Problem Solving Strategies and Computational Approaches SCS1304

Handout 3 : Time Complexity & Asymptotic notation

Prasad Wimalaratne PhD(Salford),SMIEEE

Overview

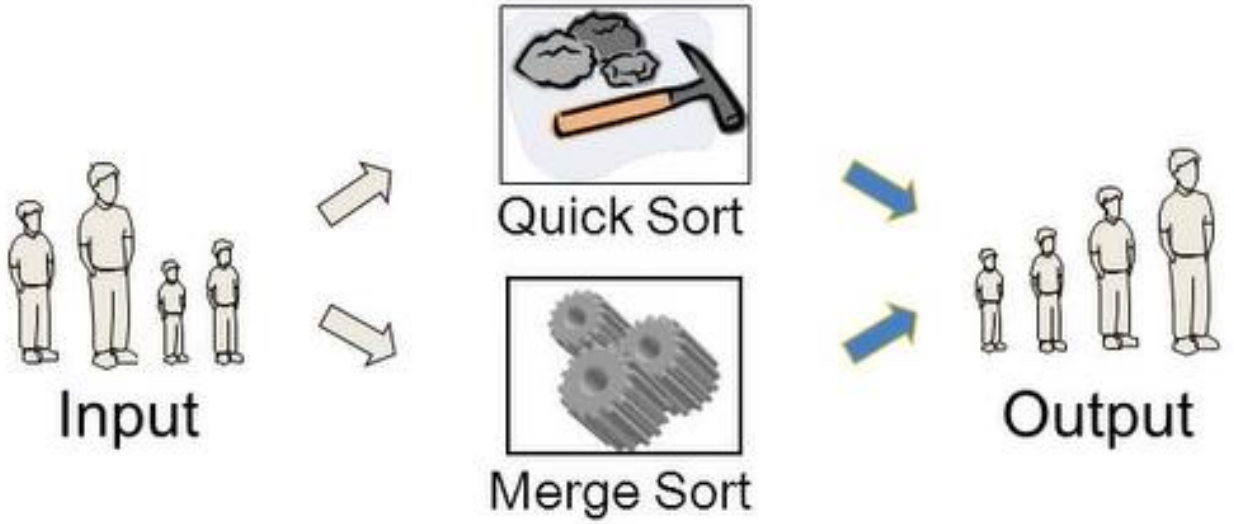
- Analysis of Algorithms
  - Order of Algorithms
  - Practical Considerations
- Worst-case time complexity
- Best-case time complexity
- Average-case time complexity

[Chapter 1] (part2)
Algorithms : Efficiency, Analysis, And Order

Analysis of Algorithms

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

How to evaluate algorithms?



- Which one is better?
- What are the criteria?

Analysis of Algorithms

- Analysis involves evaluating algorithms to understand their efficiency, performance, and behavior:

1. Correctness: Ensuring that an algorithm produces the correct output for all possible valid inputs.
2. Complexity Analysis: Determining the time and space complexity of an algorithm. This helps in predicting and comparing the efficiency of different algorithms.
3. Empirical Analysis: Testing algorithms on real data sets to measure their actual performance in practice. This complements theoretical complexity analysis.

Order of Algorithms

- "Order" categorizes algorithms based on their efficiency and complexity:
  1. Comparison of Sorting: Algorithms like Merge Sort, Quick Sort, and Heap Sort are categorized by their time complexity for sorting elements in arrays or lists.
  2. Searching Algorithms: Techniques like Binary Search and Linear Search are evaluated based on their time complexity for finding elements in data structures.
  3. Dynamic Programming: Algorithms that break down complex problems into smaller subproblems and use memoization or tabulation to optimize solutions.

