Data structures and Algorithms

ALGORITHM EFFICIENCY

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

Outline

- Algorithm: A brief review
- An analysis of algorithm efficiency
  - Measuring the efficiency of algorithms
  - Asymptotic notations: Big-O, Big-Ω and Big-Θ
- Algorithm analysis: Examples

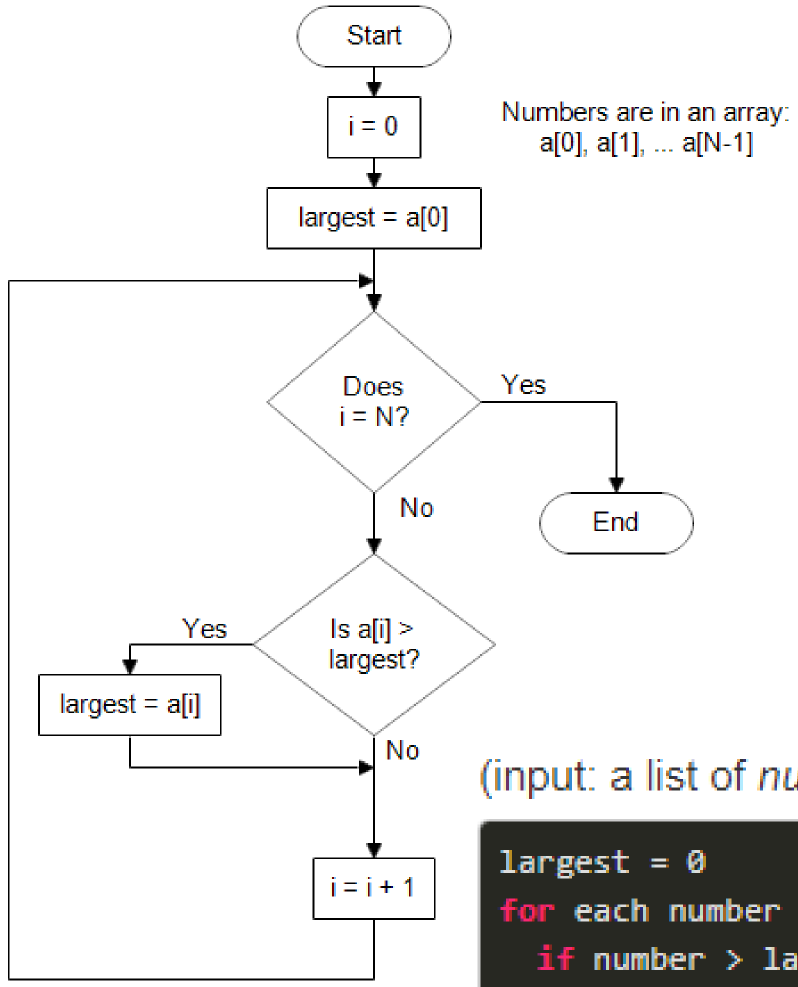
Algorithm: A brief review

What is an algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem. (*Introduction to the Design and Analysis of Algorithms, 3rd edition*)
  - We can always obtain a required output for any legitimate input in a finite amount of time.

- We can design algorithms by using verbal descriptions, pseudo-code, and flowcharts.

Example: Find the largest number in a series of numbers



1. Take a paper and pencil and put a 0 on it. Name the number stored on the paper *largest*.
2. Compare each number in the submitted numbers with largest. If any number is greater than largest, erase largest and replace it with number.
3. When you finish going through all the numbers, the value of largest will be the greatest number.

(input: a list of *numbers*)

```
largest = 0
for each number in numbers:
  if number > largest:
    largest = number   //(this assigns the value of number to largest)
return largest
```

Checkpoint 01a: Read and Understand an algorithm

Input: A list of numbers, {1, 2, 3, 5}

Output: What do you think about the output?

```
ttl = 0    // assigns value of 0 to ttl
for each number in numbers:
    ttl = (ttl + number)   // assigns new value to ttl
return ttl
```

1 + 2 + 3 + 5 = 11

Checkpoint 01b: Read and Understand an algorithm

Input: No

Output: What do you think about the output?

Step 1. Assign sum = 0 and i = 0.

Step 2.

- Assign i = i + 1
- Assign sum = sum + i

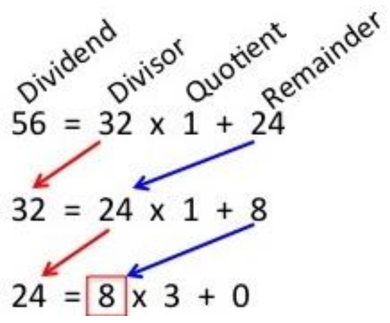
Step 3. Compare i with 10

- If i < 10, back to Step 2.
- Otherwise, if i ≥ 10, go to Step 4.

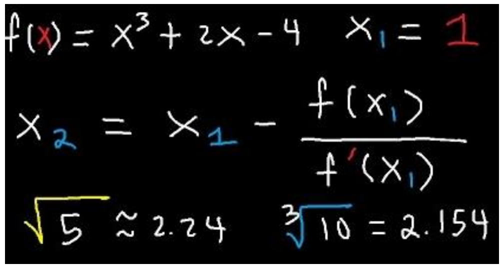
Step 4. Return sum

```
0 + 1 + … + 9
  + 10 = 55
```

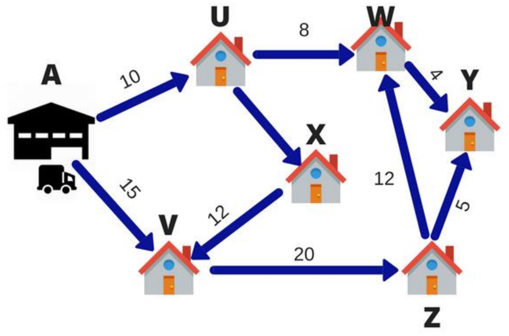
Common algorithms in practice



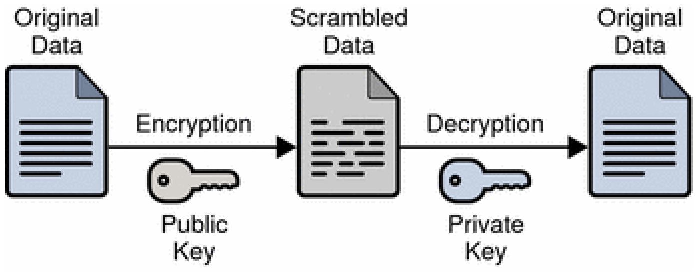
Euclidean algorithm to find the GCD of two integers



Newton's method to find the root of a number

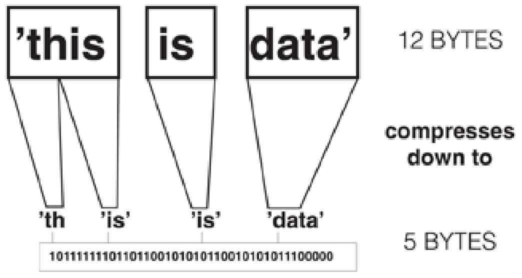


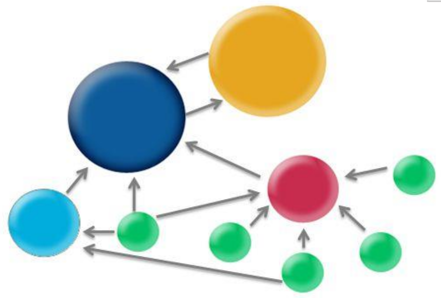
Shortest path algorithms (Dijkstra, Bellman-Ford)



Data encryption (DES, RSA…)

Data compression (Huffman, Lempel-Ziv, GIF, MPEG,…)





Website ranking in search engine (PageRank, HITS)

Characteristics of algorithms

Finiteness	• For any input, the algorithm terminates after a finite number of steps.
Correctness	• Always correct. • Give the same result for different run time.
Definiteness	• All the steps of the algorithm must be precisely defined.
Effectiveness	• We can always perform each step of the algorithm correctly in a finite amount of time.

