Theory of Algorithms

Algorithm Efficiency

- We need measures of:
    - Input size (call this n)
    - Unit of measuring time
        - The basic operation of the algorithm
- We're usually interested in growth order:
    - $O(n^2)$ vs $O(n^3)$ is more important than $1412n^2$ vs $5n^3$
- We're interested in growth order
    - So $O(n^2)$ vs $O(n^3)$ is more important than *1412*n^2 vs *5*^{3}
- We are measuring the operational complexity
- Interested in the
    - Worst case
    - Average case
    - Best case

Example: Linear Search

What are the cases?

- Worst = n
    - Looks at all n elements in the array
- Average = n/2

- On average it will look at n/2 elements before finding the element being searched
- Therefore n/2. but the 1/2 is a constant, so it is simply left as n

- Best = constant time, or O(1) - as it nly looks at one element

Generally most interested in worst case. If the best case is good enough, we are interested in that

Typical Algorithm Efficiency Classes

n	**$\log_2 n$**	**n**	**$n \log_2 n$**	**n^2**	**n^3**	**2^n**	**n!**
10	3.3	10	33.2	1E+02	1E+03	1E+03	4E+06
100	6.6	100	664.4	1E+04	1E+06	1E+30	9E+157
1,000	10.0	1,000	9,965.8	1E+06	1E+09	1E+301	
10,000	13.3	10,000	132,877.1	1E+08	1E+12		
100,000	16.6	100,000	1,660,964.0	1E+10	1E+15		
1,000,000	19.9	1,000,000	19,931,568.6	1E+12	1E+18		

Big O - Intuitively

- Algorithm Executes
  - `2n+10` operations
  - We are not interested in the 2 or the 10.
  - This is simply O(n)
- Algorithm Executes:
  - $3n^2 + 9n + 5$
  - Not interested in the 3, 9n or 5.
  - This is $O(n^2)$

Example

- Laptop sorts array of 100 million items in 30 seconds using:
  - n*logn = 2 657 542 476 -> 30 seconds
  - n = 10 000 000 000 000 000 = 10^{16}
  - 10^{16} / 2 657 542 476 = 3762875
  - 3762875 x 30 seconds = 112 886 248s
  - 3.5 years
- Insertion sort and Bubblesort are impractical for large data sets

Definition Big O

A function *$t(n) \in$ of $O(g(n))$*

If there is a c and an n_0 such that t(n) <= cg(n) for all n>=n_0.

- Example 1: 100n+5 ∈ of O(n)
  - Set c to 101 and n_0 to 6. Then prove that 100n+5 <=101n for all n >=6 Simple to prove with induction
- Example 2: 100n+5 ∈ O(n^2)
  - Set c = 21 and n_0 = 5. So prove that 100n+5<=21n^2

Example

100n+5 element O(n) set say c to 101 and n_0 to 6 Prove 100n+5 <= 100n for all n>=6 by induction

Omega Ω (Not important)

- A function t(n) ∈ Ω(g(n)) if there is a c and an n_0 such that t(n) >= cg(n) for all n >= n_0
- Same as Big O, except **bigger than** instead of **smaller than** sign

Theta Θ (Important)

- Consider sequentially summing an array
  - If the efficiency class is O(n)
  - But it is also by our definition O(n^2) and O(2^n) and O(nlogn)
  - It is NOT O(logn)
- Big O(n^2) means the algorithm's basic operations executes in proportion to n^2 or better
- How do we say that summing array's basic operation executes **exactly** proportional to n?
  - Theta Θ
- Efficiency class for summing an array is Θ(n). It is not:
  - Θ(n^2) or Θ(nlogn).
  - It is precisely Θ(n)
- A function t(n) ∈ Ω(g(n)) if there is a c_1, c_2 and n_0 such that c_2g(n) <= t(n) <= c_1g(n) for all n>=n_0
- Same as big O except functions in this class cannot be in a more efficient class
  - We use this a lot:
    - 100n+5 ∈ Θ(n)
    - 100n+5 ∈ O(n^2)
    - But 100n + 5 is NOT ∈ Θ(n^2)
- If an algorithm is in Θ(g(n)):

