Data Structure and Algorithm

Performance analysis

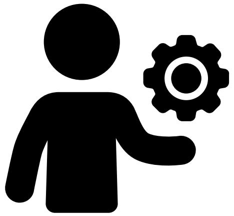
Q: What is an Algorithm?

: Algorithm :

sequence of **finite steps**
to solve a **particular problem**.

Characteristics of an Algorithm

- Clear and Unambiguous
- Well-Defined Inputs
- Well-Defined Outputs
- Finite
- Feasible (simple, generic, and practical : not contain future technology)
- Language Independent



Advantages of Algorithms

Easy to understand.

A step-wise representation of a solution.

the problem is broken down into smaller pieces

Easier to convert into program.

Disadvantages of Algorithms

Take a long time.

Difficult to explain complex logic.

Branching and Looping statements are difficult to show.

Types of Algorithms

- **1. Brute Force Algorithm:**
  - **Test every choices of answer**.
  - First approach when we see a problem.
- **2. Recursive Algorithm:**
  - A problem is **broken into several sub-parts and called the same function** again and again.

Types of Algorithms

- **3. Backtracking Algorithm:**
  - **Whenever a solution fails we trace back to the failure point and build on the next solution and continue this process till we find the solution**
  - In SudoKo solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Types of Algorithms

- **4. Searching Algorithm:**
  - **Searching** elements or groups of elements from a particular data structure.
- **5. Sorting Algorithm:**
  - Sorting is **arranging** a group of data in an increasing or decreasing manner.
- **6. Hashing Algorithm:**
  - Searching that **contain an index with a key ID** for specific data.
- **7. Divide and Conquer Algorithm:**
  - **Breaks a problem into sub-problems, solves a single sub-problem and merges the solutions together** to get the final solution.
  - It consists of the following three steps: Divide , Solve , Combine

Types of Algorithms

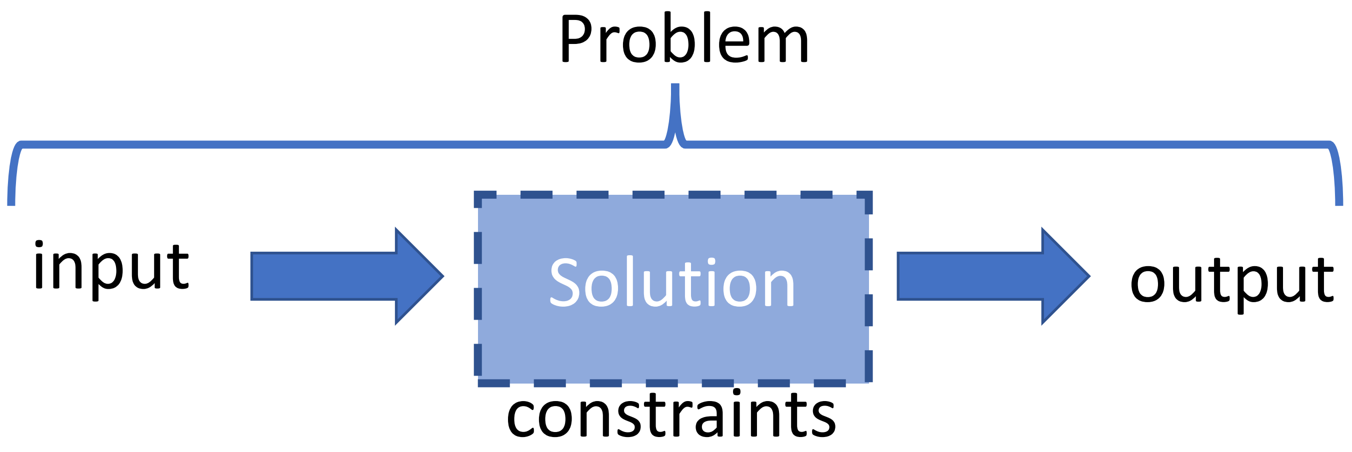
- **8. Greedy Algorithm:**
  - the solution is built part by part. The solution of the next part is built based on the immediate benefit of the next part.
  - **The one solution giving the most benefit will be chosen as the solution for the next part.**

- **9. Dynamic Programming Algorithm:**
  - **Use already found solution** to avoid repetitive calculation of the same part of the problem.
  - It **divides the problem into smaller overlapping subproblems** and solves them.
  - Ex : use array to keep factorial result and reuse it in next iteration

Types of Algorithms

- **10. Randomized Algorithm:**
  - **The random number helps in deciding the expected outcome.**

How to Design an Algorithm



1. Clear **problem** definition
2. Consider the **constraints** of the problem
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, is within the given **constraints**.

