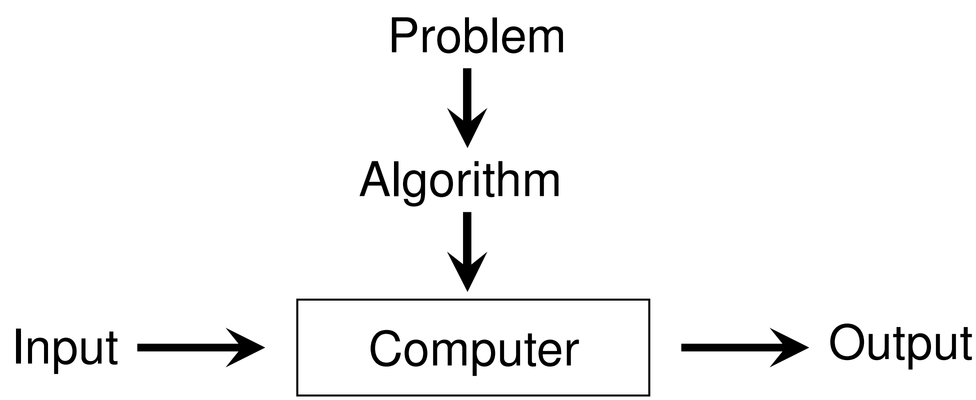
Module 1:
Analyzing the Efficiency of Algorithms

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

What is an Algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



- Important Points about Algorithms
  - The non-ambiguity requirement for each step of an algorithm cannot be compromised
  - The range of inputs for which an algorithm works has to be specified carefully.
  - The same algorithm can be implemented in several different ways
  - There may exist several algorithms for solving the same problem.
    - Can be based on very different ideas and can solve the problem with dramatically different speeds

The Analysis Framework

- **Time efficiency (time complexity):** indicates how fast an algorithm runs
  - The time complexity of an algorithm is typically represented as a function of the input size
    - E.g., sorting an array of ‘n’ integers, traversing a graph of ‘V’ vertices and ‘E’ edges
  - If the input is just one element, the time complexity is represented as function of the number of bits needed to represent the input.
    - E.g., log(n) to determine whether an integer ‘n’ is prime or not.

- **Space efficiency (space complexity):** refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output
- Algorithms whose space complexity does not increase with input size (i.e., requires the same additional space irrespective of input size) are said to be ***in-place***.

Units for Measuring Running Time

- The running time of an algorithm is to be measured with a unit that is independent of the extraneous factors like the processor speed, quality of implementation, compiler and etc.
  - At the same time, it is not practical as well as not needed to count the number of times, each operation of an algorithm is performed.
- <u>Basic Operation:</u> The operation contributing the most to the total running time of an algorithm.
  - It is typically the most time consuming operation in the algorithm's innermost loop.
    - **Examples:** Key comparison operation; arithmetic operation (division being the most time-consuming, followed by multiplication)
  - We will count the number of times the algorithm's basic operation is executed on inputs of size *n*.

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation

Number of times basic operation is executed

Examples for Input Size and Basic Operations

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer *n*	*n*'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Orders of Growth

- We are more interested in the order of growth on the number of times the basic operation is executed on the input size of an algorithm.
- Because, for smaller inputs, it is difficult to distinguish efficient algorithms vs. inefficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and n^2 respectively, then
  - if $n = 3$, then we may say there is not much difference between requiring 3 basic operations and 9 basic operations and the two algorithms have about the same running time.
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is n or n^2.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3\cdot10^1$	10^2	10^3	10^3	$3.6\cdot10^6$
10^2	6.6	10^2	$6.6\cdot10^2$	10^4	10^6	$1.3\cdot10^{30}$	$9.3\cdot10^{157}$
10^3	10	10^3	$1.0\cdot10^4$	10^6	10^9		
10^4	13	10^4	$1.3\cdot10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7\cdot10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0\cdot10^7$	10^{12}	10^{18}		

Exponential-growth functions

Source: Table 2.1 From Levitin, 3rd ed.

Best-case, Average-case, Worst-case

- For many algorithms, the actual running time may not only depend on the input size; but, also on the specifics of a particular input.
  - For example, sorting algorithms (like insertion sort) may run faster on an input sequence that is *almost-sorted* rather than on a randomly generated input sequence.

- **Worst case:** $C_{worst}(n)$ – maximum number of times the basic operation is executed over inputs of size n
- **Best case:** $C_{best}(n)$ – minimum # times over inputs of size n
- **Average case:** $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Example for Worst and Best-Case Analysis: Sequential Search

- **<u>Sequential key search</u>**
- **Inputs: Array A[0…n-1], Search Key K**
- **Begin**

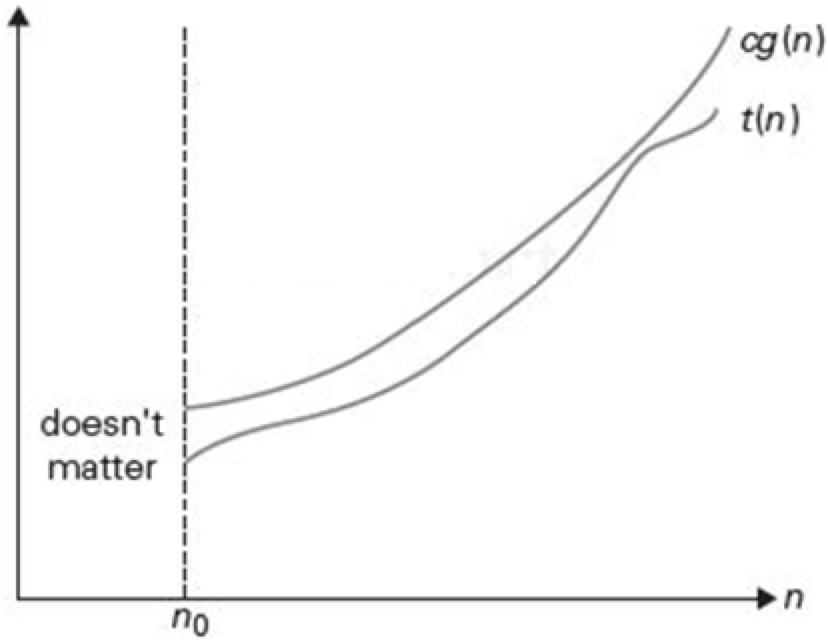
```
  for (i = 0 to n-1) do
      if (A[i] == K) then
          return "Key K found at index i"
      end if
  end for
  return "Key K not found!!"
End
```

