RMIT University

Algorithm Implementation

COSC2658 Data Structure & Algorithm

Lecturer: Dr. Yossi Nygate and Dr. Minh Dinh
Author: Huynh Nguyen Nguyen
ID: s3694703

November 22 2020

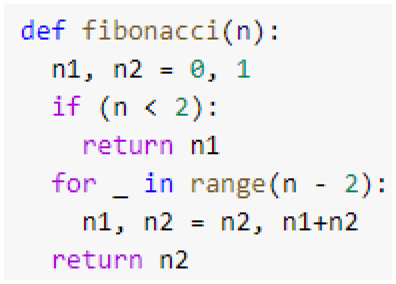
Table of Contents

List of Figures

List of Tables

1. Question 1

1.1. Develop a function to calculate the nth Fibonacci number using iteration



```
def fibonacci(n):
  n1, n2 = 0, 1
  if (n < 2):
    return n1
  for _ in range(n - 2):
    n1, n2 = n2, n1+n2
  return n2
```

Figure 1. Fibonacci program screenshot

In the above function, the value of Fibonacci number in nth position is calculated by taking the sum of its two previous values. Therefore, this approach makes use of for loop to iterate and compute all Fibonacci numbers in order until it reaches nth position as shown in the Figure.

In detail, the above function takes an input parameter of *"n"* to indicate the position of the return value in the sequence. By convention, the two leading numbers in Fibonacci sequence have values of 0 and 1, respectively. Hence, when input is either 1 or 2, the result can be returned directly without going through the loop. For inputs greater than 2, the process only needs repeat for n – 2 times (minus the two first numbers).

1.2. Using formal analysis on your code, what is the big O for calculating F(n). Explain/justify your answer

According to the Theoretical analysis of time efficiency, the running time of an algorithm $T(n)$ can be estimated using the following formula:

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

where: n is input size.

c_{op} is the amount of time spending for execution or the cost of the operation.

$C(n)$ is the number of times that the basic operation is executed.

Analyzing the function which computes nth Fibonacci number using iteration, we have:

- Input size: n
- Basic operations: Assignment, Comparison, Addition

Approach 1: TAKING THE MOST BASIC OPERATION

Since Fibonacci sequence is formed continuously by adding up two previous numbers, the addition operation can be considered as the most crucial operation among the three mentioned operation. Hence, the equation of the number of times the addition used - $C(n)$ - can be expressed as shown:

$$C(n) = \begin{cases} 0, & for\ x \le 2 \\ \sum_{i=1}^{n-2} 2 = 2 * (n-2), & for\ x > 2 \end{cases}$$

- As the first two Fibonacci number have been pre-defined, the function can return result without any addition needed when $x \leq 2$; thus, $C(n) = 0$.
- For $x > 2$, the function iterates for n – 2 times and each iteration execute two additions. One is for adding two numbers to get the next value. The other addition is the increment of index value in the for loop itself.

In sum up, as concentrating on the order of growth when $n \to \infty$, the case where $x \leq 2$ can be ignored. Applying the formula $(*)$, the running time equation is:

$$T(n) \approx c_{op} * 2 * (n - 2) \approx \mathbf{2c_{op}n - 4c_{op}}$$

Approach 2: TAKING ALL BASIC OPERATIONS

The formula (*) can be updated as follows to consider all operations such that the runtime of a function is the sum of all operations' execution time contained in that function.

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

Assuming that all operations take roughly the same amount of time, the following table describes the type of operations performed in each LOC as well as its cost and occurrence depending on given input – n.

Table 1. Fibonacci program theoretical time analysis

	OPERATION	COST - c_{op}	OCCURENCE - $C(n)$
`def fibonacci(n):`			
`    n1, n2 = 0, 1`	2 Assignments	$2c_{op}$	1
`    if (n < 2):`	Comparison	c_{op}	1
`        return n1`			
`    for _ in range(n - 2):`	Assign/Add, Comparison	$2c_{op}$	n – 1
`        n1, n2 = n2, n1+n2`	Addition & 2 Assignments	$3c_{op}$	n – 2
`    return n2`			

$$T(n) \approx 2c_{op} * 1 + c_{op} * 1 + 2c_{op} * (n - 1) + 3c_{op} * (n - 2)$$

$$T(n) \approx \mathbf{5c_{op} * n - 5c_{op}}$$

In conclusion, both approaches state that the Growth-rate notation of the Fibonacci function is **O(n)**.

