**CS 341: ALGORITHMS**

**Lecture 3: reductions**
Readings: see website

Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341

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

What is this algorithm's time complexity?

VS. linear search, which takes \( \Theta(n) \) time
TIME COMPLEXITY

- **Loop:** iterations * work per iteration
  - \( \theta(n) \cdot \theta(\log n) = \theta(n \log n) \)
  - **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**?

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"

- **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 ≤ 2SUM**

Mnemonic: 3SUM goes into 3SUM as a subproblem

3SUM TO 2SUM REDUCTION

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

\[
\text{Reduce3SUMto2SUM}(A[1], ..., T) \quad \text{for } i = 1 \ldots n
\]
\[
\quad \text{if } T = T - A[i] \quad \text{if Solve2SUM}(A, T2) \text{ return true}
\]
\[
\quad \text{return false}
\]

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

T = 9 A: 1 -7 3 0 2 -1 3 -2
i = 1 T2 = 8 Solve2SUM(A, 8) → False
i = 2 T2 = 16 Solve2SUM(A, 16) → False
i = 3 T2 = 11 Solve2SUM(A, 11) → False
i = 4 T2 = 9 Solve2SUM(A, 9) → False
i = 5 T2 = 7 Solve2SUM(A, 7) → False
i = 6 T2 = 10 Solve2SUM(A, 10) → False
i = 7 T2 = 6 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 \(\text{Solve}2\text{SUM}(A, T - A[m])\) 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 \(\text{Solve}2\text{SUM}(A, T - A[m])\) returns true

![Code Snippet](image)

  - \(\iff \exists i : \text{Solve}2\text{SUM}(A, T - A[i])\) returns true

REDUCTION RUNTIME

- \(\Theta(n)\) loop iterations
- Each iteration does \(\Theta(1)\) + Runtime(\text{Solve}2\text{SUM}) work
- Runtime depends on implementation of \text{Solve}2\text{SUM}!
- 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 \text{Solve}2\text{SUM} took \(O(n \log n)\) time for sorting, and \(O(n \log n)\) total time for searching
- Can actually improve \text{Solve}2\text{SUM} to \(O(n)\) searching time with a greedy approach
- Does not change complexity of \text{2SUM}, but we will see we can still speed up our \text{3SUM} reduction...

FAST 2SUM

\[ T = 23 \]
\[ \begin{array}{cccccccc}
A & 2 & 3 & 5 & 11 & 12 & 20 & 22 \\
\uparrow & \uparrow & \uparrow & & \uparrow & \uparrow & \uparrow & \\
i & i & i & & j & j & & \\
\end{array} \]

- **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 \text{2SUM} is still \(\Theta(n \log n)\), we can sort only once in our reduction

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

- Since \text{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:
  \[ O(n^2 (\log \log n)^{\frac{3}{2}} \log^2 n) \]
• This is a polylog factor faster than \( O(n^2) \)
• ...we suspect there is no solution faster than \( O(n^{2-\alpha(n)}) \)

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 \( \text{Multiplication} \leq \text{Squaring} \)
  ```
  1. ReduceMultiplyToSquare(x, y)
  2. a = ComputeSquare(x+y)
  3. t = ComputeSquare(x-y)
  4. return ((a-t)>>2)
  
  Oracle: ComputeSquare
  • Oracle "gives" you a solution to the subproblem...
  • If you solve ComputeSquare, you've solved 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.

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
  ```
  1. TRY_reduce_3array3SUMZeroTo_3SUMZero (A, B, C)
  2. A' = concatenation for A, B, C
  3. return Solve3SUMZero(A')
  
  Use the oracle to solve subproblem 3SUMZero(B)
  If this reduction is correct, the result should be a solution to problem 3SUM(A, T)

3SUM \leq 3SUMZero

• If and only if \( A[i] + A[j] + A[k] = T = 0 \)
• 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
  4. return Solve3SUMZero(B)
  
  Given an oracle that solves 3SUMZero, let's solve 3SUM.
  Can we find \( \sum x_i = T \) by finding \( \sum y_i = 0 \)?

```
No correct way to get a zero sum

```
1 TRY reduce_3array3SUMZero to 3SUMZero(A, B, C)
2   A' = concatenation of A, B, C
3   return Solve3SUMZero(A')
```

1 2  -2 -1  B  8  3  5  C  6  4  7

But Solve3SUMZero(A') returns true!

How to prevent picking 2+ elements from a subarray?

**THE REDUCTION**

1 Reduce 3array3SUMZero to 3SUMZero(A, B, C)
2   for i = 1..n
3     D[i] = 10A[i] + 1
4     E[i] = 10B[i] + 3
5     F[i] = 10C[i] - 3
6     A' = concatenation of D, E, F
7   return Solve3SUMZero(A')

**CORRECTNESS OF THE REDUCTION (1/3)**

• To show that this reduction is correct, we prove:
  • **true** is the correct output for **Solve3array(A, B, C)** if and only if
  • **true** is the correct output for **Solve3array3SUMZero(A')**

• **Case 1:** Assume **true** is the correct output for **Solve3array(A, B, C)**
  • Want to show **true** is the correct output for **Solve3array3SUMZero(A')**
  • By our assumption, there exist $A[i] + B[j] + C[k] = 0$
  • So $10A[i] + 1 + 10B[j] + 2 + 10C[k] - 3 = 0$
    - $D[i] + E[j] + F[k]$
  • So **true** is the correct output for **Solve3array3SUMZero(A')**

**CORRECTNESS OF THE REDUCTION (2/3)**

**Case 2:** Assume **true** is the correct output for **Solve3array3SUMZero(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 $C$

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 (3/3)**

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

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)**
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