# Lecture 5
# Searching & Others

Where did I put my stuff…..?

# Lecture Overview

- **Searching:**
  - Linear Searching
  - Binary Searching
    - Recursive Binary Searching

- **Other:**
  - Direct Addressing Table
  - Counting Sort

# Search Algorithms

- Sometimes we need to locate a particular value in an array:
  - Known as **searching problem**

- Objective of a search algorithm:
  - Returns the **index** of the first occurrence of the target value
    - Returns a **–1** if value cannot be found

  - You can then use this index to access the value

# Search Algorithms : **Unordered List**

- Unordered Lists:
  - i.e. unsorted lists, value can appears **anywhere**

- The only way to search an unordered list is by a **linear (sequential)** search
  - Look at each element until you find what you want
  - Return the index for that element, or –1 if you've reached the end of the array

# Linear Search: **Examples**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **a** | 2 | 1 | 8 | -7 | 15 | -3 | 6 | 19 | 42 | 0 |

- ***`linearSearch`*** `( a, 10, 19 )` ➔ **7**
  - Look for **19** in array ***a*** with 10 elements

  ➔ Found at index **7**

- ***`linearSearch`*** `( a, 10, -3 )` ➔ **5**

- **`linearSearch( a, 10, 33 )`** ➔ **-1**
  - **-1** indicates that **33** is not found

# Linear Search: **Implementation**

```
int linearSearch( int a[], int N, int X )
{
    int idxOfX = -1, i;

   for ( i = 0; (i < N) && (idxOfX == -1); i++ ) {
        if ( a[i] == X )
            idxOfX = i;
   }

    return idxOfX;
}
```

- Question:
  - What is the **worst case** Big-O for this algorithm?

# Search Algorithms: **Ordered List**

- Ordered list:
  - Consider the following array

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| `a` | -7 | 2 | 14 | 38 | 72 | 77 | 89 |

  - If we look for the value **25**
    1. Go through each element
    2. Once we hit **38**, since **38>25** and the list is **ordered**
       ➔ we know that 25 is not in the list
    3. **Can terminate the search early**

# Linear Search: **Implementation**

```
int linearSearch2( int a[], int N, int X ) {
    int idxOfX = -1, i;

   for ( i = 0; (i < N) &&
                (idxOfX == -1) &&
                (a[i] <= X); i++ ) {

       if ( a[i] == X )
           idxOfX = i;
   }

    return idxOfX;
}
```

- Question:
  - What is the worst case Big-O?

# Ordered List : Another Look

- This algorithm is not very smart
  - If the target **x** is the biggest value in the list, or **x** is not in the list
    
    ➔ Same as the unordered linear search algorithm `linearSearch()`
- We need a **smarter** algorithm!

# Binary Search : **Idea**

- Let's play a game:
  1. I pick a mystery number between [1…100]
  2. You make a guess
  3. I will tell you whether your guess is:
     - **Correct**, **Too high** or **Too low**
  4. Repeat until you guessed my number

- How do you beat this game in 7 guesses or less?

# Binary Search: **Idea**

- **Observations:**
  - In a **sorted array**, if the target **X** is lesser than **`array[P]`**
    - target X **cannot be found** in positions **`[P, size-1]`**
  - Similar argument for target X **larger than `array[P]`**

- Given a **sorted** array:
  - Compare **middle element** of the array with target:
  - if ( array[middle] == target ) target **found!**

    else if ( array[middle] < target )

    target is **in the upper halve of the array!**

    else target is **in the lower halve of the array!**
  - "Zoom" into upper/lower halve of the array and apply the same idea

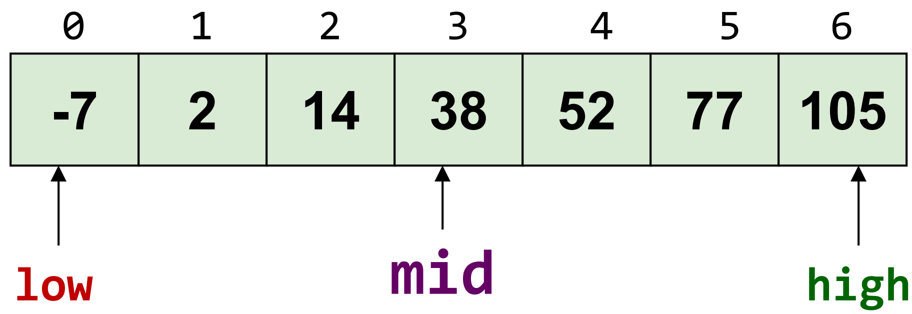
# Binary Search: **Design**

- How do we indicate the array portion that is currently under inspection?
  - Use two indices: `low` and `high`
  - `Array[low..high]` is the active portion
  - Additional index `mid` to indicate the mid-point