1.3. Using empirical analysis, generate a formula that estimates the time to calculate F(n)

Since the actual runtime of a program can vary in different software or hardware environments, the function will be run for multiple times to gain more accurate result.

Table 2. Fibonacci program runtime

INPUT	RUN 1	RUN 2	RUN 3	RUN 4	AVERAGE
1000	0.13	0.14	0.08	0.08	0.11

2000	0.26	0.33	0.20	0.19	0.24
4000	0.51	0.63	0.52	0.49	0.54
8000	1.83	1.71	1.91	1.85	1.83
12000	3.00	2.94	3.89	4.11	3.49
16000	5.21	6.17	6.41	4.99	5.70
20000	7.02	8.64	7.28	7.25	7.55

Hence, given the average runtime of each input, the graph below shows the Fibonacci program tends to grow at linear rate following the equation $y = 0.0004x - 0.7955$.

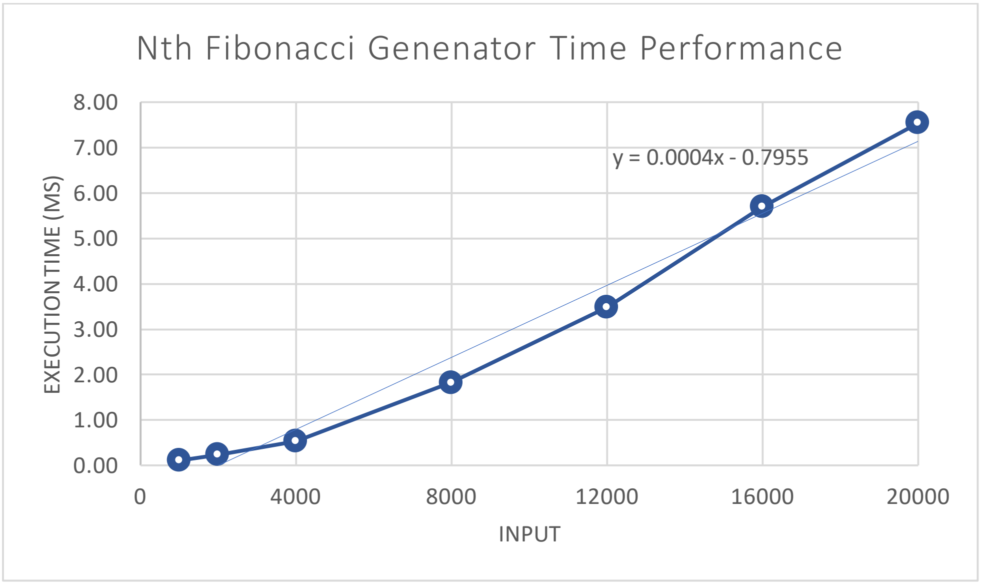


Figure 2. Line graph of Fibonacci program time performance

Justify linear regression result:

In theory, given the linear growth, as the input size is doubled, the execution time is expected to be doubled either. Examining our experimental result, we have:

- As we double $n_1 = 1000$ to $n_2 = 2000$, the execution time increases $\frac{t_2}{t_1} = \frac{0.24}{0.11} = 2.2$ times.
- As we double $n_1 = 2000$ to $n_2 = 4000$, the execution time increases $\frac{t_2}{t_1} = \frac{0.54}{0.24} = 2.25$ times.
- As we double $n_1 = 4000$ to $n_2 = 8000$, the execution time increases $\frac{t_2}{t_1} = \frac{1.83}{0.54} = 3.4$ times.
- As we double $n_1 = 8000$ to $n_2 = 16000$, the execution time increases $\frac{t_2}{t_1} = \frac{5.7}{1.83} = 3.1$ times.
- As we triple $n_1 = 4000$ to $n_2 = 12000$, the execution time increases $\frac{t_2}{t_1} = \frac{3.49}{0.54} = 6.5$ times.
- As we increase $n_1 = 2000$ 10 times to $n_2 = 20000$, the execution time increases $\frac{t_2}{t_1} = \frac{7.55}{0.24} = 31$ times.