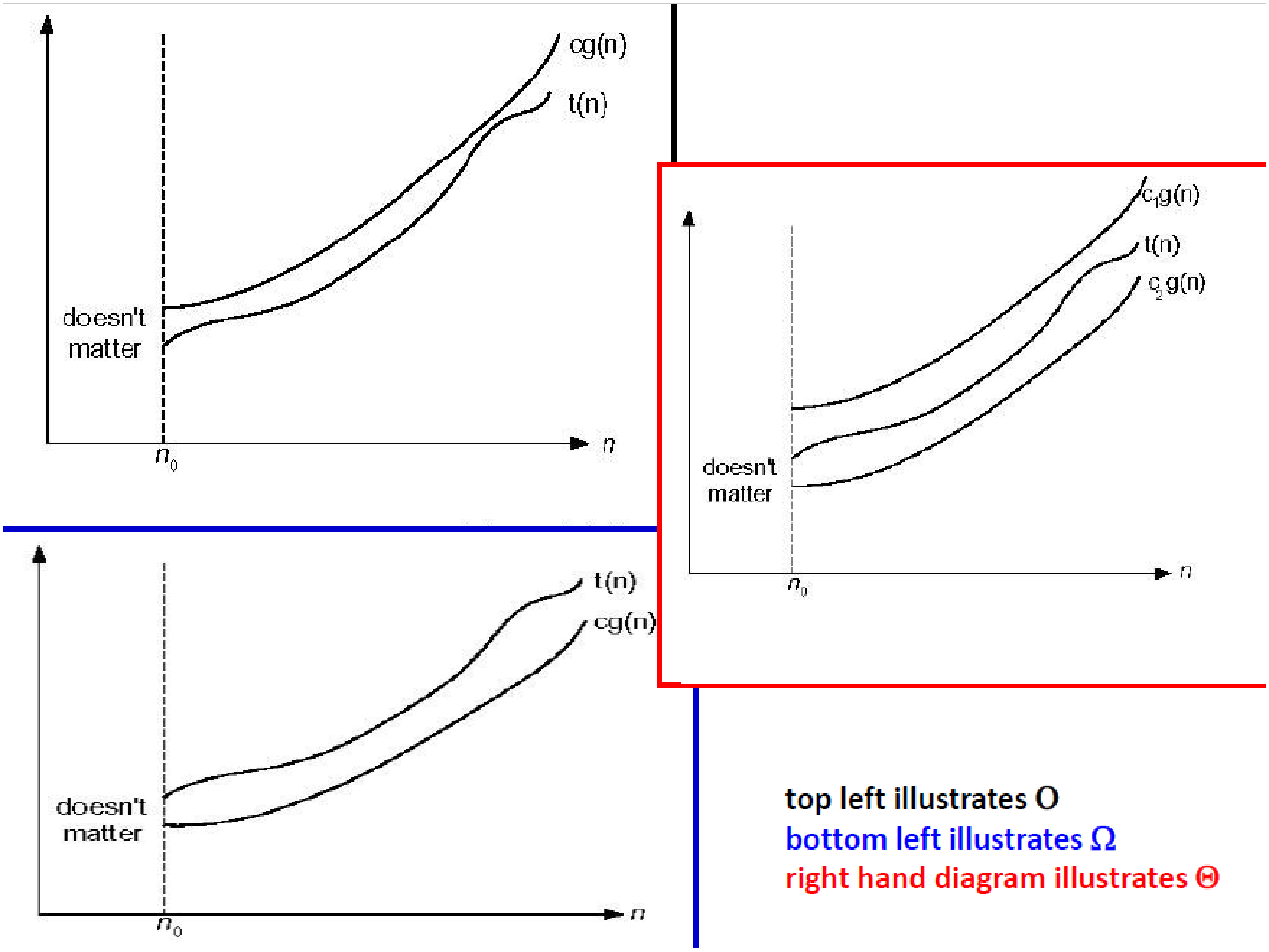
- It means that the running time of the algorithm as n (input size) gets larger is proportional to g(n)

