Quadro 4: Output do algoritmo de ordenação por mistura com acompanhamento das variáveis.

```
def mergesort(a, l, r):

    if l < r:
        m = (l + r) // 2
        mergesort(a, l, m)
        mergesort(a, m+1, r)
        merge(a, l, m, r)

a = [7, 6, 5, 4, 3, 2, 2, 1]
mergesort(a, 0, len(a)-1)
```

I	01 r (last m value): 7
	02 pre-mergesort1: m, l, r: 3 0 7
	03 r (last m value): 3
	04 pre-mergesort1: m, l, r: 1 0 3
	05 r (last m value): 1
	06 pre-mergesort1: m, l, r: 0 0 1
	07 r (last m value): 0
	08 pos-mergesort1: m, l, r: 0 0 1
	09 r (last m value): 1
	10 pos-mergesort2: m, l, r: 0 0 1
	11 merge: m, l, r: 0 0 1
	12 L: [7]
	13 R: [6]
	14 a: [6, 6, 5, 4, 3, 2, 2, 1]
	15 a: [6, 7, 5, 4, 3, 2, 2, 1]
		O valor de entrada no código é l = 0 e r = 7. Assim, naturalmente, o valor de r será 7 e o de m será 3 (= (0+7) // 2). Estes valores são inseridos, então, dentro da própria função mergesort() uma primeira vez; a função chamada no corpo do programa, então, para, e uma nova chamada sua é feita, com os valores de l e r sendo, respectivamente, 0 e m.
		Nesta primeira chamada interna (linha 03), a chamada filha, l = 0 e r = 3, fazendo m ser 1 (= (0+3) // 2).
		Estes valores são aplicados, então, em uma segunda chamada interna (linha 05), "neta" da original. Aqui, l = 0 e r = 1, fazendo m ser 0 (= (0+1) // 2). Estes valores aplicados em uma terceira chamada farão a condição de entrada ser falsa (ver linha 07); assim, este método é concluído.
		Com a conclusão do anterior, pode-se executar a segunda recursão da "chamada-neta". Ali, l = 0, r = 1 e m = 0. Porém, como esta chamada aplica o valor de r como r e l como m+1,. esta segunda chamada recebe os valores l = 1 e r = 1, que fará a condição de entrada ser falsa, concluindo este segundo chamado. Após, finalmente, o método `merge()` será chamado (linha 11) para a chamada-neta, com l = 0, m = 0 e r = 1. Isso implica na criação das listas left e right com, respectivamente, o primeiro e o segundo elementos do arranjo original. Esta criação de listas resolve a chamada-neta, adicionado seus elementos ordenadamente no início do arranjo original

II	16 pos-mergesort1: m, l, r: 1 0 3 17 r (last m value): 3 18 pre-mergesort1: m, l, r: 2 2 3 19 r (last m value): 2 20 pos-mergesort1: m, l, r: 2 2 3 21 r (last m value): 3 22 pos-mergesort2: m, l, r: 2 2 3 23 merge: m, l r: 2 2 3 24 L: [5] 25 R: [4] 26 a: [6, 7, 4, 4, 3, 2, 2, 1] 27 a: [6, 7, 4, 5, 3, 2, 2, 1]	Após, entra-se na segunda recursão da chamada filha, onde l = 0, m = 1 e r = 3 (ver linhas 04 e 16). Assim, esta segunda recursão recebe os argumentos l = 2(=m+1) e r = 3; neste caso, a condição de entrada é verdadeira, e são aplicados ao primeiro `mergesort()` (linha 18), também neto da original. Dentro dele, l = 2, r = 3 e m é definido como 2(= (3+2) // 2), conforme linha 17. Assim, esta última aplicação ocorreria para mais dois mergesort(), mas como estes valores não validam a condição de entrada, eles são mantidos para o segundo uso (no histórico) do `merge()`. Com estes valores, são criadas novas listas left e right (independentes das últimas), com os segundo e terceiro termos do arranjo original apenas, em função dos argumentos l = 2, m = 2 e r = 3, pois seus comprimentos serão unitários (n1 = m - l + 1 ⇒ n1 = 2-2+1 ⇒ n1 = 1 & n2 = r - m ⇒ n2 = 3-2 ⇒ n2 = 1) e os índices do arranjo a serem inseridos em left e right são, respectivamente, l+i (⇒ 2+0 = 2), ou seja, 3° elemento e m+j+1 (⇒ 2+0+1 = 3), ou seja, o 4° elemento. Com estas criações, a chamada neta é resolvida, colocando seus elementos ordenadamente (em relação a si) nos respectivos elementos da lista original.
III	28 pos-mergesort2: m, l, r: 1 0 3 29 merge: m, l r: 1 0 3 30 L: [6, 7] 31 R: [4, 5] 32 a: [4, 5, 4, 5, 3, 2, 2, 1] 33 a: [4, 5, 6, 7, 3, 2, 2, 1]	A seguir, entra-se na aplicação do `merge()` da chamada filha, onde l = 0, m = 1 e r = 3 (ver linhas 04, 16 e 28). Assim, merge() recebe estes argumentos, fazendo (n1 = m - l + 1 ⇒ n1 = 1-0+1 ⇒ n1 = 2 & n2 = 3 - 1 ⇒ n2 = 2) as listas left e right terem ambas comprimento igual a 2; após, estas listas são preenchidas com base na modificada no último processo. Assim, o merge() da chamada filha, que é o primeiro mergesort() original, emite a modificação da lista original com a primeira metade ordenada.
IV	34 a 65	O processo se repete similarmente por mais 32 linhas de outputs, ordenando os últimos quatro elementos da lista original, porém, de forma independente dos quatro primeiros elementos, resolvendo a chamada filha do segundo mergesort() recursivo da **chamada original**. Até aqui, portanto, temos duas "metades ordenadas", mas o arranjo, como um todo, não está ordenado.

Link para problemas resolvidos

V	66 pos-mergesort2: m, l, r: 3 0 7 67 merge: m, l r: 3 0 7 68 L: [4, 5, 6, 7] 69 R: [1, 2, 2, 3] 70 a: [1, 2, 2, 3, 1, 2, 2, 3] 71 a: [1, 2, 2, 3, 4, 5, 6, 7]	Aqui, finalmente teremos a entrada no `merge()` da chamada original com os valores l = 0, m = 3 e r = 7 (ver linhas 02 e 66 — que aparece novamente na linha 34, para iniciar a segunda metade da lista). Com estes valores, os comprimentos de left e right serão iguais a 4, recebendo, respectivamente, os quatro primeiros e os quatro últimos elementos do arranjo original conforme a última modificação (na linha 65, suprimida: [4, 5, 6, 7, 1, 2, 2, 3]). A inserção de seus elementos, então, modificará a lista original de forma integralmente ordenada.