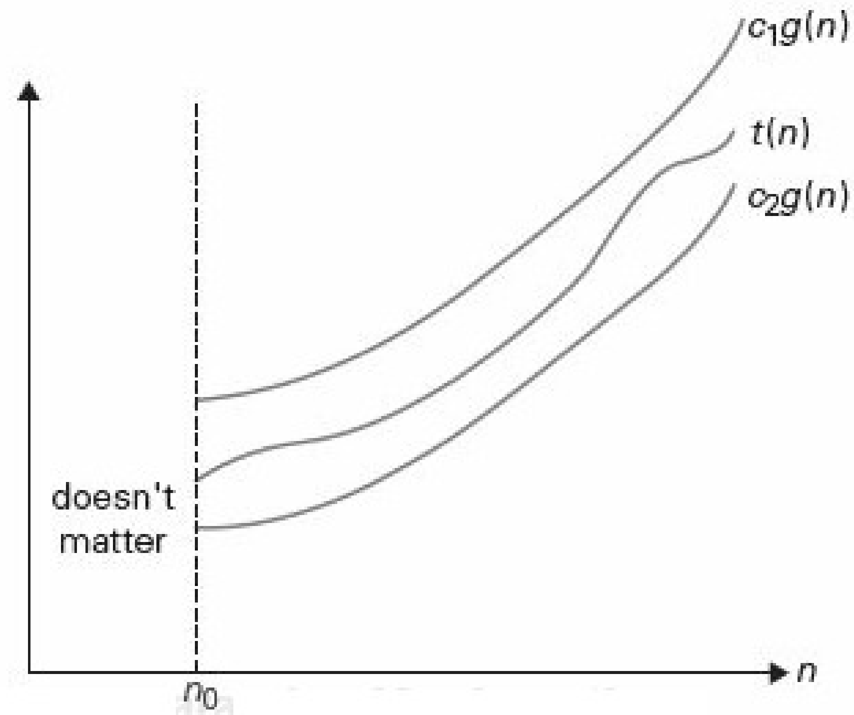
0	1	2	3	4	5	6
5	**8**	**9**	**7**	**10**	**3**	**2**

<u>Basic operation:</u> Comparison (as highlighted in red)

- <u>Worst-Case:</u> $C_{worst}(n) = n$
- <u>Best-Case:</u> $C_{best}(n) = 1$

Asymptotic Notations: Formal Intro





$t(n) = O(g(n))$

$t(n) \leq c*g(n)$ for all $n \geq n_0$

c is a positive constant (> 0)
and n_0 is a non-negative integer

$t(n) = \Theta(g(n))$

$c2*g(n) \leq t(n) \leq c1*g(n)$ for all $n \geq n_0$

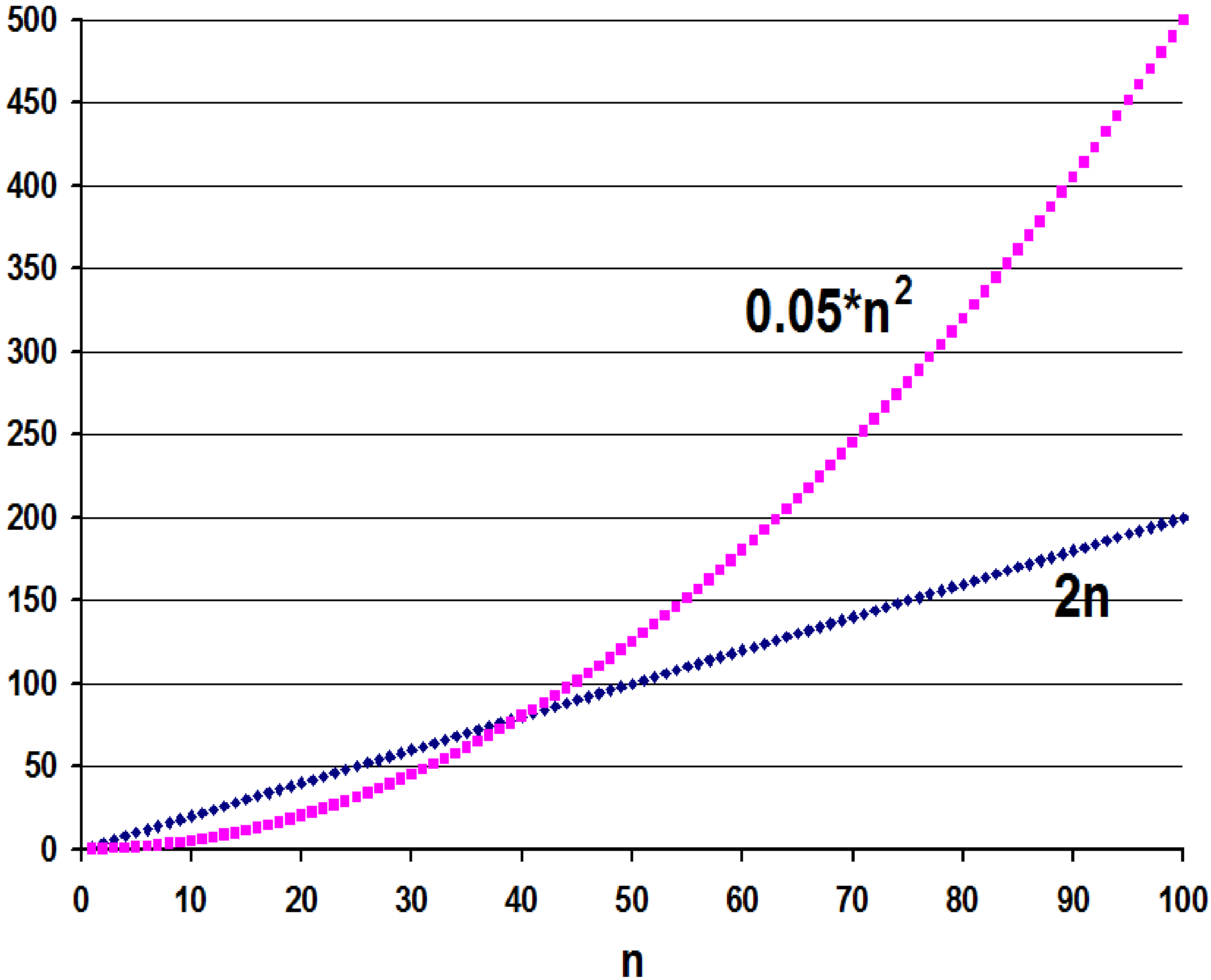
c1 and c2 are positive constants (> 0)
and n_0 is a non-negative integer

Thumb Rule for using Big-O and Big-Θ

- We say a function $f(n) = O(g(n))$ if the rate of growth of $g(n)$ is either at the same rate or faster than that of $f(n)$.
  - If the functions are polynomials, the rate of growth is decided by the degree of the polynomials.
  - Example: $2n^2 + 3n + 5 = O(n^2)$;
    $2n^2 + 3n + 5 = O(n^3)$;
  - note that, we can also come up with innumerable number of such functions for what goes inside the Big-O notation as long as the function inside the Big-O notation grows at the same rate or faster than that of the function on the left hand side.

