Asian MathSci League, Inc (AMSLI)
Website: amsliphil.com
Email address: amsliphil@yahoo.com

Student Copy

AIEP Sendoff HS Session 6

ALGORITHM EFFICIENCY & BITWISE OPERATORS

"An algorithm must be seen to be believed." ~ Donald Knuth

The backbone of programming is not the language but the algorithm, in the same way that the essence of poetry lies not in the language in which it was written but in the idea being expressed. A good programmer is not someone who knows a wide array of programming languages but someone who can apply the principles of computational thinking and formulate a well-designed algorithm to solve the task at hand.

WHAT IS AN ALGORITHM?

Coming from the Latinization of the name of the Persian mathematician Muhammad ibn Musa al-Khwarizmi, an algorithm is defined as any **"well-defined computational procedure that takes some value, or set of values as input and produces some value, or set of values, as output."**

This definition, which comes from Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms* (fondly referred to as CLRS or the bible of algorithms), emphasizes three aspects constituting an algorithm:

1. **The steps must be unambiguous.**
2. **There must always be an input and an output.**
3. **It must terminate at some point.**

WHY DO WE USE ALGORITHMS?

Humans are algorithmic creatures, and the computer is an algorithmic creation. It is not a stretch to say that even walking is an algorithm although we certainly do not think of its "well-defined computational" components.

It is, thus, clear why algorithms are ubiquitous across all disciplines. Facebook was built by computer scientists on top of complicated graph and network algorithms. The Internet was made possible through carefully laid out protocols and procedures. Engineers use an array of optimization algorithms to maximize profit, minimize costs, and fine-tune processes. Geneticists also employ specialized algorithms to sequence genes and analyze mutations of viruses.

Prepared by ***Mark Edward M. Gonzales***

WHAT IS A GOOD ALGORITHM?

A good algorithm has two important characteristics:

1. It must be **correct**.
2. It must be **efficient**.

Throughout the AIEP training sessions, we have discussed how to create algorithms and translate them into syntactically and semantically correct codes (that is, they conform to the "grammar" of Python and accomplish the given task). However, doing things correctly is not enough. We want everything, from booting up our machines to browsing the Internet, to run fast.

Ergo, the focus of the latter half of our send-off program is the creation not only of correct but also of efficient algorithms.

HOW DO WE QUANTIFY EFFICIENCY?

Consider the following scenario involving two runners Jack and Jill. The table below summarizes the time it took them to travel several distances:

Went up the Hill	1 km	3 km	20 km
Jack	20 minutes	110 minutes	5000 minutes
Jill	40 minutes	100 minutes	2000 minutes

If we want either one of them to fetch us a pail of water 1 km away, then we would certainly choose Jack. However, as the pail of water inches further and further away, the clear winner becomes Jill. This illustrates an important point about the efficiency of algorithms (time efficiency): generally, we prefer algorithms that are more "stable" even as the input size increases. In more formal terms, we want slower rates of growth.

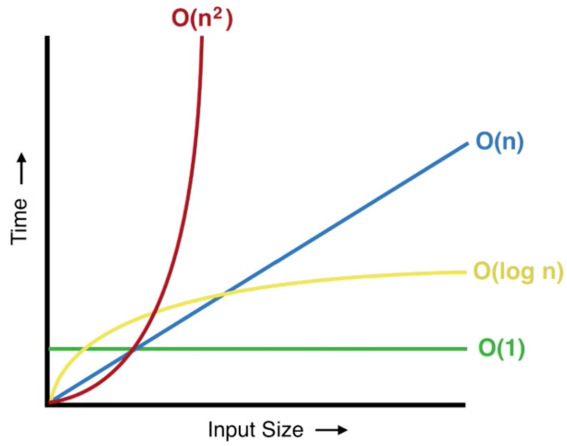
In reality, efficiency is extremely difficult to define. Are there any boundaries for what is deemed to be "stable"? Nonetheless, our interim definition is enough for our present purposes.

