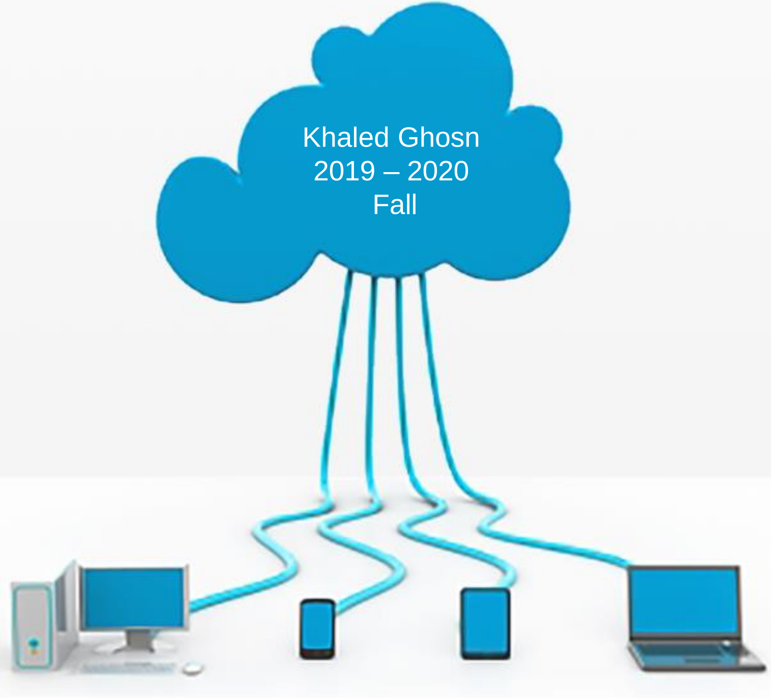
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

Algorithm Analysis

Experimental Studies
Order of Magnitude
Asymptotic Analysis



Khaled Ghosn
2019 – 2020
Fall

Principles

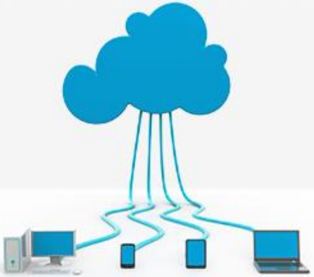
Algorithm

is a step-by-step procedure (sequence of instructions)

… to solve a specific problem

… in a finite amount of time

- The algorithm will be performed by a **processor**
- The algorithm must be expressed in **steps** that the processor is capable of performing
- The algorithm must eventually **terminate**
- The stated problem must be **solvable**, i.e., capable of solution by a step-by-step procedure



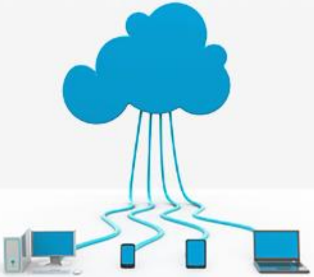
Principles

Algorithm

The word is derived from the phonetic pronunciation of the last name of

Abu Ja'far Mohammed ibn Musa ***al-Khowarizmi***

who was an Arabic mathematician who invented a set of rules for performing the four basic arithmetic operations - addition, subtraction, multiplication and division on decimal numbers



Principles

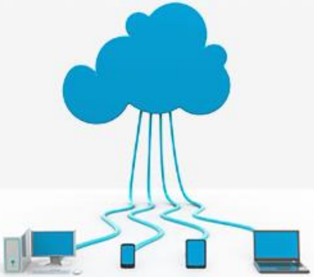
Efficiency

- Given several algorithms to solve the same problem, which algorithm is “best”?

- Given an algorithm, is it **feasible** to use it at all? Is it efficient enough to be usable in practice?

- How much **time** does the algorithm require?

- How much **space** (memory) does the algorithm require?



Principles

Efficiency

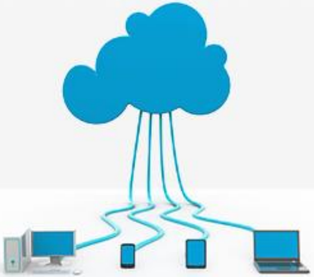
How to validate *correct* algorithms ?

(how do you measure how "GOOD" an algorithm is ?)

By measuring:

1. **Time**
   - Instructions take time.
   - How fast does the algorithm perform?
   - What affects its runtime?

2. **Space** (memory cost)
   - Data structures take space
   - What kind of data structures can be used?
   - How does choice of data structure affect the runtime?



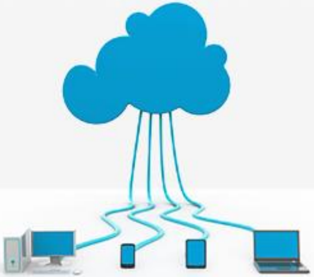
(could also be in quality, simplicity, compiler optimizations, OS, power consumption, bandwidth, cache …)

Algorithm Analysis

Algorithm Analysis is to determine the amount of resources that the algorithm will require

The performance of an algorithm is measured on the basis of following properties:

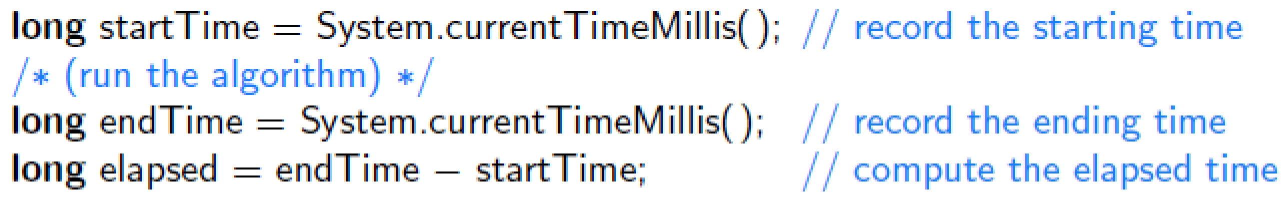
1. **Time Complexity**
2. **Space Complexity**



Algorithm Analysis

Experimental Studies

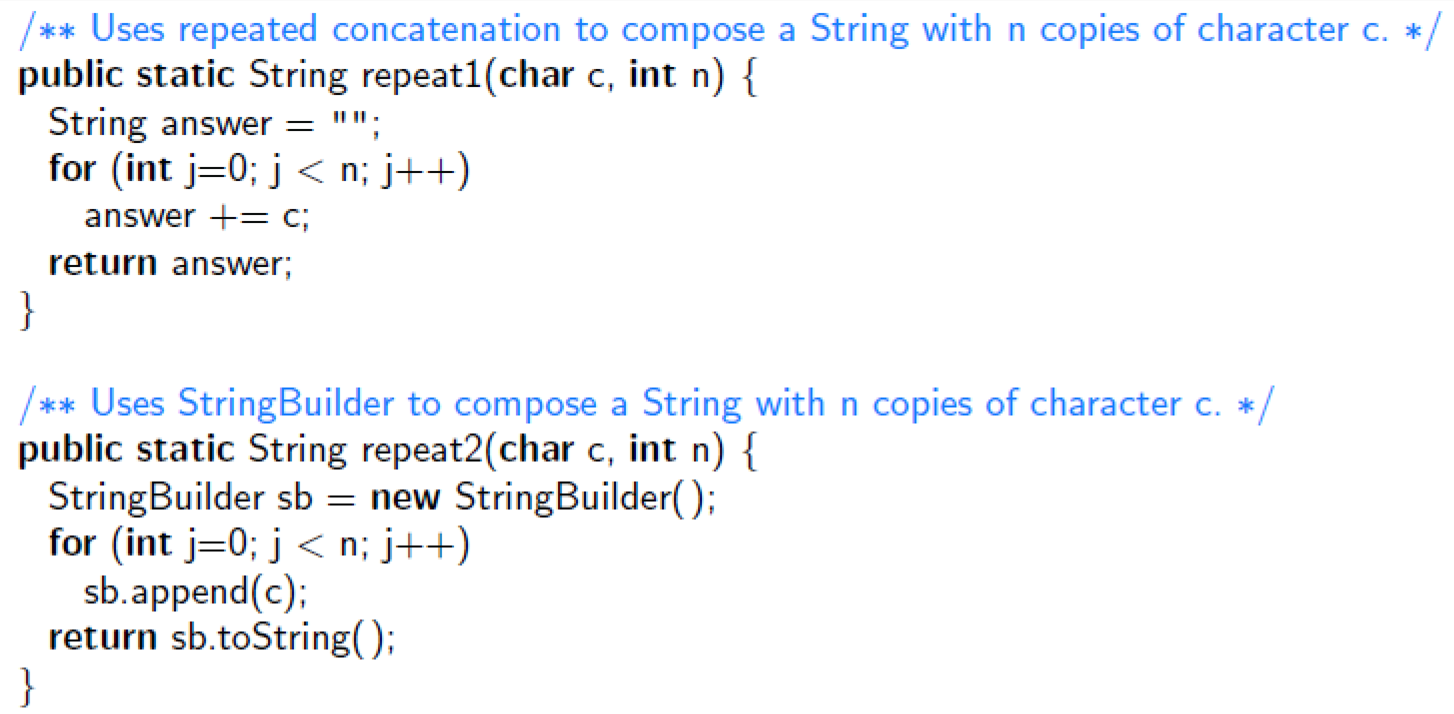
By using the following block of code, a comparison was done between two different algorithms to perform a repeated string concatenation



```
long startTime = System.currentTimeMillis( );  // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis( );    // record the ending time
long elapsed = endTime − startTime;            // compute the elapsed time
```

