BACS2063 Data Structures and Algorithms

Efficiency of Algorithms

Chapter 3

Learning Outcomes

At the end of this lecture, you should be able to

- Assess the efficiency of a given algorithm
- Compare the expected execution times of two methods, given the efficiencies of their algorithms

Multiplication - Algorithm 1

```
long firstOperand = 7562;
long secondOperand = 423;
long product = 0;
for(long ctr = secondOperand; ctr > 0; ctr--)
   product = product + firstOperand;
System.out.println(product);
```

- When `423` is changed to `100,000,000` there is a significant delay in seeing the result
- Sample Code: `Chapter3\samplecode\` **`Multiplication1.java`**

Multiplication - Algorithm 2

- Alternative way to multiply `7562` and `423`:

```
7562 * 423= 7562 * (400 + 20 + 3)
          = (7562*400) + (7562*20) + (7562*3)
          = (756200*4) + (75620*2) + (7562*3)

          = (756200+756200+756200+756200) +
            (75620+75620) + (7562+7562+7562 )

          = (7562+7562+7562) + (75620+75620) +

            (756200+756200+756200+756200)
```

- Sample Code: `Chapter3\samplecode\`
  `Multiplication2.java`

Motivation

- Even a simple program can be noticeably inefficient.
- A program's efficiency affects its overall performance (e.g., response time) and user satisfaction.
- How can we measure efficiency so that we can compare the efficiency of various approaches to solve a problem?

Two ways to measure the efficiency of algorithms:

1. Experimental studies
2. Analysis of algorithms

Method 1

Experimental Studies

1. Experimental Studies

- Implement the algorithm(s)
- Then, experiment by running the program on various test inputs while recording the time spent during each execution.
- *E.g.*, collecting running times in Java using the `System.currentTimeMillis()` method:
  - Record the time immediately before executing the algorithm and then immediately after
  - Compute the difference between those times to obtain the *elapsed* time of the algorithm's execution

Code Fragment for Timing an Algorithm in Java

```
1   long startTime = System.currentTimeMillis( );
2   /* (run the algorithm) */
3   long endTime = System.currentTimeMillis( );
4   long elapsed = endTime − startTime;
```

Sample Code: `Chapter3\samplecode\`

`TimingAlgorithmsMilli.java`

Note: for extremely quick operations, Java provides the **`nanoTime`** method that measures in nanoseconds rather than milliseconds

Experimental Method

- As we are interested in the general dependence of running time on the size and structure of the input, we should perform independent experiments on many different test inputs of various sizes.

- We can then visualize the results by plotting the performance of each run of the algorithm as a point with *x*-coordinate equal to the input size *n* and *y*-coordinate equal to the running time *t*.

- Such a visualization provides some intuition regarding the relationship between problem size and execution time for the algorithm.

Experimental Method (cont'd)

- This may be followed by a statistical analysis that seeks to fit the best function of the input size to the experimental data.

- To be meaningful, this analysis requires that we choose **good sample inputs** and **test enough** of them to be able to make sound statistical claims about the algorithm's running time.

Challenges of Experimental Analysis

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

- 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 (and these inputs may be important).

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

Method 2

Analysis of Algorithm

2. Analysis of Algorithms

- The process of measuring the **complexity** of an algorithm
- Types of complexity
  - Space complexity: the ***memory*** required to execute the code
  - Time complexity: the ***time*** the code takes to execute
- There is usually a *trade-off* between time and space requirements of a program.

Goal for Algorithm Analysis

Due to the shortcomings of experimental analysis, we need an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.

2. Is performed by studying a high-level description of the algorithm without need for implementation.

3. Takes into account all possible inputs

Chapter Focus

- The *time complexity* of algorithms (usually more important than space complexity).
- The actual time needed for an algorithm is difficult to compute accurately. Therefore,
  - We estimate the times for the *best case*, *average case* and *worst case*.
  - We express time efficiency as a factor of the problem size *(e.g.*, when searching a collection of data, the problem size is the number of items in the collection).

Best Case, Average Case, Worst Case

- **Best Case** – ***Minimum* time required** for program execution.

