Analysis and Design of Algorithms

CMP 2202

Lecture 1

Course Information

- Grading; 2 midterms: 40%, Final exam: 60%
- Office: 3014
- References:
  - Class notes
  - Thomas H. Cormen,Clara Lee, Erica Lin, 2002. *Introduction to Algorithms*
  - *The Design and Analysis of Algorithms*, Anany Levitin
  - *The Algorithm Design Manual*, Steven S. Skiena

Course Outline

- Introduction
  - Algorithms introduction
  - Algorithm types and examples
  - Asymptotic notations
- Design Techniques
- NP-Completeness
- Graph Theory

Introduction (1/2)

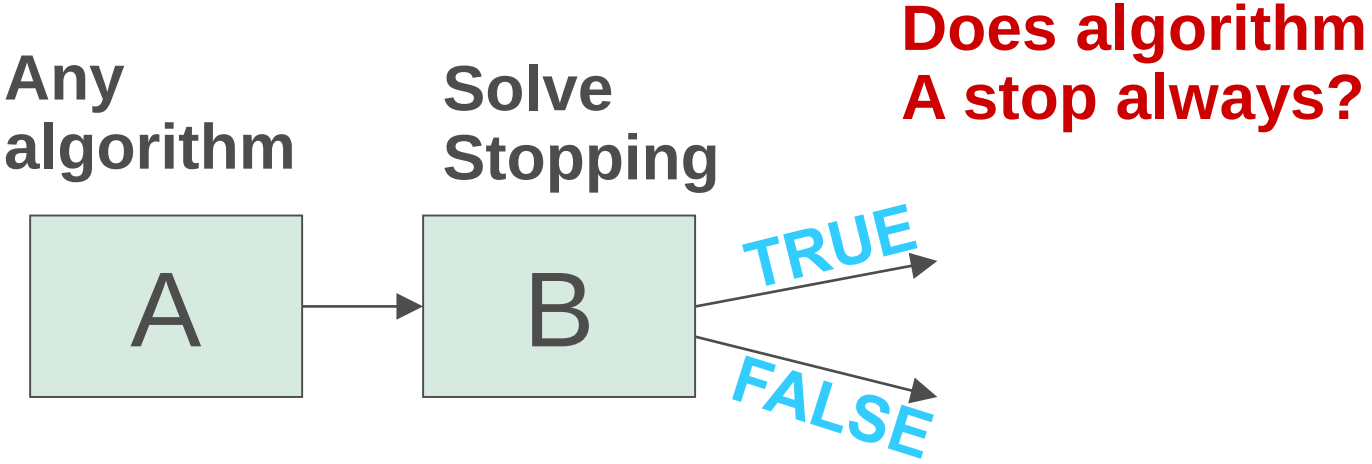
- What is an algorithm?
  - Algorithm is a *set of steps to complete a task.*
- For example;
- Task: to make a cup of tea.
- Algorithm;
  - Add water to the kettle
  - Boil it, add tea leaves,
  - Add sugar, and then serve

Introduction (2/2)

- We need a well-specified problem that tells what needs to be achieved
- Algorithm solves the problem
- It consists of a sequence of commands that takes an input and gives output

