DSC 20
Discussion Section 5

ARDA CANKAT BATI

Today's Plan

1. Time Complexity
2. Space Complexity (very brief explanation)
3. Brute Force vs Efficient Solution
4. Time Complexity Examples

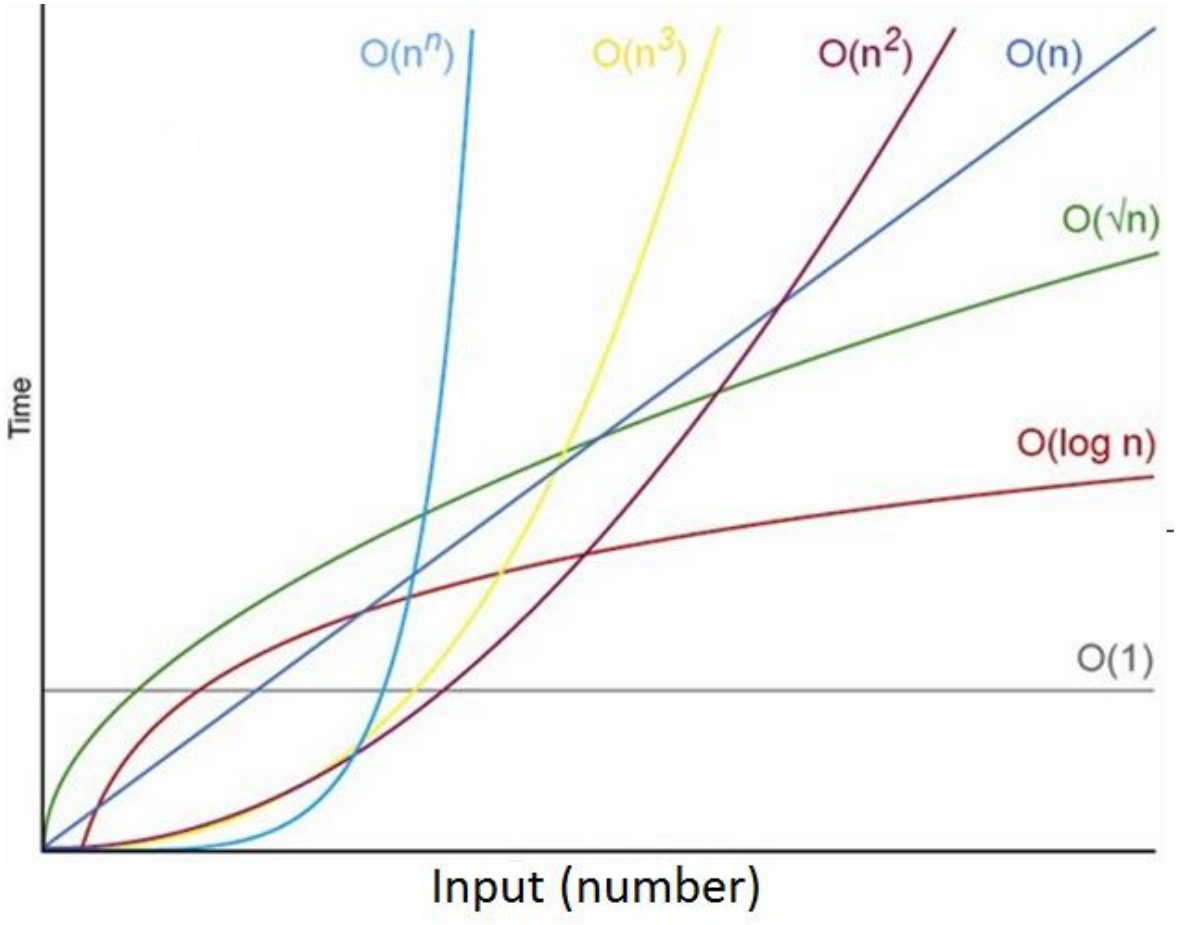
Time Complexity

Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.

In other words, time complexity is essentially efficiency, or how long a program function takes to process a given input.

https://www.techopedia.com/definition/22573/time-complexity

Time Complexity



Notice how the difference between different orders are not very clear for low n.