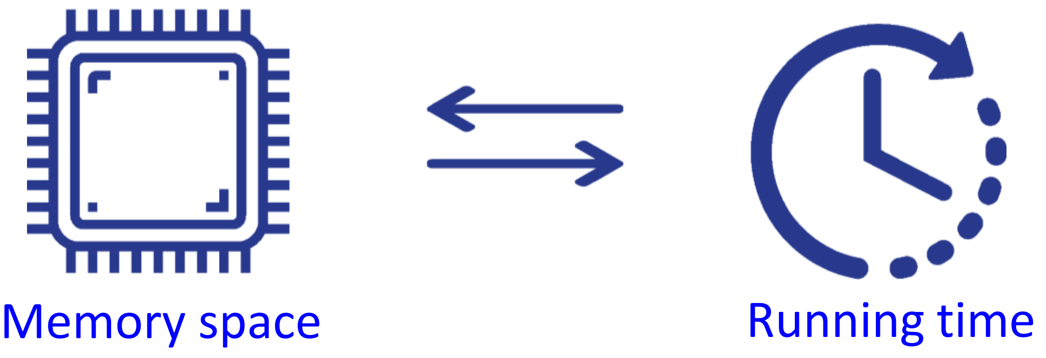
Algorithm efficiency

Analysis of algorithms

- It is an area of computer science that provides tools for contrasting the efficiency of different algorithms *(Data Abstraction & Problem Solving with C++ 6th Edition)*
- A comparison of algorithms should focus on the significant differences in efficiency.
  - We usually obtain those differences through superior algorithms rather than clever tricks in coding.

Analysis of algorithms

- The speed and memory of computers are primarily important considerations from a practical point of view.



- An analysis of algorithms examines an algorithm’s efficiency with respect to those two resources.

Example: Which algorithm is better for computing factorial?

Algorithm 1

```
int factorial1 (int n){
   if (n <= 1)
       return 1;
   return n * factorial(n-1);
}
```

Algorithm 2

```
int factorial2 (int n) {
   if (n <= 1)
      return 1;
   fact = 1;
   for (k = 2; k <= n; k++)
      fact *= k;
   return fact;
}
```

Comparing two algorithms

- How to compare the time efficiency of two algorithms that solve the same problem?
  - Implement those algorithms (into programs),
  - Calculate the execution time of those programs, and
  - Compare those time values
- Difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?
- Any analysis of efficiency must be independent of data, computer configuration, or implementations.

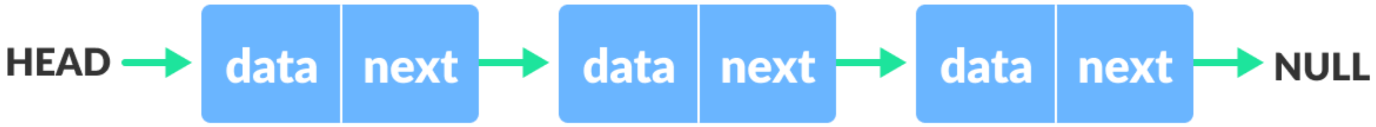
Algorithm efficiency

- Time efficiency indicates how fast an algorithm runs.
  - Time is measured by counting the number of **basic operations**.
- Space efficiency refers to the amount of memory required by an algorithm (in addition to the space needed for its input and output).
  - Space is measured by counting the maximum memory space required by the algorithm.
- In this lecture, we primarily concentrate on **time efficiency**.

The execution time of algorithms

- The execution time of an algorithm is usually related to the number of operations that algorithm requires.
- This is usually expressed in terms of the number, n, of items the algorithm must process.

Example: Traversal of linked nodes



```
Node<ItemType>* curPtr = headPtr;          ← 1 assignment
while (curPtr != nullptr)                  ← n + 1 comparisons
{
    cout << curPtr->getItem() < endl;      ← n writes
    curPtr = curPtr->getNext();            ← n assignments
}   // end while
```

If each assignment, comparison, and write operation requires a, c, and w time units, respectively,

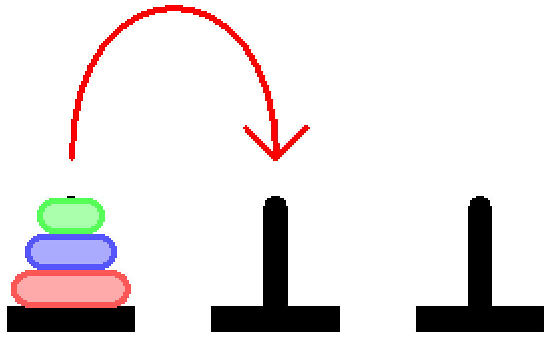
Then the statements require $(\boldsymbol{n}+\mathbf{1}) \times (\boldsymbol{a}+\boldsymbol{c}) + \boldsymbol{n} \times \boldsymbol{w}$ time units.

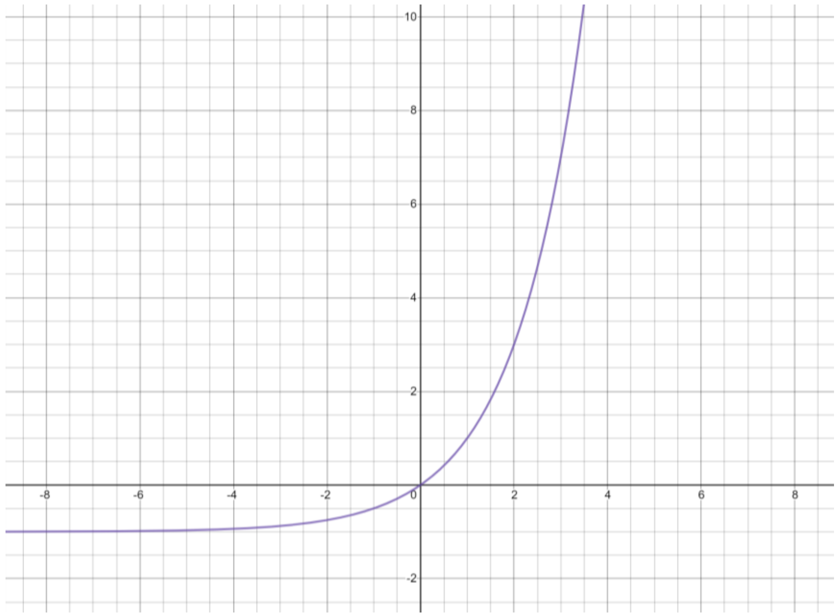
Thus, the time required to write n nodes is proportional to n.

Example: The Towers of Hanoi

The solution to the Towers of Hanoi problem with n disks requires $(2^n - 1)$ moves.

If each move requires the same time m, the solution requires $(\mathbf{2}^{\boldsymbol{n}} - \mathbf{1}) \times \boldsymbol{m}$ time units.





This time requirement increases rapidly as the number of disks increases.

Example: Nested loops

Consider an algorithm that contains nested loops of the following form

```
for ($i = 1$ through $n$)
    for ($j = 1$ through $i$)
        for ($k = 1$ through $5$)
            Task $T$
```

If task T requires t time units,

Then the innermost loop on k requires $5 \times t$ time units.