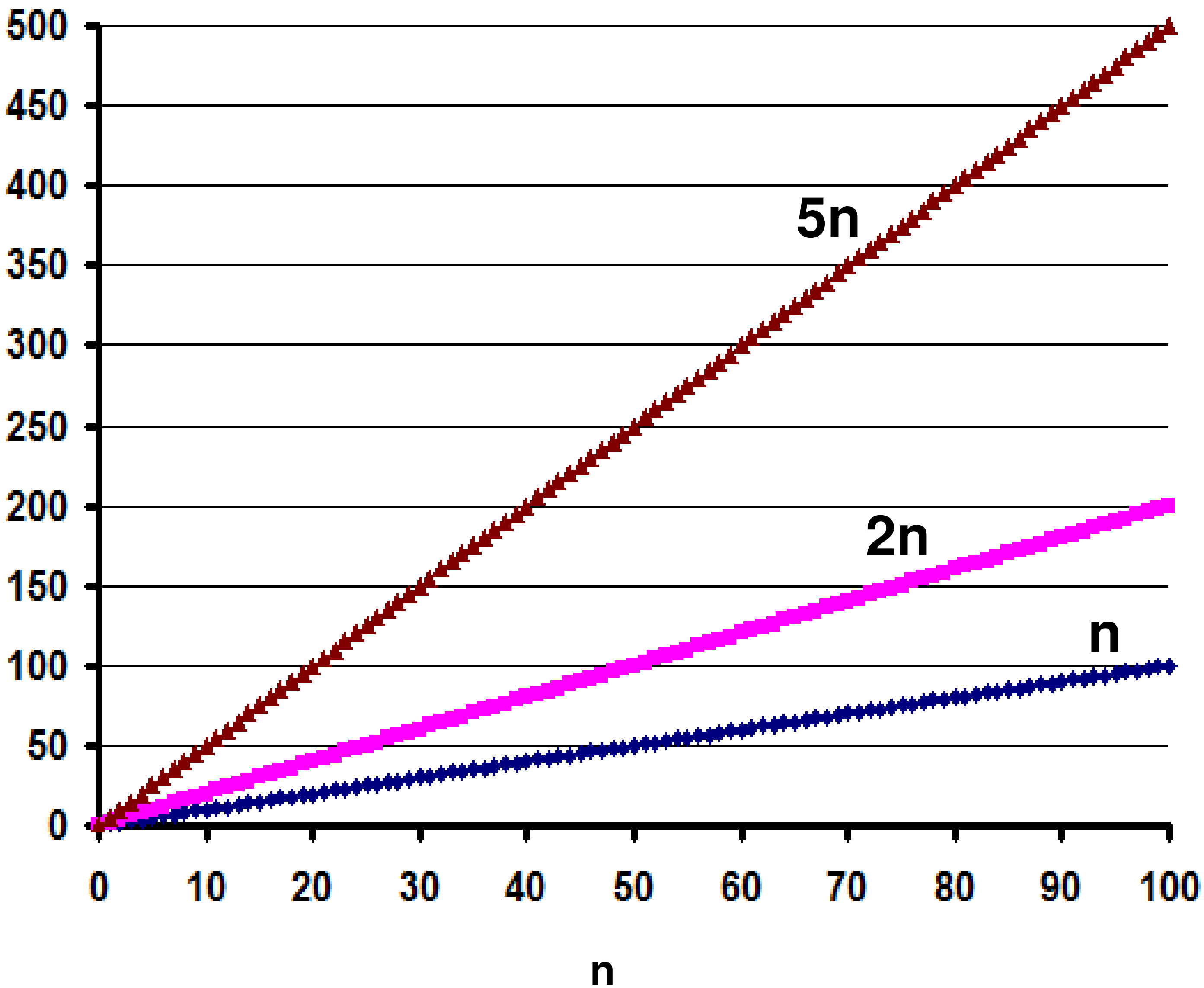
- We say a function $f(n) = \Theta(g(n))$ if both the functions $f(n)$ and $g(n)$ grow at the same rate.
  - Example: $2n^2 + 3n + 5 = \Theta(n^2)$ and not $\Theta(n^3)$;
  - For a given $f(n)$, there can be only one function $g(n)$ that goes inside the Θ-notation.

Asymptotic Notations: Example



$2n \leq 0.05\, n^2$
for $n \geq 40$
$c = 0.05$, $n_0 = 40$
$2n = O(n^2)$
More generally,
$n = O(n^2)$.

Asymptotic Notations: Example



for $n \geq 1$

$n \leq 2n \leq 5n$

$2n = \Theta(n)$

Relationship and Difference between Big-O and Big-Θ

- If $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$.
- If $f(n) = O(g(n))$, then $f(n)$ need not be $\Theta(g(n))$.
- Note: To come up with the Big-O/Θ term, we exclude the lower order terms of the expression for the time complexity and consider only the most dominating term. Even for the most dominating term, we omit any constant coefficient and only include the variable part inside the asymptotic notation.
- Big-Θ provides a tight bound (useful for precise analysis); whereas, Big-O provides an upper bound (useful for worst-case analysis).
- Examples:

  (1) $5n^2 + 7n + 2 = \Theta(n^2)$
  - Also, $5n^2 + 7n + 2 = O(n^2)$

  (2) $5n^2 + 7n + 2 = O(n^3)$,

  Also, $5n^2 + 7n + 2 = O(n^4)$, But, $5n^2 + 7n + 2 \neq \Theta(n^3)$ and $\neq \Theta(n^4)$
- The Big-O complexity of an algorithm can be technically more than one value, but the Big-Θ of an algorithm can be only one value and it provides a tight bound. For example, if an algorithm has a complexity of $O(n^3)$, its time complexity can technically be also considered as $O(n^4)$.

When to use Big-O and Big-Θ

- If the best-case and worst-case time complexity of an algorithm is guaranteed to be of a certain polynomial all the time, then we will use Big-Θ.
- If the time complexity of an algorithm could fluctuate from a best-case to worst-case of different rates, we will use Big-O notation as it is not possible to come up with a Big-Θ for such algorithms.

- **Sequential key search**
- **Inputs: Array A[0…n-1], Search Key K**
- **Begin**

```
  for (i = 0 to n-1) do
        if (A[i] == K) then
              return "Key K found at index i"
        end if
  end for
  return "Key K not found!!"
End
```

O(n) only and not Θ(n)

- **Finding the Maximum Integer in an Array**
- **Input: Array A[0…n-1]**
- **Begin**

```
      Max = A[0]
      for (i = 1 to n-1) do
        if (Max < A[i]) then
              Max = A[i]
        end if
  end for
  return Max
End
```

Θ(n) →It is also O(n)

Another Example to Decide whether Big-O or Big-Θ

Skeleton of a pseudo code

```
Input size: n
Begin Algorithm
If (certain condition) then
     for (i = 1 to n) do
       print a statement in unit time
   end for
else
     for (i = 1 to n) do
             for (j = 1 to n) do
                print a statement in unit time
             end for
     end for
End Algorithm
```

Best Case
The condition in the if block is true
-- Loop run 'n' times

Worst Case
The condition in the if block is false
-- Loop run 'n^2' times

Time Complexity: $O(n^2)$
It is not possible to come up with a Θ-based time complexity for this algorithm.

Asymptotic Notations: Examples

- Let t(n) and g(n) be any non-negative functions defined on a set of all real numbers.
- We say t(n) = O(g(n)) for all functions t(n) that have a lower or the same order of growth as g(n), within a constant multiple as n → ∞.
  - **Examples:**

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2)$$

$$n \in O(n), \qquad n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2)$$

- We say t(n) = Θ(g(n)) for all functions t(n) that have the same order of growth as g(n), within a constant multiple as n → ∞.
  - **Examples:** $an^2 + bn + c = \Theta(n^2)$;
    $n^2 + \log n = \Theta(n^2)$

Useful Property of Asymptotic Notations

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

  $$t_1(n) + t_2(n) \in O(\max\{g_1(n),\ g_2(n)\})$$

- If $t_1(n) \in \Theta(g_1(n))$ and $t_2(n) \in \Theta(g_2(n))$, then

  $$t_1(n) + t_2(n) \in \Theta(\max\{g_1(n),\ g_2(n)\})$$

Using Limits to Compare Order of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

The first case means t(n) = O(g(n)

if the second case is true, then t(n) = Θ(g(n))

The last case means g(n) = O(t(n))

L'Hopital's Rule $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

Note: t'(n) and g'(n) are first-order derivatives of t(n) and g(n)

Stirling's Formula $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Example (1)

- Let f(n) = $5n^3 + 6n + 2$. Find a function g(n) such that f(n) = O(g(n)) and f(n) ≠ Θ(g(n)). Show that your choice for g(n) is correct using the Limits approach.
- Solution:
- We need to function for g(n) that must gro faster than f(n).
- Let g(n) = n^4.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{5n^3+6n+2}{n^4} = \lim_{n\to\infty}\left(\frac{5}{n}+\frac{6}{n^3}+\frac{2}{n^4}\right) = 0$$

The limit value is 0. Hence, the denominator grows faster than the numerator.

