Notebook 1: Bipartite Graph Check

Source file: `bipartite.ipynb`

Problem

An undirected graph is **bipartite** if its vertices can be split into two sets so every edge goes between the sets, equivalently if the vertices can be 2-coloured with no edge joining two equal colours. `is_bipartite(G)` returns True/False. Six test cases follow.

Hints

• Try to 2-colour the graph with BFS: give the start vertex one colour and every newly discovered neighbour the *opposite* colour.
• If you ever meet an edge whose two endpoints share a colour, the graph cannot be bipartite.
• Remember the graph may be empty or disconnected.

Test case 1 / 6

```
is_bipartite([]) # should print True
```



No vertices, so there is nothing to colour: vacuously bipartite.

Result: True ✔ bipartite

No vertices ⇒ the loop never runs ⇒ nothing can conflict ⇒ **True**.

✔ Output True matches the notebook's expected value.

Test case 2 / 6

```
is_bipartite([[]]) # should print True
```



Colour A (blue): {0}

Colour B (orange): {}

Every edge joins A ↔ B.

Result: True ✔ bipartite

BFS 2-colouring trace: Start BFS at 0: colour 0 = A. No edge joined two equal colours, so the answer is **True**.

✔ Output True matches the notebook's expected value.

Test case 3 / 6

```
is_bipartite([[1], [0]]) # should print True
```



Colour A (blue): {0}

Colour B (orange): {1}

Every edge joins A ↔ B.

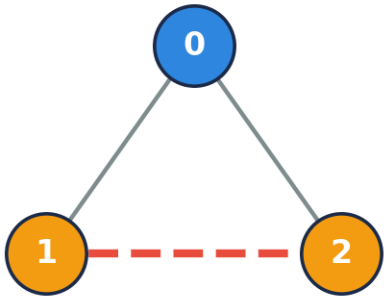
Result: True ✔ bipartite

BFS 2-colouring trace: Start BFS at 0: colour 0 = A. Visit 0: neighbour 1 is new → colour 1 = B. Visit 1: neighbour 0 already coloured A ≠ 1's colour → edge OK. No edge joined two equal colours, so the answer is **True**. Two vertices, one edge: A–B is trivially two-colourable.

✔ **Output True matches the notebook's expected value.**

Test case 4 / 6

```
is_bipartite([[1, 2], [0, 2], [1, 0]]) # should print False
```



Colour A (blue): {0}

Colour B (orange): {1, 2}

Same-colour edge(s) in red:
(1,2)

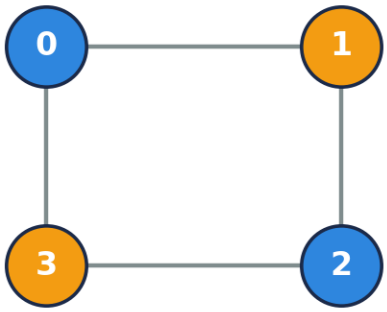
Result: False ✘ not bipartite

BFS 2-colouring trace: Start BFS at 0: colour 0 = A. Visit 0: neighbour 1 is new → colour 1 = B. Visit 0: neighbour 2 is new → colour 2 = B. Visit 1: neighbour 0 already coloured A ≠ 1's colour → edge OK. Visit 1: neighbour 2 already has colour B – the SAME as 1. Edge (1,2) joins equal colours → NOT bipartite, return False. Any odd cycle (here a triangle) can never be 2-coloured.

✔ **Output False matches the notebook's expected value.**

Test case 5 / 6

```
is_bipartite([[1, 3], [0, 2], [1, 3], [2, 0]]) # should print True
```



Colour A (blue): {0, 2}

Colour B (orange): {1, 3}

Every edge joins A ↔ B.