- If an algorithm is in O(g(n))

    - It means that the running time of the algorithm as n gets larger is at most proportional to g(n)

> Big Oh is NO worse

> Big Omega is oh my god



Check this out

Formal Definitions - Summary

- Definition: *f(n) ∈ O(g(n))* iff there exists positive constant c and non-negative integer n_0 such that ** f(n) <= c g(n)* for every n >=n_0

- Definition: *f(n) ∈ Ω(g(n))* iff there exist positive constant c and non-negative integer n_0 such that

    - *f(n) >= c g(n)* for every n >= n_0

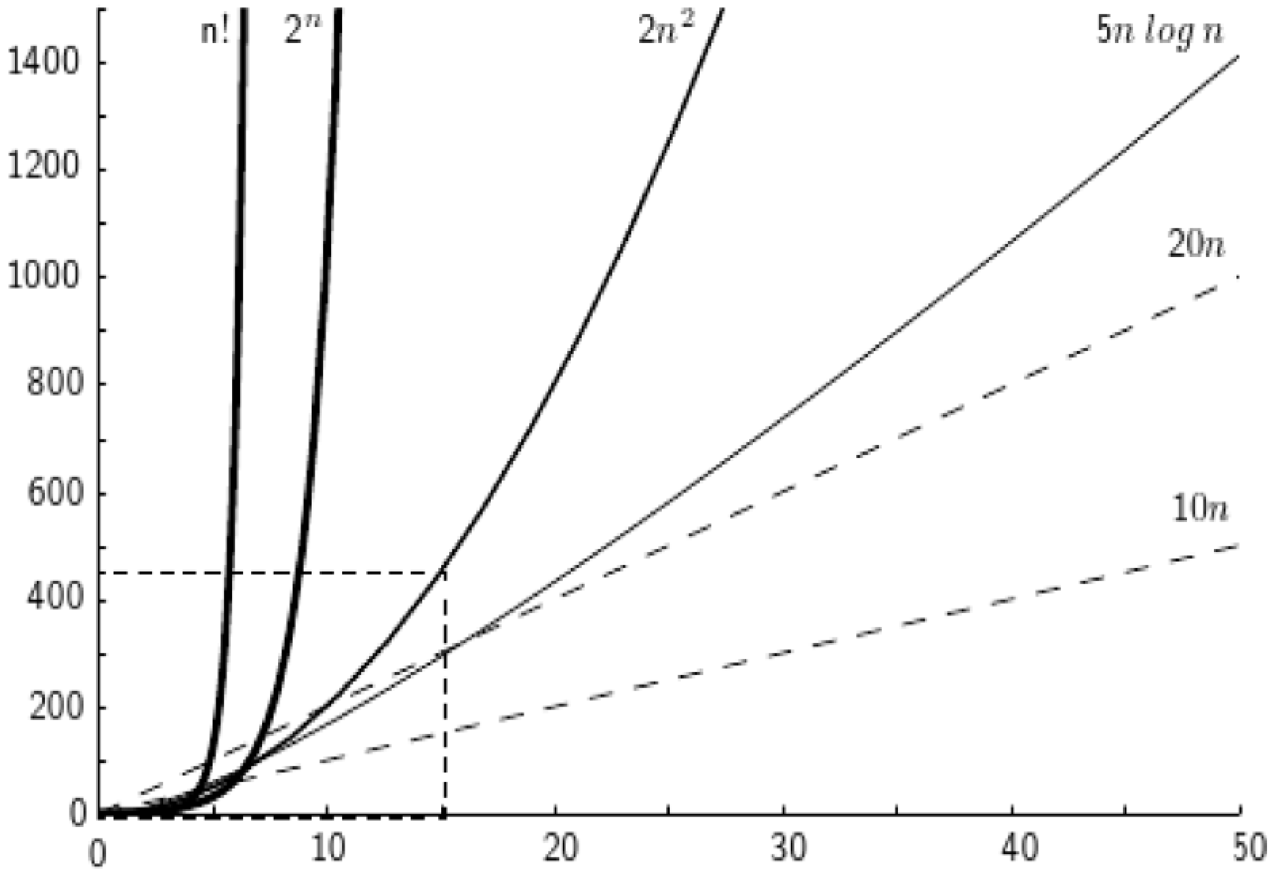
- Definition: *f(n) ∈ Θ(g(n))* iff there exist positive constants c_1 and c_2 and non-negative integer n_0 and non-negative integer n_0 such that