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

$5n^3 + 6n + 2 = O(n^4)$

Example (2)

- Let $f(n) = \sqrt{5n^2 + 4n + 2}$
- Find a function g(n) such that f(n) = Θ(g(n)) using the Limits approach.
- Solution:
- The most dominating term inside the square root is the n^2 term.

$$g(n) = \sqrt{n^2} = n$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n\to\infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$ **(a non-zero constant)**

Hence, f(n) = Θ(g(n)) = Θ(n)

Example (3)

- Relate the two functions f(n) = n(n-1)/2 and g(n) = n^2 using the most appropriate asymptotic notation.
- <u>Solution:</u>

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{\left(\frac{n(n-1)}{2}\right)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)=\frac{1}{2}$$

(a non-zero constant)

Hence, f(n) = Θ(g(n)); that is, n(n-1)/2 = Θ(n^2).

Logarithm Basics

$$\log_b^n = a \Rightarrow n = b^a$$

$$\log_2^8 = 3 \Rightarrow 8 = 2^3$$

$$\log_b^{p*q} = \log_b^p + \log_b^q$$

$$\log_b^{p/q} = \log_b^p - \log_b^q$$

$$\log_b^n = \frac{\log_e^n}{\log_e^b}$$

$$\log_b^n = \frac{\log_{10}^n}{\log_{10}^b}$$

$$\log_2^n = \frac{\log_e^n}{\log_e^2}$$

$$\frac{d}{dn}\left(\log_e^{f(n)}\right) = \frac{1}{f(n)}\left(\frac{d}{dn} f(n)\right)$$

$$\frac{d}{dn}\left(\log_e^n\right) = \frac{1}{n}\left(\frac{d}{dn} n\right)$$

$$= \frac{1}{n} * 1 = \frac{1}{n}$$

$$\frac{d}{dn}\left(\log_e^{n^2}\right) = \frac{1}{n^2}\left(\frac{d}{dn} n^2\right) = \frac{1}{n^2} * 2n = \frac{2}{n}$$

Example (4)

$$f(n) = \log_2^n$$
$$g(n) = \sqrt{n}$$

- Compare the growth rate of the two functions logn and √n
- Solution:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2^n}{\sqrt{n}} = \frac{\infty}{\infty}$$

Apply L'Hopital's Rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Differentiate the numerator and denominator separately with respect to n

$$f'(n) = \frac{d}{dn}\log_2^n = \frac{d}{dn}\left(\frac{\log_e^n}{\log_e^2}\right)$$
$$= \frac{1}{\log_e^2}\frac{d}{dn}\log_e^n = \frac{1}{\log_e^2} * \frac{1}{n} = \frac{1}{n * \log_e^2}$$

$$g'(n) = \frac{d}{dn}\sqrt{n} = \frac{d}{dn} n^{1/2}$$
$$= \frac{1}{2} n^{\frac{1}{2}-1} = \frac{1}{2n^{1/2}}$$

Example (4)

$$f(n) = \log_2^n$$
$$g(n) = \sqrt{n}$$

- Compare the growth rate of the two functions logn and √n
- Solution:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\log_2^n}{\sqrt{n}} = \frac{\infty}{\infty}$$

Apply L'Hopital's Rule

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$
$$= \lim_{n\to\infty}\frac{\left(\frac{1}{n*\log_e^2}\right)}{\left(\frac{1}{2n^{1/2}}\right)} = \lim_{n\to\infty}\frac{2}{\log_e^2} * \frac{n^{1/2}}{n} = 0$$

Hence, the denominator grows faster.

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

$$\log_2^n = O\left(\sqrt{n}\right)$$

Example (5)

$$f(n) = \log^2 n$$
$$g(n) = \log n^2$$

- Compare the growth rate of the two functions $\log^2 n$ and $\log n^2$
- Solution:

$$f(n) = \log^2 n = \log n * \log n$$
$$g(n) = \log n^2 = \log(n * n)$$
$$= \log n + \log n = 2 * \log n$$

$$\log n^2 = O\left(\log^2 n\right)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n * \cancel{\log n}}{2 * \cancel{\log n}} = \infty$$

Hence, f(n) grows faster than g(n).
g(n) = O(f(n))

Time Efficiency of Non-recursive Algorithms: *General Plan for Analysis*

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*, if the number of times the basic operation gets executed varies with specific instances (inputs)
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Useful Summation Formulas and Rules

$$\sum_{i=L}^{U} 1 = (U - L + 1) \qquad \sum_{i=1}^{n} 1 = (n - 1 + 1) = n \in \Theta(n)$$

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

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{2} = \Theta(n^3)$$

In general, $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + a^2 + \ldots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad \text{for } a \neq 1$$

$$\sum_{i=0}^{10} 2^i = \frac{2^{10+1} - 1}{2 - 1} = 2048 - 1 = 2047$$

Examples on Summation

- 1 + 3 + 5 + 7 + …. + 999

$$= [1 + 2 + 3 + 4 + 5 + \ldots.\ 999] - [2 + 4 + 6 + 8 + \ldots + 998]$$

$$= \frac{999 * 1000}{2} - 2[1 + 2 + 3 + \ldots. + 499]$$

$$= 999 * 500 - 2\left[\frac{499 * 500}{2}\right] = 999 * 500 - 499 * 500$$

$$= 500*(999\text{-}499) = 500*500 = 250{,}000$$

- 2 + 4 + 8 + 16 + … + 1024

$$= 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}$$

$$= [2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{10}] - 1$$

$$= \left[\sum_{i=0}^{10} 2^i\right] - 1 = \left[2^{11} - 1\right] - 1 = \mathbf{2046}$$

$$\sum_{i=3}^{n+1} 1 = [(n+1) - 3 + 1] = n+1 - 2 = n-1 = \Theta(n)$$

$$\sum_{i=3}^{n+1} i = 3 + 4 + ... + (n+1) = [1 + 2 + 3 + 4 + ... + (n+1)] - [1 + 2]$$

$$= \frac{(n+1)(n+2)}{2} - 3 = \Theta(n^2) - \Theta(1) = \Theta(n^2)$$

$$\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1} i^2 + i = \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i$$

$$= \left[\frac{[n-1][(n-1)+1][2(n-1)+1]}{6}\right] + \left[\frac{[n-1][(n-1)+1]}{2}\right]$$

$$= \left[\frac{[n-1][n][2n-1]}{6}\right] + \left[\frac{[n-1][n]}{2}\right]$$

$$= \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$$

$$\sum_{i=0}^{n-1}(i^2+1)^2 = \sum_{i=0}^{n-1}(i^4+2i^2+1) = \sum_{i=0}^{n-1} i^4 + 2\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} 1$$

$$\in \Theta(n^5) + \Theta(n^3) + \Theta(n) = \Theta(n^5)$$

Example 1: Finding Max. Element

ALGORITHM $MaxElement(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

- The basic operation is the comparison executed on each repetition of the loop.
- In this algorithm, the number of comparisons is the same for all arrays of size n.
- The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n-1 (inclusively). Hence,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Note: Best case = Worst case for this problem

Example 2: Sequential Key Search

```
Input: Array A[0...n-1], Search Key K
Begin
        for (index i = 0 to n-1) do
                if (A[i] == K)
                        return "index i"
                end if
        end for
        return "index not found"
End
```

Basic Operation: Comparison

Asymptotic time complexity: O(n)

- Worst-Case: $C_{worst}(n) = n$
- Best-Case: $C_{best}(n) = 1$

Example 3: Element Uniqueness Problem

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

					n-2	n-1
0	1	2	3	4	5	6
5	**8**	**9**	**7**	**10**	**3**	**2**

<u>Best-case situation:</u>

If the two first elements of the array are the same, then we can exit after one comparison. Best case = 1 comparison.

