

1. **Question 1:** What is the big-Oh (O) time complexity for the following algorithm (shown in pseudocode) in terms of input size n? Show all necessary steps:

(a) **Algorithm MyAlgorithm (A,B):**

```

Input: Arrays A and B each storing n >= 1 integers.
Output: What is the output? (Refer to part b below)
Start:
count = 0
for i = 0 to n-1 do {
  sum = 0
  for j = 0 to n-1 do {
    sum = sum + A[0]
    for k = 1 to j do
      sum = sum + A[k]
    }
    if B[i] == sum then count = count + 1
  }
}
return count
    
```

(b) Document a hand-run on MyAlgorithm for input arrays A = [1 2 5 9] and B = [2 29 40 57] and show the final output.

**Solution:**

(a) The big-Oh (O) time complexity of the algorithm MyAlgorithm (A,B):

count = 0	<b>1</b>
for i = 0 to n-1 do {	
sum = 0	<b>n</b>
for j = 0 to n-1 do {	
sum = sum + A[0]	<b>n<sup>2</sup></b>
for k = 1 to j do	
sum = sum + A[k]	
}	
if B[i] == sum then count =	<b>n<sup>3</sup></b>
count + 1	
}	
return count	
<b>The big-Oh time complexity is O(n<sup>3</sup>).</b>	

(b) Hand-run the function MyAlgorithm MyAlgorithm for input arrays A = [1 2 5 9] and B = [2 29 40 57]

Step 1	count= 0	Step 45	j= 2	Step 87	j= 3
Step 2	i= 0	Step 46	A[0]= 34	Step 88	A[0]= 71
Step 3	j= 0	Step 47	sum= 34	Step 89	sum= 71
Step 4	A[0]= 1	Step 48	k= 1	Step 90	k= 1
Step 5	sum= 1	Step 49	A[k]= 1	Step 91	A[k]= 1
Step 6	j= 1	Step 50	sum= 1	Step 92	sum= 1
Step 7	A[0]= 2	Step 51	k= 2	Step 93	k= 2
Step 8	sum= 2	Step 52	A[k]= 2	Step 94	A[k]= 2
Step 9	k= 1	Step 53	sum= 2	Step 95	sum= 2
Step 10	A[k]= 1	Step 54	j= 3	Step 96	k= 3

COMP352 – Winter 2018 – Assignment 2

Step 11		sum= 1	Step 55	A[0]= 42	Step 97		A[k]= 3
Step 12	j= 2		Step 56	sum= 42	Step 98		sum= 3
Step 13	A[0]= 5		Step 57		Step 99	B[i] = 40and the sum now =	
Step 14	sum= 5		Step 58		Step 100	Is B[i] == sum: false	
Step 15		k= 1	Step 59		Step 101	i= 3	
Step 16		A[k]= 1	Step 60		Step 102	j= 0	
Step 17		sum= 1	Step 61		Step 103	A[0]= 88	
Step 18		k= 2	Step 62		Step 104	sum= 88	
Step 19		A[k]= 2	Step 63		Step 105	j= 1	
Step 20		sum= 2	Step 64		Step 106	A[0]= 89	
Step 21	j= 3		Step 65		Step 107	sum= 89	
Step 22	A[0]= 13		Step 66	B[i] = 29and the sum now =	Step 108		k= 1
Step 23	sum= 13		Step 67	Is B[i] == sum: false	Step 109		A[k]= 1
Step 24		k= 1	Step 68	i= 2	Step 110		sum= 1
Step 25		A[k]= 1	Step 69	j= 0	Step 111	j= 2	
Step 26		sum= 1	Step 70	A[0]= 59	Step 112	A[0]= 92	
Step 27		k= 2	Step 71	sum= 59	Step 113	sum= 92	
Step 28		A[k]= 2	Step 72	j= 1	Step 114		k= 1
Step 29		sum= 2	Step 73	A[0]= 60	Step 115		A[k]= 1
Step 30		k= 3	Step 74	sum= 60	Step 116		sum= 1
Step 31		A[k]= 3	Step 75		Step 117		k= 2
Step 32		sum= 3	Step 76		Step 118		A[k]= 2
Step 33	B[i] = 2and the sum now =		Step 77		Step 119		sum= 2
Step 34	Is B[i] == sum: false		Step 78	j= 2	Step 120	j= 3	
Step 35	i= 1		Step 79	A[0]= 63	Step 121	A[0]= 100	
Step 36	j= 0		Step 80	sum= 63	Step 122	sum= 100	
Step 37	A[0]= 30		Step 81		Step 123		k= 1
Step 38	sum= 30		Step 82		Step 124		A[k]= 1
Step 39	j= 1		Step 83		Step 125		sum= 1
Step 40	A[0]= 31		Step 84		Step 126		k= 2
Step 41	sum= 31		Step 85		Step 127		A[k]= 2
Step 42		k= 1	Step 86		Step 128		sum= 2
Step 43		A[k]= 1			Step 129		k= 3
Step 44		sum= 1			Step 130		A[k]= 3
					Step 131		sum= 3
					Step 132	B[i] = 57and the sum now =	
					Step 133	Is B[i] == sum: false	
					Step 135	final count variable is 0	
					Step 136	final sum variable is 116	