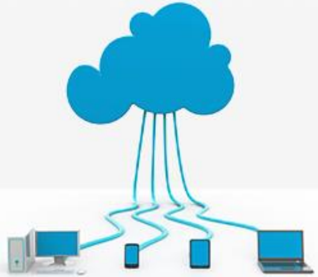
Algorithm Analysis

Experimental Studies



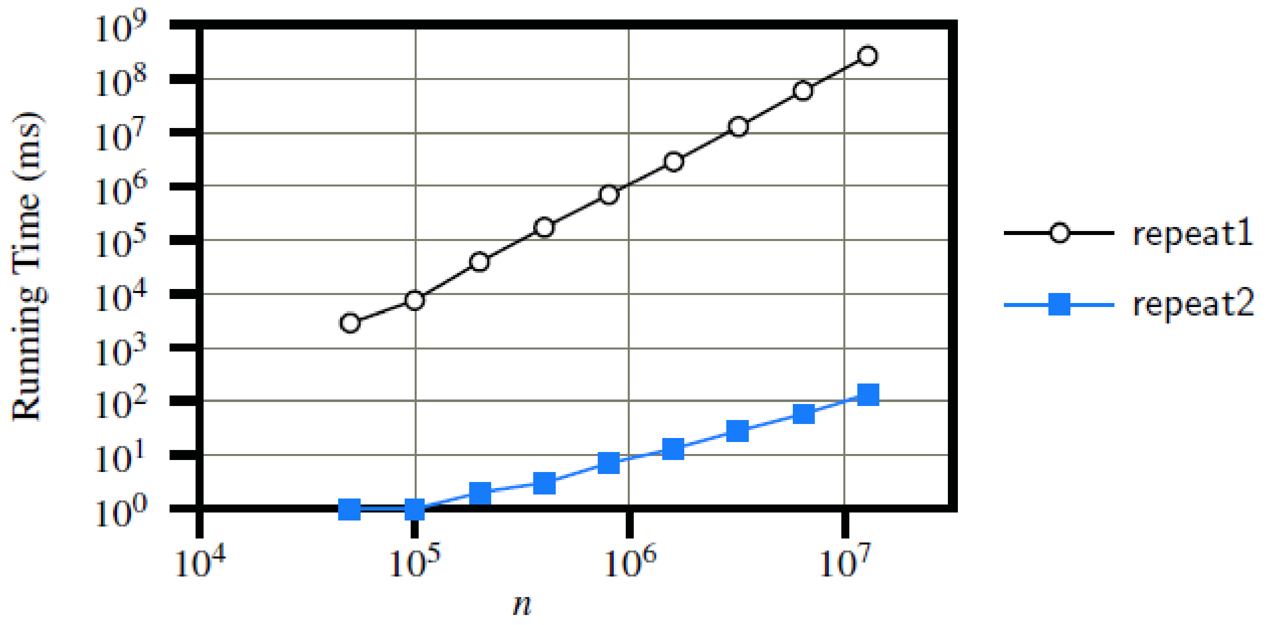
```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```



Algorithm Analysis

Experimental Studies



Algorithm Analysis

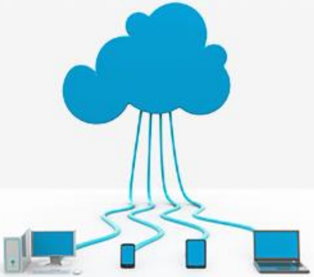
Challenges of Experimental Analysis

Three major limitations:

1. Experimental running times of two algorithms are difficult to directly compare (unless the experiments are performed in the same hardware and software environments).

2. Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment.

3. An algorithm must be fully implemented in order to execute it to study its running time experimentally.



Algorithm Analysis

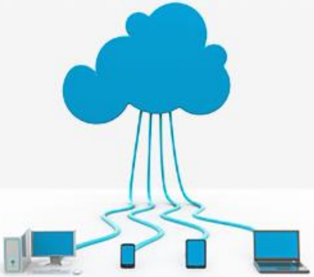
Moving Beyond Experimental Analysis

Analyzing the efficiency of algorithms:

1. Evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.

2. Studying a high-level description of the algorithm without need for implementation.

3. Takes into account all possible inputs.



Algorithm Analysis

How to compute time?

Measure time in seconds?

- + is useful in practice
- – depends on hardware (processor …), programming language (compiler …), execution context …

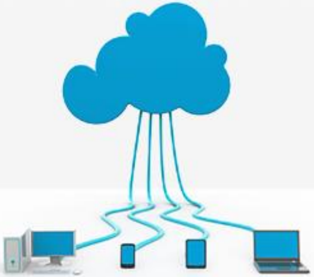
Count algorithm steps?

- + does not depend on compiler or processor
- – depends on granularity of steps

Count **characteristic** / **primitive** operations?

(e.g., arithmetic ops in math algorithms, comparisons in searching algorithms)

- + depends only on the algorithm itself
- + measures the algorithm’s intrinsic efficiency



Algorithm Analysis

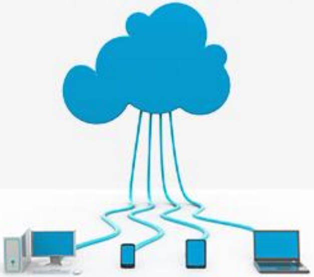
Execution time of Algorithms

Each instruction has a unique cost of 1

- 1 “time cycle”, let’s say

Although some instructions cost more:

- Addition / subtraction < Multi < Division
- CPU tasks < Memory access < Disk access



Algorithm Analysis

Execution time of Algorithms

Each instruction takes a certain amount of time

➔ a unique cost of 1 “time cycle”

`count + = 1;` ➔ take a certain amount of time, but it is constant

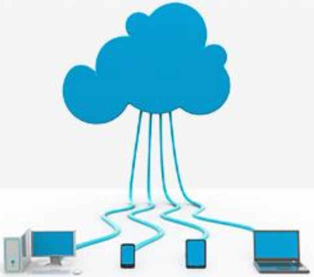
A sequence of operations:

```
count + = 1;
sum + = i;
```

Cost: c_1

Cost: c_2

Total Cost = $c_1 + c_2$



Execution time of Algorithms

IF - Else

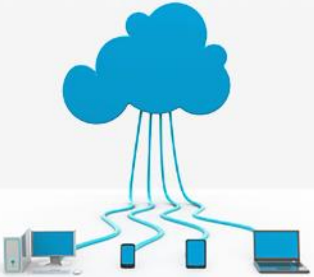
the runtime is:

the sum of the test +
the larger of the running times of If-block or Else-block

e.g.: *Simple If-Statement*

```
                                        Cost
if (n < 0)                              c1
   absval = -n                          c2
else
        absval = n;                     c3
```

Total Cost <= c1 + max (c2 , c3)



Execution time of Algorithms

Simple Loop

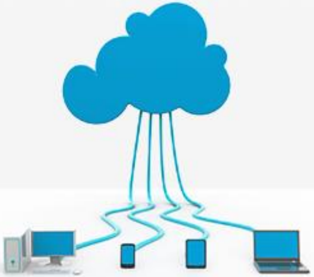
Loop accumulates its instructions costs

- Even if we may break earlier, but we always consider the worst case

```
                                   Cost        Times
i = 1;                              c1           1
sum = 0;                            c2           1
while (i <= n) {                    c3         n + 1
       i = i + 1;                   c4           n
       sum = sum + i;               c5           n
}
```

Total Cost = c1 + c2 + [(n + 1) * c3] + [n * c4] + [n * c5]

➔ The time required for this algorithm is proportional to **n**



Execution time of Algorithms

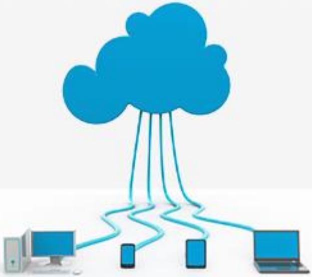
Nested Loop

Nested loops => Analyze inside out & Multiply