As shown, the runtime does not grow linearly as expected. The further evaluation for this empirical result will be discussed below.

1.4. Plot a graph that compares your code's performance versus the performance predicted by the empirical analysis for n = 1000, 10,000, 20,000, etc. Continue to do so while the performance is still reasonable

Table 3. Fibonacci program runtime in second check

INPUT	PREDICTED RESULT	ACTUAL RESULT	DEVIATION
1000	-0.395	0.086	N/A
10,000	3.205	3.069	-0.136
20,000	7.205	8.624	1.420
30,000	11.205	17.625	6.421
40,000	15.205	28.832	13.628
50,000	19.205	43.195	23.991
60,000	23.205	59.83	36.626
70,000	27.205	76.13	48.926
80,000	31.205	99.1	67.896
90,000	35.205	120.77	85.566
100,000	39.205	148.97	109.766

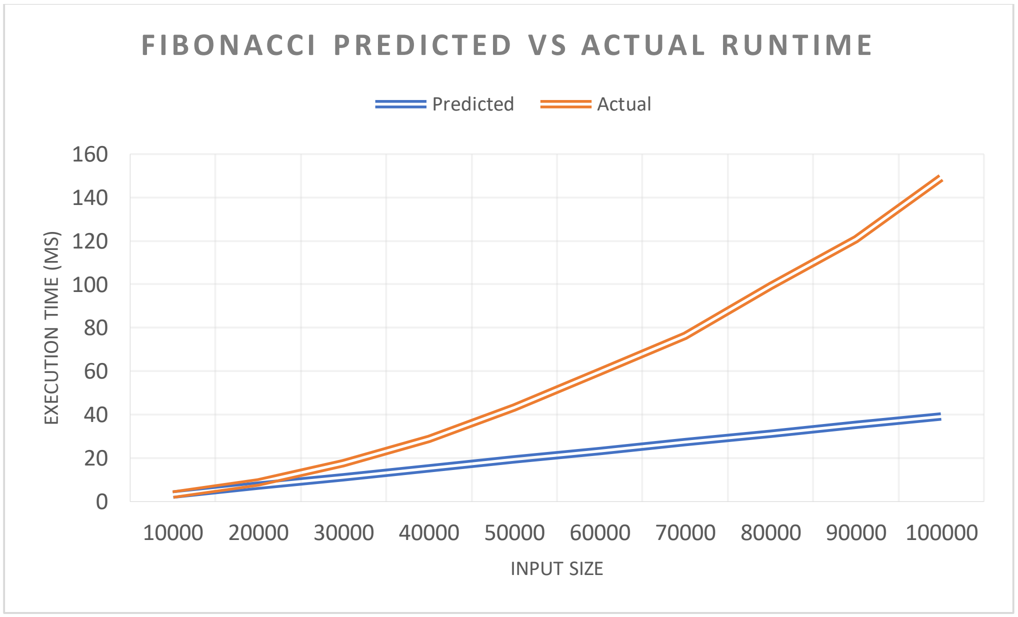


Figure 3. Line graph of predicted vs actual Fibonacci runtime

1.5. Did the performance track the formula you predicted. Can you characterize where the performance did not match your predictions? Can you explain why?

To verify once again the correctness of the equation $y = 0.0004x - 0.7955$ in the runtime prediction, the program was re-run with larger input values. As can be observed from Table …., this equation is completely inconsistent with the actual value as the larger the input size, the greater the distance between the predicted and actual results. Furthermore, the plotted graph shows that in practice the execution time for generating the nth Fibonacci number grows with a quadratic rate rather than in a linear way as in theoretical analysis.

As for the explanation for this behavior, it lays upon the assumption in the previous theoretical analysis that all the operations take roughly the same amount of time. However, in reality, regarding the addition operation $n_1 + n_2$ to generate the next Fibonacci, its time complexity is not $O(1)$ [1]. According to the documentation for the integer implementation [2], Python 3 offers arbitrary precision integer type to support an unlimited number of digits. As in arbitrary-precision arithmetic [3], the runtime for each addition operation corresponds to the length of the longest operand – $O(n)$. In result, since the addition operation happens n times and has $O(n)$ as its order of growth, generating the nth Fibonacci number requires to have $O(n^2)$ time complexity in practice.

