# 1.

Solve the following problems in the textbook using the limit rule as appropriate. The limit rule is defined on Canvas Module 1 in the Big-Oh Cheat Sheet. Your work must be shown for credit.

a. R-1.12

b. R-1.13

c. R-1.14

d. R-1.15

## a. R-1.12

Perform a similar analysis for method Loop2 shown in Algorithm 1.21.

```
Algorithm Loop2(n):      Primitive Operations
    p ← 1                        1
    for i ← 1 to 2n do           2n
        p ← p · i                constant
```

Total primitive operations : 1 + 2n

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} (\frac{1}{n} + 2) = 2$$

$$0 <= 2 < \infty$$

`Conclusion:` Method Loop2 runs in O(n) time.

## b. R-1.13

Perform a similar analysis for method Loop3 shown in Algorithm 1.21.

```
Algorithm Loop3(n):      Primitive Operations
    p ← 1                        1
    for i ← 1 to n^2 do          n^2
        p ← p · i                constant
```

Total primitive operations : 1 + $n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} (\frac{1}{n^2} + 1) = 1$$

$$0 <= 1 < \infty$$

`Conclusion:` Method Loop3 runs in O($n^2$) time.

## c. R-1.14

Perform a similar analysis for method Loop4 shown in Algorithm 1.21.

```
Algorithm Loop4(n):      Primitive Operations
    s ← 0                          1
    for i ← 1 to 2n do             2n
        for j ← 1 to i do          i: 1 -> 2n, average: (2n + 1) / 2
            s ← s + i              constant
```

s ← 0 run time: 1

two nested loops run time: 1 + 2 + 3 + ... + 2n = 2n * (2n + 1) / 2 = $2n^2$ + n

Total Primitive Operations = 1 + $2n^2$ + n

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty}(\frac{1}{n^2} + 2 + \frac{1}{n}) = 2$

$0 <= 2 < \infty$

`Conclusion:` Method Loop4 runs in O($n^2$) time.

## d. R-1.15

Perform a similar analysis for method Loop5 shown in Algorithm 1.21.

```
Algorithm Loop5(n):          Primitive Operations
    s ← 0                            1
    for i ← 1 to n^2 do              n^2
        for j ← 1 to i do            i: 1 -> n^2, average: (1+n^2)/2
            s ← s + i                constant
```

s ← 0 run time: 1

two nested loops run time: $n^2$ * (1+$n^2$)/2 = 0.5$n^4$ + 0.5$n^2$

Total Primitive Operations = 1 + 0.5$n^4$ + 0.5$n^2$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty}(\frac{1}{n^4} + 0.5 + 0.5\frac{1}{n^2}) = 0.5$

$0 <= 0.5 < \infty$

`Conclusion:` Method Loop5 runs in O($n^4$) time.

# 2.

Solve the following problems in the textbook using the limit rule as appropriate. The limit rule is defined on Canvas Module 1 in the Big-Oh Cheat Sheet. Your work must be shown for credit.

a. R-1.20

b. R-1.21

c. R-1.22

d. R-1.23

e. R.1-24

## a. R-1.20

Show that $(n + 1)^5$ is $O(n^5)$.

`Proof:`

$(n + 1)^5 = a \cdot n^5 + b \cdot n^4 + c \cdot n^3 + d \cdot n^2 + e \cdot n^1 + 1$ (a,b,c,d,e are all contants)

For this polynomial, $a \cdot n^5 + b \cdot n^4 + c \cdot n^3 + d \cdot n^2 + e \cdot n^1 + 1 <= (a + b + c + d + e + 1) \cdot n^5 = O((a + b + c + d + e + 1)n^5)$

So, $(n + 1)^5$ is $O(n^5)$, C = (a + b + c + d + e + 1), $n >= n_0 = 1$

## b. R-1.21

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

`Proof:`

$2^{n+1} = 2 * 2^n = O(2 * 2^n) = 2 * O(2^n)$

C = 2

$0 <= 2 < \infty$

So, $2^{n+1}$ is $O(2^n)$.

## c. R-1.22

Show that n is o(n log n).

`Proof:`

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{nlogn} = \lim_{n\to\infty} \frac{1}{logn} = 0$$

So, n is o(n log n).

## d. R-1.23

Show that $n^2$ is $\omega(n)$.

