Project Title: Implement binary search using 8086 assembly language.

Objective:

1. Learn the binary search algorithm
2. Implement the binary search algorithm
3. Search a specific value in a given number of sorted lists.
4. To find the position of a number and whether the element being searched is before or after the current position in the list.
5. To determine the searching time complexity and memory complexity.
6. Understand the "Divide and conquer" technique.

Theory:

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Given an array **A** of **n** elements with values or records A_0, A_1, A_2, A_3, …... A_{n-1} sorted such that $A_0 \le A_1 \le A_2 \le A_3, \ldots\ldots \le A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of **T** in **A**.

1. Set $\boldsymbol{L}$ to $\mathbf{0}$ and $\boldsymbol{R}$ to $\boldsymbol{n-1}$.
2. If $\boldsymbol{L} > \boldsymbol{R}$, the search terminates as unsuccessful.
3. Set **m** (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
4. If $A_m < T$, set ***L*** to ***m+1*** and go to step 2.
5. If $A_m > T$, set ***R*** to ***m-1*** and go to step 2.
6. Now $A_m = T$, the search is done; return ***m***.

This iterative procedure keeps track of the search boundaries with the two variables L and R. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above, floor is the floor function, and unsuccessful refers to a specific value that conveys the failure of the search.

Design:

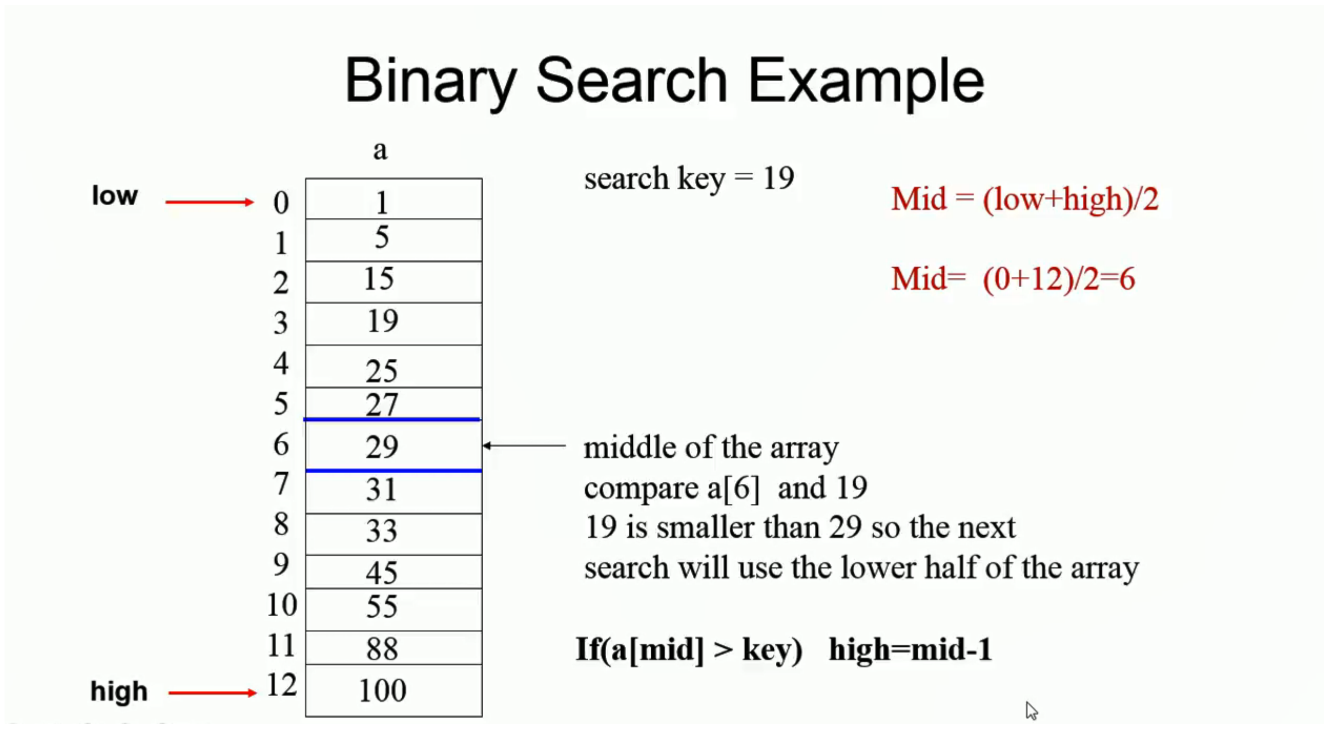
Search a key element in a list of 'n' 16-bit numbers using the Binary search algorithm:

