Short-term Hands-on Supplementary Course on C Programming

SESSION 5: Array Operations

KARTHIK D
NIVEDHITHA D

Time: 6:30 - 8:00 PM
Date: 30 May 2022
Location: Online

Agenda

1. Administrative Instructions
2. Time Complexity
3. Searching:
   a. Linear Search
   b. Binary Search
4. Sorting:
   a. Bubble Sort
   b. Selection
   c. Insertion Sort
   d. Merge Sort
5. Tutorial: Insertion Sort
6. Next Session

Administrative Instructions

- Please fill out the feedback form - will be shared in the chat
- Join us on Microsoft Teams,

  Team Code: **rzlaicv**

GITHUB REPOSITORY! ⭐🌟

Computational Complexity

How does one go about analyzing programs to compare how the program behaves as it scales? E.g., let's look at a **vectorMax()** function:

```
int vectorMax( int v[]                ){
    int currentMax = v[0];
    int n =  sizeof(v)/sizeof(v[0]);
    for (int i=1; i < n; i++){
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

How will estimate time efficiency?

If we want to see how this algorithm behaves as n changes, we could do the following:

1. Code the algorithm in C
2. Determine, for each instruction of the compiled program the time needed to execute that instruction (need assembly language)
3. Determine the number of times each instruction is executed when the program is run.
4. Sum up all the times we calculated to get a running time.

…might work, but it is complicated, especially for today's machines that optimize everything "under the hood." (and reading assembly code takes a certain patience)

```
0x000000010014adf0 <+0>:  push   %rbp
0x000000010014adf1 <+1>:  mov    %rsp,%rbp
0x000000010014adf4 <+4>:  sub    $0x20,%rsp
0x000000010014adf8 <+8>:  xor    %esi,%esi
0x000000010014adfa <+10>: mov    %rdi,-0x8(%rbp)
0x000000010014adfe <+14>: mov    -0x8(%rbp),%rdi
0x000000010014ae02 <+18>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae07 <+23>: mov    (%rax),%esi
0x000000010014ae09 <+25>: mov    %esi,-0xc(%rbp)
0x000000010014ae0c <+28>: mov    -0x8(%rbp),%rdi
0x000000010014ae10 <+32>: callq  0x10014afb0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+304>
0x000000010014ae15 <+37>: mov    %eax,-0x10(%rbp)
0x000000010014ae18 <+40>: movl   $0x1,-0x14(%rbp)
0x000000010014ae1f <+47>: mov    -0x14(%rbp),%eax
0x000000010014ae22 <+50>: cmp    -0x10(%rbp),%eax
0x000000010014ae25 <+53>: jge    0x10014ae6c <vectorMax(Vector<int>&)+124>
0x000000010014ae2b <+59>: mov    -0xc(%rbp),%eax
0x000000010014ae2e <+62>: mov    -0x8(%rbp),%rdi
0x000000010014ae32 <+66>: mov    -0x14(%rbp),%esi
0x000000010014ae35 <+69>: mov    %eax,-0x18(%rbp)
0x000000010014ae38 <+72>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae3d <+77>: mov    -0x18(%rbp),%esi
0x000000010014ae40 <+80>: cmp    (%rax),%esi
0x000000010014ae42 <+82>: jge    0x10014ae59 <vectorMax(Vector<int>&)+105>
0x000000010014ae48 <+88>: mov    -0x8(%rbp),%rdi
0x000000010014ae4c <+92>: mov    -0x14(%rbp),%esi
0x000000010014ae4f <+95>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae54 <+100>:   mov    (%rax),%esi
0x000000010014ae56 <+102>:   mov    %esi,-0xc(%rbp)
0x000000010014ae59 <+105>:   jmpq   0x10014ae5e <vectorMax(Vector<int>&)+110>
0x000000010014ae5e <+110>:   mov    -0x14(%rbp),%eax
0x000000010014ae61 <+113>:   add    $0x1,%eax
0x000000010014ae64 <+116>:   mov    %eax,-0x14(%rbp)
0x000000010014ae67 <+119>:   jmpq   0x10014ae1f <vectorMax(Vector<int>&)+47>
0x000000010014ae6c <+124>:   mov    -0xc(%rbp),%eax
0x000000010014ae6f <+127>:   add    $0x20,%rsp
0x000000010014ae73 <+131>:   pop    %rbp
0x000000010014ae74 <+132>:   retq
```

Algorithm Analysis: Primitive Operations

Instead of those complex steps, we can define *primitive operations* for our C code.

- Assigning a value to a variable
- Calling a function
- Arithmetic (e.g., adding two numbers)
- Comparing two numbers
- Indexing into a Vector
- Returning from a function

We assign "1 operation" to each step. We are trying to gather data so we can compare this to other algorithms.

