1. Load Balancer Design

A key component of the architecture for a high-traffic website is its load balancer, which distributes network or application traffic across many servers to ensure no single server is overwhelmed.

Which pair of pseudo-code snippets correctly implement a *Round Robin* and a *Least Connection* load balancing algorithm?

Pick **ONE OR MORE** options

- [ ] Array servers = getServerList() Function getNextServer() Server nextServer = servers[randomWeightedIndex()] return nextServer
  // Round Robin Array servers = getServerList() Integer index = 0 Function getNextServer() Server nextServer = servers[index] index = (index + 1) % servers.length return nextServer

- [ ] Array servers = getServerList() Function getServerWithLeastConnections() Server leastConnectionServer = servers[0] for each server in servers if server.connectionCount < leastConnectionServer.connectionCount leastConnectionServer = server return leastConnectionServer
  // Least Connection Array servers = getServerList() Function getServerWithLeastConnections() Server leastConnectionServer = servers[0] for each server in servers if server.connectionCount < leastConnectionServer.connectionCount leastConnectionServer = server return leastConnectionServer

- [ ] Array servers = getServerList() Integer index = 0 Function getNextServer() Server nextServer = servers[index] index = (index + 1) % servers.length return nextServer
  Array servers = getServerList() Array weights = getServerWeights() // Assuming each server has an associated weight Integer index = 0 Integer cumulativeWeight = sum(weights) Function getNextServer() while weights[index] <= 0 weights[index] += cumulativeWeight index = (index + 1) % servers.length weights[index] -= 1 return servers[index]
  // Round Robin Array servers = getServerList() Function getNextServer() Server nextServer = servers[0] for each server in servers if server.connectionCount < nextServer.connectionCount nextServer = server return nextServer

- [ ] Array servers = getServerList() Function getRandomServer() Server randomServer = servers[randomIndex()] return randomServer
  // Least Connection Array servers = getServerList() Integer index = 0 Function getServerWithLeastConnections() Server leastConnectionServer = servers[index] index = (index + 1) % servers.length return leastConnectionServer

Clear Selection

2. Graph Algorithm Implementation

A graph is built using the following class.

```
Class Graph {
   nodes
   edges
   // methods
   addNode(node)
   addEdge(node1, node2, cost)
}
```

The task is to extend this class with two popular shortest-path algorithms: *Dijkstra's Algorithm* and the *Bellman-Ford Algorithm*.

- *Dijkstra's Algorithm*: Finds the shortest paths from a start node to all other nodes in a graph with non-negative edge weights.

- *Bellman-Ford Algorithm*: Finds the shortest paths from a start node to all other nodes, even with negative edge weights (assuming no negative cycles).

Which pair of pseudo-code snippets extend this Graph class to implement these algorithms correctly?

Pick **ONE OR MORE** options

- [ ]
```
// Dijkstra's Algorithm
Class DijkstraGraph extends Graph {
   Function shortestPath(start, end) {
      // initialization
      set all node distances to infinity
      set start node distance to 0
      while there are unvisited nodes {
         select the unvisited node with the smallest distance
         for each neighbor of the current node {
            calculate tentative distance through the current node
```

```
// Dijkstra's Algorithm
Class DijkstraGraph extends Graph {
   Function shortestPath(start, end) {
      // initialization
      set all node distances to infinity
      set start node distance to 0
      while there are unvisited nodes {
         select the unvisited node with the smallest distance
         for each neighbor of the current node {
            calculate tentative distance through the current node
            if tentative distance < neighbor's current distance {
               update neighbor's distance
            }
         }
         mark current node as visited
      }
      return end node distance
   }
}
```

```
// Bellman-Ford Algorithm
Class BellmanFordGraph extends Graph {
   Function shortestPath(start, end) {
      // initialization
      set all node distances to infinity
      set start node distance to 0
      for each node in the graph {
         for each edge in the graph {
            if node distance + edge cost < edge's end node distance {
               update end node's distance to node distance + edge cost
            }
         }
      }
      return end node distance
   }
}
```

```
// Dijkstra's Algorithm
Class DijkstraGraph extends Graph {
   Function shortestPath(start, end) {
      // initialization
      set all node distances to infinity
      set start node distance to 0
      for each node in the graph {
         for each edge in the graph {
            if node distance + edge cost < edge's end node distance {
               update end node's distance to node distance + edge cost
            }
         }
      }
      return end node distance
   }
}
```

```
// Bellman-Ford Algorithm
Class BellmanFordGraph extends Graph {
   Function shortestPath(start, end) {
      // initialization
      set all node distances to infinity
      set start node distance to 0
      while there are unvisited nodes {
         select the unvisited node with the smallest distance
         for each neighbor of the current node {
            calculate tentative distance through the current node
            if tentative distance < neighbor's current distance {
               update neighbor's distance
            }
         }
         mark current node as visited
      }
      return end node distance
   }
}
```

Clear Selection

3. Spend it All

There is a *budget* and an array of *n costs*. Repeat the following process until *budget* is less than the minimum element in *cost*.

