# *Searching and Sorting Algoritthms*

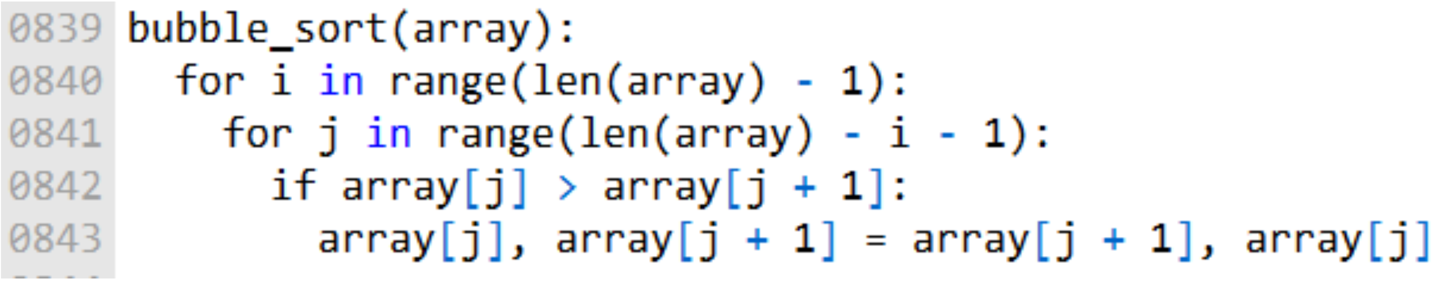
## Bubble sort

The bubble sort algorithm is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. The algorithm starts at the beginning of the array and compares the first two elements.

If the first element is greater than the second element, the two elements are swapped. The algorithm then moves on to the next two elements and repeats the process.

The algorithm continues to iterate through the array until it reaches the end.

The following is a pseudocode implementation of the bubble sort algorithm:



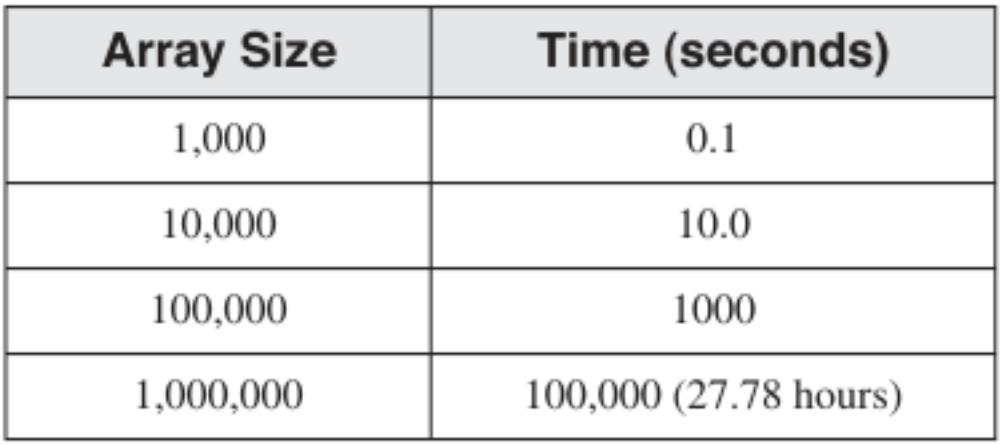
```
bubble_sort(array):
  for i in range(len(array) - 1):
    for j in range(len(array) - i - 1):
      if array[j] > array[j + 1]:
        array[j], array[j + 1] = array[j + 1], array[j]
```

The bubble sort algorithm is a simple and straightforward algorithm, but it is not very efficient for large arrays.