`Proof:`

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} n = \infty$$

So, $n^2$ is $\omega(n)$.

## e. R.1-24

Show that $n^3 \log n$ is $\Omega(n^3)$.

`Proof:`

$n^3 \log n - n^3 = n^3 * (\log n - \log 10) = n^3 * log\frac{n}{10} >= 0$, when n >= 10.

Thus $n^3 \log n >= n^3$, when $n >= 10 = n_0$, c = 1.

Finally, I can get $n^3 \log n$ is $\Omega(n^3)$.

# 3.

## R-1.25

Show that $\lceil f(n) \rceil$ is $O(f(n))$ if $f(n)$ is a positive nondecreasing function that is always greater than 1.

`Proof:`

To show that $\lceil f(n) \rceil$ is $O(f(n))$ under the given conditions, we need to find constants $C$ and $n_0$ such that for all $n \geq n_0$, $\lceil f(n) \rceil \leq C \cdot f(n)$.

Given that $f(n)$ is a positive non-decreasing function that is always greater than 1, we can use the following reasoning:

Since $f(n)$ is always greater than 1, we know that $\lceil f(n) \rceil$ will be at least 2 for all $n$.

Because $f(n)$ is non-decreasing, it will only increase as $n$ increases.

With these observations in mind, we can choose $C = 2$ and $n_0 = 1$. For all $n \geq n_0$, we have:

$\lceil f(n) \rceil \leq 2 \cdot f(n)$

This holds because $\lceil f(n) \rceil$ is always at least 2 (greater than or equal to 2), and $f(n)$ is a positive non-decreasing function. Therefore, for all $n \geq n_0$, $\lceil f(n) \rceil$ is bounded above by $2 \cdot f(n)$.

Thus, we have shown that $\lceil f(n) \rceil$ is $O(f(n))$ with $C = 2$ and $n_0 = 1$, as required.

# 4.

Solve the following problems in the textbook using the limit rule as appropriate. The limit rule is defined on Canvas Module 1 in the Big-Oh Cheat Sheet. Your work must be shown for credit.

## a. Is $3^x = O(2^x)$?

$\lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty} 1.5^x = \infty$

We can't find contant C to do $3^x <= C * 2^x$.

Therefore, it is false that $3^x = O(2^x)$.

## b. Is $\log_3 x = O(\log_2 x)$?

$\lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty} \frac{\log_3 x}{\log_2 x} = \lim_{x\to\infty} \frac{\log_2 x}{\log_2 x * \log_2 3} = \frac{\log_2 2}{\log_2 3} = log_3 2$

$log_3 2$ is a contant, so $C = log_3 2$,

Thus, we have $\log_3(x) \leq C \cdot \log_2(x)$,for all $x \geq x_0$, where $x_0$ is a constant.

Therefore, it is true that $\log_3 x = O(\log_2 x)$.

# 5.

The following problems give the closed form of the recurrence equations. For no points, convince yourself that the closed form is correct (do not submit this). Now, for points, complete exercises:

## a. C-1.6

Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if n=0} \\ T(n-1) + 2^n & \text{otherwise,} \end{cases}$$

Show, by induction, that $T(n) = 2^{n+1} - 1$.

`Proof:`

**Base Case (n = 0, 1, 2, 3):**

T(0) = 1 = $2^{0+1} - 1$

T(1) = T(0) + $2^1$ = 1 + 2 = 3 = $2^{1+1} - 1$

T(2) = T(1) + $2^2$ = 3 + 4 = 7 = $2^{2+1} - 1$

So, the formula holds for the base case.

**Inductive Hypothesis:**

Now, assume that for some integer k >= 0, we have:

$T(k) = 2^{k+1} - 1$, when k >= 0

**Inductive Step:**

We need to prove that the formula holds for k + 1, assuming it holds for k. So, we want to show:

$T(k+1) = 2^{k+1+1} - 1 = 2^{k+2} - 1$

According recurrence relation,

$T(k+1) = T(k) + 2^{k+1} = 2^{k+1} - 1 + 2^{k+1}$

Now, simplify:

$T(k+1) = 2^{k+2} - 1$

This is the formula we wanted to prove for k + 1.

Finally, therefore, it is True that $T(n) = 2^{n+1} - 1$.

