1.

```
// Pseudocode that generates all possible subsets of a given set T containing n elements
function powerSet (T) :
// Get the size of the set T
n = size (T)
// Initialize empty list in order to store all subsets
result = [ ]
// Initialize a queue and begin by adding the empty subset
set = Queue ( )
set.enqueue ([ ])
// Check every subset until the queue is empty
while not set.isEmpty( ) :
        // Dequeue current subset then add it to the result list
        currentSub = set.dequeue ( )
        result.append (currentSub)

// Iterate over the elements from 0 till the last element

for i from 0 to n-1 :

          // Make another subset by copying the current one

          newSub = currentSub.copy ( )
          // Add the current element to the newly created subset
          newSub.append (T [i] )
         // Enqueue the new subset
          set.enqueue (newSub)

// Return the list of all subsets

 return result
```

A)

The time complexity of my algorithm is $O(2^n)$ because the while loop runs 2^n times, once for each subset and in the worst case each element is processed once.

B)

The space complexity of my algorithm is $O(2^n)$ as the worst case in the queue remains 2^n.

2.

A)

The tightest bound of this algorithm is $O(n^6)$ because the outer loop, i, runs from 1 to n^2, the j loop runs from 1 to i that's n^2, the k loop runs from 1 to j that's also n^2 and the total is n^6.

B)

i = 1:

sum calculation:

j = 1: sum = 10 (A [0] added)

k = 1: sum = 32sum (A [1] added)

i < n: B[0] ≤ 32 ⇒ count = 1

i = 2:

sum calculation:

j = 1: sum = 10 (A [0] added)

k = 1: sum = 32 (A [1] added)

j = 2: sum = 42

k = 1: sum = 64

k= 2: sum = 119

i < n: B[1] ≤ 119 ⇒ count = 2

i = 3:

sum calculation:

j = 1: sum = 10 (A [0] added)

k = 1: sum = 32 (A[1] added)

j = 2: sum = 42

k = 1: sum = 64

k = 2: sum = 119

j = 3: sum = 129

k = 1: sum = 151

k = 2: sum = 206

k = 3: sum = 296

i < n: B[2] ≤ 296 ⇒ count = 3

Thus, the output is count = 3

3.

Algorithm consecutiveSimilarElements (A):

Input: Array A of size n

Output: Values of consecutive similar elements

Initialize empty stack ← S

Initialize variable i ← 0

Initialize variable n ← length(A)

```
While i < n:
        count =1
        While i + 1 < n and A[i] == A[i +1]:
                count = count +1
        if count > 1:
                Push (A[i], i-count + 1, count) to S
        i = i + 1

While not S.isEmpty():
        (element, start, count) = S.pop()
        Print “Element “ + Element + “ is repeated “ + count + “ times from index“ + start
```

A)

I'll be using the stack method since it is useful for comparing elements, the last in first out helps facilitate the process since the ends can be accessed straightforwardly. Simply put, I will iterate through every element of the array and update the values if an element happens to be smaller than the current minimum, if the difference is larger then update the value for the current maximum.

B)

The array is traversed once only thus the worst case is O(n).

C)

In the best case, the entire array is still traversed once thus the best case is Ω(n).

D)

The space complexity is O(n) since although we are stacking at maximum one element, we could have a scenario where most elements in the array are unique, which causes the stack to grow to a number that is similar to the array before the pop. However, the stack will usually be in constant time.

4.

A)

Outer loop runs from 0 to n, inner loop runs from 1 to 1024 but it remains constant, therefore the total time complexity is O(n).

B)

Outer loop has a time complexity of O(log n), inner loop runs from 0 to i and the time complexity is O(i), therefore the total time complexity is O(n log n).

C)

Outer loop runs from 1 to n and doubling with a time complexity of O(log n), inner loop runs from 1 to i doubling with a time complexity of O(log i), therefore the total time complexity is $O((\log n)^2)$.

D)

Outer loop runs from 0 to n with a time complexity of O(n), the inner loop has a time complexity of O (log n), therefore the total time complexity is O(n log n).

5.

A)

Algorithm PolyalphabeticEncryption(P, K):

Input:Plaintext P, key K

Output: Ciphertext C

Initialize C ← empty string

n = length (P)

for i = 0 to n-1 do

    shift = K + (i mod 3)

    if P[i] is a letter:

        newChar = ((ord(P[i]) – ord (‘A’) + shift) % 26) + ord (‘A’)

        Append newChar to C

    else:

    Append P[i] to C

return C

B)

Algorithm PolyalphabeticDecryption(C,K):

Input: Ciphertext C, key K

Output: Plaintext P

Initialize P ← empty string

n = length (C)

for i = 0 to n-1 do

    shift = K + (i mod 3)

    if C[i] is a letter:

        newChar = ((ord(C[i]) – ord (‘A’) – shift + 26) % 26) + ord (‘A’)

        Append newChar to P

else:

    Append C[i] to P

return P

C)

The time complexity of the encryption algorithm is O(n) since the algorithm iterates through every character of the input string P. The time complexity of the decryption algorithm is O(n) since the algorithm iterates through every character of the input string C.

D)

The space complexity of my encryption algorithm is O(n) since the space required for the output string C is O(n) where n is the length of the plaintext P. The space complexity of my decryption algorithm is O(n) since the space required for the output string P is O(n) where n is the length of the ciphertext C.

6.

i) Relationship is $f(n) = (\Omega\ g(n))$ because n^4 grows faster than $(\log n)^2$ and $\log (\log n)$.

ii) Relationship is $f(n) = (O\ g(n))$ because $(\log n)^2$ grows faster than $2\log n^2$.

iii) Relationship is $f(n) = (\Omega\ g(n))$ because $n!$ grows faster than 2^n.

iv) Relationship is $f(n) = (\Omega\ g(n))$ because $n^{3/2}$ grows faster than $(\log n)^2$.

v) Relationship is $f(n) = (O\ g(n))$ because $10n^n$ grows faster than both 2^{2n} and 10^n.

vi) Relationship is $f(n) = (O\ g(n))$ because n^n grows faster than $n!$.

vii) Relationship is $f(n) = (O\ g(n))$ because n^n grows faster than 2^{n*2n}.

viii) Relationship is $f(n) = (\Omega\ g(n))$ because n grows faster than $\log^2 n$

ix) Relationship is $f(n) = (\Omega\ g(n))$ because $\sqrt{n}$ grows faster than $\log n$.

x) Relationship is $f(n) = (O\ g(n))$ because n^{3n} grows faster than 2^n.