1.6. Describe two challenges that would you need to address to calculate the 1,000,000th Fibonacci number.

In calculating 1,000,000th Fibonacci number or the large Fibonacci number in general, depending on the approach chosen, the challenges will differ but still concentrate on two major aspects: time and space. The following table will discuss the challenges of using traditional approach $F_n = F_{n-1} + F_{n-2}$ and Binet formula $F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$ [4].

Table 4. Time and Space comparison between traditional and Binet formula for Fibonacci program

	TRADITIONAL APPROACH	**BINET FORMULA**
Time Complexity	As discussed previously, this addition approach might have quadratic growth rate in practice when the target value becomes bigger.	Providing constant access time
Space Complexity	Space is not a problem as Python supports unlimited integer with space optimization method by saving it as an array of digits in base 2^{30} [5].	Binet formula consists of golden ratio which is a decimal value which only can be estimated roughly by the binary fractions [6]. Hence, to produce the exact value using Binet formula, it is required with the higher precision of floating point; thus, increasing storage cost.

1.7. Can you find a way to estimate the 1,000,000th Fibonacci number?

For estimating the 1,000,000th Fibonacci number without any programming or calculator, Binet formula can be utilized as it provides the general equation for calculating the nth term in Fibonacci sequence [4]:

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} \ (1)$$

Where: F_n is the nth term in Fibonacci number

n is the position of value in Fibonacci sequence

ϕ is the golden ratio, $\phi = \frac{1+\sqrt{5}}{2}$

The (1) equation can be re-written as:

$$F_n = \frac{\phi^n}{\sqrt{5}} - \frac{1}{\sqrt{5}(-\phi)^n} = \frac{1}{\sqrt{5}}A + B \ \ (2)$$

Where: $A = \phi^n$

$B = -\frac{1}{\sqrt{5}(-\phi)^n}$

Estimate B

Based on basic limit formula, as n approaches ∞, c^n equation where c is a constant will become greater and approach ∞. Hence, since $\frac{1}{\infty} \approx 0$, $\frac{1}{c^n}$ will get closer to 0 when n is an extraordinarily considerable number.

$$\lim_{n\to\infty} c^n = \pm\infty \Longrightarrow \lim_{n\to\infty} \frac{1}{c^n} = 0$$

Considering the (2) equation, as in our case, the n value is enormous – $n = 10^6$, the B part can be neglected.

$$\lim_{n\to\infty} B = \frac{-1}{\sqrt{5}} * \lim_{n\to\infty} \frac{1}{(-\phi)^n} = \frac{-1}{\sqrt{5}} * 0 = 0$$

Hence, the equation for 1,000,000th Fibonacci number will be $\frac{1}{\sqrt{5}}A$

Estimate A

As $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6$, A can be re-written:

$$A = \phi^n \approx \left(\frac{16}{10}\right)^n = \frac{16^n}{10^n}$$

With $n = 1000000$, we have: $16^{10^6} = 2^{4*10^6} = 2^{10*4*10^5} \approx 10^{3*4*10^5} = 10^{12*10^5}$

Applying the above calculated result into equation $A \approx \frac{10^{12*10^5}}{10^{10^6}} = 10^{2*10^5}$, the 1,000,000th Fibonacci number is:

$$F_{10^6} = \frac{1}{\sqrt{5}}A \approx \frac{1}{\sqrt{5}} * 10^{2*10^5} \approx \frac{10^{2*10^5}}{2}$$

In conclusion, the estimation result states that the 1,000,000th Fibonacci number is approximately half 10^{2*10^5} and contains around $2 * 10^5 = 200{,}000$ digits which is close to the actual result conducted by the program – 208 988 digits [7].

2. Question 2

2.1. Your boss asks you to determine how many prime numbers there are between 10^9 and 10^{12} using the algorithm we described in class. How long would it take your code to run?

Since 10^{12} is an enormous number compared to 10^9, the number of prime numbers between 10^9 and 10^{12} can be estimated by the number of prime numbers less than 10^{12}. To solve, we do the trial division method as discussed in class to test the primality of each number from 1 to 10^{12}:

- Step 1: If n is 1, n is not a prime number. If n is 2 or 3, n is a prime number.
- Step 2: If n is divisible by 2 and 3, it is not a prime number. Otherwise, continue.
- Step 3: Check that n is divisible by any number with the form $6k \pm 1$. If yes, n is not a prime. Until $6k \pm 1 \geq \sqrt{n}$, stop the program and conclude that n is a prime number.

