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
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”
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!

```plaintext
1  2SUM_Improved(A[1..n], T)
2    sort(A)
3    for i = 1 .. n
4      j = binary search for T-A[i]
5      in the subarray A[1..n]
6      if search is successful return true
7    return false
```

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

What is this algorithm's time complexity?
TIME COMPLEXITY

$\Theta(n \log n)$

```c
1 2SUM_Improved(A[1..n], T)
2    sort(A)
3    for i = 1 .. n
4        j = binary search for T-A[i]
5          in the subarray A[1..n]
6          if search is successful return true
7    return false
```

$\Theta(n)$ iterations

$\Theta(\log(n))$ * Loop: iterations * work per iteration

- $\Theta(n) * \Theta(\log n) = \Theta(n \log n)$

- 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 $A = [A[1], \ldots, A[n]]$ 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?
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 an “oracle”

1. Reduce3SUMto2SUM(input_to_3sum)
2. input_to_2sum = process(input_to_3sum)
3. return Solve2SUM(input_to_2sum)

• 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 $3\text{SUM} \leq 2\text{SUM}$

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?

```plaintext
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
```
```python
Reduce3SUMto2SUM(A[1..n], T)
for i = 1 .. n
    T2 = T - A[i]
    if Solve2SUM(A, T2) return true
return false
```

<table>
<thead>
<tr>
<th>$T$</th>
<th>$A$</th>
<th>$T2$</th>
<th>$i$</th>
<th>$T2$</th>
<th>$i$</th>
<th>$T2$</th>
<th>$i$</th>
<th>$T2$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>16</td>
<td>3</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-7</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>3</td>
<td>-2</td>
<td>7</td>
</tr>
</tbody>
</table>

Solve2SUM(A, 8) → False
Solve2SUM(A, 16) → False
Solve2SUM(A, 11) → False
Solve2SUM(A, 9) → False
Solve2SUM(A, 7) → False
Solve2SUM(A, 10) → False
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

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

  • $\Leftrightarrow \exists i : \text{Solve2SUM}(A, T - A[i])$ 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
```

- $\Theta(n)$ 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: $\Theta(n) \times \Theta(n \log n) = \Theta(n^2 \log n)$
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

\[ T = 23 \quad A \begin{array}{ccccccc} 2 & 3 & 5 & 11 & 12 & 20 & 22 \end{array} \]

- Correctness
- Invariant: if there exists a solution \( i' < j' \) then \( i' \geq i \) and \( j' \leq j \)
- Exercise: fill in the proof details

sum = 24
too large! move \( j \)

sum = 22
too small! move \( i \)

sum = 23
Just right!

However, if \( i \) and \( j \) meet, there is no solution
FAST 3SUM TO 2SUM REDUCTION

• Although fast 2SUM is still $\Theta(n \log n)$, we can sort only once in our reduction.

```python
1   Reduce3SUMto2SUM(A[1..n], T)
2       sort(A)
3       for i = 1 .. n
4           T2 = T - A[i]
5           if Fast2SUM(A, T2) return true
6       return false
```

• 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 \textbf{Multiplication} ≤ \textbf{Squaring}

```
1  ReduceMultiplyToSquare(x, y)
2  s = ComputeSquare(x+y)
3  t = ComputeSquare(x-y)
4  return (s-t) >> 2
```

• \textbf{Oracle}: ComputeSquare
  • Oracle “gives” you a solution to the \textbf{subproblem}…
  • If \textbf{you} solve ComputeSquare, you’ve solved \textbf{Multiply}
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

• 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

```
1 Reduce_3SUM_to_3SUMZero(A, T)
2    for i = 1..n
3        B[i] = 3*A[i] - T
4 return Solve3SUMZero(B)
```

Preprocess input to create new array $B$

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

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

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!

1. TRYReduce3Array3SUMZeroTo3SUMZero(A, B, C)
2. A' = concatenation of A, B, C
3. 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

<table>
<thead>
<tr>
<th>2</th>
<th>-2</th>
<th>-1</th>
<th>8</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>-20</td>
<td>-10</td>
<td>80</td>
<td>30</td>
<td>50</td>
<td>60</td>
<td>40</td>
<td>70</td>
</tr>
</tbody>
</table>

| 21 | -19| -9 | 82 | 32 | 52 | 57 | 37 | 67 |

$A[i] \mod 10 = 1$  $A[j] \mod 10 = 2$  $A[j] \mod 10 = 7$

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!
1 Reduce_3array3SUMZero_to_3SUMZero(A, B, C)
2     for i = 1..n
3        D[i] = 10A[i] + 1
4        E[i] = 10B[i] + 2
5        F[i] = 10C[i] - 3
6 A' = concatenation of D, E, F
7 return Solve3SUMZero(A')
CORRECTNESS OF THE REDUCTION (1/3)

```python
1. Reduce_3array3SUMZero_to_3SUMZero(A, B, C)
   2. for i = 1..n
      3. D[i] = 10A[i] + 1
      4. E[i] = 10B[i] + 2
      5. F[i] = 10C[i] - 3
   6. A' = concatenation of D, E, F
   7. return Solve3SUMZero(A')
```

For brevity let’s just call this Solve3array ...

- 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 \texttt{Solve3SUMZero}(A') \textit{if and only if}
- **true** is the correct output for \texttt{Solve3array}(A, B, C)

**Case 1:** Assume true is the correct output for \texttt{Solve3array}(A, B, C)
- Want to show true is the correct output for \texttt{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 this is the correct output for \texttt{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, 3array3SUMZero \leq 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