Método 2: Greedy con Orden Aleatorio de Pedidos para Minimizar $\max(W_j)$

1. Propósito del Orden Aleatorio

En el Método 1 (Greedy determinista), los pedidos se ordenaban de mayor a menor tiempo promedio y se asignaban en esa secuencia, produciendo siempre el mismo resultado si se mantenían los datos sin cambios. Ahora, en este Método 2 (Greedy con Orden Aleatorio), la idea principal es:

- la lógica de asignación que minimiza $\max(W_j)$ de forma codiciosa (greedy),
- Cambiar la fase de ordenar pedidos y, en su lugar, barajar o mezclar la lista de pedidos aleatoriamente antes de asignarlos.

De esta manera, cada ejecución puede potencialmente producir una solución distinta, explorando otras secuencias de asignación y pudiendo en ocasiones descubrir distribuciones más balanceadas que las logradas por un orden determinista.

2. Detalles de la Aleatorización

2.1. Cuándo y cómo se barajan los pedidos

- Al iniciar el método, en lugar de calcular el tiempo promedio y ordenar de mayor a menor (como en el Método 1), se toma el listado `pedidos` y se aplica una función de shuffle (por ejemplo, `random.shuffle(pedidos)`) que mezcla sus elementos de modo aleatorio.
- Por tanto, el primer pedido asignado podría ser uno que, en el método anterior, se colocaba al final, o viceversa. Rompiendo así la estructura fija y predecible de las asignaciones, agregando diversidad en las soluciones.

2.2. Efectos sobre la calidad y la repetibilidad

- **Calidad**: Al asignar los pedidos en un orden distinto, podemos encontrar en ocasiones soluciones mejores, y en otros casos, peores. Pero el método local de minimizar la subida de $\max(W_j)$ se mantiene.
- **Repetibilidad**: Si no se fija una semilla aleatoria, cada ejecución produce una secuencia diferente.
- **Multi-ejecución**: Se debe ejecutar varias veces este método y luego seleccionar la mejor de las soluciones halladas. Esto no se hacía en el Método 1, que siempre daba la misma respuesta.

3. Pseudocódigo del Método Greedy con Orden Aleatorio

ALGORITMO `build_greedy_random_order()`**:**

1. **LEER datos del problema** (idénticos al Método 1):
   - `pedidos`, `salidas`, `zonas`
   - `n_sal`[j], `s`[(j,k)], `tiempo`[(i,k)]
2. **Aleatorizar la lista de pedidos:**
   - `pedidos_aleatorios` ← Copia de `pedidos`
   - `shuffle`(`pedidos_aleatorios`) (mezcla sus elementos)
3. **Inicializar estructuras de asignación**:
   - $X = \{\}$ (diccionario: X[i] = k),
   - `used_positions` $= \emptyset$,
   - `zone_usage`$[j] = 0$ para cada zona j,
   - $W[j] = 0.0$ para cada zona j.
4. **PARA cada pedido i en** `pedidos_aleatorios`:

   (4.1) `best_k` = NULO, `best_valor` $= +\infty$

   (4.2) **PARA cada salida k en** `salidas`:
   - si k **no** está en `used_positions`
   - `zona_k` = zona a la que pertenece k
   - si `zone_usage`[`zona_k`] < `n_sal`[`zona_k`]:
     - Asignar $i \rightarrow k$ *temporalmente*
     - Calcular `new_max` $= \max(W[j])$
     - si `new_max` < `best_valor`: actualizar `best_k`, `best_valor`
     - revertir la asignación

   (4.3) si `best_k` $\neq$ NULO:
   - Asignar $i \rightarrow$ *best_k* en firme (marcar salida ocupada, aumentar W de la zona, etc.)
5. **Al final**: `max_time` $= \max(W[j])$.
6. **RETORNAR**: (X, W, `max_time`).