By running the program with different inputs, the result is as shown:

Table 5. Prime counting runtime

INPUT	X = LOG10(INPUT)	Y = LOG10(TIME)	RUNTIME (MS)
10^3	3	-0.189096	0.647
10^4	4	0.9265482	8.444
10^5	5	2.0072355	101.68
10^6	6	3.3621689	2302.337
10^7	7	4.8021661	63411.225
10^8	8	6.2060186	1607010

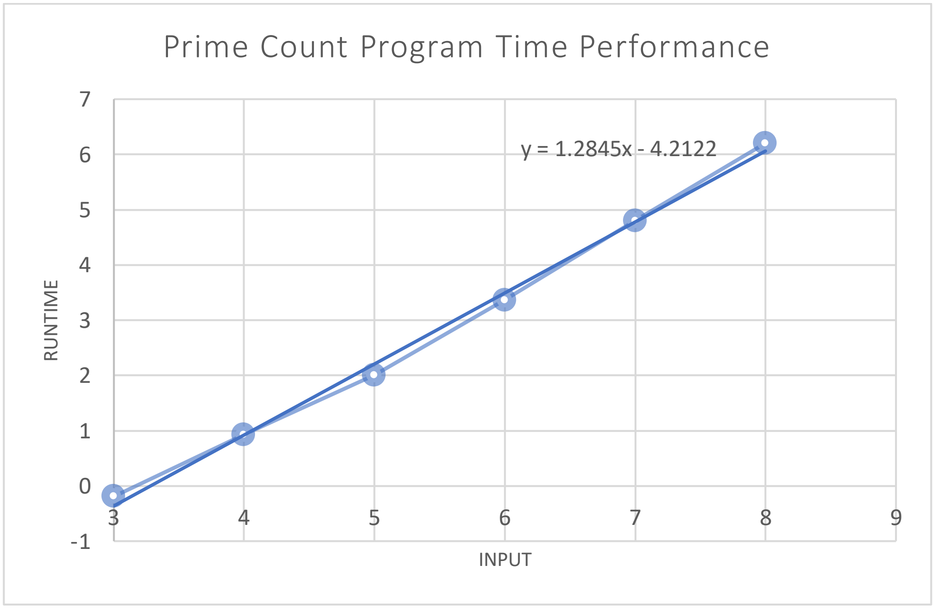


Figure 4. Line graph in log scale for prime count time performance

Since the chart is plotted in Logarithmic scale, the straight line indicates that a power relationship between input and runtime. Also, as $y = 1.2845 * 12 - 4.2122 = 11.2$, the time taken for find all primes less than 10^{12} is

$$time = 10^{11.2} = 1.58 * 10^{11}\ (millisecond) = \frac{15.8 * 10^7\ (second)}{\pi * 10^7} \approx 5\ years$$

Hence, as 10^{12} is too enormous compared to 10^9, the amount of time to count primes between 10^9 and 10^{12} is also approximately 5 years when using the algorithm discussed in class.

2.2. Research to find the performance of the best algorithms to determine if a number is prime? How does this compare to the approach above?

The best algorithm for primality test will depend on the magnitude of input size and its application. If doing the primality test for a small number, the Trial Division discussed is good enough which provides $O(\sqrt{n})$ where n is the target value. For those giant numbers with hundreds of digits, there are two possible approaches: Probabilistic and Deterministic test. If the speed of the program is preferred, some Probabilistic algorithm would be applicable such as the Fermat method has $O(k * \log n)$ [8]. Otherwise, it is possible to use the deterministic algorithm - AKS primality test which is currently considered as the best primality test algorithm. The AKS test provides 100% certainty and has a polynomial time expression over the number of digits in the input number, not the input value [9].

However, applying into the earlier problem, although AKS method offers both 100% accuracy and faster growth rate, its constant of proportionality is noticeably big compared to our target range $1 - 10^{12}$; thus, creating hidden cost in running time and space.

2.3. Research to find the performance of the best algorithm to determine the divisors of a number? How does this surprise you?

