Data Structure
- Analysis Tools -

Prof. Gi Seok Park
Intelligent Networking Lab
Department of Computer Engineering
Dongguk University

dongguk UNIVERSITY

Chapter. 4.1
– Seven Functions

1. Constant Function

- The simplest **constant function**

$$\boldsymbol{f(n) = c}$$

  - where c is constant (e.g., $c = 5, c = 27, c = 2^{10}$)
  - It doesn't matter what the value of n is, $f(n)$ is always be equal to the constant value c.

2. Logarithm Function

☐ Logarithm function

$$x = log_b n \text{ if and only if } b^x = n$$

- where $b > 1$. When base b is 2,

$$log\ n = log_2 n$$

- **Proposition)** Logarithm Rules
  - $a > 0,\ b > 1, c > 0$, and $d > 1$, we have

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5*

3. Linear Function

☐ Linear function

$$f(n) = n$$

- This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements.
  - Comparing a number x to each element of an array with size n requires n comparisons.
- The linear function represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects

4. N-Log-N Function

☐ n-log-n function

$$f(n) = nlogn$$

- This function grows a little faster than the linear function and a lot slower than the quadratic function.

5. Quadratic Function

☐ Quadratic function

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

- The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times.

6. Cubic Function and Other Polynomials

☐ Cubic function

$$f(n) = n^3$$

☐ Polynomials

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

- Coefficients : $a_0, a_1, a_2, \ldots, a_d$
- Degree : the highest power in the polynomial d

7. Exponential Function

☐ Exponential function

$$f(n) = b^n$$

- b is a positive number, called the base
- n is the exponent
- $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times.

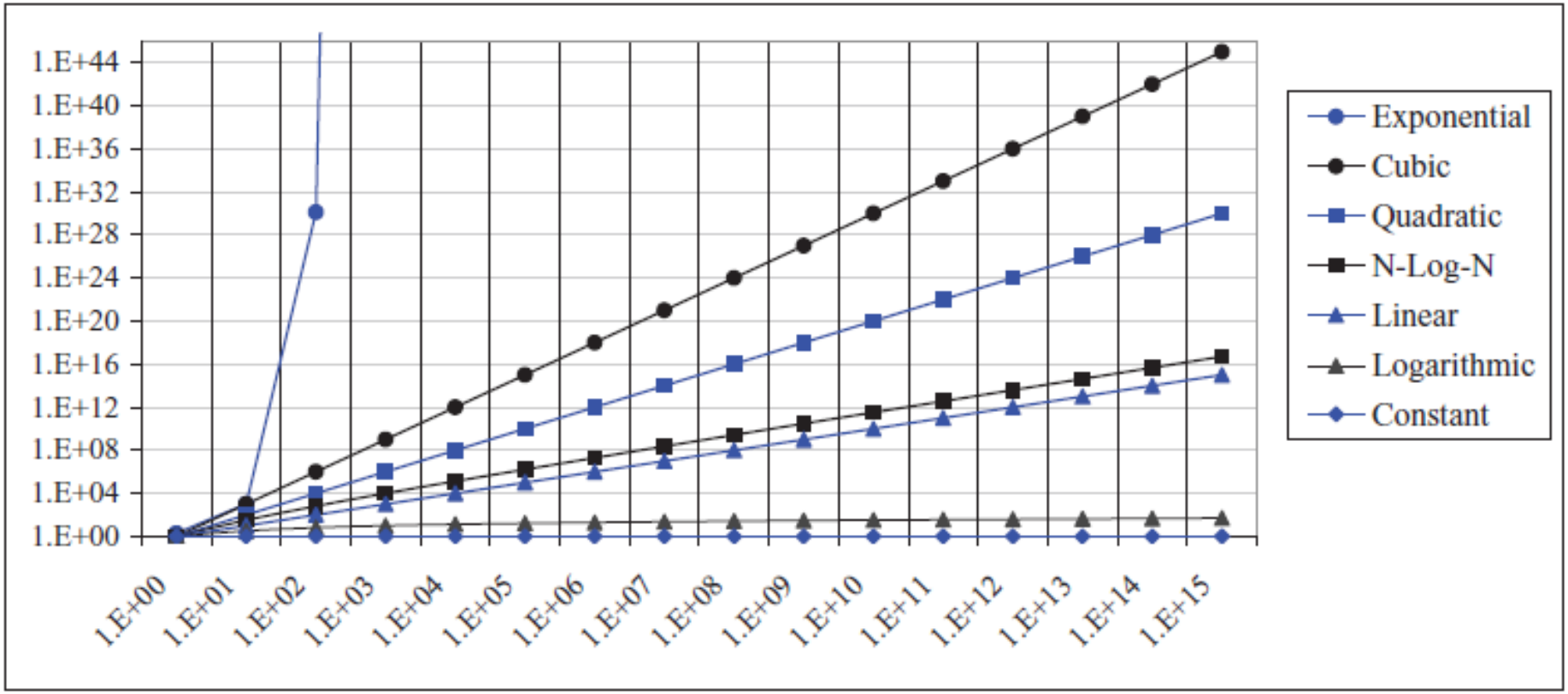
- Geometric sums
  - Suppose we have a loop where each iteration takes a multiplicative factor longer than the previous one.
  - For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation

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

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

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.



Chapter. 4.2
– Analysis of Algorithms

Analysis of Algorithms

- **Data Structure**
  - Systematic way of organizing and accessing data
- **Algorithm**
  - Step-by-step procedure for performing some task in a finite amount of time

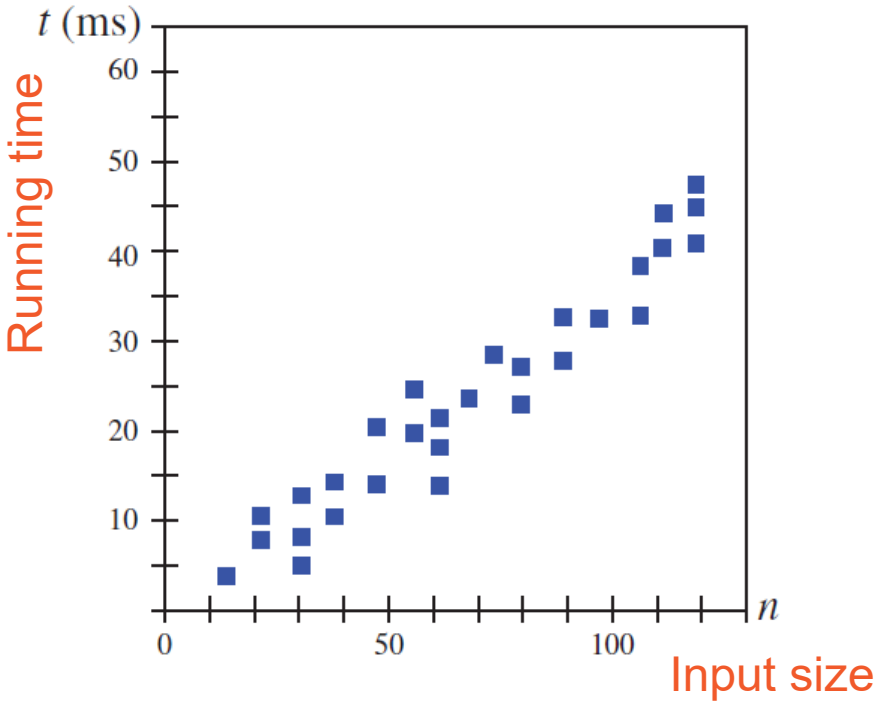
To classify some data structures and algorithms as “good,”
we need to have precise ways of analyzing them.

