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 $A = \{A[1], \ldots, A[n]\}$ 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”

AN IMPROVEMENT
  we can rearrange to get $A[j] = T - A[i]$
• Instead of looping over $j$,
  search the array for $T - A[i]$

How to do this efficiently?

SIMPLE (BRUTE FORCE) SOLUTION

Given: $A[1..n], T$
for $i = 1..n$
for $j = i..n$
return true
return false

Runtime $O(n^2)$ by similar arguments to earlier...

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

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?
TIME COMPLEXITY

- \( \Theta(n \log n) \) for \( \text{Improved}(A[1..n], T) \)
- Loop: Iterations \( \cdot \) work per iteration
  - \( R(n) = \Theta(n \log n) \)
  - \( \text{Entire algorithm: } \Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n) \)

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?

REDUCTION FROM 3SUM TO 2SUM

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

REDUCTIONS

- Suppose we already have a solution to 2SUM called \( \text{Solve2SUM} \)
  - Suppose we design an algorithm \( \text{Reduce3SUMto2SUM} \) that solves 3SUM, and this algorithm calls \( \text{Solve2SUM} \) as a subroutine

\[ \text{Solve2SUM} \text{ is a black-box subroutine that we call an “oracle”} \]

1. \( \text{Reduce3SUMto2SUM}(\text{input to 3sum}) \)
2. \( \text{input to 2sum = process (input to 3sum)} \)
3. \( \text{return Solve2SUM(input to 2sum)} \)

- \( \text{Reduce3SUMto2SUM} \) is called a reduction from 3SUM to 2SUM
- Could also process input / call \( \text{Solve2SUM} \) multiple times
- If 3SUM can be reduced to 2SUM, we denote this by 3SUM \( \leq \) 2SUM

Mnemonic: 2SUM goes into 3SUM as a subproblem

\[ A = [1, 2, 3, 1, 2, 1, 2, 3] \]

\[ T = 9 \]

- If \( 1 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 8) \) \( \rightarrow \) \( \text{False} \)
- If \( 2 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 16) \) \( \rightarrow \) \( \text{False} \)
- If \( 3 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 11) \) \( \rightarrow \) \( \text{False} \)
- If \( 4 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 9) \) \( \rightarrow \) \( \text{False} \)
- If \( 5 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 7) \) \( \rightarrow \) \( \text{False} \)
- If \( 6 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 10) \) \( \rightarrow \) \( \text{False} \)
- If \( 7 \) \( \rightarrow \) \( \text{Solve2SUM}(A, 6) \) \( \rightarrow \) \( \text{True} \)
REDUCTION CORRECTNESS

• Must prove: 3SUM(A, T) ⇔ ∃i: 2SUM(A, T = A[i])
• In other words,
  • Let A,T be any input to 3SUM
  • There exist A[i], A[j], A[k] that sum to T if and only if
  • there exists some A[m] such that Solve2SUM(A, T = A[m]) returns true

REDUCTION CORRECTNESS 2

• ∃A[m] such that Solve2SUM(A, T = A[m]) returns true

  • ⇔ ∃A[m]: Solve2SUM(A, T - A[i]) returns true

REDUCTION RUNTIME

1. Reduce3SUM→2SUM(A[1...n], T)
2. for i = 1...n
3.   T2 = T - A[i]
4.   if Solve2SUM(A, T2) return true
5. return false

• Θ(n) loop iterations
• Each iteration does Θ(1) + Runtime(Solve2SUM) work
• Runtime depends on implementation of Solve2SUM!
• Brute force: Θ(n) + Θ(n^2) = Θ(n^2)
• Binary search: Θ(n) + Θ(n log n) = Θ(n 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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>T = 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

• Correctness
• Invariant: if there exists a solution i’ < j’
  then i’ ≥ i and j’ ≤ j
• Exercise: fill in the proof details

FAST 3SUM TO 2SUM REDUCTION

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

• Since 3SUM is given a pre-sorted array, it takes Θ(n) time!
• We get runtime Θ(n log n) + Θ(n) Θ(n) = Θ(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: $O(n^{2.376})$.
  - This is a polylog factor faster than $O(n^2)$.
  - ... we suspect there is no solution faster than $O(n^{2+o(1)})$.

**A TRIVIAL REDUCTION**

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

```
1. ReduceMultiplyToSquare(x, y)
2. a = ComputeSquare(x + y)
3. t = ComputeSquare(x - y)
4. return ((a - t) >> 32)
```

- **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 ≤ 3SUMZero. This would allow code re-use.

---

**3SUM ≤ 3SUMZERO**

- If and only if $\sum_{i=1}^n x_i = T$ by finding $\Sigma y = 0$.
- This suggests the following approach:

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

**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_3array3SUMZero_to_3SUMZero(A, B, C)
2. A' = concatenation of A, B, C
3. return Solve3SUMZero(A')
```

**PROGRESSIVELY HARDER WORKED EXAMPLES**
No correct way to get a zero sum!

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

\[
\begin{align*}
A' & = \text{concatenation of } A, B, C \\
A'[i] & = 10A[i] + 1 \\
B'[j] & = 10B[j] + 2 \\
C'[k] & = 10C[k] + 3 \\
\end{align*}
\]

\[
A' = \text{concatenation of } D, E, F
\]

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 SolveArray(A, B, C)
- Case 1: Assume true is the correct output for SolveArray(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 Solve3SUMZero(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 cases...

Example case: Suppose, for contradiction, that \(A[i], A[j], A[k]\) are all in B


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

So, there is a sum \(i\) with one from each of A, B and C.
So, true is the correct output for SolveArray(A, B, C)

So, 
Solve3SUMZero ≤ SolveSUMZero
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.