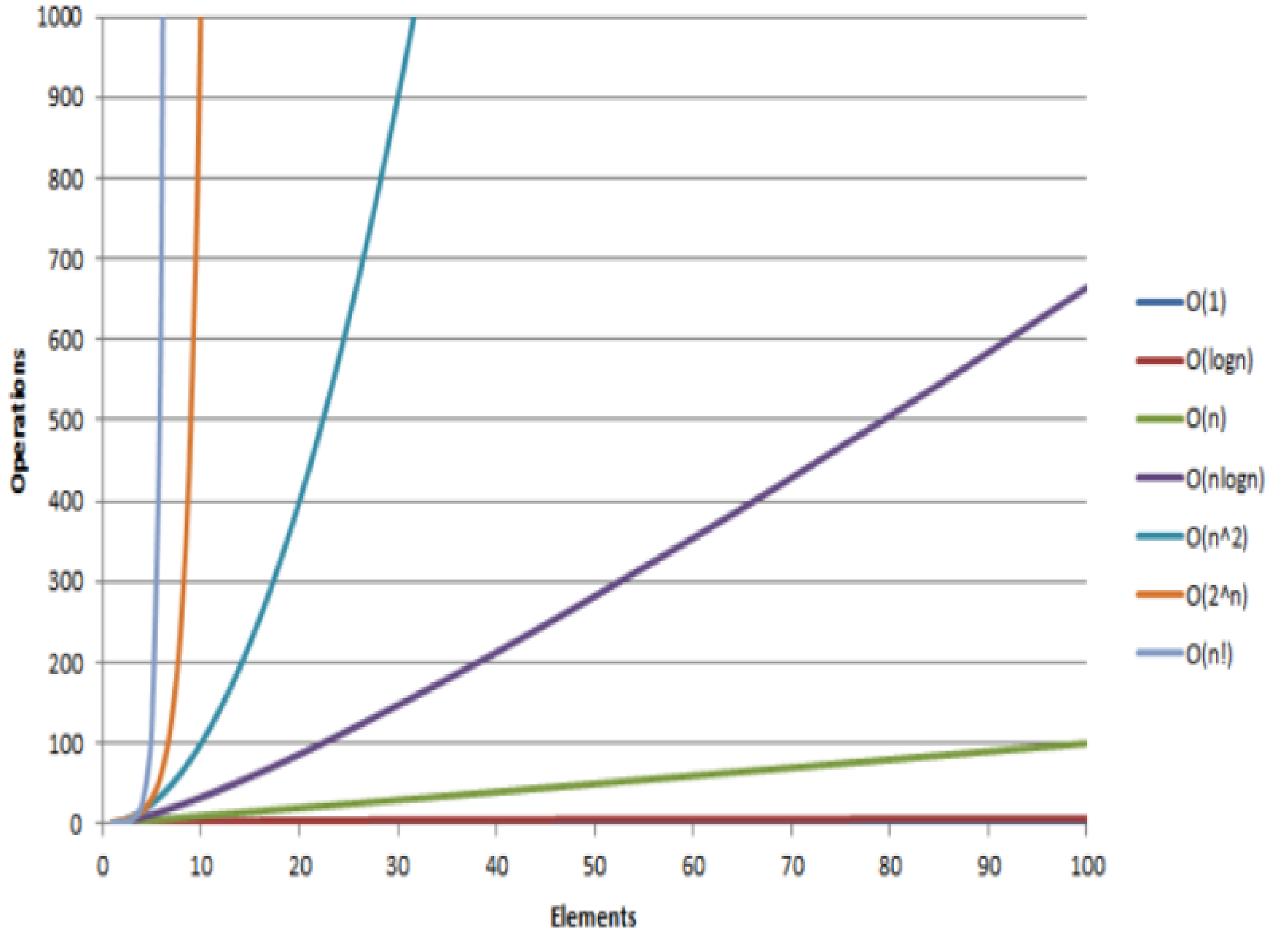
- $c_1\, g(n) <= f(n) <= c_2\, g(n)$ for every n >= n_0