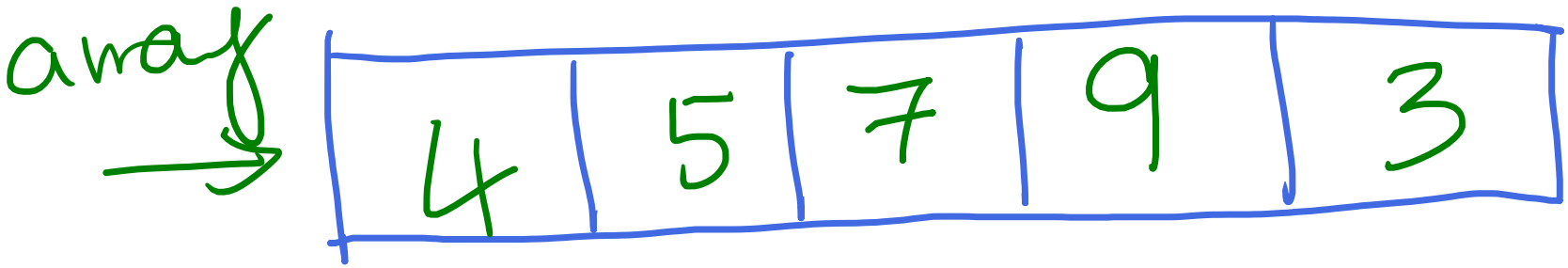
- **Average Case** – ***Average* time required** for program execution.

- **Worst Case** – ***Maximum* time required** for program execution.

BEST CASE

WORST CASE

Task: Find the number 10 in array



Counting Primitive Operations

- To analyze the running time of an algorithm without performing experiments, analyze the algorithm by counting *primitive operations*.
- Example of primitive operations:
  - Assigning a value to a variable
  - Following an object reference
  - Performing an arithmetic operation
  - Comparing two numbers
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method

Counting Primitive Operations (cont'd)

- Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

- This operation count will correlate to an actual running time in a specific computer.

- Thus, the number t of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

Measuring Operations as a Function of Input Size

- To capture the order of growth of an algorithm's running time, we will associate a function ***f(n)*** with the algorithm.

- This function ***f(n)*** will characterize the number of primitive operations that are performed as a function of the input size *n*.

Difficulty of Average-Case Analysis

- An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size.

- Unfortunately, such an average-case analysis is quite challenging as it requires us to define a probability distribution on the set of inputs.

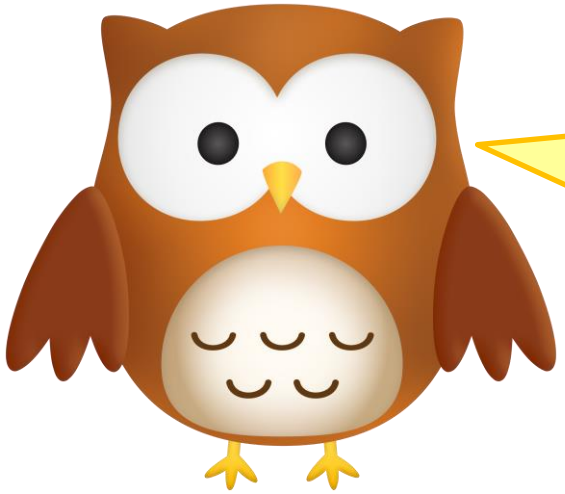
Focusing on the Worst-Case

- **Worst-case analysis** is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple.

- Also, this approach typically leads to better algorithms due to the focus on designing to cope for better performance in the worst case.

Note: Useful formulas

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



Read more about Arithmetic Sequence and Sum here:

https://www.mathsisfun.com/algebra/sequences-sums-arithmetic.html

Analysis of Algorithms

- **Step 1**: Count the number of Primitive Operations
- **Step 2**: Derive its Big-O Notation
- **Step 3**: Compare the growth rate of different algorithms

Consider this operation:

- Sum the number from 1 until n
- i.e. 1 + 2 + 3 + 4 …. + n

Problem: Analyzing the efficiency of algorithms for computing 1 + 2 +…n for an integer n > 0

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
`for i = 1 to n`
`  sum = sum + i`
	`sum = 0`
	`for i = 1 to n`
	`{ for j = 1 to i`
	`  sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

Fig. 2.1: Three algorithms for computing 1 + 2 + … n for an integer n > 0

Analysis of the 3 Algorithms

	Algorithm A	Algorithm B	Algorithm C
Assignments	$n + 1$	$1 + n\,(n + 1) / 2$	1
Additions	n	$n\,(n + 1) / 2$	1
Multiplications			1
Divisions			1
Total operations	$\mathbf{2n + 1}$	$\mathbf{n^2 + n + 1}$	**4**