One problem, many solutions

Why to worry about performance?

- **Cost** of time and space in application
- Performance == **Scale**
- Example :
  - Service use 1 second to finish job at the first deploy. Next year, it use 10 minutes.
  - Text editor use 1 second to spell check each page but user use this program with 1000 pages file.
  - Data analytics cases : process 100,000 data unit
    - 20 seconds per 1 data unit -> 23 days
    - 5 data unit per seconds -> 5 hours

Better performance

Better (applications and programmers) life

Algorithm complexity

Time Factor

amount of time that is required by the algorithm to execute and get the result.

counting the number of key operations

Space Factor

the amount of memory used by the algorithm to store the variables and get the result.

counting the maximum memory space required

How to analyze an Algorithm

1.Priori Analysis:

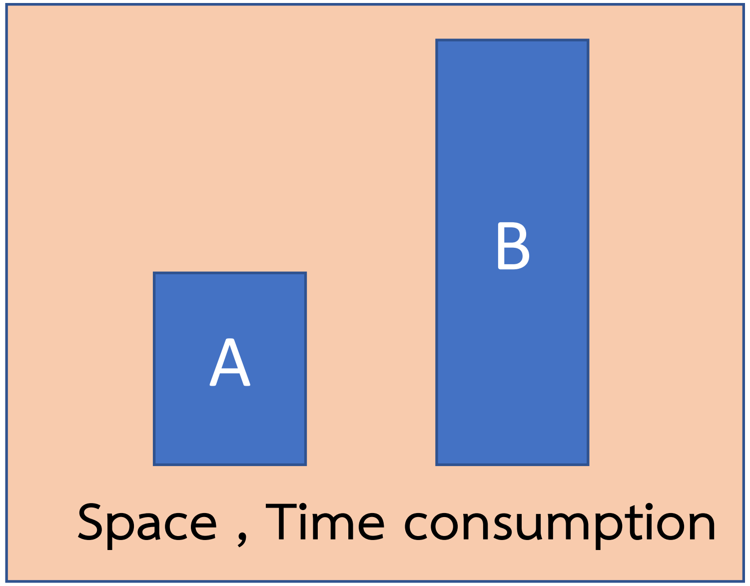
- checking before its implementation (“Priori” = “before”)
- Assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- approximate answers for the complexity of the program.
- Independent with language , compiler and hardware

2.Posterior Analysis:

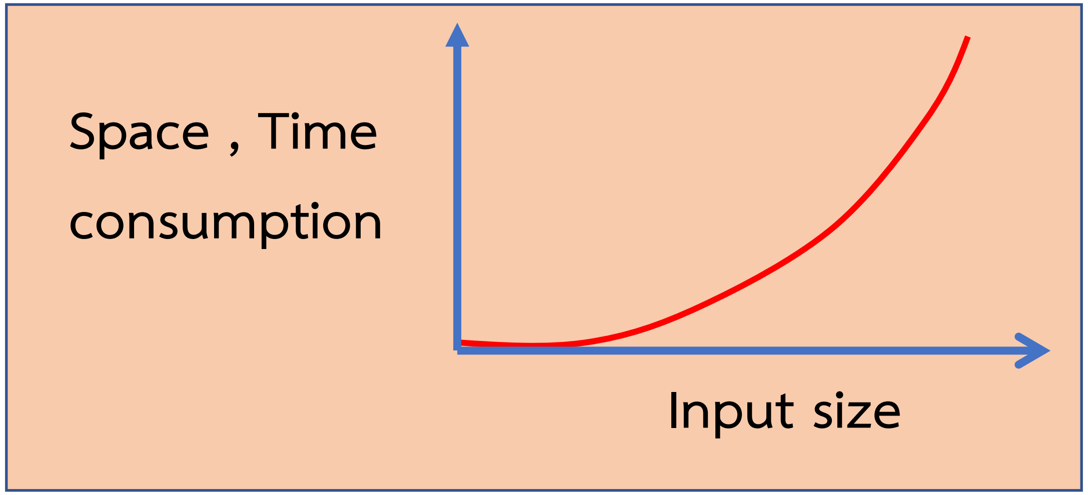
- checking the algorithm after its implementation (“Posterior” = “after”)
- checked by implementing and executing it.
- get the actual analysis about correctness, space required, time consumed etc.
- Depend on the language of the compiler and the type of hardware used.