This is because the algorithm has to compare every pair of elements in the array for each iteration. For an array of size n, the bubble sort algorithm has a **time complexity of O(n$^2$).**

### *Analysis of bubble sort performance.*

The following table shows the sort times for various array sizes, assuming that 1000 array elements can be sorted in 0.1 second:



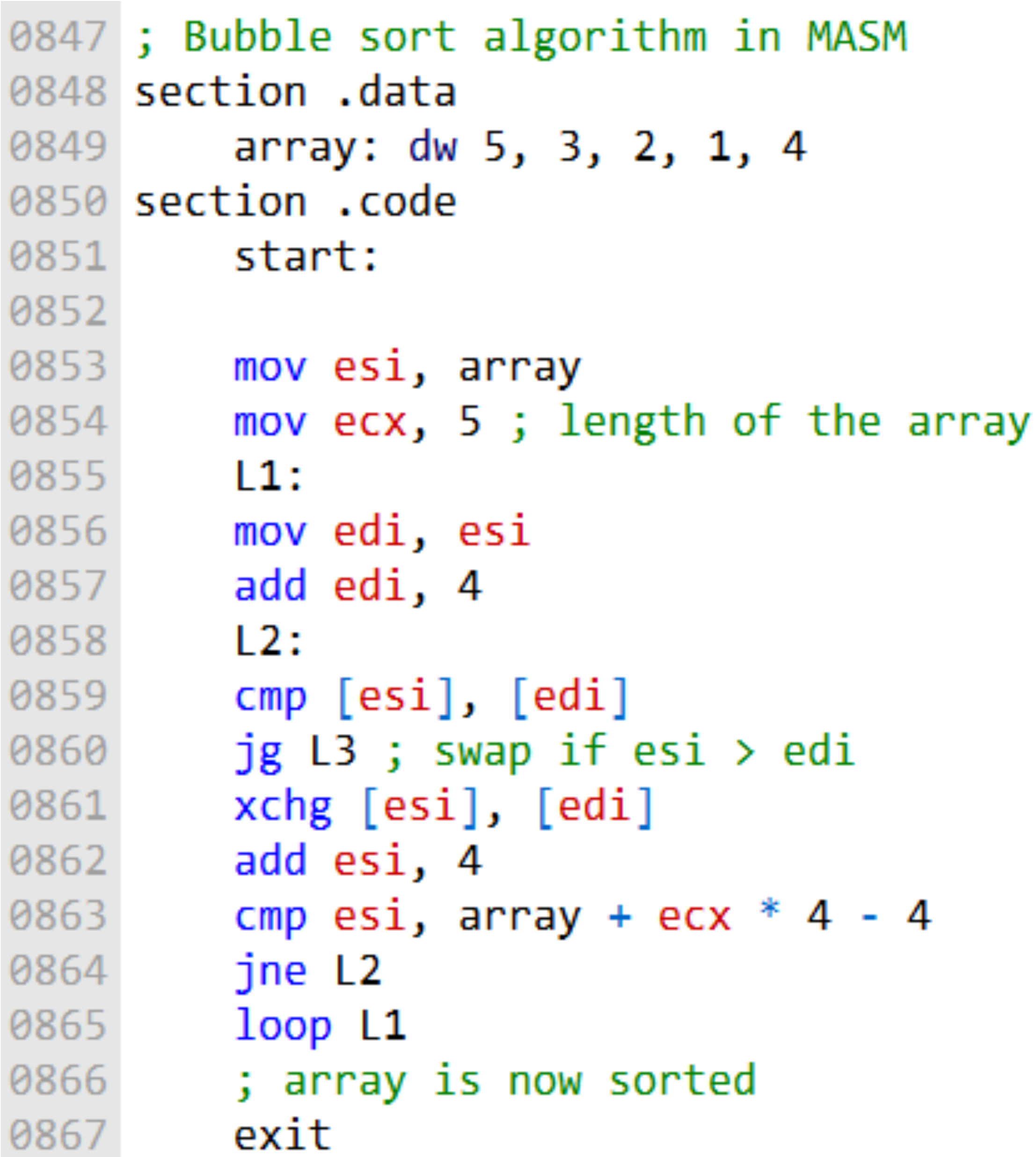
| Array Size | Time (seconds) |
|---|---|
| 1,000 | 0.1 |
| 10,000 | 10.0 |
| 100,000 | 1000 |
| 1,000,000 | 100,000 (27.78 hours) |

As you can see, the sort time increases quadratically with the array size. This means that the bubble sort algorithm is not very efficient for large arrays.

The bubble sort algorithm is a simple and straightforward sorting algorithm, but it is not very efficient for large arrays.