The middle loop on j requires $5 \times t \times i$ time units,

And the outermost loop on i requires

$$\sum_{i=1}^{n} (5 \times t \times i) = 5 \times t \times (1 + 2 + \cdots + n) = \mathbf{5 \times \boldsymbol{t} \times \boldsymbol{n} \times (\boldsymbol{n} + 1)/2} \quad \text{time units}$$

The execution time of algorithms

- You may want to count the number of times that each operation of the algorithm is executed

```
double x = 2.5;
double y = 3.0;
for (int i = 0; i < n; i++) {
    a[i] = x * y;
    x = 2.5 * x;
    y = y + a[i];
}
```

1 assignment

1 assignment

1 assignment (i=0) + (n+1) test (i<n) + (1 addition 1 assignment)$\times$n for (i++)

3 assignments + 2 multiplications + 1 addition for the loop body $\rightarrow$ 6n

The total number of operations is **$6 \times n + 2 \times n + (n + 1) + 3 = 9 \times n + 4$**

- This is both excessively difficult and usually unnecessary.

The execution time of algorithms

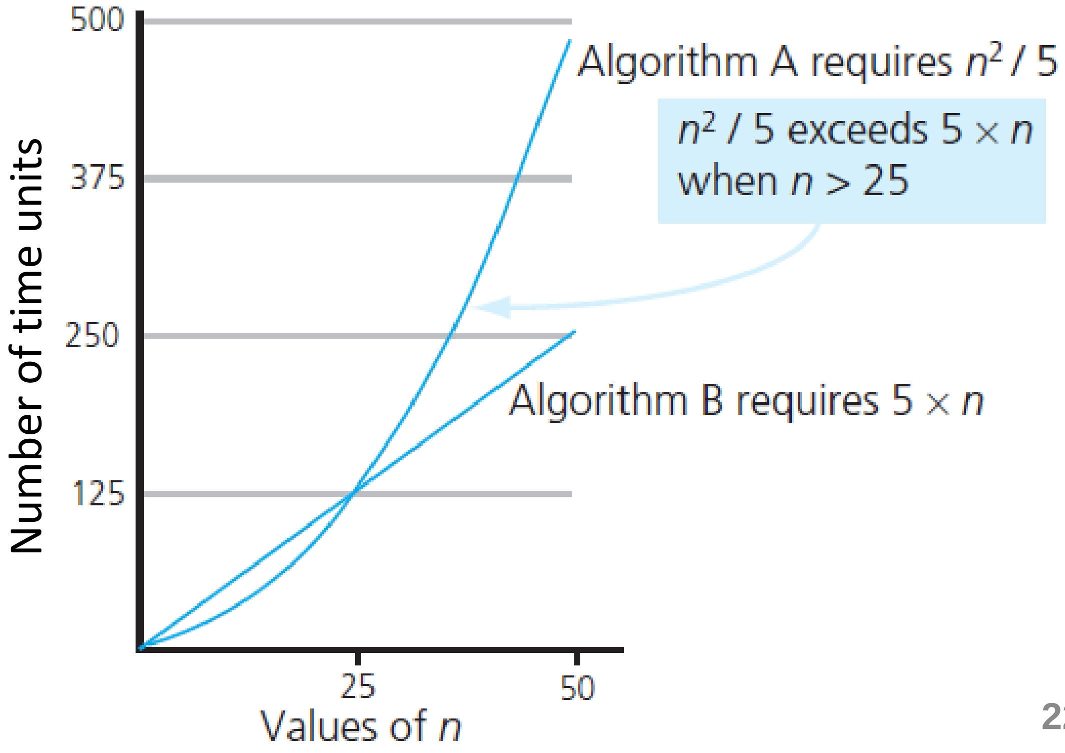
- The time efficiency of an algorithm is measured by counting the number of times the basic operation(s) is executed.

Basic operation

- It is usually the most time-consuming operation.
- It is usually in the algorithm’s innermost loop.
- It usually relates to the data that needs to be processed.

Algorithm growth rates

- An algorithm's time requirement is measured as a function of the problem size, which is application-dependent.
  - E.g., number of nodes in a linked chain, number of disks in the Towers of Hanoi, etc.



Analysis and Big-O notation

- An algorithm that requires time proportional to $g(n)$ is said to be order $g(n)$, which is denoted as $\boldsymbol{O(g(n))}$.

Growth-rate function

Big-O notation

- Algorithm $\boldsymbol{A}$ is $O(g(n))$ if some positive constants k and non-negative integer n_0 exist such that $\boldsymbol{A}$ requires no more than $k \times g(n)$ time units to solve a problem of size $n \geq n_0$.

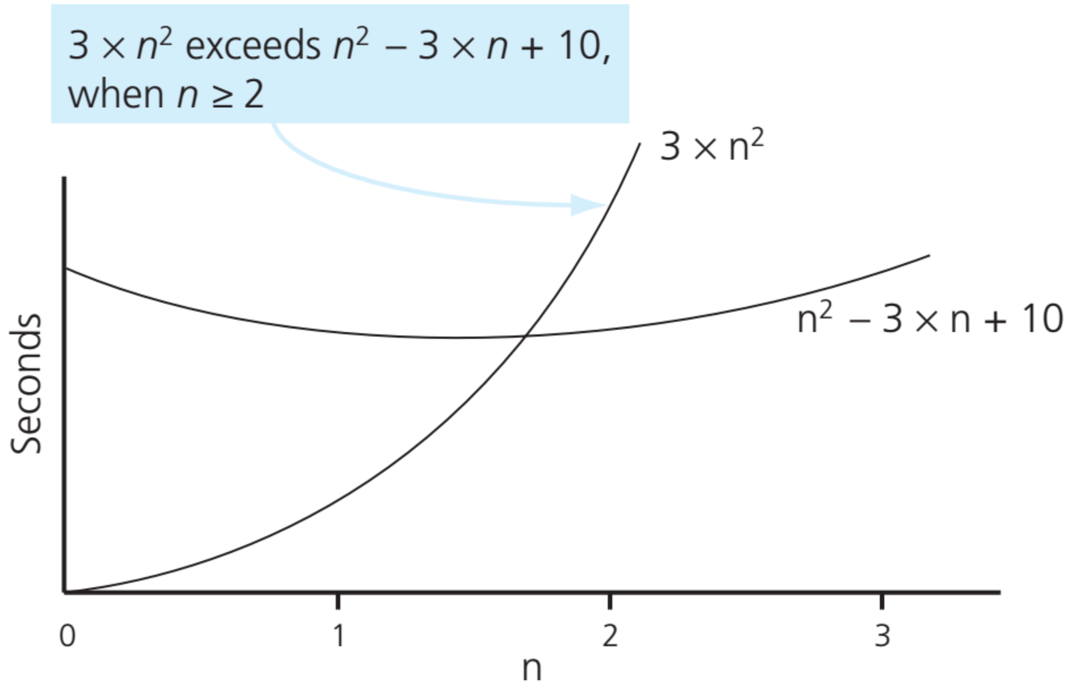
Example: Find the values of k and n_0

Suppose that an algorithm requires $n^2 - 3 \times n + 10$ seconds to solve a problem of size n.

There exists $k = 3$ and $n_0 = 2$ such that $\mathbf{3n^2} > n^2 - 3 \times n + 10$

Thus, the algorithm requires no more than $k \times n^2$ time units for $n \geq n_0$.

Therefore, $\boldsymbol{O(n^2)}$.