2. **Question 2:** Consider the following code fragments (a), (b) and (c) where n is the variable specifying data size and C is a constant. What is the big-Oh time complexity in terms of n in each case? Show all necessary steps.

(a)

COMP352 – Winter 2018 – Assignment 2

```
for (int i = 0; i < n; i = i + C)
for (int j = 0; j < 10; j++)
Sum[i] += j * Sum[i];
```

(b)

```
for (int i = 1; i < n; i = i * C)
for (int j = 0; j < i; j++)
Sum[i] += j * Sum[i];
```

(c)

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

**Solution:**

(a)

for (int i = 0; i < n; i = i + C);	$n/c$
for (int j = 0; j < 10; j++);	$10*n/c$
Sum[i] += j * Sum[i];	$10*n/c + n$
The big-Oh time complexity is $O(n)$ .	

(b)

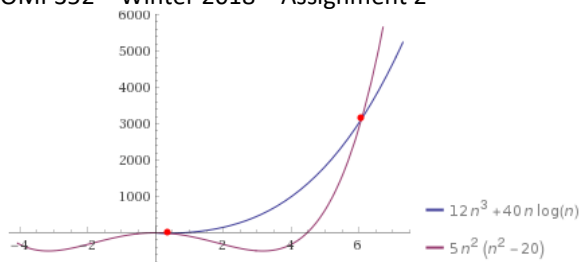
for (int i = 1; i < n; i = i * C)	$\log(n)$
for (int j = 0; j < i; j++)	$n\log(n)$
Sum[i] += j * Sum[i];	$n\log n + n$
The big-Oh time complexity is $O(n\log n)$ .	

(c)

for (int i = 1; i < n; i = i * 2)	$\log(n)$
for (int j = 0; j < n; j = j + 2)	$(n/2)\log(n)$
Sum[i] += j * Sum[i];	$(n/2)\log(n) + n$
The big-Oh time complexity is $O(n\log n)$ .	

3. **Question 3:** The number of operations executed by algorithms A and B are  $12n^3 + 40n \log n$  and  $5n^4 - 100n^2$  respectively. Determine an  $n_0$  such that B is greater than A for  $n \geq n_0$ .

**Solution:**



The graphs above are showing the behavior of these algorithms beginning with A higher (slower) than B, and finally cross. After the crossing point, B is always higher than A. Therefore we need to find the point where they cross.

That is the value where  $12n^3 + 40n \log n = 5n^4 - 100n^2$ .

Applying algebra we get:

$$12n^3 + 40n \log n = 5n^4 - 100n^2$$

$$\Rightarrow + 40 \log n =$$

$$\Rightarrow 40 \log n = 5n^3 - 12n^3 - 100n^2$$

$\Rightarrow$  Solving for n we get  $n \approx 6.08006101894724\dots$ . Thus  $n_0 = 7$ , since for all  $n \geq 7$ , A will be faster than B (at 6.0800... they're equal).

4. **Question 4:** Answer the following questions:

(a) Show that if  $d(n)$  is  $O(f(n))$  and  $e(n)$  is  $O(g(n))$ , then  $d(n) + e(n)$  is  $O(f(n) + g(n))$ .

(b) Show that if  $d(n)$  is  $O(f(n))$  and  $e(n)$  is  $O(g(n))$ , then  $d(n) - e(n)$  is not necessarily  $O(f(n) - g(n))$ .

(c) Show that  $2^{n+1} + n^2$  is  $O(2^n)$

(d) Show that  $f(n) = \sum_{i=1}^n i^2$  is  $O(n^3)$

**Solution:**

(a) True.

$d(n)$  is  $O(f(n))$ , and  $e(n)$  is  $O(g(n))$

$\rightarrow d(n) \leq K f(n)$  for  $n \geq N$ , and  $e(n) \leq L g(n)$  for  $n \geq M$

$\rightarrow d(n) + e(n) \leq K f(n) + L g(n)$

$\leq (K + L)(f(n) + g(n))$ , for  $n \geq \max(N, M)$

(b) False.

We have  $d(n) = 2n$  and  $e(n) = n$ . So,  $d(n) - e(n) = n$ .

