SCHOOL OF INTERNATIONAL & INTERDISCIPLINARY ENGINEERING PROGRAMS
King Mongkut's Institute of Technology Ladkrabang

Algorithm Analysis

Dr. Sirasit Lochanachit

Algorithm Analysis

- If computers were infinitely fast, **any correct method** for solving a problem would do.
- Computing time and space in memory are a limited resource.
- Algorithms that are efficient in terms of **time or space** are preferred.

Algorithm Analysis

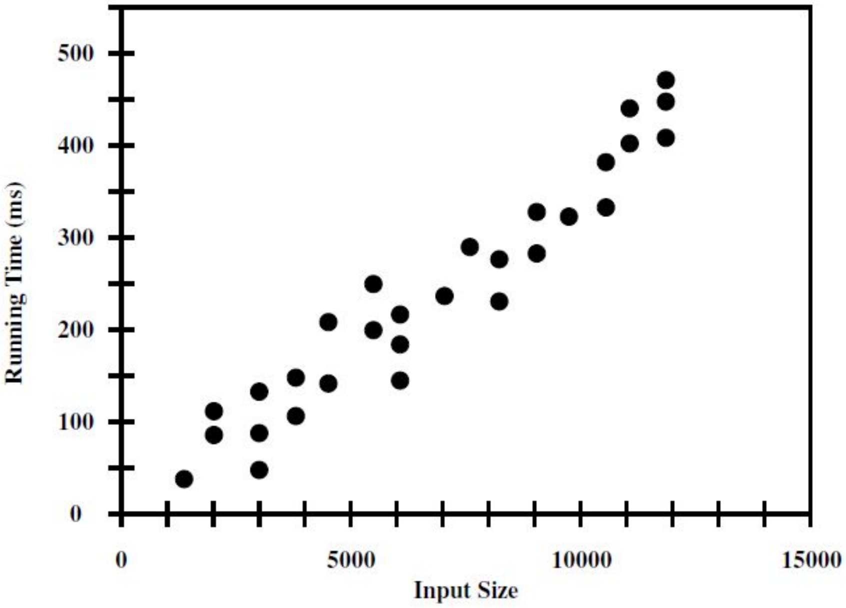
- How do we measure algorithm **efficiency** or **performance**?
    - Use running time as an indicator.

Running Time

- Example: Summation of *n* integers
  - Time required seems to increase as we increase the input size (*n*).
- Running time also depends on many factors
  - Hardware (CPU, RAM, etc.)
  - Software (OS, Programming language, etc.)

Experimental Studies

- Implement an algorithm then study its running time by
  - Executing with different test inputs of various sizes and
  - Recording time spent for each input size
  - Plot the results



Challenges of Experiments

- To directly compare between two different algorithms, the same hardware and software environments must be used.
- Limited set of test inputs.
- An algorithm must be fully implemented to study its running time.

High-level Analysis

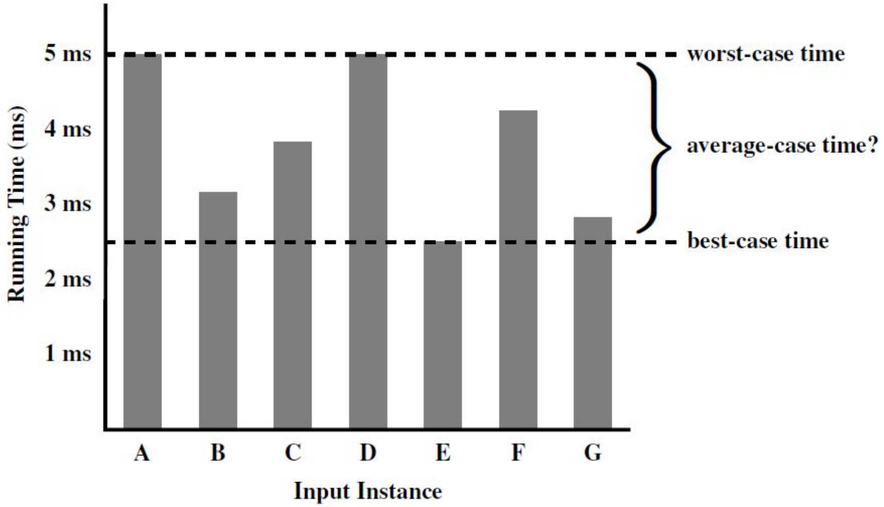
- Instead of implementing an algorithm and perform experiments, we can study a high-level description of the algorithm.
  - Either in the form of an actual code or pseudo-code.
- Takes into account all possible inputs
- Allows us to evaluate the efficiency of an algorithm independent of hardware & software environment.