```
int vectorMax( int v[] ){
    int currentMax = v[0];
    int n =  sizeof(v)/sizeof(v[0]);
    for (int i=1; i < n; i++){

        if (currentMax < v[i]) {

            currentMax = v[i];
        }

    }
    return currentMax;
}
```

executed once (2 ops)

executed once (2 ops)

executed once (1 op)

ex. n times (*n* ops)

executed *n-1* times (2*(*n*-1) ops))

ex. n-1 times (2*(n-1) ops)

ex. at most n-1 times (2*(n-1) ops), but as few as zero times

ex. once (1 op)

Summary:

Primitive operations for **`vectorMax()`**:

at least: $2 + 2 + 1 + n + 4 * (n - 1) + 1 = 5n + 2$

at most: $2 + 2 + 1 + n + 6 * (n - 1) + 1 = 7n$

i.e., if there are *n* items in the Vector, there are between 5*n*+2 operations and 7*n* operations completed in the function.

Do we *really* need this much detail? Nope!

Let's simplify: we want a "big picture" approach.

It is enough to know that `vectorMax()` grows

linearly proportionally to n

In other words, as the number of elements increases, the algorithm has to do proportionally more work, and that relationship is linear. 8x more elements? 8x more work.

Algorithm Analysis: Big-O

Dirty little trick for figuring out Big-O: look at the number of steps you calculated, throw out all the constants, find the "biggest factor" and that's your answer:

$$5n + 2 \text{ is } O(n)$$

Why? Because constants are not important at this level of understanding.

Algorithm Analysis: Big-O

We will care about the following functions that appear often in data structures:

constant	logarithmic	linear	n log n	quadratic	polynomial (other than n^2)	exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a>1)$

When you are deciding what Big-O is for an algorithm or function, simplify until you reach one of these functions, and you will have your answer.

Algorithm Analysis: Big-O

constant	logarithmic	linear	n log n	quadratic	polynomial (other than n^2)	exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a>1)$

Practice: what is Big-O for this function?

$$20n^3 + 10n \log n + 5$$

Answer: $O(n^3)$

First, strip the constants: $n^3 + n \log n$

Then, find the biggest factor: n^3

Rule of Sums

Sequential Segments

<

. . . Program Segment 1 $T_1(n)$

. . .

>

<

. . . Program Segment 2 $T_2(n)$

. . .

>

$T(n) = T_1(n) + T_2(n)$

Sum Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O(max(f(n), g(n)))$

Rule of Products

Iteration

$<$

. . . Program Segment

. . .

$>$

executed $T_2(n)$ times

$T_1(n)$

$$T(n) = T_1(n) \times T_2(n)$$

Product Rule

Suppose $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T(n)$ is $O(f(n)g(n))$

Algorithm Analysis: Nested Loops