Fig. 2.2: The number of operations required by the algorithms for Fig 2.1

(Note: operations for the **`for`** *loops are ignored for simplicity)*

Analysis of Algorithm A

Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + i
```

	Algorithm A
Assignments	$n + 1$
Additions	n
Multiplications	
Divisions	
Total operations	$\mathbf{2n + 1}$

Analysis of Algorithm A

```
sum = 0
for i = 1 to n
   sum = sum + i
```

Total operations

= total assignments + total additions

= (1 assignment for initializing sum + n assignments in for loop body) + (n additions in for loop body)

= (1 + n) + (n)

= **2n + 1**

Analysis of Algorithm B

Algorithm B

```
sum = 0
for i = 1 to n
{ for j = 1 to i
    sum = sum + 1
}
```

	Algorithm B
Assignments	$1 + n\,(n + 1) / 2$
Additions	$n\,(n + 1) / 2$
Multiplications	
Divisions	
Total operations	$\boldsymbol{n^2 + n + 1}$

Analysis of Algorithm B

```
sum = 0
for i = 1 to n {
   for j = 1 to i
      sum = sum + 1
}
```

i	j	Total =	Total +
1	1..1	1	1
2	1..2	2	2
3	1..3	3	3
. . .	. . .	. . .	. . .
n	1..n	n	n

Total operations

= total assignments + total additions

= (1 assignment for initializing sum + n assignments in for loop body) + (n additions in for loop body)

= (1 + [1+2+3…+n]) + [1+2+3+…+n]

= (1 + [n(n+1)/2]) + [n(n+1)/2]

= $\mathbf{n^2 + n + 1}$

Analysis of Algorithm C

Algorithm C

```
sum = n * (n + 1) /  2
```

	Algorithm C
Assignments	1
Additions	1
Multiplications	1
Divisions	1
Total operations	**4**

Analysis of Algorithm C