<u>Worst-case situation:</u>

- The basic operation is the comparison in the inner loop. The worst-case happens for two-kinds of inputs:
  - Arrays with no equal elements
  - Arrays in which only the last two elements are the pair of equal elements

Example 3: Element Uniqueness Problem

- For these kinds of inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits i+1 and n-1; and this is repeated for each value of the outer loop i.e., for each value of the loop's variable i between its limits 0 and n-2. Accordingly, we get,

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}[(n-1)-(i+1)+1]=\sum_{i=0}^{n-2}(n-1-i-1+1)$$

$$\sum_{i=0}^{n-2}(n-1-i) = \underset{i=0}{(\text{n-1})} + \underset{i=1}{(\text{n-2})} + \ldots + \underset{i=n-2}{(\text{n-1-(n-2)})}$$

$$= (\text{n-1}) + (\text{n-2}) + \ldots + 1 = (\text{n-1})(\text{n-1+1})/2 = (\text{n-1})(\text{n})/2 \approx \text{n}^2/2$$

n(n-1)/2

Best-case: 1 comparison
Worst-case: $n^2/2$ comparisons

O(n(n-1)/2)

Asymptotic time complexity = $O(n^2)$

Example 4: Bubble Sort

- A classical sorting algorithm in which (for an array of ‘n’ elements, with indexes 0 to n-1) during the ith iteration, the (n-i-1)th largest element is bubbled all the way to its final position.
- During the ith iteration, starting from index j = 0 to n-i-2, the element at index j is compared with the element at index j+1 and is swapped if the former is larger than the latter.
  - Optimization: If there is no swap during an iteration, the array is sorted and we can stop!
- Example

	45	78	23	12	59	72
Iteration 0	45	23	12	59	72	**78**
Iteration 1	23	12	45	59	**72**	**78**
Iteration 2	12	23	45	**59**	**72**	**78**
Iteration 3	12	23	**45**	**59**	**72**	**78**

(no swap: STOP!!)

	0	**1**	**2**	**3**	**4**	**5**
	78	**72**	**59**	**45**	**23**	**12**
Iteration 0	**72**	**59**	**45**	**23**	**12**	**78**
Iteration 1	**59**	**45**	**23**	**12**	**72**	**78**
Iteration 2	**45**	**23**	**12**	**59**	**72**	**78**
Iteration 3	**23**	**12**	**45**	**59**	**72**	**78**
Iteration 4	**12**	**23**	**45**	**59**	**72**	**78**

Bubble Sort: Pseudo Code and Analysis

```
Input: Array A [0….n-1]
Begin
    for (i = 0 to n-2) do
         boolean didSwap = false
        for (j = 0 to n-i-2) do
             if A[j] > A[j+1] then
                  swap(A[j], A[j+1])
                  didSwap = true
             end if
        end for
        if (didSwap == false) then
            return; // STOP the algorithm
       end if
    end for
End
```

Best Case (array is already sorted):
1 Iteration
(i = 0): j = 0 to n-2
<u>n-1 comparisons ~ n</u>

Worst Case (array is reverse sorted): all iterations

$$\sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 1$$

$$\sum_{i=0}^{n-2} [n-i-2]-[0]+1$$

$$= \sum_{i=0}^{n-2} [n-1]-i$$

$$= (n-1)+(n-2)+\ldots+1$$

$$= \frac{n(n-1)}{2} \sim \mathbf{n^2}$$

Asymptotic time complexity = $O(n^2)$

Example 5: Insertion Sort

- Given an array A[0…n-1], at any time, we have the array divided into two parts: A[0,…,i-1] and A[i…n-1].
  - The A[0…i-1] is the sorted part and A[i…n-1] is the unsorted part.
  - In any iteration, we pick an element *v* = A[i] and scan through the sorted sequence A[0…i-1] to insert *v* at the appropriate position.
    - The scanning is proceeded from right to left (i.e., for index j running from i-1 to 0) until we find the right position for *v*.
    - During this scanning process, v = A[i] is compared with A[j].
    - If A[j] > v, then we v has to be placed somewhere before A[j] in the final sorted sequence. So, A[j] cannot be at its current position (in the final sorted sequence) and has to move at least one position to the right. So, we copy A[j] to A[j+1] and decrement the index j, so that we now compare v with the next element to the left.

$$\underbrace{A[0] \le \cdots \le A[j]}_{\text{smaller than or equal to } A[i]} < \underbrace{A[j+1] \le \cdots \le A[i-1]}_{\text{greater than } A[i]} \;\Big|\; A[i] \cdots A[n-1]$$

    - If A[j] ≤ v, we have found the right position for v; we copy v to A[j+1]. This also provides the stable property, in case v = A[j].

An Iteration of Insertion Sort

Sorted Portion | **Unsorted Portion**

0	1	2	3	4	5	6	7
12	**20**	**32**	**37**	**45**	**30**	**8**	**19**

How do we find
Where to insert so
that sorted portion
extended by one more
element stays sorted?

v = 30 @ index i = 5

				j	i		
0	1	2	3	4	5	6	7
12	**20**	**32**	**37**	**45**	**30**	**8**	**19**

			j		i		
0	1	2	3	4	5	6	7
12	**20**	**32**	**37**	**45**	**45**	**8**	**19**

		j			i		
0	1	2	3	4	5	6	7
12	**20**	**32**	**37**	**37**	**45**	**8**	**19**

	j				i		
0	1	2	3	4	5	6	7
12	**20**	**32**	**32**	**37**	**45**	**8**	**19**

	j	j+1				i	
0	1	2	3	4	5	6	7
12	**20**	**30**	**32**	**37**	**45**	**8**	**19**

Insertion Sort
Pseudo Code and Analysis

```
Input: Array A[0...n-1]
Begin
for (index i = 1 to n-1) do
    v = A[i]
    index j = i-1
    while (index j ≥ 0) do
        if (v ≥ A[j]) then
            break 'j' loop
        else      // v < A[j]
            A[j+1] = A[j]
        end if
        j = j-1
    end while
    A[j+1] = v
End
```

Best Case: If the array is already sorted For each value of index i, we just do one comparison (A[i] with A[i-1]), and decide to keep v = A[i] at its current location. Index i varies from 1 to n-1. Hence, there are 'n-1' comparisons.

Since the sub array from index 0 to i-1 is sorted, there is no way we can move 'v' further to the left, if we come across an A[j] such that v ≥ A[j]

The element A[j] is not in its final position Needs to be moved to the right

Worst Case: If the array is reverse sorted. For each value of index i, the element A[i] needs to be compared with all the values to its left (i.e., from j index i-1 to 0).

$$\sum_{i=1}^{n-1}\sum_{j=i-1}^{0} 1 = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1}(i-1)-0+1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Average Case: On average for a random input sequence, we would be visiting half of the sorted sequence A[0…i-1] to put A[i] at the proper position.