Process
Start at element *0* and work to *n - 1*.
At each element *cost[i]*:

- If *cost[i] <= budget*, reduce *budget* by *cost[i]* and move to the next index. This is a *purchase*.
- If *cost[i] > budget*, move to the next index.

The array is circular, so the next index after *n - 1* is index *0*.
Continue until there is not enough budget to make a *purchase*.

Determine how many purchases are made.

Example
cost = [5, 8, 3]
budget = 12

Process:

- Buy item 0 for 5 -> *budget = 12 - 5 = 7*.
- Item 1 is too expensive, *budget = 7*.
- Buy item 2 for 3, *budget = 7 - 3 = 4*.
- Items 0 and 1 are too expensive, *budget = 4*.
- Buy item 2 for 3, *budget = 4 - 3 = 1*.
- Now *budget = 1*, and there are no more items that can be *purchased*.
- Return 3, the number of items purchased.

Function Description
Complete the function *countPurchases* in the editor below.

countPurchases has the following parameters:
int cost[n]: the costs of all the items
long budget: the starting amount of the budget to be spent

Returns

Language Pypy 3 | Environment | Autocomple

```
#!/bin/python3 ⋯

#
# Complete the 'countPurchases' function below.
#
# The function is expected to return a LONG_INTEGER.
# The function accepts following parameters:
#  1. INTEGER_ARRAY cost
#  2. LONG_INTEGER budget
#

def countPurchases(cost, budget):
    # Write your code here

if __name__ == '__main__': ⋯
```

Test Results **Custom Input**

4. Subsequences of Three

Given an array of *n* integers, *arr[n]*, determine all of its subsequences *S* of three elements and find the validity of *arr*.
*validity = min(3 * abs(mean(S) - median(S)) for all S*
A subsequence is a sequence that can be derived from a sequence by deleting zero or more elements without changing the order of the remaining elements, for example [3, 4] is a subsequence of [5, 3, 2, 4].

Note
mean(A) is the average of the array (the sum of the array divided by the size of the array).
median(A) is the middle value of an ordered set of numbers with an odd number of elements.
abs(x) is the absolute value of an integer.

Example
n = 4
arr = [2, 3, 1, 4]

subsequence	mean	median	validity
[2, 3, 1]	(2 + 3 + 1) / 3 = 2	2	3 * |2 – 2| = 0
[2, 3, 4]	(2 + 3 + 4) / 3 = 3	3	3 * |3 – 3| = 0
[2, 1, 4]	(2 + 1 + 4) / 3 = 2.33	2	3 * |2.33 – 2| = 1
[3, 1, 4]	(3 + 1 + 4) / 3 = 2.67	3	3 * |2.67 – 3| = 1

The validity is equal to min([0, 0, 1, 1]) = 0.

Function Description
Complete the function *calculateValidity* in the editor below.

calculateValidity has the following parameters:
int arr[n]: the series of integers

Returns
int: the validity of the series of integers

Constraints

- $3 \le n \le 10^3$
- $1 \le arr[i] \le 10^9$
- *arr* contains distinct elements.

Language Pypy 3 | Autocomplete not supported
Environment

```
#!/bin/python3 ⋯

#
# Complete the 'calculateValidity' function below.
#
# The function is expected to return an INTEGER.
# The function accepts INTEGER_ARRAY arr as parameter.
#

def calculateValidity(arr):
    # Write your code here

if __name__ == '__main__': ⋯
```

Line: 10 Col

5. Maximum XOR Suffix

An array of *n* integers, *arr*, is given. Pick any index then calculate the XOR of the array from that index through the highest index. Append the value to the array. Repeat this process zero or more times. Determine the highest value possible.

Example
n = 5
arr = [8, 2, 4, 12, 1].
One way to achieve the maximum is shown. The ⊕ symbol indicates the bitwise XOR operation.

Operation (0-indexed)	Resulting Array
Choose i = 4 arr[5] = arr[4]	[8, 2, 4, 12, 1, 1]
Choose i = 1 arr[6] = arr[1] ⊕ arr[2] ⊕ ... ⊕ arr[5]	[8, 2, 4, 12, 1, 1, 10]
Choose i = 2 arr[7] = arr[2] ⊕ arr[3] ⊕ ... ⊕ arr[6]	[8, 2, 4, 12, 1, 1, 10, 6]
Choose i = 0. arr[8] = arr[0] ⊕ arr[1] ⊕... ⊕ arr[7]	[8, 2, 4, 12, 1, 1, 10, 6, 14]

The maximum strength possible is 14.

Function Description
Complete the function *maximumValue* in the editor below.

maximumValue has the following parameter:
int arr[n]: the starting array

Return
int: the maximum possible value in the array

Constraints

- $1 \le n \le 10^5$
- $0 \le arr[i] < 2^{30}$

▸ **Input Format for Custom Testing**

▾ **Sample Case 0**

Environment

```
#!/bin/python3 ⋯

#
# Complete the 'maximumValue' function below.
#
# The function is expected to return an INTEGER.
# The function accepts INTEGER_ARRAY arr as parameter.
#

def maximumValue(arr):
    # Write your code here

if __name__ == '__main__': ⋯
```

Test Results Custom Input Run Code Run Tests Submit