```java
public Node deepCopyLinkedListWithRandomPointer(Node head) {
    if (head == null) return null;
    // Lookup map to avoid duplicate. Key: original node. Value: copied node.
    Map<Node, Node> lookup = new HashMap<>();
    Node newHead = new Node(head.value);
    lookup.put(head, newHead);
    Node curr = newHead;
    while (head != null) {
        if (head.next != null) {    // copy next
            if (!lookup.containsKey(head.next)) {
                // Hasn't been copied over due to random pointer.
                lookup.put(head.next, new Node(head.next.value));
            }
            curr.next = lookup.get(head.next);
        }
        if (head.random != null) {  // copy random
            if (!lookup.containsKey(head.random)) {
                // Hasn't been copied over previously.
                lookup.put(head.random, new Node(head.random.value));
            }
            curr.random = lookup.get(head.random);
        }
        head = head.next;
        curr = curr.next;
    }
    return newHead;
}
```

Solution 2: DFS (recursion)

```
class Node {
      int value;
      vector<Node*> neighbors;    // Java: List<Node> neighbors
}
```

```
// Recursive manner. Use map to store whether a node has been copied before.
// For every single recursion function call, we make a copy of the input node, and leave all other copies of
// the successors to the recursion functions.
public Node cloneGraph(Node input, Map<Node, Node> lookup) {
    if (input == null) return null;
    if (lookup.containsKey(input)) {
        return lookup.get(input);
    }
    Node copyNode = new Node(input.value);
    lookup.put(input, copyNode);
    for (Node neighbor : input.neighbors) {
        copyNode.neighbors.add(cloneGraph(neighbor, lookup));
    }
    return copyNode;
}
```

Q2: **k-way merge problems**:

Q2.1 How to merge **k sorted arrays** into one big sorted array?

Assumptions:

- length
- ascending/descending
- duplicate
- data type
- fit in memory

Solution 1: k pointers all together

A1 xxxxxxxxxxxxxxxxxxx

p1 →

A2 yyyyyyyyyyyyyyyyyyy

p2 →

A3 zzzzzzzzzzzzzzzzzzz

p3 →

...

Ak

pk →

Use an heap to keep the k pointers.

Time = O(nk*log(k))

```
class Element {
        int index_of_array;     // 1, 2, ... k
        int index_in_array;     // 1, 2, ... n
        int value;              // heap
}
```

Solution 2: Binary reduction

A1
A2 → A12 (2n)
A3 → A14
A4 → A34 (2n)
A5 → A18
A6 → A56
A7 → A58
A8 → A78

kn 2n * (k/2) = kn kn kn .... log(k) columns
Time = O(nk * log(k))

Solution 3: Iterative

A1 A2 → A12 2n
A3 → A13 3n
A4 → A14 4n
A5 ...
A1k kn

Time = O(k^2 * n)

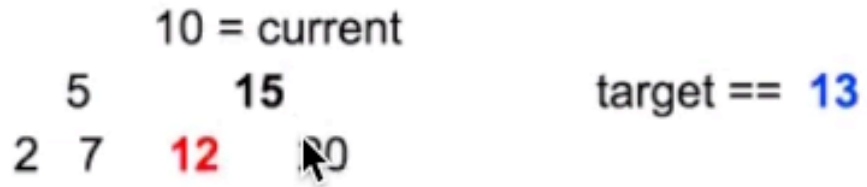
Discussion

Space: Space wise: solution 2 = O(nk) vs solution 1 = O(k)

Time: complexity is the same. However, if the array sizes are big, and we need to get file read and write involved. Solution1 only needs to read and write each element once and only once. Solution2 needs to read and write each element **log(k)** times which is the bottleneck of the system.

Q3.1: (Find a node whose value is closest to the target value)

Given a **BST**, how to find the node with its value **closest** to a target value x?



Solution:

Maintain a closest node.

Start from the root node as cur.