The Big O

One of the ways to quantify efficiency is through the so-called big O notation (also spelled as *big Oh*), which describes the upper bound of the growth rate. We call it an **asymptotic** measurement since we are considering the performance as the input size becomes astronomically large, that is, as it tends to infinity.

In mathematical terms, when we say that $O(f(n)) = g(n)$, then we mean that there exist positive numbers c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. *(Note that there are other asymptotic measures, such as big omega and big theta. Moreover, observe that the equals sign is misleading; it is a good example of what constitutes an "abuse of notation.")*

For our present purposes, there is no need for us to delve into the rigors of the big O notation. Bear in mind, however, that some programming competitions specify a big O requirement; therefore, it is important to have at least an intuitive feel of what it means.



https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59

An important thing to remember is that $\log n$ in computer science is understood to be $\log_2 n$ (which is different from the convention in mathematics). This makes sense because computers operate in binary. We often categorize algorithms based on their big O, as seen in the non-exhaustive table below:

Big O	Name	Remarks
$O(1)$	Constant	**Best!** Performance is independent of input size
$O(\log n)$	Logarithmic	**Usually** involves halving the input size
$O(n)$	Linear	**Usually** involves 1 loop
$O(n \log n)$	Linearithmic	Goal of **most** divide-and-conquer algorithms
$O(n^2)$	Quadratic	**Usually** involves 2 nested loops
$O(n^3)$	Cubic	**Usually** involves 3 nested loops
$O(2^n)$	Exponential	**Impractical!** Growth rate is rapid and intolerable

We discuss some examples to concretize our understanding of big O and how it relates to algorithm efficiency. We also insert some basic principles on how to reduce the running time of an algorithm.

Finding the Sum of the First n Positive Integers

A. $O(n)$ Solution

This can be solved iteratively:

```
def sum_n_pos_linear(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + i
    return sum
```

As the input size, that is, the number of positive integers that we are summing, increases, the algorithm also takes more time (to be precise, in a linear proportion) to return the sum. Although this "inefficiency" is not discernible for small values of n, it will become apparent as we try to increase n to, say. 2^{2048}.

B. $O(1)$ Solution

This requires some degree of mathematical insight.

```
def sum_n_pos_constant(n):
    return n * (n + 1) // 2
```

The formula discards the need for a loop. No matter how big n is, it just gets plugged into the formula to return the sum. The key takeaway here is that, if a problem can be solved by deriving a mathematical formula, then this formula should be used to achieve constant-time complexity.

Searching for a Number in a Sorted List

Given a key, determine whether it can be found in the sorted list. If it can be found, then return the key. Otherwise, return -1.

A. $O(n)$ Solution (Linear Search)

The naive approach is to iterate over the entire list and compare each element with the key.

```
def linear_search(array, key):
    for i in range(len(array)):
        if (array[i] == key):
            return i
    return -1
```

B. $O(\log n)$ Solution (Binary Search)

Consider this exchange from the children's game "Guess a Number."

Player 1: Is it 5?

Player 2: No.

Player 1: Larger?

Player 2: Yes.

Player 1: Okay, so is it 10?

Logically, Player 1 will not guess anything smaller than or equal to 5. We use this same principle in implementing binary search.

The iterative implementation is as follows:

```
def binary_search_iterative(array, key):
    left = 0
    right = len(array) - 1

    while left <= right:
        mid = (left + right) // 2
        if key == array[mid]:
            return mid
        elif key < array[mid]:
            right = mid - 1
        else:
            left = mid + 1

    return -1
```

Binary search can also be implemented recursively. However, if something can be done iteratively, then we generally prefer the iterative version since recursion builds a stack of calls, "abusing" the computer memory and potentially leading to stack overflow. Nonetheless, we present it here for completeness:

```
def binary_search_recursive(array, left, right, key):
    if left <= right:
        mid = (left + right) // 2
        if key == array[mid]:
            return mid
        elif key < array[mid]:
            return binary_search_recursive(array, left, mid - 1, key)
        else:
            return binary_search_recursive(array, mid + 1, right, key)
    else:
        return -1
```

The advantage of binary search is that, at every iteration, the "size" of the list being searched is halved. If the number belongs to the right half, we only need to search the right half; the left half is simply disregarded. The same goes if the number belongs to the left half. This is an $O(\log n)$ algorithm in its worst case.