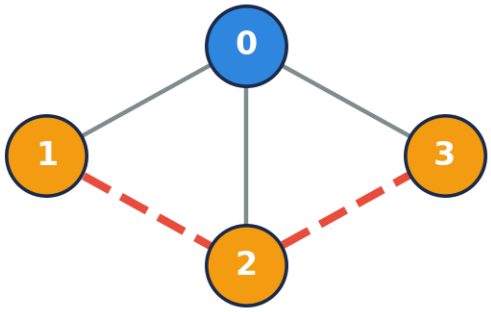
Result: True ✔ bipartite

BFS 2-colouring trace: Start BFS at 0: colour 0 = A. Visit 0: neighbour 1 is new → colour 1 = B. Visit 0: neighbour 3 is new → colour 3 = B. Visit 1: neighbour 0 already coloured A ≠ 1's colour → edge OK. Visit 1: neighbour 2 is new → colour 2 = A. Visit 3: neighbour 2 already coloured A ≠ 3's colour → edge OK. Visit 3: neighbour 0 already coloured A ≠ 3's colour → edge OK. Visit 2: neighbour 1 already coloured B ≠ 2's colour → edge OK. Visit 2: neighbour 3 already coloured B ≠ 2's colour → edge OK. No edge joined two equal colours, so the answer is **True**. The even 4-cycle alternates A,B,A,B: U={0,2} and V={1,3} as in the notebook intro.

✔ Output True matches the notebook's expected value.

Test case 6 / 6

```
is_bipartite([[1, 2, 3], [0, 2], [0, 1, 3], [2, 0]]) # should print False
```



Colour A (blue): {0}

Colour B (orange): {1, 2, 3}

Same-colour edge(s) in red:
(1,2), (2,3)

Result: False ✘ not bipartite

BFS 2-colouring trace: Start BFS at 0: colour 0 = A. Visit 0: neighbour 1 is new → colour 1 = B. Visit 0: neighbour 2 is new → colour 2 = B. Visit 0: neighbour 3 is new → colour 3 = B. Visit 1: neighbour 0 already coloured A ≠ 1's colour → edge OK. Visit 1: neighbour 2 already has colour B – the SAME as 1. Edge (1,2) joins equal colours → NOT bipartite, return False. (Edge (2,3) is also same-coloured; BFS stops at the first conflict it meets.)

✔ Output False matches the notebook's expected value.

Notebook 2: Sales Revenue in a Company Hierarchy

Source file: `bipartite (1).ipynb`

Problem

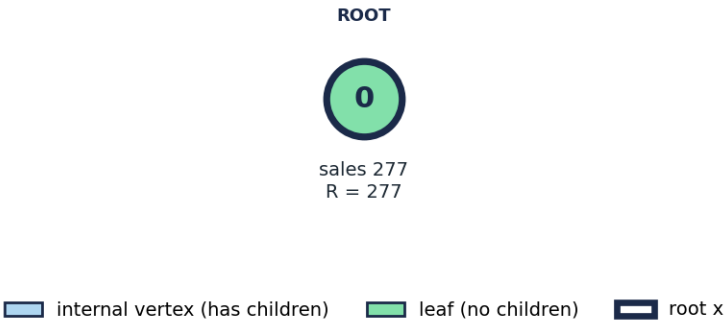
A company hierarchy is a tree rooted at vertex x. Only **leaf employees** make sales. `sales_value(G, x, sales)` returns *revenue*, where a leaf's revenue is its own sales and any other vertex's revenue is the sum of its children's revenues. Seven test cases follow.

Hints

• A DFS/BFS from the root x tells you which neighbours are children: the ones you haven't visited yet. Arrows in the figures show values flowing **up** from child to parent.
• A vertex with no children reports its own sales; every other vertex reports the sum of its children's revenues.
• Careful: a root with a single neighbour is *not* a leaf, while a lone root vertex is.