- Input must be sorted array.
- Eliminates one half of the elements after each comparison

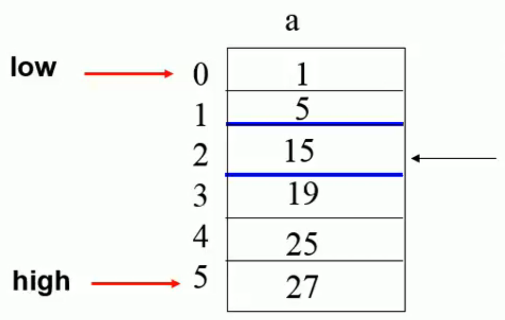
Algorithm

7. Locate the middle of the array
8. Compare the value at that location with the search key
9. If they are equal – done.
10. Otherwise, decide which half of the array contains the search key.
11. Repeat the search on that half of the array and ignore the other half.
12. The search continues until the key is matched or no elements remain to be searched.

Diagram and Example



Binary Search Pass 2



search key = 19

use this as the middle of the array
Compare a[2] with 19

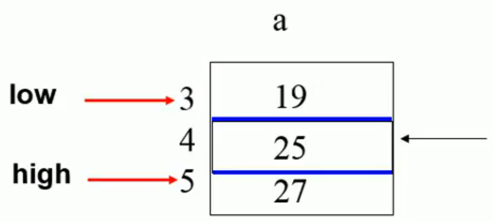
15 is smaller than 19 so use the top half for the next pass

