2MD3 Assignment 3 – Michael Giancola – 400372137

1. From smallest to largest asymptotic growth rate:

   2^{10}
   $2^{\log(n)}$
   3n + 100log(n)
   4n
   nlog(n)
   4nlog(n) + 2n
   $2 + n^2$
   $n^2 + 10n$
   n^3
   2^n

2. If d(n) is O(f(n)) it means that there are constants c and n_o such that for all n >= n_o , d(n) <= c*f(n). If ad(n) is also O(f(n)) for a > 0, this means that there are constants C and N that for all n >= N, ad(n) <= C*f(n). Using the definition of O(f(n)) we can substitute ad(n) for d(n) as shown below:

   ad(n) <= (ac)f(n)

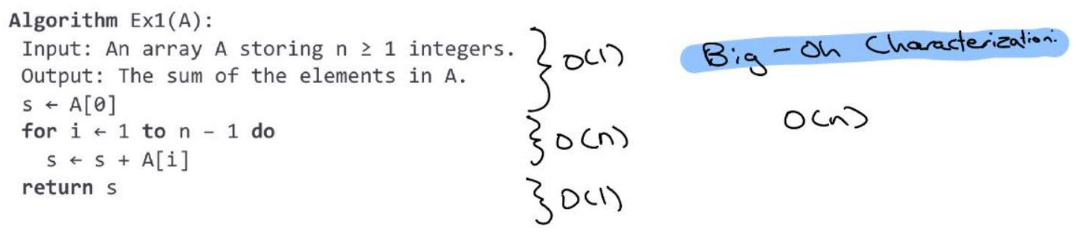
   We can now let C = ac and N = n_o because d(n) is O(f(n)) for n >= n_o. Now, for all n >= N, ad(n) <= C f(n). Therefore, with the above expression proven, ad(n) is O(f(n))

   This can be demonstrated with some sample code below:

```
bool is_adn_O_fn(int (*d)(int), int (*f)(int), double a) {
    double c = 1.0;
    int n0 = 1;
    while (d(n0) > c * f(n0)) {
        n0++;
    }
    for (int n = n0; n < INT_MAX; n++) {
        if (a * d(n) > c * f(n)) {
            return false;
        }
    }
    return true;
}
```

It takes two pointers ‘d’ and ‘f’ that represent d(n) and f(n) and also uses a constant ‘a’ as input. It then iterates through the loop to find the smallest value of n0 that satisfies the definition of Big-O and then checks if it holds for ad(n) and f(n0 with the same constant c and the new value of n0.

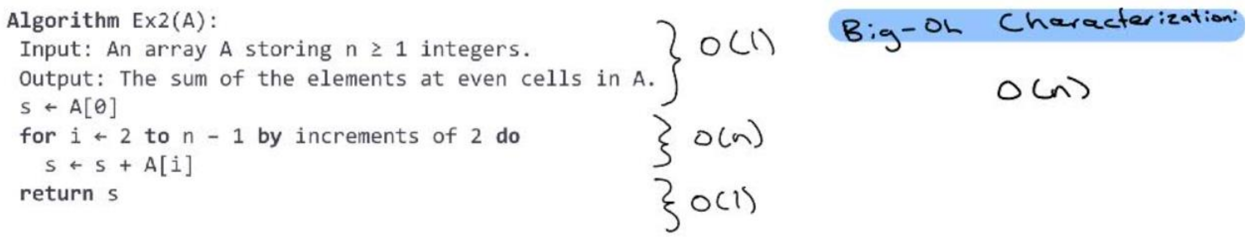
3.



```
Algorithm Ex1(A):
 Input: An array A storing n ≥ 1 integers.
 Output: The sum of the elements in A.
 s ← A[0]
 for i ← 1 to n − 1 do
   s ← s + A[i]
 return s
```

The loop starts at i=1 and goes up to i=n-1 which means it executes n-1 times. In the loop when it says "s ← s + A[i]", it takes constant time so the time spent in the loop is proportional to n-1. This shows the running time grows linearly with the size of the input so the big-Oh characterization is O(n).