- How do we "zoom" into lower/upper half?
  - Reduce the `high` index to zoom into lower halve
  - Increase the `low` index to zoom into upper halve

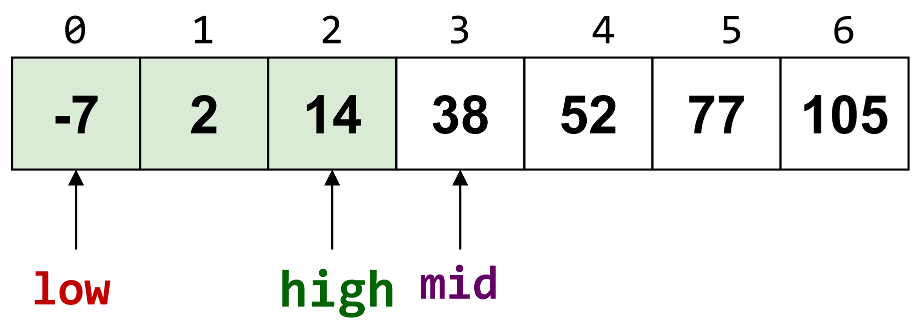
- How do we know the target does not exist?
  - If  the array portion shrink down to zero element
    ➔ not found!

# Binary Search (**Searching for 25**)



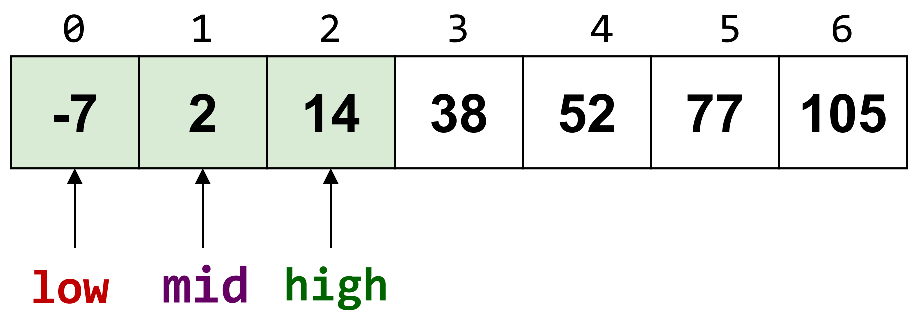

mid
= (low + high) / 2
= (0 + 6) / 2
= 3

Array portion being searched

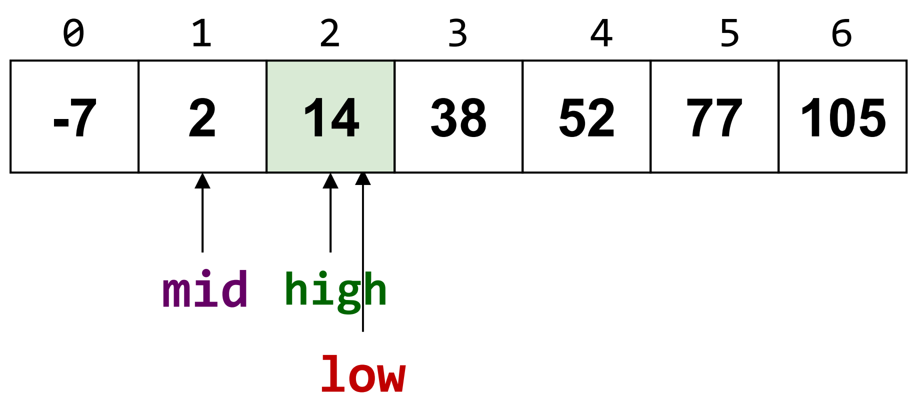



Since **x[mid]=38 > 25**, set **high = mid - 1**

# Binary Search (**Searching for 25**)

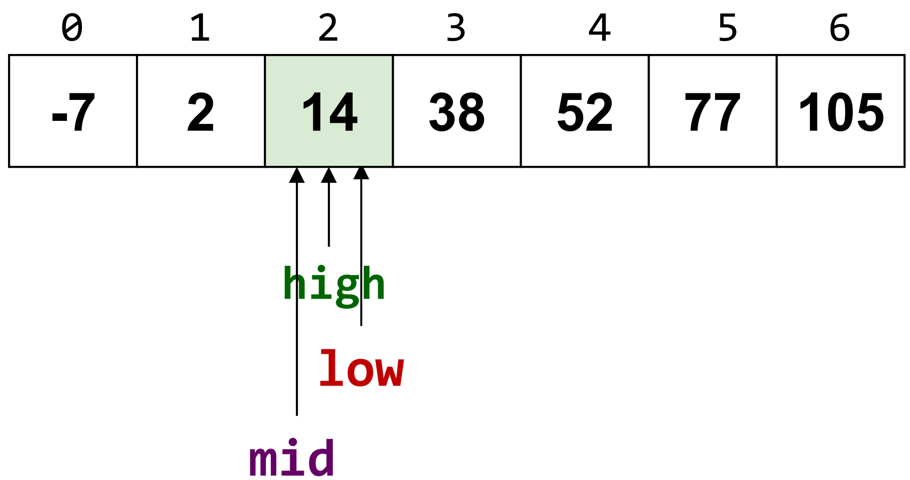



mid
= (low + high) / 2
= (0 + 2) / 2
= 1

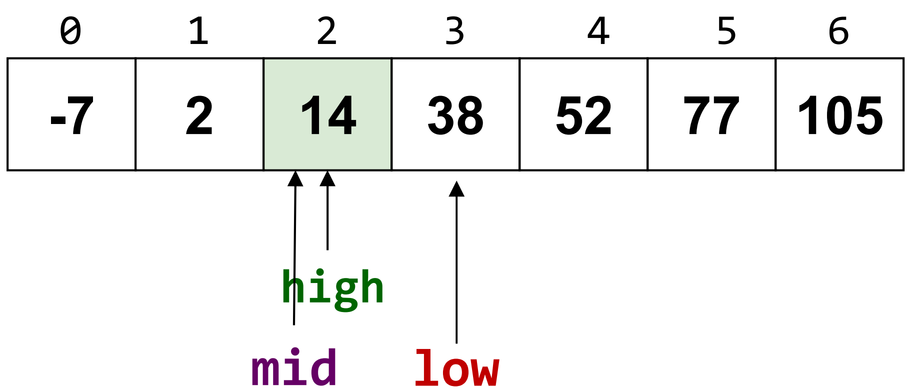



Since `x[mid]=2 < 25`, set `low = mid + 1`

# Binary Search (**Searching for 25**)




mid
= (low + high) / 2
= (2 + 2) / 2
= 2




Since `x[mid]=14 < 25`, set `low = mid + 1`

Since `low > high`, we can conclude 25 is not found

# Binary Search: **Implementation**

```
int binarySearch( int a[], int N, int X) {
   int idxOfX = -1, mid, low, high;

   low = 0;
   high = N - 1;
   while ( (low <= high) && (idxOfX == -1) ) {
        mid = (low + high) / 2;

      if ( a[mid] == X )
           idxOfX = mid;
      else if ( a[mid] < X )
           low = mid + 1;
      else
           high = mid - 1;
   }

   return idxOfX;
}
```

The whole array is "active" initially

Found!

Target should be in upper halve

Target should be in lower halve

- With this algorithm, what is the Big-O for the **worst case**?