Example: Find the values of k and n_0

Traversal of linked nodes. Displaying a linked chain's first n items requires $(n+1)\times(a+c)+n\times w$ time units. Then we have

$$(\mathbf{2}\times \boldsymbol{n})\times(\boldsymbol{a}+\boldsymbol{c})+\boldsymbol{n}\times\boldsymbol{w}\geq(n+1)\times(a+c)+n\times w \text{ for all } n\geq 1$$

After factoring n on the left side of the inequality, we have

$$(\mathbf{2}\times\boldsymbol{a}+\mathbf{2}\times\boldsymbol{c}+\boldsymbol{w})\times\boldsymbol{n}\geq(n+1)\times(a+c)+n\times w \text{ for all } n\geq 1$$

Thus, this task is $\boldsymbol{O}(\boldsymbol{n})$. Here, k is $2\times a+2\times c+w$ and n_0 is 1.

The Towers of Hanoi. This problem requires $(2^n-1)\times m$ time units. Because $m\times 2^n\geq(2^n-1)\times m$ for $n\geq 1$, the solution is $\boldsymbol{O}(\mathbf{2}^{\boldsymbol{n}})$.

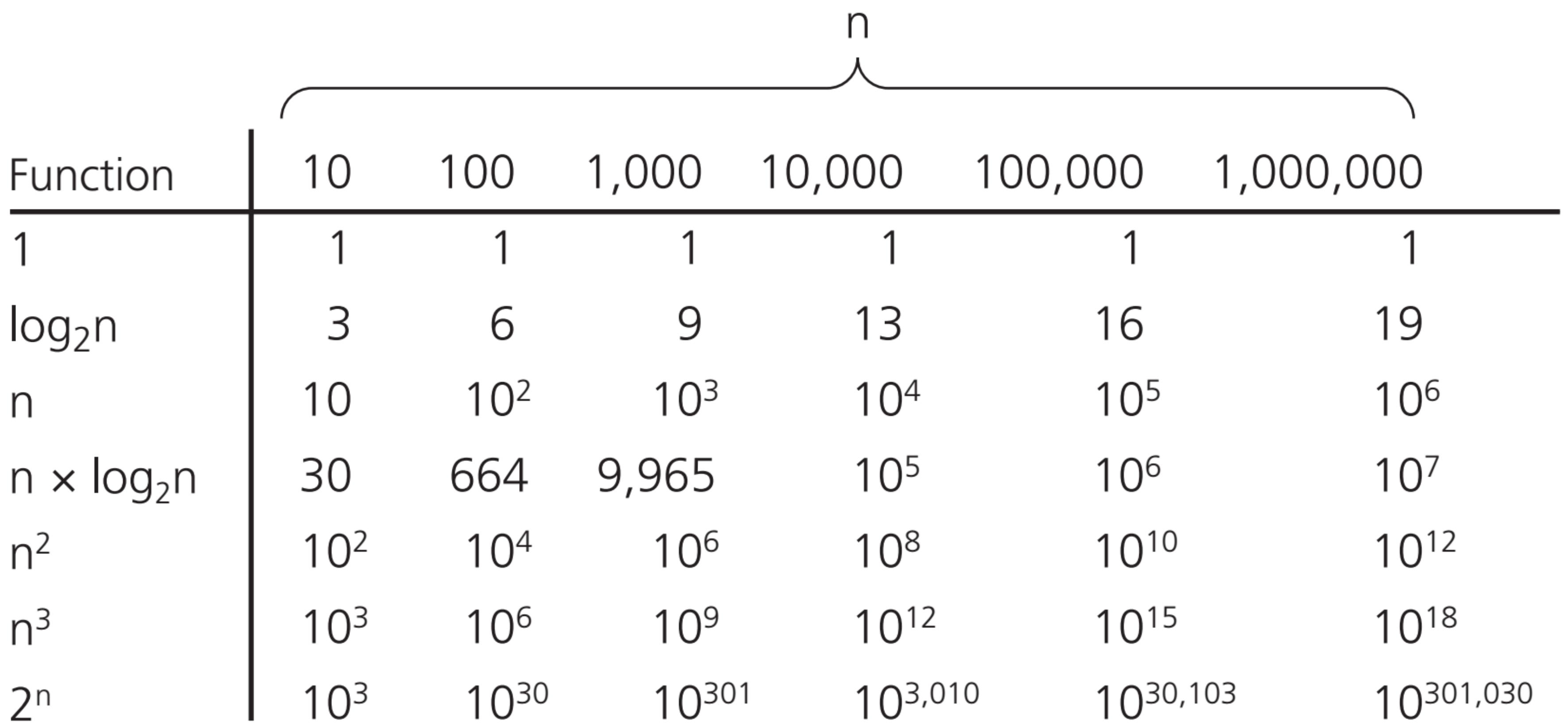
Common growth-rate functions

$\boldsymbol{g(n)}$	**Name**	$\boldsymbol{g(n)}$	**Name**
1	Constant	n^2	Quadratic
$\log_2 n$	Logarithmic	n^3	Cubic
n	Linear	2^n	Exponential
$n \times \log_2 n$	Linearithmic		

$$\boldsymbol{O(1) < O(\log_2 n) < O(n) < O(n \times \log_2 n) < O\left(n^2\right) < \mathrm{O}\left(n^3\right) < O(2^n)}$$

Common growth-rate functions

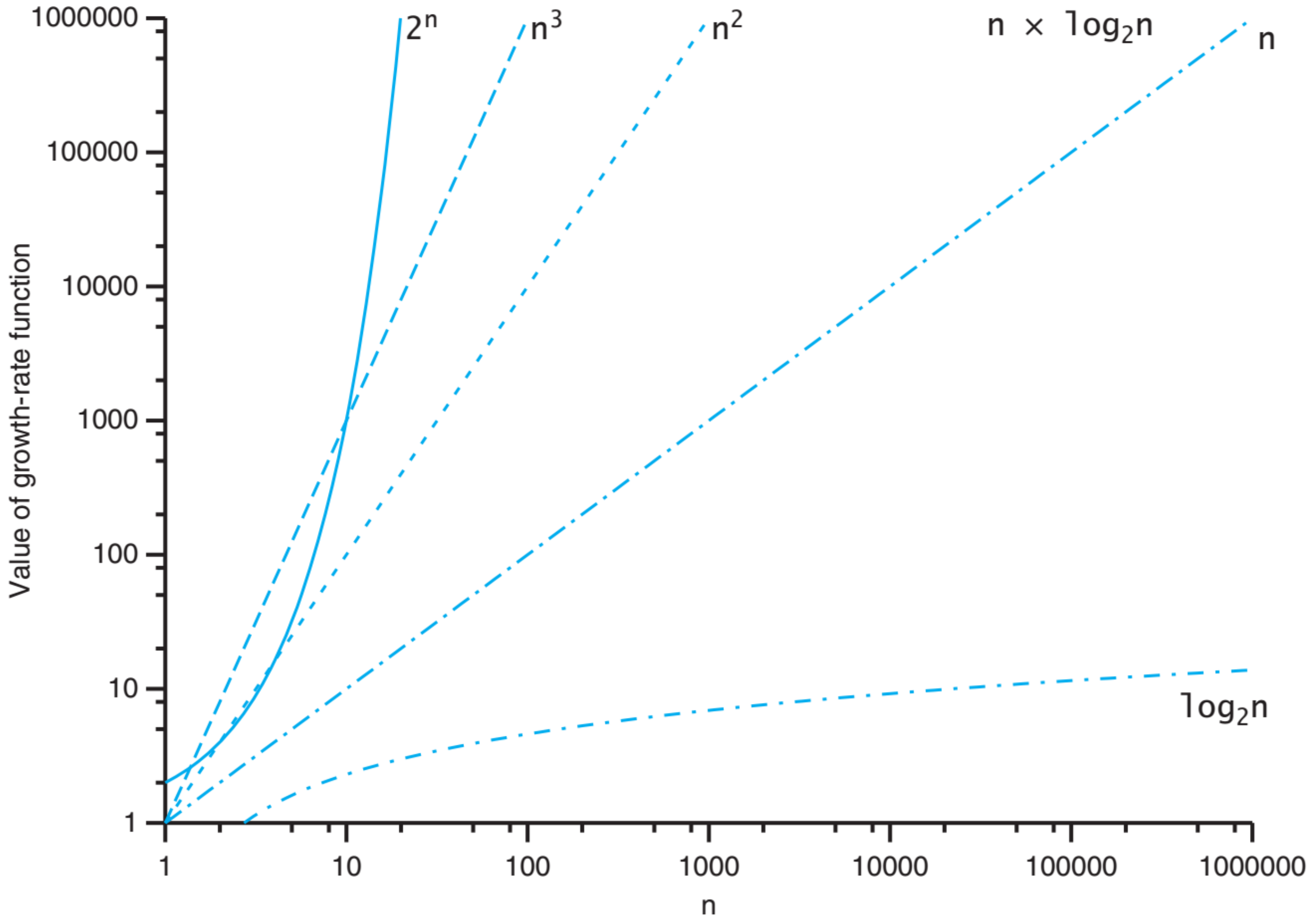
- A comparison of growth-rate functions in tabular form.