## b. C-1.7

Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if n=0} \\ 2T(n-1) & \text{otherwise,} \end{cases}$$

Show, by induction, that $T(n) = 2^n$.

`Proof:`

**Base Case (n = 0, 1, 2, 3):**

T(0) = 1 = $2^0$

T(1) = 2 * T(0) = 2 * $2^0$ = $2^1$

T(2) = 2 * T(1) = 2 * $2^1$ = $2^2$

T(3) = 2 * T(2) = 2 * $2^2$ = $2^3$

So, the formula holds for the base case.

**Inductive Hypothesis:**

Now, assume that for some integer k >= 0, we have:

$T(k) = 2^k$, when k $>= 0$.

**Inductive Step:**

We need to prove that the formula holds for k + 1, assuming it holds for k. So, we want to show:

$T(k+1) = 2^{k+1}$

According recurrence relation,

$T(k+1) = 2 * T(k) = 2 * (2^k)$

Now, simplify:

$T(k+1) = 2^{k+1}$

This is the formula we wanted to prove for k + 1.

Finally, therefore, it is True that $T(n) = 2^n$.

# 6.

An array A of n elements contains unique integers in the range of [0, n]. Thus, one integer in the range from [0, n] is missing. E.g. If n = 5, then A might contains “4, 1, 0, 5, 2” so 3 is missing. Hint: there is a little trick to this problem. How could you easily determine that number 3 is missing? Only one loop is required for an efficient solution. Can you figure it out?

```
a. Give a O(n)-time algorithm (pseudocode) for finding that number. You are allowed to use onl
y O(1) additional space besides the array A itself.
b. Explain the main concept of your algorithm and why it solves the problem.
c. Give an example of running your algorithm.
d. Explain why its runtime is O(n).
e. Explain why your extra space usage is only O(1).
```

## a

Algorithm findMissingNumber(A, n):

```
Input: An array A of n elements contains unique integers in the range of [0, n].
Output: Missing Number.

actual_sum <- 0
for i from 0 to n - 1:
    actual_sum <- actual_sum + A[i]
    if i = n - 1 then
        return (n * (n + 1)) / 2 - actual_sum
```

## b

Explanation:

We can solve this problem efficiently by taking advantage of the mathematical properties of summation. Since we know that the array contains unique integers from 0 to n, we can calculate the expected sum of those integers `(n * (n + 1)) / 2` and subtract the actual sum of the elements ( `actual_sum` ) in the array to find the missing number.

## c

For example: A = [4, 1, 0, 5, 2]

Let's apply the algorithm to this array:

actual_sum = 0

Loop through the array and calculate `actual_sum:`

- Iteration 1: actual_sum = 0 + 4 = 4
- Iteration 2: actual_sum = 4 + 1 = 5
- Iteration 3: actual_sum = 5 + 0 = 5
- Iteration 4: actual_sum = 5 + 5 = 10
- Iteration 5: actual_sum = 10 + 2 = 12

`return` = 15 - 12 = 3

So, in this example, the missing number in the array [4, 1, 0, 5, 2] is indeed 3, and the algorithm correctly identifies it as the missing element.

## d

Algorithm findMissingNumber(A, n):

```
Input: An array A of n elements contains unique integers in the range of [0, n].
Output: Missing Number.
                                          Primitive operations
actual_sum <- 0---------------------------------------- 1
for i from 0 to n - 1:-------------------------------- O(n)
    actual_sum <- actual_sum + A[i]
    if i = n - 1 then
        return (n * (n + 1)) / 2 - actual_sum
```

Total Primitive operations = 1 + O(n) = O(n) So, Runtime = O(n)

## e

The algorithm for finding the missing number uses only O(1) extra space because it maintains a constant amount of additional memory that does not depend on the size of the input array. Here's why the extra space usage is O(1):

Constant Variables: The algorithm uses a few constant variables for calculations and temporary storage. These variables include n, i, actual_sum. Regardless of the size of the input array, these variables occupy a fixed amount of memory. Therefore, they contribute to a constant amount of extra space, which is O(1).

Therefore, the algorithm's extra space usage remains constant and does not grow with the input size. It uses only a few fixed-size variables to perform calculations, making its extra space complexity O(1).

# 7. C-8.3

Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an O(n)-time method for computing a sequence representing the set A ∪ B (with no duplicates).

