (or a specified set of) other vertices once before returning to the start vertex?

Discussion: How do these problems differ from each other? Which are easier and which are harder? Which are tractable (e.g., can be computed in polynomial time) and which are potentially intractable (e.g, require exponential time)?

Complexity theory will be the topic of our second to last lecture.

Abstract Data Types

Algorithms and data structures go together. We often study algorithms in the context of Abstract Data Types (ADTs). But let's start with Data Structures.

Data Structures

You are already familiar with Data Structures. They are defined by:

- **Operations:** Specifications of external appearance of a data structure.
- **Storage Structures:** Organizations of data implemented in lower level data structures. (We are almost always building abstractions on layers of abstractions above the actual physical implementation.)
- **Algorithms:** Description of how to manipulate information in the storage structures to obtain the results defined for the operations

The definition of a data structure requires that you specify implementation details such as storage structures and algorithms. It would be better to hide these details until we are ready to deal with them.

Also we may want to write a specification in terms of desired behavior (Operations) and then compare alternative storage structures and algorithms as possible solutions meeting those specifications. Data structures don't work because they already assume a given solution. Abstract Data Types or ADTs let us do this by abstracting the representations and algorithms.

Definition of Abstract Data Types (ADTs)

An ADT is a class of instances (i.e., data objects) with a set of the operations that can be applied to the data objects.

An ADT tells us *what to do* instead of how to do it. This provides the specifications againsts which we can design different algorithms: the *how* part.

An ADT is specified by

1. **the type(s) of data objects involved**
2. **a set of operations that can be applied to those objects**, and
3. **a set of properties (called axioms) that all the objects and operations must satisfy**.

Example: Stack ADT

Objects:
    Stack, and Elements (of arbitrary type)

Operations (categorized into three types):

Constructor
    `new()` creates the empty stack and returns it.

Accessors
    `empty(`*s*`)` returns whether stack *s* is empty.
    `top(`*s*`)` returns the element of stack *s* that has been inserted into s last.
Mutators (or Modifiers)
    `push(`*s*`,`*e*`)` inserts an element *e* into *s*.
    `pop(`*s*`)` deletes the top element from stack *s*.

Properties:

- `top(push(s,e)) returns value e`
- `pop(push(s,e)) leaves s` in the same state
- `empty(new()) = true`
- `empty(push(s,i)) = false`
- `pop(new()) is an error`
- `top(new()) is an error`

Specification and Implementation

ADTs can be specified in different languages:

- formal languages (axiomatic, algebraic, functional, denotational semantics, etc.)
- natural language

Implementation of an ADT requires

- defining the storage for the data structures
- implementing the algorithms for the operations

Advantages of ADTs

Modularity (Encapsulation)
: Abstract operations mean a program using an ADT are isolated from (need not know about or be affected by) the implementation of the ADT.
→ Implementation of ADT can be changed without modifying programs using ADT.
→ Makes a program smaller, simpler, and have less side effects
→ Helps to construct correct programs

Hierarchical Specification
: Supports Top-Down Design and Stepwise Refinement

Implementation
: ADTs map well to Object-Oriented Programming Languages

Discussion: What is the difference between an ADT and a data structure?

Some of the material in this page was adopted with permission (and significant editing) from Kazuo Sugihara's spring 2011 Lecture Notes #02.

Dan Suthers
Last modified: Fri Aug 26 02:18:27 HST 2011