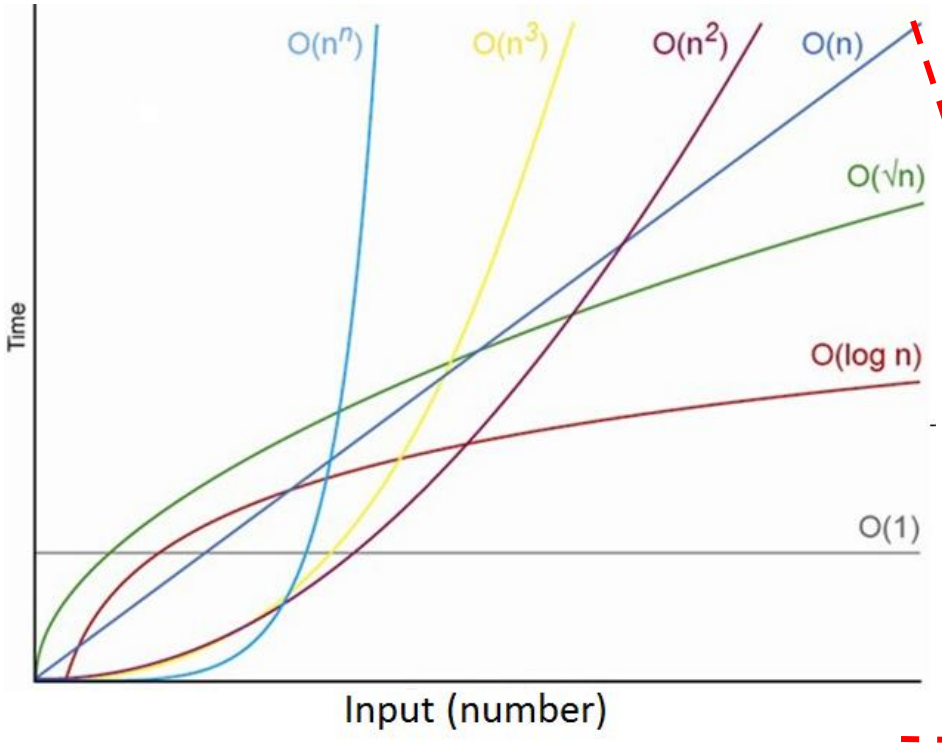
As n gets higher we start seeing the real difference between each order.

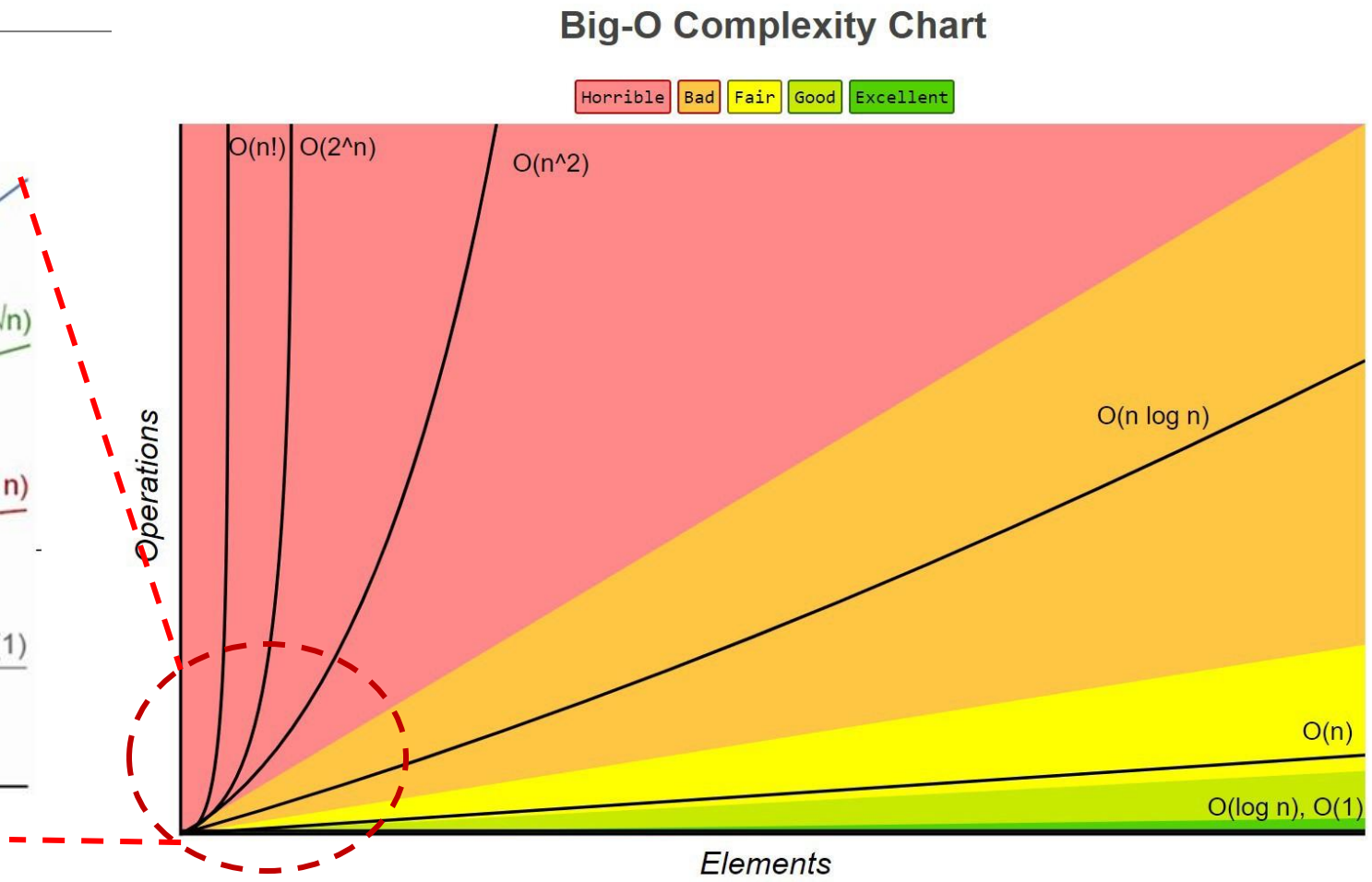
Algorithms with exponentials n^n, a^n are almost never feasible to use.