## - a. Explain the major concept of your algorithm.

**Algorithm** Union(A, B, S):

```
Input: two n-element sorted sequences A and B, and an empty array, S, of size at least 2n
Output: Union of sequences A and B with no duplicates, stored in array S

i ← 0
j ← 0
k ← 0

while i < length(A) and j < length(B) do
    if A[i] = B[j]:
        S[k] ← A[i]
        i ← i + 1
        j ← j + 1
        k ← k + 1
    elif A[i] < B[j]:
        S[k] ← A[i]
        i ← i + 1
        k ← k + 1
    else:
        S[k] ← B[j]
        j ← j + 1
        k ← k + 1

while i < length(A) and A[i] = S[k - 1]:
    i ← i + 1
while j < length(B) and B[j] = S[k - 1]:
    j ← j + 1

while i < length(A):
    S[k] ← A[i]
    i ← i + 1
    k ← k + 1

while j < length(B):
    S[k] ← B[j]
    j ← j + 1
    k ← k + 1

return S
```

**Explanation:**

**Main Loop:** The core of the algorithm is a while loop that runs as long as there are elements left in both sequences A and B. Inside this loop:

1. It compares the elements at the positions pointed to by i and j.
2. If the elements at A[i] and B[j] are equal, it means there's a duplicate element in both A and B. In this case, the algorithm adds one of these duplicates to the result array S, advances both pointers i and j, and also advances k to the next position in S.
3. If A[i] is less than B[j], it means the element in A is smaller, so it adds A[i] to S, advances i, and advances k in S.
4. If B[j] is less than A[i], it means the element in B is smaller, so it adds B[j] to S, advances j, and advances k in S.

After merging the sequences in the main loop, the algorithm adds two additional while loops to handle duplicates properly. These loops skip over any remaining duplicate elements in sequences A and B by comparing them with the last element added to S (S[k-1]). Finally, after handling duplicates, the algorithm ensures that any remaining elements in A and B are added to S to complete the union operation. The algorithm returns the array S, which contains the union of sequences A and B, with no duplicate entries.

## - b. Give pseudocode.

You can see in the answer of a.

## - c. Give an example of running your algorithm.

Sequence A: [1, 3, 5, 5, 7, 9] Sequence B: [2, 4, 4, 6, 8]

i = 0 (pointer for sequence A) j = 0 (pointer for sequence B) k = 0 (pointer for array S)

Start the main loop:

A[i] = 1 and B[j] = 2 (A[i] < B[j]):

- Add 1 to S: S[0] = 1
- Increment i: i = 1
- Increment k: k = 1

A[i] = 3 and B[j] = 2 (A[i] > B[j]):

- Add 2 to S: S[1] = 2
- Increment j: j = 1
- Increment k: k = 2

A[i] = 3 and B[j] = 4 (A[i] < B[j]):

- Add 3 to S: S[2] = 3
- Increment i: i = 2
- Increment k: k = 3

A[i] = 5 and B[j] = 4 (A[i] > B[j]):

- Add 4 to S: S[3] = 4
- Increment j: j = 2
- Increment k: k = 4

A[i] = 5 and B[j] = 4 (A[i] > B[j]):

- Add 4 to S: S[4] = 4
- Increment j: j = 3
- Increment k: k = 5

A[i] = 5 and B[j] = 6 (A[i] < B[j]):

- Add 5 to S: S[5] = 5
- Increment i: i = 3
- Increment k: k = 6

A[i] = 7 and B[j] = 6 (A[i] > B[j]):

- Add 6 to S: S[6] = 6
- Increment j: j = 4
- Increment k: k = 7

A[i] = 7 and B[j] = 8 (A[i] < B[j]):

- Add 7 to S: S[7] = 7
- Increment i: i = 4
- Increment k: k = 8

A[i] = 9 and B[j] = 8 (A[i] > B[j]):

- Add 8 to S: S[8] = 8
- Increment j: j = 5
- Increment k: k = 9

Both sequences are exhausted, so we need to add any remaining elements from A and B to S:

- Add 9 to S: S[9] = 9
- Increment i: i = 5
- Increment k: k = 10

Sequence B is also exhausted, and there are no remaining elements in B.