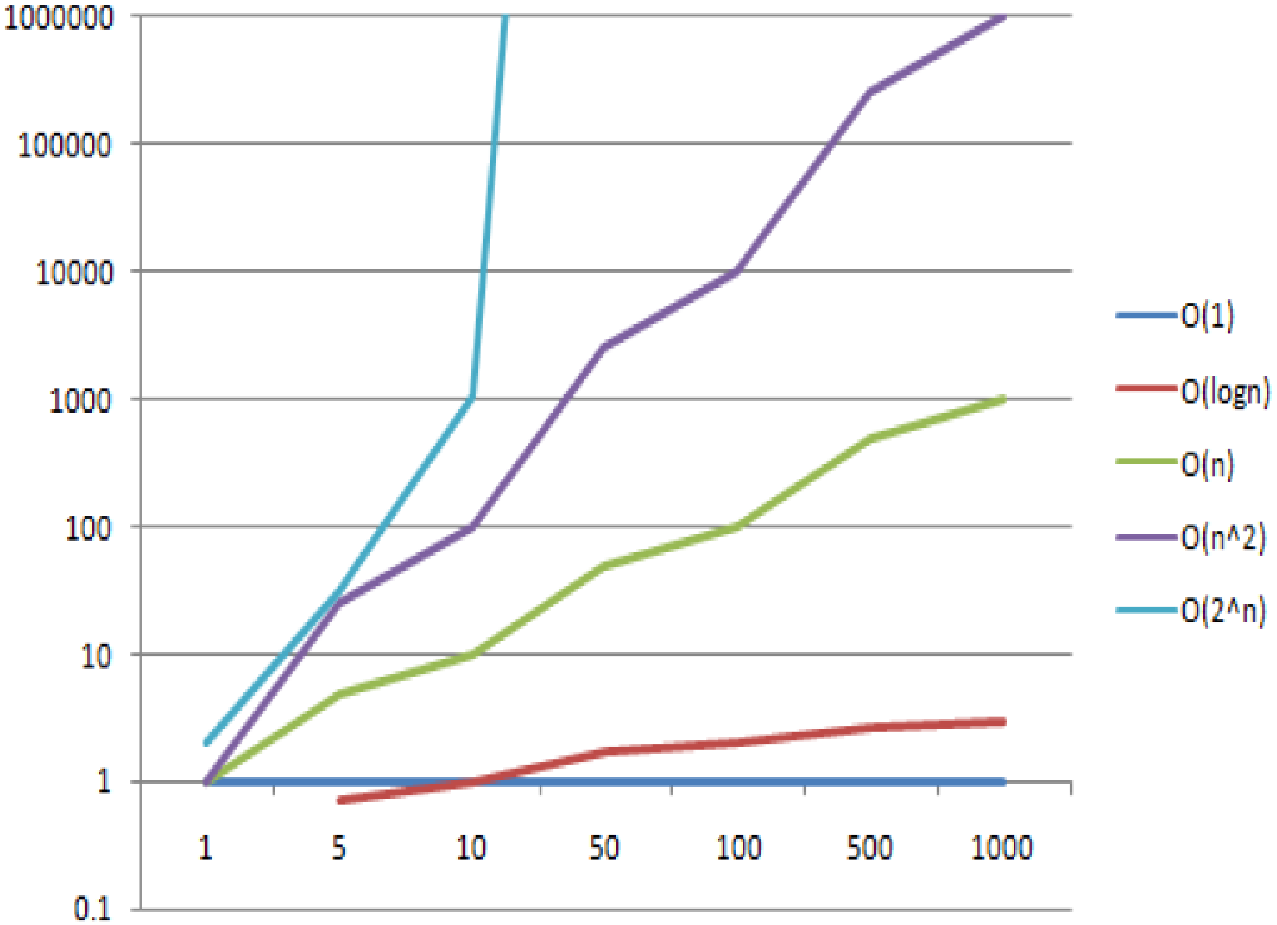
- O(g(n)): functions that grow no faster than g(n)
- Ω(g(n)): functions that grow at least as fast as g(n)
- Θ(g(n)): functions that grow at same rate as g(n)
- O(g(n)): functions no worse than g(n)
- Ω(g(n)): functions at least as bad as g(n)
- Θ(g(n)): functions as efficient as g(n)