Will binary search work if the list is not sorted? No.
Will linear search work if the list is not sorted? Yes.
Should I sort the list before searching it? It depends. Note that sorting also takes some time (an entire subset of the study of data structures and algorithms revolves around sorting). If we are just going to search the list once and never again, sorting it might not be too good of an idea. However, if we are going to search the list repeatedly, then sorting might improve efficiency.

In Session 10, we are going to take a cursory look at a hash map (Python dictionary) as an efficient data structure for searching.

Raising a Number to a Nonnegative Integer Power

Python is unique among programming languages in the sense that it has a built-in exponentiation operator. For instance, in C and C++, we still need to include the header files `math.h` and `cmath`, respectively. In this section, we will try to implement our own function that will raise a number to a nonnegative integer power.

A. $O(n)$ Solution

The naive solution is to multiply the number by itself $n-1$ times, where n is the exponent. *Is this really what we do when we compute for the value by hand?*

```
def power_linear(base, exp):
    answer = 1
    for i in range(exp):
        answer = answer * base
    return answer
```

B. $O(\log n)$ Solution

This is one of the interesting cases wherein the recursive version is better than the iterative version, at least in the context of time efficiency. The inspiration for our optimized code comes from two mathematical observations:

1. $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$ if n is even
2. $x^n = x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x$ if n is odd

To see how this algorithm reduces the number of operations, we consider how 6^{21} is computed:

- $6^{21} = 6^{10} \times 6^{10} \times 6$
- $6^{10} = 6^5 \times 6^5$
- $6^5 = 6^2 \times 6^2 \times 6$
- $6^2 = 6 \times 6$

```
def power_log(base, exp):
    if exp == 0:
        return 1
    if exp % 2 == 0:
        return power_log(base * base, exp // 2)
    else:
        return power_log(base * base, exp // 2) * base
```

Finding the Largest Continuous Sum

Adapted from a problem found on a sample paper of the Global Mathematics Coding Competition (GMCC).

Given a list of n integers, find the maximum sum of consecutive segments (an empty segment is allowed). Here are some sample runs:

- For the list $[1, 2, 3]$, the maximum is $1 + 2 + 3 = 6$.
- For the list $[5, -6, 7, -2, 3, -5, 4]$, the maximum is $7 + (-2) + 3 = 8$.
- For the list $[-1, -2, -3]$, the maximum is 0 (just sum up none of them)

A. $O(n^3)$ Solution

Take note of the three nested loops (highlighted) that slow down the algorithm:

```
def largest_cont_sum_cubic(array):
    largest_sum = 0

    for i in range(len(array)):
        for j in range(i, len(array)):
            current_sum = 0
            for k in range(i, j + 1):
                current_sum = current_sum + array[k]
            if current_sum > largest_sum:
                largest_sum = current_sum

    return largest_sum
```

Can you spot the redundancy in the cubic-time algorithm above?

B. $O(n^2)$ Solution

We can reduce the number of nested loops to two (highlighted):

```
def largest_cont_sum_quadratic(array):
    largest_sum = 0

    for i in range(len(array)):
        current_sum = 0
        for j in range(i, len(array)):
            current_sum = current_sum + array[j]
            if current_sum > largest_sum:
                largest_sum = current_sum

    return largest_sum
```

There is an $O(n \log n)$ solution that uses a technique known as "divide and conquer." However, the discussion of this strategy is deferred to Session 8.

C. $O(n)$ Solution

The most efficient solution to this problem surprisingly takes only linear time (with just a single loop) although the underlying idea (dynamic programming) is uncannily clever, and it does take some effort to be convinced of its validity:

```
def largest_cont_sum_linear(array):
    largest_sum = 0
    current_sum = 0

    for i in range(len(array)):
        current_sum = current_sum + array[i]
        if current_sum > largest_sum:
            largest_sum = current_sum
        elif current_sum < 0:
            current_sum = 0

    return largest_sum
```

The point is that, although efficient algorithms are not immediately intuitive and may even seem like sorcery, they are extremely rewarding once unearthed!

