Being Clever while solving problems
Example: SLMP

Tanvir Ahmed, PhD

Department of Computer Science,

University of Central Florida

Motivation

- You already learned various data structures, and programming logics from your previous programming courses.
- However, in CS2, your target should not be just produce the correct output for a given problem.
  - You have to be clever while solving problem.
  - The code you are writing should be efficient
  - Instead of just a naïve solution, you need to figour out whether you can decrease the run-time of the code you are writing.
- As part of it, in this note we will try to understand how a simple problem can be solved in many different ways, the concept of performance, and how we can improve the solutions to do it with less number of steps.
- Let’s look at the example in the next slide

Sorted List Matching Problem (SLMP)

- Given two sorted lists of distinct numbers, output the numbers common to both lists.
- How would you attack the problem?
  - For each number on list 1, do the following:
    - a) Search for the current number in list 2.
    - b) If the number is found, output it.

- If a list is unsorted, steps a and b (which is simply a linear search) might take n steps (n = number of elements in list 2)

SLMP brute force solution O(n^2)

i	10
	20
	30
	35
	40
	45
	50
	55
	60
	65

j	12
	15
	20
	25
	40
	50
	52
	60
	62
	70

Let’s see how you would do it in real life:

- For each item in list1, search for it in list 2
  - If you find it, print it
- So, take 10 from list1, and search for 10 in list2.
- Take 20 from list1 and lok for 20 in list2.
  - As soon as you find 20, print it and don’t keep looking for 20
- And so on

SLMP

- If you don't use the information that the list is sorted, we can do a brute force solution:

```
static void printMatchesN2(int list1[], int list2[])
{
    int i,j;
    for (i=0; i < list1.length; i++)
    {
        for (j=0; j<list2.length; j++) //linear search
        {
            //cntlinearsearch++;
            if (list1[i] == list2[j])
            {
                System.out.println(list1[i]);
                break;
            }
        }
    }
}
```

- How many steps it might take in total, if the size of list1 = n and size of list2 = n?
- n^2 => $O(n^2)$

SLMP with Binary Search O(n log n)

- But we know both lists are already sorted.
- Thus, we can use binary search in step a.
    - It means for each number in list1, we do a binary search for that number in list2
- A binary search takes about log n steps.
- We have to repeat n times. So, total around *n log n.* Much better than ***n^2***.

```
static void printMatchesBinIter(int list1[], int list2[])
{
    int i,j;
    for (i=0; i < list1.length; i++)
    {
        if(binSearchIter(list2, list1[i]) != -1)
            System.out.println(list1[i]);
    }
}
```

Going back to SLMP

Enhance your code to find common items in two arrays (n * log n)
So, just using binary search in our last SLMP code can result in n * log n as binary search works for log n and we want to use binary search n times.
#O(n * log n)

#Can you even improve it further???:

O(n) SLMP

i	10		j	12
	20			15
	30			20
	35			25
	40			40
	45			50
	50			52
	55			60
	60			62
	65			70

Let’s see how you would do it in real life:

- You would compare 10 with 12. Immediately you would know that they are not matching.
  - As 10 <12, you will take the next number from list1 and compare 20 (from list1) with 12
  - As 20>12, you are sure that there will not be any number less than 12 in list1, so go the next number for list2.
  - So, compare 20 (from list1) with 15
    - For the same reason go to the next number (20) of list 2
  - Now, we found a match!
    - So, print the number and we go to the next numbers for both of the list
  - We repeat this until we reach to the end of any one of the lists.
- So, in this technique, we are not repeating the same number from the list again as part of searching!
- See the more formalized version of the algorithm in the next slide.

O(n) SLMP

i	10
	20
	30
	35
	40
	45
	50
	55
	60
	65

j	12
	15
	20
	25
	40
	50
	52
	60
	62
	70

1. Start two “trackers”, one for each list, at the beginning of both lists. (let’s say i for list1 and j for list2)
2. Repeat the following steps until one tracker has reached the end of its list (until i or j reaches to its corresponding array length).
   a. Compare the two items that the markers are pointing at. (compare list1[i] with list2[j])
   b. If they are equal, output the number and advance BOTH (i++ and j++);
   c. If they are NOT equal, simply advance the tracker pointing to the number that comes earlier one spot.