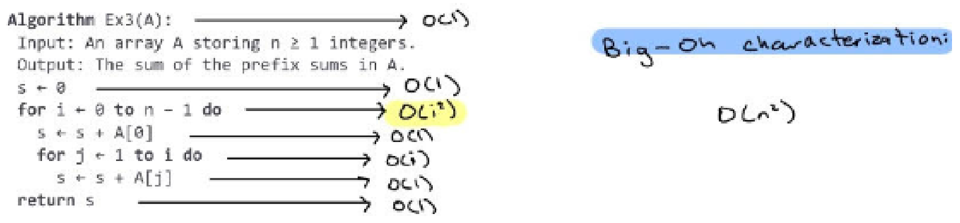
4.



```
Algorithm Ex2(A):
 Input: An array A storing n ≥ 1 integers.
 Output: The sum of the elements at even cells in A.
 s ← A[0]
 for i ← 2 to n − 1 by increments of 2 do
   s ← s + A[i]
 return s
```

The loop iterates n/2 times (from 2 to n-1 by increments of 2), and each iteration takes constant time meaning it's adding the value at an even-indexed cell to the running sum. The total running time is proportional to n/2, so the Big-Oh characterization is O(n).

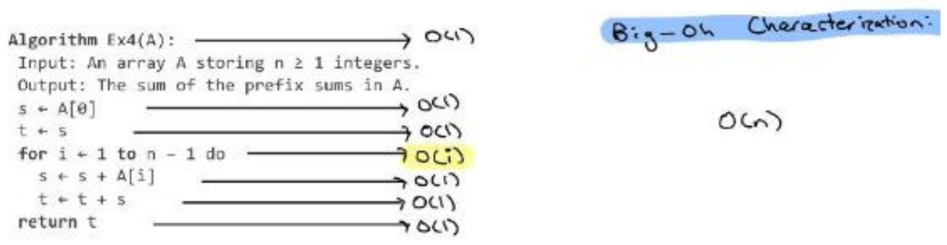
5.



```
Algorithm Ex3(A):
 Input: An array A storing n ≥ 1 integers.
 Output: The sum of the prefix sums in A.
 s ← 0
 for i ← 0 to n − 1 do
   s ← s + A[0]
   for j ← 1 to i do
     s ← s + A[j]
 return s
```

This example has two nested loops with the outer loop iterating n times and the inner loop iterating up to I times for each iteration of the outer loop. The total number of iterations of the inner loop is the sum of the first n integers which is proportional to n^2. Therefore, the Big-Oh characterization is O(n^2).

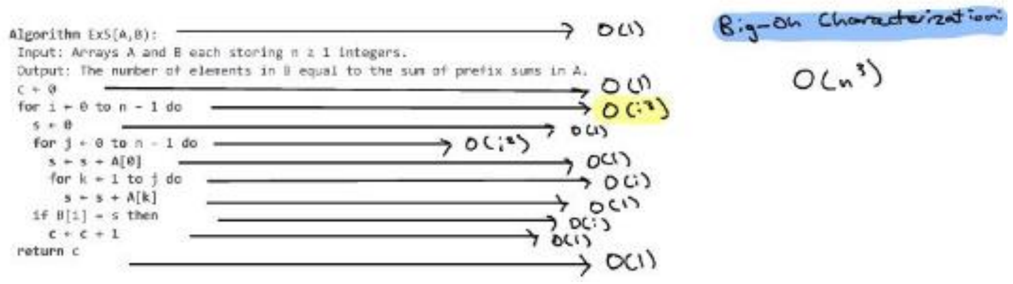
6.



```
Algorithm Ex4(A):                    → O(1)
 Input: An array A storing n ≥ 1 integers.
 Output: The sum of the prefix sums in A.
 s ← A[0]                            → O(1)
 t ← s                               → O(1)
 for i ← 1 to n − 1 do               → O(i)
   s ← s + A[i]                      → O(1)
   t ← t + s                         → O(1)
 return t                            → O(1)
```

The loop iterates through each element of array A once, doing constant time operations each time. Therefore, the total time complexity is proportional to the size of the input which is n, so the Big-Oh characterization is O(n).

