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

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**

Since the output is true/false, this is called a “decision problem”
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!

VS. linear search, which takes $O(n)$ time

What is this algorithm's time complexity?

```python
2SUM_Improved(A[1..n], T)
    sort(A)
    for i = 1 .. n
        j = binary search for T-A[i]
        in the subarray A[1..n]
        if search is successful return true
    return false
```
TIME COMPLEXITY

- **Loop**: iterations * work per iteration
  - $\Theta(n) \times \Theta(\log n) = \Theta(n \log n)$
- **Entire algorithm**: $\Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$
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 $A = [A[1], ..., A[n]]$ 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?
Suppose we already have a solution to 2SUM called Solve2SUM.

Suppose we design an algorithm \texttt{Reduce3SUMto2SUM} that solves \texttt{3SUM}, and this algorithm calls \texttt{Solve2SUM} as a subroutine.

\texttt{Reduce3SUMto2SUM} is called a \textbf{reduction from 3SUM to 2SUM}.

\texttt{Solve2SUM} is a \textbf{black-box subroutine} that we call an “oracle”.

\begin{verbatim}
1 Reduce3SUMto2SUM(input_to_3_sum)
2     input_to_2_sum = process(input_to_3_sum)
3     return Solve2SUM(input_to_2_sum)
\end{verbatim}

\texttt{Reduce3SUMto2SUM} is called a \textbf{reduction from 3SUM to 2SUM}.

Could also process input / call \texttt{Solve2SUM} multiple times.

If \texttt{3SUM} can be reduced to \texttt{2SUM}, we denote this by \textbf{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?

```python
1  def Reduce3SUMto2SUM(A[1..n], T):
2      for i = 1 .. n
3          T2 = T - A[i]
4          if Solve2SUM(A, T2) return true
5      return false
```
Reduce3SUMto2SUM(A[1..n], T)
for i = 1 .. n
    T2 = T - A[i]
    if Solve2SUM(A, T2) return true
return false

\[ T = 9 \]
\[ A = \begin{array}{ccccccc}
1 & -7 & -3 & 0 & 2 & -1 & 3 & -2
\end{array} \]

\( i = 1 \) \quad T2 = 8 \quad \text{Solve2SUM(A, 8) → False}

\( i = 2 \) \quad T2 = 16 \quad \text{Solve2SUM(A, 16) → False}

\( i = 3 \) \quad T2 = 11 \quad \text{Solve2SUM(A, 11) → False}

\( i = 4 \) \quad T2 = 9 \quad \text{Solve2SUM(A, 9) → False}

\( i = 5 \) \quad T2 = 7 \quad \text{Solve2SUM(A, 7) → False}

\( i = 6 \) \quad T2 = 10 \quad \text{Solve2SUM(A, 10) → False}

\( i = 7 \) \quad T2 = 6 \quad \text{Solve2SUM(A, 6) → True}
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 $\text{Solve2SUM}(A, T - A[m])$ returns true
REDUCTION CORRECTNESS 2

- **WTP:** \( \exists A[i], A[j], A[k] \) that sum to \( T \) if and only if
- \( \exists A[m] \) such that \( \text{Solve2SUM}(A, T - A[m]) \) returns true

```
1 Reduce3SUMto2SUM(A[1..n], T)
2   for i = 1 .. n
3     T2 = T - A[i]
4     if Solve2SUM(A, T2) return true
5   return false
```

  - \( \iff \exists i : \text{Solve2SUM}(A, T - A[i]) \) returns true
### Reduction Runtime

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>Reduce3SUMto2SUM(A[1..n], T)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>for i = 1 .. n</code></td>
</tr>
<tr>
<td>3</td>
<td><code>T2 = T - A[i]</code></td>
</tr>
<tr>
<td>4</td>
<td><code>if Solve2SUM(A, T2) return true</code></td>
</tr>
<tr>
<td>5</td>
<td><code>return false</code></td>
</tr>
</tbody>
</table>