# Problem: **Recursive Binary Search**

- How do we define binary search as a recursive function?

- Slight change of the function header for simplicity:

```
int binarySearch(int a[], int low, int high, int X);
```

  - low and high are the indices for the leftmost and rightmost element of the array portion

# Binary Search: **Recursion!**

```
int binarySearch( int a[], int low, int high, int X) {
    int mid;

    if (low > high)
        return -1;

    mid = (low + high) / 2;

    if (a[mid] == X){
        return mid;
    } else if (a[mid] < X ){


    } else {


    }

}
```

- Fill in the blank of the two indicated places?

# Problem: **Fit in the box!**

- Given:
  - An array of N unsorted positive integers
  - S, a specific sum you want to achieve

- Your task:
  - Find a pair numbers in the array that adds up to S

- Example:

| | | | | | | |
|---|---|---|---|---|---|---|
| **29** | **10** | **14** | **37** | **13** | **7** | **6** |

  - Find a pair of numbers that add up to **20**

# OTHER SEARCHING / SORTING RELATED PROBLEMS

# Direct Addressing

- Motivating Example:
  - Suppose you are writing an app to store the information of bus services in Singapore
  - e.g. find out the route of bus 98, where is the last stop for bus 989? etc.
- Assumptions:
  - We consider only the basic bus service and ignore "specialized" route like 98M, 143e etc
  - The bus services run from 1 to 999 only
- How do we store the information for ease of retrieval?

