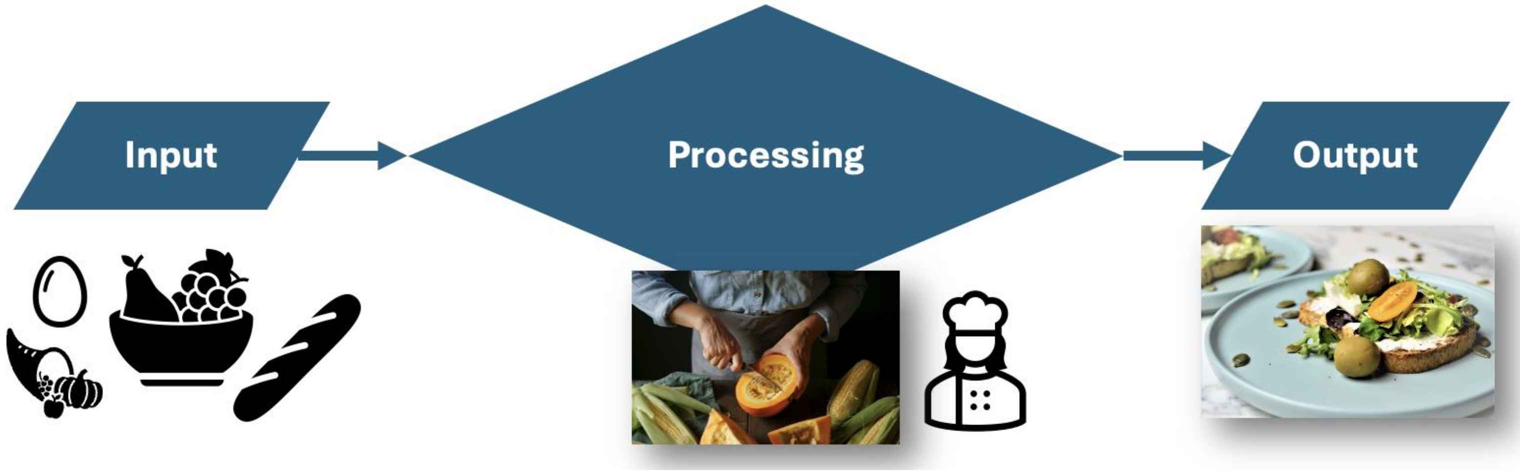
CDS2003: Data Structures and Object-Oriented Programming

Lecture: Algorithm Analysis

Review

- Essential elements of an algorithm
  - Input
  - Processing unit
  - Output
- Principles of algorithm design
  - Readability
  - Correctness
  - Robustness
  - Efficiency
- Efficiency
  - Time factor
  - Space factor
- Analysis after implementation



Algorithm analysis – After implementation

- Two programs for obtaining a Fibonacci number F_n
  - Recursion

```
# Algorithm 1
def get_fn_1(n):
    if n < 2:
        fn = n
    else:
        fn = get_fn_1(n-1) + get_fn_1(n-2)
    return fn
```

  - Iteration

```
# Algorithm 2
def get_fn_2(n):
    if n < 2:
        fn = n
    else:
        first = 0
        second = 1
    for _ in range(n-1):
        sum = first + second
        first = second
        second = sum
    fn = second
    return fn
```

Algorithm analysis – After implementation

- Two programs for obtaining a Fibonacci number F_n
  - Execution times at $n = 30$

```
import time
n = 30

start = time.time()
results = get_fn_1(n)
end = time.time()
print('The results of f_{} is {}.'.format(n, results))
print('The Recursion algorithm takes {} second to calculate!'.format(end - start))

start = time.time()
results = get_fn_2(n)
end = time.time()
print('The results of f_{} is {}.'.format(n, results))
print('The Iteration algorithm takes {} second to calculate!'.format(end - start))
```

Output:

```
The results of f_30 is 832040.
The Recursion algorithm takes 3.15065598487854 second to calculate!
The results of f_30 is 832040.
The Iteration algorithm takes 0.00010323524475097656 second to calculate!
```

Algorithm analysis – After implementation

- The execution time highly relies on
  - The hardware configuration
  - The programming language
  - The quality of code
  - Other environmental factors

- The execution time is useful for evaluating the empirical performance of an algorithm; however, can we characterize the resource requirements before implementing a program for the algorithm?