Application (or algorithm) performance

1. Compare Space , Time consumption



2. Relation of Space , Time consumption with input size



Measure the Execution Time

```
import time

# get the start time
st = time.time()

# main program
# find sum to first 1 million numbers
sum_x = 0
for i in range(1000000):
    sum_x += i
```

```
# wait for 3 seconds
time.sleep(3)
print('Sum of first 1 million numbers is:', sum_x)

# get the end time
et = time.time()

# get the execution time
elapsed_time = et - st
print('Execution time:', elapsed_time, 'seconds')
```

```python
import time

s1=[]
s2 = {}
for i in range(0,19999999):
    s1 += [i]
    s2[i]=0

# algorithm 1 -> search in list
st = time.time()
if 19999999 in s1:
    print('found')
time.sleep(1)
en = time.time()
elapsed_time = en - st
print('Execution time (search in list) :',
elapsed_time, 'seconds’)

# Algorithm 2 -> search in dict
st = time.time()
if 19999999 in s2:
    print('found')
time.sleep(1)
en = time.time()
elapsed_time = en - st
print('Execution time (search in dict) :',
elapsed_time, 'seconds')
```

```
Output :
Execution time (search in list) : 1.2972080707550049 seconds
Execution time (search in dict) : 1.000472068786621 seconds
```

2 algorithms , which one is better

- Basic way : run 2 algorithm , finding which one take less time….
- But…
  - 1) It might be possible that **for some inputs, first algorithm performs better than the second.** And for some inputs second performs better.
  - 2) It might also be possible that **for some inputs, first algorithm perform better on one machine and the second works better on other machine** for some other inputs.

We need algorithm analysis method that independent with unrelated attributes.

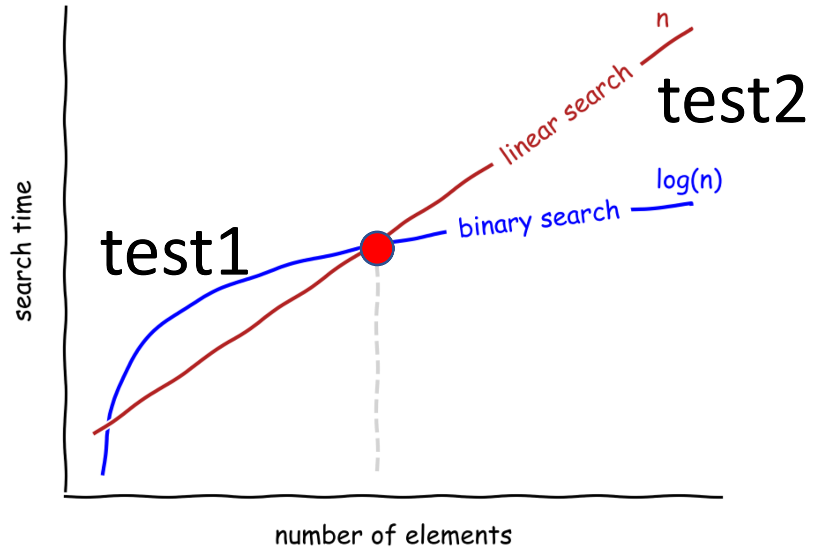
Asymptotic Analysis (easy to use , not perfect)

- we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, **how the time (or space) taken by an algorithm increases with the input size.**

Test 1 :

- small data set, Linear Search , Fast Computer A

- small data set, Binary Search , Slow Computer B



Test 2 :

- big data set, Linear Search , **Fast Computer A**

- Big data set, Binary Search , Slow Computer B

Ps. we can't judge which one is better for application , depand on maximum data size and cutting edge