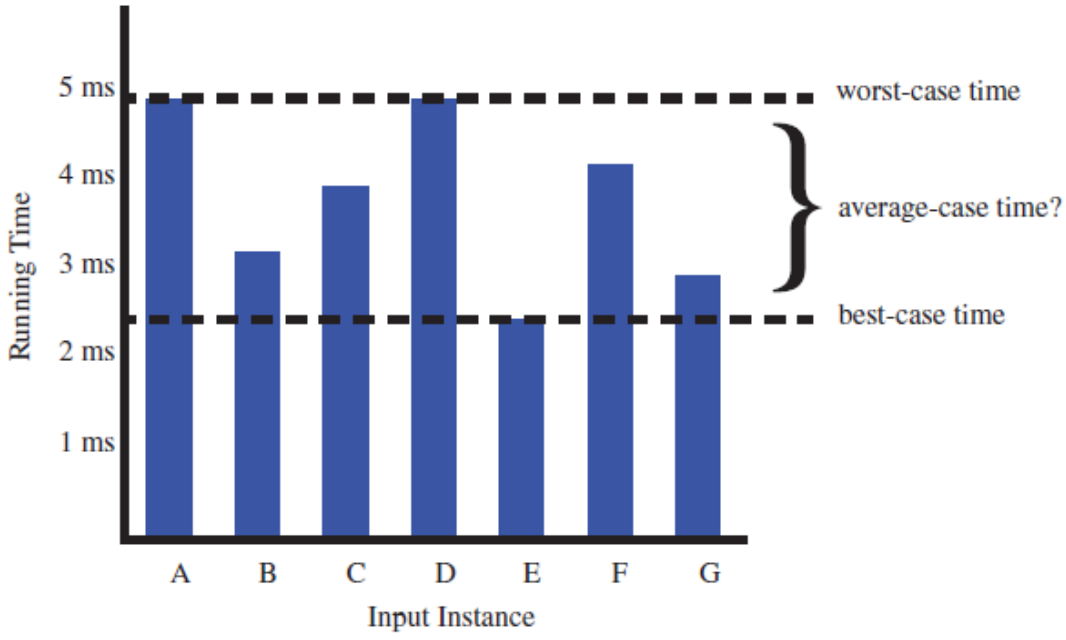
We are interested in characterizing an algorithm’s running time as a function of the input size.

Experimental Studies

- If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution.

- We are interested in determining the general dependence of running time on the size of the input.





Primitive Operations

- If we wish to analyze a particular algorithm without performing experiments on its running time, we can perform an analysis directly on the high-level pseudo-code instead.
  - **Definition**) *primitive operations* — low-level instruction with a constant execution time
    - *Assigning a value to a variable*
    - *Performing an arithmetic operation (for example, adding two numbers)*
    - *Comparing two numbers*
    - *Indexing into an array*
    - *Following an object reference*
    - *Returning from a function*

The number, t, of primitive operations an algorithm performs is proportional to the actual running time of that algorithm.
➔ Simply count how many primitive operations are executed

Asymptotic Notation

```
Algorithm arrayMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
  currMax ← A[0]
  for i ← 1 to n − 1 do
    if currMax < A[i] then
      currMax ← A[i]
  return currMax
```

Running time of `arrayMax` grows **proportionally** to the input size n

With its true running time being n times a constant factor that depends on the specific computer

Disregards constant factors!