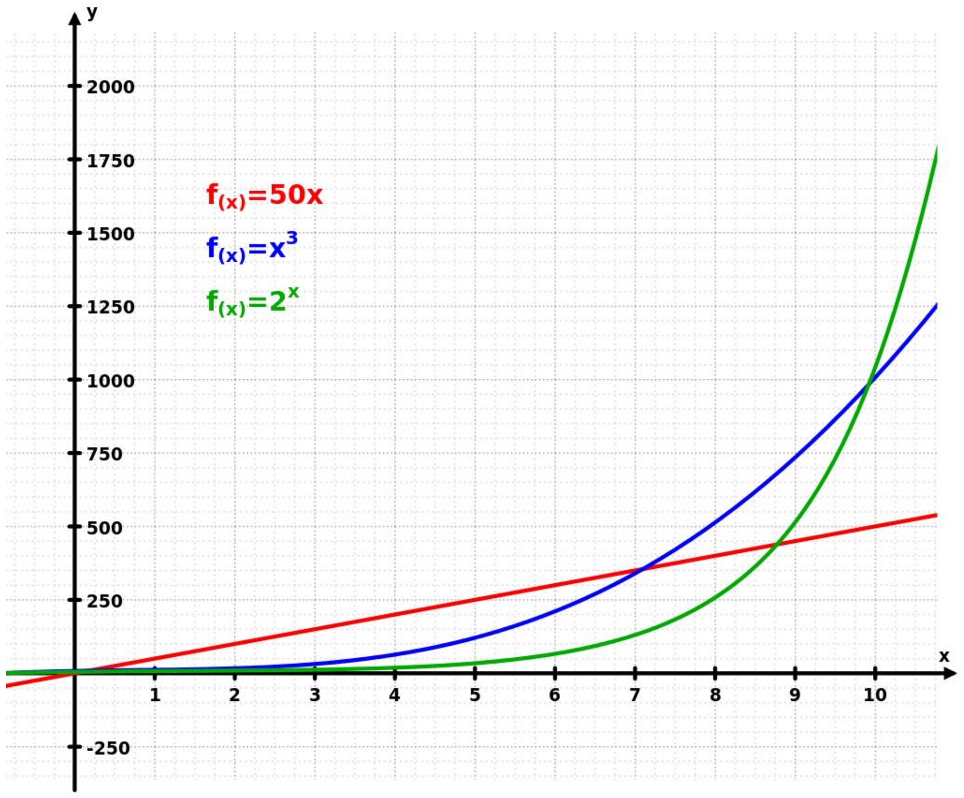
	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n \times \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Common growth-rate functions

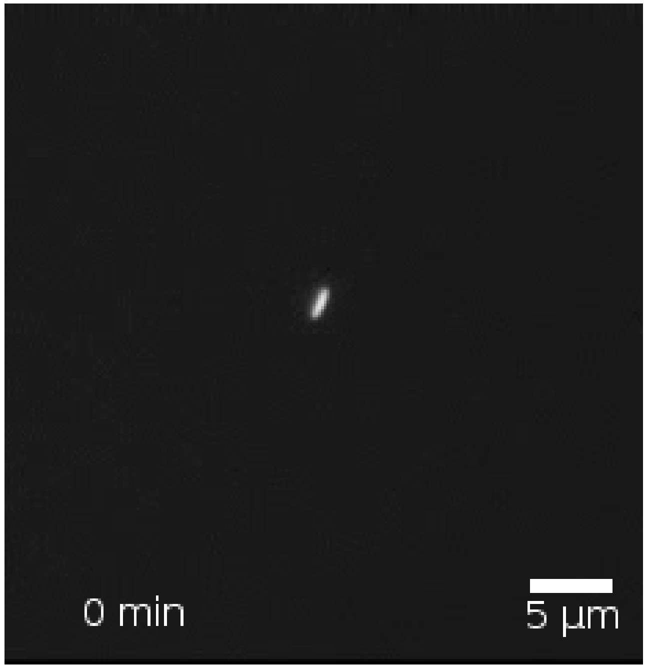
- A comparison of growth-rate functions in graphical form.