```
                                          Cost   Times
i=1;                                      c1       1
sum = 0;                                  c2       1
while (i <= n) {                          c3     n + 1
       j=1;                               c4     n
       while (j <= n) {                   c5     n * ( n + 1 )
            sum = sum + i;                c6     n * n
            j = j + 1;                    c7     n * n
   }
   i = i +1;                              c8     n
}
```

Total Cost = **c1 + c2 + [(n+1)*c3] + [n*c4] + [n*(n+1)*c5]+[n*n*c6]+[n*n*c7]+[n*c8]**

➔ The time required for this algorithm is proportional to $\mathbf{n^2}$



Order of Magnitude

Measuring Operations as a Function of Input Size

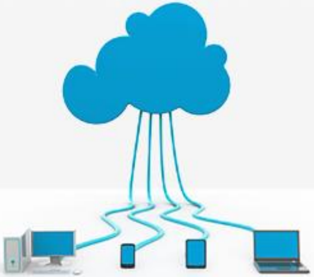
The amount of time and space required by an algorithm depend on the algorithm’s input (amount / size of the input)

- More data means that the program takes more time
- An array to search in has a size of 10 elements needs time less than that of size 1000 elements

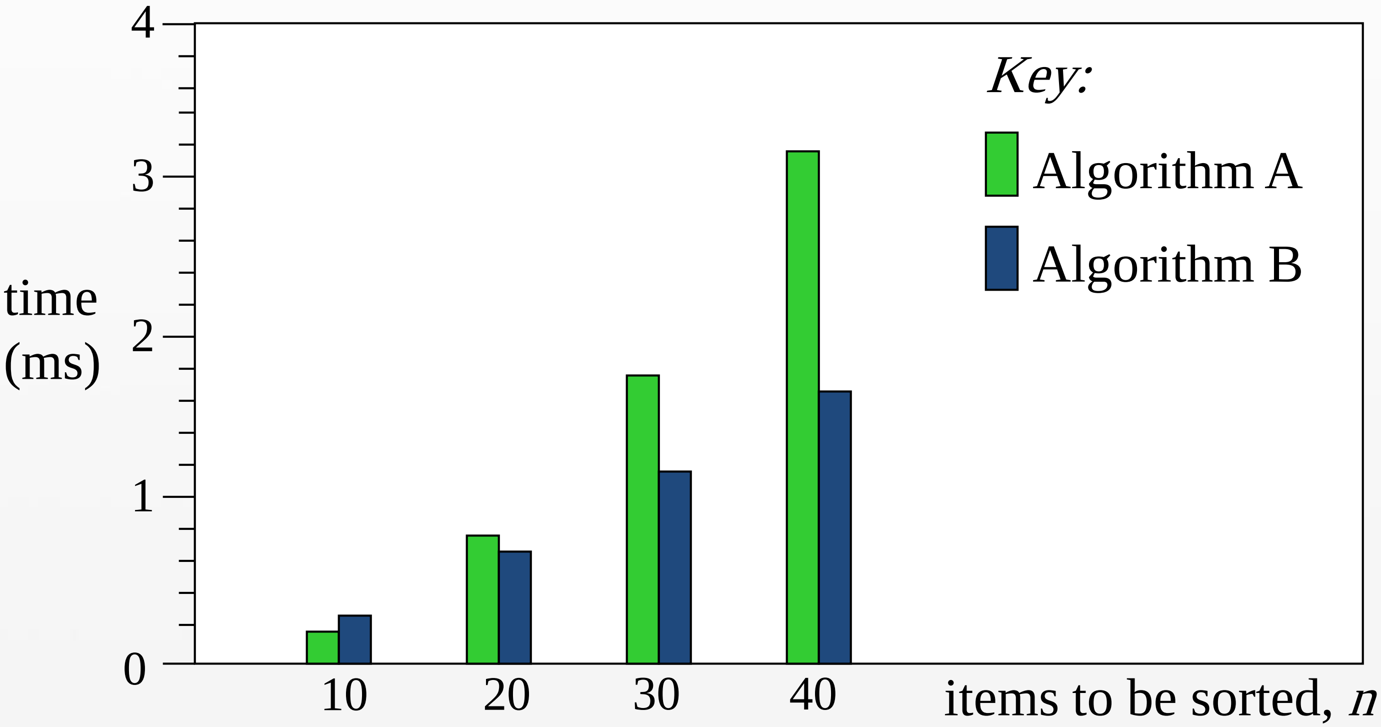
Other factors:

1. Speed of the host machine (hardware)
2. Quality of the compiler (programming language …)
3. Quality of the program (in some cases)
   (+ Methodology; such as procedural vs. object-oriented)

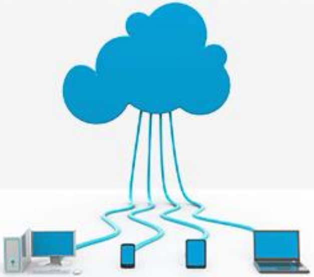
`



Hypothetical profile of two sorting algorithms:



Algorithm B's time grows more slowly than A's

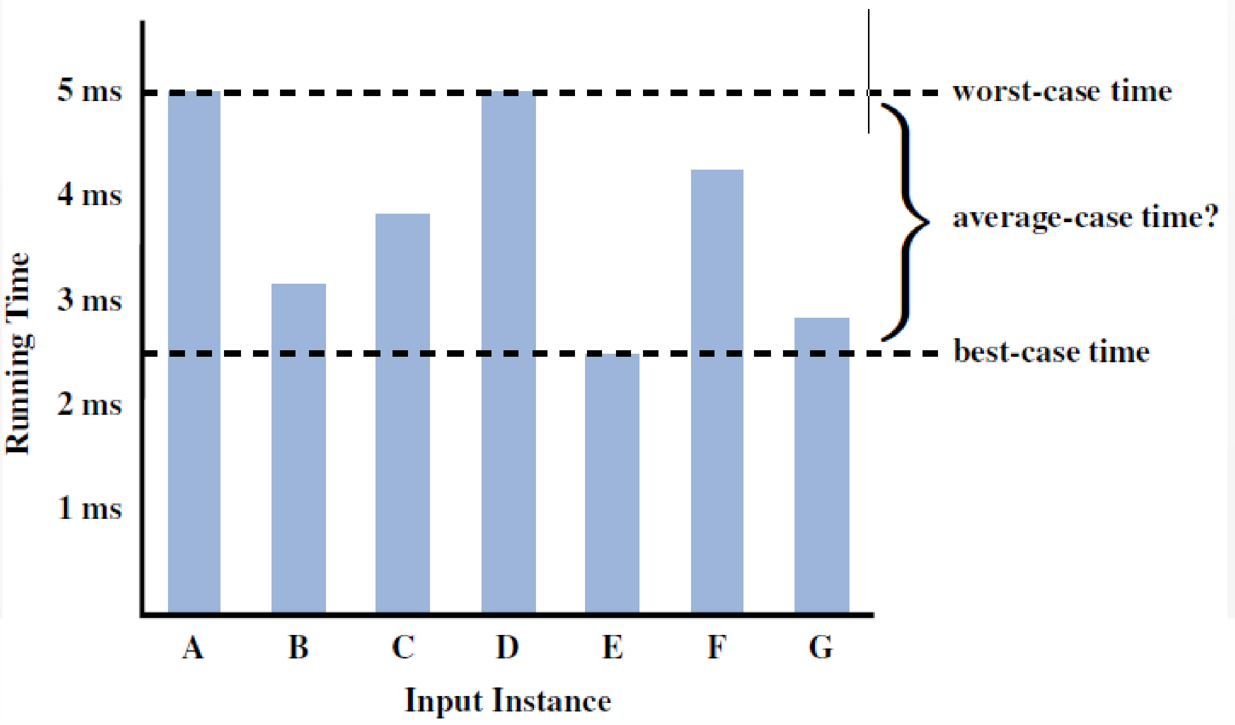


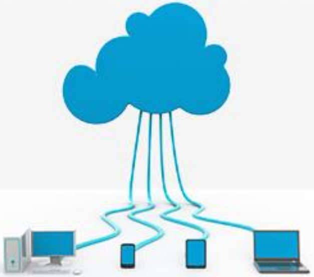
Order of Magnitude

Order of Growth

- Focus on the Worst-Case Input

Running time of an algorithm on a different possible input



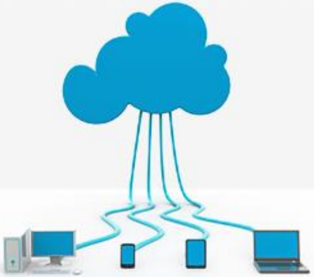


Order of Magnitude

Order of Growth

Generally an algorithm might have

- **Best case**
  - is the function defined by the minimum number of steps taken on any instance of size n
- **Worst case**
  - is the function defined by the maximum number of steps taken on any instance of size n
  - is a guarantee over all inputs of some size
- **Average case**
  - is the function defined by an average number of steps taken on any instance of size n
  - the running time is measured as an average over all of the possible inputs of size n



Order of Magnitude

Measuring the functions’ Rates of Growth

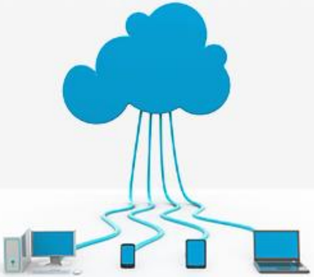
For many interesting algorithms, the exact number of operations is too difficult to analyze mathematically

To simplify the analysis:

- identify the fastest-growing term
- neglect slower-growing terms
  (Small values of *N* generally are not important)
- neglect the constant factor in the fastest-growing term.
  (The exact value of the leading constant of the dominant term is not meaningful across different machines)

The resulting formula is the algorithm’s **time complexity**.

➔ It focuses on the **growth rate** of the algorithm’s time requirement.

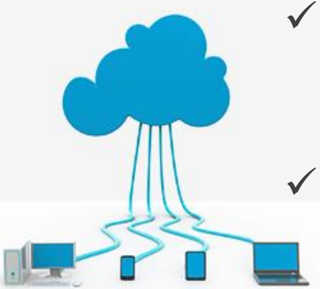


Similarly for **space complexity**.

Order of Magnitude

Measuring the functions' Rates of Growth

- ✓ Small values of *N* generally are not important
  - avoid details, they don't matter when input size (N) is big enough
  - the difference between the best and worst algorithm is less than a blink of the eye
- ✓ For polynomials, use only leading term, ignore coefficients: linear, quadratic
  - the exact value of the leading constant of the dominant term is not meaningful
- ✓ The value of the cubic function is almost entirely determined by the cubic term
  - $N = 1{,}000;\ 10N^3 + N^2 + 40N + 80 = 10{,}001{,}040{,}080$ -> ~ 10,000,000,000 because of N^3
- ✓ For small amounts of input, making comparisons between functions is difficult (because leading constants become very significant)
  - The function N + 2,500 is larger N2 than when N is less than 50
- ✓ When input sizes are very small, use the simplest algorithm
  - very effective tool, but has limitations; not appropriate for small amounts of input



Big-O Notation

We express complexity using ***Big-O* notation**

We use ***big-O* notation** to:

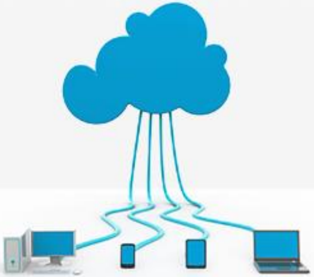
- capture the most dominant term in a function
- and to represent the growth rate

Big-O defines an upper bound on the complexity of the algorithm

T(n) is ***O* (F(n))** if there are: positive constants **c** and **n_0** ; such that

$$T(n) \leq cF(n) \text{ when } n \geq n_0$$

- **n** = is the number of inputs (array / list of size **n**)
- **T(n)** = time to run on an **n** inputs

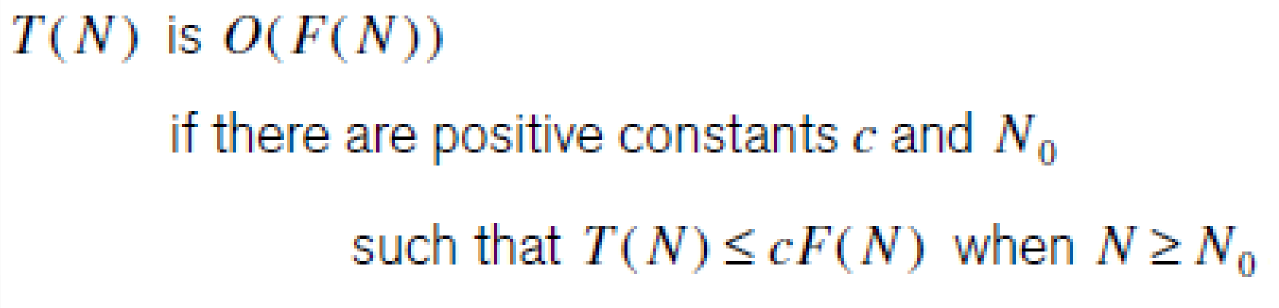


Big-O Notation

General *Big-Oh* rules

Think of $O(f(N))$ as "less than or equal to" f (N)

- Upper bound: "grows slower than or same rate as" f (N)

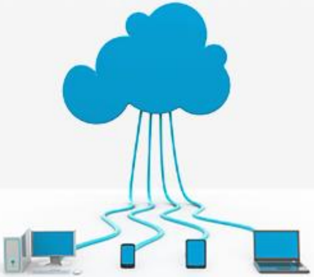


$T(N)$ is $O(F(N))$

if there are positive constants c and N_0

such that $T(N) \le cF(N)$ when $N \ge N_0$

Other notations are less used: **Big-Omega , Big-Theta , Little-Oh**



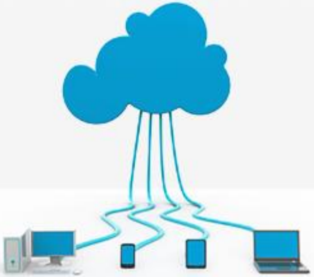
Seven Fundamental Functions

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size *n*, taking a “big-picture” approach.

F (n): characterizes the number of primitive operations that are performed as a function of the input size **n**.

The Seven common functions:

1. Constant function
2. Logarithm function
3. Linear function
4. N-Log-N function
5. Quadratic function
6. Cubic function (and other polynomials)
7. Exponential function



Constant Function

$\boldsymbol{O}$ **(1)** $f(n) = c$

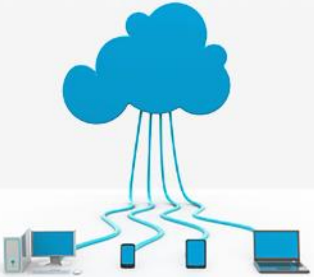
An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size.

Examples:

- **array**: accessing any element
- **fixed-size stack**: push and pop methods
- **fixed-size queue**: enqueue and dequeue methods

<u>e.g.</u>