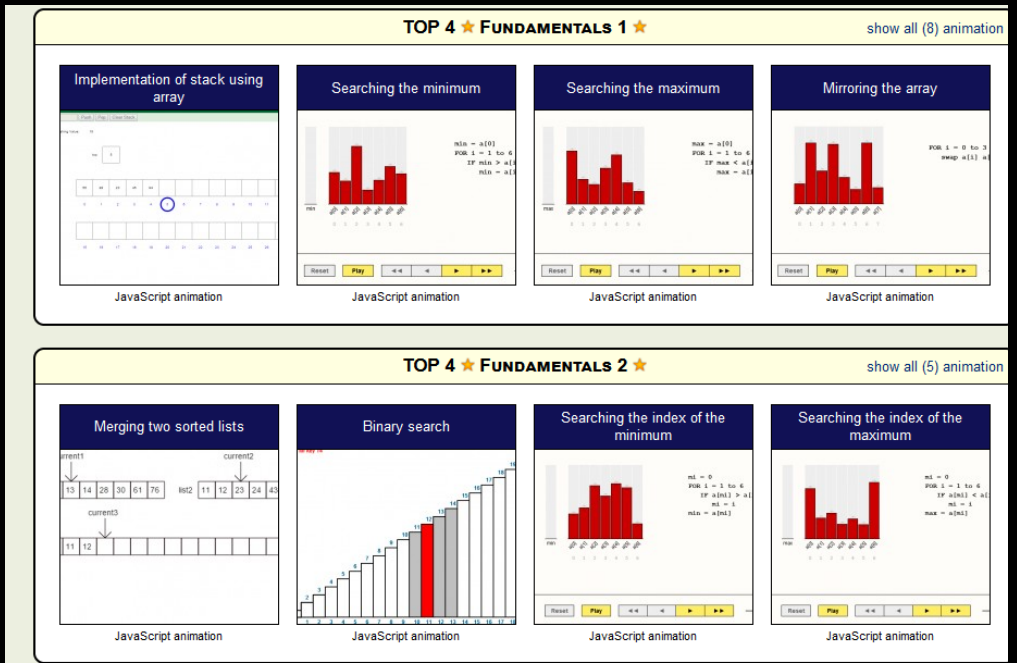
Practical Considerations

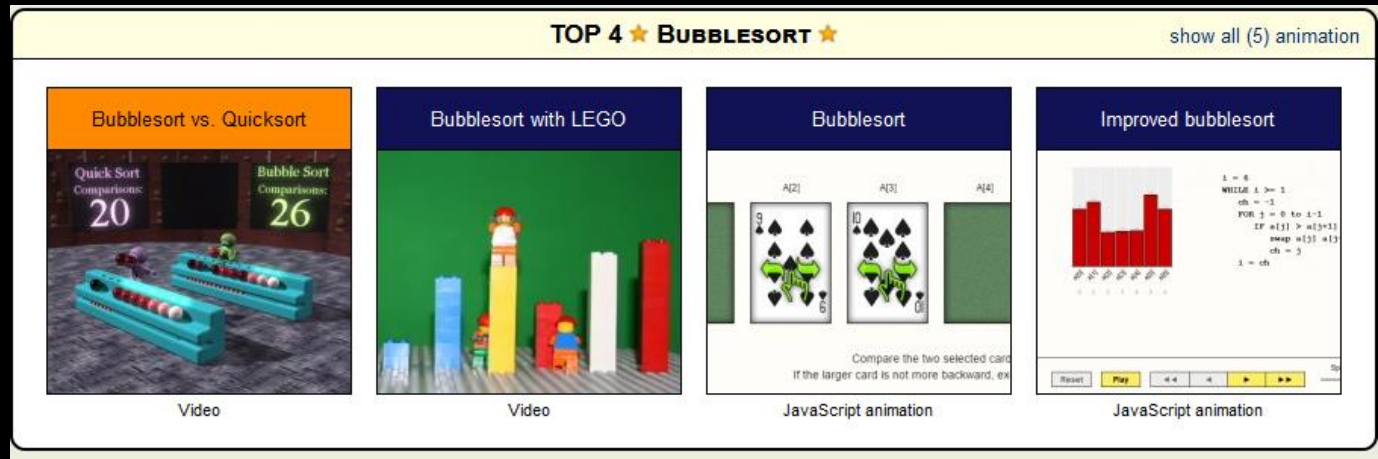
1. Algorithm Selection: Choosing the right algorithm based on problem requirements, input size, and desired performance characteristics.
2. Optimization Techniques: Strategies such as pruning, memoization, and parallelism to improve algorithm efficiency and reduce time or space complexity.
3. Algorithm Design Paradigms: Understanding different approaches like divide and conquer, greedy algorithms, and dynamic programming for solving diverse computational problems efficiently.

(to be be covered in upcoming lessons)

Algorithm Animations and Visualizations

- collection of computer science algorithm animations and visualizations
- Refer the following URL for visualizing commonly used algorithms
- https://algoanim.ide.sk/





Analysis of Algorithms

- In general, the running time of an algorithm increases with the size of the input, and the total running time is roughly proportional to how many times some basic operation (such as a comparison instruction) is carried out.
- We therefore analyze the algorithm's efficiency by determining the number of times some basic operation is carried out as a function of the size of the input.
- After determining the input size, we pick some instruction or group of instructions such that the total work done by the algorithm is roughly proportional to the number of times this instruction or group of instructions is done.
- We call this instruction or group of instructions the **basic operation** in the algorithm.
  - e.g how many times x is compared with elements of the array in earlier examples ?

Analysis of Algorithms ctd..

- In general, a time complexity analysis of an algorithm is the determination of how many times the basic operation is carried out for each value of the input size.
- Although we do not want to consider the details of how an algorithm is implemented, we will ordinarily assume that the basic operation is implemented as efficiently as possible.
- best, worst, and average cases of a given algorithm express what the resource usage is at least, at most and on average, respectively.
- Usually the resource being considered is running time, i.e. time complexity, but could also be memory or some other resource.

Worst-case time complexity analysis

- The first is to consider the maximum number of times the basic operation is done.
- For a given algorithm, W(n) is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n.
- W(n) is called the worst-case time complexity of the algorithm, and the determination of W(n) is called a worst-case time complexity analysis.