O(1) is the best, but it needs very specific structures and problem types to have such algorithms.

Usually something with log(n) will be more commonplace.

Time Complexity





Time Complexity

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)

This is too much detail for now, but you can check out the website:

https://www.bigocheatsheet.com/

Which is very helpful for quick reference of algorithm / data structure complexity.

Space Complexity

Space complexity is a measure of the amount of the working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point in the algorithm.

As with time complexity, we're mostly concerned with how the space needs grow, in big-Oh terms, as the size N of the input problem grows.

https://courses.cs.northwestern.edu/311/html/space-complexity.html

Time & Space Complexity

- It is best to have both low Time & Space complexity.
- Sometimes due to the structure of the problem, you will have to sacrifice one to get better in the other.
- **Case 1:** Consider cluster computing such as Apache Spark which uses the main memory (RAM) of hundreds of computers to work on big data. Which complexity is it trying to reduce? (relatively to the other one)
- **Case 2:** Consider a small embedded device (for example smart watch), that is recording and displaying your health data. Which complexity is it focusing on to reduce? (relative to the other one)

Time & Space Complexity

Answers here are not definite. But for the most part, we can consider the following:

- If you have a lot of memory, but very limited time you may be willing to increase space complexity to make your algorithm faster. Apache Spark case is similar to this.
- You may have a device with low memory, doing non time critical tasks. You are worried about memory size so you might be willing to lose a bit more time, so that you can use memory more efficiently. The smart watch case is similar to this.

Brute Force vs Efficient Solution

- For almost every programming task that is solvable, you can write a brute force algorithm to solve it. This alogrithm will get the job done, but it will usually be very inefficient by time complexity and/or space complexity.
- After the brute force solution, you can start considering other, more efficient solutions. These types of solutions exploit certain structures in the data or the required task. Each time you improve your solution, you can compare it to the brute force solution, as a measure of progress.
- These types of efficient solutions lead us to sqrt(n), log(n), nlog(n) etc. types of complexities which are very common in algorithms.