```
sum = n * (n + 1) / 2
```

Total operations

= total assignments + total multiplications + total additions + total divisions

= `1 + 1 + 1 + 1`

= **4**

PYQ 8-June-2023 Q2c

c) Big O notation is used to determine the time efficiency of an algorithm. Calculate the Big O for the following algorithms in the Figure 3.

Algorithm A	Algorithm B	Algorithm C

`sum = 0`
`for i = 1 to n`
`  sum = sum + i`
	`sum = 0`
	`for i = 1 to n`
	`{ for j = 1 to i`
	`  sum = sum + 1`
	`}`
		`sum = n * (n + 1) / 2`

Figure 3: Algorithms A, B and C

(5 marks)

Answer

1. Algorithm A – O(n) - O(n) represents the complexity of a function that increases linearly and in direct proportion to the number of inputs. B
2. Algorithm A – $O(n^2)$ - $O(n^2)$ represents a function whose complexity is directly proportional to the square of the input size.
3. Algorithm A – O(1) - O(1) represents a function that always takes the same take regardless of input size. C

Comparing Algorithm Efficiency

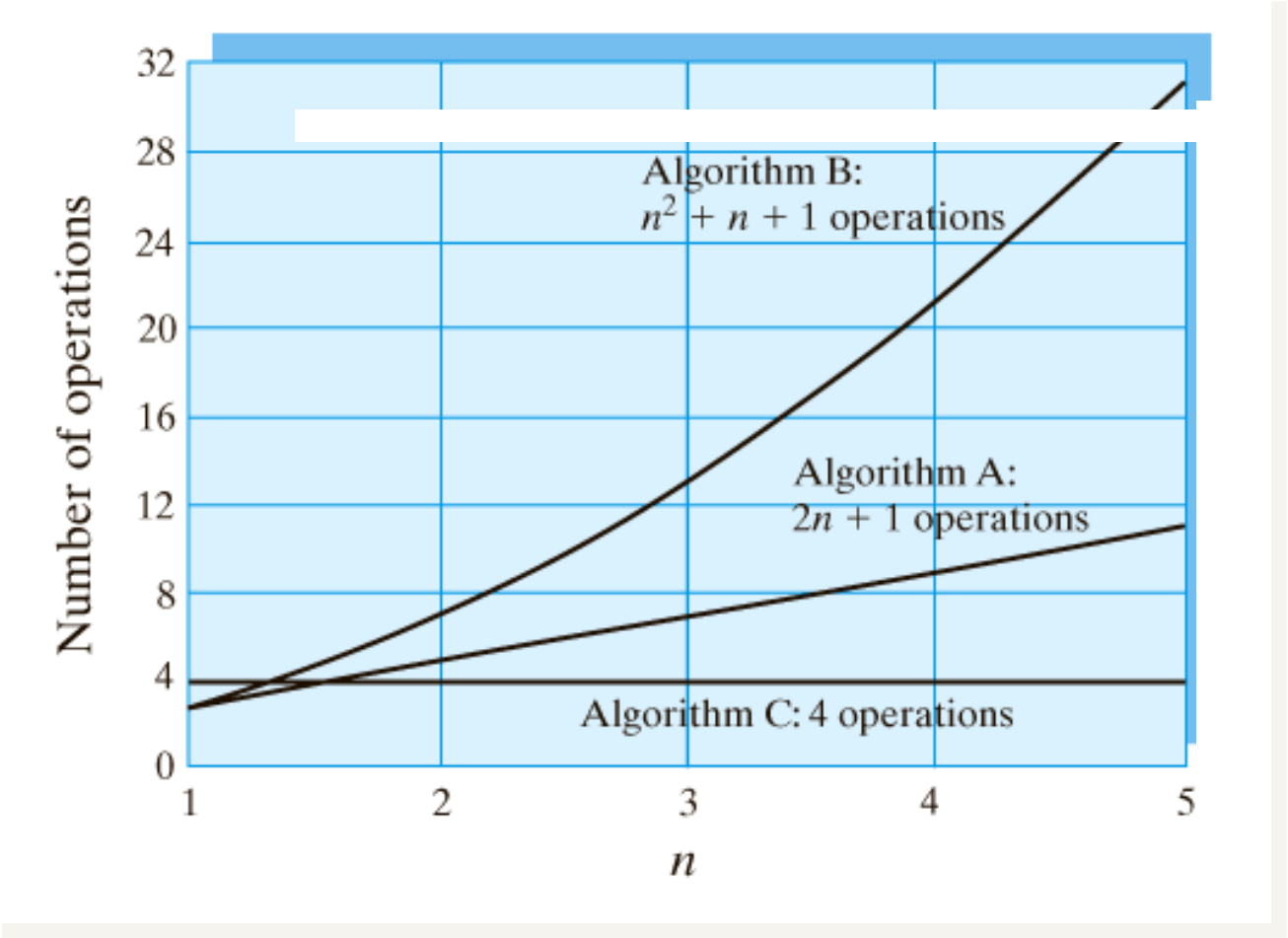
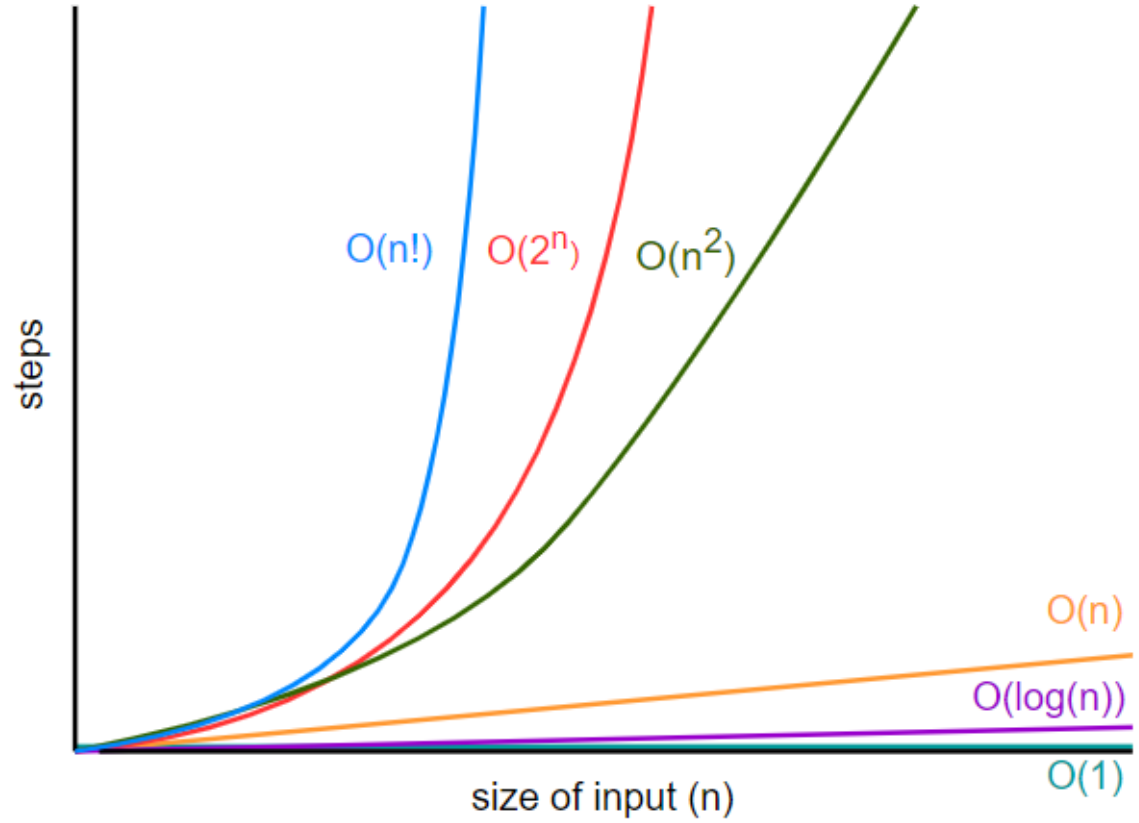


Fig. 2.3: The number of operations required by the algorithms in Fig. 9.1 as a function of *n*

Common classes in Big O



Source:
https://www.data-structures-in-practice.com/big-o-notation/

Common classes in Big O

Big O	Name
O(1)	constant
O(log n)	logarithmic
O(n)	linear
$O(n^2)$	quadratic
$O(2^n)$	exponential

Source: https://www.data-structures-in-practice.com/big-o-notation/

Expressing Algorithm Efficiency

- The actual time requirement of an algorithm cannot be computed. Instead,
  - We find a function of the problem size that represents the algorithm's time requirement.
  - As the problem size increases by some factor, the value of the function increases by the same factor and this result indicates the increase in the algorithm's time requirement.
- Computer scientists use the **Big O notation** to represent an algorithm's efficiency / complexity.

Deriving the Big O notation

- Steps to derive the Big O notation for an algorithm
  - use the given identities and
  - ignore smaller terms in a growth-rate function
- *E.g.*, if the growth-rate function is $4n^2 + 5on - 10$,

  $O(4n^2 + 5on - 10)$

  $= O(4n^2)$ by ignoring the smaller terms

  $= O(n^2)$ by ignoring the constant multiplier / coefficient

Exercise 3.1

Derive the Big O notations for Algorithm A, B & C.

Algorithm	Growth-rate function	Big O Notation
Algorithm A	**$2n + 1$**	**$O(n)$**
Algorithm B	**$n^2 + n + 1$**	**$O(n^2)$**
Algorithm C	**4**	**$O(1)$**

Algorithm A = **$2n + 1$**

= **$2n$**, ignoring smaller terms

= **n**, ignoring the coefficient 2

$\Rightarrow$ **$O(n)$**

Verbalizing Big O Notations

- To say "Algorithm A has a worst-case time requirement proportional to *n*"
  - We say A is **O(n)**
  - Read "Big O of *n*"
- For the other two algorithms
  - Algorithm B is **O(n²)**
  - Algorithm C is **O(1)**

PYQ 19-Jan-2023 Q2c

c) Describe **TWO (2)** importance of Big-O notation used in the time efficiency of algorithms. (5 marks)

Answer:

The actual time requirement of an algorithm cannot be computed. Instead

- We find a function of the problem size that represents the algorithm's time requirement.
- As the problem size increases by some factor, the value of the function increases by the same factor and this result indicates the increase in the algorithm's time requirement.