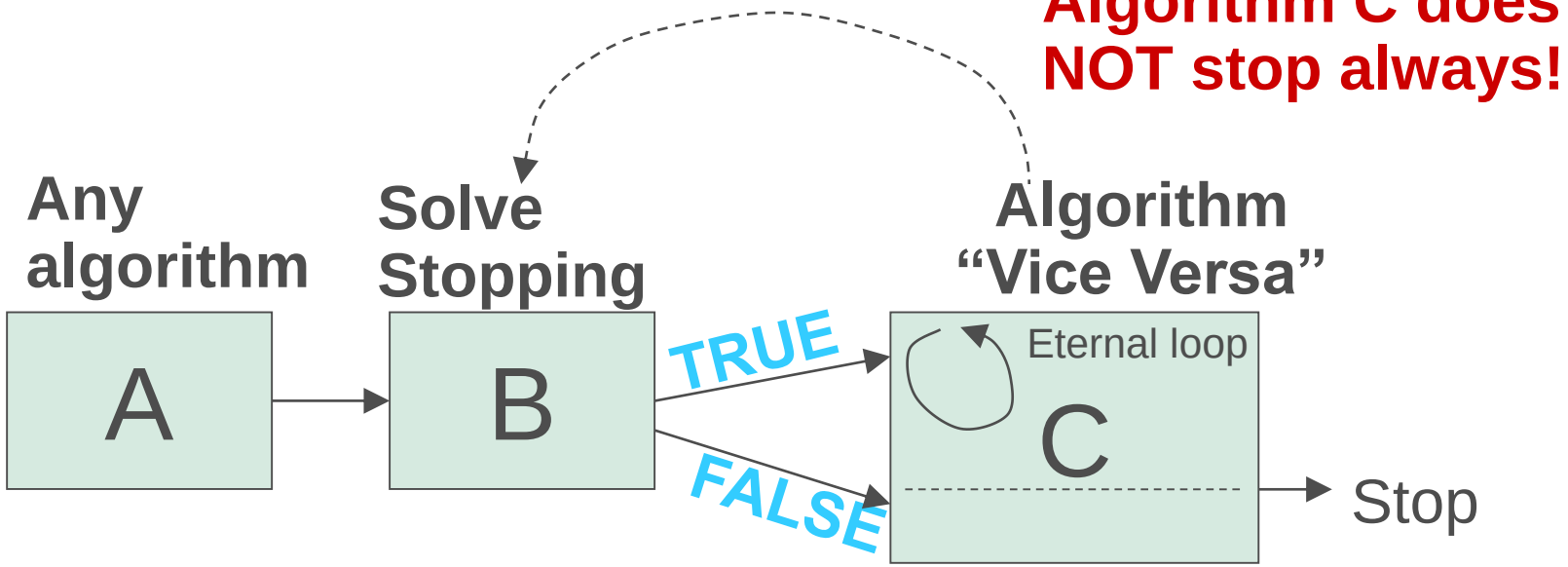
Non-solvable problems (1/2)

- Give list of all rational numbers in [0..1]
- Longest sequence of π without the 0 digit
- Stopping problem

