

|  |
| --- |
| IS103  Computational Thinking:  **Project 1 Report** |
|  |
| **March 5**  Singapore Management University  Authored by: Wei Minn (G5) |



# Question 1

|  |  |
| --- | --- |
| *import* random  def r6\_loaded():  arr = [1, 1, 1, 1, 1, 2, 3, 4, 5, 6]  randPos = random.randint(0, 9)  *return* arr[randPos] | Import random module |
| Populate the array with the number of 1’s (5 out of 10) proportional to achieve probability of 50% |
| Generate the random number with the range 0 to 9 |
| Access the array with the generated random number |

Big-O notation for algorithm: **O(log n)**.

Time complexity of random.randInt(0, x) is [O(log n)](https://stackoverflow.com/questions/29461787/what-is-big-o-runtime-of-the-standard-random-number-generator-in-python-worst), and Time complexity of accessing array index is O(1).

# Question 2a

The number of times a cell value can be multiplied to be added to the total value is derived by summing the number of function calls on the cells to the left and above it as shown on the right. The pattern is akin to [Pascal’s triangle](https://www.mathsisfun.com/pascals-triangle.html) which, similarly, is composed of numbers that are sum of numbers to the top-left and top-right.

|  |  |
| --- | --- |
| *from* collections *import* deque  def sv\_iterative(*m*):  total = 0  q = deque()  *for* i in range(len(m[0])-1):  q.append(1)  *for* i in range(len(m)):  last = 1  *for* j in range(len(m[i])):  *if* i == 0 or j == 0:  total += (m[i][j] \* 1)  *else*:  times = q.popleft() + last  total += (m[i][j] \* times)  q.append(times)  last = times  *return* total | Import deque that allows us to create a queue object |
| Declare total variable for storing values from the arrays |
| Declare Queue object, q |
| Populate q with 1’s which are supposed to multiply with the elements in the first array. |
| Initialize starting value, 1, in last variable which multiples with the first element for every array and will be incremented across the 2D-array. |
| Traverse the elements of the current array |
| If the element belongs to the first array or is the first element of its array, multiply it with 1 and add to the total |
| Dequeue q and sum it with the multiplier value from the last cell, multiple the sum with the value of the current cell, and add it to the total. Store the current multiple value in the last variable, to be used by the next cell |

Big-O notation for algorithm: **O(n)**

The total number of inner-loops is in linear relation to the total number of elements in the 2D-array.

# Question 2b

We need to create a global variable in the code, \_m, which is accessed by the function value(a, b). Otherwise, we need to pass in the array, m, as a parameter of value such that value(a, b, m)and this will cause a copy of m to be made with every recursive call and negatively affect the performance.

|  |  |
| --- | --- |
| \_m = []  def sv\_recursive(*m*):  global \_m  \_m = m[:]  *return* value(0, 0)  def value(*a*, *b*):  *if*(a >= len(\_m) or b >= len(\_m[0])):  *return* 0  *else*:  *return* \_m[a][b] + value(a+1, b) + value(a, b+1) | Declare array \_m outside of the functions as a global variable |
| Assign the array passed in the parameter into \_m |
| Return the value from the recursive function calls |
| If the index goes out of the bounds of either inner or outer array, return 0 as specified |
| Sum up the value of the cell with the values returned by the recursive calls on the cells to the right and below it |

Big-O notation for algorithm: **O(2n/n2)**

The number of operations increases exponentially with an increase in n referring the to the rate of increase for every row in Pascal’s triangle to which the pattern corresponds greatly.

# Question 3

|  |  |
| --- | --- |
| def ttsum(*nums*, *sum*):  table = dict()  arr = []  *for* i in range(0, len(nums)):  *if* nums[i] not in table:  table[nums[i]] = [i]  *else*:  table[nums[i]] += [i]  *for* i in range(0, len(nums)):  sub = sum - nums[i]  *if* sub in table:  *for* s in table[sub]:  *if* s > i:  arr.append([i, s])  *for* i in range(0, len(nums)):  sub = sum - nums[i]  *for* j in range(i+1, len(nums)):  sub2 = sub - nums[j]  *if* sub2 in table:  *for* s in table[sub2]:  *if* s > j:  arr.append([i, j, s])  *return* arr |  |
| Create dictionary to keep track of the position of the numbers |
| Create array to store all the combinations for the sum |
| Loop through the first time to record the position of the numbers in the table dictionary |
| Calculate for every number |
| The difference between it and the sum |
| Check if the difference exists in the table dictionary |
| Get all the positions s of that “difference” number, pair it with current position i and store it in arr array.  The position s should be greater than i to prevent overlaps. |
| Calculate for every number, |
| the difference sub between it and the sum. |
| From the next position (i+1) onwards |
| Calculate the difference, sub2, between the difference, sub, and the next number nums[j] |
| Check if the difference, sub2, exists in the table dictionary |
| Get all the positions s of that “difference” number, pair it with current position i and store it in arr array.  The position s should be greater than j to prevent overlaps. |

Big-O notation for algorithm: **O(n3)**

The worst case is **O(n3)** where the size of the array for the table dictionary entry approaches n. However, it is still more efficient than brute-forcing the combination with 3 nested loops.

# Appendix

## Alternative solutions

### Alternative Solution 1 for Question 1

def r6\_loaded():

*# your code here*

randPos = random.randint(1, 100)

*if* randPos > 50:

*return* 1

*elif* randPos > 40:

*return* 2

*elif* randPos > 30:

*return* 3

*elif* randPos > 20:

*return* 4

*elif* randPos > 10:

*return* 5

*else*:

*return* 6

### Alternative Solution 2 for Question 1

*import* random

*# TODO: fill r6\_loaded()*

def r6\_loaded():

*# your code here*

randPos = random.uniform(0, 1)

*if* randPos > .50:

*return* 1

*elif* randPos > .40:

*return* 2

*elif* randPos > .30:

*return* 3

*elif* randPos > .20:

*return* 4

*elif* randPos > .10:

*return* 5

*else*:

*return* 6

### Alternative Solution 3 for Question 1

Cython, a superset of Python, has a function in its random module namely getrandint32 which as time complexity of O(1) which is considerably more efficient than randInt in current Python version which as time complexity of O(log n)

### Alternative Solution for Question 3: The naïve way

Loop in a loop in a loop to brute-force all the possible combination for the sum.

def ttsum(*nums*, *t*):

*# your code here*

arr = []

*for* f in range(len(nums)):

*for* s in range(f, len(nums)):

*if* nums[f]+nums[s] == t:

*if* f != s:

arr.append([f,s])

*for* f in range(len(nums)):

*for* s in range(f, len(nums)):

*for* trd in range(s, len(nums)):

*if* nums[f]+nums[s]+nums[trd] == t:

*if* f != s != trd:

arr.append([f,s,trd])

*# print(arr)*

*return* arr