The challenge of exponential growth



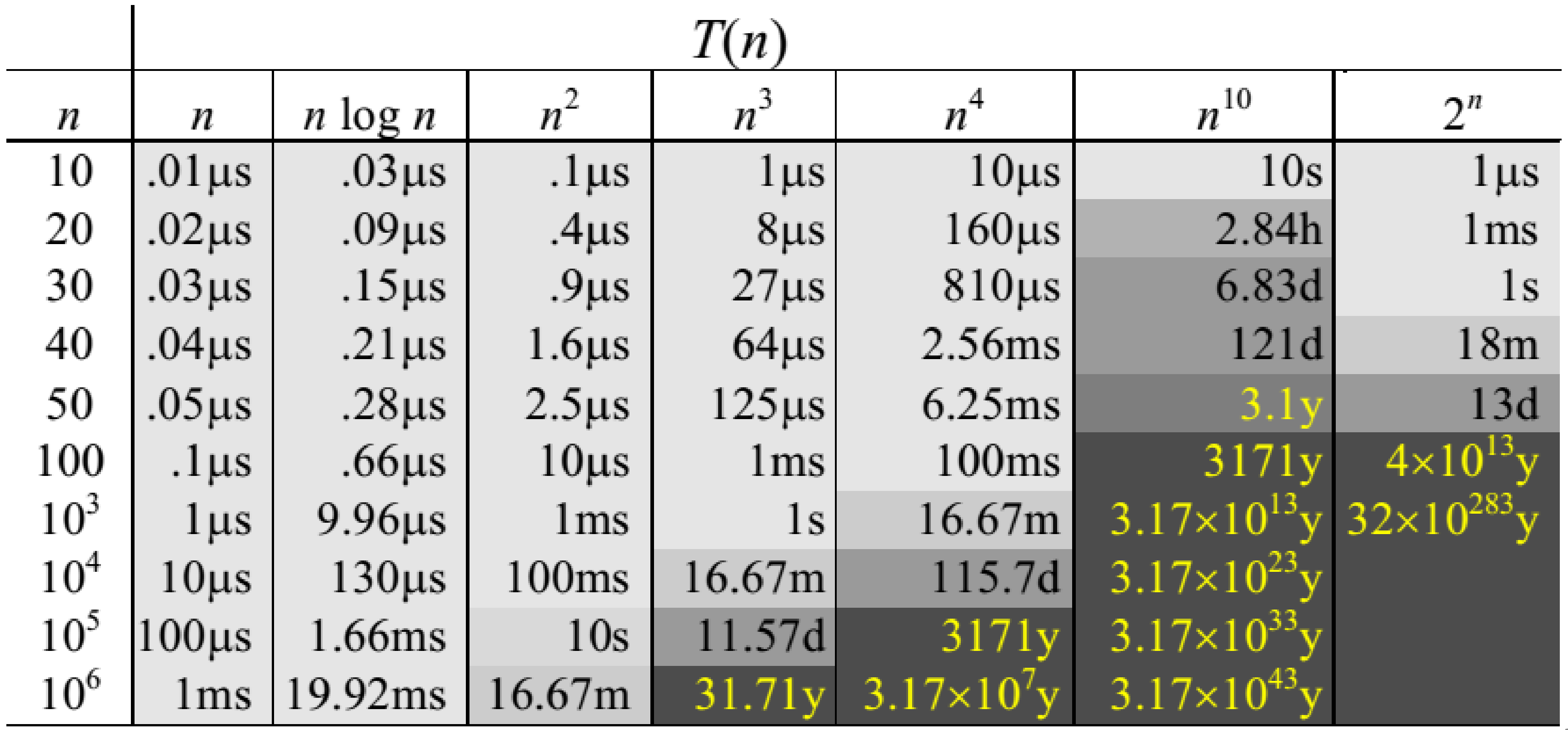
The exponential growth (green) surpasses both linear (red) and cubic (blue) growth.



Bacteria exhibit exponential growth under optimal conditions.

Example: How long do those growth-rate functions take?

Consider a hypothetical problem that can be solved by using one of the algorithms whose complexities are shown below. Assume that these algorithms are run on a computer that performs 10^9 operations/second.



	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01μs	.03μs	.1μs	1μs	10μs	10s	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84h	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1s
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121d	18m
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1y	13d
100	.1μs	.66μs	10μs	1ms	100ms	3171y	4×10^{13}y
10^3	1μs	9.96μs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10μs	130μs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100μs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^{7}y	3.17×10^{43}y	

Properties of growth-rate functions

1. You can ignore low-order terms in a growth-rate function.
   - If an algorithm is $O(n^3 + 4 \times n^2 + 3 \times n)$, it is also $O(n^3)$. For large n, the growth rate of $n^3 + 4 \times n^2 + 3 \times n$ is the same as that of n^3.
2. You can ignore a multiplicative constant in the high-order term of growth-rate function.
   - If an algorithm is $O(5 \times n^3)$, it is also $O(n^3)$. This follows the definition of $O(g(n))$ with $k = 5$.
3. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
   - If an algorithm is $O(n^2) + O(n)$, it is also $O(n^2 + n)$ and it can be written simply as $O(n^2)$ by applying Property 1.
   - Analogous rules hold for multiplication.

Example: Why ignore low-order terms?

Assume that there are two different algorithms which are designed to solve the same problem.

Let $f_1(n) = 0.1 \times n^2 + 10 \times n + 100$ and $f_2(n) = 0.1 \times n^2$ be the number of times the basic operation is executed by the first and second algorithm, respectively

$\boldsymbol{n}$	$\boldsymbol{f_1(n)}$	$\boldsymbol{f_2(n)}$	$\boldsymbol{f_1(n)}$ **/** $\boldsymbol{f_2(n)}$
10^1	210	10	21
10^2	2100	1000	2.1
10^3	110100	100000	1.101
10^4	100010000100	100000000000	1.000100001

Some useful results for analysis

- **Constant multiplication:** If $f(n)$ is $O(g(n))$ then $c \times f(n)$ is $O(g(n))$, where c is a constant.
- **Polynomial function:**

$$f(n) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \text{ is } O(x^n)$$

- **Summation function:** If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then

$$f_1(n) + f_2(n) \text{ is } O\left(max(g_1(n), g_2(n))\right)$$

- **Multiplication function:** If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then

$$f_1(n) \times f_2(n) \text{ is } O(g_1(n) \times g_2(n))$$

Important notes for Big-O notation

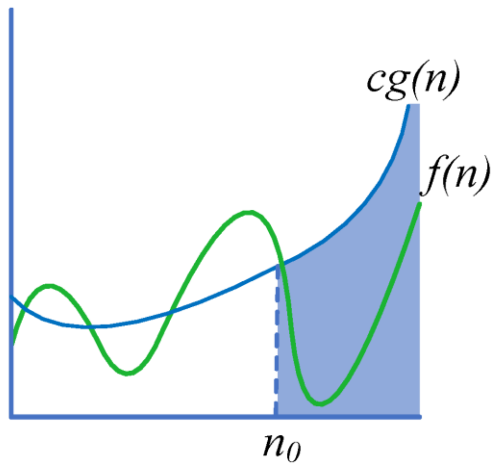
- $O(g(n))$ represents an inequality.
  - It is not a function but simply a notation that means “is of order $g(n)$” or “has order $g(n)$”.
- Do not use like this
  - $f(x) = O(g(x))$
  - $f(x) > O(g(x))$
- Use like this
  - $f(x)$ is $O(g(x))$, or
  - $f(x)$ is of order $g(x)$, or
  - $f(x)$ has order $g(x)$

Asymptotic notations

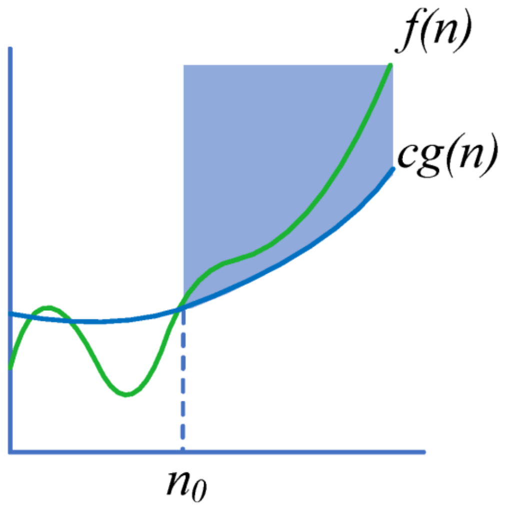
- Big-O: A function $t(n)$ is $O(g(n))$ if $t(n)$ is bounded above by some positive constant multiple of $g(n)$ for all large n.
  - There exists some positive constant c and nonnegative integer n_0 such that $f(n) \leq c \times g(n)$ for $n \geq n_0$.
- Big-Ω: A function $t(n)$ is $\Omega(g(n))$ if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n.
  - There exists c and n_0 such that $c \times g(n) \leq f(n)$ for $n \geq n_0$.
- Big-Θ: A function $t(n)$ is $\Theta(g(n))$ if $t(n)$ is bounded both above and below by some positive constant multiple of $g(n)$ for all large n.
  - There exists c_1, c_2 and n_0 such that $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for $n \geq n_0$.
  - That is, $f(n)\ has\ O(f(n))$ and $f(n)\ has\ \Omega(f(n))$

Asymptotic notations

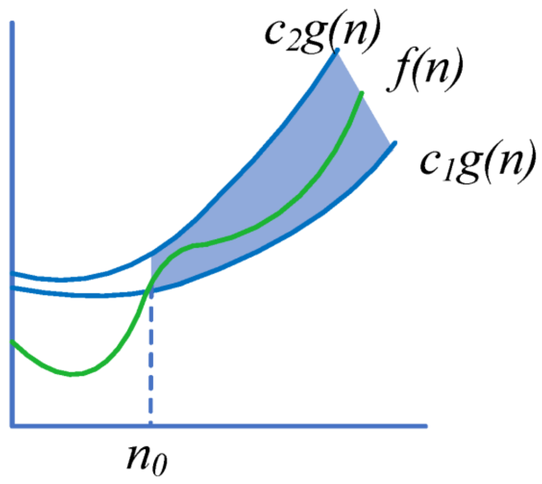
- A demonstration of asymptotic notations in graphical form.