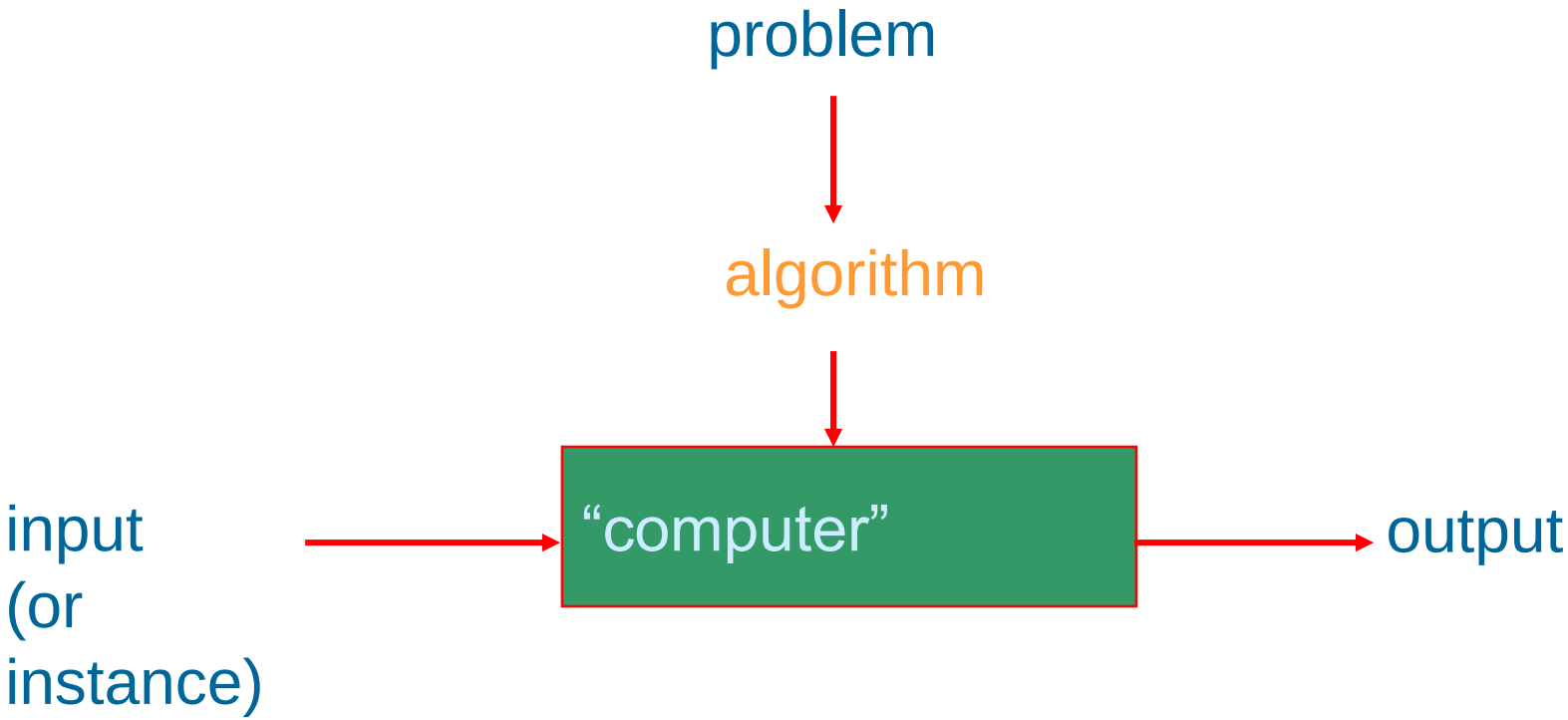


Non-solvable problems (2/2)

- Give list of all rational numbers in [0..1]
- Longest sequence of π without the 0 digit
- Stopping problem



Notion of algorithm and problem



algorithmic solution
(different from a conventional solution)

Why study algorithms? (1/2)

- *Language* for thinking about program behavior
  - analyzing correctness and resource usage
  - most computing research is algorithmic
- Standard set of algorithms and design techniques
- Feasibility (what can and cannot be done)
  - halting problem, NP-completeness

Why study algorithms? (2/2)

- Successful companies (Google, Akamai, …)
- Computation is fundamental to understanding the world
  - –cells, brains, social networks, physical systems,
- Exercise for your brain
- Fun!

Types of Algorithms

- Probabilistic Algorithms
- Heuristic Algorithms
- Approximation Algorithms

Probabilistic Algorithms

- In this algorithm, chosen values are used in such a way that the probability of choosing each value is known and controlled.

e.g. Randomize Quick Sort

Heuristic Algorithms

- This type of algorithm is based largely on optimism and often with minimal theoretical support. Here error can not be controlled but may be estimated how large it is.

Approximation Algorithms

In this algorithm, the answer is obtained that is as precise as required in decimal notation. In other words it specifies the error we are willing to accept.

For example, two figures accuracy or 8 figures or whatever is required.