PYQ 14-Oct-2022 Q2b

b) Computer scientists use the Big-O notation to represent an algorithm's efficiency or complexity. Describe why Big-O notation is better than experimental studies. (7 marks)

Answer:

- The actual time requirement of an algorithm cannot be computed. Instead,
  - We find a function of the problem size that represents the algorithm's time requirement.
  - As the problem size increases by some factor, the value of the function increases by the same factor and this result indicates the increase in the algorithm's time requirement.
- Computer scientists use the Big O notation to represent an algorithm's efficiency / complexity.

The 7 Most Common Functions in Algorithm Analysis

1.) The Constant Function: $f(n) = c$

- Where c is a fixed constant (*e.g.*, $c = 5$, $c = 2^{10}$)
- *I.e.*, for any argument *n*, the constant function *f(n)* assigns the value *c*.
- Therefore, it doesn't matter what the value is *n* is; *f(n)* will always be equal to the constant value *c*.
- This function characterizes the number of steps needed to do a basic operation (*e.g.*, adding 2 numbers, assigning a value to a variable, comparing 2 numbers)

1) The Constant Function: $f(n) = c$

Regardless of *n*, the running time is always the same.

2) The Logarithm Function: $f(n) = \log_b n$

- For some constant $b > 1$
- b is the ***base*** of the logarithm and for any base $b > 0$, $\log_b 1 = 0$
- $x = \log_b n$ **iff** $b^x = n$
- The most common base in computer science is 2 as computers store integers in binary. Therefore, the default logarithm base for us is 2, i.e. $\log n = \log_2 n$

3.) The Linear Function: $f(n) = n$

- *I.e.*, given an input value n, the linear function f assigns the value n itself.
- This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements.
  - E.g., comparing a number x to each element of an array of size n will require n comparisons.

3.) The Linear Function: $f(n) = n$

The running time will grow proportionately in a linear fashion according to the problem size n.

4.) The n-log-n function: $f(n) = n \log n$

- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function.

5.) The 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.
- Thus, in such cases, the algorithm performs $n * n = n^2$ operations

6.) The Cubic Function: $f(n) = n^3$

- This function appears less frequently than the constant, linear and quadratic functions.

7.) The Exponential Function: $f(n)=b^n$

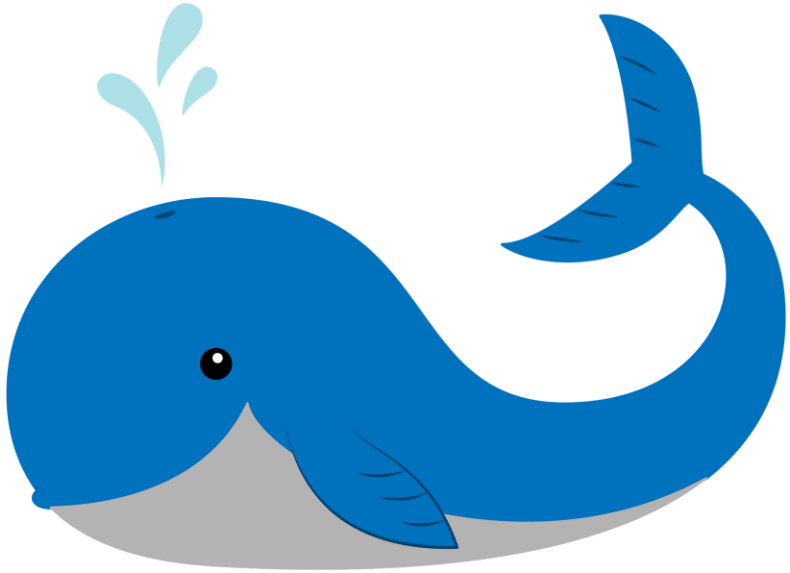
- Where ***b*** is a positive constant called the ***base***, and the argument ***n*** is the ***exponent***.
- In algorithm analysis, the most common base for the exponential function is $b = 2$.
- E.g., if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n.

7.) The Exponential Function: $f(n)=b^n$

Very fast growth rate!

Not desirable!

Conclusion:



Choose the
slower growing
function!

Growth-Rate Functions

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Fig. 2.4: Typical growth-rate functions evaluated at increasing values of *n*

(Note: logarithms are base-2)

$O(\log_{10} n)$

n	Number of Digits	$\lfloor \log_{10} n \rfloor$
10 – 99	2	1
100 – 999	3	2
1000 – 9999	4	3

Fig. 2.5: The number of digits in an integer *n* compared with the integer portion of $\log_{10} n$

Picturing Efficiency