      (if list1[i]<list2[j] then i++
      else j++)

This will improve the run time and will result in 2n steps. => O(n) => Linear time

O(n) SLMP

i

0	10
1	20
2	30
3	35
4	40
5	45
6	50
7	55
8	60
9	65

j

0	12
1	15
2	20
3	25
4	40
5	50
6	52
7	60
8	62
9	70

Example:

Initially, i=0, j=0;

Compare list1[i] (10) with list2[j] (12)

As 10<12

i++

Now, i = 1

Output:

O(n) SLMP

0	10
1	20
2	30
3	35
4	40
5	45
6	50
7	55
8	60
9	65

i

j

0	12
1	15
2	20
3	25
4	40
5	50
6	52
7	60
8	62
9	70

Example:

Current value of i=1, j=0;

Compare list1[i] (20) with list2[j] (12)

As 20 > 12

j++

Now, i = 1, j=1

Output:

O(n) SLMP

	0	10
i	1	20
	2	30
	3	35
	4	40
	5	45
	6	50
	7	55
	8	60
	9	65

	0	12
	1	15
j	2	20
	3	25
	4	40
	5	50
	6	52
	7	60
	8	62
	9	70

Example:

Current value of i=1, j=1;

Compare list1[i] (20) with list2[j] (15)

As 20 > 15

j++

Now, i = 1, j=2

Output:

O(n) SLMP

0	10
1	20
2	30
3	35
4	40
5	45
6	50
7	55
8	60
9	65

i

j

0	12
1	15
2	20
3	25
4	40
5	50
6	52
7	60
8	62
9	70

Example:

Current value of i=1, j=2;

Compare list1[i] (20) with list2[j] (20)

As 20 == 20

Print 20

i++

j++

Now, i = 2, j=3

Output so far: 20

O(n) SLMP

	0	10
	1	20
i	2	30
	3	35
	4	40
	5	45
	6	50
	7	55
	8	60
	9	65

	0	12
	1	15
	2	20
	3	25
j	4	40
	5	50
	6	52
	7	60
	8	62
	9	70

Example:

Current value of i=2, j=3;

Compare list1[i] (30) with list2[j] (25)

As 30 > 25

j++

Now, i = 2, j=4

Output so far: 20

O(n) SLMP

	0	10
	1	20
	2	30
i	3	35
	4	40
	5	45
	6	50
	7	55
	8	60
	9	65

	0	12
	1	15
	2	20
	3	25
j	4	40
	5	50
	6	52
	7	60
	8	62
	9	70

Example:

Current value of i=2, j=4;

Compare list1[i] (30) with list2[j] (40)

As 30 < 40

i++

Now, i = 3, j=4

Output so far: 20

O(n) SLMP

0	10
1	20
2	30
3	35
4	40
5	45
6	50
7	55
8	60
9	65

i

j

0	12
1	15
2	20
3	25
4	40
5	50
6	52
7	60
8	62
9	70

Example:

Current value of i=3, j=4;

Compare list1[i] (35) with list2[j] (40)

As 35 < 40

i++

Now, i = 4, j=4

Output so far: 20

O(n) SLMP

	0	10
	1	20
	2	30
	3	35
	4	40
i	5	45
	6	50
	7	55
	8	60
	9	65

	0	12
	1	15
	2	20
	3	25
	4	40
j	5	50
	6	52
	7	60
	8	62
	9	70

Example:

Current value of i=4, j=4;

Compare list1[i] (40) with list2[j] (40)

As 40 == 40

Print 40

i++

j++

Now, i = 5, j=5

Output so far: 20 40

O(n) SLMP

Example:

and so on...

	0	10
	1	20
	2	30
	3	35
	4	40
i	5	45
	6	50
	7	55
	8	60
	9	65

j

0	12
1	15
2	20
3	25
4	40
5	50
6	52
7	60
8	62
9	70

Output so far: 20 40 .....

Here is the O(n) code to the SLMP problem!

```
static void slmplinear(int list1[], int list2[]) {

     int i = 0, j = 0;
     int m = list1.length, n = list2.length;

     // Go while we still have numbers in both lists.
     while (i < m && j < n) {
          //cnttwoTracker++;

          // Safe to advance list 1 pointer.
          if (list1[i] < list2[j]) i++;

          // Safe to advance list 2 pointer.
          else if (list2[j] < list1[i]) j++;

          // Match!
          else {
               System.out.println(list1[i]);
               i++;
               j++;
          }
     }
}
```

At each step in the while loop, either i or both are increasing, that results in one scan to each of the arrays in total.
So, total steps would be (n+m) which is O (n+m).