Time Complexity Example 1

```
# Assume list manipulations are constant time

def some_conditions(list1):
    the_sum = 0
    if len(list1) < 42
        return the_sum + 5
    else:
        list1 = list1[1:]
        list1[40] = 100
        return list1[41]
    return -1
```

iClicker Question

len(list1) = n

A. O (1)
B. O (n)
C. O (log n)
D. O (n logn n)
E. Something Else

Time Complexity Example 2

```
# Assume list manipulations are constant time

def many_conditions(list1):
    the_sum = 0
    if len(list1) < 42
        return the_sum + 5
    else:
        list1 = list1[1:]
        list1[40] = 100
        if list1[30] == list1[3] ** 3
            return True
        else:
            for i in range(len(list1),0,1):
                the_sum += i
        return the_sum // 5
    return -1

# Hint: consider the difference between
# print(list(range(5, 0, 1)))
# print(list(range(5, 0, -1)))
```

iClicker Question

len(list1) = n

A. O (1)
B. O (n)
C. O (log n)
D. O (n log n)
E. Something Else

Time Complexity Example 3

```
def the_loop(i):
    the_sum = 0
    for j in range(i):
        the_sum = the_sum + j
    return sum

the_sum = 0
for i in range(1, n+1):
    the_sum = the_sum * the_loop(i)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 4

```
def the_loop(n):
    the_sum = 0
    j = n
    while j > 1:
        the_sum = the_sum + j
        j = j // n
    return the_sum

the_sum = 0
for i in range(1, n+1):
    the_sum = the_sum * the_loop(n)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 5

```
def the_loop(n):
    the_sum = 0
    j = n
    while j > 0:
        the_sum = the_sum + j
        j = j // 2
    return the_sum

the_sum = 0
for i in range(1, n+1):
    the_sum = the_sum * the_loop(n)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 6

```
def the_loop(n):
    the_sum = 0
    j = n
    while j > 0:
        if j >= 100:
            the_sum = the_sum + j
        j = j // 2
    return the_sum

the_sum = 0
for i in range(1, n+1):
    the_sum = the_sum * the_loop(n)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 7

```
def the_loop(n):
    the_sum = 0
    j = n
    while j > 1:
        if j <= n:
            j = 6
        else:
            j = n - 1
        j = j // 5
    return the_sum

the_sum = 0
for i in range(1, n+1):
    the_sum = the_sum * the_loop(n)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 8

```
def the_loop(i):
    the_sum = 0
    for j in range(i):
        if j == 0:
            break
        else:
            the_sum += j
    return the_sum

the_sum = 0
for i in range(n):
    the_sum = the_sum * the_loop(i // 5)
```

iClicker Question

A. O (1)
B. O (n)
C. O (log n)
D. O (n ^ 2)
E. O (n log n)

Time Complexity Example 9

```
# Assume n is an integer
the_sum = 0
for i in range(0, n):
    j = n ** 3
    while j > 0:
        the_sum += j
        j = j // 2
```

iClicker Question

A. O (n)
B. O (log n)
C. O (log n^3)
D. O (n log n)
E. O (n log n^3)

Time Complexity Example 10

```
the_sum = 0
for i in range(0, n):
    j = n ** 2
    while j > 0:
        the_sum += j
        j = j // 2
        if(the_sum >= 0):
            for k in range(n):
                print('k ** 2 is:', k ** 2)
        else:
            the_sum += 1
```

iClicker Question

A. O (n)
B. O (n ^ 2)
C. O (log n)
D. O (n^2 log n)
E. O (n log n)

Time Complexity Example 11

```
# n is a positive integer
the_sum = 0
j = n
while j > 0:
    the_sum += j
    j -= 1
    the_map = map (lambda x: x ** 2, range(n))

list(the_map)
```

iClicker Question

A. O (n)
B. O (n ^ 2)
C. O (log n^2)
D. O (n log n)
E. Something Else

Time Complexity Example 12

```
# n is a positive integer
the_sum = 0
j = n
res_list = []
while j > 0:
    the_sum += j
    j = j // 2
    the_map = map(lambda x: x ** 2, range(n))
    res_list.append(list(the_map))
```

iClicker Question

A. O (n)
B. O (n ^ 2)
C. O (log n)
D. O (n log n)
E. O (n^2 log n)

Time Complexity Example 13

```
# n is a positive integer

def annoying_loop(n):
    n = n ** 2
    for i in range(n):
        print('Annoying message')
    return n

# n is a positive integer

j = n
res_list = []
while j > 0:
    the_sum += j
    j = j // 2
    the_map = map(annoying_loop, range(n))
    res_list.append(list(the_map))
```

iClicker Question

A. O (n)
B. O (n ^ 2)
C. O (log n)
D. O (n log n)
E. O (n^2 log n)