The best algorithm to find all divisors of a number I can find is Trial Division. This algorithm shows the similarity to the primality test one as its operation is to traverse through and test all integers from 1 to $\sqrt{n}$ as a divisor. The time complexity is $O(\sqrt{n})$ [10].

What surprise me is the fact that I could not find any algorithm better than Trial Division for this problem. At first, I thought of doing the Prime Factorization to get the list of primes and forward compute the divisors. However, this approach faces the issue with the Prime Factorization algorithm since there is no universal solution for all integers instead the best solution will be based on the size of the number. Therefore, in general, the Trial Division will be the best resolution which supplies a straightforward answer. In a specific situation, the conclusion for the best approach may be re-evaluated depending on the size of the inputs.

2.4. If your boss wants to know approximately how many primes there are between 10^9 and 10^{12} can you find a better approach?

As a consequence of Prime Number Theorem [11], the number of primes less than or equal to x – called as $\pi(x)$ – can be estimated by the equation:

$$\pi(x) \sim \frac{x}{\ln(x)}$$

$$\pi(10^{12}) \approx 10^{12} * \frac{\log(e)}{\log(10^{12})} = 10^{12} * \frac{\log(e)}{12} \approx 36 * 10^9$$

Hence, as 10^{12} is too enormous compared to 10^9, the number of primes between 10^9 and 10^{12} is approximately $36 * 10^9$.

3. Question 3

The discussion below focuses on the complexity of the best algorithm to solve the following problems:

3.1. Display all the even integers in an array of length n

Table 6. "Display all integers" program theoretical time analysis

PSEUDOCODE	OPERATION	COST	OCCURENCE
For each integer in the array	Assign, Comparison	$2c$	$n+1$
If integer is even	Comparison	c	n
Print integer		c	$[0, n]$

Worst case: $T(n) = 2c * (n+1) + c * n + c * n = \mathbf{4c * n + 2c}$

Best case: $T(n) = 2c * (n+1) + c * n + c * 0 = \mathbf{3c * n + 2c}$

Hence, in both situations, the function has a linear growth-rate **O(n)**.

3.2. Compute the sum of the first n even integers

To sum the first n even integers, we must traverse each element in the array and find enough even numbers. Hence, in the worst-case scenario, we must examine all elements. In other words if we called:

l is the length/the number of elements in the array

n_e is the number of even integers in the array.

n is the input standing for the number of even integers need to be found

According to the correlation of the input n and the actual number of even integers in the array - n_e, the execution time of this function can be represented by varied factors.

- Case 1: When $n \leq n_e$, the time complexity of this function will depend on n.
- Case 2: When $n > n_e$, the time complexity of this function will depend on l (Worst case).

Table 7. "Sum first n even integers" program theoretical time analysis

PSEUDOCODE	OPERATION	COST	OCCURENCE CASE 1	OCCURRENCE CASE 2
Set count to 0	Assignment	c	1	1
Set sum to 0	Assignment	c	1	1
For each integer in the array	Assign, Comparison	$2c$	n	$l+1$
If integer is even	Comparison	c	n	l
Increase count by 1	Addition	c	n	$[0, l]$

Add integer to sum	Addition	c	n	$[0, l]$
If count is equal to n	Comparison	c	n	l
Break the loop		c	1	1
Return result which is sum		c	1	1

Case 1: $T(n) = c * 1 + c * 1 + 2c * n + c * n + c * n + c * n + c * n + c * 1 + c * 1 = \boldsymbol{6c * n + 4c}$

Case 2 (Worst case): $T(l) = c * 1 + c * 1 + 2c * (l + 1) + c * l + c * l + c * l + c * l + c * 1 + c * 1 = \boldsymbol{6c * l + 6c}$

Hence, based on the worst-case scenario, the function has a linear growth-rate **O(n)** with respect to the array length.

3.3. Display one element in an array

Table 8. "Display on element" program theoretical time analysis

PSEUDOCODE	OPERATION	COST	OCCURENCE
Get element value by its index	Assignment	c	1
Print out element value		c	1

Since the element of an array can be accessed directly by its index/position, the execution time for this operation does not depend on the magnitude of input value.

$$T(n) = c * 1 + c * 1 = 2c.$$

Hence, the function has constant growth-rate **O(1)**.