Worst-Case Time Complexity (Sequential Search)

- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.
- The basic operation is done at most n times, which is the case if x is the last item in the array or if x is not in the array. Therefore,

$$W(n) = n.$$

Algorithm 1.1

Sequential Search

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

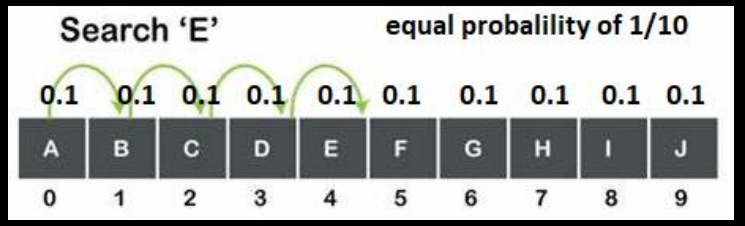
```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)

{
   location = 1;
   while (location<= n && S[location] != x)
      location++;
   if (location > n)
      location=0;
}
```

Worst-Case Time Complexity (Sequential Search)

- Although the worst-case analysis informs us of the absolute maximum amount of time consumed, in some cases we may be more interested in knowing how the algorithm performs on the average.
- For a given algorithm, A(n) is defined as the average (expected value) of the number of times the algorithm does the basic operation for an input size of n.
- A(n) is called the average-case time complexity of the algorithm, and the determination of A(n) is called an average-case time complexity analysis.

Average -Case Time Complexity (Sequential Search)

- To compute A(n), we need to assign probabilities to all possible inputs of size n.
- It is important to assign probabilities based on all available information.
- For example, our next analysis will be an average-case analysis of Sequential Search
- Assumption: We will assume that if x is in the array, it is equally likely to be in any of the array slots.
- If we know only that x may be somewhere in the array, our information gives us no reason to prefer one array slot over another.



Average -Case Time Complexity (Sequential Search) ctd.

- Therefore, it is reasonable to assign equal probabilities to all array slots.
- This means that we are determining the average search time when we search for all items the same number of times.

- Note: If we have information indicating that the inputs will not arrive according to this distribution, we should not use this distribution in our analysis.

Average -Case Time Complexity (Sequential Search) ctd..

- For example, if the array contains first names and we are searching for names that have been chosen at random from all people in a country, an array slot containing the common names will probably be searched more often than one containing the uncommon names
- We should not ignore this information and assume that all slots are equally likely.
- it is usually harder to analyze the average case than it is to analyze the worst case.

Average-Case Time Complexity (Sequential Search)

- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.

- We first analyze the case in which it is known that x is in S, where the items in S are all distinct, and where we have no reason to believe that x is more likely to be in one array slot than it is to be in another.
- Based on this information, for $1 \leq n$, the probability that x is in the k^{th} array slot is 1/n.

Average-Case Time Complexity (Sequential Search) ctd..

- If x is in the k^{th} array slot, the number of times the basic operation is done to locate x (and, therefore, to exit the loop) is k.
- This means that the average time complexity is given by

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- As we would expect, on the **average, about half the array** is searched.

Average-Case Time Complexity (Sequential Search) ctd..

- Next we analyze the case in which x may not be in the array.
- To analyze this case we must assign some probability p to the event that x is in the array.
- If x is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n.
- The probability that x is in the k^{th} slot is then p/n, and the probability that it is not in the array is 1 – p.

Average-Case Time Complexity (Sequential Search) ctd..

- Recall(Algo 1.2) that there are k passes through the loop if x is found in the k^{th} slot, and n passes through the loop if x is not in the array.
- The average time complexity is therefore given by

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{p}{n}\right) + n(1-p)$$

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}.$$

- The last step in this triple equality is derived with algebraic manipulations. If p = 1, A(n) = (n + 1)/2, as before, whereas if p = 1/2, A(n) = 3n/4 + 1/4. This means that about 3/4 of the array is searched on the average.

Average-Case Time Complexity (Sequential Search) ctd..

- Caution1: An average can be called “typical” only if the actual cases do not deviate much from the average (that is, only if the standard deviation is small).

Best-case time complexity analysis.

- A final type of time complexity analysis is the determination of the smallest number of times the basic operation is done.
- For a given algorithm, B(n) is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n.
- So B(n) is called the best-case time complexity of the algorithm, and the determination of B(n) is called a best-case time complexity analysis.

