Problem solving 1

- There are many problems that use sorting as like a pre-step.
- This is where you would need to pick a sorting algorithm based on various trade offs.

- Problems we will look at:
  - sum to target problem
  - Intersection of two sorted arrays

sum to target problem

- Input:
  - An array of numbers
  - Target number

- Output:
  - Return any pair of numbers from the input that add up to the target number.

  - Example 1:
    - Input: (2, 7, 16, 25) and target = 27
    - Output: 2, 25

  - Example 2:
    - Input: (2, 7, 16, 25) and target = 34
    - Output: empty set

sum to target problem

- What are our options for a solution?

- Brute Force?

```
for each number num1 in input
        num2 = target – num1
        Linear search for num2 in input.
        if found
                return (num1, num2)

return empty set
```

Time complexity: ?

Space: ?

What about a better option?

sum to target problem

- How about if we sort the input?
  - What will that cost?

```
sort input

for each number num1 in input
        num2 = target – num1
        Binary search for num2 in input.
        if found
                return (num1, num2)

return empty set
```

Time complexity: ?

Space: ?

Any other option after having sorted the array?

sum to target problem

Once we sort the array, we could use a 2 indices (pointers).

Traverse sorted array from both ends, each time summing the two numbers and comparing with target.

Input: 7 9 11 14 16 17 21 26
Target: 25

leftElement = 7	rightElement = 26	sum = 33
leftElement = 7	rightElement = 21	sum = 28
leftElement = 7	rightElement = 17	sum = 24
leftElement = 9	rightElement = 17	sum = 26
leftElement = 9	rightElement = 16	sum = 25 **Found a solution (9, 16)**
Answer found, we are done. Can continue if more pairs r needed. If so, ++ leftIndex or -- rightIndex.		
leftElement = 9	rightElement = 14	sum = 23
leftElement = 11	rightElement = 14	sum = 25 **Found a solution (11, 14)**
leftElement = 11	rightElement = 11	leftIndex == rightIndex, we scanned the entire input array

sum to target problem

- Once we sort the array, we could use 2 indices (pointers).
  - Traverse sorted array from both ends, each time summing the two numbers and comparing with target.
- Input: 7 9 11 14 16 17 21 26 Target: 25

```
sort input

leftIndex = 0;
rightIndex = n-1;                                   // n is size of input

while leftIndex < rightIndex

        sum = input [ leftIndex ] + input [ rightIndex ]

        if sum == target
                return (input [ leftIndex ], input [ rightIndex ])
        else if ( sum > target )                    // sum is bigger, so we want smaller numbers
                -- rightIndex                       // we move right index towards the left
        else
                ++ leftIndex                        // we want bigger numbers

return empty set
```

Time complexity: Sorting + looking for solution : ?

Space: ?

sum to target problem

- Any other options ?

sum to target problem

- Using a hash table

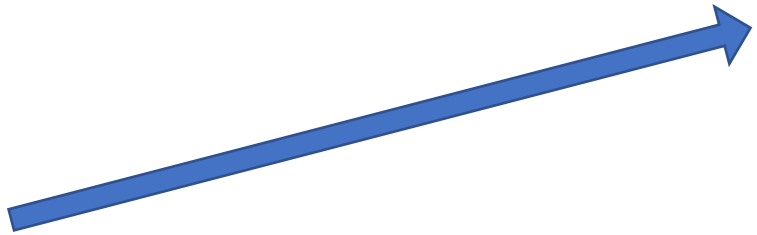
- Would the following work, or do u see any issues?

```
HashTable HT;
for each number num1 in input
        HT.Insert( num1 )

for each number num1 in input
        num2 = target  - num1
        if  HT.Contains ( num2 )
                return (num1, num2)
 return empty set
```

sum to target problem

- Lets look at the bug here:
  - Consider this input: 7 2 1 8 25 3 Target: 16

```
HashTable HT;
for each number num1 in input
        HT.Insert( num1 )

for each number num1 in input
        num2 = target  - num1
        if  HT.Contains ( num2 )
                return (num1, num2)
 return empty set
```



HT
2
25
7
1
3
8

num1	num2	num2 In HT?
7	9	no
2	14	no
1	15	no
8	8	yes

sum to target problem

- We will fix the bug:
  - input: 7 2 1 8 25 3 Target: 16

```
HashTable HT;

for each number num1 in input
        num2 = target  - num1
        if  HT.Contains ( num2 )
                return (num1, num2)
        else
                HT.Insert( num1 )
 return empty set
```

num1	num2	num2 In HT?	HT
7	9	no	7
2	14	no	7,2
1	15	no	7,2,1
8	8	no	7,2,1,8
25	-9	no	7,2,1,8,25
3	13	No	7,2,1,8,25,3

Time complexity: ?

Space: ?

sum to target problem (summary)

Option	Time	Space
Brute force	O (N^2)	O (1)
Sort, then binary search	O (N log N) + O (N log N) = O (N log N)	O (1) if in-place sort.
Using 2 pointers	O (N log N) + O (N) = O (N log N)	O (1) if in-place sort.
Hash table	O (N)	O (N)

LAB 1 of 2 (sum to target problem)

- Write a function **GetPairSum** (input, target)

- Input:
  - An array of numbers: 7 2 1 8 25 3
  - Target number: 16

- Output:
  - Return any pair of numbers from the input that add up to the target number, if no such pair exists, return an empty pair.