```
x = a[5]          // O (1)
sum += x          // O (1)
```



Logarithmic Function

O **(log n)** $f(n) = \log_b n$

definition: For any $B, N > 0$, $\log_B N = K$ if $B^K = N$

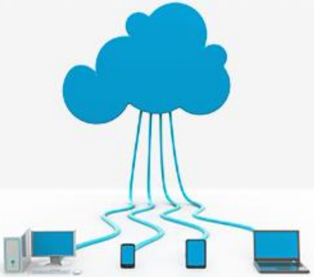
B is the base of the logarithm.

is defined as the inverse of a power, as follows:

$x = \log_b n$ if and only if $b^x = n$

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size

- e.g. **Dichotomy** (**Binary search**) T(n)= T(n/2) + O (1) = O (log n)



An algorithm is O (log N) if it takes constant (O (1)) time to cut the problem size by a constant fraction (which is usually 2)

Logarithmic Function

***O* (log n)** $f(n) = \log_b n$

The logarithm is a slowly growing function

- the logarithm of 1,000,000 with the typical base 2, is only 20
- the logarithm grows more slowly than a square or cube (or any) root

log 2 = 1

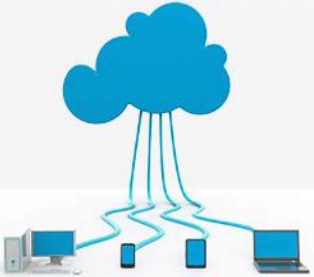
log 4 = 2

log 32 = 5

log 1024 = 10

log 1000 000 000 ≈ 30

Proof: log 109 = log 103 x 103 x 103
= log 103 + log 103 + log 103
≈ 10 + 10 + 10 = 30



In computer science, when the base is omitted, it defaults to 2, for several reasons

Logarithmic Function

O **(log n)** $f(n) = \log_b n$

If the iterative variable is incremented geometrically, then it's *O* (log n)

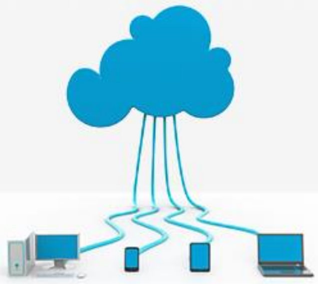
- e.g. For (i=1 ; i<n ; i = i * 2) // ***O*** **(log n)**

because the algorithm divides the working area in half with each iteration

<u>Example</u>:

. **repeated doubling**: Starting from *X* = 1, how many times should *X* be doubled before it is at least as large as *N*?

. **repeated halving**: Starting from *X* = *N*, if *N* is repeatedly halved, how many iterations must be applied to make *N* smaller than or equal to 1?



Note that, the implementations don't have to be using loops, they maybe implemented using recursion.

Linear Function

***O* (n)** $f(n) = n$

A ***linear function*** has a dominant term that is some constant times ***N***

Linear Algorithm : in which time essentially is directly proportional to amount of input

- e.g. downloading small files need less time than bigger files

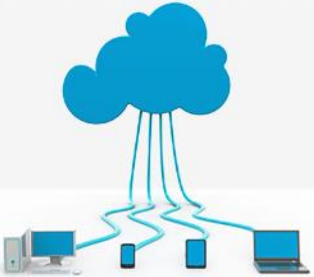
If there is a single iteration, and the iterative variable is incrementing linearly then it's ***O* (n)**

- e.g. **Sequential search** $T(n)= T(n-1) + O(1) = O(n)$
- e.g. **Traversal tree** $T(n) = 2T(n/2) + O(1) = O(n)$

$T(N) = cN \Rightarrow T(\alpha N) = c(\alpha N) \Rightarrow T(\alpha N) = \alpha cN = \alpha T(N)$

<u>e.g.</u>

```
for (i=0 ; i<n ; i++)                // O (n)
for (i=0 ; i<n ; i = i + 4)          // O (n)
```



N-Log-N Function

O **(n log n)** $f(n) = n \log n$

The ***O*** **(N log N)** expression represents a function whose dominant term is *N* times the logarithm of *N*.

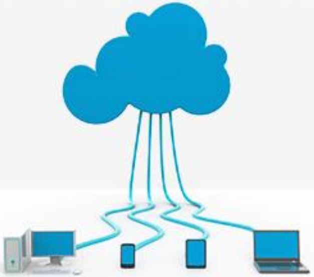
e.g. **Quick sort** $T(n) = 2T(n/2) + O(n) = O(n \log n)$

If there is nested loop, where one has a complexity of ***O*** **(n)** and the other ***O*** **(log n)**, then overall complexity is ***O*** **(n log n)**;

e.g.