```
for i = 1 to n
    sum = sum + i
```

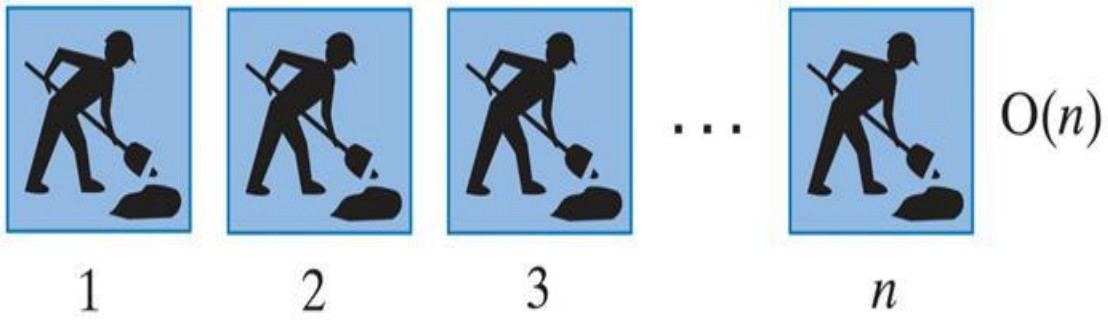


Fig. 2.6: an `O(n)` algorithm.

Picturing Efficiency