Asymptotic Notations and Analysis

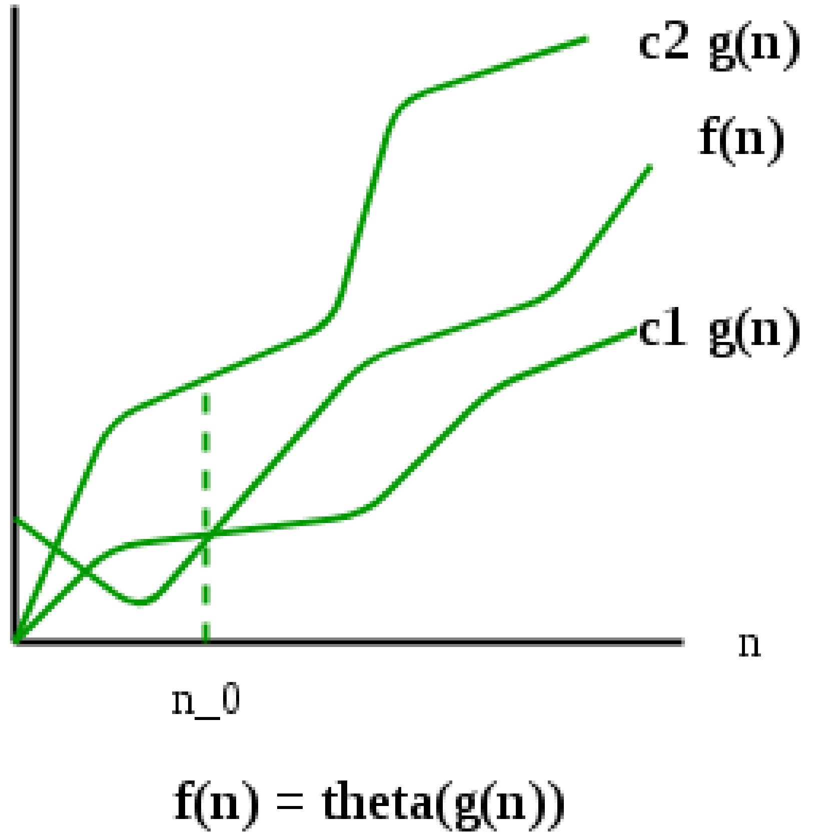
- a measure of the **efficiency** of algorithms
- **don't depend on** machine-specific constants
- **don't** require algorithms to be **implemented** and time taken by programs to be compared.
- **Asymptotic notations** are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

Asymptotic notations

- **3 asymptotic notations** are mostly used

1. Θ Notation (theta notation)
2. Big-O Notation
3. Ω Notation (omega notation)

- **1. Θ Notation: The theta notation** bounds a function from **above and below**, so it defines exact asymptotic behavior.



Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that

0 <= c1*g(n) <= f(n) <= c2*g(n)

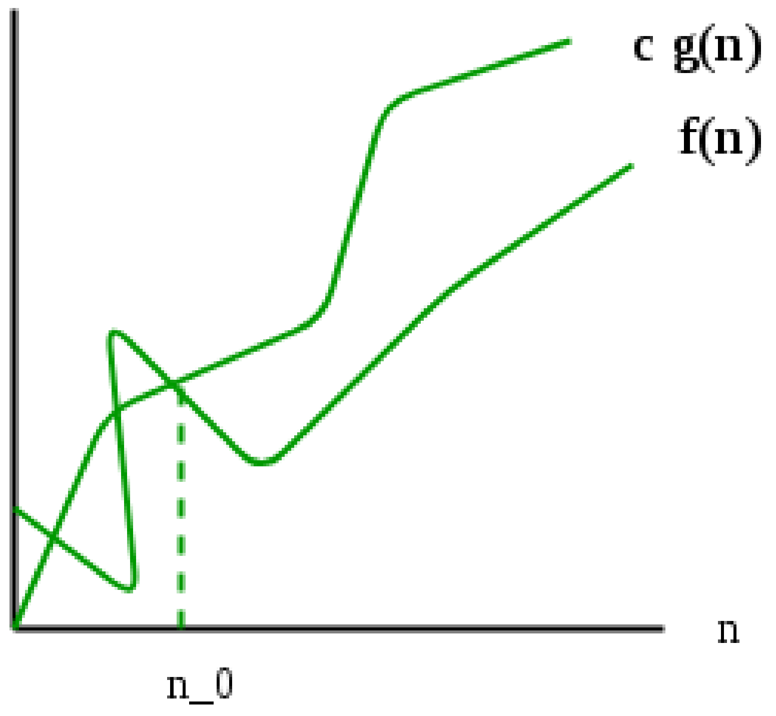
for all n >= n0}

- A simple way to get the Theta notation of an expression is to **drop low-order terms and ignore leading constants.** For example, consider the following expression.