- $\Theta(n)$ loop iterations
- Each iteration does $\Theta(1) + \text{Runtime}(\text{Solve2SUM})$ work
- Runtime depends on implementation of Solve2SUM!
- Brute force: $\Theta(n) \times \Theta(n^2) = \Theta(n^3)$
- Binary search: $\Theta(n) \times \Theta(n \log n) = \Theta(n^2 \log n)$
• 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…
Correctness

Invariant: if there exists a solution $i' < j'$ then $i' \geq i$ and $j' \leq j$

Exercise: fill in the proof details
Although fast 2SUM is still $\Theta(n \log n)$, we can sort only once in our reduction.

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:

\[ O(n^2 (\log \log n)^{O(1)}/\log^2 n) \]

○ This is a polylog factor faster than \(O(n^2)\)
○ … 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: \( xy = \frac{(x+y)^2 - (x-y)^2}{4} \)
- This allows us to show that Multiplication \( \leq \) Squaring.

Oracle: ComputeSquare

- Oracle “gives” you a solution to the subproblem...
- If you solve ComputeSquare, you’ve solved Multiply.
A MEDIUM REDUCTION

- **3SUMZero** problem

- 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.
if and only if $3A[i] + 3A[j] + 3A[k] - 3T = 0$

This suggests the following approach:

1. Reduce $3SUM$ to $3SUMZero(A, T)$
   for $i = 1..n$
   $B[i] = 3*A[i] - T$
   return $Solve3SUMZero(B)$

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

Can we find $\sum x_i = T$ by finding $\sum y_i = 0$?

Preprocess input to create new array $B$

Use the oracle to solve subproblem $3SUMZero(B)$
If this reduction is correct, the result should be a solution to problem $3SUM(A, T)$
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**

```python
TRY_reduce_3array3SUMZero_to_3SUMZero(A, B, C)
    A' = concatenation fo A, B, C
    return Solve3SUMZero(A')
```

Is this reduction correct?

Problem: **Solve3SUMZero** 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)
\[ A' = \text{concatenation fo } A, B, C \]
\[ \text{return } \text{Solve3SUMZero}(A') \]

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-2</td>
<td>-1</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Add +1  

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>-19</td>
<td>-9</td>
<td>82</td>
<td>32</td>
<td>52</td>
<td>57</td>
<td>37</td>
<td>67</td>
<td></td>
</tr>
</tbody>
</table>

Add +2

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add -3

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If sum of 3 elements is 0, so is their sum mod 10

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

```
1  Reduce_3array3SUMZero_to_3SUMZero(A, B, C)
   for i = 1..n
   D[i] = 10A[i] + 1
   E[i] = 10B[i] + 2
   F[i] = 10C[i] - 3
   A' = concatenation of D, E, F
   return Solve3SUMZero(A')
```
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($A, B, C$)
To show that this reduction is correct, we prove: \textit{true} is the correct output for \textit{Solve3SUMZero}(\(A'\)) if and only if \textit{true} is the correct output for \textit{Solve3array}(\(A, B, C\))

**Case 1:** Assume \textit{true} is the correct output for \textit{Solve3array}(\(A, B, C\))

- Want to show \textit{true} is the correct output for \textit{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 \textit{true} is the correct output for \textit{Solve3SUMZero}(\(A'\))
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 $A'[i] + A'[j] + A'[k] = 0$

Claim: this sum consists of one element from each of $A, B$ and $C$

By cases... Example case:
Suppose, for contradiction, that $A'[i], A'[j], A'[k]$ are all in $B$

Then the sum $A'[i] + A'[j] + A'[k] = 10B[...] + 2 + 10B[...] + 2 + 10B[...] + 2$, which is not zero! Contradiction!

Consider the sum modulo 10...
only way to get 0 is to pick one element from each of $A, B, C$

So, there is a sum=0, with one from each of $A, B$ and $C$. So, true is the correct output for Solve3array($A, B, C$)

So, $3\text{array3SUMZero} \leq 3\text{SUMZero}$
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.