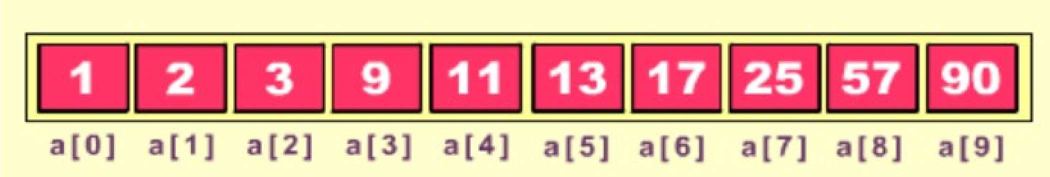
If both array size was n, then it would be 2n => O(n) linear time. (removing any constant factors and lower order terms will give you big – O O(n))

We will implement all of these approaches or I will show you and do some experiment to see how many steps are taken by each of these approaches!

The next couple of slides just for reviewing the C implementation of binary search and its run-time analysis for your reference. We will not go through this in the class as you have learned it in CS1

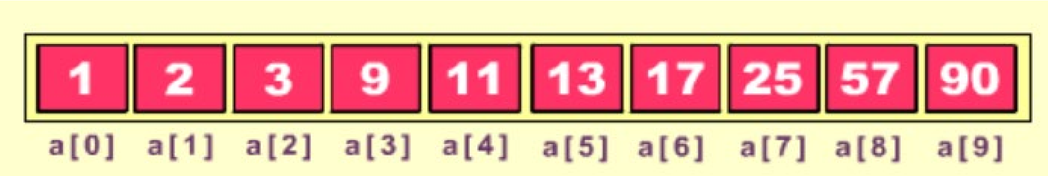
Binary Search

- If you know that the array is sorted, we can guess better what part of the array the data should be located
- For example see the following array:

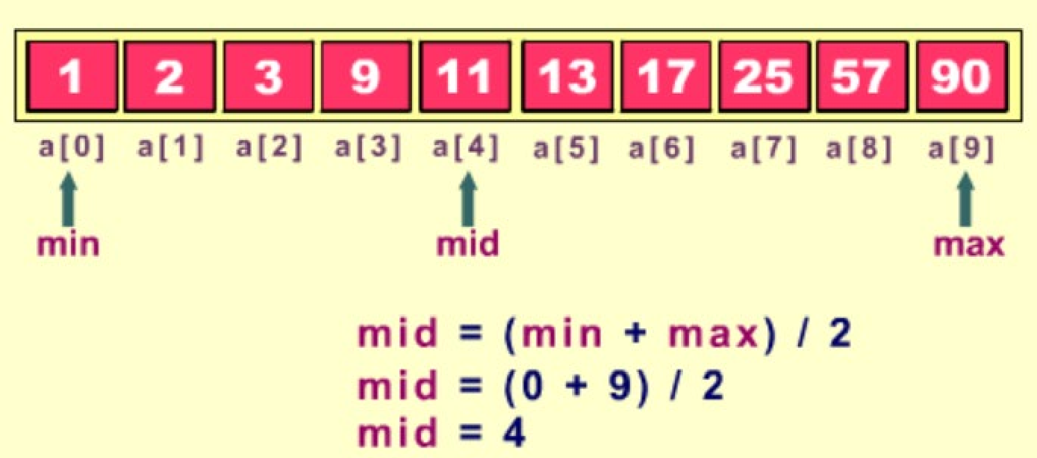


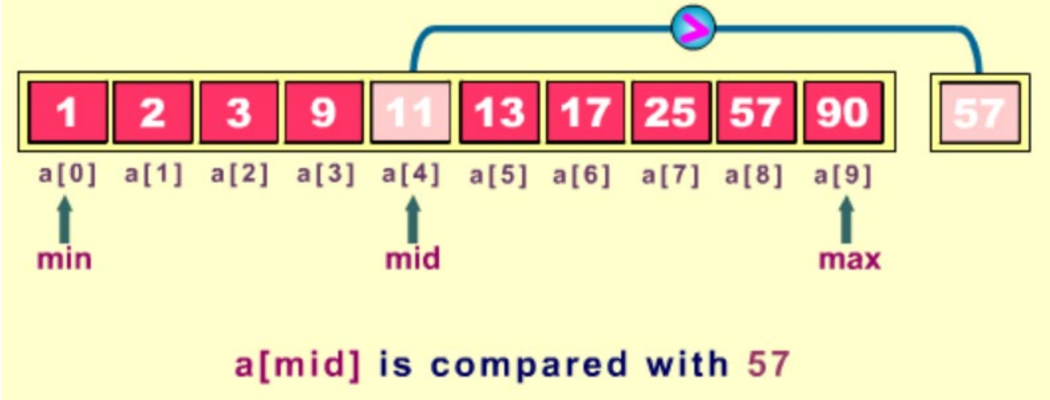
- If you want to search for 57, we can directly start our search in the upper half of the array
- That upper half of the array also can be treated as another array and we can even look upper half of that new array
- and so on….
- We divide our search space like this until we find the item or we are sure that our item does not exist
- So, what is the mid point of the above array?
  - (left most index + right most index)/2 = (0+9)/2 = 9/2 = 4
- So, we need two numbers, the ***low index and high index*** and calculate:
  - mid index = (low index + high index)/2
- What would be your mid point if your low=4 and high = 9?
  - (4+9)/2 = 6
- This approach of searching is very intuitive when searching in a sorted list.
  - Let's see an step by step example in the next slide

Binary Search Simulation

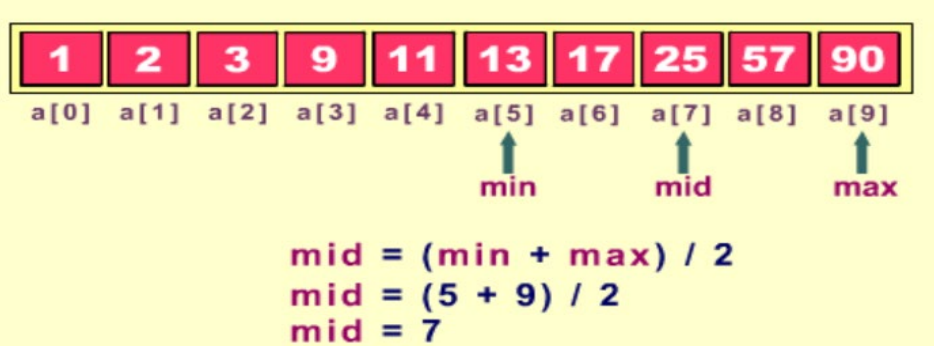


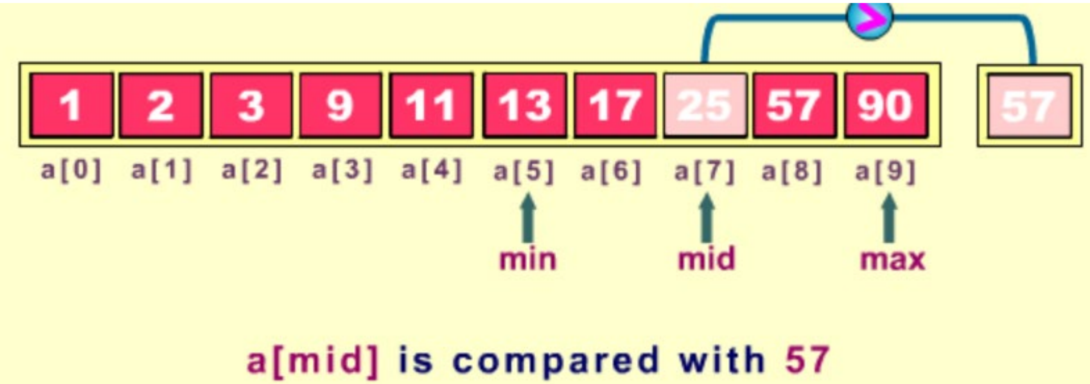
Suppose the data that is to be searched is 57