WHAT ARE BITWISE OPERATORS?

Inside the computer, everything is just a bunch of 1s and 0s. Programming languages provide a certain class of operators to exploit the "affinity" of our machines for binary digits. These are known as bitwise operators, with "bit" standing for "binary digit."

The table below presents the bitwise operators in Python. The first entry (bitwise NOT) is the operator with the highest precedence (or priority during evaluation).

	Operator	Name	Description	Example
1	~	Bitwise NOT	"Flips" all the bits	~60 = –61 *Why?* 60 = 0011 1100b 1100 0011b = –61 Please refer to NOT SO FAST! Caveat #3.

2	<<	Bitwise left shift	The bits of the left operand are moved to the left by the number of bits specified by the right operand	60 << 2 = 240 *Why?* 60 = 0011 1100b 1111 0000b = 240
	>>	Bitwise right shift	The bits of the left operand are moved to the right by the number of bits specified by the right operand	60 >> 2 = 15 *Why?* 60 = 0011 1100b 0000 1111b = 15
3	&	Bitwise AND	Returns 1 if and only if both bits are 1	60 & 15 = 12 *Why?* 60 = 0011 1100b 15 = 0000 1111b ========== 0000 1100b = 12
4	^	Bitwise XOR	Returns 1 if and only if both bits are different	60 ^ 15 = 51 *Why?* 60 = 0011 1100b 15 = 0000 1111b ========== 0011 0011b = 51
5	|	Bitwise OR	Returns 0 if and only if both bits are 0	60 | 15 = 63 *Why?* 60 = 0011 1100b 15 = 0000 1111b ========== 0011 1111b = 63

WHY ARE BITWISE OPERATORS IMPORTANT?

Bitwise operators allow us to take advantage of how computers internally represent data as binary numbers. This makes our code run faster since it is a "bit closer" to the hardware itself. This is very important, especially for embedded systems (those that run microwaves, medical devices, etc.) that require as much optimization as possible due to speed and memory constraints. For these machines, every bit counts.

There are also some cool "hacker-esque" techniques — which run faster than simply using the "boring" non-bitwise operators — that can be done with bitwise operators:

1. Check if a number is odd or even

```
def is_num_odd(num):
    return num & 1
```

2. Divide a number by 2

```
def divide_by_2(num):
    return num >> 1
```

3. Multiply a number by 2

```
def multiply_by_2(num):
    return num << 1
```

4. Check if two numbers are equal

```
def check_equality(num1, num2):
    return num1 ^ num2
```

5. Convert a letter to its uppercase equivalent

```
def to_uppercase(ch):
    return chr(ord(ch) & ~32)
```

Remark: Compared to other languages, such as C or C++, where `char` *is a subset of* `int`*, Python needs to invoke* `ord()` to obtain the ASCII value of a character and `chr()` to obtain the character given the ASCII value.

NOT SO FAST!

Before we end our discussion of algorithm efficiency and bitwise operators, we offer three caveats:

1. Correctness comes before efficiency. Design a working algorithm first, then think about how we can make them faster — not the other way around. Remember that writing a fast-performing algorithm takes time.
2. The order of operations in Python differs from that in C or C++ when it comes to bitwise operators. This may cause a bit of confusion.
    a. Python operator precedence
    https://docs.python.org/3/reference/expressions.html
    b. C operator precedence
    https://en.cppreference.com/w/c/language/operator_precedence

3. The representation of negative numbers inside the computer memory involves a lengthy discussion of the so-called "two's complement." Consequently, bitwise operations with negative integers may seem non-intuitive.

   This behavior is explained in https://wiki.python.org/moin/BitwiseOperators.

EXERCISES

1. Write an $O(1)$ algorithm to accomplish the following tasks:
   a. Find the sum of the squares of the first n positive integers.
   b. Find the sum of the cubes of the first n positive integers.
   c. Find the maximum number of points at which n lines intersect.
   d. Find the number of diagonals of a polygon with n sides. (Hint: The diagonals are the line segments that connect two vertices of a polygon — except for the sides.)

