CS 341: ALGORITHMS
Lecture 3: reductions
Readings: see website
Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341
  trevor.brown@uwaterloo.ca

PROBLEM REDUCTIONS

Why reinvent the wheel?
Reduce to another problem that you have already solved.

2SUM PROBLEM
• Input: Array \( A = [A[1], ... , A[n]] \) of integers and a target \( T \)
• Output: true if there exist two values in \( A \) (possibly the same value twice) whose sum equals \( T \), else false

Since the output is true/false, this is called a "decision problem"

Additional definitions:
A yes-instance is an input to a decision problem, for which the correct output is true
A no-instance is an input to a decision problem, for which the correct output is false

SIMPLE (BRUTE FORCE) SOLUTION

Runtime \( \Theta(n^2) \) by similar arguments to earlier...

Idea: let's turn the innermost loop into something more efficient...

AN IMPROVEMENT

Instead of looping over \( j \), search the array for \( T - A[i] \)

How to do this efficiently?

IMPROVED ALGORITHM

Use binary search:
Searches \( n \) elements in \( O(\log n) \) time
Requires elements to be sorted!

Runtime \( \Theta(n \log n) \) VS. linear search, which takes \( \Theta(n) \) time

What is this algorithm's time complexity?
TIME COMPLEXITY

- Loop: iterations * work per iteration
  - $O(n) + O(n) = O(n \log n)$
- Entire algorithm: $O(n \log n) + O(n \log n) = O(n \log n)$

PREPROCESSING

- The sort is an example of **pre-processing**
- It modifies the input to permit a more efficient algorithm (binary search as opposed to linear search)
- Note that a pre-processing step is only done once

3SUM PROBLEM

- Input: Array $A = [A[1], \ldots, A[n]]$ of integers and a target $T$
- Output: true if there exist three values in $A$ (possibly taking the same value two or three times) whose sum equals $T$, else false

This is quite similar to 2SUM... Can we reduce to 2SUM?

3SUM PROBLEM

- Input: Array $A = [A[1], \ldots, A[n]]$ of integers and a target $T$
- Output: true if there exist three values in $A$ (possibly taking the same value two or three times) whose sum equals $T$, else false

PREPROCESSING

- The sort is an example of **pre-processing**
- It modifies the input to permit a more efficient algorithm (binary search as opposed to linear search)
- Note that a pre-processing step is only done once

REDUCTIONS

- Suppose we already have a solution to 2SUM called Solve2SUM
- Suppose we design an algorithm **Reduce3SUMto2SUM** that solves 3SUM, and this algorithm calls Solve2SUM as a subroutine
  - **Solve2SUM** is a black-box subroutine that we call as “oracle”

- **Reduce3SUMto2SUM** is called a reduction from 3SUM to 2SUM
- Could also process input / call Solve2SUM multiple times
- If 3SUM can be reduced to 2SUM, we denote this by $3SUM \leq 2SUM$
  - Mnemonic: 2SUM goes into 3SUM as a subproblem

REDUCTION FROM 3SUM TO 2SUM

- How can we use Solve2SUM to solve 3SUM?
  - By changing the array $A$ somehow?
  - By changing the target $T$ somehow?

**REDUCTION FROM 3SUM TO 2SUM**

- How can we use Solve2SUM to solve 3SUM?
  - By changing the array $A$ somehow?
  - By changing the target $T$ somehow?

<table>
<thead>
<tr>
<th>$T = 9$</th>
<th>$A$</th>
<th>$i$</th>
<th>$T_2$</th>
<th>$T_2 = T - A[i]$</th>
<th>If Solve2SUM($A$, $T_2$) return true</th>
<th>return false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>$T_2 = 8$</td>
<td>Solve2SUM($A, 8$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 2$</td>
<td>$T_2 = 16$</td>
<td>Solve2SUM($A, 16$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 3$</td>
<td>$T_2 = 11$</td>
<td>Solve2SUM($A, 11$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 4$</td>
<td>$T_2 = 9$</td>
<td>Solve2SUM($A, 9$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 5$</td>
<td>$T_2 = 7$</td>
<td>Solve2SUM($A, 7$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 6$</td>
<td>$T_2 = 10$</td>
<td>Solve2SUM($A, 10$)</td>
<td>$\rightarrow$</td>
<td>$False$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i = 7$</td>
<td>$T_2 = 6$</td>
<td>Solve2SUM($A, 6$)</td>
<td>$\rightarrow$</td>
<td>$True$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**REDUCTION CORRECTNESS**
- **Must prove:** \( 3\text{SUM}(A, T) \iff \exists i : 2\text{SUM}(A, T - A[i]) \)
- In other words,
- Let \( A, T \) be any input to \( 3\text{SUM} \)
- There exist \( A[i], A[j], A[k] \) that sum to \( T \) if and only if
- there exists some \( A[m] \) such that \( \text{Solve2SUM}(A, T - A[i]) \) returns true

**REDUCTION RUNTIME**

\[
\theta(n) \text{ loop iterations}
\]
- Each iteration does \( \theta(1) + \text{Runtime(Solve2SUM)} \) work
- Runtime depends on implementation of Solve2SUM!
- Brute force: \( \theta(n) \cdot \theta(n^2) = \theta(n^3) \)
- Binary search: \( \theta(n) \cdot \theta(n \log n) = \theta(n^2 \log n) \)

**FURTHER IMPROVEMENT**
Recall our fastest Solve2SUM took \( O(n \log n) \) time for sorting, and \( O(n \log n) \) total time for searching
- Can actually improve 2SUM to \( O(n) \) searching time with a **greedy** approach
- Does not change complexity of 2SUM, but we will see we can still speed up our 3SUM reduction...