Test case 1 / 7

```
sales_value([[]], 0, [277]) # should print [277]
```



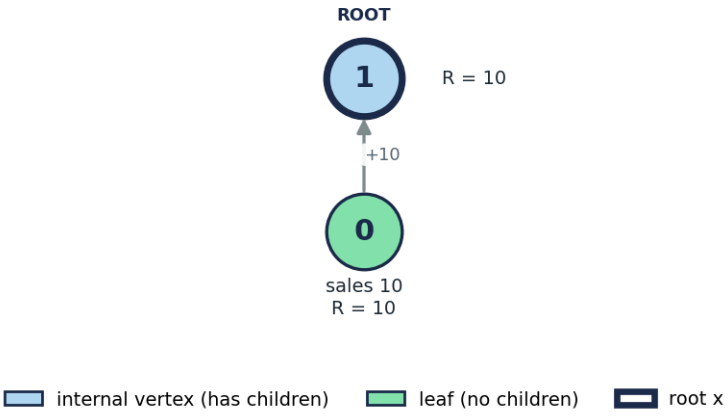
vertex	0
revenue	277

The tree is a single vertex. It is the root and also a leaf (no children), so its revenue is just its own sales: $R_0 = 277$.

✔ Output matches the notebook's expected value.

Test case 2 / 7

```
sales_value([[1], [0]], 1, [10, 0]) # should print [10, 10]
```



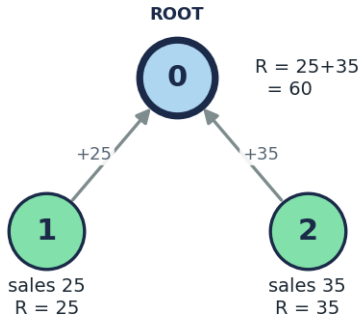
vertex	0	1
revenue	10	10

The root is x = 1 (not 0), so the tree is re-oriented from 1: 1 → 0. **Leaves** carry the sales: $R_0 = 10$. **Internal vertices** (bottom-up, children first): $R_1 = R_0 = 10$. Answer: **[10, 10]**

✔ Output matches the notebook's expected value.

Test case 3 / 7

```
sales_value([[1, 2], [0], [0]], 0, [0, 25, 35]) # should print [60, 25, 35]
```





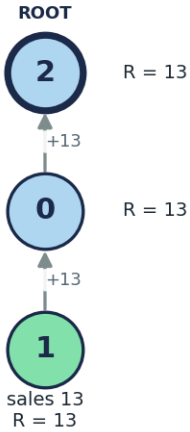
vertex	0	1	2
revenue	**60**	**25**	**35**

Leaves carry the sales: $R_1 = 25$, $R_2 = 35$. **Internal vertices** (bottom-up, children first): $R_0 = R_1 + R_2 = 25 + 35 = 60$. Answer: **[60, 25, 35]**

✔ **Output matches the notebook's expected value.**

Test case 4 / 7

```
sales_value([[1, 2], [0], [0]], 2, [0, 13, 0]) # should print [13, 13, 13]
```





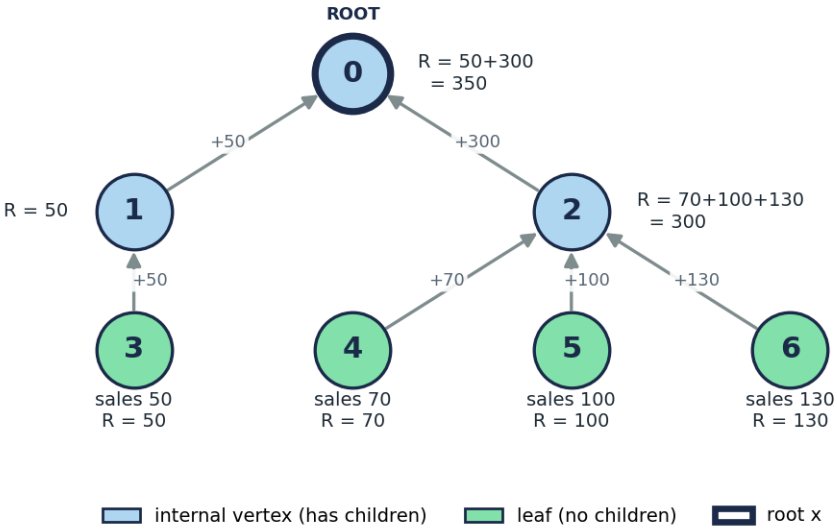
vertex	0	1	2
revenue	**13**	**13**	**13**