```
for ( i=0 ; i<n ; i++) {                              // O (n)
   for ( j=1 ; j<n ; j=j*3) {                         // O (log n)
      /* … */
   }
}
```

Overall: ***O*** **(n log n)**



Quadratic Function

O (n^2) $f(n) = n^2$

A ***quadratic function*** is a function whose dominant term is some constant times **N^2**

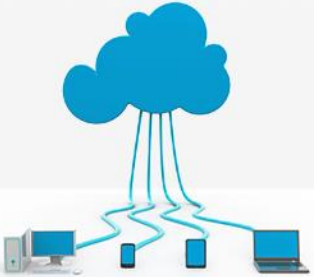
Quadratic algorithms are almost always impractical when the input size is more than a few thousand

$T(N) = cN^2 \;\; => \;\; T(\alpha\ N) = c(\alpha\ N)^2 \;\; => \;\; T(\alpha\ N) = \alpha^2 cN^2 = \alpha^2 T(N)$

An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size

- e.g. **bubble sort , insertion sort**

  selection sort $T(n) = T(n-1) + O(n) = O(n^2)$



Cubic Function & Other Polynomials

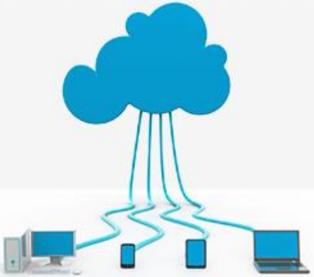
***O* (n³)** $f(n) = n^3$

A ***cubic function*** is a function whose dominant term is some constant times **N³**

- e.g. $10\ N^3 + N^2 + 40\ N + 80$

Cubic algorithms are impractical for input sizes as small as a few hundred

$T(N) = cN^3 \Rightarrow T(\alpha\ N) = c(\alpha\ N)^3 \Rightarrow T(\alpha\ N) = \alpha^3 cN^3 = \alpha^3\ T(N)$



Exponential Function

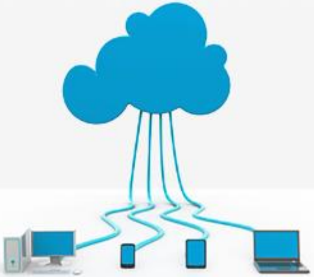
O **(b^N)** $f(n) = b^n$

- ***b*** : a positive constant, called the ***base***
- ***n*** **:** the ***exponent***.

That is, function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times.

- e.g. Geometric Sums

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

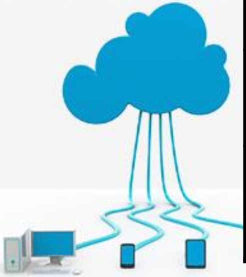


Functions in order of increasing growth rate

Increasing cost

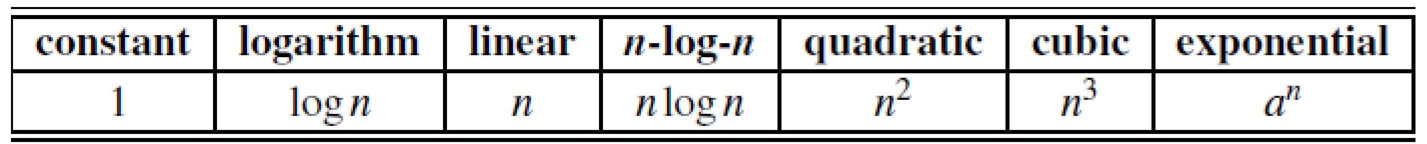
Name	Big-Oh	Comment
Constant	O (1)	Can't beat it!
Log log	O (log log N)	Extrapolation search. **Feasible**
Logarithmic	O (log N)	Typical time for good searching algorithms. **Feasible**
Log-squared	O (log^2 N)	**Feasible**
Linear	O (N)	This is about the fastest that an algorithm can run given that we need O (n) just to read the input. **Feasible**
N log N	O (N log N)	Most sorting algorithms. **Feasible**
Quadratic	O (N^2)	Acceptable when the data size is small ($n < 1000$). **Sometimes Feasible**
Cubic	O (N^3)	Acceptable when the data size is small ($n < 1000$). **Sometimes Feasible**
Exponential	O (b^N)	Only good for really small input sizes ($n \leq 20$). **Rarely Feasible**

Polynomial time