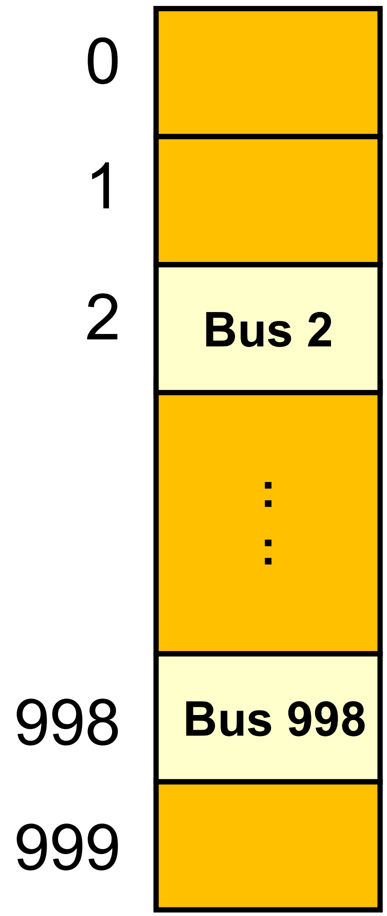
# Possible Solutions

- What's the Big-O for the "standard" approaches:
    1. **Linear Searching** on unordered list
    2. **Binary Searching** on ordered list

- Can we exploit the key characteristics of this problem and come up with **better** solution?
    1. Limited range [1 to 999]
    2. At most one item at a location

# Direct Addressing Table

- Declare an array of 1000 elements:
  - Index from [0 …. 999]
  - Each element can be a **bus information structure**
- How do we:
  - Get the information of Bus 143?
  - Update the information of Bus 998?
- What is the time complexity?



# Direct Addressing Table: Limitations

- The value used to access a location is known as a **key**
  - Bus number is the key in this case

1. Keys must be **non-negative integer** values
   - What happen for key values 151A and NR10?
2. Range of keys must be **small**
3. Keys must be **dense**
   - i.e. not many gaps in the key values

# Counting Sort

- Motivating Example:
  - Suppose we want to print out the list of students based on their midterm score
  - Can we do this efficiently?
- For simplicity:
  - Consider only an array of **N** midterm score
  - Score is between **0 to 10**

# Possible Solution and New Idea

- The straightforward way is to:
  1. Sort the student record based on midterm score
  2. Print the sorted list of students

- Time complexity is dominated by the sorting
  - **O($N^2$)** using the sorting algorithms we know

- Can we do better **in this case**?
  - Limited range of possible values [0…10]
  - Ideas learned from direct addressing table?

# Counting Sort: **Idea**

- We first count the frequency of the scores

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **score** | **10** | **5** | **7** | **3** | **5** | **7** | **7** | **9** | **2** | **5** | **7** | **3** |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **freq** | **0** | **0** | **1** | **2** | **0** | **3** | **0** | **4** | **0** | **1** | **1** |

```
//Given score[], freq[], N is the number of scores

//Figure out the simple loop to produce freq[]?
```

# Counting Sort: **Cumulative Frequency**

- With a score **X**, the rough final position:
  - Must be after all scores from **0… X-1** (this is the **cumulative frequency**)
- Can easily produce the cumulative frequency:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **freq** | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 0 | 1 | 1 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **cfreq** | 0 | 0 | 1 | 3 | 3 | 6 | 6 | 10 | 10 | 11 | 12 |

```
//Figure out the code, given freq[] and cfreq[]?
```

# Counting Sort: **Finally**

- With the cumulative frequency, we can figure out the exact final position of each score:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **s** | **10** | **5** | **7** | **3** | **5** | **7** | **7** | **9** | **2** | **5** | **7** | **3** |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **cf** | **0** | **0** | **1** | **3** | **3** | **6** | **6** | **10** | **10** | **11** | **12** |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **final** | **2** | **3** | **3** | **5** | **5** | **5** | **7** | **7** | **7** | **7** | **9** | **10** |

```
//Figure out the code, given score[], cfreq[] and final[]
```

# Counting Sort: **All Together Now**

```
void counting_sort(int score[], int N, int final[])
{
    int freq[11] = {0};
    int cfreq[11] = {0};

    //1. Compute Frequency



    //2. Compute Cumulative Frequency



    //3. Produce final position




}
```

- What is the Big-O?

# Summary

- Searching:
  - Unsorted List: Linear Search
  - Sorted List: Binary Search

- Special cases:
  - Direct addressing table
  - Counting sort