```
for i = 1 to n
{   for j = 1 to i
        sum = sum + 1
}
```

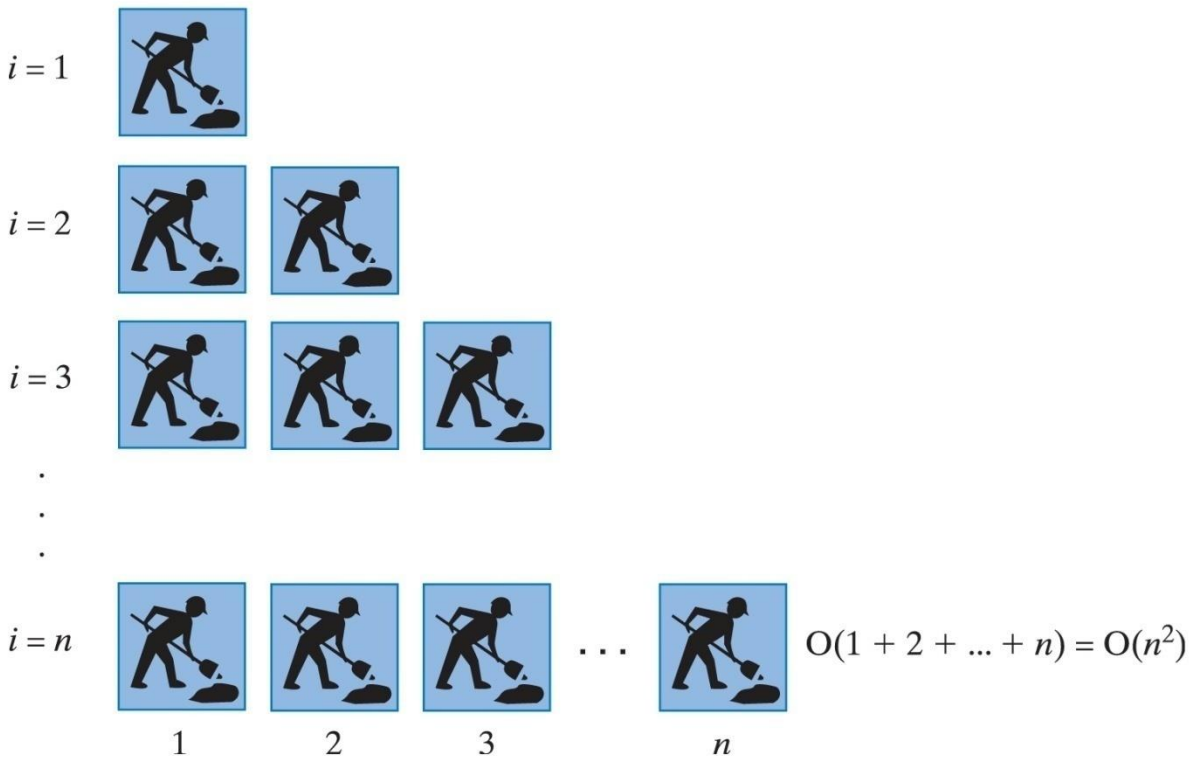


Fig. 2.7: An $O(n^2)$ algorithm.

Picturing Efficiency