O vs Θ

- O is an upper bound on performance
- Θ is a tight bound
    - It is the upper and lower bound




1000
900
800
700
600
500
400
300
200
100
0
Operations
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)


1000000
100000
10000
1000
100
10
1
0.1
1
5
10
50
100
500
1000
O(1)
O(logn)
O(n)
O(n^2)
O(2^n)

Examples

- Average of $O(n^2)$ -> algorithms grows at most as fast as n^2 with average case input
  - E.g.: Bubble sort
- Worst case of $O(n^3)$ -> algorithm grows at most as fast as n^3 with its worst case
  - E.g.: brute force matrix multiplication, which is also $\Theta(n^3)$
- Best case of $\Theta(n)$ -> algorithm grows linearly in the best case
  - E.g.: sum an array
- Worst case of $\Omega(2^n)$ -> algorithm grows at best exponentially in worst case
  - E.g.: Create the power set
- With practice it often becomes easy for many algorithms

Test Your Understanding

In this section there are 8 question answer s.

1. True or false: $\Theta(n + \log n) = \Theta(n)$?

▶ View answer

2. True or false: $O(n + \log n) = O(n)$

▶ View answer

3. True or false: $\Theta(n \log_2 n) = \Theta(n \log_{10} n)$

▶ View answer

True or false: $\Theta(\log^2 n) = \Theta(\log n)$

▶ View answer

True or false: $O(n \log n) = O(n)$

▶ View answer

True or false: if $x \in O(n \log n)$ then $x \in O(n^2)$

▶ View answer

True or false: if $x \in \Theta(n \log n)$ then $x \in \Theta(n^2)$

▶ View answer

True or false: if $x \in O(n \log n)$ then $x \in O(n)$

▶ View answer

How would you prove that $\Theta(n + \log n) = \Theta(n)$

▶ View answer

Asymptotic Complexity classes

Class	Name	Description
1	constant	Best case
$\log n$	logarithmic	Divide then ignore part
n	linear	Examine each
$n \log n$	n-log-n or linearithmic	Divide then use all parts
n^2	quadratic	Doubly nested loop
n^3	cubic	Triply nested loop
2^n	exponential	All subsets
$n!$	factorial	All permutations

Calculating Algorithm Efficiency

- Identify its basic operations
    - Operations that are executed repeatedly at the core of the algorithm
    - E.g.: comparisons and swapping in sorting
    - Multiplications and additions in matrix multiplication
- Set up an equation which counts the number of basic operations for a given input of size n

Example

```
int MysteryFunction(A[0.. n -1])
    MysteryVal = A[0]
    For i = 1 to n - 1:
        If A[i] > MysteryVal:
            MysteryVal = A[i]
    Return MysteryVal
```

What does this algorithm do?