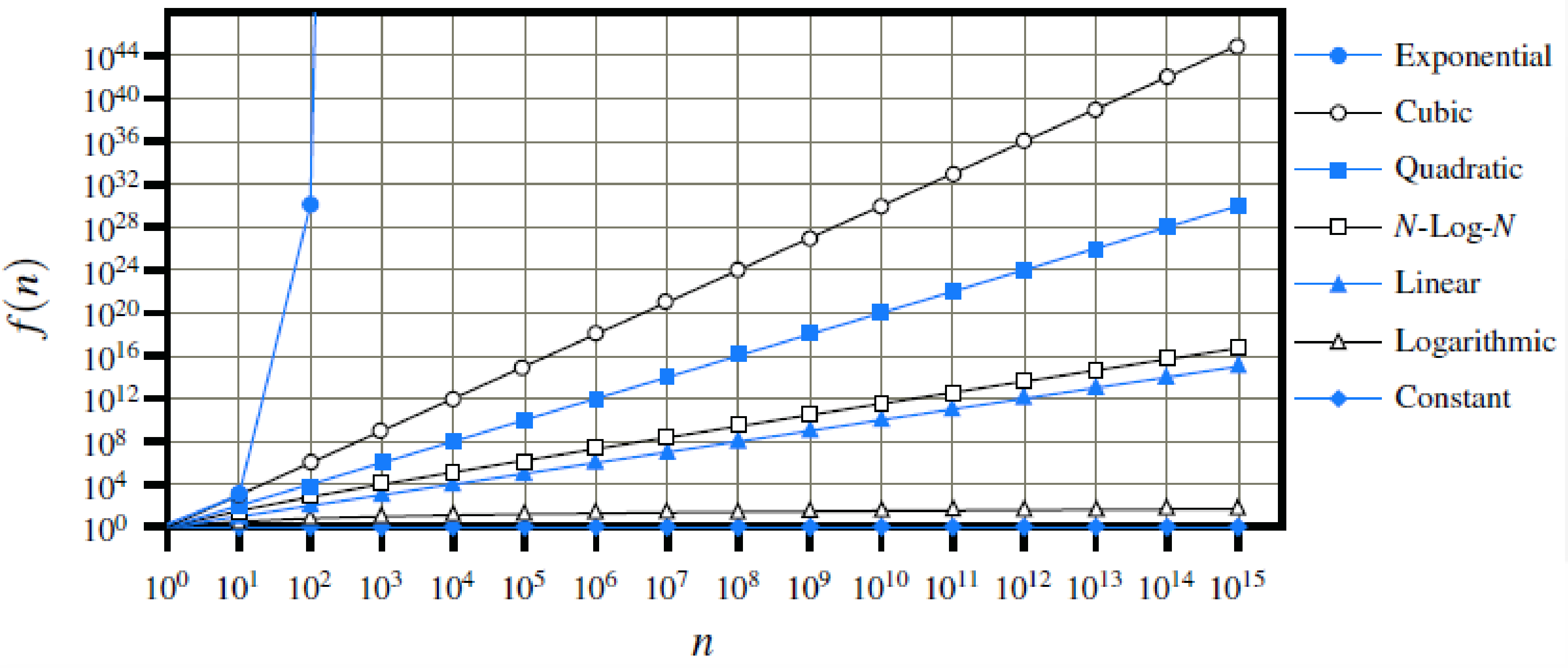


Seven Fundamental Functions

Comparing Growth Rates

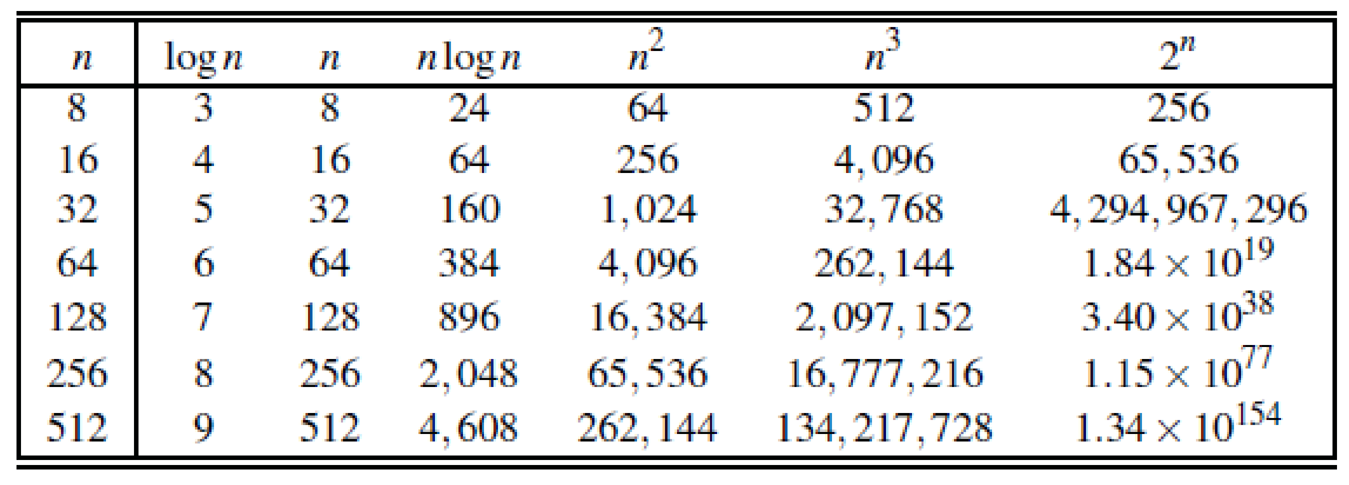


constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

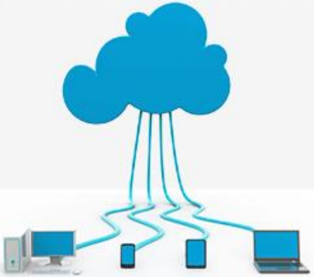


Seven Fundamental Functions

Comparing Growth Rates



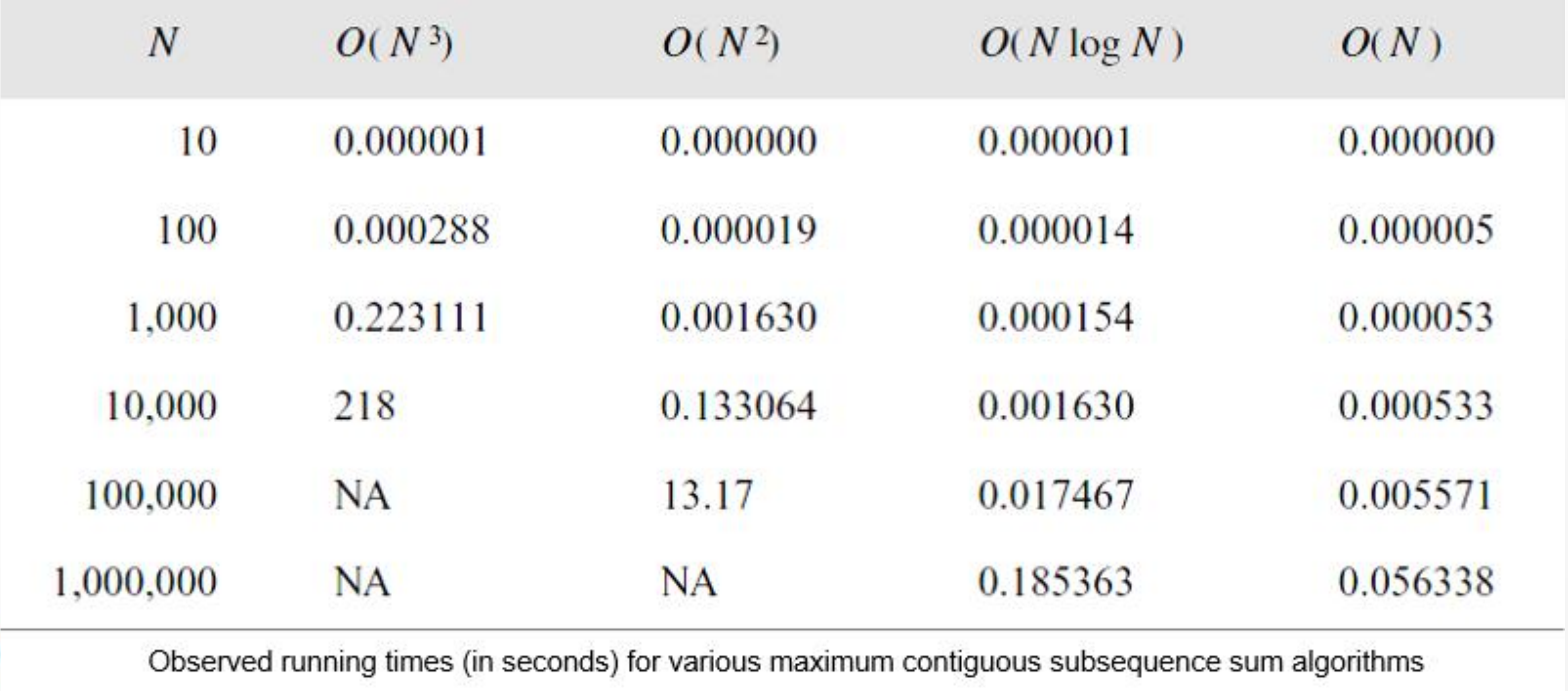
n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}



Refer to the text book p.170:

4.3.3 **Examples of Algorithm Analysis**

Example for comparing running times (in seconds)

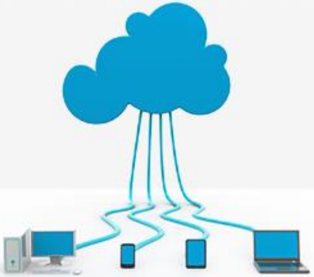


N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.000001	0.000000	0.000001	0.000000
100	0.000288	0.000019	0.000014	0.000005
1,000	0.223111	0.001630	0.000154	0.000053
10,000	218	0.133064	0.001630	0.000533
100,000	NA	13.17	0.017467	0.005571
1,000,000	NA	NA	0.185363	0.056338

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

Example for comparing running times (in seconds)

N	O(log N)	O(N)	O(N log N)	$O(N^2)$
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

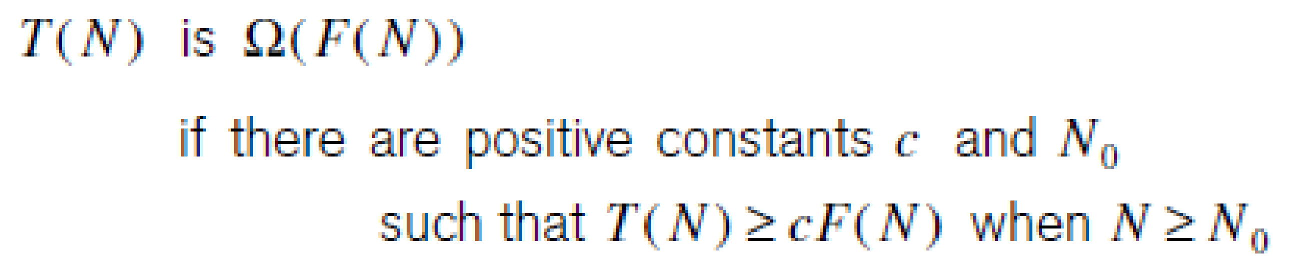


Other Notations

Big-Omega Ω

Think of Ω (f (N)) as “greater than or equal to” f (N)

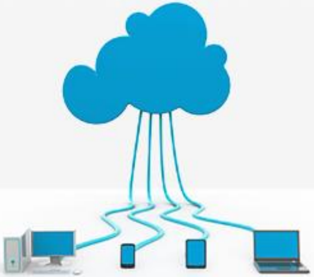
- Lower bound: “grows faster than or same rate as” f (N)
- advanced analysis



$T(N)$ is $\Omega(F(N))$

if there are positive constants c and N_0

such that $T(N) \geq cF(N)$ when $N \geq N_0$



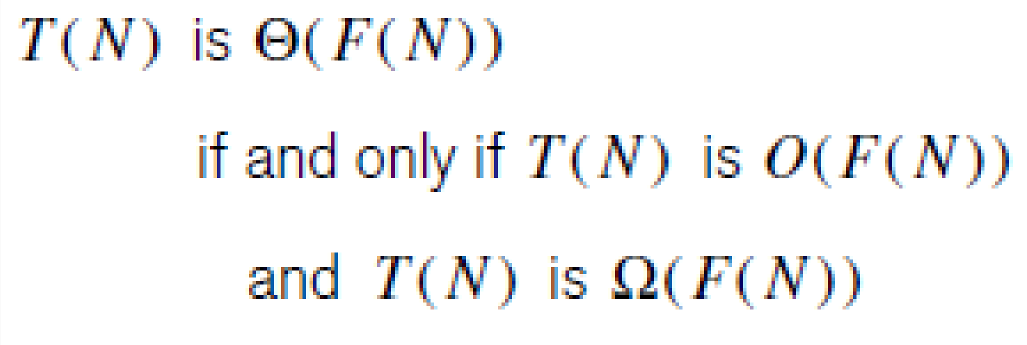
Other Notations

Big-Theta Θ

Exact bound (upper + lower) => Alg. cannot be improved

Think of Θ (f (N)) as “equal to” f (N)

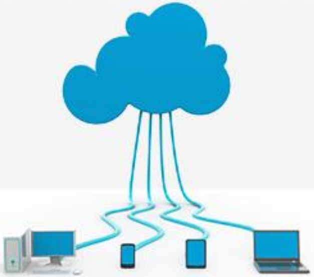
- “Tight” bound: same growth rate



$T(N)$ is $\Theta(F(N))$

if and only if $T(N)$ is $O(F(N))$

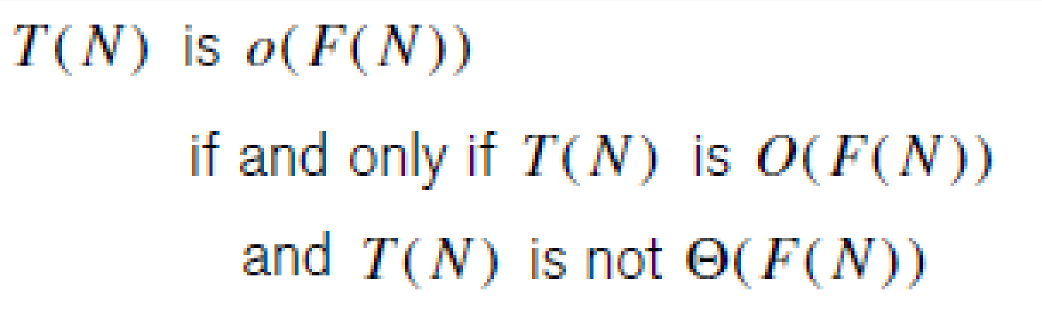
and $T(N)$ is $\Omega(F(N))$



Other Notations

Little-Oh *o*

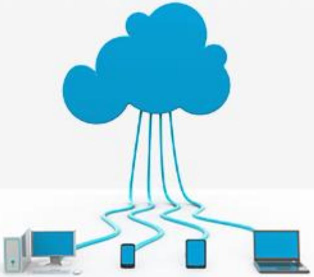
Strict upper bound (hard to verify and less significant)



$T(N)$ is $o(F(N))$

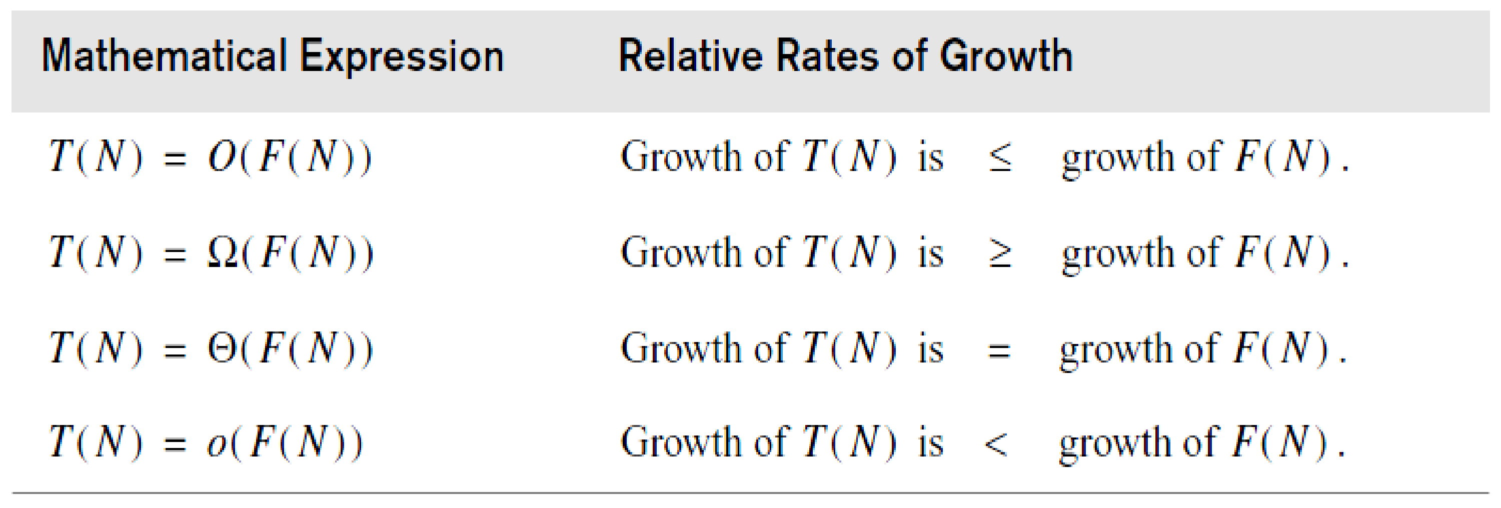
if and only if $T(N)$ is $O(F(N))$

and $T(N)$ is not $\Theta(F(N))$

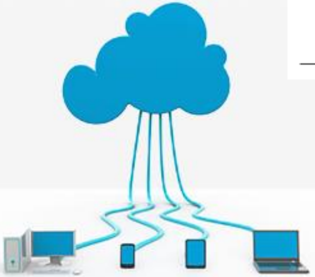


Other Notations

Meanings of the various growth functions



Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$.



Example 1

Finding the sum of an array of numbers

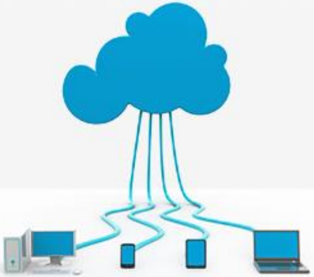
```
int Sum(int A[], int N) {
  int sum = 0;