Algorithm analysis – Before implementation

- The computational complexity of an algorithm is a function describing the algorithm's efficiency in terms of the amount of (input) data.

- The computational complexity is calculated based on theoretical analysis.
- The resource to be consumed by carrying out an algorithm can be estimated from the computational complexity of the algorithm

- Time complexity $T(n)$
  - A function that describes the amount of computer time an algorithm takes to run in terms of the input size n

- Space complexity $S(n)$
  - A function that describes the amount of computer space (memory storage) an algorithm requires to run in terms of the input size n

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Algorithm analysis – Before implementation

- Time complexity $T(n)$
  - An algorithm consists of elementary operations.
  - The running time is estimated by

$$\sum time\ for\ executing\ an\ elementary\ operation\ \times\ number\ of\ the\ operation$$

- Space complexity $S(n)$
  - Input space: the memory used by the input of the algorithm.
  - Auxiliary space: any other memory the algorithm uses during execution, such as the extra space used for constants, variables, data structures, and function calls
  - The running space is estimated by

$$Input\ space + Auxiliary\ Space$$

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Examples

```python
def algorithm_1(n):
    a = 0
    b = 0
    a += n
    b -= n
    c = a * b
    return c
```

```python
def algorithm_2(n):
    a = 0
    b = 0
    if n < 1:
        a += n
    else:
        b -= n
    c = a * b
    return c
```

```python
def algorithm_3(n):
    a = 0
    for i in range(n):
        a += 1
    return a
```

Algorithm analysis – Asymptotic analysis

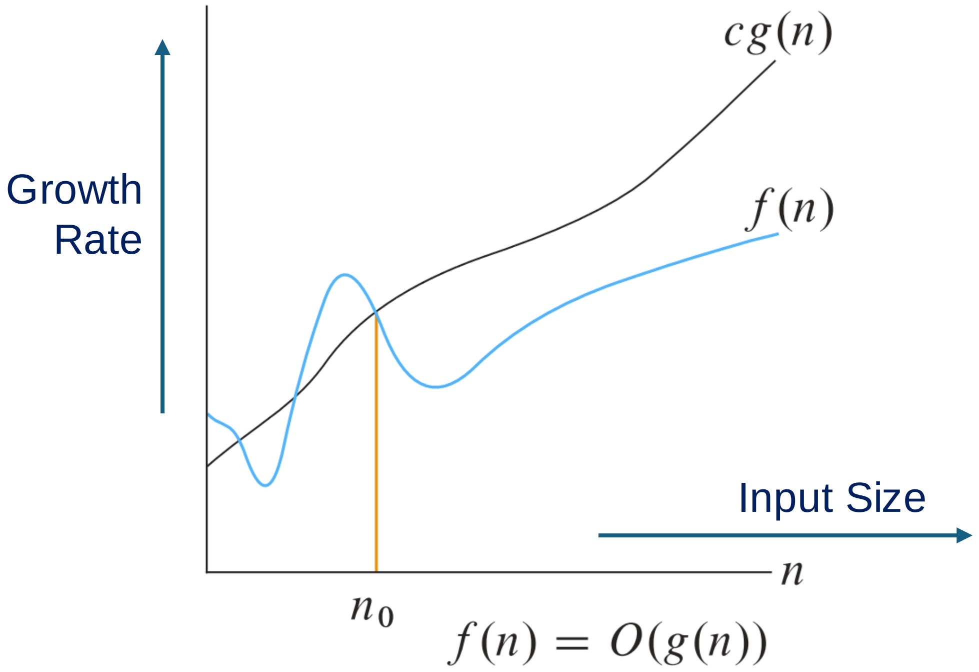
- The comparison of algorithm efficiency should be independent of any particular dataset or programming language