Case 1: If (cur.value == target) just return cur

Case 2: If (cur.value < target) calculate the diff, if the |diff| is smaller than the closest node, update it. Goto right

Case 3: If (cur.value > target) calculate the diff, if the |diff| is smaller than the closest node, update it. Goto left

Q3.2 Given a **BST**, how to **find** the largest element in the tree that is smaller than a target number x.

Solution:

Case 1: If (cur.value < target)

- Update the solution
- Goto right

Case 2: If (cur.value >= target)

- Do not update the solution
- Goto left

There is a wooden stick with length L >= 1, we need to cut it into pieces, where the cutting positions are defined in an int array A.

The positions are guaranteed to be in ascending order in the range of [1, L - 1].

The cost of each cut is the length of the stick segment being cut.

Determine the minimum total cost to cut the stick into the defined pieces.

Examples L = 10, A = {2, 4, 7}, the minimum total cost is 10 + 4 + 6 = 20 (cut at 4 first then cut at 2 and cut at 7)

Base case: The shortest wood piece that cannot be cut any further.

M[0][1] = 0

M[1][2] = 0

M[2][3] = 0

M[3][4] = 0

Induction rule:

M[i][j] represents the minimum cost of cutting the wood between index i and index j in the input array A. So, the **final solution** to return is the value of M[0][4].

Size = 1: adjacent index [left = i, right = i+1]

M[0][1] = 0
M[1][2] = 0
M[2][3] = 0
M[3][4] = 0

Size = 2: [left = i, right = i+2]

(1) M[0][2] = M[0][1] + M[1][2] + (A[2] - A[0]) = 4
左大段 右大段 cost of cutting 4

(2) M[1][3] = M[1][2] + M[2][3] + (A[3] - A[1]) = 5
左大段 右大段 cost of cutting 5

(3) M[2][4] = M[2][3] + M[3][4] + (A[4] - A[2])
左大段 右大段 cost of cutting 4

Size = 2: [left = i, right = i+2]

(1) M[0][2] = M[0][1] + M[1][2] + (A[2] - A[0]) = 4
左大段 右大段 cost of cutting 4

(2) M[1][3] = M[1][2] + M[2][3] + (A[3] - A[1]) = 5
左大段 右大段 cost of cutting 5

(3) M[2][4] = M[2][3] + M[3][4] + (A[4] - A[2]) = 6
左大段 右大段 cost of cutting 6

Size = 3: [left = i, right = i+3]

(1) M[0][3] =

Case 1: we cut at index 1
M[0][1] + M[1][3] + (A[3] - A[0]) = 12
左大段 右大段 cost of cutting 7

Case 2: we cut at index 2
M[0][2] + M[2][3] + (A[3] - A[0]) = 11
左大段 右大段 cost of cutting 7

M[0][3] = min(Case1, Case2) = min(12, 11) = 11

(2) M[1][4] =

Case 1: we cut at index 2
M[1][2] + M[2][4] + (A[4] - A[1]) = 14

Case 2: we cut at index 3
M[1][3] + M[3][4] + (A[4] - A[1]) = 13

M[1][4] = min(Case1, Case2) = min(14, 13) = 13

index	0	1	2	3	4
0		**x**	**0**	**4**	**11** **x**
1		**x**	**x**	**0**	**5** **13**
2		**x**	**x**	**x**	**0** **6**
3		**x**	**x**	**x**	**x** **0**
4		**x**	**x**	**x**	**x** **x**

index	0	1	2	3	4
0	**x**	**0**	**4**	**11**	**x**
1	**x**	**x**	**0**	**5**	**13**
2	**x**	**x**	**x**	**0**	**6**
3	**x**	**x**	**x**	**x**	**0**
4	**x**	**x**	**x**	**x**	**x**

Observation: In order to fill in a value at M[i][j] we must know the value of to its left, and its value below it, so we fill in the table from bottom up and from left to right.

Time = O(n^2 * n) = O(n^3)
Space = O(n^2)