The root is x = 2 (not 0), so the tree is re-oriented from 2: 2 → 0 → 1. **Leaves** carry the sales: $R_1 = 13$. **Internal vertices** (bottom-up, children first): $R_0 = R_1 = 13$; $R_2 = R_0 = 13$. Answer: **[13, 13, 13]**

✔ **Output matches the notebook's expected value.**

Test case 5 / 7

```
sales_value([[1, 2], [0, 3], [0, 4, 5, 6], [1], [2], [2], [2]], 0, [0, 0, 0, 50, 70, 100, 130]) # should print
[350, 50, 300, 50, 70, 100, 130]
```



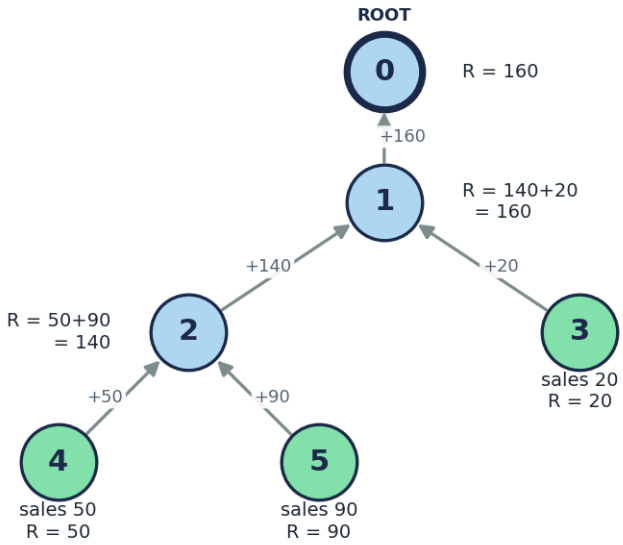
vertex	0	1	2	3	4	5	6
revenue	350	50	300	50	70	100	130

Leaves carry the sales: $R_3 = 50$, $R_4 = 70$, $R_5 = 100$, $R_6 = 130$. **Internal vertices** (bottom-up, children first): $R_2 = R_4 + R_5 + R_6 = 70 + 100 + 130 = 300$; $R_1 = R_3 = 50$; $R_0 = R_1 + R_2 = 50 + 300 = 350$. Answer: **[350, 50, 300, 50, 70, 100, 130]**

✔ **Output matches the notebook's expected value.**

Test case 6 / 7

```
sales_value([[1], [0, 2, 3], [1, 4, 5], [1], [2], [2]], 0, [0, 0, 0, 20, 50, 90]) # should print [160, 160, 140,
20, 50, 90]
```



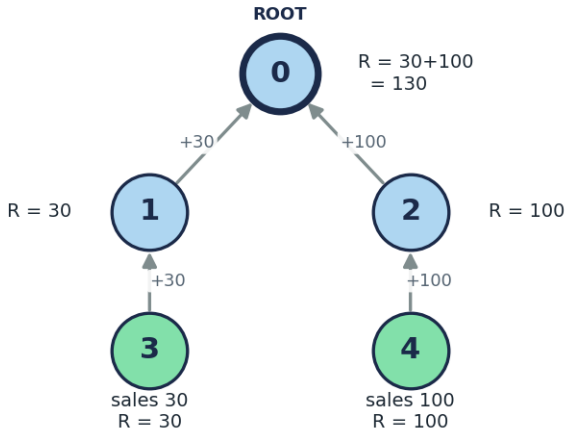
vertex	0	1	2	3	4	5
revenue	160	160	140	20	50	90