- The order of growth of an algorithm matters, instead of the exact value.
  - How quickly the resource requirement grows relative to the input size
  - $T(n)$ and $S(n)$ versus n

- Asymptotic analysis is not perfect, but effective for analyzing algorithms

Algorithm analysis – Big-Oh notation

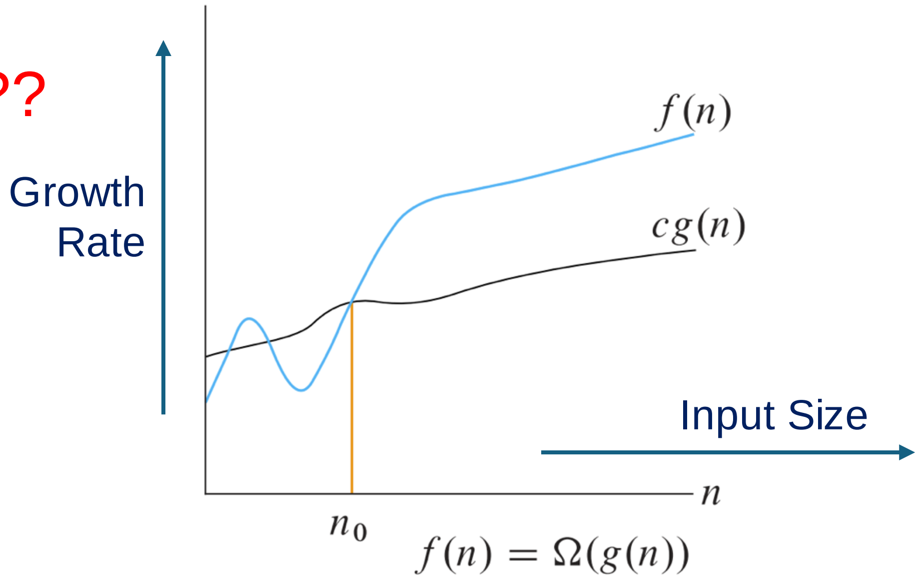
- The “Big-Oh notation” is commonly used for algorithm complexity.
  - $f(n) = O(g(n))$ if there exist positive constants c and n_0.
    such that $f(n) \leq cg(n)$ when $n \geq n_0$.
  - In other words, $cg(n)$ gives an upper bound for $f(n)$.
  - The function $f(n)$ growth is slower than $cg(n)$.
  - Example: $n^2 + n = O(n^2)$.
  - Example: $n^2 + n = O(n^3)$???
  - There are many upper bounds.
  - Which one is better?



[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Algorithm analysis – Big-Omega notation

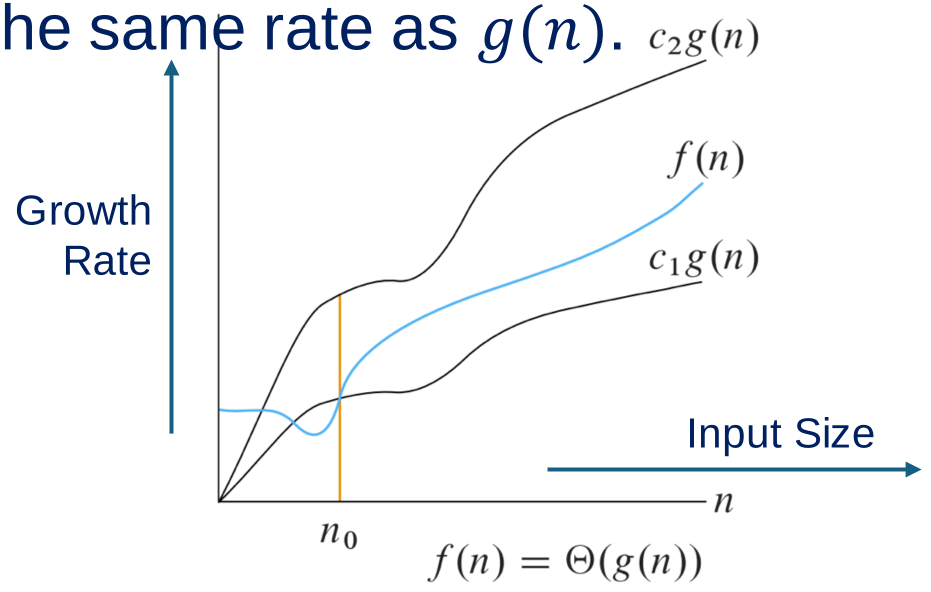
- The "Big-Omega notation"
  - $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0. such that $f(n) \geq cg(n)$ when $n \geq n_0$.
  - In other words, $cg(n)$ gives a lower bound for $f(n)$.
  - The function $f(n)$ growth is faster than $cg(n)$.
  - Example: $f(n) = c^n$ and $g(n) = n^c$ give $f(n) = \Omega(g(n))$.
  - Example: $f(n) = n^3 + 2n^2 = \Omega(n^3)$.
  - Example: $f(n) = n^3 + 2n^2 = \Omega(n^{2.5})$???
  - There are many lower bounds.
  - Which one is better?