```
for i = 1 to n
{   for j = 1 to n
        sum = sum + 1
}
```

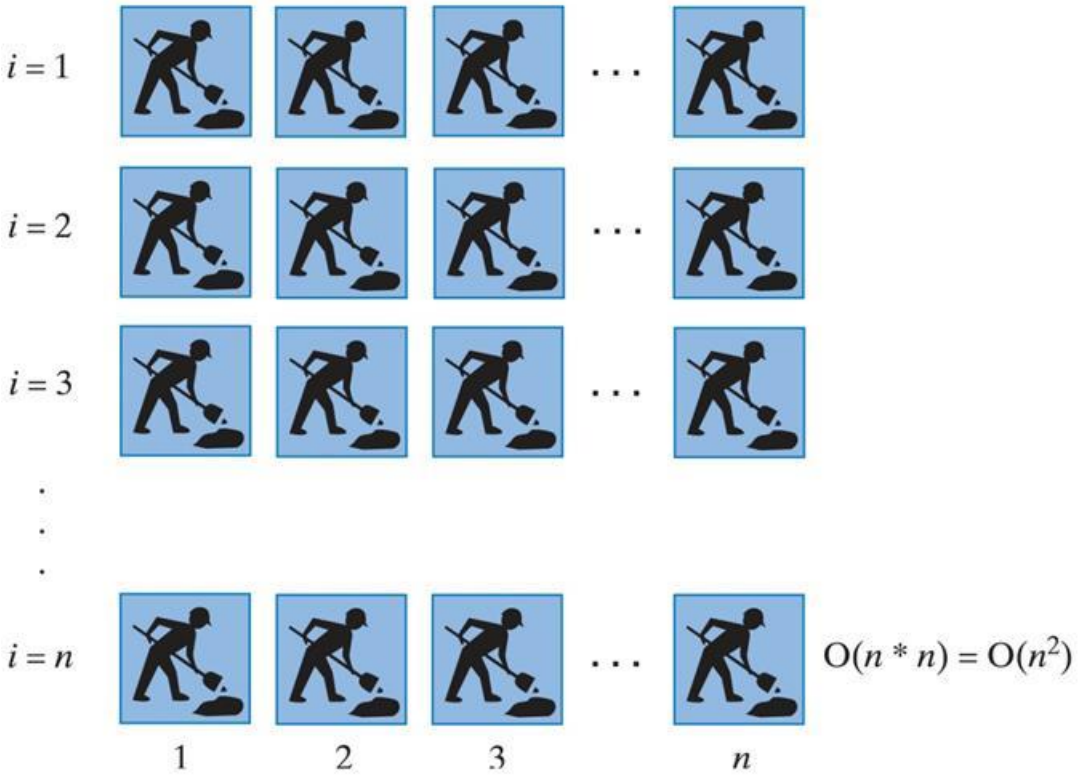


Fig. 2.8: Another $O(n^2)$ algorithm.

Picturing Efficiency

Growth-Rate Funtion for Size n Problems	Growth-Rate Funtion for Size $2n$ Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

Fig. 2.9: The effect of doubling the problem size on an algorithm's time requirement.

Logarithm Rules

Rule 1: $\log_b (M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b \left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b \left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b (1) = 0$

Rule 5: $\log_b (b) = 1$

Rule 6: $\log_b \left(b^k\right) = k$

Rule 7: $b^{\log_b (k)} = k$

Where : $b > 1$, and M, N and k can be any real numbers

but M and N must be positive!

Source:
https://www.chilimath.com/lessons/advanced-algebra/logarithm-rules/

Picturing Efficiency

If 1,000,000 operations / second

If n = 1,000,000

Growth-Rate Funtion g	$g(10^6) / 10^6$
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Fig. 2.10: The time to process one million items by algorithms of various orders at the rate of one million operations per second.

Comments on Efficiency

- A programmer can use $O(n^2)$, $O(n^3)$ or $O(2^n)$ as long as the problem size is small
- At one million operations per second it would take 1 second …
  - For a problem size of 1000 with $O(n^2)$
  - For a problem size of 100 with $O(n^3)$
  - For a problem size of 20 with $O(2^n)$

Exercise 3.2

Identify the Big O notation of the following method:

```
public String toString() {
  String str = “”;
  for (int i = 0; i < length; i++)
    str += array[i] + “\n”;
  return str;
}
```

O(n):

The code traverses through each array element to concatenate its value to the string variable `str`.

Exercise 3.3

Using Big O notation, indicate the time efficiency for the worst case of each task below. State your assumptions, if any.

a. After arriving at a party, you shake hands with each person there. O(n)

b. Each person in a room shakes hands with everyone else in the room. $O(n^2)$

c. You climb a flight of stairs. O(n)

d. You slide down the banister. O(h)

e. After entering an elevator, you press a button to choose a floor. O(1)

f. You ride the elevator from the ground floor up to the n^{th} floor. O(n)

g. You read a book twice. O(n)

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson
- https://www.javacodegeeks.com/2011/04/simple-big-o-notation-post.html

Learning Outcomes

You should now be able to

- Assess the efficiency of a given algorithm
- Compare the expected execution times of two methods, given the efficiencies of their algorithms