Leaves carry the sales: $R_4 = 50$, $R_5 = 90$, $R_3 = 20$. **Internal vertices** (bottom-up, children first): $R_2 = R_4 + R_5 = 50 + 90 = 140$; $R_1 = R_2 + R_3 = 140 + 20 = 160$; $R_0 = R_1 = 160$. Answer: **[160, 160, 140, 20, 50, 90]**

✔ **Output matches the notebook's expected value.**

Test case 7 / 7

```
sales_value([[1, 2], [0, 3], [0, 4], [1], [2]], 0, [0, 0, 0, 30, 100]) # should print [130, 30, 100, 30, 100]
```





vertex	0	1	2	3	4
revenue	**130**	**30**	**100**	**30**	**100**

Leaves carry the sales: $R_3 = 30$, $R_4 = 100$. **Internal vertices** (bottom-up, children first): $R_2 = R_4 = 100$; $R_1 = R_3 = 30$; $R_0 = R_1 + R_2 = 30 + 100 = 130$. Answer: **[130, 30, 100, 30, 100]**

✔ **Output matches the notebook's expected value.**

Notebook 3: Fair Inheritance Split in a Family Tree

Source file: `bipartite (2).ipynb`

Problem

A family tree is a tree rooted at vertex x. `inheritance_split(G, x, total)` returns each member's entitlement, assuming every parent splits what they received **evenly among their children** (values rounded to 2 decimals). Seven test cases follow.

Hints

• Traverse from the root x and pass each vertex's amount **down** to its children (arrows in the figures).
• Count children, not neighbours: the parent is also a neighbour, so a non-root vertex's degree is one too large.
• Round only the final values with `round(v, 2)`.

Test case 1 / 7

```
inheritance_split([[]], 0, 277) # should print [277]
```





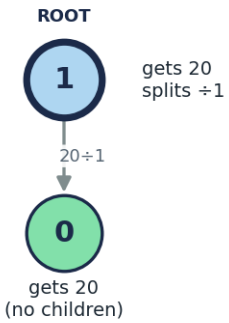
vertex	0
amount	277

Only one vertex: the head of the family. Nobody to split with, so it keeps the whole 277.

✔ **Output matches the notebook's expected value.**

Test case 2 / 7

```
inheritance_split([[1], [0]], 1, 20) # should print [20, 20]
```





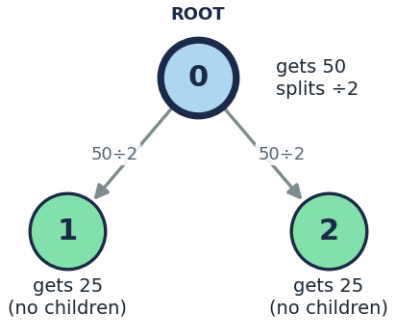
vertex	0	1
amount	20	20

The head of the family is x = 1 (not 0), so the tree is re-oriented from 1. The root starts with the full total = 20. **Splitting top-down:** 1 has 1 child (0): each gets 20/1 = 20. Vertices with no children stop the splitting. Answer: **[20, 20]**

✔ **Output matches the notebook's expected value.**

Test case 3 / 7

```
inheritance_split([[1, 2], [0], [0]], 0, 50) # should print [50, 25, 25]
```





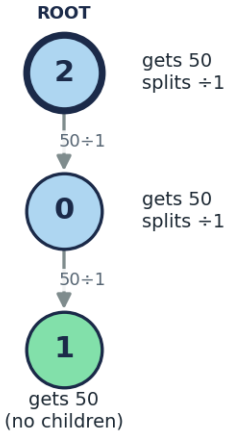
vertex	0	1	2
amount	50	25	25

The root starts with the full total = 50. **Splitting top-down:** 0 has 2 children (1, 2): each gets 50/2 = 25. Vertices with no children stop the splitting. Answer: **[50, 25, 25]**

✔ Output matches the notebook's expected value.