```
int nestedLoop1(int n){
        int result = 0;
        for (int i=0;i<n;i++){
                for (int j=0;j<n;j++){
                        result++;
                }
        }
        return result;
}
```

Also go through the outer loop n times

Inner loop complexity: *O(n)*

Total complexity: *$O(n^2)$ (quadratic)*

In general, we don't like $O(n^2)$ behavior! Why?

As an example: let's say an $O(n^2)$ function takes 5 seconds for a container with 100 elements. How much time would it take if we had 1000 elements?

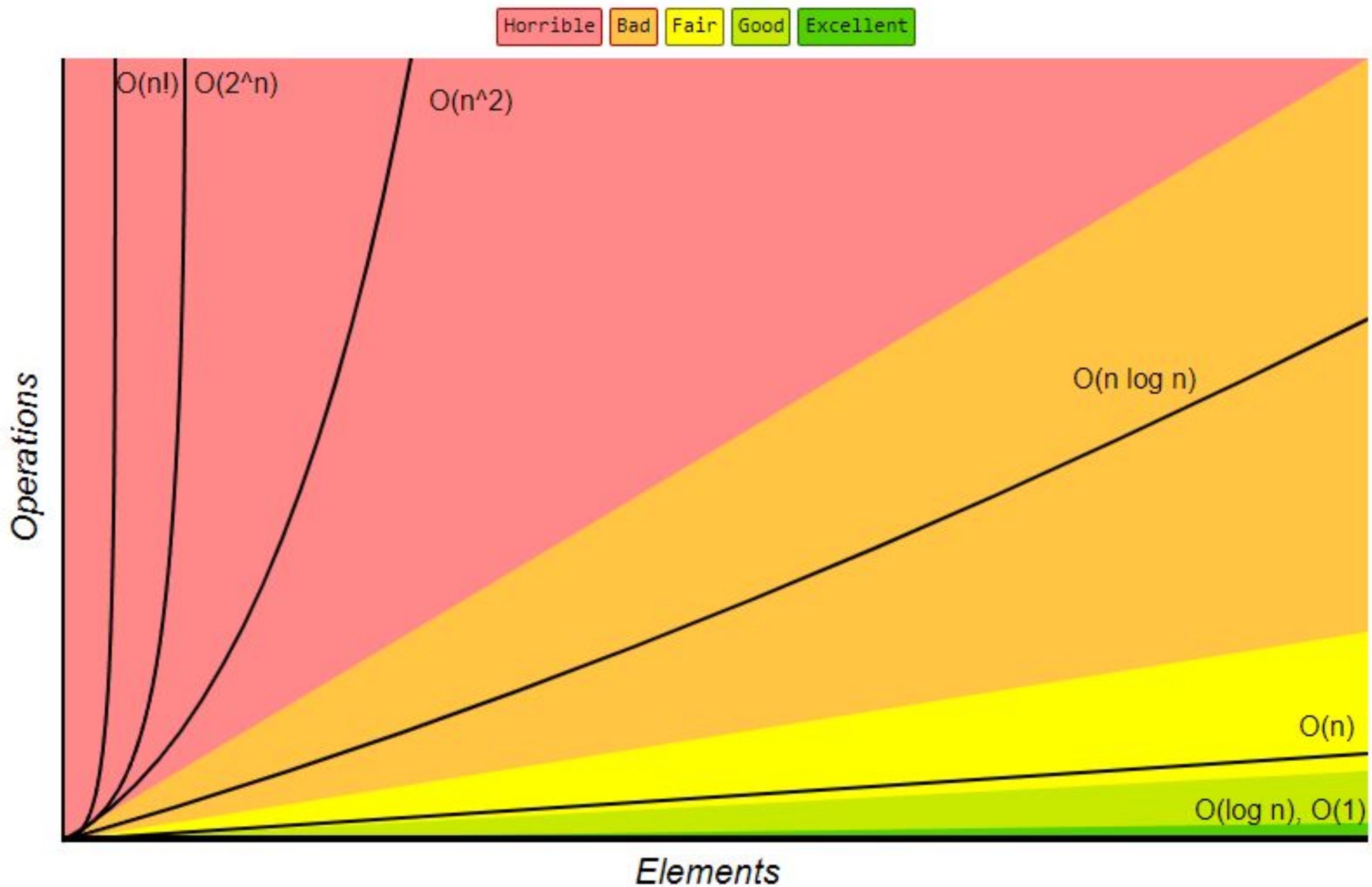
500 seconds! This is because 10x more elements is (10^2)x more time!

Algorithm Analysis: Nested Loops

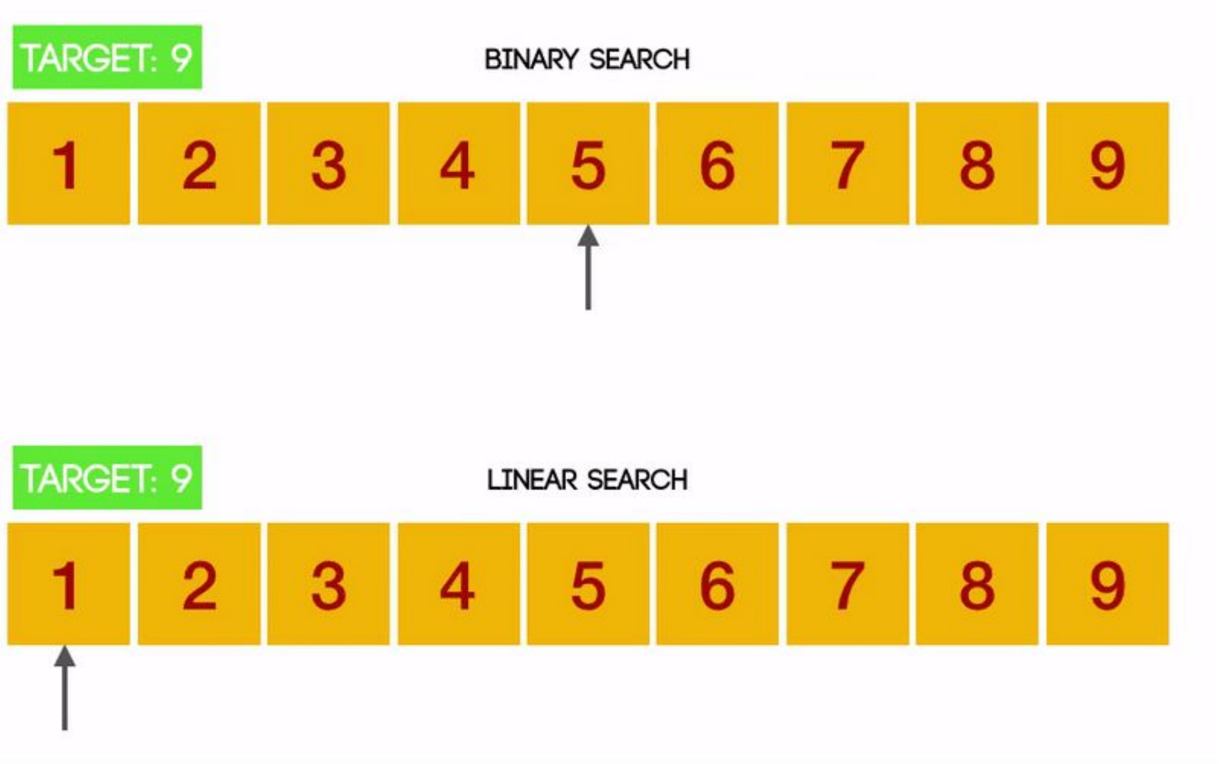
```
int nestedLoop1(int n){
        int result = 0;
        for (int i=0;i<n;i++){
                for (int j=0;j<n;j++){
                        for (int k=0;k<n;k++)
                                result++;
                }
        }
        return result;
}
```

What would the complexity be of a 3-nested loop?

Big-O Complexity Chart



