**Hoja de Problemas**

**Grupo M**

1. Problema 1

Pseudocódigo de suma\_max\_subarrays(A[1…n], i, j)

If i = j

Return A[i], i, j

Else

k <- ⌊(i + j) / 2

Mayor1 <- Maximo numero negativo

Tot <- 0

For s= k, s < i - 1, s <- s - 1

Tot <- Tot + A[s]

If tot > Mayor1

Mayor1 <- tot

Mayor2 <- Maximo numero negativo

Tot <- 0

For s= k + 1, s < j + 1, s <- s + 1

Tot <- Tot + A[s]

If tot > Mayor2

Mayor2 <- Tot

Sum1, i1, j1< suma\_max\_subarray(A[1…n], i, j)

Sum2, i2, j2<-suma\_max\_subarray(A[1…n], k + 1, j)

Aux <- 0

Ia<- 0

ja <- 0

If sum1 >= sum2

Aux <- sum1

Ia = i1

Ja = j1

Else

Aux <- sum2

Ia <- i2

Ja <- j2

If auz >= (mayor1 + `mayor2)

Return Aux, ia, ij

Else

Return (mayor1 + mayor2), ia, ja

1. Problema 2

Pseudocódigo de Iterativo2(A[1…2n + 1])

A’ 🡨 A

M = 0

Hueco = n/2

Ficha = hueco - 1

P 🡨 ()

While m < 2n – 1

Apilar(A’, p)

A[Hueco’] <-> A[Ficha’]

If ficha = n + 1

M 🡨 M + 1

If n <= Ficha’ <= n + 2

M🡨 M + 1

If Hueco’ – 1 <= Ficha’ <= Hueco + 1

Ficha’ 🡨 Hueco’ – A[Hueco’]

Elseif A[Hueco’] = A[Ficha’ + A[Hueco’]]

Ficha’ 🡨 Ficha’ + A[Hueco’]

Else

Ficha’ 🡨 Ficha’ + A[Hueco’] \* 2

Y 🡨 0

If A.length %2 == 0

Desapilar(p)

Y 🡨 Y + 1

While not vacía(p)

C[] 🡨 Desapilar(p)

Hueco’ 🡨 n – c[1] + 1

Ficha’ 🡨 n – c[2] + 1

A[Hueco’] <-> A[Ficha’]

Print(A[1…2n + 1])

M’ 🡨 c[3]

Y 🡨 y + 2

Return Y

1. Problema 3

Pseudocódigo de Stoogesort(A[1…n])

If A.[1] > A[n]

A[1] <-> A[n]

If n > 2

m ← ⌊n/3⌋

Stoogesort(A[1...n-m])

Stoogesort(A[m + 1...n])

Stoogesort(A[1...n-m])

Aplicando el algoritmo del teorema maestro podemos calcular el coste temporal del algoritmo de divide y vencerás.

T(n) = a \* T(n/3) + O(n^c)

De esta formula podemos diferenciar 3 casos

* a > b^c en el que T(n) es O(n^c)
* a = b^c en el que T(n) es O(
* a > b^c en el que T(n) es O()

Procedomos a dar los valores a,b y c

a vale 3 porque hace tres llamadas de la función recursiva debido hay que hay tres subproblemas.

Como el algoritmo se divide en 2/3 ya que si observamos la primera y ultima llamada recorre el array de los dos primeros tercios y en la segundo recorre solo los dos últimos tercios, entonces podemos decir que b tiene un valor de 3/2.

Por ultimo podecimos decir que como es complejidad constante c tiene un valor de 0.

Entonces nos queda la siguiente formula

T(n) = 3 \* T(2n/3) + O(1)

Por lo que nos queda que es el tercer caso

Debido a que a es mayor b^c

1. (3/2)^0

Ahora nos queda el siguiente algoritmo

O() = O() = O(

Concluimos que el algoritmo tiene un complejidad mayor que O(n^2)

pero menor O(n^3) aun así no es un algoritmos de ordenación que rente hacer debido a que existen otros dos algoritmos bastante más rápidos como lo es Quicsort y Mergesort.

4)Problema 4