	Motivación y Coherencia
1	Menciona que la **motivación** es aprender a emplear algoritmos utilizando grafos, programación dinamica y algoritmos codiciosos para la resolución de problemas.
1	Indicar **relevancia de técnicas algorítmicas**, como prog. dinámica y algoritmos codiciosos.
1	Indicar **relevancia de grafos**.
1	**Conclusión coherente**. Cierre y resumen claro de lo mencionado anteriormente
1	**Coherencia general**. El informe es coherente, empleando un razonamiento profesional y atinente a los contenidos del curso.
	Parte 1: Cerebros Codiciosos

2	Siendo **P planetas y R rango**, el algoritmo debe obtener la **cantidad mínima** de naves de rango R que cubran cada planeta P. Debería ser un algortimo similar al siguiente:
	function GREEDY(P, R)
	    P = **Ordenar**(P) ▷ *Complejidad de tiempo* $\mathcal{O}(P\log(P))$
	    posición_útima_nave_puesta = P[0] + R ▷ *Por simplicidad, se asume* $\|P\| > 0$
	    cantidad_naves_puestas = 1
	    for i_planeta = 1 **to** |P| **do** ▷ *Complejidad de tiempo* $\mathcal{O}(P)$
	        if posición_útima_nave_puesta + R < P[i_planeta] **then**: ▷ *No es cubierto*
	            posición_útima_nave_puesta = P[i_planeta] + R
	            cantidad_naves_puestas += 1
	    return naves_puestas ▷ *Mínima cantidad de naves necesarias*
	Se espera que quede claro que, al realizar el ordenamiento, el óptimo local sea el óptimo global en el algoritmo. Esto es debido a que posicionar las naves lo más a la derecha del primer planeta libre (ordenado de izquierda a derecha), máximiza la cobertura que tiene al resto de los planetas a la derecha. Al maximizar hacia la derecha, se reduce la cantidad de planetas a cubrir, optimizando globalmente la cantidad de naves.
	En el caso de que se pidiera la posición de las naves, se usaría un arreglo en vez de posición_útima_nave_puesta y cantidad_naves_puestas.

1	El algoritmo es correcto y hace **solamente lo pedido**. No guarda un arreglo con las posiciones de las naves.
1	Explica la complejidad de **tiempo** promedio es $\mathcal{O}(N\log(N))$ y que **depende del ordenamiento**. Se considera correcto cuando se menciona el peor caso que tiene el ordenamiento implementado.
1	Explica que la complejidad de **memoria** es $\mathcal{O}(P)$. Si es que almacena el arreglo de posiciones de la nave, debe tener comeplejidad $\mathcal{O}(P+N)$.

<table>
<tr><th colspan="2">Parte 2: Acumulando Basura</th></tr>
<tr><td>2</td><td>
Se explica un algoritmo similar al problema de vuelto con monedas.
Se entrega un **objetivo** n y una **lista de valores** V. Se asume valores mayores que 0.

```
function DP(n)
    Inicializar el arreglo dp[] de tamaño n + 1 con valores infinitos
    dp[0] ← 0                                   ▷ 0 valores para alcanzar la suma 0
    for i to n do                               ▷ Por cada DP(i)
        for v in V do
            if i − v ≥ 0 then
                dp[i] ← min(dp[i], dp[i − v] + 1)
    if dp[n] es infinito then
        return −1                               ▷ No es posible alcanzar la suma n con los valores dadas
    else
        return dp[n]                            ▷ El número mínimo de valores para alcanzar la suma S
```

El arreglo dp podría tener un número especial (como -1) en vez de infinito, de tal manera que $dp[j] \leftarrow dp[j - V_i] + 1$ cuando $dp[j]$ es el numero especial (siempre es considerado como el mayor). Cualquier número mayor a n cumple esa propiedad sin requerir cambios en la función.
Debe quedar claro que la función está definida similarmente a:

$$DP(n) = \begin{cases} 1 + \min\left(\{+\infty\} \cup \{DP(n - v_i) \mid v \in V \wedge n - v \geq 0\}\right) & n > 0 \\ 0 & n = 0 \end{cases}$$

