1. Find Last Page
   We will use bfs to search each node until we get one with no children, which is what we want. We will use a visited to record all visited nodes to prevent duplication. TC is O(n)

```
from collections import defaultdict
def findLastPage(clicks, startPoint):
 children = defaultdict(list)
 visited = set()
 node = startPoint
 visited.add(node)
 for start, end in clicks:
   children[start].append(end)
 if not children[node]:
   return node
 cur = set([node])
 while cur:
   next_ite = set()
   for n in cur:
     if not children[n]:
       return n
     else:
       for child in children[n]:
         if child not in visited:
           next_ite.add(child)
           visited.add(child)
   cur = next_ite
 return False

print(findLastPage(arr, 'A'))
```

2. LRU Cache
   We will use OrderedDict to solve this question very easily. When we want to get an item from dict, we will delete it from dict and add it back. Then return value. If we want to set new key-value pair, we need to check the rest capacity is larger than 0, if not, we will popitem(last=True), or deduct rest by 1. In the end, we will set that key and value pair. TC is O(1)
   from collections import OrderedDict
   class LRUCache:

      def __init__(self, capacity: int):

```
        self.dict = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key in self.dict:
            val = self.dict[key]
            self.dict.pop(key)
            self.dict[key] = val
            return val
        else:
            return -1

    def put(self, key, value):
        if key in self.dict:
            self.dict.pop(key)
        else:
            if self.capacity > 0:
                self.capacity -= 1
            else:
                self.dict.popitem(last=False)
        self.dict[key] = value
```

341. Flatten Nested List Iterator
We will use dfs to flatten all nested list and store it in an array. But we need to reverse it after finishing appending operation. TC is O(n), O(1), O(1)

```
class NestedIterator(object):

    def __init__(self, nestedList):
        """
        Initialize your data structure here.
        :type nestedList: List[NestedInteger]
        """
        self.store = []
        def dfs(nestedList):
            for e in nestedList:
                if e.isInteger():
                    self.store.append(e.getInteger())
                else:
                    dfs(e.getList())

        dfs(nestedList)
        self.store.reverse()
```

```
    def next(self):
        """
        :rtype: int
        """
        return self.store.pop()

    def hasNext(self):
        """
        :rtype: bool
        """
        return len(self.store) > 0
```

347. Top K Frequent Elements
We will first count each element's frequence, then use minheap to maintain most frequent k freq-element pairs. Then in the end we will sort it and return what we want. TC is O(nlogk + klok + n) = O(n)

```
from collections import Counter
from heapq import heappush, heappop
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        memo = Counter(nums)
        result = []

        for key, value in memo.items():
            heappush(result, (value, key))
            if len(result) > k:
                heappop(result)

        return list(map(lambda a: a[1], sorted(result, reverse=True)))
```

64. Minimum Path Sum
We will use dynamic programming to compute accumulate all min previous path weight. Then the last element is the result. TC is O(mn)

```
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])
        for i in range(1, m):
            grid[i][0] += grid[i - 1][0]

        for j in range(1, n):
            grid[0][j] += grid[0][j - 1]
```

```
for i in range(1, m):
    for j in range(1, n):
        grid[i][j] += min(grid[i - 1][j], grid[i][j - 1])

return grid[-1][-1]
```