Sorting Example

- Solves a general, well-specified problem
  - given a sequence of n keys, $a_1,\ldots,a_n$, as input, produce as output a reordering $b_1,\ldots,b_n$ of the keys so that $b_1 \leq b_2 \leq \ldots \leq b_n$.
- Problem has specific instances
  - [Dopey, Happy, Grumpy] or [3,5,7,1,2,3]
- Algorithm takes every possible instance and produces output with desired properties
  - insertion sort, quicksort, heapsort, …

Challenge

- Hard to design algorithms that are
  - correct
  - efficient
  - implementable
- Need to know about
  - design and modeling techniques
  - resources - don't reinvent the wheel

Correctness

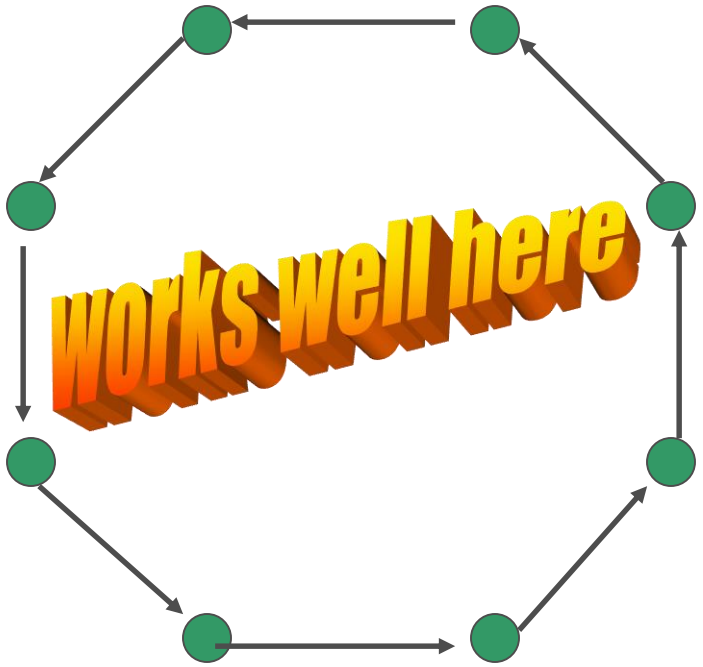
- How do you know an algorithm is correct?
  - produces the correct output on every input
- Since there are usually infinitely many inputs, it is not trivial
- Saying "it's obvious" can be dangerous
  - often one's intuition is tricked by one particular kind of input

Tour Finding Problem

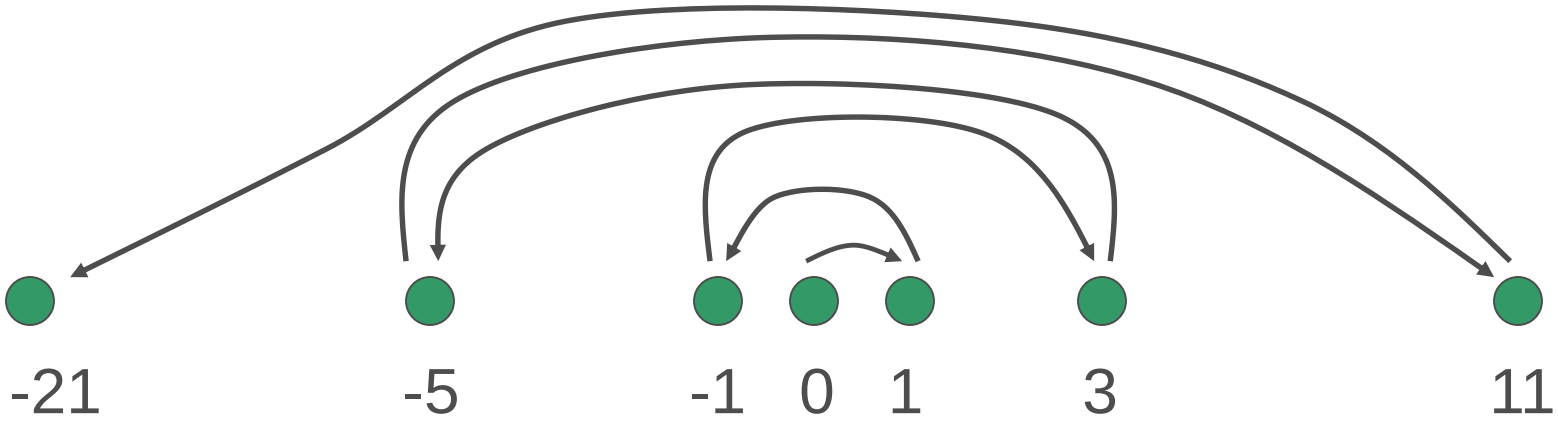
- Given a set of *n* points in the plane, what is the shortest tour that visits each point and returns to the beginning?
  - application: robot arm that solders contact points on a circuit board; want to minimize movements of the robot arm
- How can you find it?