Best-Case Time Complexity (Sequential Search)

- Basic operation: the comparison of an item in the array with x.
- Input size: n, the number of items in the array.
- Because n ≥ 1, there must be at least one pass through the loop, If x = S[1], there will be one pass through the loop regardless of the size of n.
- Therefore,

$$B(n) = 1.$$

Algorithm 1.1

Sequential Search

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)

{
   location = 1;
   while (location<= n && S[location] != x)
      location++;
   if (location > n)
      location=0;
}
```

Time Complexity

- An average-case analysis is valuable because it tells us how much time the algorithm would take when used many times on many different inputs.
- This would be useful, for example, in the case of a sorting algorithm that was used repeatedly to sort all possible inputs.
- Often, a relatively slow sort can occasionally be tolerated if, on the average, the sorting time is good. Quicksort(Covered in later lessons) , that does exactly this.
- Quicksort is one of the most popular sorting algorithms.

Time Complexity

- As noted previously, an average-case analysis would not suffice in a system that monitored a nuclear power plant.
  - Ex: Life Critical Systems or Mission Critical Systems? State examples
- In this case, a worst-case analysis would be more useful because it would give us an upper bound on the time taken by the algorithm.
- For both the applications just discussed, a best-case analysis would be of little value.

Further examples

- Best Case : Best case performance used in computer science to describe an algorithm's behavior under optimal conditions. An example of best case performance would be trying to sort a list that is already sorted using some sorting algorithm. e.g. [1,2,3] --> [1,2,3]
- Average Case : Average case performance measured using the average optimal conditions to solve the problem. For example a list that is neither best case nor, worst case order that you want to be sorted in a certain order. e.g. [2,1,5,3] --> [1,2,3,5] OR [2,1,5,3] --> [5,3,2,1]
- Worst Case : Worst case performance used to analyze the algorithm's behavior under worst case input and least possible to solve the problem. It determines when the algorithm will perform worst for the given inputs. An example of the worst case performance would be a list of names already sorted in ascending order that you want to sort in descending order. e.g. [1,2,3,5] --> [5,3,2,1]

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Analysis of an Algorithm

- "analysis of an algorithm" means an efficiency analysis in terms of either time or memory.
- Algorithms with time complexities such as n and 100n are called linear-time algorithms because their time complexities are linear in the input size n,
  - whereas algorithms with time complexities such as n^2 and $0.01n^2$ are called quadratic time algorithms because their time complexities are quadratic in the input size n.