Big-O notation



Big-Ω notation



Big-Θ notation

Checkpoint 03: Identify the Big-O complexities

What order is an algorithm that has as a growth-rate function

a) $8 \times n^3 - 9 \times n$

b) $7 \times \log_2 n + 20$

c) $7 \times \log_2 n + n$

Worst case, Best case, and Average case

- The efficiency of algorithms depends not only on the input size but also on the distribution of a particular input.

- Worst case: the longest running time for any input of size n
  - It is an upper bound on the running time for any input.
  - The analysis of algorithms focuses on the worst cases.
- Best case: the fastest running time for any input of size n
  - It is a lower bound on the running time for any input.
  - Almost impractical, it guarantees nothing about the bounds.

Worst case, Best case, and Average case

- Average case: an estimate of the average running time for all possible inputs of size n
  - Difficult to be estimated in practice and thus it is rarely done
  - We must know the mathematical distribution of all possible inputs.

- Note that, the average case requires understanding of the mathematical distribution of all possible inputs
  - It cannot be obtained by simply taking the average of the worst-case and the best-case efficiencies.

Example: Worst case, Best case, and Average case

Consider the following pseudo-code for the Bubble Sort algorithm.

```
BubbleSort(a[1 .. n]) {
   for (i = 2; i ≤ n; i++)
      for (j = n; j ≥ i; j--)
         if (a[j - 1] > a[j])
            a[j - 1] ⇄ a[j];
}
```

Let $C(i)$ be the number of comparisons made on a data set size n.

Then, $C(i) = \frac{n(n-1)}{2} \in \Theta(n^2)$

for all distributions of the input array.

Example: Worst case, Best case, and Average case

Consider an improved version of the previous Bubble Sort algorithm.

```
BubbleSortImproved(a[1 .. n]) {
   flag = true;
   m = 1;
   while (flag) {
      flag = false;
      m++;
      for (j = n; j ≥ i; j--)
         if (a[j - 1] > a[j]) {
            a[j - 1] ⇄ a[j];
            flag = true;
         }
   }
}
```

- Best case:

$$B(n) = n - 1$$

- Worst case:

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

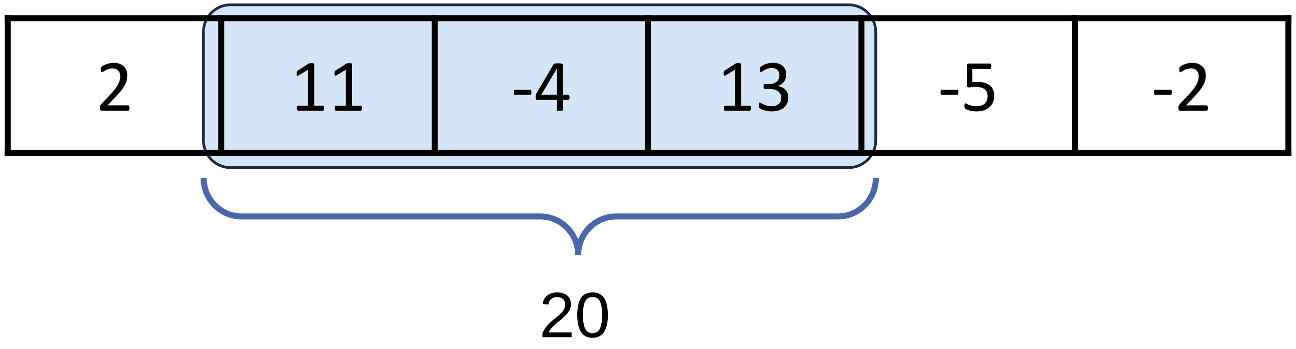
- Average case

$$A(n) = \frac{1}{n-1}\sum_{i=1}^{n-1} C(i) \in \Theta(n^2)$$

Algorithm analysis: Examples

Subsequence with largest sum

- Given an array of n integers $a_1, a_2, .., a_n$.
- The task is to find indices i and j with $1 \leq i \leq j \leq n$ such that the sum $\sum_{k=i}^{j} a_k$ is as large as possible.
- If the array contains all non-positive numbers, then the largest sum is 0.
- For example,



Subsequence with largest sum

- Brute force version with running time $O(n^3)$

```
MaxContSubSum(a[1 .. n]) {
  maxSum = 0;                          O(1)
  for (i = 1; i ≤ n; i++)
    for (j = i; j ≤ n; j++) {
      curSum = 0;                      O(1)
      for (k = i; k ≤ j; k++)
        curSum += a[k];                O(1)
      if (curSum > maxSum)
        maxSum = curSum;               O(1)
    }
  return maxSum;
}
```

Find the largest $A_i + A_{i+1} + .. + A_j$

It is 0 if the sum is non-positive.

$\mathbf{O}(j-i)$ $\mathbf{O}\left(\sum_{j=i}^{n-1}(j-i)\right)$ $\mathbf{O}\left(\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i)\right)$

Subsequence with largest sum

- An improved version with running time $O(n^2)$

```
MaxContSubSum(a[1 .. n]) {
  maxSum = 0;
  for (i = 1; i ≤ n; i++) {
    curSum = 0;
    for (j = i; j ≤ n; j++) {
      curSum += a[j];
      if (curSum > maxSum)
        maxSum = curSum;
    }
  }
  return maxSum;
}
```

Given the sum from i to $j-1$, calculate the sum from i to j in a constant time.

Subsequence with largest sum

- Dynamic programming version with running time $O(n)$

```
MaxContSubSum(a[1 .. n]) {
  maxSum = curSum = 0;
  for (j = 1; j ≤ n; j++) {
    curSum += a[j];
    if (curSum > maxSum)
      maxSum = curSum;
    else
      if (curSum < 0)
        curSum = 0;
  }
  return maxSum;
}
```