7.



```
Algorithm Ex5(A,B):                  → O(1)
 Input: Arrays A and B each storing n ≥ 1 integers.
 Output: The number of elements in B equal to the sum of prefix sums in A.
 c ← 0                               → O(1)
 for i ← 0 to n − 1 do               → O(i³)
   s ← 0                             → O(1)
   for j ← 0 to n − 1 do             → O(i²)
     s ← s + A[0]                    → O(1)
     for k ← 1 to j do               → O(i)
       s ← s + A[k]                  → O(1)
   if B[i] = s then                  → O(i)
     c ← c + 1                       → O(1)
 return c                            → O(1)
```

This example has three nested loops with each of them iterating up to n times. The first loop iterates n times, the second iterates n times for each iteration of the first loop, and the third iterates up to j times for each iteration of the second loop. The total number of iterations is n*n*n = n^3. Therefore, the Big-Oh characterization is O(n^3).

8. For the first element X[0], Algorithm E will run in O(0) time because there are no previous elements to process. For the second element X[1], Algorithm E will run O(1) time because it has to process one previous element and so on. As stated, for the i-th element X[i], Algorithm E will run in O(i) time because it must process i-1 elements. Therefore, the total running time of Algorithm D can be expressed as the sum of the running times of Algorithm E for each element.

   T(n) = O(0) + O(1) + O(2)+…+O(n-1)

   This can be further simplified through the summation formula to T(n) = O(n^2), which is the worst case running time.

9. The sum of the integers in the range [0, n-1] is equal to n*(n-1)/2. To get the missing number you can sum all of the integers in Array A and subtract it from the expected sum to get the missing number. This is shown in the code below.

```
int findMissingNumber(int A[], int n) {
    int expectedSum = n*(n-1)/2;
    int actualSum = 0;
    for (int i = 0; i < n-1; i++) {
        actualSum += A[i];
    }
    return expectedSum - actualSum;
}
```

The time complexity of this algorithm is O(n) as we loop through the array once to calculate the entire sum.

10. We can use a function reorder_dequeue(D) and an initially empty queue Q to result in storing the elements (1,2,3,4,5,6,7,8) in the correct order.

Pseudocode:

```
function reorder_dequeue(D):
   while D is not empty:
   x ← D.popleft( )
   if x ← 4:
       Q.enqueue(x)
   else:
       D.append(x)

   while Q is not empty( ):
       x ← Q.dequeue( )
       D.insert(D.index(5)+1, x)
```

In the above pseudocode, the function iterates over the elements in D using ‘popleft( )’ that removes and returns the leftmost element of D. If the element is 4, it is added to Q using ‘enqueue’, which adds the element to the back of the queue. Otherwise, it is added back to the right end of D using ‘append( )’. After all of D’s elements have been processed, the function then iterates over all the elements in Q using ‘dequeue’ that removes and returns the frontmost element of Q. The function finds the index of 5 which returns the index of the specified element. The element that’s dequeued is then inserted right after 5 using ‘insert’. Therefore, D will store the elements in the order (1,2,3,5,4,6,7,8).

11.

Pseudocode:

Algorithm findLargest (stack<int>& s):
Input: A stack with 3 integers
Output: the largest value of the 3 with a 2/3 success rate

```
x ← s.top()
s.pop()

if s.top() > x then
  x ← s.top()
  s.top()
return x
```

Firstly, the x integer is set to the value at the top of the stack and then remove that value from the stack.

```
int findLargest(stack<int>&s) {
    int x = s.top();
    s.pop();
```

The single comparison happening is between x and the top of the stack. If the value at the top of the stack is larger it becomes x and is removed from the stack, else, the current x value is returned.

```
    if (s.top() > x) {
        x = s.top();
        s.pop();
    return x;
```

Using the pseudocode above, it is evident that the final value of ‘x’ will be the largest of the three distinct integers with a probability of at least 2/3. The algorithm will correctly find the largest number on the first comparison if Alice arranged the integers in ascending order (since the second and third comparisons will never update x). It will correctly find the greatest integer in each of the three comparisons if Alice arranged the integers in descending order (since each comparison will update x). As there are three alternative orders for the largest integer to appear in the stack, and the code correctly identifies it in two of those orders, if Alice had arranged the integers in any other order, the code would accurately identify the largest integer with a probability of exactly 2/3.