Counting Primitive Operations

- Formally, a primitive operation corresponds to a low-level instruction with an execution time that is constant.
  - Assign a variable to an object
  - Determining the object associated with a variable
  - Performing an arithmetic operation
  - Comparing two numbers
  - Accessing a single element of a Python list by an index
  - Calling a function (excluding operations within the function)
  - Returning from a function

Measuring Operations as a Function of Input Size

- To capture the order of growth of an algorithm's running time
  - f(*n*) characterizes the running time as a function of the input size *n*.
- Input of same size
  - Best-case, average-case or worst-case analysis?



Seven Important Functions

- Seven fundamental functions in algorithm analysis:
  - Constant: $f(n) = c$
  - Logarithmic: $f(n) = \log_b n$
  - Linear: $f(n) = n$
  - N-Log-n: $f(n) = n \log n$
  - Quadratic: $f(n) = n^2$
  - Cubic: $f(n) = n^3$
  - Exponential: $f(n) = 2^n$

Asymptotic Analysis

- To see long-term / big picture trends of running time
  - Given an algorithm that takes *input size* ***n***, find a function **T(*n*)** that describes the *runtime* of the algorithm

Asymptotic Analysis

- *Input size* might be:
  - the *magnitude of the input value* (e.g., for numeric input)
  - the *number of items* in the input (e.g., as in a list)
- An algorithm may also be dependent on *more than one input*.

Algorithm Analysis

- Fundamentally, runtime is determined by the *primitive operations*
- Running time can be expressed as the number of operations or steps executed.
  - theSum = 0
  - for i in range(1, n+1):

    theSum = theSum + i

Asymptotic Notation

- Example:
  - $T(n) = 2n^2 + n + 1$
  - The running time of this algorithm grows as $\boldsymbol{n^2}$.
- Asymptotic notation represents algorithm's complexity.
  - <u>Ignores constant factors</u> and slower growing terms.
  - <u>Focus on the main components</u> that affect the growth.
  - Big-O notation

Big-O Notation

- Objectively describe the efficiency of code without the use of concrete units (seconds/bytes).
- Provide a big picture of how the time and space requirements scale w.r.t input size.
- Focus on worst-case scenario.

Big-O Notation

- Formally, $f(n) = O(g(n))$:
  - If there exist positive constants c and n_0
  - such that $0 < f(n) < c * g(n)$ for all $n >= n_0$
- $f(n)$ is big-O of $g(n)$
  - Intuitively means that g (multiplied by a constant factor) set an *upper bound* of f as n gets large - i.e., an *asymptotic bound*

Simplifying Big-O

- Product Rule
  - If the Big-O is the product of the multiple terms, **drop the constant terms**

$O(1024 * n) =$

$O(n / 10) =$

$O(7 * n * n) =$

$O(345) =$

Simplifying Big-O

- Sum Rule
  - If the Big-O is the sum of the multiple terms, **only keep the largest term, drop the rest.**