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i-1}^{(i-1)/2} 1 = \sum_{i=1}^{n-1} \frac{(i-1)}{2} + 1 = \sum_{i=1}^{n-1} \frac{(i+1)}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (i+1)$$

$$= \frac{1}{2}[2+3+\ldots+n] = \frac{1}{2}[1+2+3+\ldots+n-1] = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) = \frac{n^2+n-1}{4}$$

Example: Given sequence (also initial): **45** 23 8 12 90 21

Index -1

Iteration 1 (v = 23):

45	**45**	8	12	90	21
23	**45**	8	12	90	21

Iteration 2 (v = 8):

23	**45**	**45**	12	90	21
23	**23**	**45**	12	90	21
8	**23**	**45**	12	90	21

Iteration 3 (v = 12):

8	**23**	**45**	**45**	90	21
8	**23**	**23**	**45**	90	21
8	**12**	**23**	**45**	90	21

Iteration 4 (v = 90):

8	**12**	**23**	**45**	90	21
9	**12**	**23**	**45**	**90**	21

Iteration 5 (v = 21):

9	**12**	**23**	**45**	**90**	**90**
9	**12**	**23**	**45**	**45**	**90**
9	**12**	**23**	**23**	**45**	**90**
9	**12**	**21**	**23**	**45**	**90**

Considering the Best case and Worst case

Asymptotic time complexity = $O(n^2)$

The **colored** elements are in the sorted sequence and the circled element is at index *j* of the algorithm.

Property: It takes $\lceil \log_k^n \rceil = \Theta(\log n)$ steps to 'k-tuple' an integer from 1 to the smallest value that is greater than or equal to n.

- Verification (k=2): It takes $\lceil \log_2^n \rceil$ steps to <u>double</u> an integer from 1 to the smallest value that is greater than or equal to n.
  - Example: Let n = 30
  - Initial: j = 1
  - Step 1: j = j * 2 = 1 * 2 = 2
  - Step 2: j = j * 2 = 2 * 2 = 4
  - Step 3: j = j * 2 = 4 * 2 = 8
  - Step 4: j = j * 2 = 8 * 2 = 16
  - **Step 5: j = j * 2 = 16 * 2 = 32**
- Note: $\lceil \log_2^{30} \rceil = 5$

- Verification (k=3): It takes $\lceil \log_3^n \rceil$ steps to <u>triple</u> an integer from 1 to the smallest value that is greater than or equal to n.
  - Example: Let n = 30
  - Initial j = 1
  - Step 1: j = j * 3 = 1 * 3 = 3
  - Step 2: j = j * 3 = 3 * 3 = 9
  - Step 3: j = j * 3 = 9 * 3 = 27
  - **Step 4: j = j * 3 = 27 * 3 = 81**
- Note: $\lceil \log_3^{30} \rceil = 4$

Example 6 (1): Logarithmic Time Complexity Analysis

```
Input: n
k  = 0
for ( i = n/2; i ≤ n; i++) {
    for ( j = 1; j ≤ n; j = j * 2 ) {
        k = k + n/2
    }
}
```

```
Input: n

k  = 0
for ( i = n/2; i ≤ n; i++) {
    for ( j = 2; j ≤ n; j = j * 2 ) {
        k = k + n/2
    }
}
```

Basic operation: The division '/' inside the inner loop

We know the j-loop will run Θ(logn) times for a particular value of i.

 # times the basic operation is executed

Still, for a particular value of i, we can say that the j-loop will run Θ(logn) times (even though the starting value for j is 2 and not 1) and the whole algorithm will run in Θ(nlogn) time

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

Example 6 (2): Logarithmic Time Complexity Analysis

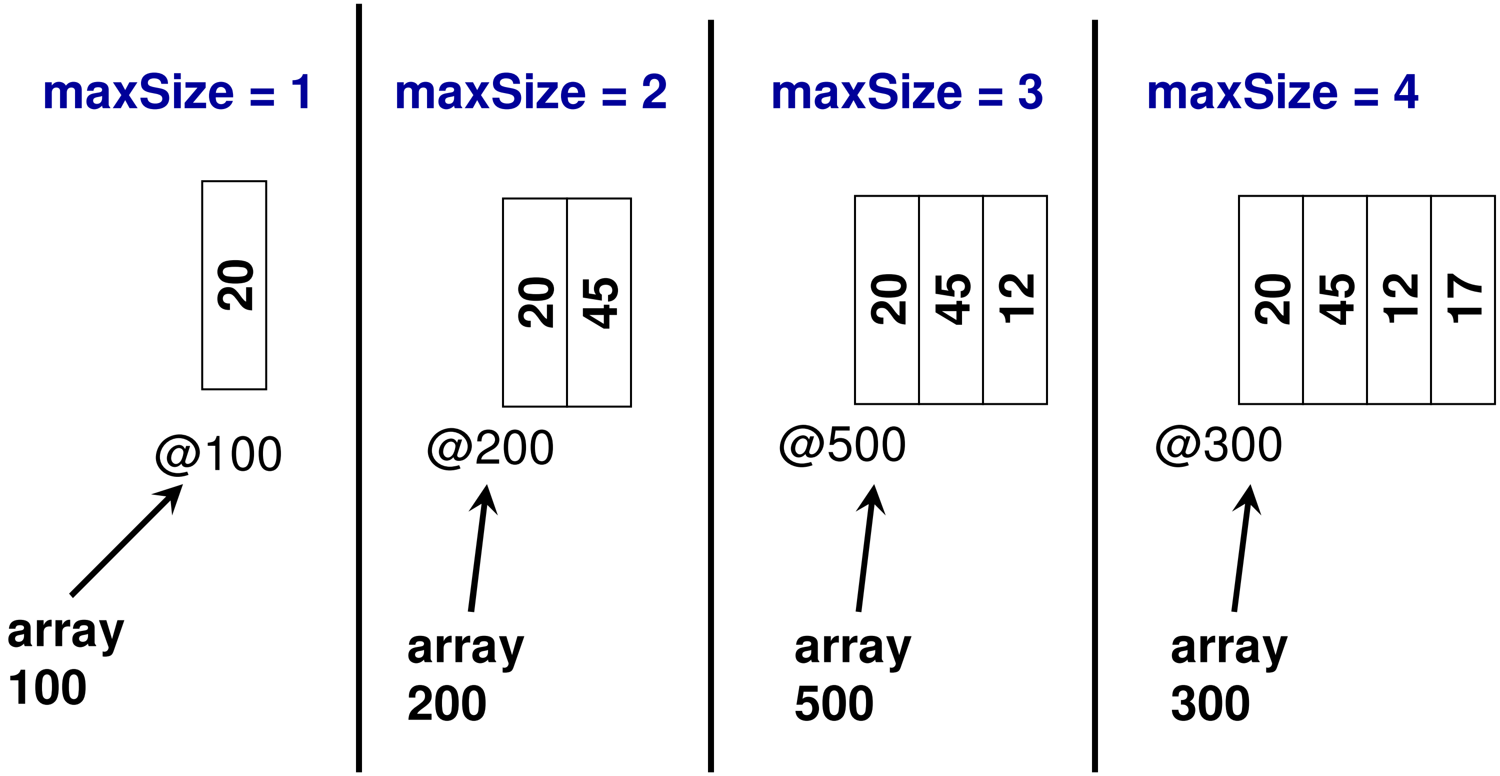
```
Input: n
a = 0; j = n

while ( j > 0) do
    a = a + j
    j = j / 2
end while
```

The property can also be applied for division: It takes ϴ(logn) steps to reduce an integer by a factor of 1/k in each step, all the way to 1.