2. Write an $O(\log n)$ algorithm to compute for the greatest common factor of two positive integers. (Hint: This is achieved using the Euclidean algorithm.)

3. Use the idea of binary searching to calculate the floor function of the square root of a positive integer n in $O(\log n)$ time without using any predefined function (like `math.sqrt()`, `math.floor()`, or the `**` operator). (Hint: The floor function of a number is the greatest integer that is less than or equal to the given number.)

For nos. 4 & 5, the solution must run in $O(n)$ time and take only a single look/pass at the values.

4. In the actual Global Mathematics Coding Challenge (GMCC) question, each segment must have at least one element (so that an empty sum is NOT allowed). Modify our solution to the largest continuous sum problem to accomplish this.

5. Write an algorithm to take the largest continuous product instead. Note that we can have an "empty" product, defined to be 1. (Caveat: Be careful since the product of two small negative numbers results in a large positive number. For example, $(-10)\times(-12)$ is 120.)

6. Use bitwise operators to accomplish the following tasks:
   a. Convert a letter to its lowercase equivalent.
   b. Check if a number is a power of 2. (Caveat: Be careful since 0 is NOT a power of 2.)
   c. Divide a number by a power of 2.
   d. Multiply a number by a power of 2.
   e. A Mersenne prime is a prime number of the form $2^p - 1$, where p is also a prime number. Generate the first 8 Mersenne primes.

7. Insertion Sort. Given below is the implementation of an algorithm used to sort a set of data values, known as insertion sort. Study the code, and answer the questions that follow:

Line of Code	Frequency Count
`def insertion_sort(array):`	NOT COUNTED
`    for i in range(1, len(array)):`	n
`        element = array[i]`	$n-1$
`        j = i`	$n-1$
`        while j > 0 and array[j - 1] > element:`	$\frac{n^2+n-2}{2}$
`            array[j] = array[j - 1]`	$\frac{n^2-n}{2}$
`            j = j - 1`	$\frac{n^2-n}{2}$
`        array[j] = element`	$n-1$

   a. What real-life sorting scheme/scenario is similar to insertion sort?
   b. Study the frequency count table above (assume that n is the number of values to be sorted). The frequency count refers to the number of times each line of code is executed assuming worst case.
   c. Knowing the total frequency count gives a more formal way to determine the big O of an algorithm (rather than just having an intuitive feel for it based on the loops). What is the big O of insertion sort in its worst case?
   d. Interested trainees may research on Shellsort, an efficient generalization of insertion sort. As a challenge, try implementing it using Python.

8. Selection Sort. Another algorithm to sort a set of data values is selection sort (also known as straight selection sort). The idea of the algorithm is as follows:
   - Look for the smallest value in the list and place it as the first element.
   - Look for the second smallest value and place it as the second element.
   - Look for the third smallest value and place it as the third element.
   - Continue this process until the entire list is sorted.

a. Implement selection sort using Python.
b. What is the big O of selection sort in its worst case?
c. What makes selection sort inefficient?
d. Does the order of the data values in the input affect the performance of the algorithm?

9. **Swapping Four Ways.** Swapping two numbers is an introductory challenge given to programming "neophytes" to test their understanding of variables. In this problem, we are going to spice things up a bit and explore four such ways:
   a. Write an algorithm to swap two numbers (freestyle).
   b. Write an algorithm to swap two numbers using only two variables.
   c. Write an algorithm to swap two numbers using bitwise operators.
   d. Python is an interesting programming language that has its set of tricks, called "idioms." A code that makes good use of these "idioms" is described as **Pythonic**. The Pythonic way to swap is as follows:

      $$x, y = y, x$$

      Be careful when using this! Other programming languages, such as C, C++, and Java, do not support this idiom.

10. **Every Day I'm Shuffling.** Write an $O(n)$ algorithm that (randomly) shuffles a given list of numbers. (Hint: You can import the `random` module and use the `random.randint()` function to generate random integers. Refer to https://docs.python.org/3/library/random.html)

`-- return 0; --`