[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Algorithm analysis – Big-Theta notation

- The "Big-Theta notation"
  - $f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2, and n_1. such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ when $n \geq n_1$.
  - In other words, $c_1 g(n)$ gives a lower bound for $f(n)$,
  - And $c_2 g(n)$ gives an upper bound for $f(n)$,
  - The function $g(n)$ is an asymptotically tight bound on $f(n)$.
  - In other words, the function $f(n)$ grows at the same rate as $g(n)$.
  - Example: $f(n) = n^3 + n^2 + n = \Theta(n^3)$.

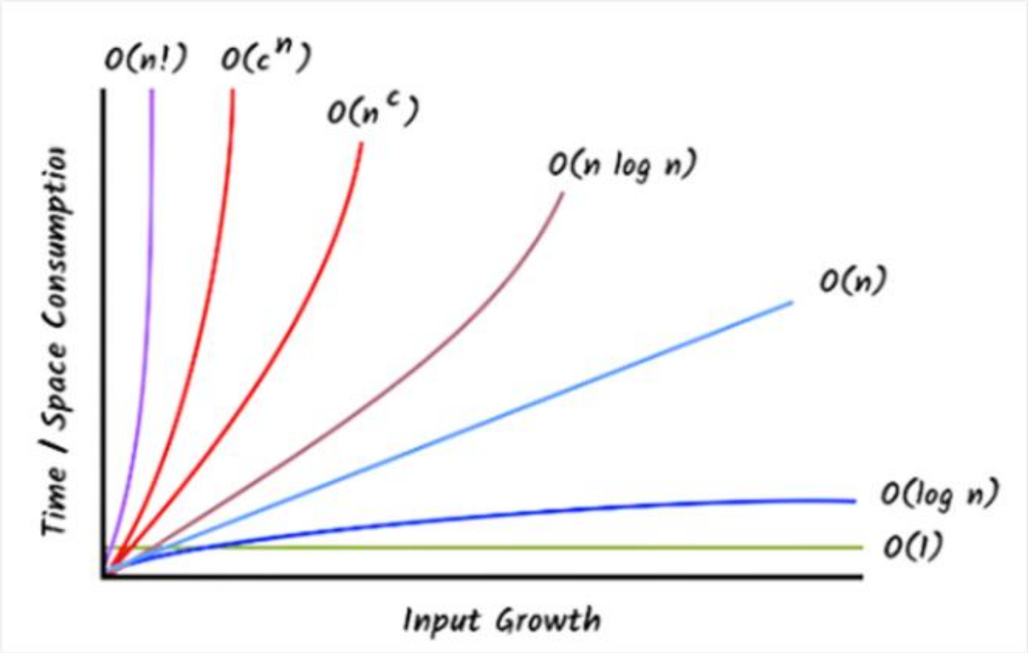


[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Main types of complexities

- Constant complexity $O(1)$
  - Independent of the input size n
- Logarithmic complexity $O(\log n)$
- Square root complexity $O(\sqrt{n})$
- Linear complexity $O(n)$
- N-LogN complexity $O(n \log n)$
- Quadratic complexity $O(n^2)$
- Polynomial complexity $O(n^c)$
- Exponential complexity $O(c^n)$
- Factorial complexity $O(n!)$ or $O(n^n)$

Note that $c > 1$ is a constant.