Example 7: Space Complexity Analysis with Dynamic List ADT

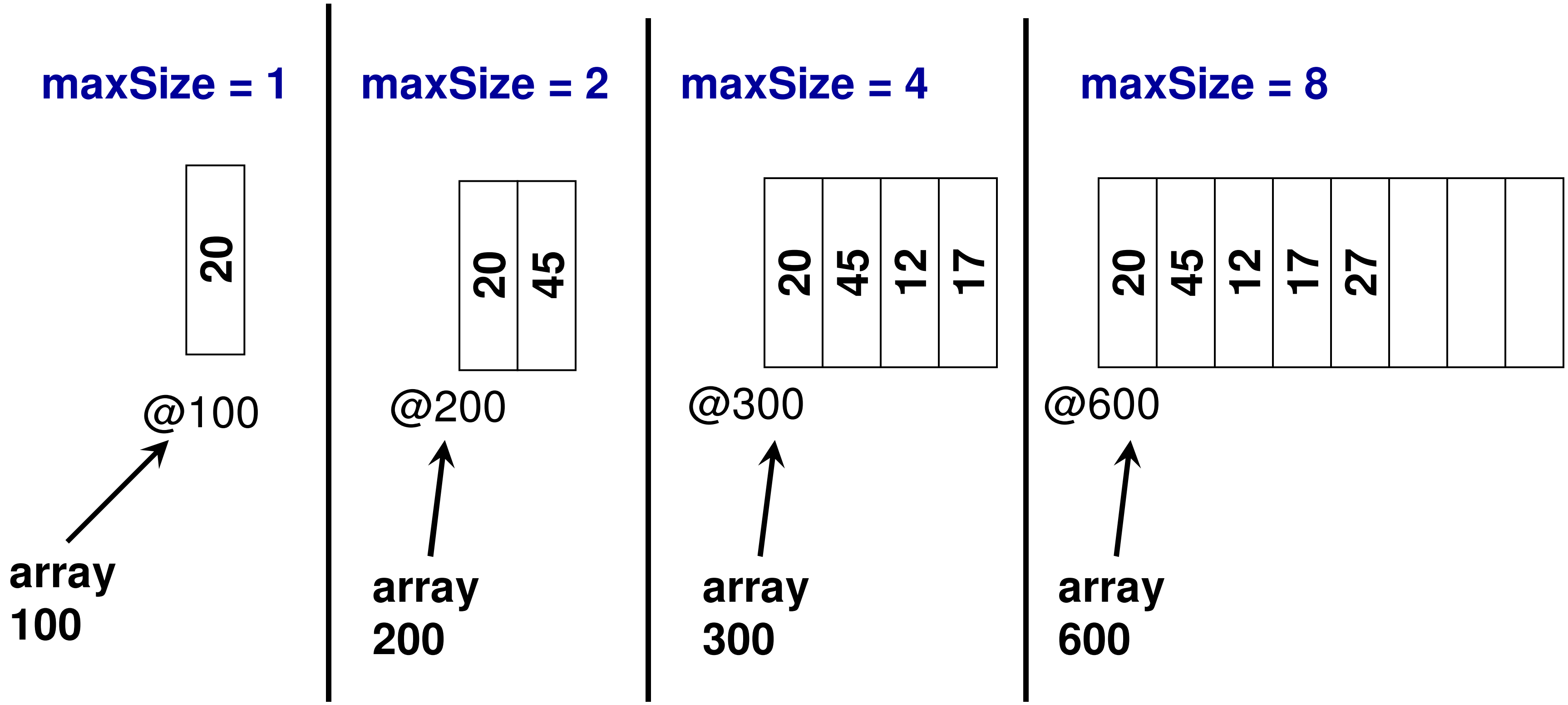
- Consider a dynamic List ADT implemented using arrays.
- Analyze the space complexity involved in incrementing the size of the array by 1 and doubling the size of the array by 2, whenever needed.

Incrementing the array size by one



memory cells allocated to eventually store a list of 4 elements is
1 + 2 + 3 + 4 = 10;
For 8 elements: 1 + 2 + 3 + …. + 8 = 8*(8+1)/2 = 36
In general, $1 + 2 + 3 + 4 + \ldots + n = n(n+1)/2 = \Theta(n^2)$

Doubling the array size by 2



 # memory cells allocated to eventually store a list of 8 elements is:
$1 + 2 + 4 + 8 = 15$
In general, the number of cells to store at most 'n' elements, where 'n' is a perfect square of 2; i.e., $n = 2^k$, where k is an integer >= 0; $k = \log_2(n)$
$1 + 2 + 4 + 8 + \ldots + n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^k$, where $k = \log_2(n)$
$= 2^{(k+1)} - 1 = 2*2^k - 1 = 2n - 1 = \Theta(n)$

Time Efficiency of Recursive Algorithms: *General Plan for Analysis*

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Recursive Evaluation of n!

Definition: $n! = 1 * 2 * \ldots *(n-1) * n$ for $n \geq 1$ and $0! = 1$

- Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$ and

  $F(0) = 1$

YouTube Link: https://www.youtube.com/watch?v=K25MWuKKYAY

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

$M(0) = 0$

the calls stop when $n = 0$ — no multiplications when $n = 0$

Hint: To find the upper limit for i, put **n-i** is equal to the value of of n in the basic condition; in this case it is 0

M(n-1) = M(n-2) + 1; **M(n-2) = M(n-3)+1**

M(n) = [M(n-2)+1] + 1 = M(n-2) + 2 = [M(n-3)+1+2] = M(n-3) + 3 = M(n-i) + i
for 0 ≤ i ≤ n

Put i = n; M(n) = M(n-n) + n = M(0) + n = 0 + n = n

Overall time Complexity: Θ(n)

Counting the # Bits of an Integer

ALGORITHM $BinRec(n)$
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

bits (n) = # bits($\lfloor n/2 \rfloor$) + 1; for n > 1
bits (1) = 1

1	1 bit
2-3	2 bits
4-7	3 bits
8-15	4 bits
16-31	5 bits
32-63	6 bits

Either Division or Addition could be considered the Basic operation, as both are executed once for each recursion. We will treat "addition" as the basic operation.

Let A(n) be the number of additions needed to compute # bits(n)

 # Additions $A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$

Since the recursive calls end when *n* is equal to 1 and there are no additions made, the initial condition is: A(1) = 0.

Counting the # Bits of an Integer

Solution Approach: If we use the backward substitution method (as we did in the previous two examples, we will get stuck for values of n that are not powers of 2).

We proceed by setting $n = 2^k$ for $k \geq 0$.