Analysis of an Algorithm ctd..

- Any linear-time algorithm is **eventually** more efficient than any quadratic-time algorithm.
- In the theoretical analysis of an algorithm, we are interested in **eventual behavior**.
- Next we will show how **algorithms can be grouped according to their eventual behavior**.
- In this way we can readily determine whether one algorithm's eventual behavior is better than another's.

An Intuitive Introduction to Order?

- Functions such as $5n^2$ and $5n^2 + 100$ are called **pure quadratic** functions because they contain no linear term, whereas a function such as $0.1n^2 + n + 100$ is called a **complete quadratic** function because it contains a linear term. Table shows that eventually the quadratic term dominates this function.

n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

- That is, the values of the other terms eventually become insignificant compared with the value of the quadratic term.

An Intuitive Introduction to Order

- Therefore, although the function is **not a pure quadratic function**, we can **classify** it with the **pure quadratic** functions.
- This means that if some algorithm has this time complexity, we can call the algorithm a quadratic-time algorithm.
- Intuitively, it seems that we should always be able to throw away low-order terms when classifying complexity functions.
- For example, it seems that we should be able to classify $0.1n^3 + 10n^2 + 5n + 25$ with pure cubic functions.

Representative Order Functions

- O(lg n)
- O(n) : linear
- O(n lg n) log-Linear
- O(n^2) : quadratic
- O(n^3) : cubic
- O(2^n) : exponential
- O($n!$) : combinatorial

O	Complexity	
O(1)	constant	fast
O(log n)	logarithmic	
O(n)	linear time	
O(n * log n)	log linear	
O(n^2)	quadratic	
O(n^3)	cubic	
O(2^n)	exponential	
O(n!)	factorial	slow

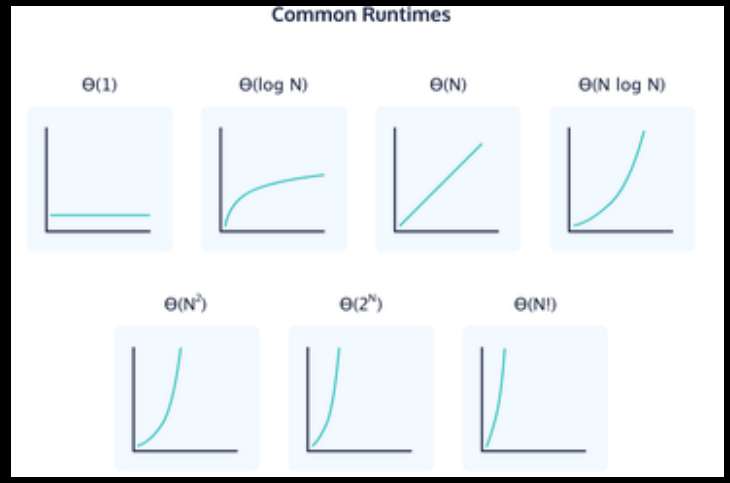
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Base of Log Matter in Time Complexity?

- In Big-O complexity analysis, it does not matter what the logarithm base is. (they are asymptotically the same,
  - i.e. they differ by only a constant factor/implies that it is only a constant factor difference):
  - $O(\log_2 N) = O(\log_{10} N) = O(\log_e N)$
- Mostly in mathematics it implicitly mean to base *e*.
- In Computing it tend to favors base 2, but it does not actually matter.

Common runtimes examples

- O(1) Constant Time: The algorithm's running time does not depend on the size of the input; it performs a fixed number of operations.
- O(log n) Logarithmic Time: The algorithm's running time grows logarithmically with the size of the input.
- O(n) Linear Time: The algorithm's running time scales linearly with the size of the input.
- O(n log n) Linearithmic Time: The algorithm's running time grows in proportion to n times the logarithm of n.