If you need to sort a large array, you should use a more efficient sorting algorithm, such as the quicksort algorithm or the merge sort algorithm.

----------------------------------------



```
; Bubble sort algorithm in MASM
section .data
    array: dw 5, 3, 2, 1, 4
section .code
    start:

    mov esi, array
    mov ecx, 5 ; length of the array
    L1:
    mov edi, esi
    add edi, 4
    L2:
    cmp [esi], [edi]
    jg L3 ; swap if esi > edi
    xchg [esi], [edi]
    add esi, 4
    cmp esi, array + ecx * 4 - 4
    jne L2
    loop L1
    ; array is now sorted
    exit
```

The array section in the data segment initializes an array with values to be sorted.

The code begins by setting up registers, with esi pointing to the start of the array and ecx containing the length of the array (in this case, 5).

The outer loop, labeled as L1, iterates through the array. This corresponds to the outer loop counter (cx1) in the notes.

Inside the outer loop, the inner loop labeled as L2 is used to compare and swap elements, corresponding to the inner loop counter (cx2) in the notes.

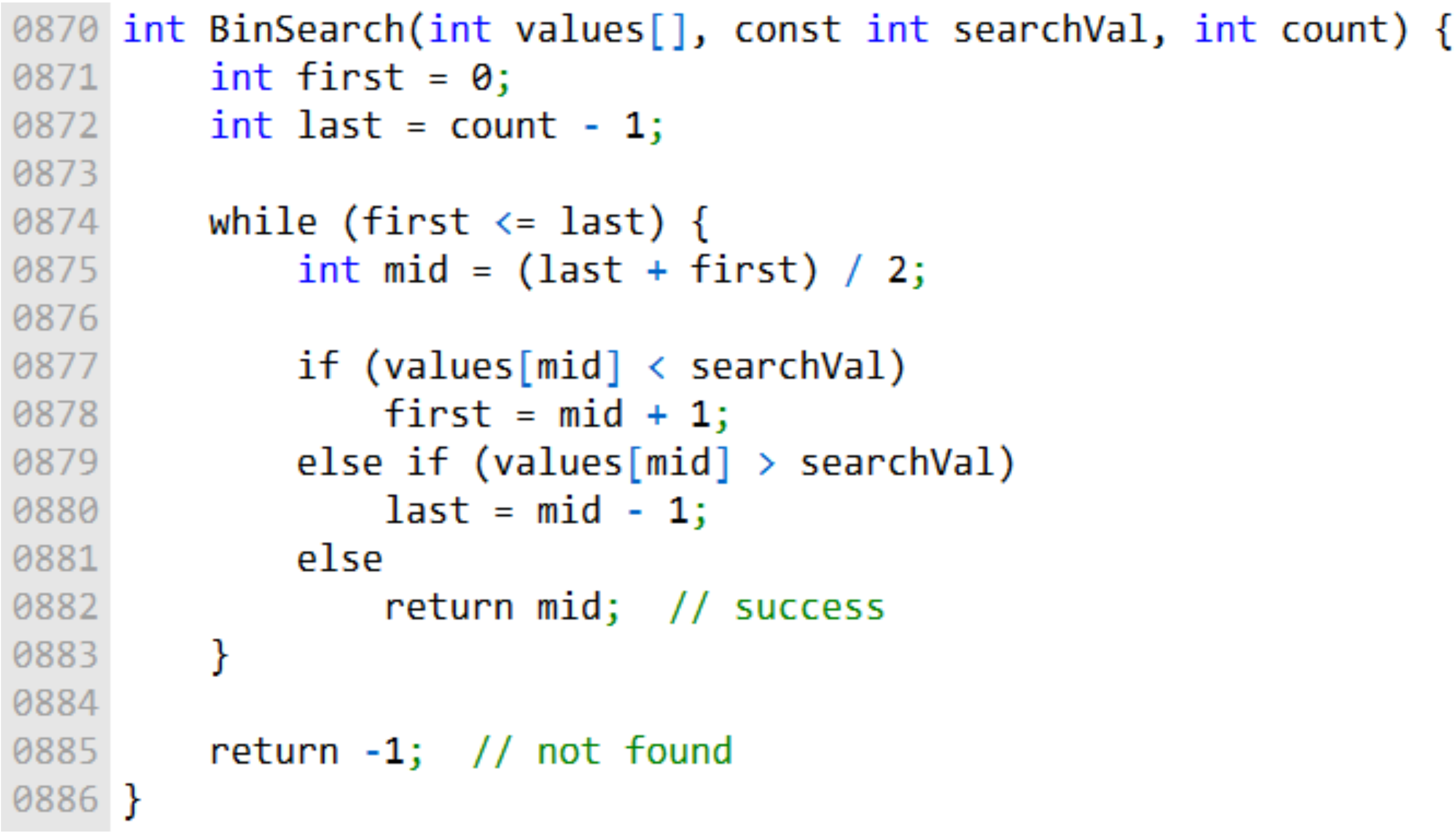
The comparison is done using cmp, and if the current element ([esi]) is greater than the next element ([edi]), a swap is performed using xchg.