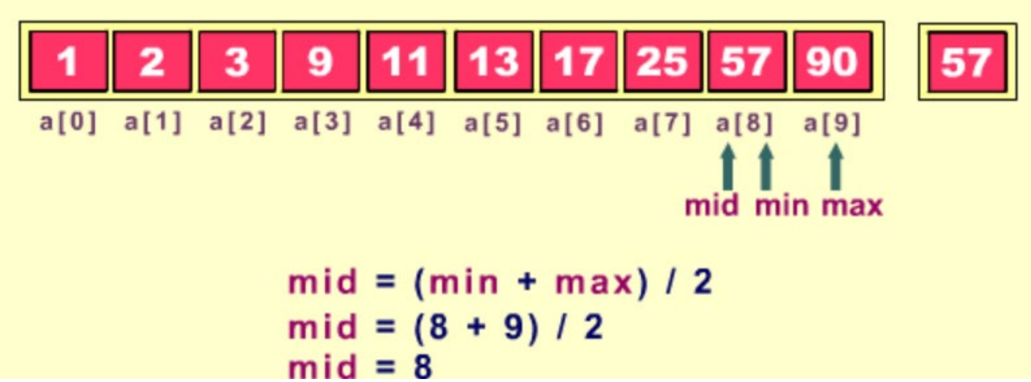


As 11 is less than 57, lower bound min is pointed to a[5]





As 25 is less than 57, lower bound min is pointed to a[8]



a[mid] is compared with 57

Search is successful as 57 is found

How about if the item search is 58?

Binary search. Search for item = 20 in the bellow array:

0	1	2	3	4	5	6
-15	18	20	25	30	35	112

- Lets work with the array index:

  l = 0 and h = 6. So, mid = (l+h)/2 = 3

- What is in Array[mid]?
  - It is 25.
  - It is > 20.
  - So, our item is actually before it.
  - So, our new h = mid-1 = 2.
  - New mid = (0+2)/2 = 1
  - Now, Array[mid] = 18 < 20
  - So, our item is actually after mid.
  - So, l = mid + 1 = 2
  - Array[mid] = 20 == item. So, return 1 (success)
- Let's try searching for 32
- **How would you know that the item is not available?**
  - **If low>high**

Binary search

- So, during the iteration, we update our low or high based on the condition
- Because we want to check the data in that part of the array and we want new mid

```
int binarySearch(int list[], int item, int len)
{
   int l = 0, h = len - 1;
   int mid;
   while (l <= h)
   {
      mid = (l + h) / 2;
      // Check if item is present at mid
      if (list[mid] == item)
         return mid;
       // If item greater, ignore left half
      if (list[mid] < item)
         l = mid + 1;

      // If item is smaller, ignore right half
      else
         h = mid - 1;
   }
   // if we reach here, then element was
   // not present
   return -1;
}
```

```
#include <Stdio.h>

int binarySearch(int list[], int item, int len)
{
    int l = 0, h = len - 1;
    int mid;
    while (l <= h)
    {
        mid = (l + h) / 2;
        // Check if item is present at mid
        if (list[mid] == item)
            return mid;
         // If item greater, ignore left half
        if (list[mid] < item)
            l = mid + 1;

        // If item is smaller, ignore right half
        else
            h = mid - 1;
    }
    // if we reach here, then element was
    // not present
    return -1;
}


int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int searchitem = 10;

    int result = binarySearch(arr, searchitem, 4);
    (result == -1) ? printf("Element is not present"
                            " in array")
                   : printf("Element is present at "
                            "index %d",
                            result);
    return 0;
}
```

Analyzing the number of steps by Binary Search

- Our search space starts at n,
- After 1 step search space is no bigger than n/2
- After 2 steps, the search space is no bigger than n/4
- After 3 steps, the search space is no bigger than n/8
- So, after k steps, the search space is no bigger than $n/2^k$
- The algorithm will stop after our search space is size 1 (once low exceeds high, this loop will not run again.
  - Let's solve the following equation for k, the number of steps this algorithm takes:
    - $n/2^k = 1$
    - $n = 2^k$
    - $K = \log_2 n$

- Thus, binary search takes no more than roughly **$\log_2 n$ steps. $O(\log_2 n)$**
- So, if we're searching in 2 million items, we will only make 20 comparisons, at most!
- HUUUGGEEE saving!

Binary search recursive code (we will do it after learning some recursions)

```
int binSearch(int *values, int low, int high, int searchval)
{
        int mid;
        if (low <= high)
        {
                mid = (low+high)/2;
                if (searchval < values[mid])
                        return binSearch(values, low, mid-1, searchval);
                else if (searchval > values[mid])
                        return binSearch(values, mid+1, high, searchval);
                elsereturn 1;
        }
        return 0;
}
```