Mid = (low+high)/2

Mid= (0+5)/2=2

Binary Search Pass 3

search key = 19

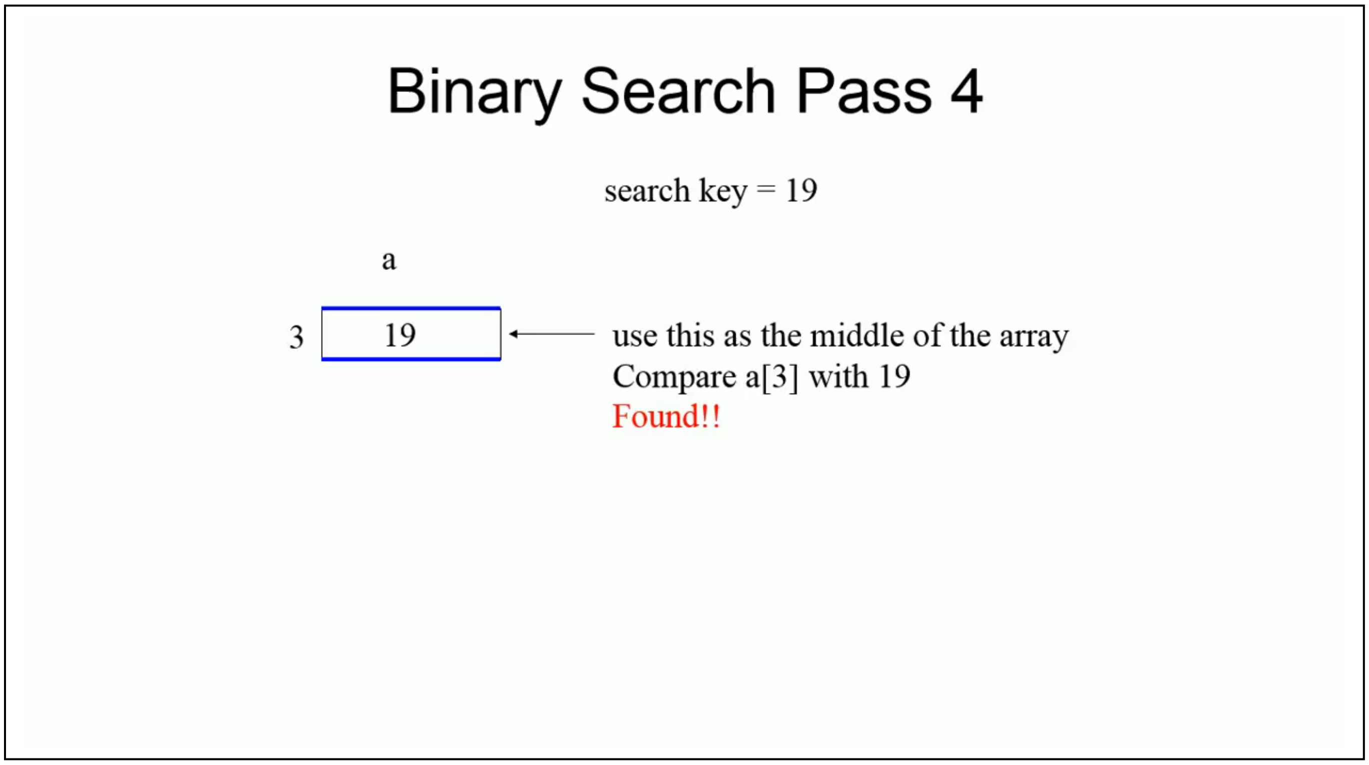


use this as the middle of the array
Compare a[4] with 19

25 is bigger than 19 so use the bottom half

Mid = (low+high)/2

Mid= (3+5)/2=4



Binary Search Pass 4

search key = 19

a

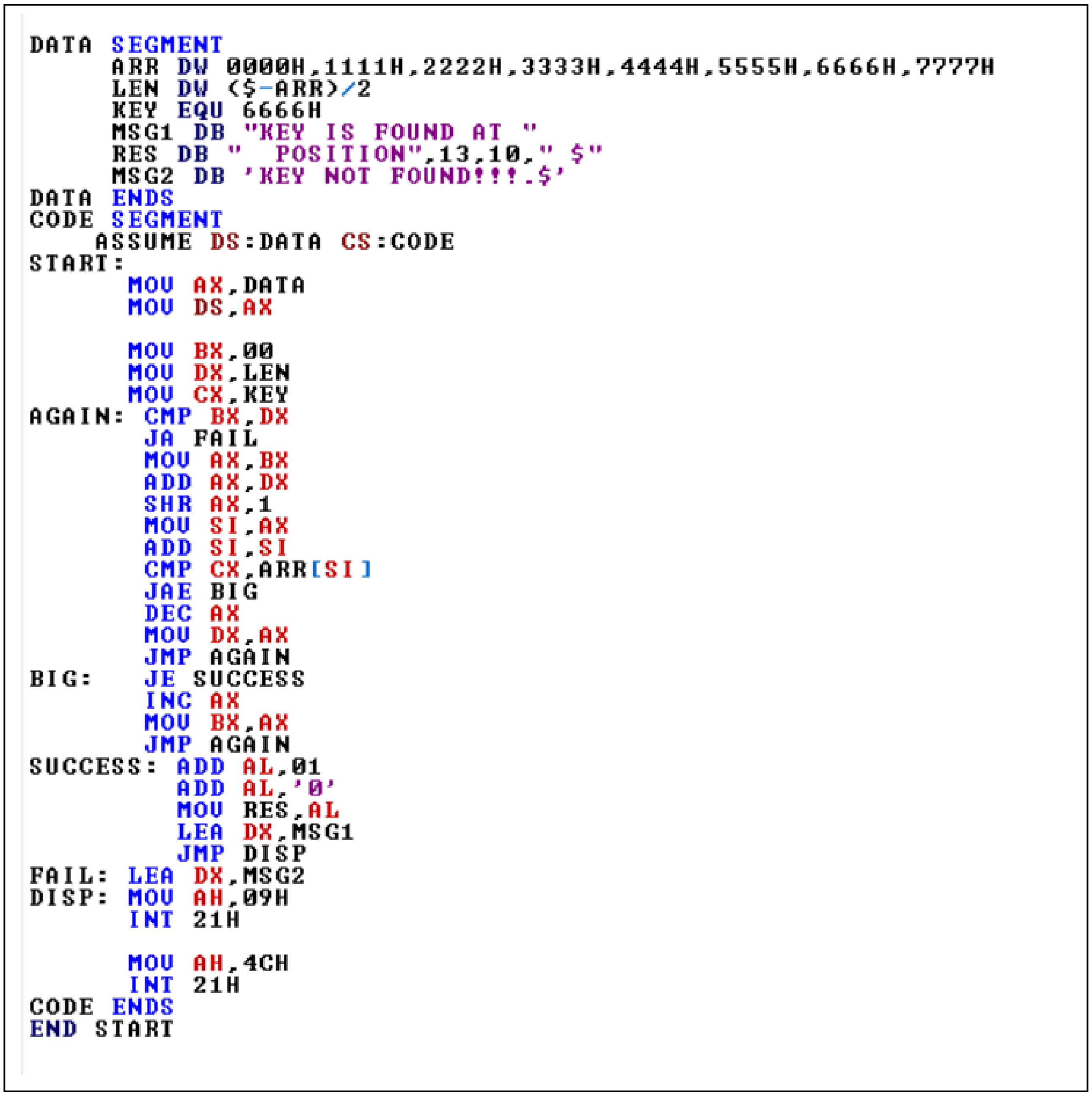
3 | 19 ← use this as the middle of the array