The code ensures that the inner loop (L2) iterates through the entire array by comparing esi to the end of the array (array + ecx * 4 - 4).

After completing the inner loop for a given pass through the array, it uses loop to decrement the outer loop counter and repeats the process until the outer loop counter is equal to 0.

Once the sorting is complete, the array is in ascending order.

C++ version:



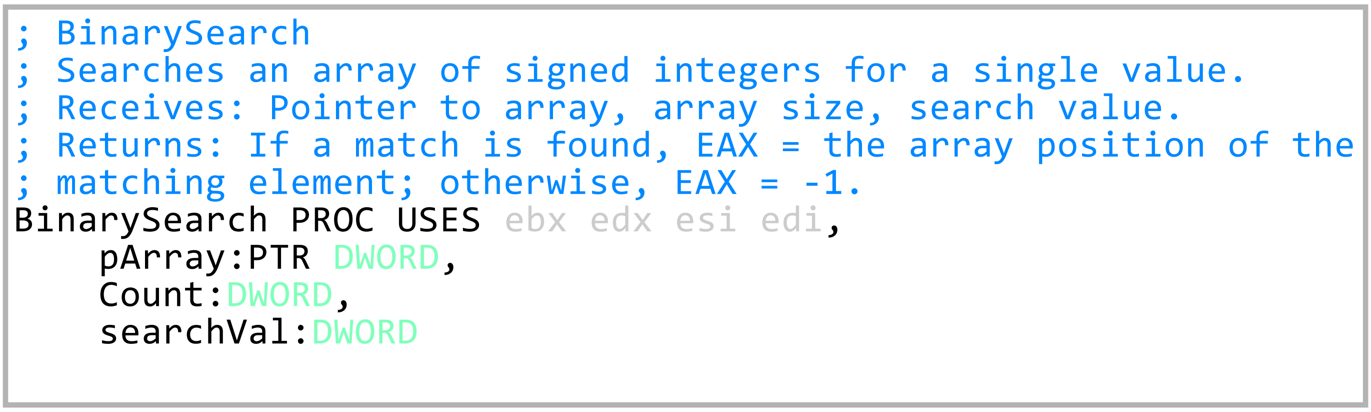
```
int BinSearch(int values[], const int searchVal, int count) {
    int first = 0;
    int last = count - 1;

    while (first <= last) {
        int mid = (last + first) / 2;

        if (values[mid] < searchVal)
            first = mid + 1;
        else if (values[mid] > searchVal)
            last = mid - 1;
        else
            return mid;   // success
    }

    return -1;   // not found
}
```