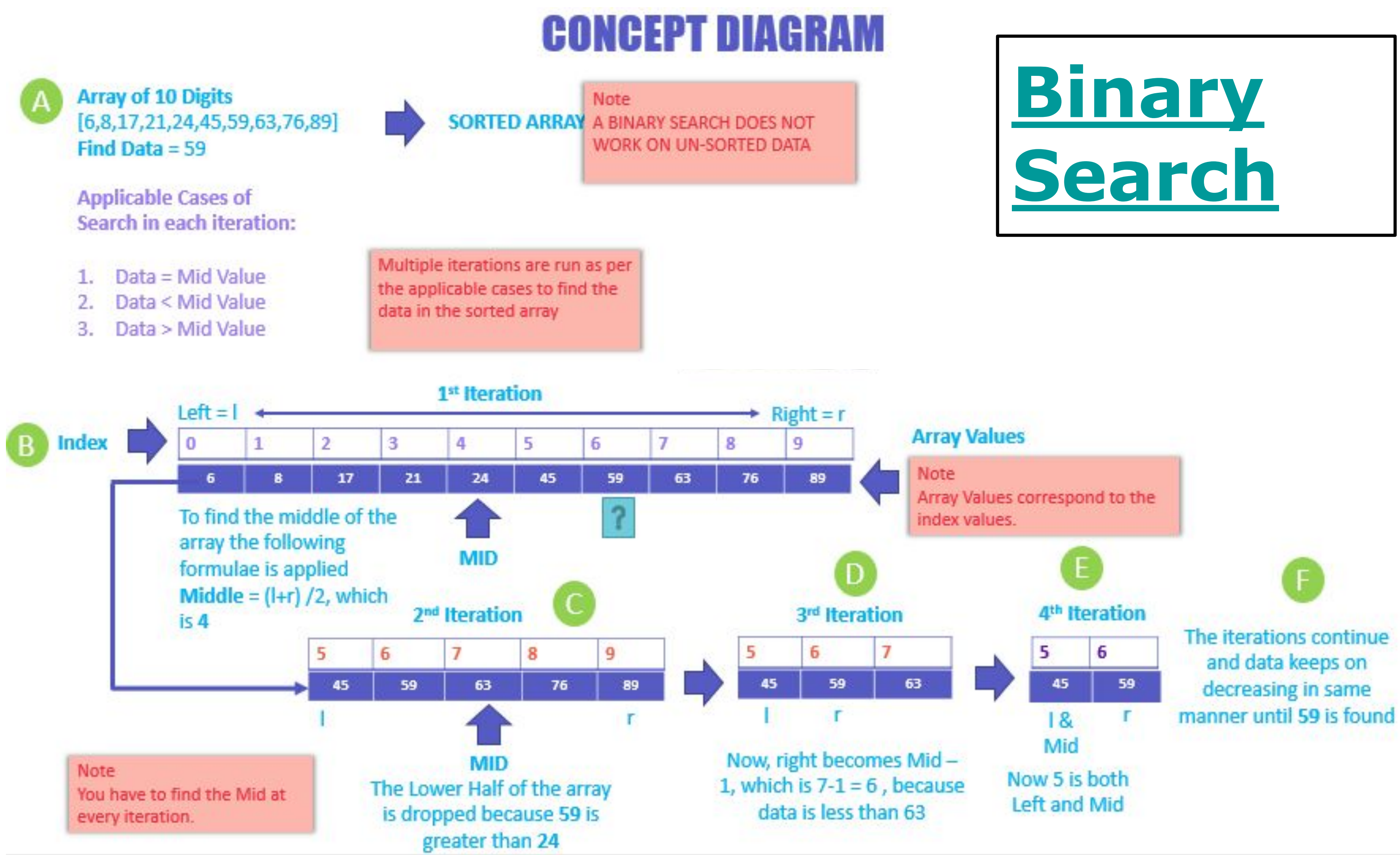
Searching



Searching for an element in a 1D array!!!

KARTHIK!!!!!!!

Binary Search



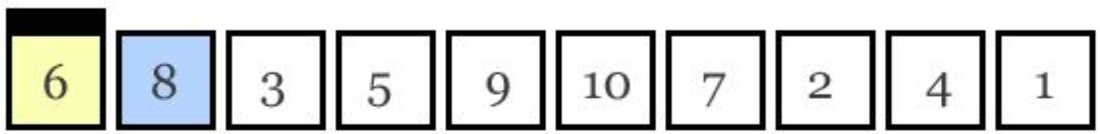
Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Bubble Sort

6 5 3 1 8 7 2 4

Selection Sort



Yellow is smallest number found
Blue is current item
Green is sorted list

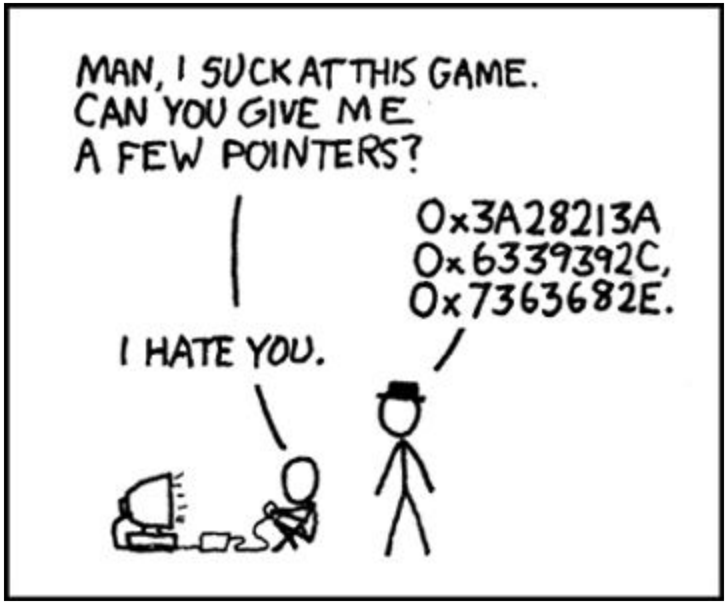
Insertion Sort

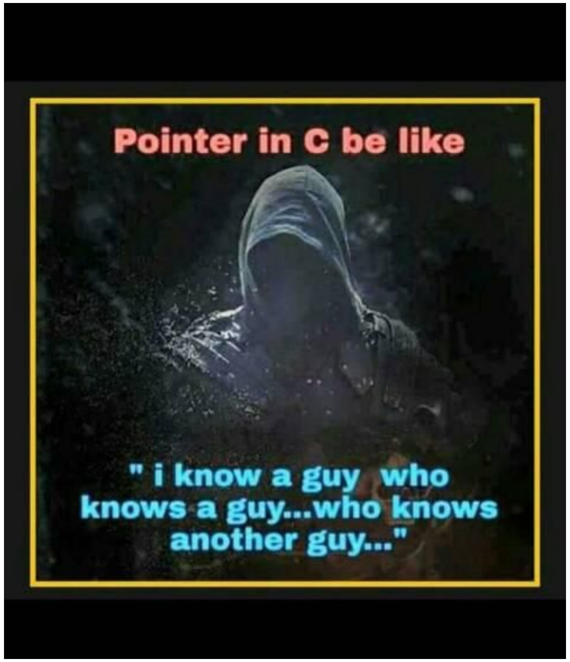
Merge Sort: Tutorial

6 5 3 1 8 7 2 4

Next Session

POINTERS!!!





Any Questions