Reset the current sum to 0 if it becomes negative

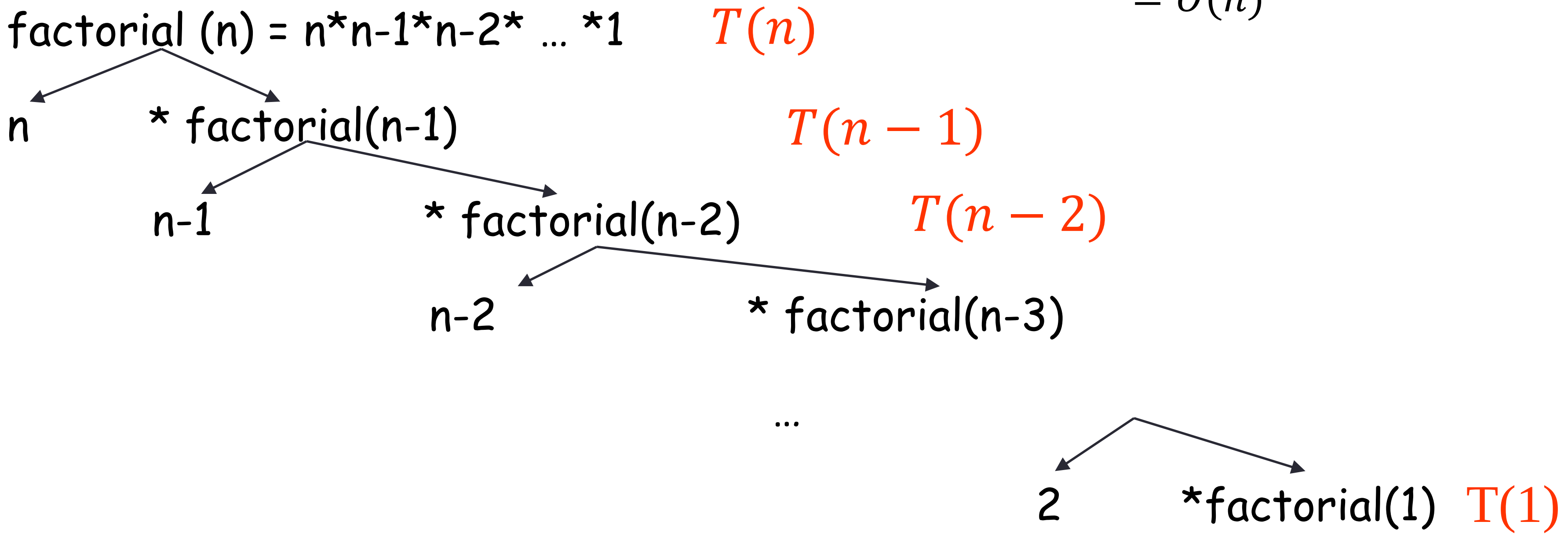
Calculate the new sum value and update the maximum sum so far

Factorial calculation

- The recursive version with running time $O(n)$

```
int factorial1 (int n){
   if (n <= 1)
       return 1;
   return n * factorial(n-1);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + d \\
&= T(n-2) + d + d \\
&= \dots. \\
&= T(1) + (n-1) * d \\
&= c + (n-1) * d \\
&= O(n)
\end{aligned}
$$



Factorial calculation

- The non-recursive version with running time $O(n)$

```
int factorial2 (int n) {
  if (n <= 1)
    return 1;
  fact = 1;
  for (k = 2; k <= n; k++)
    fact *= k;
  return fact;
}
```

$O(1)$

$O(1)$ $O(n)$

The Towers of Hanoi

- The number of steps move n disks from one peg to another is $2^n - 1$.
- For n number of disks, the way to accomplish the task in a minimum number of steps is
    - Move the top $n-1$ disks to an intermediate peg,
    - Move the bottom disk to the destination peg, and
    - Move the $n-1$ disks from the intermediate peg to the destination.
- Thus, the recurrence relation for this puzzle would become

$$T(1) = 1$$

$$T(n) = 2T(n-1) + 1,\ n \geq 2$$

The Towers of Hanoi

$T(n)$

$= 2T(n-1) + 1$

$= 2^2T(n-2) + 2 + 1$

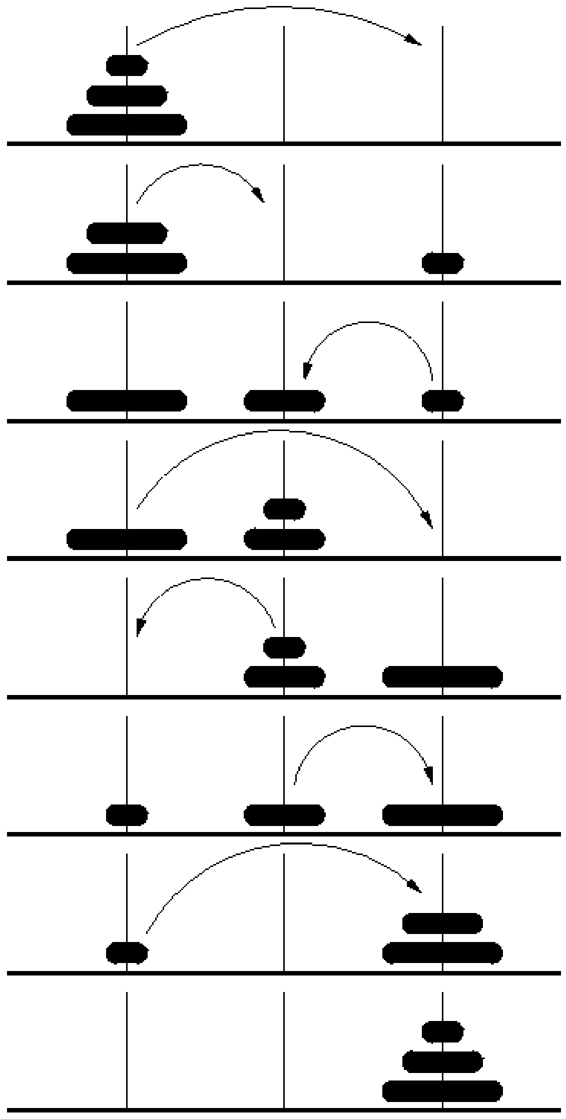
$= 2^3T(n-3) + 2^2 + 2 + 1$

$= \ldots$

$= 2^{n-1}T(1) + (2^{n-2} + \ldots + 2 + 1)$

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

$= O(2^n)$



Acknowledgements

This part of the lecture is adapted from the following materials.

[1] Pr. Nguyen Thanh Phuong (2020) "*Lecture notes of CS163 – Data structures*" University of Science - Vietnam National University HCMC.

[2] Pr. Van Chi Nam (2019) "*Lecture notes of CSC14004 – Data structures and algorithms*" University of Science - Vietnam National University HCMC.

[3] Frank M. Carrano, Robert Veroff, Paul Helman (2014) "*Data Abstraction and Problem Solving with C++: Walls and Mirrors*" Sixth Edition, Addion-Wesley. **Chapter 10.**

[4] Anany Levitin (2012) "*Introduction to the Design and Analysis of Algorithms*" Third Edition, Pearson.

Exercises

01. Algorithm and Complexity

- Propose an algorithm to calculate the value of S defined below.

$$S = 1 + \frac{1}{2} + \frac{1}{6} + .. + \frac{1}{n!}$$

- What order does the algorithm have?

02. Count the number of operations

- How many comparisons and assignments are there in the following code fragments with the size n?

a)

```
sum = 0;
for (i = 0; i < n; i++){
  cout << x;
  sum = sum + x;
}
```

b)

```
for (i = 0; i < n ; i++)
  for (j = 0; j < n; j++){
    C[i][j] = 0;
    for (k = 0; k < n; k++)
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
  }
```

03. Big-O complexities

- What is the Big-O complexity of each of the following code fragments?

a)

```
int sum = 0;
for (int i = 1; i < n; i *= 2)
  for (int j = 0; j < n; j++)
    sum++;
```

b)

```
int sum = 0;
for (int i = 1; i < n; i *= 2)
  for(int j = 0; j < i; j++)
    sum++;
```

c)

```
int sum = 0;
for (int i = n; i > 0; i /= 2)
  for(int j = 0; j < i; j++)
    sum++;
```

Hint is here.

...the end.