$\rightarrow d(n) - e(n) = n$ .

Because  $d(n) = O(n)$  and  $e(n) = O(n)$ , so we have:

$f(n) = g(n) = n$ ,

so

$O(f(n) - g(n)) = O(n - n) = O(1)$ ,

and obviously  $n \neq O(1)$ . Contradict to the question.

(c) True

$2^{n+1} = O(2^n)$ , but  $2^{2n} \neq O(2^n)$ . To show that  $2^{n+1} = O(2^n)$ , we must find constants  $c; n_0 > 0$  such that  $0 \leq 2^{n+1} \leq c 2^n$  for all  $n \geq n_0$ . Since  $2^{n+1} = 2 \cdot 2^n$  for all  $n$ , we can satisfy the definition with  $c = 2$  and  $n_0 = 1$ .

To show that  $2^{2^n} \neq O(2^n)$ , assume there exist constants  $c; n_0 > 0$  such that  $0 \leq 2^{2^n} \leq c \cdot 2^n$  for all  $n \geq n_0$ . Then  $2^{2^n} = 2^n \times 2^n \leq c \cdot 2^n$ . But no constant is greater than  $2^n$  for all  $n$ , and so the assumption leads to a contradiction.

(d) Choose  $k=1$

Assuming  $n > 1$ , then

$$\frac{f(n)}{g(n)} = \frac{\sum_{i=1}^n i^2}{n^3} \leq \frac{\sum_{i=1}^n n^2}{n^3} = \frac{n^3}{n^3} = 1$$

Choose  $c=1$ . Note that  $i \leq n$  because  $n$  is the upper limit. Hence,  $\sum_{i=1}^n i^2$  is  $O(n^3)$  because  $\sum_{i=1}^n i^2 \leq n^3$  whenever  $n > 1$ .

5. **Programming Questions:** Referring to the slides from text book, Chapter 5, also added to moodle, there are two versions of Fibonacci number calculators: BinaryFib( $n$ ) and LinearFibonacci( $n$ ). The first algorithm has exponential time complexity, while the second one is linear.

(a) In this programming assignment, you will implement in Java both the versions of Fibonacci calculators and experimentally compare their runtime performances. For that, with each implemented version you will calculate Fibonacci (5), Fibonacci (10), etc. in increments of 5 up to Fibonacci (100) (or higher value if required for your timing measurement) and measure the corresponding run times. You need to use Java's built-in time function for this purpose. You should redirect the output of each program to an out.txt file. You should write about your observations on timing measurements in a separate text or pdf file. You are required to submit the two fully commented Java source files, the compiled executables, and the text/pdf files. The set of strings over  $\{0; 1\}$ , such that there are two 0's separated by a number of positions that is a multiple of 4. Note that 0 is an allowable multiple of 4.

(b) Briefly explain why the first algorithm is of exponential complexity and the second one is linear (more specifically, how the second algorithm resolves some specific bottleneck(s) of the first algorithm). You can write your answer in a separate file and submit it together with the other submissions.

(c) Do any of the previous two algorithms use tail recursion? Why or why not? Explain your answer. If your answer is "No" then:

- i. design the pseudo code for a tail recursive version of Fibonacci calculator;
- ii. implement the corresponding Java program and repeat the same experiments as in part (a) above. You will need to submit both the pseudo code and the Java program, together with your experimental results.

### Solution:

(a) See the attachment.

(b) The first algorithm has the exponential complexity of  $O(3^n)$  because it calls itself not only recursively but it calls itself recursively within itself. A call to a loop inside the method would raise the power  $x$  of  $n^x$ . However, a call to itself recursively raises the power of  $x$  in  $x^n$ . The second algorithm calls itself  $O(n)$  times by reducing the amount of times it calls back to itself, once instead of three times.

(c) The first algorithm uses tail recursion, because the last thing the function does before it returns is recursively call itself. The second algorithm does not use tail-recursion. A tail recursive method could not be implemented because it is printing out the results and manipulating data after each recursive call. This could not be changed.

i. design the pseudo code for a tail recursive version of Fibonacci calculator

```
# A tail recursive function to calculate nth fibonacci number
function tail fibonacci (n, a = 0, b = 1):
    if (n == 0):
        return a
    if (n == 1):
        return b
    return fibonacci (n-1, b, a+b);
```

ii. implement the corresponding Java program and repeat the same experiments as in part (a) above. You will need to submit both the pseudo code and the Java program, together with your experimental results. See the attachment of .java file.

---

<https://www.geeksforgeeks.org/factorial-large-number/>

<https://github.com/whitef0x0/lab5>

<http://learn.hfm.io/expressions.html>