$$\cancel{3}n^3 \cancel{+ 6n^2 + 6000} = \Theta(n^3)$$

- Examples :
  - { 100 , log (2000) , 10^4 } belongs to Θ(1)
  - { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to Θ(n)
  - { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to Θ(n^2)
- Θ provides exact bounds .

- 2) Big O Notation: defines **an upper bound** of an algorithm, it bounds a function only from above.



f(n) = O(g(n))

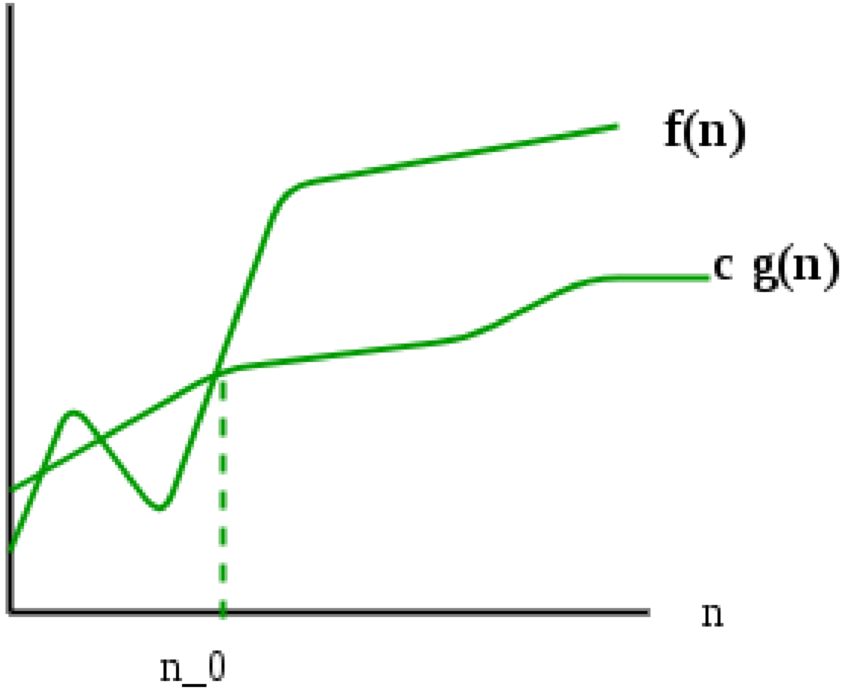
O(g(n)) = { f(n): there exist positive constants c and n0 such that

$0 <= f(n) <= c*g(n)$

for all n >= n0}

- Examples :
  - { 100 , log (2000) , 10^4 } belongs to O(1)
  - U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to O(n)
  - U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to O(n^2)
  - Here U represents union , we can write it in these manner because O provides exact or upper bounds .

- **3) Ω Notation:** provides an asymptotic **lower bound**.
- the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
- we are generally interested in worst-case and sometimes in the average case.



f(n) = Omega(g(n))

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that

0 <= c*g(n) <= f(n)

for all n >= n0}.

Examples :

- { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to Ω(n^2)
- U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to Ω(n)
- U { 100 , log (2000) , 10^4 } belongs to Ω(1)
- Here U represents union , we can write it in these manner because Ω provides exact or lower bounds .

Analysis of Algorithms example

O(1)

- Time complexity of a function (or set of statements) is considered as O(1) if it **doesn’t contain loop, recursion, and call to any other non-constant time function.**
- A loop or recursion that runs a **constant number of times** is also considered as O(1)

```
// Here c is a constant
  for (int i = 1; i <= c; i++) {
       // some O(1) expressions
  }
```

O(n)

- Time Complexity of a loop is considered as O(n) **if the loop variables are incremented/decremented by a constant amount**

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

$O(n^c)$

- Time complexity of **nested loops** is equal to **the number of times the innermost statement is executed.**
- For example, Selection sort and Insertion Sort have O(n^2) time complexity.