Como la llamada a DP siempre es a un valor menor, se puede usar programación dinámica.
Una solución alternativa usa un arreglo de **dos dimensiones**, donde la segunda dimensión es el valor. Antes de ver la condición $i - v \geq 0$, se obtiene la cantidad obtenida con el valor anterior: $dp[i][j\_valor] \leftarrow dp[i][j\_valor - 1]$ con $i \geq 1$ y $dp[i][0] = +\infty$ (diapo 7 de clase 19 sec 1). Esta versión permite obtener la cantidad de basurales usada por cada tipo, que no es pedido aquí.
</td></tr>
<tr><td>1</td><td>
Se explica **una** de las siguientes optimizaciones:

- Uso de un **arreglo de una dimensión**.
- Eliminar valores repetidos en V sin aumentar la complejidad de tiempo.
- No consideran valores mayores a S al crear la lista V.

Puede ser explicado implicitamente en el algoritmo.
</td></tr>
<tr><td>1</td><td>Explica que la **complejidad de tiempo** es $\mathcal{O}(nV)$. En el caso que se ordene los valores, debería quedar en $\mathcal{O}(nV + V\log(V))$. En ese caso, se considera malo el punto anterior (optimizaciones).</td></tr>
<tr><td>1</td><td>Explica que la **complejidad de memoria** es $\mathcal{O}(n)$ con n el número objetivo.
De forma alternativa, si no reducen en una dimensión el arreglo, la complejidad es $\mathcal{O}(nV)$.</td></tr>
</table>

<table>
<tr><th colspan="2">Parte 3: Identificación de regiones</th></tr>
<tr><td>2</td><td>Explica un algoritmo que itera por el grafo y marca los nodos de forma que permita contar la cantidad de componentes.

function RecorrerGrafoDFS(n, vértices)
    Marcar n como recorrido
    for (n, v) en vértices do
        if v no ha sido recorrido then
            RecorrerGrafoDFS(v, vértices)
function RecorrerGrafoBFS(raíz, vértices)
    Crear cola [raíz]
    while exista nodo en la cola do
        Sacar un nodo n de la cola
        Marcar n como recorrido
        for (n, v) en vértices do
            if v no ha sido recorrido then
                Añadir v a la cola
function ObtenerComponentesConexos(vértices)
    recorridos = 0
    for cada nodo n_i do
        if n_i no ha sido recorrido then
            recorridos += 1
            RecorrerGrafo(n_i, vértices)   ▷ Puede ser por DFS o BSF

De forma alternativa, se podría utilizar un algoritmo que no almacena los vértices, usando una estructura como set disjuntos (UnionFind), armando árboles por cada componente.
En ese caso, la complejidad de tiempo es aproximadamente y no exacta a $\mathcal{O}(N+V)$, y la complejidad de memoria es $\mathcal{O}(N)$.</td></tr>
<tr><td>1</td><td>Se describe claramente como se almacenan y recorren por los vértices. No se entrega puntaje si se modela con matriz de adyacencia, que es ineficiente en grafos no dirigidos, no ponderados, y con múltiples componentes.
Si el algoritmo que no almacena vértices, se debe explicar como este modela la unión de nodos para llegar a una complejidad de tiempo cercana a $\mathcal{O}(N+V)$.</td></tr>
<tr><td>1</td><td>Explica que la complejidad de tiempo es $\mathcal{O}(N+V)$ (puede ser aproximada).
Esto es debido a que cada nodo se recorre 1 vez, y en cada nodo, se ve cada uno de sus vértices.</td></tr>
<tr><td>1</td><td>Explica que la complejidad de memoria es $\mathcal{O}(N+V)$.
Es necesario guardar un arreglo $\mathcal{O}(N)$ que indique si un nodo fue recorrido o no. También, es necesario guardar una lista de adyacencia $\mathcal{O}(N+V)$, que tiene forma de diccionario donde donde la llave es un nodo $\mathcal{O}(N)$ y los valores es una lista de nodos $\mathcal{O}(V)$.</td></tr>
</table>