Test case 4 / 7

```
inheritance_split([[1, 2], [0], [0]], 2, 50) # should print [50, 50, 50]
```





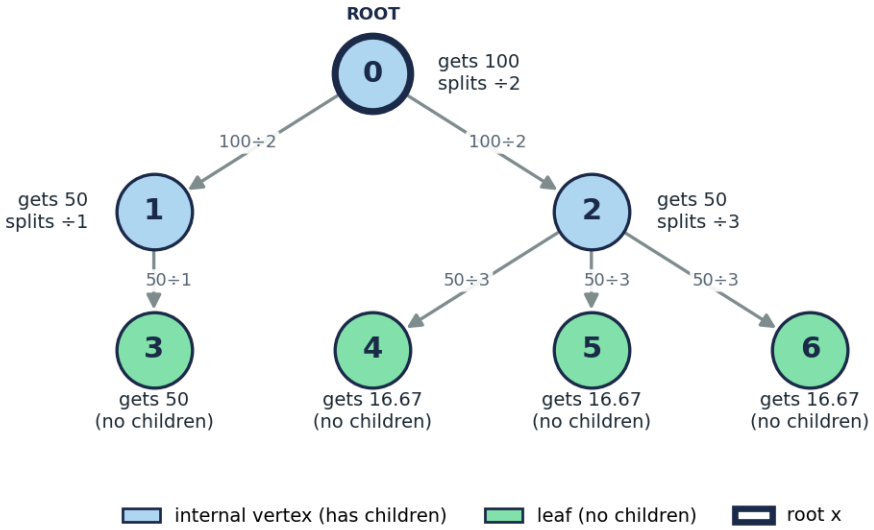
vertex	0	1	2
amount	50	50	50

The head of the family is x = 2 (not 0), so the tree is re-oriented from 2. The root starts with the full total = 50. **Splitting top-down:** 2 has 1 child (0): each gets 50/1 = 50; 0 has 1 child (1): each gets 50/1 = 50. Vertices with no children stop the splitting. Answer: **[50, 50, 50]**

✔ Output matches the notebook's expected value.

Test case 5 / 7

```
inheritance_split([[1, 2], [0, 3], [0, 4, 5, 6], [1], [2], [2], [2]], 0, 100) # should print [100, 50, 50, 50,
16.67, 16.67, 16.67]
```



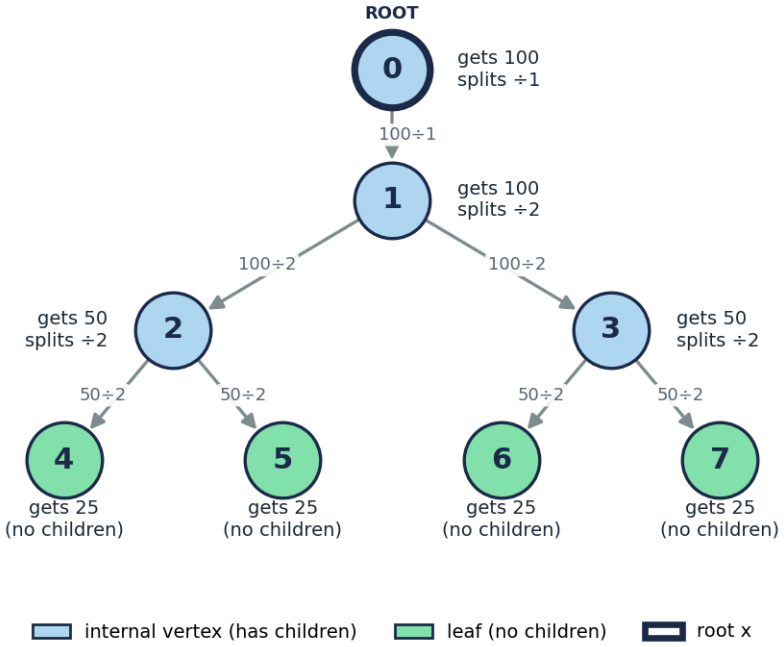
vertex	0	1	2	3	4	5	6
amount	100	50	50	50	16.67	16.67	16.67