Finding a Tour: Nearest Neighbor

- start by visiting any point
- while not all points are visited
  - choose unvisited point closest to last visited point and visit it
- return to first point



Nearest Neighbor Counter-example



How to Prove Correctness?

- There exist formal methods
  - even automated tools
  - more advanced than this course
- In this course we will primarily rely on more informal reasoning about correctness
- Seeking counter-examples to proposed algorithms is important part of design process

Efficiency

- Software is always outstripping hardware
  - need faster CPU, more memory for latest version of popular programs
- Given a problem:
  - what is an efficient algorithm?
  - what is the most efficient algorithm?
  - does there even exist an algorithm?

How to Measure Efficiency

- Machine-independent way:
  - analyze "pseudocode" version of algorithm
  - assume idealized machine model
    - one instruction takes one time unit
- "Big-Oh" notation
  - order of magnitude as problem size increases
- Worst-case analyses
  - safe, often occurs most often, average case often just as bad

Faster Algorithm vs. Faster CPU

- A faster algorithm running on a slower machine will always win for large enough instances
- Suppose algorithm S1 sorts n keys in $2n^2$ instructions
- Suppose computer C1 executes 1 billion instruc/sec
  - When n = 1 million, takes 2000 sec
- Suppose algorithm S2 sorts n keys in $50n\log_2 n$ instructions
- Suppose computer C2 executes 10 million instruc/sec
  - When n = 1 million, takes 100 sec

Caveat

- No point in finding fastest algorithm for part of the program that is not the bottleneck
- If program will only be run a few times, or time is not an issue (e.g., run overnight), then no point in finding fastest algorithm

Modeling the Real World

- Cast your application in terms of well-studied abstract data structures

Concrete	*Abstract*
arrangement, tour, ordering, sequence	permutation
cluster, collection, committee, group, packaging, selection	subsets
hierarchy, ancestor/descendants, taxonomy	trees
network, circuit, web, relationship	graph
sites, positions, locations	points
shapes, regions, boundaries	polygons
text, characters, patterns	strings

Real-World Applications

- Hardware design: VLSI chips
- Compilers
- Computer graphics: movies, video games
- Routing messages in the Internet
- Searching the Web
- Distributed file sharing
- Computer aided design and manufacturing
- Security: e-commerce, voting machines
- Multimedia: CD player, DVD, MP3, JPG, HDTV
- DNA sequencing, protein folding
- and many more!

Some Important Problem Types

- Sorting
  - a set of items
- Searching
  - among a set of items
- String processing
  - text, bit strings, gene sequences
- Graphs
  - model objects and their relationships
- Combinatorial
  - find desired permutation, combination or subset
- Geometric
  - graphics, imaging, robotics
- Numerical
  - continuous math: solving equations, evaluating functions

Algorithm Design Techniques

- Brute Force & Exhaustive Search
  - follow definition / try all possibilities
- Divide & Conquer
  - break problem into distinct subproblems
- Transformation
  - convert problem to another one
- Dynamic Programming
  - break problem into overlapping subproblems
- Greedy
  - repeatedly do what is best now
- Iterative Improvement
  - repeatedly improve current solution
- Randomization
  - use random numbers