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
      // some O(1) expressions
    }
}

for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
      // some O(1) expressions
}
```

O(Logn)

- Time Complexity of a loop is considered as O(Logn) if the **loop variables are divided/multiplied by a constant amount.** And also for recursive call in recursive function the Time Complexity is considered as O(Logn).
- For example, Binary Search(refer iterative implementation) has O(Logn) time complexity.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

```
void recurse(n)
{
   if(n==0)
      return;
   else{
      // some O(1) expressions
   }
   recurse(n-1);
}
```

O(LogLogn)

- Time Complexity of a loop is considered as O(LogLogn) if the loop variables are **reduced/increased exponentially by a constant amount**.

```
// Here c is a constant greater than 1
  for (int i = 2; i <=n; i = pow(i, c)) {
     // some O(1) expressions
  }
  //Here fun is sqrt or cuberoot or any
other constant root
  for (int i = n; i > 1; i = fun(i)) {
     // some O(1) expressions
  }
```

complexities of consecutive loops

- When there are **consecutive loops**, we **calculate time complexity as a sum of time complexities of individual loops**.

```
for (int i = 1; i <=m; i += c) {
    // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
    // some O(1) expressions
}
```

Time complexity of above code is **O(m) + O(n)** which is **O(m+n)**. If m == n, the time complexity becomes O(2*n) which is O(n).

time complexity when there are many if, else statements inside loops

- **consider the worst case**.
- We evaluate the situation when values in if-else conditions cause a **maximum number of statements to be executed**.
- When the code is **too complex** to consider all if-else cases, we can get an upper bound by **ignoring if-else and other complex control statements.**

Worst, Average and Best Cases

- Worst Case :
  - Find **maximum number** of operations to be executed
  - For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.
  - Therefore, the worst-case time complexity of linear search would be $\Theta(n)$.
  - Big Theta = bounded both above and below asymptotically

Worst, Average and Best Cases

- Average Case :
  - take all possible inputs and calculate computing time for all of the inputs
  - **Sum all the calculated values and divide the sum by the total number of inputs**
  - **We must know (or predict) the distribution of cases.**
  - For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

$$\sum_{i=1}^{n} \frac{\Theta(i)}{(n+1)} = \frac{\Theta((n+1) * (n+2)/2)}{(n+1)} = \Theta(n)$$