The root starts with the full total = 100. **Splitting top-down:** 0 has 2 children (1, 2): each gets 100/2 = 50; 1 has 1 child (3): each gets 50/1 = 50; 2 has 3 children (4, 5, 6): each gets 50/3 = 16.67. Vertices with no children stop the splitting. Answer: **[100, 50, 50, 50, 16.67, 16.67, 16.67]**

✔ Output matches the notebook's expected value.

Test case 6 / 7

```
inheritance_split([[1], [0, 2, 3], [1, 4, 5], [1, 6, 7], [2], [2], [3], [3]], 0, 100) # should print [100, 100, 50, 50, 25, 25, 25, 25]
```



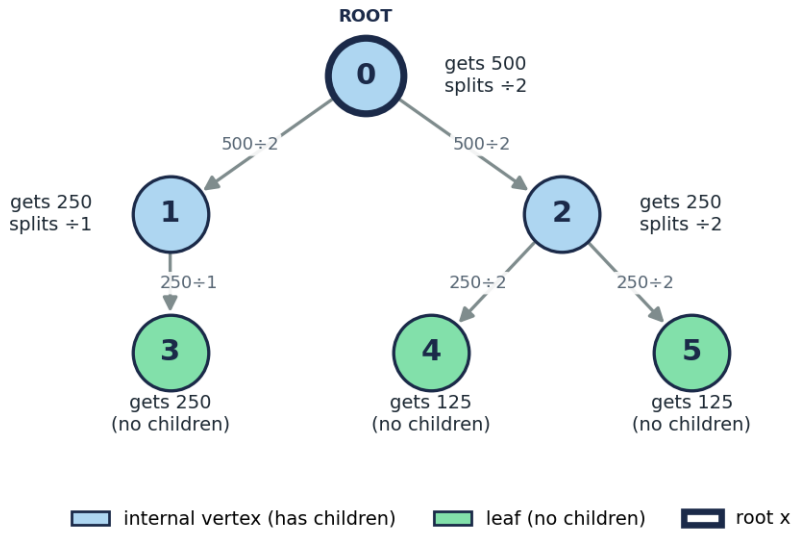
vertex	0	1	2	3	4	5	6	7
amount	100	100	50	50	25	25	25	25

The root starts with the full total = 100. **Splitting top-down:** 0 has 1 child (1): each gets 100/1 = 100; 1 has 2 children (2, 3): each gets 100/2 = 50; 2 has 2 children (4, 5): each gets 50/2 = 25; 3 has 2 children (6, 7): each gets 50/2 = 25. Vertices with no children stop the splitting. Answer: **[100, 100, 50, 50, 25, 25, 25, 25]**

✔ Output matches the notebook's expected value.

Test case 7 / 7

```
inheritance_split([[1, 2], [0, 3], [0, 4, 5], [1], [2], [2]], 0, 500) # should print [500, 250, 250, 250, 125, 125]
```



vertex	0	1	2	3	4	5
amount	500	250	250	250	125	125

The root starts with the full total = 500. **Splitting top-down:** 0 has 2 children (1, 2): each gets 500/2 = 250; 1 has 1 child (3): each gets 250/1 = 250; 2 has 2 children (4, 5): each gets 250/2 = 125. Vertices with no children stop the splitting. Answer: **[500, 250, 250, 250, 125, 125]**

✔ Output matches the notebook's expected value.