- O(n^2) Quadratic Time: The algorithm's running time is directly proportional to the square of the input size.
- O(2^n) Exponential Time: The algorithm's running time doubles with each increase in the input size.



Example

- The quadratic term eventually determines

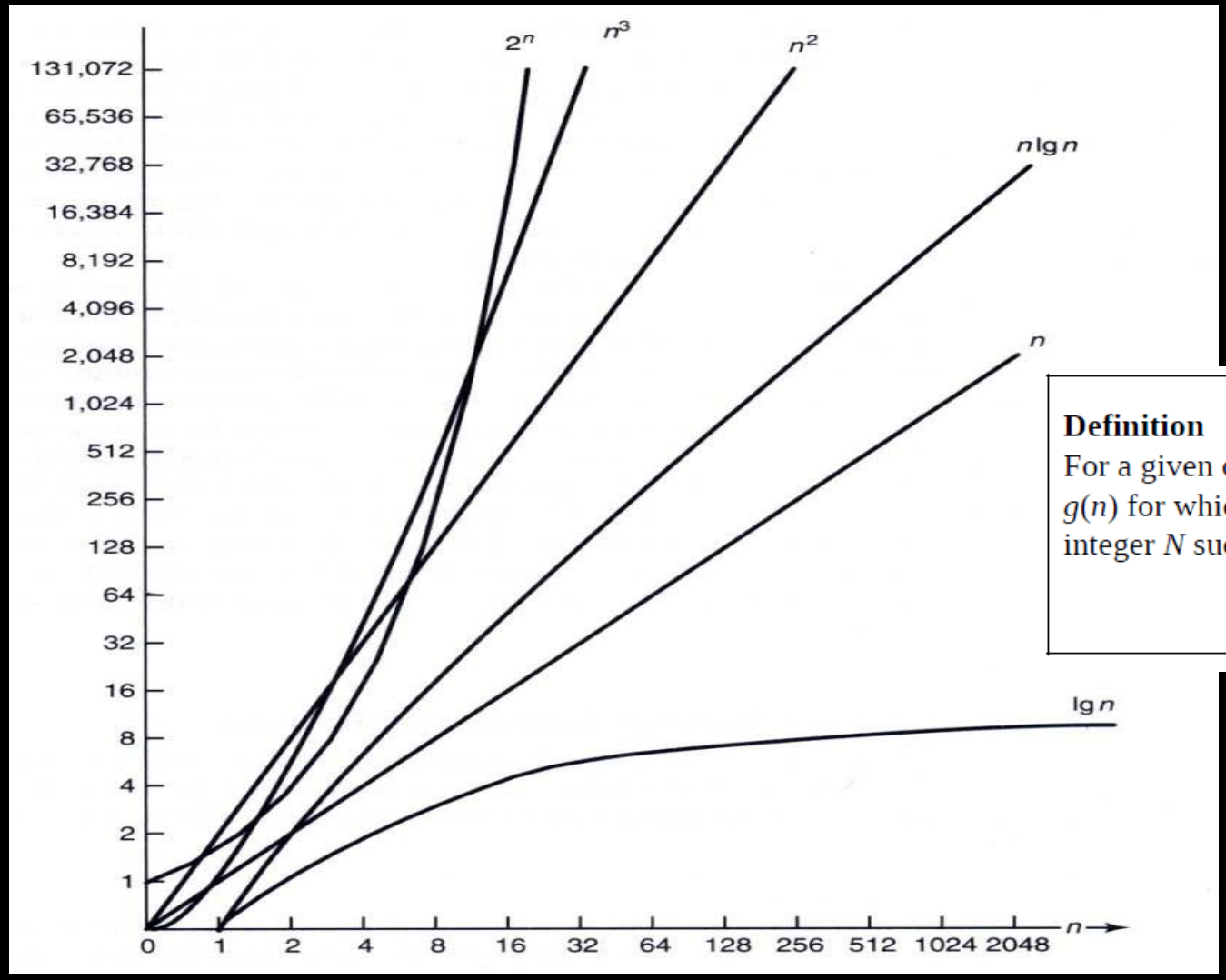
n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

Big O

denoted as O(n), is a measure of the growth of the running time of an algorithm proportional to the size of the input.

- The “O” in Big O stands for “order ” while the value within parentheses indicates the growth rate of the algorithm.
- In the case of O(N), we refer to it as complexity. This implies that the execution time of the algorithm increases proportionally with respect, to the size of the input. If we double our input size we can expect twice as much execution time.
- If droping the non-dominant terms, then how about n*log n?"
  - While it is true that we drop the non-dominant terms in Big O, that is generally when we are adding two different complexities, such as $n^2 + n$.
  - Here, we are using multiplication. We can not simplify n * log n any further, so we keep both terms.

Growth Rates of Some Complexity Functions



$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Definition
For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

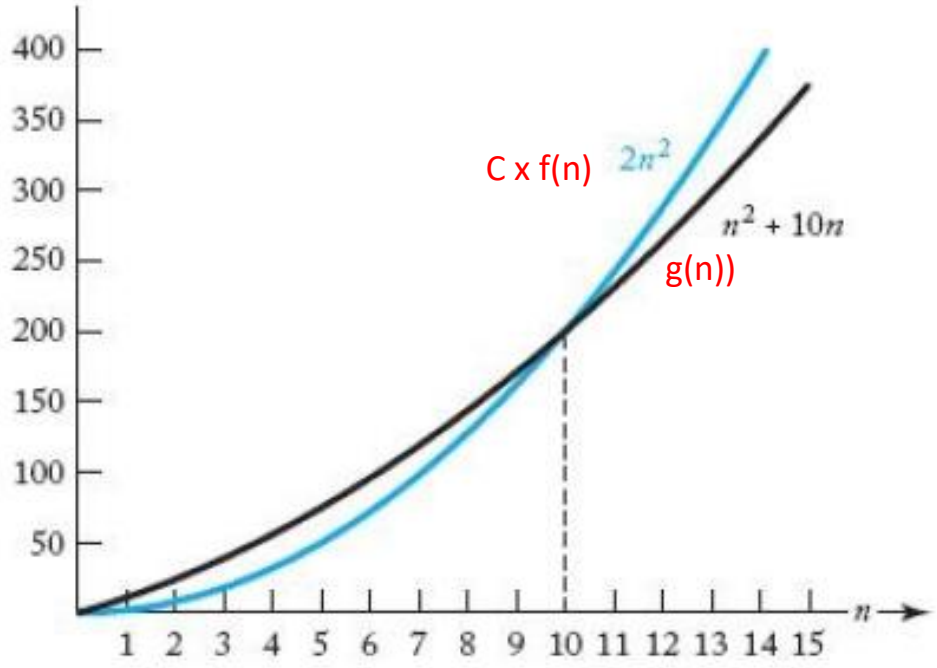
$$g(n) \leq c \times f(n).$$

If $g(n) \in O(f(n))$, we say that $g(n)$ is ***big O*** of $f(n)$.

Although $g(n)$ starts out above $cf(n)$ in that figure, eventually it falls beneath $cf(n)$ and stays there. Figure 1.5 shows a concrete example. Although $n^2 + 10n$ is initially above $2n^2$ in that figure, for $n \geq 10$

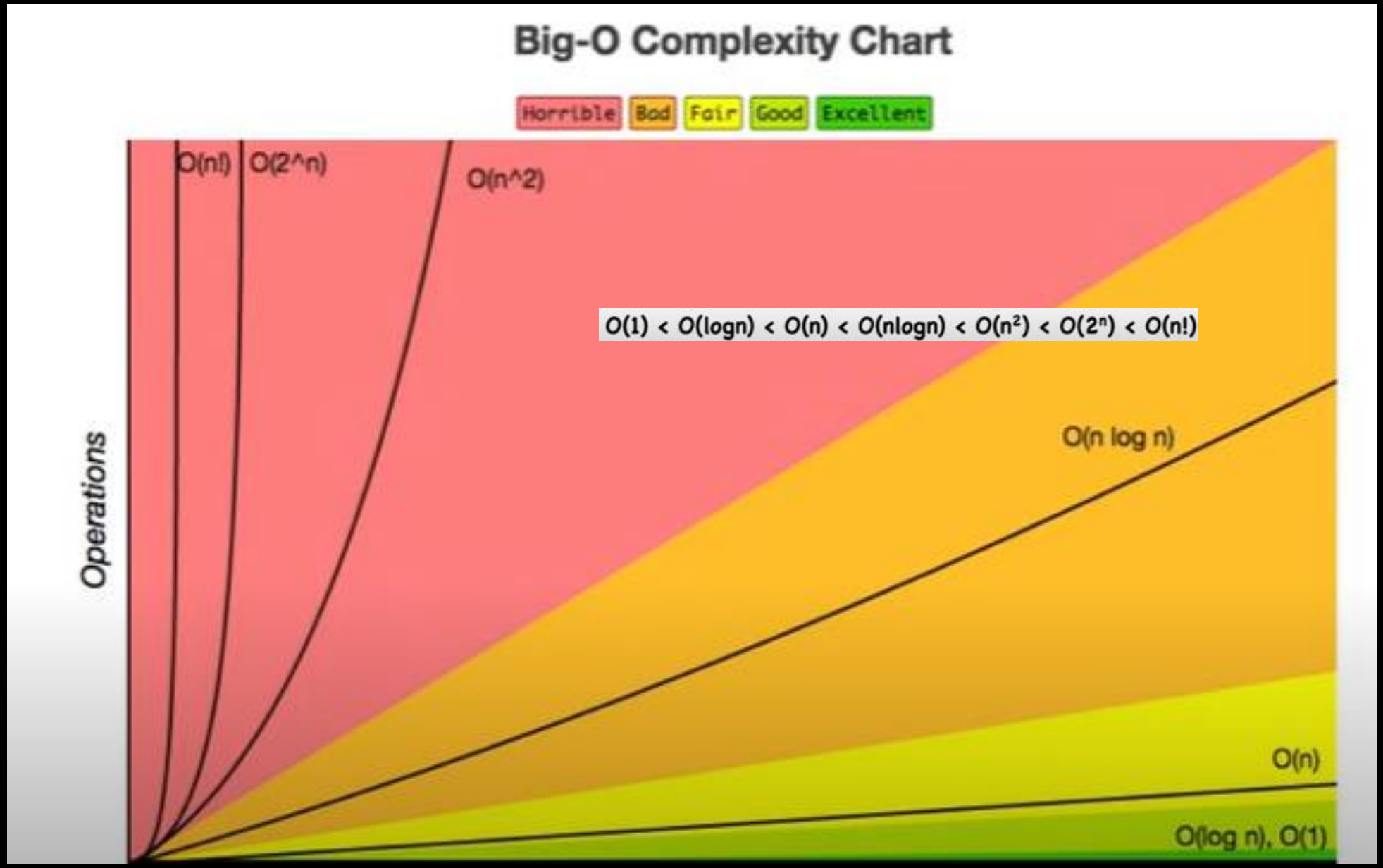
$$g(n) \leq c \times f(n).$$

Figure 1.5 The function $n^2 + 10n$ eventually stays beneath the function $2n^2$.



Example: Execution Times for Algorithms with the Given Time Complexities

n	$f(n) = lg\, n$	$f(n) = n$	$f(n) = n\, lg\, n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years	
10^7	0.023 μs	0.01 s	2.660 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.660 s	115.70 days	3.17×10^7 years	
10^9	0.030 μs	1.00 s	29.900 s	31.70 years		



Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

Big O?

- Simplified Analysis of an Algorithms Efficiency

1. complexity in terms of input size, N
2. machine-independent
3. basic computer steps
4. time & space

Type of Measurement?

worst-case
best-case
average-case

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

Big O?

- General Rules

1. ignore constants

   5n → O(n)

2. certain terms "dominate" others

   $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

   i.e., ignore low-order terms

in practice

1. constants matter
2. be cognizant of best-case and average-case

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

constant time

O(1) "big oh of one"