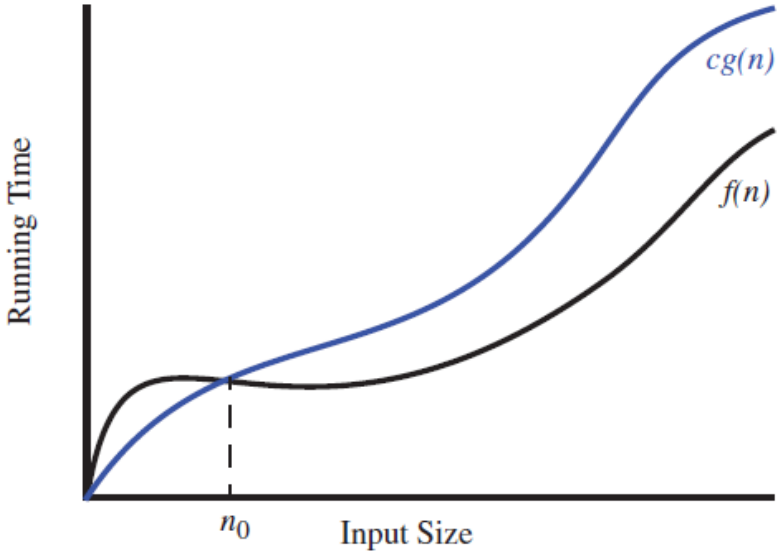
Asymptotic Notation

☐ "**Big-Oh**" Notation

$\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ **:** functions mapping nonnegative integers to real numbers

$\boldsymbol{f(n)}$ **is** $\boldsymbol{O(g(n))}$ if there is a real constant $\boldsymbol{c > 0}$ and an integer constant $\boldsymbol{n_0 > 1}$

➔ $f(n) \in O(g(n))$ such that $f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0$



Example: function $8n - 2$ is $O(n)$

Big-Oh notation allows us to say that a function $f\ (n)$ is "less than or equal to" another function $g(n)$ up to a constant factor and in the ***asymptotic*** sense as n grows toward infinity.

Asymptotic Notation

☐ Characterizing Running Times using Big-Oh Notation

The Algorithm `arrayMax`, for computing the maximum element in an array of n integers, runs in $O(n)$ time.

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
  $currMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $currMax < A[i]$ **then**
      $currMax \leftarrow A[i]$
  return $currMax$

Asymptotic Notation

- Some Properties of Big-Oh Notation
  - Big-Oh notation allows us to **ignore constant factors and lower order terms** and focus on the main components of a function that affect its growth.

Example 4.8: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *is* $O(n^4)$.

Justification: Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)n^4 = cn^4$, for $c = 15$, when $n \geq n_0 = 1$. ■

Proposition 4.9: *If* $f(n)$ *is a polynomial of degree* d, *that is,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, *then* $f(n)$ *is* $O(n^d)$.

Justification: Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \cdots \leq n^d$; hence,

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (a_0 + a_1 + a_2 + \cdots + a_d) n^d.$$

Therefore, we can show $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 + \cdots + a_d$ and $n_0 = 1$. ■

Asymptotic Notation

The highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial

Example 4.10: $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

Justification: $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 2$ (note that $n\log n$ is zero for $n = 1$). ■

Example 4.11: $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$. ■

Example 4.12: $3\log n + 2$ *is* $O(\log n)$.

Justification: $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case. ■

Example 4.13: 2^{n+2} *is* $O(2^n)$.

Justification: $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ■

Example 4.14: $2n + 100\log n$ *is* $O(n)$.

Justification: $2n + 100\log n \leq 102n$, for $n \geq n_0 = 2$; hence, we can take $c = 102$ in this case. ■

Asymptotic Notation

☐ "**Big-Omega**" Notation

- provide an asymptotic way of saying that a function grows at a rate that is **"greater than or equal to"** that of another

$\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ **:** functions mapping nonnegative integers to real numbers

$\boldsymbol{f(n)}$ **is** $\boldsymbol{\Omega(g(n))}$ if $\boldsymbol{g(n)}$ is $\boldsymbol{O(f(n))}$, that is, there is a real constant $\boldsymbol{c > 0}$ and an integer constant $\boldsymbol{n_0 \geq 1}$ such that $f(n) \geq cg(n), \quad \text{for} \quad n \geq n_0$

Example 4.15: $3n\log n + 2n$ *is* $\Omega(n\log n)$.

Justification: $3n\log n + 2n \geq 3n\log n$, for $n \geq 2$.