  - Example 1:
    - Input: (2, 7, 16, 25) and target = 27
    - Output: 2, 25

  - Example 2:
    - Input: (2, 7, 16, 25) and target = 34
    - Output: empty set

Intersection of two sorted arrays

- Lets say we need to find the intersection of two sorted arrays.
- Many ways to do this.

- Use case?
  - ?

- **Brute force**

```
for each element a in array 1
        search a in array 2
        if found
                add to result collection
```

  - Time complexity ?
  - Space complexity ?

Intersection of two sorted arrays

- But of course, we wont use brute force

- What are some other options?

Intersection of two sorted arrays

- What are some other options?

  - Traverse one and do binary search on the other.

```
for each element a in array1                    // or if array2 , then use array1 below for binary search
        if ( BinarySearch ( array2, a ) )
                add to result collection
```

Assume one array is size *m*, other is size *n*

  - Time complexity: ?

  - Space complexity ?

Would you have a preference on which array to traverse and which one to run a binary search on?

Intersection of two sorted arrays

- Another option : Using 2 indices (one in each array). Traverse both arrays, looking for common elements.

  array1: 1 4 7 11
  array2: 5 7 9 11

```
index1 = 0
index2 = 0

while  index1  < array1_Length    and    index2  < array2_Length

        if  array1 [ index1 ] < array2 [ index2 ]
                    ++ index1
        else if  array1 [ index1 ] > array2 [ index2 ]
                    ++ index2
        else
                     add to result collection
                    ++ index1  and  ++ index2
```

What's the space complexity of pseudocode above?

What's the time complexity of pseudocode above?

Intersection of two sorted arrays

```
while  index1  < array1_Length   and index2  < array2_Length

    if  array1[ index1 ] < array2 [ index2 ]
            ++ index1
    else if  array1[ index1 ] > array2 [ index2 ]
            ++ index2
    else
             add to result collection
            ++ index1  and  ++ index2
```

Worst case: Traverse both arrays fully → O (m + n)

Best case: Have to traverse only one array → O (m) or O (n), depending on which array is traversed.

Lets see what sort of input would lead to the worst and best case.

Intersection of two sorted arrays

- Worst case: O (m + n)
  - Traverse both arrays fully… last element in both arrays is the same.

    array1: 1 4 7 11

    array2: 5 7 9 11

- Best case: O (m) or O (n)
  - You end up traversing only one of the arrays

    array1: 1 4 7 11

    array2: 15 17 19 25
  - Which array would you traverse fully for the inputs above?

Intersection of two sorted arrays

- Question for u all:
  - Which one do u think will give u better runtime for finding the intersection of two sorted arrays
    A. Traverse one and run a binary search on the other?
       - O (m log n)
    B. Using 2 indices : Linear traversal of both arrays, looking for common elements?
       - O (m + n)
    C. Cannot tell without more information.
    D. Both approaches, binary search and 2 indices, are about the same

Intersection of two sorted arrays

Lets take some input sizes and see how the math works out.

Let m = 1000,000 and
n = 2000,000

- Traverse one and run a **binary search** on the other?
  - O (m log n)
  - 1000,000 * log (2000,000)
  - 1000,000 * 21
  - 21,000,0000

- **Linear traversal of both arrays**, looking for common elements?
  - O (m + n)
  - 1000,000 + 2000,000
  - 3,000,000

- Which one gives u better performance?

Intersection of two sorted arrays

Let m = 1,000,000
n = 20,000,000,000

- Traverse one and run a **binary search** on the other?
  - O (m log n)
  - 1000,000 * log (20,000,000,000)
    - Now, log (2 * 10,000,000,000) = log (2) + log(10,000,000,000)
    - = log (2) + log(10 * 1,000,000,000)
    - = log (2) + log(10) + log (1,000,000,000)
    - approx = 1 + 3 + 30 = 34
  - So, 1000,000 * log (20,000,000,000) = 1000,000 * 34 = 34,000,000

- **Linear traversal of both arrays**, looking for common elements?
  - O (m + n)
  - 1000,000 + 20,000,000,000
  - approx 20,000,000,000 (can ignore 1000,000)

- Which one gives u better performance in this scenario?

Intersection of two sorted arrays

Could we use a hash table to find the intersection?

If yes, what would be:

Time complexity ?

Space complexity?

Pseudocode on next slide

Intersection of two sorted arrays

Using a hash table

```
for each element a in array1                ← O ( m )
      Insert a in HT

for each element a in array2                ← O ( n )
      if  HT.Contains( a )                  ← O ( 1 )
                add to result collection
```

Time complexity : O (m + n)

Space : O (m)

LAB 2 of 2 (intersection of 2 sorted arrays)

- Input:
  - Two sorted arrays (or unsorted, and call sort in initialization step).

- Output:
  - Collection of elements that form the intersection of two given arrays

- Write this function using binary search
  - ***FindIntersection_BinarySearch***(array1, array2)