Compare a[3] with 19

Found!!

Implementation:

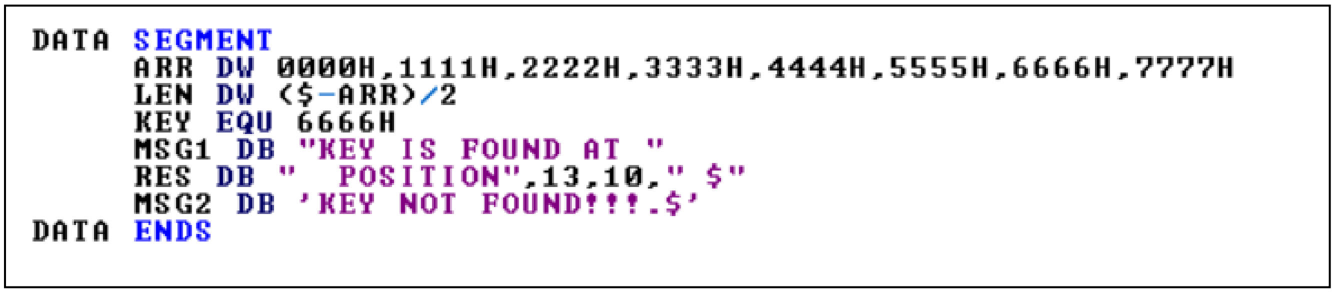
Assembly Code



```
DATA SEGMENT
     ARR DW 0000H,1111H,2222H,3333H,4444H,5555H,6666H,7777H
     LEN DW ($-ARR)/2
     KEY EQU 6666H
     MSG1 DB "KEY IS FOUND AT "
     RES DB "  POSITION",13,10," $"
     MSG2 DB 'KEY NOT FOUND!!!.$'
DATA ENDS
CODE SEGMENT
    ASSUME DS:DATA CS:CODE
START:
      MOV AX,DATA
      MOV DS,AX

      MOV BX,00
      MOV DX,LEN
      MOV CX,KEY
AGAIN: CMP BX,DX
       JA FAIL
       MOV AX,BX
       ADD AX,DX
       SHR AX,1
       MOV SI,AX
       ADD SI,SI
       CMP CX,ARR[SI]
       JAE BIG
       DEC AX
       MOV DX,AX
       JMP AGAIN
BIG:   JE SUCCESS
       INC AX
       MOV BX,AX
       JMP AGAIN
SUCCESS: ADD AL,01
         ADD AL,'0'
         MOV RES,AL
         LEA DX,MSG1
         JMP DISP
FAIL: LEA DX,MSG2
DISP: MOV AH,09H
      INT 21H

      MOV AH,4CH
      INT 21H
CODE ENDS
END START
```

DATA SEGMENT

DATA SEGMENT is the starting point of the Data Segment in a Program and DATA is the name given to this segment and SEGMENT is the keyword for defining Segments, Where we can declare our variables.



```
DATA SEGMENT
     ARR DW 0000H,1111H,2222H,3333H,4444H,5555H,6666H,7777H
     LEN DW ($-ARR)/2
     KEY EQU 6666H
     MSG1 DB "KEY IS FOUND AT "
     RES DB "  POSITION",13,10," $"
     MSG2 DB 'KEY NOT FOUND!!!.$'
DATA ENDS
```

DB – Define Byte (Size – 1 Byte)

DW – Define Word (Size – 2 Byte)

ARR DW **0000H,1111H,2222H,3333H,4444H,5555H,6666H,7777H** ……this line is a declaration of 16-bit Numbers Array initialized with 0000H,1111H,2222H,3333H,4444H,5555H,6666H,7777H……the numbers are separated by Comma (,).