Asymptotic Notation

☐ "**Big-Theta**" Notation

- allows us to say that two functions grow at the same rate, up to constant factors.

$\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ **:** functions mapping nonnegative integers to real numbers

$\boldsymbol{f(n)}$ **is** $\boldsymbol{\Theta(g(n))}$ if $\boldsymbol{g(n)}$ is $\boldsymbol{O(f(n))}$ and $\boldsymbol{f(n)}$ is $\boldsymbol{\Omega(g(n))}$, that is, there are real constants $\boldsymbol{c' > 0}$ and $\boldsymbol{c'' > 0}$, and an integer constant $\boldsymbol{n_0 \geq 1}$ such that $c'g(n) \leq f(n) \leq c''g(n), \quad \text{for} \quad n \geq n_0$

Example 4.16: $3n\log n + 4n + 5\log n$ *is* $\Theta(n\log n)$.

Justification: $3n\log n \leq 3n\log n + 4n + 5\log n \leq (3+4+5)n\log n$ for $n \geq 2$. ■

Asymptotic Analysis

☐ Asymptotic growth rate

$1 \quad \log n \quad n \quad n\log n \quad n^2 \quad n^3 \quad 2^n$ ⟶ increasing growth rate

- Algorithm **A** : running time of $\boldsymbol{O(n)}$
- Algorithm **B** : running time of $\boldsymbol{O(n^2)}$

➔ Algorithm A is asymptotically better than algorithm B

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}

<Selected values of fundamental functions in algorithm analysis>

Using the Big-Oh Notation

- Example) computing **prefix averages** of a sequence of numbers
  - Given an array X storing n numbers, we want to compute an array A such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$, for $i = 0,\ldots,n-1$, that is,

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}$$

  - **Quadratic-Time Algorithm**

Algorithm prefixAverages1(X):
Input: An n-element array X of numbers.
Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.

```
Let A be an array of n numbers.                    O(n)
for i ← 0 to n − 1 do                              n times
    a ← 0                                          i + 1 times
    for j ← 0 to i do                              1 + 2 + ··· + n times = n(n + 1)/2
        a ← a + X[j]
    A[i] ← a/(i + 1)                               n times
return array A
```

prefixAverages1 : $O(n^2)$

Using the Big-Oh Notation

- Example) computing **prefix averages** of a sequence of numbers
  - Two consecutive averages $A[i-1]$ and $A[i]$ are similar

$$\begin{aligned} A[i-1] &= (X[0]+X[1]+\cdots+X[i-1])/i \\ A[i] &= (X[0]+X[1]+\cdots+X[i-1]+X[i])/(i+1) \end{aligned}$$

  - **Linear-Time Algorithm**

$$\boldsymbol{S_i = X[0] + X[1] + \cdots + X[i]}, \text{ then } \boldsymbol{A[i] = S_i/(i+1)}$$

Algorithm prefixAverages2(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$.

Let A be an array of n numbers.
$s \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $s \leftarrow s+X[i]$
  $A[i] \leftarrow s/(i+1)$
return array A

`prefixAverages2` : $\boldsymbol{O(n)}$

Recursive Algorithm for Computing Powers

- Example) power function $p(x,n)$, defined as $p(x,n) = x^n$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{otherwise} \end{cases}$$

$O(n)$

$$\begin{aligned}
2^4 &= 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\
2^5 &= 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\
2^6 &= 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\
2^7 &= 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128
\end{aligned}$$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Each recursive call of function Power(x,n) divides the exponent, n, by two.

Algorithm Power(x,n):
Input: A number x and integer $n \geq 0$
Output: The value x^n

if $n = 0$ **then**
  return 1
if n is odd **then**
  $y \leftarrow$ Power$(x,(n-1)/2)$
  return $x \cdot y \cdot y$
else
  $y \leftarrow$ Power$(x,n/2)$
  return $y \cdot y$

$O(\log n)$