```
x = 5 + (15 * 20);
```

independent of input size, N

constant time

```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

total time = O(1) + O(1) + O(1) = O(1)

3 * O(1)

linear time

N * O(1) = O(N)

```
for x in range (0, n):
    print x; // O(1)
```

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

```
y = 5 + (15 * 20);          O(1)
for x in range (0, n):      } O(N)
    print x;
```

total time = O(1) + O(N) = O(N)

$O(N^2)$

```
x = 5 + (15 * 20);          O(1)
for x in range (0, n):      } O(N)
    print x;
for x in range (0, n):      } O(N^2)
    for y in range (0, n):
        print x * y;
```

$O(N^2)$

```
if x > 0:
    // O(1)
else if x < 0:
    // O(logn)
else:
    // O(n^2)
```

$O(N^2)$

```
for x in range (0, n):
    for y in range (0, n):
        print x * y; // O(1)
```

Big-O notation in 5 minutes
https://www.youtube.com/watch?v=__vX2sjlpXU

Examples

- The algorithm complexity ignores the constant value in algorithm analysis and takes only the highest order.
- Suppose we had an algorithm that takes, $5n^3+n+4$ time to calculate all the steps, then the algorithm analysis ignores all the **lower order polynomials and constants** and takes only $O(n^3)$.

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Examples ctd..

Simple Statement

This statement takes O(1) time.

```
int y= n + 25;
```

Constant Time Complexity O(1):
The time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain a loop, recursion, and call to any other non-constant time function.
i.e. set of non-recursive and non-loop statements

If Statement

The worst case O(n) if the if statement is in a loop that runs n times, best case O(1)

```
if( n> 100)
{
    …
}else{
    ..
}
```

Further Examples ctd..