3.4. Search an unsorted array of length n for a particular item

Table 9. "Search in unsorted array" program theoretical time analysis

PSEUDOCODE	OPERATION	COST	OCCURENCE
For each item in the array	Assign, Comparison	$2c$	$n + 1$
If item is the target item	Comparison	c	$[1, n]$
Return item		c	1

When the target item is at the end of the array or not in the array, the function must iterate through all elements in the array. Thus, in worst case, $T(n) = 2c * (n + 1) + c * n + c * 1 = \boldsymbol{3c * n + 3c}$

The best case is when the target item is the first element in the array which has $T(n) = 2c * (n + 1) + c * 1 + c * 1 = \boldsymbol{2c * n + 4c}$

Hence, in both situations, the function has a linear growth-rate **O(n)**.

3.5. Search a sorted array of length n for a particular item

Since the given array has been sorted, we will use Binary Search algorithm to search for a particular item. The whole idea is to make use of the sort characteristic, divide the array into two halves and compare the middle value to the target. If it is greater than the target, continue with the right half. Otherwise, go with the left one. Until we found the middle which is equal to the target value [12]. The pseudocode of this algorithm is:

```
1. Set min to 0
2. Set max to n – 1
3. If max < min, return -1 (which reveals that the whole array is checked and it does not have the target
   value)
4. Set middle is the average of min and max, round up the result if it is not an integer
5. Compare the array at the middle position to the target value
       If middle = target, the target is found, return middle
       If middle > target, move to the right array by setting min to mid + 1
       If middle < target, move to the left array by setting max to mid – 1
6. Repeat the process from Step 3
```

As can be seen, the Binary Search operates by using recursion and after one recursion, the size of the array will reduce by half. Hence, after k recursion when the target has been found, the length of the array is 1.

$$\frac{n}{2^k} = 1 \Longrightarrow k = \log_2 n \ \ [13]$$

Where k is the number of recursion or the number of times that the array is divided into half.

Therefore, when searching for a particular item in a sorted array, the time complexity will be **O(logn)**.

3.6. Prints out the unique elements of a list. For example, if the list is [10, 6, 7, 4, 6, 8, 9, 7, 11, 10], the output will be the values [11, 10, 6, 4, 7, 8, 9].

This problem can be solved by utilizing the hash data structure to track each unique element [14].

Table 10. "Print unique" program theoretical time analysis

PSEUDOCODE	OPERATION	COST	OCCURENCE
For each item in the array	Assign, Comparison	$2c$	$n + 1$
If not presented in the hash table	Comparison	c	n
Print item		c	$[1, n]$
Save to hash table	Assign	c	$[1, n]$

Worst case: $T(n) = 2c * (n + 1) + c * n + c * n + c * n = \mathbf{5c * n + 2c}$

Best case: $T(n) = 2c * (n + 1) + c * n + c * 1 + c * 1 = \mathbf{3c * n + 4c}$

Hence, in both situations, the function has a linear growth-rate **O(n)**.

4. Question 4

The rate of growth of different equation can be compared by evaluating the limit of the first function $t(n)$ over the second one $g(n)$ as shown:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \; when\; order\; of\; growth\; t(n)\; is\; slower \\ c > 0 \; when\; two\; orders\; is\; the\; same \\ \infty \; when\; order\; of\; growth\; t(n)\; is\; faster \end{cases} \quad [15]$$

4.1. 100n^2 and 0.01n^3

$$\lim_{n\to\infty} \frac{100n^2}{0.01n^3} = \lim_{n\to\infty} \frac{100n^2}{0.01n^3} = 10000 * \lim_{n\to\infty} \frac{1}{n} = 0$$

Hence, $100n^2$ has slower order of growth compared to $0.01n^3$

4.2. log^2(n) and ln(n)

$$\lim_{n\to\infty} \frac{(\log n)^2}{\ln(n)} = \lim_{n\to\infty} \frac{(\frac{\ln(n)}{\ln(10)})^2}{\ln(n)} = \lim_{n\to\infty} \frac{\ln(n)}{(\ln(10))^2} = \infty$$

Hence, $(\log n)^2$ has faster order of growth compared to $\ln(n)$

4.3. (n – 1)! and n!

$$\lim_{n\to\infty} \frac{(n-1)!}{n!} = \lim_{n\to\infty} \frac{(n-1)!}{n*(n-1)!} = \lim_{n\to\infty} \frac{1}{n} = 0$$

Hence, (n – 1)! has slower order of growth compared to n!