  for  (i=0; i < N; i++){
    sum += A[i];
  } //end-for

  return sum;
} //end-Sum
```

How many steps does this algorithm take to finish?

We define a step to be a unit of work that can be executed in constant amount of time in a machine.



Example 1

Finding the sum of an array of numbers

```
int Sum(int A[], int N) {
   int sum = 0;

   for  (i=0; i < N; i++){
      sum += A[i];
   } //end-for

   return sum;
} //end-Sum
```

Times Executed

1

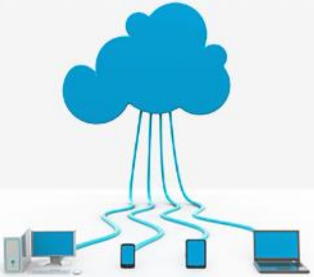
N + 1

N

1

Total: 1+N+1+N+1 = **2N + 3**

- Running Time: T(N) = 2N + 3
  - N is the input size (number of ints) to add



Homework

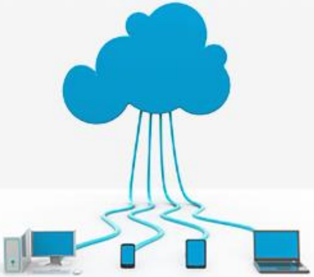
I. Solve the 5 exercises on the next slides

Referring to the textbook, solve the following exercises:

II. page 183, for each one of the 5 example functions, compute:

- the total time cost needed to execute
- the time complexity in terms of ***Big-Oh*** notation

III. page 185, R-4.33



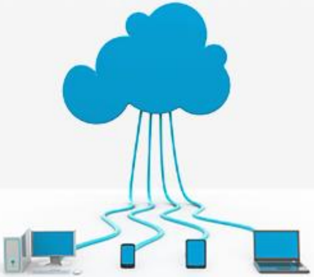
Exercise 1

We have an O(n) algorithm,

- For 160 elements takes 0.3 seconds
- For 480 elements takes 0.9 seconds
- For 800 elements takes ….?

We have an O(n2) algorithm

- For 220 elements takes 2.2 seconds
- For 440 elements takes 8.8 seconds
- For 660 elements takes …?



Exercise 2

If an algorithm takes 0.06 second to run with the problem size 100, what is the time requirement (approximately) for that algorithm with the problem size 400?

T(16) = 0.2 second

If its order is:

O(1) → T(40) =

$O(\log_2 n)$ → T(40) =

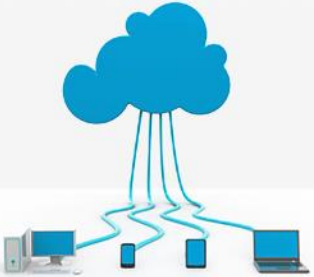
O(n) → T(40) =

$O(n*\log_2 n)$ → T(40) =

$O(n^2)$ → T(40) =

$O(n^3)$ → T(40) =

$O(2^n)$ → T(40) =

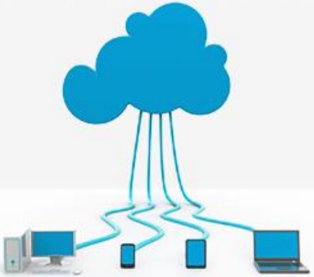


Exercise 3

	Cost	Times
`i = 1;`	c1	?
`sum = 0;`	c2	?
`while (i <= n)`	c3	?
`{`		
`        i = i + 1;`	c4	?
`        sum = sum + i;`	c5	?
`}`		

T(n) = ?

➔ So, the growth-rate function for this algorithm is ?

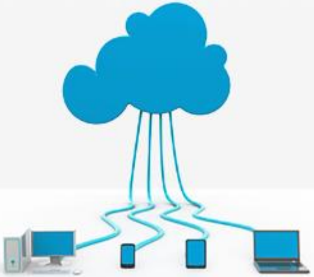


Exercise 4

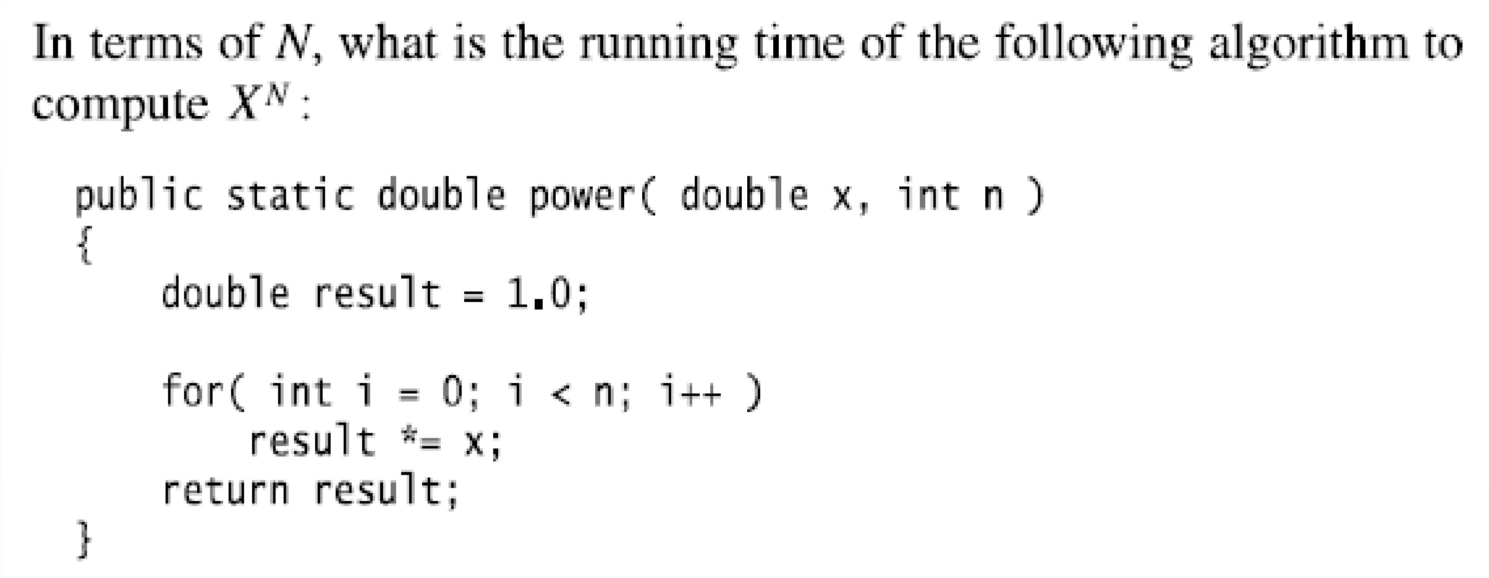
```
                                   Cost          Times
i=1;                                c1             ?
sum = 0;                            c2             ?
while (i <= n)                      c3             ?
{
     j=1;                           c4             ?
     while (j <= n)                 c5             ?
     {
        sum = sum + i;              c6             ?
        j = j + 1;                  c7             ?
     }
     i = i +1;                      c8             ?
}
```

T(n) = ?

➔ So, the growth-rate function for this algorithm is ?



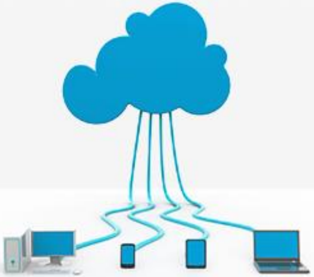
Exercise 5



In terms of N, what is the running time of the following algorithm to compute X^N :