LEN DW $-ARR is used to Save the Length of the Array which will be generated by $-Name of the array i.e. $-ARR.

KEY EQU 7777H is used to save given KEY to be Searched in the Array and is equal to (EQU) 6666H.

MSG1 DB "KEY IS FOUND AT" this line is a declaration of Character Array initialized with "KEY IS FOUND AT". (A Character is of a BYTE Hence we have to use only DB Define Byte) and Similarly to **RES** and **MSG2**. Detailed explanation is given below.

DATA ENDS

DATA ENDS is the End point of the Data Segment in a Program. We can write just ENDS But to differentiate the end of which segment it is of which we have to write the same name given to the Data Segment.

<u>CODE SEGMENT</u>

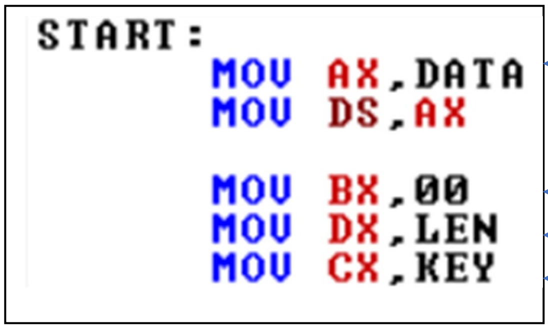
CODE SEGMENT is the starting point of the Code Segment in a Program and CODE is the name given to this segment and SEGMENT is the keyword for defining Segments, where we can write the coding of the program.

<u>ASSUME DS:DATA CS: CODE</u>

In this Assembly Language Programming, there are Different Registers present for Different Purpose So we have to assume DATA is the name given to Data Segment register and CODE is the name given to Code Segment register (SS, ES are used in the same way as CS, DS).

<u>START:</u>

START is the label used to show the starting point of the code which is written in the Code Segment. This is a LABEL and all the words ending in colon (:).



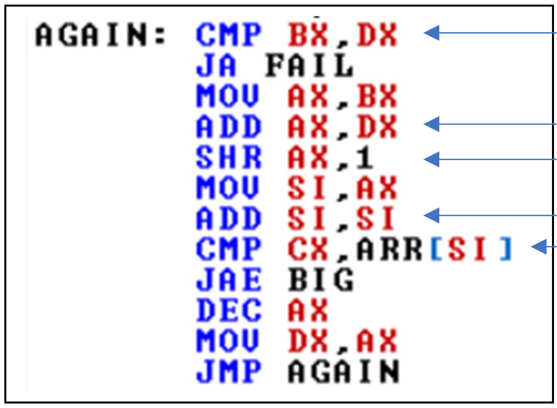
```
START:
        MOV  AX,DATA
        MOV  DS,AX

        MOV  BX,00
        MOV  DX,LEN
        MOV  CX,KEY
```

But we cannot move DATA Directly to DS due to MOV commands restriction. Hence, we move DATA to AX and then from AX to DS.

LOW PTR

HIGH PTR

KEY PTR



```
AGAIN: CMP BX,DX
       JA  FAIL
       MOV AX,BX
       ADD AX,DX
       SHR AX,1
       MOV SI,AX
       ADD SI,SI
       CMP CX,ARR[SI]
       JAE BIG
       DEC AX
       MOV DX,AX
       JMP AGAIN
```