▶ View Answer

This example's basic operation is to check the value at a position in the array vs the current maximum value (MysteryVal) and return the maximum value in the array. Or: `A[i] > MaxVal`

Let `c(n)` = number of times it has executed

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \theta(n)$$

Maths

Some maths to know

$$\sum_{i=1}^{u} a_i \pm b_i = \sum_{i=1}^{u} a_i \pm \sum_{i=1}^{u} b_i$$

$$\sum_{i=1}^{u} ca_i = c \sum_{i=1}^{u} a_i$$

$$\sum_{i=j}^{n} 1 = n - j + 1$$

:point_up_2: This means that if you add 1 *n* times, you will simply get n

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = n\,(n+1)/2 \approx n^2 \in \theta(n^2)$$

:point_up_2: This means that `1+2+3+4+5+...+n-1+n = n(n+1)/2`

Set Example

Think up algorithms to determine if there are duplicate values in array A. Find out if it is a set, the time complexities of the solution and the most efficient solution.

Brute Force Solution

```
boolean isSet(A){
    for (i = 0; i < A.length - 1;i++){
        for(j = i+1; j < A.length;j++){
            if(A[i]==A[j]){
                return false;
            }
        }
    }
    return true;
}
```

The above solution can be broken down into the following format `(n-1 + n-2 + n-3 + ... + 1) = n(n-1)/2`. This is because the inner loop is executed n-1 times, and the outer loop is executed n times.

Therefore, the worst case is $\Theta(n^2)$, and the average case is that too.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Using S1 on the second summation and simplifying:

$$\sum_{i=0}^{n-2} (n-1-i)$$

Using R2, this simplifies to:

$$\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

Using R1 on the first summation and S2 on the second, this simplifies to:

$$(n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

Using S1 on the first summation and simplifying:

$$(n-1)^2 - \frac{(n-2)(n-1)}{2} = (n-1)n/2 \approx \frac{1}{2}n^2 \in \theta(n^2)$$

Decrease & Conquer

```
boolean isSet(Array A, int index = 0){
    if(index >= A.length-1){
        return true;
    }

    for(i = index+1; i < A.length;i++){
        if(A[index]==A[i]){
            return false;
        }
    }
    return isSet(A,index+1);
}
```

This algorithm is identical to the brute force one. Same complexities as above.

Transform & Conquer

```
boolean isSet(A){
    sort(A);
    for(int i = 0; i < A.length-1;i++){
        if(A[i]==A[i+1]){
            return false;
```

```
            return false;
        }
    }
    return true;
}
```

The most efficient sorting algorithms have a complexity of Θ(nlogn). The for loop is linear for the worst case: Θ(n)

It runs the sort, then the for loop so this is Θ(nlogn+n), the worst case order of an algorithm is the efficiency of its worst part. So the algorithm is Θ(nlogn) which is faster than the brute force solution of $\Theta(n^2)$.

The transform and conquer method of sorting then searching to test for a set is faster than the brute force method.

I have added the python code for this here

Analysing Recursive Factorial

```
int F(n){
    if n == 0:
        return 1;
    return F(n-1)*n;
}
```

- We want to count the multiplication
- Multiplies once for every recursive call
- Code can be found here

```
M(0)=0
And M(n)=M(n-1)+1
M(n) = M(n-1) + 1 = [M(n-2) + 1]+1
= [M(n-3 + 1) + 2]= M(n-3)+3

Thus, in general:
M(n) = M(n-k)+k
M(n) = M(n-n) + n = M(0) + n = n
```

Backwards Substitution:

1. Express x(n-1) successively as a function of x(n-2), x(n-3)
2. Derive x(n-j) as a function of j
3. Substitute n-j = base condition

The above equation can be solved by backward substitution:

```
M(n) = M(n-1)+1
```

```
Substitute M(n-1) = M(n-2) + 1
-> M(n) = [M(n-2) + 1]+1 = M(n-2) + 2
Substitute M(n-2) = M(n-3) + 1
    M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3
-> Pattern: M(n) = M(n-j) + j
Ultimately: M(n) = M(n-n)+n = M(0) + n = n
```

Recurrence Relations

- Recurrence relation is a recursive mathematical function e.g.:

```
M(n) = M(n-1)+1
```

- We will consider these recurrence relations

```
T(n) = aT(n-k) + f(n)
OR
T(n) = aT(n/b) + f(n)
```