```
public static double power( double x, int n )
{
    double result = 1.0;

    for( int i = 0; i < n; i++ )
        result *= x;
    return result;
}
```



Powers

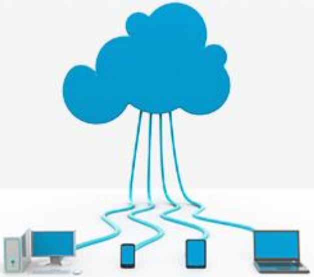
Simple power algorithm

Consider a number *b* and a non-negative integer *n*.
Then ***b*** **to the power of** ***n*** (written $\boldsymbol{b^{n}}$) is the multiplication of *n* copies of *b* :

$$b^{n} = b \text{ x } \ldots \text{ x } b$$

E.g.:

$$b^{3} = b \text{ x } b \text{ x } b$$
$$b^{2} = b \text{ x } b$$
$$b^{1} = b$$
$$b^{0} = 1$$

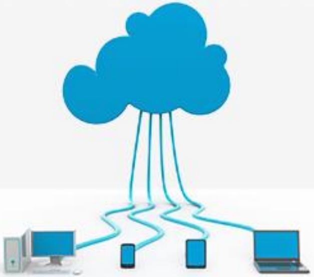


Powers

Simple power algorithm

```
To compute $b^n$:
    1. Set $p$ to 1
    2. For $i$ = 1, …, $n$, repeat:
        2.1 Multiply $p$ by $b$
    3. Terminate with answer $p$
```

```
static int power (int b, int n) {
// Return b^n (where n is non-negative)
        int p = 1;
        for (int i = 1; i <= n; i++)
            p *= b;
        return p;
}
```



Powers

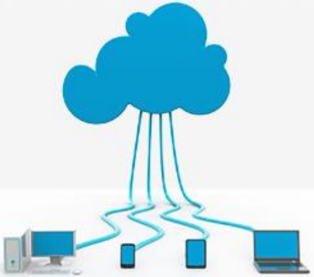
Simple power algorithm - Analysis

Counting multiplications:

- Step 2.1 performs a multiplication
- This step is repeated *n* times
- No. of multiplications = *n*
- Time taken is approximately proportional to n.

➢ Time complexity is **of order n**

✓ This is written ***O*** **(n)**



Powers

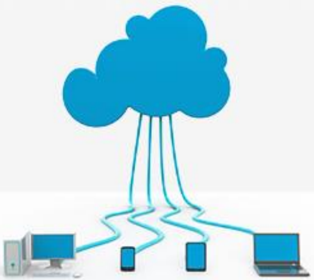
Smart power algorithm

- Idea: $b^{1000} = b^{500} \times b^{500}$. If we know b^{500}, we can compute b^{1000} with only 1 more multiplication!

- To compute b^n:

1. Set p to 1, set q to b, and set m to n
2. While $m > 0$, repeat:
   2.1. If m is odd, multiply p by q
   2.2. Halve m
   (discarding any remainder)
   2.3. Multiply q by itself
3. Terminate with answer p

```
static int power (int b, int n) {
// Return b^n (where n is non-negative)
        int p = 1, q = b, m = n;
        while (m > 0) {
            if (m%2 != 0)  p *= q;
            m /= 2;  q *= q;
        }
        return p;
}
```



Powers

Smart power algorithm - Analysis

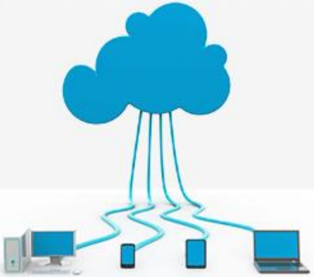
Counting multiplications:

Steps 2.1: 3 together perform at most 2 multiplications.

They are repeated as often as we must halve the value of n (discarding any remainder) until it reaches 0,

i.e., floor($\log_2 n$) + 1 times

Max. no. of multiplications $= 2 (\text{floor} (\log_2 n) + 1)$
$= 2 \text{ floor} (\log_2 n) + 2$



Powers

Smart power algorithm - Analysis

Max. no. of multiplications =
2 floor ($\log_2 n$) + 2

Neglect slow-growing term, +2

Simplify to 2 floor ($\log_2 n$)

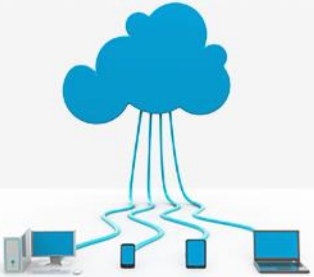
Neglect constant factor, 2

then to floor ($\log_2 n$)

then to $\log_2 n$

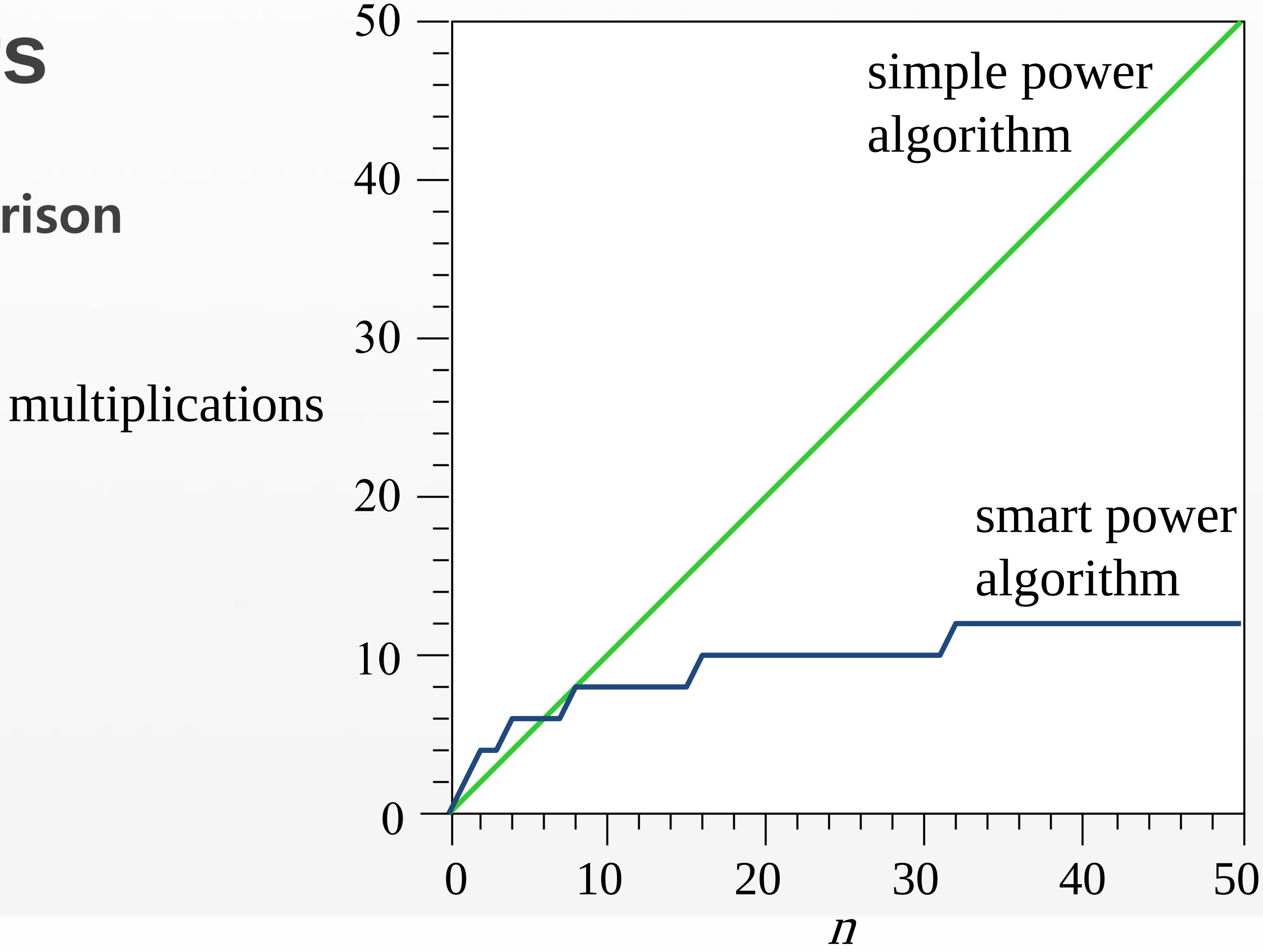
Neglect floor(), which on average subtracts 0.5, a constant term

- Time complexity is **of order log n**
  - ✓ This is written ***O*** **(log n)**



Powers

Comparison


multiplications
n
simple power algorithm
smart power algorithm