For While Loops

The for loop takes n time to complete and so it is O(n).

```
for(int i=0; i<n ;I ++)
{
   ..
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Examples ctd..

- The while loop takes n time as well to complete and so it is O(n).

```
int i=0;
while( i<n)
{
    ..
    i++;
}
```

- If the for loop takes n time and i increases or decreases by a constant, the cost is O(n)

```
for(int i = 0; i < n; i+=5)
   sum++;

for(int i = n; i > 0; i-=5)
   sum++;
```

denoted as O(n), is a measure of the growth of the running time of an algorithm proportional to the size of the input. In an O(n) algorithm, the running time increases linearly with the size of the input.

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Examples ctd..

- If the for loop takes n time and i increases or decreases by a multiple, the cost is O(log(n))

```
for(int i = 1; i < =n; i*=2)
  sum++;

for(int i = n; i > 0; i/=2)
  sum++;
```

The time Complexity of a loop is considered as O(log n) if the loop variables are divided/multiplied by a constant amount. And also for recursive calls in the recursive function, the Time Complexity is considered as O(log n).

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Examples ctd..

Nested loops

If the nested loops contain sizes n and m, the cost is O(nm)

```
for(int i=0; i<n; i++)
{
   for(int i=0; i<m; i++){
      ..
   }
}
```

Ex: What is the time complexity if C = AB for an n × m matrix A and an m × p matrix B, (then C is an n × p matrix with entries)

Input: matrices A and B

Let C be a new matrix of the appropriate size

For i from 1 to n:

- For j from 1 to p:
  - Let sum = 0
  - For k from 1 to m:
    - Set sum $\leftarrow$ sum + $A_{ik} \times B_{kj}$
  - Set $C_{ij} \leftarrow$ sum

Return C

Refer: https://www.geeksforgeeks.org/how-to-analyse-loops-for-complexity-analysis-of-algorithms/

Examples ctd..

- If the first loop runs n times and the inner loop runs log(n) times or (vice versa), the cost is O(n*log(n))

```
for(int i=0; i<n ; i++)
{
   for(int j=1; i<=n; j*=2){
       ..
       ..
   }
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

Examples ctd..

- If the first loop runs n^2 times and the inner loop runs n times or (vice versa), the cost is $O(n^3)$

```
for(int j=0 ;j<n*n ;j++)
{
   for(int i=0 ;i<n ;i++){
      ..
       ..
   }
}
```

https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

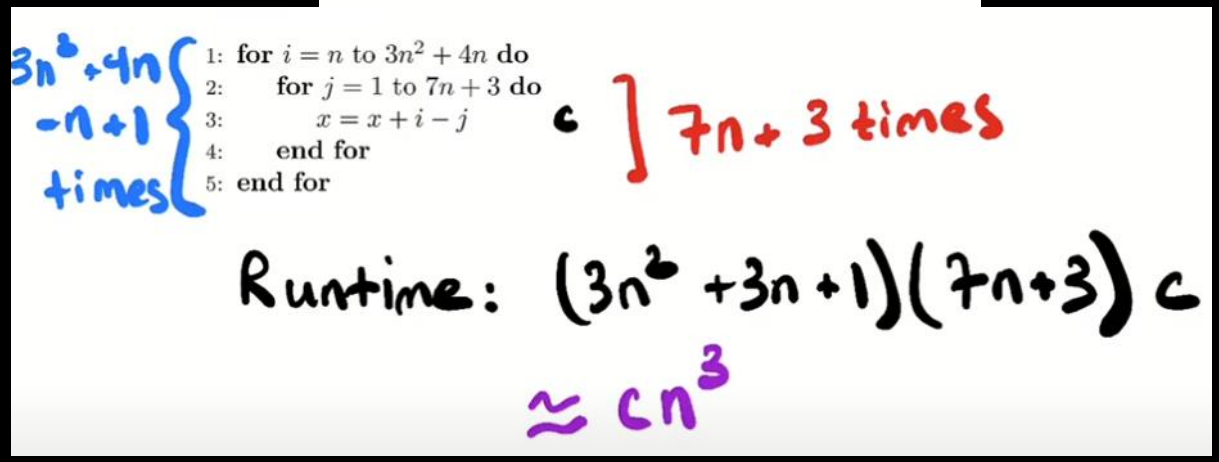
Examples ctd..

- If the first loop runs n times and the inner second loop runs n^2 times and the third loop runs n^2, then $O(n^5)$

```
for(int i = 0; i < n; i++)
    for( int j = 0; j < n * n; j++)
        for(int k = 0; k < j; k++)
            sum++;
```

Ex: What is O(?)

```
1: for i = n to 3n^2 + 4n do
2:     for j = 1 to 7n + 3 do
3:         x = x + i - j
4:     end for
5: end for
```



https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html

References

- Refer Asymptotic Notations 101: Big O, Big Omega, & Theta (Asymptotic Analysis Bootcamp)
  - https://www.youtube.com/watch?v=0oDAlMwTrLo&t=50s