The resulting array S contains the union of sequences A and B with no duplicate entries:

S = [1, 2, 3, 4, 5, 6, 7, 8, 9] So, the algorithm correctly computed the union of sequences A and B while eliminating duplicate entries, and the result is stored in array S.

## - d. Give and justify its runtime.

i ← 0 O(1) j ← 0 O(1) k ← 0 O(1)

##########################################

while i < length(A) and j < length(B) do

```
    if A[i] = B[j]:
        S[k] ← A[i]
        i ← i + 1
        j ← j + 1
        k ← k + 1
    elif A[i] < B[j]:
        S[k] ← A[i]
        i ← i + 1
        k ← k + 1
    else:
        S[k] ← B[j]
        j ← j + 1
        k ← k + 1
```

while i < length(A) and A[i] = S[k - 1]:
i ← i + 1
while j < length(B) and B[j] = S[k - 1]:
j ← j + 1

while i < length(A):
S[k] ← A[i]
i ← i + 1
k ← k + 1

while j < length(B):
S[k] ← B[j]
j ← j + 1
k ← k + 1
##########################################

In the part between "##########################################", the runtime is **O(2n)**.

return S O(1)

**Total primitive operations** = O(2n) + O(1) + O(1) + O(1) + O(1) = O(n)

## - e. Justify its correctness.

Final Result: The algorithm returns the array S, which contains the union of sequences A and B with no duplicates.

Efficiency: The algorithm operates in linear time O(n), where n is the total number of elements in A or B. It efficiently processes each element exactly once and maintains the sorted order, ensuring that no elements are missed or duplicated.

In conclusion, the algorithm correctly computes the union of two sorted sequences A and B with no duplicates while maintaining efficiency and correctness throughout its execution.

# 8. C-9.12

Given an array, A, of n numbers in the range from 1 to n, describe an O(n)-time method for finding the mode, that is, the number that occurs most frequently in A.

- a. Explain the major concept of your algorithm.
- b. Give pseudocode.
- c. Give an example of running your algorithm.
- d. Give and justify its runtime.
- e. Justify its correctness.

## a, b

The major concept of this algorithm is to exploit the fact that the values in the array A are in the range from 1 to n. Therefore, each value directly corresponds to an index in a frequency count array, where we can efficiently count the occurrences of each value. By iterating through the array A once and updating the counts in the frequency array, we can determine the mode.

**Algorithm** FindMode(A):

```
Input: An array A of n numbers in the range from 1 to n, and frequencyArray is an array of siz
e n, initialized with all zeros.

Output: The mode (most frequent number) in A

maxCount ← 0
mode ← -1

for i from 0 to n-1 do
    currentNumber ← A[i]
    frequencyArray[currentNumber - 1] ← frequencyArray[currentNumber - 1] + 1
    if frequencyArray[currentNumber - 1] > maxCount then
        maxCount ← frequencyArray[currentNumber - 1]
        mode ← currentNumber

return mode
```

## c

A = [2, 3, 3, 1, 2, 4, 2]

Initialize frequencyArray and other variables. Start iterating through A.

- For A[0] = 2, increment frequencyArray[1] to 1 (value 2 - 1).
- For A[1] = 3, increment frequencyArray[2] to 1 (value 3 - 1).
- For A[2] = 3, increment frequencyArray[2] to 2.
- For A[3] = 1, increment frequencyArray[0] to 1.
- For A[4] = 2, increment frequencyArray[1] to 2.
- For A[5] = 4, increment frequencyArray[3] to 1.
- For A[6] = 2, increment frequencyArray[1] to 3.

During these steps, update maxCount and mode when needed. After processing all elements in A, mode is 2, which is the most frequent number.

## d

The runtime complexity of this algorithm is O(n), where n is the length of the input array A. Here's the justification:

- The initialization of variables and arrays takes constant time, O(1).
- The loop that iterates through the elements of A runs from index 0 to n-1, performing a constant number of operations for each element.
- Within the loop, updating the frequency array, comparing frequencies, and updating maxCount and mode all take constant time.
- Thus, the overall time complexity is O(n).

## e

The algorithm is correct because it correctly counts the frequency of each number in the input array A and correctly identifies the mode as the number with the highest frequency.

It efficiently uses a frequency array to count occurrences and updates the mode during the counting process. The algorithm processes each element exactly once, ensuring its correctness.