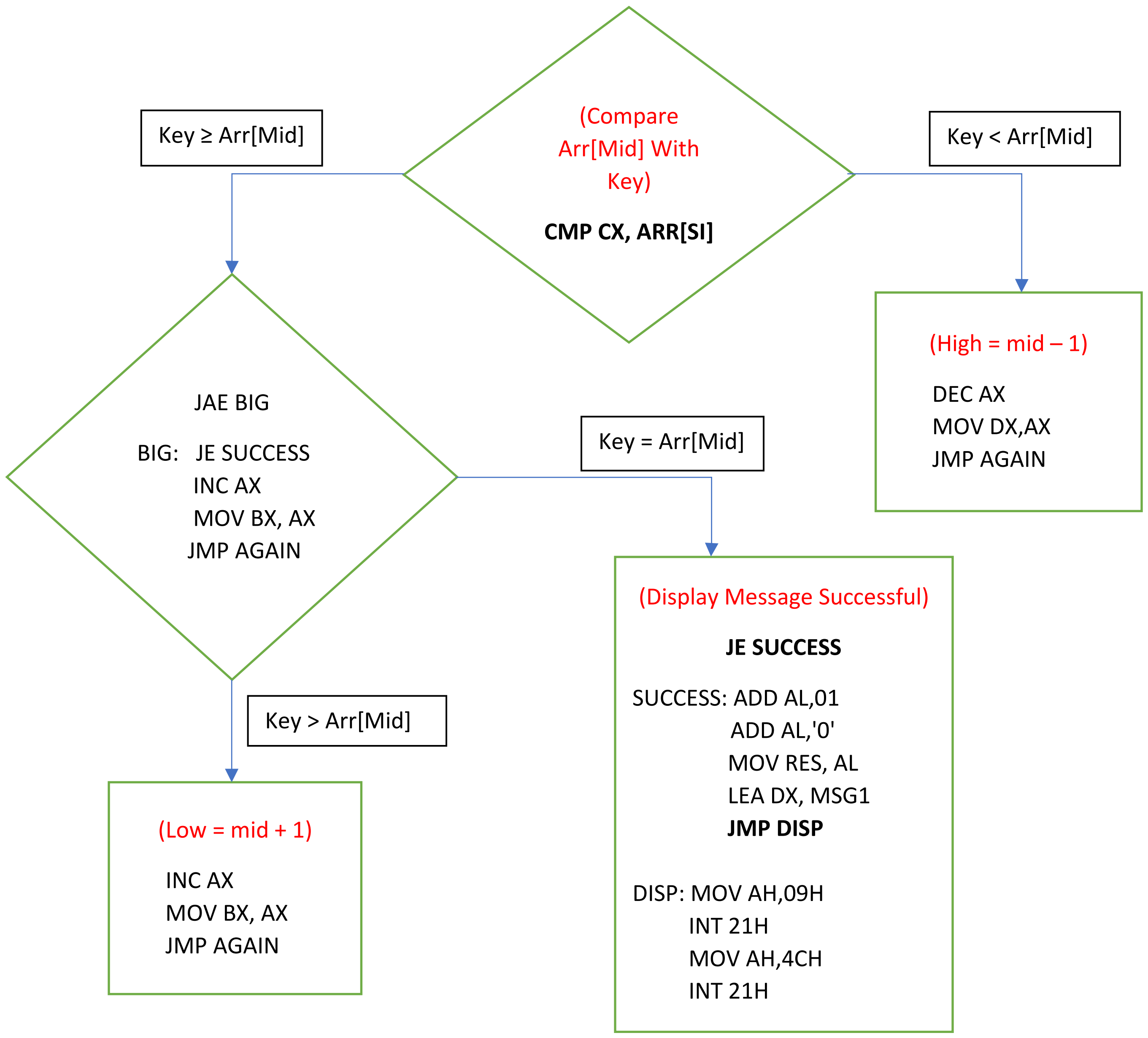
CMP BX, DX is used to compare Element of BX register with DX register and JA FAIL Short Jump if first operand (i.e. BX) is Above second operand (i.e. DX) to the respective LABEL FAIL.

MID (LOW + HIGH)