$O(100 + n) =$

$O(n^2 + n) =$

$O(n + 500 + n^3 + n^2) =$

Big-O Notation

- Example:
  - $T(n) = 1 + n$,

    then $T(n) =$
  - $T(n) = 5n^2 + 10n + 12$,

    then $T(n) =$

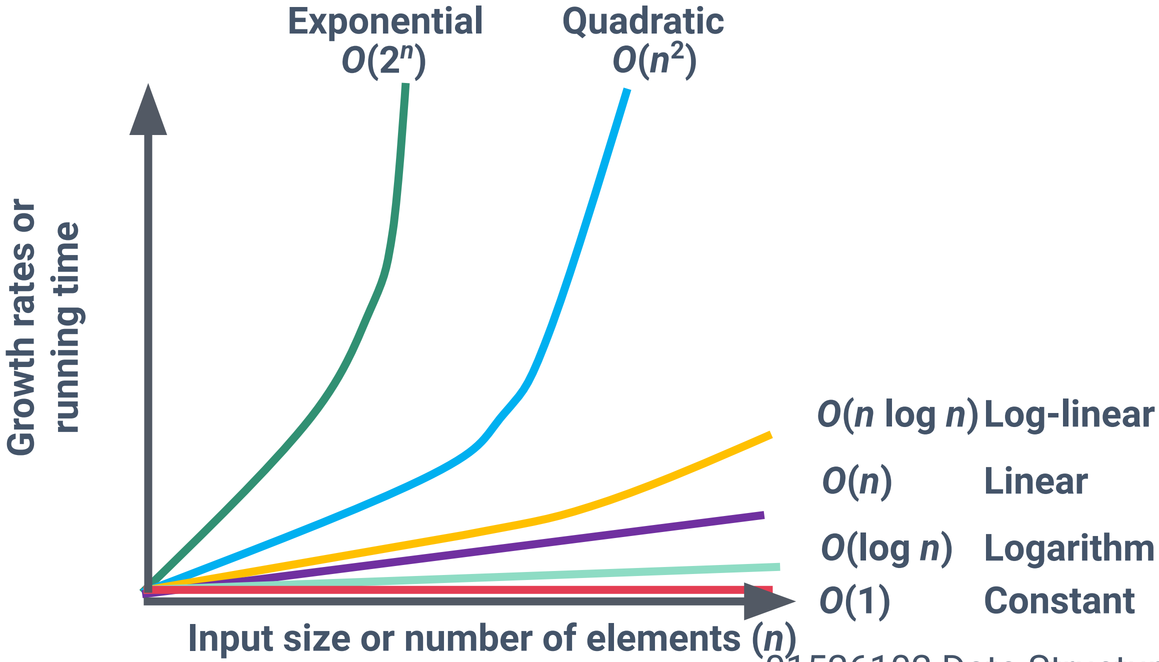
- **$O(n^2)$ means time complexity will never exceed n^2.**

Big-O Notation (Ordered)

- Common f(n):
  - O(1): constant
  - O(log n): logarithm
  - O(n): Linear
  - O(n log n): Log linear
  - O(n^2): Quadratic
  - O(2^n): Exponential
  - O(n!): Factorial

Time Complexity

In general, the standard functions of input size n are shown in figure.



Big-O Exercise

Time complexity T(n)

- $n^2 + 100n$
- $n^2 * n + 100n^2 \log n$
- $123 + \log 657$
- $(n + \log n)^3$
- $n(2 + \log n)$
- $(n/3)^6 + 10n$
- $1 + 2 + 3 + ... + n$

Big-O

Loops

	Python Code	Time complexity
Linear	`for i in range (0, n):` `    do something`	$T(n) = n, O(n)$
Linear	`for i in range (0, n, 2):` `    do something`	$T(n) = n/2, O(n)$

Nested	`for i in range (0, n):`
	`    for j in range (0, n):`
	`        do something`
		$T(n) = n^2, O(n^2)$

A log is a Repeated Division

Python Code

Time complexity

n=1000

Logarithmic

```
i = 1          Start
while i < n:   Stop
    do something
    i = i * 2  Step
```

```
1
2
4
8
16
32
64
128
256
512
```