Worst, Average and Best Cases

- Best Case :
  - we calculate the lower bound on the running time of an algorithm. **We must know the case that causes a minimum number of operations to be executed.**
  - In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$

- **Most of the times, we do worst-case analysis to analyze algorithms.** In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

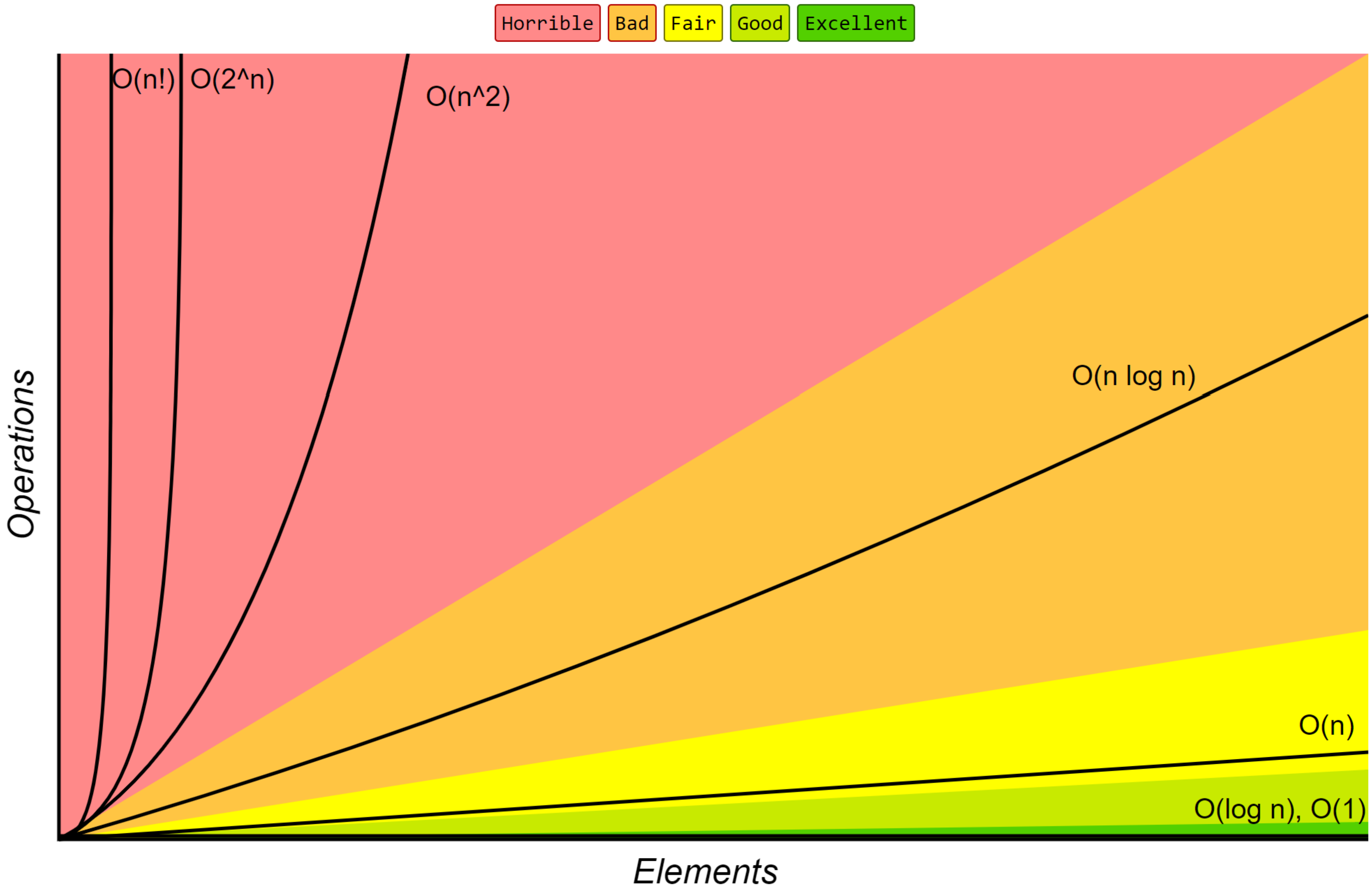
- **The average case analysis is not easy to do in most practical cases and it is rarely done.** In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

- **The Best Case analysis is bogus**. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Activity : Time complexity of following Code

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)
    # optimize code, so if the array is already sorted, it
doesn't need
    # to go through the entire process
    swapped = False
    # Traverse through all array elements
    for i in range(n-1):
        # range(n) also work but outer loop will
        # repeat one time more than needed.
        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            # if we haven't needed to make a single swap, we
            # can just exit the main loop.
            return


# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print("Sorted array is:")
for i in range(len(arr)):
    print("% d" % arr[i], end=" ")
```

Big-O Complexity Chart



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

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

Space Complexity

- Space Complexity of an algorithm is the **total space taken by the algorithm** with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

- Auxiliary Space is the extra space or temporary space used by an algorithm.

- **Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n*n, this will require O(n^2) space.**

In recursive calls stack space also counts.

```
int add (int n){
    if (n <= 0){
        return 0;
    }
    return n + add (n-1);
}
```

- Here each call add a level to the stack :

1. add(4)
2. -> add(3)
3. -> add(2)
4. -> add(1)
5. -> add(0)

- **Each of these calls is added to call stack** and takes up actual memory.
- So it takes **O(n) space.**

However, just because you have n calls total doesn't mean it takes O(n) space.

```
int addSequence (int n){
    int sum = 0;
    for (int i = 0; i < n; i++){
        sum += pairSum(i, i+1);
    }
    return sum;
}
```

```
int pairSum(int x, int y){
    return x + y;
}
```

- There will be roughly **O(n) calls** to pairSum. However, those calls do not exist simultaneously on the call stack,
- so you only need **O(1) space**.