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
- For a given slot $A[i]$
    - 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
```
2SUM_BruteForce(A[1..n], T)    
for l = 1.. n                 
for i = l.. n                 
    return true               
return false                 
``` 
Runtime $\Theta(n^2)$ 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:
```
2SUM_Improved(A[1..n], T)      
sort(A)                        
for i = 1.. n                  
  j = binary search for T-A[i]  
  in the subarray A[i..n]      
  if search is successful return true
return false                   
``` 
VS. linear search, which takes $O(n)$ time

What is this algorithm’s time complexity?
TIME COMPLEXITY

- \( \Theta(n \log n) \)
- \( \Theta(n) \) improved (A to T)
- \( \Theta(n) \) iterations
- \( \Theta(n \log(n)) \)

- **Loop:** Iterations \( \cdot \) work per iteration
  - \( n \) \( \cdot \) \( \Theta(\log(n)) = \Theta(n \log(n)) \)
- **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 \( A = [A[1], \ldots, A[n]] \) 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 \( T \), false otherwise

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

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} is a black-box subroutine that we call an **“oracle”**

- **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

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?

\text{Reduce3SUMto2SUM}(A[1..n], T)

for \( i = 1 \) to \( n \)

\( T2 = T - A[i] \)

if \text{Solve2SUM}(A, T2) return true

return false

T = 9

\( A = [1, 7, 3, 0, 2, -1, 5, 2] \)

\( i = 1 \)

\( T2 = 8 \), \( \text{Solve2SUM}(A, 8) \rightarrow False \)

\( i = 2 \)

\( T2 = 16 \), \( \text{Solve2SUM}(A, 16) \rightarrow False \)

\( i = 3 \)

\( T2 = 11 \), \( \text{Solve2SUM}(A, 11) \rightarrow False \)

\( i = 4 \)

\( T2 = 9 \), \( \text{Solve2SUM}(A, 9) \rightarrow False \)

\( i = 5 \)

\( T2 = 7 \), \( \text{Solve2SUM}(A, 7) \rightarrow False \)

\( i = 6 \)

\( T2 = 10 \), \( \text{Solve2SUM}(A, 10) \rightarrow False \)

\( i = 7 \)

\( T2 = 6 \), \( \text{Solve2SUM}(A, 6) \rightarrow True \)
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

• WTP: ∃j, A[i], A[k] that sum to T if and only if
• ∃A[m] such that Solve2SUM(A, T = A[m]) returns true

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

REDUCTION RUNTIME

1. Reduce 3SUM to 2SUM(A[1..n], T)
2. for i = 1 to 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^3) = Θ(n^3)
• Binary search: Θ(n) + Θ(n log n) = Θ(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

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>11</th>
<th>12</th>
<th>20</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
• Reduce 3SUM to 2SUM(A[1..n], T)
• sort(A)
• for i = 1 to n
•    T2 = T - A[i]
•    if Solve2SUM(A, T2) return true
• return false

• Since 2SUM 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:
  \[ O(n^2 \log \log n)^{1/4} \log^2 n \]
  - 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: \( xy = \frac{(x+y)^2 - x^2 - y^2}{4} \)
- This allows us to show that \textbf{Multiplication is Squaring}

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

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

A MEDIUM REDUCTION

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

\textbf{3SUM} \( \leq \textbf{3SUMZero} \)

- 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)
    for i = 1...n
    for j = i+1...n
      return Solve3SUMZero(A)
  ```

  - Given an oracle that solves \textbf{3SUMZero}, let's solve \textbf{3SUM}
  - Can we find \( \sigma x_i = T \) by finding \( \Sigma y_i = 0 \)?

A HARD REDUCTION

- \textbf{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 \textbf{3SUMZero}

  ```
  1. TRY_reduce_3array3SUMZero_to_3SUMZero(A, B, C)
  2. A' = concatenation of A, B, C
  3. return Solve3SUMZero(A')
  ```

  - Is this reduction correct?
  - Problem: \textbf{Solve3SUMZero} might choose \( \geq 2 \) elements from the same array!
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

Add +1
Add +2
Add -3

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!

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 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,
called many-one reductions
  - (also known as polynomial transformations or Karp reductions)
- We will many examples of these in the section on intractability.