$T(n) = \log n$, $O(\log n)$

Logarithmic

```
i = n           Stop
while i >= 1:   Start
    do something
    i = i // 2  Step
```

```
1000
500
250
125
62
31
15
7
3
1
```

$T(n) = \log n$, $O(\log n)$

SCHOOL OF INTERNATIONAL & INTERDISCIPLINARY ENGINEERING PROGRAMS
King Mongkut's Institute of Technology Ladkrabang

Linear Logarithmic Loops

Python Code

Time complexity

Linear logarithmic

```
for i in range (0,n):        n
    j = 1
    while j < n:             log n
        do something
        j = j*2
```

$T(n) = n \log n, O(n \log n)$

Quadratic Loops

Python Code

Time complexity

Dependent Nested

```
for i in range (0,n):  n
    for j in range(0, i+1):(n+1)/2
        do something
```

T(n) = $n(n+1)/2$, $O(n^2)$

Number of iterations of the inner loop depends on the outer loop

For the inner loop, the number of iterations is (n+1)/2

For example, n = 3, i = 0 then j = [0],
i = 1 then j = [0, 1],
i = 2 then j = [0, 1, 2]

Recall: Arithmetic series

e.g., 1+2+3+4+5 = 15

Sum can also be found by:

- adding first and last term (1+5=6)
- dividing by two (to find average) (6/2=3)
- multiplying by num of values (3×5=15)

i.e., $1 + 2 + \cdots + n = \sum_{t=1}^{n} t = \frac{n(n+1)}{2}$

and $1 + 2 + \cdots + (n-1) = \sum_{t=1}^{n-1} t = \frac{(n-1)n}{2}$

Exponential

Python Code

Time complexity

Double Recursive

```
def foo(n):
        if (n==1):
                Return True
        foo(n-1)
        foo(n-1)
```

$T(n) = 1 + 2^n, O(2^n)$

Factorial

Python Code

Time complexity

Loop with Recursive

```
def foo(n):
    if (n==1):
        Return True
    for i in range(n):
        foo(n-1)
```

$T(n) = n * (n-1) * (n-2) * \ldots * 2 * 1,$

$O(n!)$

Calculating Time Complexity

Python Code: Factorial

```
def factorial1(n):
  if n <= 1:
    return 1
  else:
    fact = 1
    for i in range(2,n+1):
      fact *= i
    return fact
```

Calculating Time Complexity

Python Code: Simple nested loops

```python
def simple(n):
  for i in range(n):
    for j in range(n):
      print("i: {0}, j: {1}".format(i,j))
```

Calculating Time Complexity

Python Code: Element uniqueness v1

```python
def unique1(s):
  for i in range(len(s)):
    for j in range(i+1, len(s)):
      if s[i] == s[j]:
        return False  # Found duplicate pair
  return True      # All elements are unique
```

Calculating Time Complexity

Python Code: Element uniqueness v2

```
def unique2(s):
  temp = sorted(s) # create a sorted copy of s
  for i in range(1, len(temp)):
    if temp[i-1] == temp[i]:
      return False    # Found duplicate pair
  return True         # All elements are unique
```

Calculating Time Complexity

Python Code: Prefix averages v1

```
def prefix_average1(s):
  n = len(s)
  a = [0] * n                  # create list of n zeros
  for i in range(n):
    total = 0                  # compute each element
    for j in range(i+1):
      total += s[j]
    a[i] = total / (i+1)       # record the average
  return a
```

Calculating Time Complexity

Python Code: Prefix averages v2

```
def prefix_average2(s):
  n = len(s)
  a = [0] * n                    # create list of n zeros
  total = 0
  for i in range(n):
    total += s[i]     # update total sum to include s[i]
    a[i] = total / (i+1) # compute average based on 
current sum
  return a
```

Algorithm Analysis

- How do we measure algorithm **efficiency** or **performance**?
  - Use running time as an indicator.
- Running time can be expressed as the number of operations or steps executed.
- Asymptotic notation represents algorithm’s complexity.
  - Big-O notation