New recurrence relation to solve:

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots\\
&\ldots\\
&= A(2^{k-i}) + i\\
&\ldots\\
&= A(2^{k-k}) + k.
\end{aligned}$$

To find the upper limit for i

Put $2^{k-i} = 2^0$

$k-i = 0$

$i = k$

$$\boxed{A(n) = \log_2 n \in \Theta(\log n).}$$

Examples for Solving Recurrence Relations

X(n) = X(n-1) + 5, for n > 1, X(1) = 0

$$
\begin{aligned}
x(n) &= x(n-1)+5 \\
&= [x(n-2)+5]+5 = x(n-2)+5\cdot 2 \\
&= [x(n-3)+5]+5\cdot 2 = x(n-3)+5\cdot 3 \\
&= \ldots \\
&= x(n-i)+5\cdot i \\
&= \ldots \\
&= x(1)+5\cdot(n-1) = 5(n-1).
\end{aligned}
$$

Put n – i = 1
i = n – 1
X(n-i) = X(1)

= Θ(n)

X(n) = 3*X(n-1) for n > 1, X(1) = 4

$$
\begin{aligned}
x(n) &= 3x(n-1) \\
&= 3[3x(n-2)] = 3^2x(n-2) \\
&= 3^2[3x(n-3)] = 3^3x(n-3) \\
&= \ldots \\
&= 3^ix(n-i) \\
&= \ldots \\
&= 3^{n-1}x(1) = 4\cdot 3^{n-1}.
\end{aligned}
$$

Put n – i = 1
i = n – 1

= (4/3)3^n = Θ(3^n)

X(n) = X(n/3) + 1 for n > 1, X(1) = 1 [Solve for n = 3^k]

$$
\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= \ldots \\
&= x(3^{k-i}) + i \\
&= \ldots \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}
$$

Put $3^{k-i} = 3^0$

$k - i = 0$

$i = k$

X(n) = Θ(logn)

Space-Time Tradeoff

In-place vs. Out-of-place Algorithms

- An algorithm is said to be "in-place" if the amount of additional memory required by the algorithm does not grow with increase in the input size.
  - For example, algorithms like Insertion Sort and Bubble Sort are in-place because the amount of additional memory (like the use of temporary variables) needed by these algorithms does not grow with input size.
- In-place algorithms are said to have $\Theta(1)$ space complexity.

- An algorithm is said to be "out-of-place" if the amount of additional memory required by the algorithm grows with increase in input size.
  - For example, if an algorithm copies the contents of the input array to another new array, then the amount of additional memory (to be allocated for the new array) grows with increase in the size of the input array. E.g., Merge Sort (Module 2).

Hash table

- Maps the elements (values) of a collection to a unique key and stores them as key-value pairs.
- Hash table of size m (where m is the number of unique keys, ranging from 0 to m-1) uses a hash function H(v) = v mod m
- The hash value (a.k.a. hash index) for an element v is H(v) = v mod m and corresponds to one of the keys of the hash table.
- The size of the Hash table is typically a prime integer.
- Example: Consider a hash table of size 7. Its hash function is H(v) = v mod 7.
- Let an array A = {45, 67, 89, 45, 85, 12, 88, 90, 13, 14}

Value, v	45	67	89	45	85	12	88	90	13	14
H(v) = v mod 7	3	4	5	3	1	5	4	6	6	0

0	**1**	**2**	**3**	**4**	**5**	**6**
↓	↓		↓	↓	↓	↓
14	85		45	67	89	90
			45	88	12	13

We will implement Hash table as an array of singly linked lists

Space-Time Tradeoff

- Note: At the worst case, there could be only one linked list in the hash table (i.e., all the elements map to the same key).
- On average, we expect the ‘n’ elements to be evenly divided across the ‘m’ keys, so that the length of a linked list is n/m. Nevertheless, for a hash table of certain size (m), ‘n’ is the only variable.
- Space complexity: $\Theta(n)$
  - For an array of ‘n’ elements, we need to allocate space for ‘n’ nodes (plus the ‘m’ head nodes) across the ‘m’ linked lists.
  - Since usually, n >> m, we just consider the overhead associated with storing the ‘n’ nodes
- Time complexity:
  - Insert/Delete/Lookup: O(n), we may have to traverse the entire linked list
  - isEmpty: O(m), we have to check whether each index in the Hash table has an empty linked list or not.

Example: Number of Comparisons

Array, A = {45, 23, 11, 78, 89, 44, 22, 28, 41, 30}
H(v) = v mod 7

Hash table

0	1	2	3	4	5	6	Successful Search, # comparisons
28	78	23	45	11	89	41	1
	22	44					2
		30					3

Average Number of Comparisons for a Successful Search (Hash table)

$$= \frac{(7*1) + (2*2) + (1*3)}{10} = \frac{14}{10} = \mathbf{1.4}$$

Worst Case Number of Comparisons for a Successful Search (Hash table) = 3

Worst Case Number of Comparisons for an Unsuccessful Search (Hash table) = 3

The worst case number of comparisons for both successful and unsuccessful searches depends on the length of the longest linked list.

Example: Number of Comparisons

Array, A = {45, 23, 11, 78, 89, 44, 22, 28, 41, 30}
H(v) = v mod 7

Hash table

0	1	2	3	4	5	6	Successful Search, # comparisons
28	78	23	45	11	89	41	1
	22	44					2
		30					3

Average Number of Comparisons for a Successful Search (Hash table)

$$= \frac{(7*1) + (2*2) + (1*3)}{10} = \frac{14}{10} = \mathbf{1.4}$$

Average Number of Comparisons for a Successful Search (Array)

$$= \frac{1 + 2 + 3 + \ldots + 10}{10} = \frac{10*11/2}{10} = \mathbf{5.5}$$

Worst Case Number of Comparisons	**Hash table**	**Array**
For a Successful Search	**3**	**10**
For an unsuccessful Search	**3**	**10**

Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- Given an array A of unique integers, we want to find the contiguous subsequences of length 2 or above as well as the length of the largest subsequence.
- Assume it takes Θ(1) time to insert or search for an element in the hash table.

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7

0	1	2	3	4	5	6
56	36	44		32	33	41
35	92					34
42	43					

36	41	56	35	44	33	34	92	43	32	42
35	40	55	34	43	32	33	91	42	31	41
	42	57					93		33	
	43								34	
	44								35	
	45								36	
									37	

	41	**56**					**92**		**32**	
	42								**33**	
	43								**34**	
	44								**35**	
									36	

Applications of Hashing (1)

Finding Consecutive Subsequences in an Array

- Algorithm

```
Insert the elements of A in a hash table H
Largest Length = 0
for i = 0 to n-1 do
   if (A[i] – 1 is not in H) then
         j = A[i]   // A[i] is the first element of a possible cont. sub seq.
         j = j + 1
         while ( j  is in H) do
             j = j + 1
         end while
         if ( j – A[i] > 1) then  // we have found a cont. sub seq. of length > 1
              Print all integers from A[i] … (j-1)
               if (Largest Length < j – A[i]) then
                        Largest Length = j – A[i]
             end if
         end if
   end if
 end for
```

L searches in the Hash table H for sub sequences of length L

Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- **Time Complexity Analysis**
- For each element at index i in the array A we do at least one search (for element A[i] – 1) in the hash table.
- For every element that is the first element of a sub seq. of length 1 or above (say length L), we do L searches in the Hash table.
- The sum of all such Ls should be n.
- For an array of size n, we do n + n = 2n = Θ(n) hash searches. The first 'n' corresponds to the sum of all the lengths of the contiguous sub sequences and the second 'n' is the sum of all the 1s (one 1 for each element in the array)

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7

0	1	2	3	4	5	6
56	36	44		32	33	41
35	92					34
42	43					

36	41	56	35	44	33	34	92	43	32	42
35	40	55	34	43	32	33	91	42	31	41
	42	57					93		33	
	43								34	
	44								35	
	45								36	
									37	