5. Question 5

From the previous question - question 4, it can be inferred that once n approaches some significantly large value, the function coefficients as well as the lower orders are negligible. Hence, with large input size, each equation will approximately become:

- $(n-2)!$ will still the same
- $5\log(n+100)^{10} = 50 * \log(n+100) \approx \log(n)$
- $2^{2n} = 4^n$
- $0.0001n^4 + 3000n^3 + 1 \approx n^4$
- $(\ln(n))^2 = (\frac{\log(n)}{\log(e)})^2 \approx (\log(n))^2$ (Because $(\log(e))^2$ is a constant)
- $n^{1/3}$ will still the same
- 3^n will still the same

Seven equations can be classified into four complexity types:

- Factorial: $(n-2)!$
- Logarithmic: $\log(n) < (\log(n))^2$
- Exponential: $3^n < 4^n$
- Polynomial: $n^{1/3} < n^4$

As the growth rate in ascending order is Logarithmic < Polynomial < Exponential < Factorial [16], we have:

$$\log(n) < (\log(n))^2 < n^{1/3} < n^4 < 3^n < 4^n < (n-2)!$$

Hence, $5\log(n+100)^{10} < (\ln(n))^2 < n^{1/3} < 0.0001n^4 + 3000n^3 + 1 < 3^n < 2^{2n} < (n-2)!$

6. Reference

[1] "In Practice, Linear Time Algorithm for Finding Fibonacci Numbers is Quadratic", Catonmat.net, 2020. [Online]. Available: https://catonmat.net/linear-time-fibonacci?fbclid=IwAR1pdgyiZpXrRcQr4ivu_uTTK8sqhAL7GR3ShBOvLrUKg7hOEODXsjbOeeQ.

[2] A. Golubin, "Python internals: Arbitrary-precision integer implementation", Artem Golubin, 2020. [Online]. Available: https://rushter.com/blog/python-integer-implementation/.

[3] "Arbitrary-Precision Arithmetic - Competitive Programming Algorithms", Cp-algorithms.com, 2020. [Online]. Available: https://cp-algorithms.com/algebra/big-integer.html.

[4] "Art of Problem Solving", Artofproblemsolving.com, 2020. [Online]. Available: https://artofproblemsolving.com/wiki/index.php/Binet%27s_Formula.

[5] A. Golubin, "Python internals: Arbitrary-precision integer implementation", Artem Golubin, 2020. [Online]. Available: https://rushter.com/blog/python-integer-implementation/.

[6] "15. Floating Point Arithmetic: Issues and Limitations — Python 3.9.0 documentation", Docs.python.org, 2020. [Online]. Available: https://docs.python.org/3/tutorial/floatingpoint.html.

[7] "The Fibonacci sequence number of “1 000 000”?", IT Array, 2020. [Online]. Available: https://www.itarray.net/fibonacci-sequence-number-of-1-000-000/.

[8] "Primality test", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Primality_test.

[9] "AKS Primality Test - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/aks-primality-test/. [Accessed: 22- Nov- 2020].

[10] "Find all divisors of a natural number | Set 1 - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/find-divisors-natural-number-set-1/.

[11] "How many primes are there?", Primes.utm.edu, 2020. [Online]. Available: https://primes.utm.edu/howmany.html.

[12] "Binary Search - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/binary-search/.

[13] "Complexity Analysis of Binary Search - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/.

[14] "Print All Distinct Elements of a given integer array - GeeksforGeeks", GeeksforGeeks, 2020. [Online]. Available: https://www.geeksforgeeks.org/print-distinct-elements-given-integer-array/.

[15] "Comparing the Growth Rate of Two Functions Using Limits", Nagwa, 2020. [Online]. Available: https://www.nagwa.com/en/videos/930165824608/.

[16] "PennCalc | Main / OrdersOfGrowth", Calculus.seas.upenn.edu, 2020. [Online]. Available: http://calculus.seas.upenn.edu/?n=Main.OrdersOfGrowth.