[1] https://www.scholarhat.com/tutorial/datastructures

Some useful formulas

- Addition in asymptotic notation: $f(n) + g(n) = O(\max(f(n), g(n)))$
- Multiplication in asymptotic notation: $f(n) * g(n) = O(f(n) * g(n))$
- Exponents:
  - $x^a x^b = x^{a+b}$; $(x^a)^b = x^{ab}$; $x^n + x^n = 2x^n \neq x^{2n}$; $2^n + 2^n = 2^{n+1}$;
- Logarithms: $a, b, c > 0$
  - $\log_a b = \frac{\log_c b}{\log_c a}, where\ a \neq 1$; $\log ab = \log a + \log b$; $\log(a^b) = b \log a$;
- Series
  - $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$; $\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}\ if\ 0 < a < 1$;
  - $\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{|k+1|}, where\ k \neq 1$; $\sum_{i=1}^{n} \frac{1}{i} \approx \log_e n$;

[1] Jain, H. (2016). Problem Solving in Data Structures & Algorithms Using Java: The Ultimate Guide to Programming. CreateSpace Independent Publishing Platform.

Constant time complexity $O(1)$

- The running time is independent of the input size n.
  - Each statement is assumed to take a constant amount of time to run.

- Examples
  - Assigning a value to a variable
  - Determining a number is odd or even
  - Printing out a phase like “Hello World”
  - Accessing n^{th} element of an array
  - A push or pop operation of a stack
  - …

```
a = 5
print(a % 2 == 1)
print("Hello World!")
b = [0, 2, 1]
x = b[1]
b.append(a)
print(a)
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Linear time complexity $O(n)$

- The running time is proportional to the input size
- When a function checks all values in an input data set or traverses all the nodes of a data structure, the complexity is no less than $O(n)$.
- Examples
  - Array operations like searching element, finding min, finding max, and so on
  - Linked list operations like traversal, finding min, finding max, and so on

```
def main(n):
  for i in range(n):
    print(i)
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Linear time complexity $O(n)$

- Examples

```
def main(n):
    for i in range(n):
        print(i)
```

```
def sum_n(inputs):
    result = 0
    for i in inputs:
        result += i
    return result
```

```
# factorial with Recursion
def factorial_Recur(n):
    if n == 0:
        return 1
    return n * factorial_Recur(n-1)
```

[1] https://www.simplilearn.com/tutorials/data-structure-tutorial

Logarithmic time complexity $O(\log n)$

- The running time is proportional to the logarithm of the input size.

- An example
  - 1, 2, 4, 8, 16, …, 2^k,…
  - $2^k \leq n \Rightarrow k \leq \log_2 n$

```
def log_print(n):
   i = 1
   while i <= n:
      print("Hello World !!!")
      i = 2 * i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

N-LogN time complexity $O(n \log n)$

- An example
  - Inner loop: $\log_2 n$ iterations
  - Outer loop : n iterations

```
def nlog_print(n):
    for j in range(n):
        i = 1
        while i <= n:
            print("Hello World !!!")
            i = 2 * i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

Double logarithmic time complexity $O(\log\log n)$

- An example
  - $j = 1, i = 3 \rightarrow 9 = 3^2$
  - $j = 2, i = 9 \rightarrow 81 = 3^4 = 3^{2^2}$
  - $j = 3, i = 81 \rightarrow 3^8 = 3^{2^3}$
  - $\dots\ j = k, i = 3 \rightarrow 3^{2^k} \dots$
  - $3^{2^k} \leq n \Rightarrow \log 2^k \leq \log n$
  - $\Rightarrow k \leq \log\log n$

```
def loglog_print(n):
    i = 3
    for j in range(2,n+1):
        if(i >= n):
            break
        print("Hello World !!!")
        i *= i
```

[1] https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/