MID (LOW + HIGH)/2

Double the value. as each number is 2 bytes

Compare the key value with the mid value's position in array



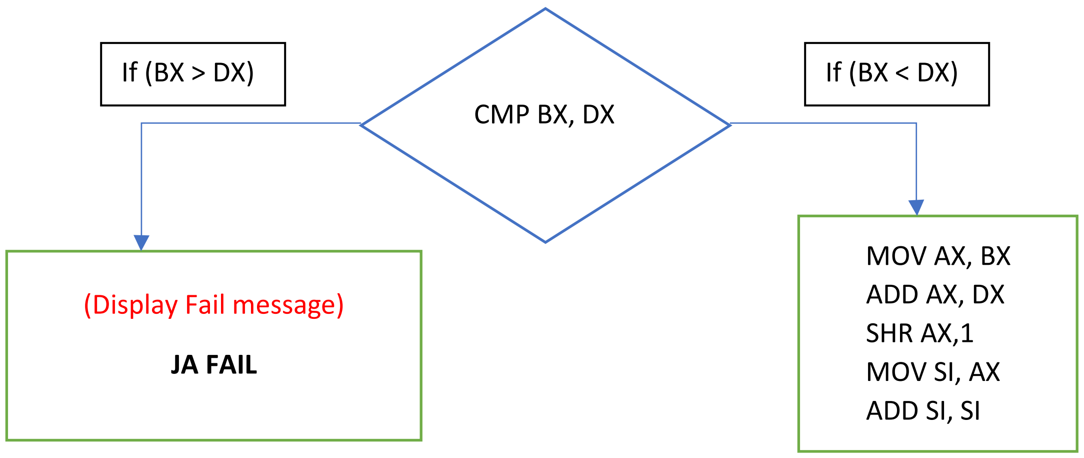
```
CMP CX, ARR[SI]
```

```
JAE BIG

BIG:   JE SUCCESS
       INC AX
       MOV BX, AX
      JMP AGAIN
```

```
DEC AX
MOV DX,AX
JMP AGAIN
```

```
INC AX
MOV BX, AX
JMP AGAIN
```

```
JE SUCCESS

SUCCESS: ADD AL,01
         ADD AL,'0'
         MOV RES, AL
         LEA DX, MSG1
         JMP DISP

DISP: MOV AH,09H
      INT 21H
      MOV AH,4CH
      INT 21H
```



```
CMP BX, DX
```

```
JA FAIL
```

```
MOV AX, BX
ADD AX, DX
SHR AX,1
MOV SI, AX
ADD SI, SI
```

CODE ENDS

CODE ENDS is the End point of the Code Segment in a Program. We can write just ENDS But to differentiate the end of which segment it is of which we have to write the same name given to the Code Segment.

END START

END START is the end of the label used to show the ending point of the code which is written in the Code Segment.

Debugging-Test-run:

We all have heard about binary search and knows how it can be used for quickly finding something in a sorted array. It's no surprise given how easy it is to grasp and how amazingly efficient it is. Just think that finding something in an array of 1000 elements only takes 8 checks. That's magic!

But I keep getting surprised how many of us only think of binary search when facing exactly that problem (finding something in an array) and nowhere else. It's such a useful tool I thought I'd mention some of the other uses of it I make.

I have run this code about 5/6 time with different sorted values and keys where sometimes the key was present in the list and sometimes it wasn't. But every time this algorithm gave me the correct search result and correct position.

Results analysis: (The Analysis part should discuss other aspects, like complexity of algorithms in terms of average and worst-case complexity for time and space, robustness of the approach used, finer technical details, etc.)

Time complexity

Binary search runs in logarithmic time in the **worst case** and **average case**, making **O (log n)** comparisons and **Best-case** complexity is O (1), where n is the number of elements in the array, the O is Big O notation, and **log** is the logarithm.

Space complexity

Binary search requires three pointers to elements, which may be array indices or pointers to memory locations, regardless of the size of the array. Therefore, the space complexity of binary search is **O (1)** in the Word RAM model of computation.

Binary search is an efficient way to find an item in a sorted list. For instance, if you are looking for a specific page in a book (say, pp. 147) you'd open the book near the middle and determine if the opened page is before or after the page you are looking for. Then you'd pick the section you've narrowed it down to and repeat the process: split it in half and determine which half contains page 147. Even better, you can guess how far in page 147 is—not far if the book is very long and near the end of a short book—and use that guess as the first division point. This variation on binary search is called **interpolation search**. Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.

Conclusion and Future Improvements:

Binary searching is an interesting, useful, and beautiful algorithm. Its code is simple enough to be motivated, discussed and analysed in a beginning programming course.

Though it has some slight lacking's when this procedure causes an execution error whenever Size = 0. In the procedure above, with Size = 0, the first evaluation of the expression (Low + High) DIV 2 will result in the value 0, which causes a **subrange-out-of-bounds execution error**. But it is often the first algorithm to be presented with a nontrivial runtime analysis - using **big-O** notation. Although it contains code that is only slightly more complex than linear searching, binary searching offers a tremendous speed improvement for searching large arrays. And because searching is such a frequently occurring operation in software, binary search procedures are often used as subroutines in large systems.

Bibliography:

1. https://stackoverflow.com/
2. https://en.wikipedia.org/
3. https://www.youtube.com/
4. http://cssimplified.com/
5. www.geeksforgeeks.org
6. www.tutorialspoint.com
7. www.gatevidyalay.com