**FAST 2SUM**

\[
T = 23 \\
\begin{array}{cccccccc}
2 & 3 & 5 & 11 & 12 & 20 & 22 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\end{array}
\]
- **Correctness**
- Invariant: if there exists a solution \( i' < j' \) then \( i' \geq i \) and \( j' \leq j \)
- Exercise: fill in the proof details

**FAST 3SUM TO 2SUM REDUCTION**
Although fast 2SUM is still \( \theta(n \log n) \), we can **sort only once** in our reduction

\[
\begin{array}{cccccccc}
1 & \text{Reduce3SUM to 2SUM} \{ A[1..n], T \} \\
2 & \text{for } i = 1 \ldots n \\
3 & \text{if } T - T - A[i] \text{ then return false} \\
4 & \text{if } \text{Solve2SUM}(A, T) \text{ return true} \\
5 & \text{return false} \\
\end{array}
\]
- Since 2SUM is given a pre-sorted array, it takes \( \theta(n) \) time!
- We get runtime \( \theta(n \log n) + \theta(n) \theta(n) = \theta(n^2) \)
IS THERE A FASTER 3SUM ALGORITHM?

For many years, people thought this was likely optimal
However faster algorithms appeared in 2014, 2017

Best known solution is:
\[ O(n^{2.373}) \]
This is a polylog factor faster than \( O(n^3) \)
... we suspect there is no solution faster than \( O(n^2 - \Omega(1)) \)

PROGRESSIVELY HARDER WORKED EXAMPLES

A TRIVIAL REDUCTION

Suppose we want to multiply two integers, \( x \) and \( y \)
Consider the algebraic identity: \( xy = (x+y)^2 - x^2 - y^2 \)
This allows us to show that \( \text{Multiplication} \leq \text{Squaring} \)

Oracle: ComputeSquare
- Oracle "gives" you a solution to the subproblem...
- If you solve ComputeSquare, you've solved Multiply

3SUM ≤ 3SUMZERO

- If and only if \( 3A[i] + 3A[j] + 3A[k] - 3T = 0 \)
- If and only if \( (3A[i] - T) + (3A[j] - T) + (3A[k] - T) = 0 \)
- This suggests the following approach

3SUM ≤ 3SUMZero

Given an oracle that solves 3SUMZero, let's solve 3SUM

A MEDIUM REDUCTION

3SUMZero problem
- Input: array \( A = [A[1], ..., A[n]] \) of integers
- Suppose we have solved 3SUMZero and want to solve 3SUM
It is straightforward to modify any algorithm for 3SUMZero so it solves 3SUM
Another approach is to find a reduction \( 3SUM \leq 3SUMZero \). This would allow code re-use.

3SUM ≤ 3SUMZero

A HARD REDUCTION

3array3SUMZero problem
- Input: three arrays of \( n \) integers: \( A, B \) and \( C \)
- Output: true if there exist \( A[i], B[j], C[k] \), whose sum equals \( 0 \), else false
- Let's try to reduce this to 3SUMZero

A REDUCTION

Given an oracle that solves 3SUMZero, let's solve 3SUM

Problem: Solve 3SUMZero might choose \( \geq 2 \) elements from the same array.
No correct way to get a zero sum!

TRY_reduce_3array3SUMzero to 3SUMZero(A, B, C)
  1. \( A' \) = concatenation of \( A, B, C \)
  2. return Solve3SUMZero(A')

But Solve3SUMZero(A') returns true!

How to prevent picking 2+ elements from a subarray?

Somehow ensure the sum cannot be zero unless we pick one element from each subarray

Multiply by 10: preserves sets of elements that sum to 0

Add +1
Add +2
Add -3

So, only way to get 0 is to pick one from each subarray!

THE REDUCTION

To show that this reduction is correct, we prove:

true is the correct output for Solve3SUMZero(A') if and only if
true is the correct output for Solve3array(A, B, C)

For brevity let's just call this SolveArray ...

CORRECTNESS OF THE REDUCTION (1/3)

To show that this reduction is correct, we prove:

true is the correct output for Solve3SUMZero(A') if and only if
true is the correct output for Solve3array(A, B, C)

CORRECTNESS OF THE REDUCTION (2/3)

To show that this reduction is correct, we prove:

true is the correct output for Solve3SUMZero(A') if and only if
true is the correct output for Solve3array(A, B, C)

Case 1: Assume true is the correct output for Solve3array(A, B, C)
- Want to show true is the correct output for Solve3SUMZero(A')
- By our assumption, there exist \( A[i] + B[j] + C[k] = 0 \)
- So \( 10A[i] + 1, 10B[j] + 2, 10C[k] - 3 = 0 \)
- So true is the correct output for Solve3SUMZero(A')

CORRECTNESS OF THE REDUCTION (3/3)

Case 2: Assume true is the correct output for Solve3SUMZero(A')
- Want to show true is the correct output for Solve3array(A, B, C)
- By our assumption, there exist \( A[i] + B[j] + C[k] = 0 \)
- Claim: this sum consists of one element from each of \( A, B \) and \( C \)
  - By case...
  - Example case: suppose, for contradiction, that \( A[i], A[j], A[k] \) are all in \( B \)
  - which is \textit{not zero} Contradiction!

So, Solve3array3SUMZero ≤ 3SUMZero
MANY-ONE REDUCTIONS

- The previous three reductions had a very special structure
  - We transformed (reduced) an instance of the first problem to an instance of the second problem
  - We called the oracle once, on the transformed instance

Reductions of this form, in the context of decision problems, are called **many-one reductions**
- (also known as polynomial transformations or Karp reductions)

We will many examples of these in the section on **intractability**