Assembly version:



```
; BinarySearch
; Searches an array of signed integers for a single value.
; Receives: Pointer to array, array size, search value.
; Returns: If a match is found, EAX = the array position of the
; matching element; otherwise, EAX = -1.
BinarySearch PROC USES ebx edx esi edi,
    pArray:PTR DWORD,
    Count:DWORD,
    searchVal:DWORD
    LOCAL first:DWORD,
        last:DWORD,
        mid:DWORD

    mov first, 0
    mov eax, Count
    dec eax
    mov last, eax
    mov edi, searchVal
    mov ebx, pArray

L1:
    mov eax, first
    cmp eax, last
    jg L5

    mov eax, last
    add eax, first
    shr eax, 1
    mov mid, eax

    mov esi, mid
    shl esi, 2
    mov edx, [ebx+esi]

    cmp edx, edi
    jge L2
    mov eax, mid
    inc eax
    mov first, eax
    jmp L4

L2:
    cmp edx, edi
    jle L3
    mov eax, mid
    dec eax
    mov last, eax
    jmp L4

L3:
    mov eax, mid
    jmp L9

L4:
    jmp L1

L5:
    mov eax, -1

L9:
    ret

BinarySearch ENDP
```

Program 2:

```
; Bubble Sort and Binary Search (BinarySearchTest.asm)
; Bubble sort an array of signed integers and perform a binary search.

INCLUDE Irvine32.inc
INCLUDE BinarySearch.inc  ; Include procedure prototypes

LOWVAL = -5000
HIGHVAL = +5000
ARRAY_SIZE = 50

.data
array DWORD ARRAY_SIZE DUP(?)

.code
main PROC
    call Randomize  ; Initialize random number generator

    ; Fill an array with random signed integers
    INVOKE FillArray, ADDR array, ARRAY_SIZE, LOWVAL, HIGHVAL

    ; Display the array
    INVOKE PrintArray, ADDR array, ARRAY_SIZE
    call WaitMsg

    ; Perform a bubble sort and redisplay the array
    INVOKE BubbleSort, ADDR array, ARRAY_SIZE
    INVOKE PrintArray, ADDR array, ARRAY_SIZE

    ; Demonstrate a binary search
    call AskForSearchVal

    ; Perform the binary search and display the results
    INVOKE BinarySearch, ADDR array, ARRAY_SIZE, eax
    call ShowResults

    exit
main ENDP

; Prompt the user for a signed integer
AskForSearchVal PROC
    .data
    prompt BYTE "Enter a signed decimal integer in the range of -5000 to +5000 to find in the array: ",0
    .code
    call Crlf
    mov edx, OFFSET prompt
    call WriteString
    call ReadInt
    ret
AskForSearchVal ENDP

; Display the resulting value from the binary search
ShowResults PROC
    .data
    msg1 BYTE "The value was not found.",0
    msg2 BYTE "The value was found at position ",0
    .code
    .IF eax == -1
        mov edx, OFFSET msg1
        call WriteString
    .ELSE
        mov edx, OFFSET msg2
        call WriteString
        call WriteDec
    .ENDIF
    call Crlf
    call Crlf
    ret
ShowResults ENDP

END main
```

The provided assembly code, BinarySearchTest.asm, is a program that demonstrates the use of the bubble sort and binary search functions to work with an array of signed integers. Here's an overview of what the code does:

**Initialization:**

It starts by including necessary libraries and defining constants for the minimum and maximum values (LOWVAL and HIGHVAL) and the size of the array (ARRAY_SIZE).

The code also defines the array array and contains the .data and .code sections.

**Main Procedure (main):**

Calls the Randomize function to initialize the random number generator.

Invokes the FillArray procedure to fill the array with random signed integers within the specified range (LOWVAL to HIGHVAL).

Displays the original content of the array using the PrintArray procedure and waits for a message (WaitMsg).

Performs a bubble sort on the array using the BubbleSort procedure to sort the integers in ascending order.

Displays the sorted array using the PrintArray procedure. Prompts the user to enter a signed integer with the AskForSearchVal procedure.

**AskForSearchVal Procedure:**

Prompts the user to enter a signed integer within the specified range. Reads the integer entered by the user and returns it in the EAX register.

**ShowResults Procedure:**

Displays the result of the binary search.

If the binary search returns -1, indicating that the value was not found, it prints "The value was not found."

If the binary search returns the position of the value in the array, it prints "The value was found at position X," where X is the position.

In summary, this program generates a random array of signed integers, sorts the array using the bubble sort algorithm, and then performs a binary search on the sorted array to find a user-specified value.

It displays the results of the binary search, indicating whether the value was found and, if so, at what position in the array.