Foobar Challenge, completed on 3/06/2020, PC

1

Given a string replace the lower case letters with their alphabetical conjugates, i.e. a - z, b - y, and so on. The follwoing solution is O(n).

In [36]:
```python
def solution(s):
    dec = "" #for storing the deciphered string

    for i in range(len(s)):
        char = s[i]

        #check whether lower case by comparing ASCII values
        if "a" <= char and char <= "z":
            #replace original character with decoded character
            char = chr(219 - ord(char))

        dec += char #concatenate the decoded string to dec

    return dec

s = "Yvzs! I xzm'g yvorvev Lzmxv olhg srh qly zg gsv xlolmb!!"
solution(s)
```

Out[36]: "Yeah! I can't believe Lance lost his job at the colony!!"

2.1

Given a list of n elements find the sublist with maximum product. The abs(element) < 1000 Thr provided solution is O(n).

In [37]:
```python
from math import inf

def solution(xs):
    n = len(xs)

    if n == 1:
        return str(xs[0])  #in case there is just one panel

    max_neg, neg_c, zero_c = - 1001, 0, 0
    # stores the maximum negative power, the number of negative power
    # and the number of zero powers respectively

    prod = 1
    #for storing the product of non-zero panel powers

    for i in xs:
        if i == 0:
            zero_c += 1

        else:
            prod = prod * i

            if i < 0:
                neg_c += 1
                if i >= max_neg:
                    max_neg = i

    if (zero_c == n - 1 and neg_c == 1) or (zero_c == n):
        return "0"
        # corner cases

    elif neg_c % 2 == 0: return str(int(prod))
    # if there are even number of negative powers then the maximum
    # product comprises of all the non zero powers

    else: return str(int(prod / max_neg))
    # if there are an odd number of them, we take out from the product
    # of non-zero powers the negative power with least absolute value


L = [-2, -3, 4, -5]
solution(L)
```

Out[37]: '60'

2.2

Given a list of n digits from 0 - 9 find the largest number divisible by 3 that can be made out of elements of the list. The provided solution is O(n^2).

In [38]:
```python
def solution(L):
    n = len(L)
    L = sorted(L)
    # sort the list as the largest number would be beginning with the
    # largest digit of the subarray whose sum is divisible by 3

    D = [x % 3 for x in L] # stores the numbers reduced modulo 3

    T = [ [0 for m in range(3)] for i in range(n) ]
    pi = [ [(None, None) for m in range(3)] for i in range(n) ]
    # define T[i][m] as the longest subsequence ending in L[i] congruent to
    # m mod 3, and pi[i][m] stores the parent of i in that subsequence

    def relax(i, j, m, k):
        if T[j][k] != 0:
            if (1 + T[j][k]) >= T[i][m]:
                T[i][m] = 1 + T[j][k]
                pi[i][m] = (j, k)

    for i in range(n):
        T[i][D[i]] = 1
        # implements the base cases

        for m in range(3):
            k = (m - D[i]) % 3
            for j in range(i):
                relax(i, j, m, k)

    # relax implements the following recurrence
    # T[i][m] =  max_{j < i} {1 + T[j][k]}, where k + L[i] = m mod 3
    # if j particularly maximises the above recurrence then paren(i, m) = (j, k)

    maxloc = 0 # stores the starting location of the largest possible digit
    for i in range(n):
        if T[i][0] >= T[maxloc][0]: maxloc = i

    answer = 0
    loc = (maxloc, 0)
    # this variable would recursively call the parents one by one to construct the largest n
umber

    if T[maxloc][0] != 0:
    # eliminating the case when no such number exists
        while loc != (None, None):
            answer = answer * 10 + L[loc[0]]
            loc = pi[loc[0]][loc[1]]

    return answer

L = [7, 4, 1, 4, 6, 8, 0, 8]
solution(L)
```

Out[38]: 8764410

3.1

If the first worker has ID 17 (start) and the checkpoint holds four (length) workers, the process would look like: 17 18 19 20 / 21 22 23 / 24 25 26 / 27 28 29 / 30 31 32 which produces the checksum 17^18^19^20^21^22^23^25^26^29 == 14.

All worker IDs (including the first worker) are between 0 and 2000000000 inclusive, and the checkpoint line will always be at least 1 worker long.

In [39]:
```python
def solution(start, length):

    # calculate the XOR in the range [1, n] and observe that the value has period 4
    def xortill(n):
        if n % 4 == 1: return 1
        elif n % 4 == 2: return (n + 1)
        elif n % 4 == 3: return 0
        else: return n

    answer = 0

    for j in range(length):
        # XOR[x, y] = XOR[1, x - 1] ^ XOR[1, y]
        # Calculate the boundaries for each line
        a = xortill( start + (j * length) - 1 )
        b = xortill( start + ((j + 1) * length) - (j + 1) )
        answer = answer ^ (a ^ b)

    return answer

solution(17, 4)
```

Out[39]: 14

3.2

And finally, you were only able to smuggle one of each type of bomb - one Mach, one Facula - aboard the ship when you arrived, so that's all you have to start with. (Thus it may be impossible to deploy the bombs to destroy the LAMBCHOP, but that's not going to stop you from trying!)

You need to know how many replication cycles (generations) it will take to generate the correct amount of bombs to destroy the LAMBCHOP. Write a function solution(M, F) where M and F are the number of Mach and Facula bombs needed. Return the fewest number of generations (as a string) that need to pass before you'll have the exact number of bombs necessary to destroy the LAMBCHOP, or the string "impossible" if this can't be done! M and F will be string representations of positive integers no larger than 10^50. For example, if M = "2" and F = "1", one generation would need to pass, so the solution would be "1". However, if M = "2" and F = "4", it would not be possible

In [40]:
```python
def solution(x, y):
    # the problem is equivalent to finding the number of steps
    # required by the euclidean algorithm to calculate the gcd
    # of x and y. We can reach 1 iff 1 = gcd(x, y)

    def gcd(x, y, count):
        if y == 0:
            return (x, count - 1)
            # count - 1 as we are considering the step till reaching (a, a) and not (a, 0)

        else:
            q, r = divmod(x, y)
            count += q
            # if q is the quotient we'll need to do (x - y) q times until x becomes less tha
n q
            return gcd(y, r, count)

    gcd, count = gcd(int(x), int(y), 0)

    if gcd == 1:
        return str(count)

    else: return "impossible"
    # if gcd(x, y) != 1 then no linear combination yields the (1, 1) state

solution(65, 85)
```

Out[40]: 'impossible'

3.3

Count the number of partitions of an integer into distinct parts

In [41]:
```python
def solution(m):
    # initialize memo table T with 0s
    T = [[0 for i in range(m + 1)] for j in range(m + 1)]
    T[1][1] = 1 # base case

    # define T(n, i): number of distinct partitions of n
    # such that no term is smaller than i

    # Recurrence used,
    # T(n, i) = 1 + sum T(n - j, j + 1) over i <= j <= floor {(n - 1)/2}

    for n in range(2, m + 1):
        for i in range(1, m + 1):
            result = 1
            lim = int((n - 1)/2)

            for j in range(i, lim + 1):
                result += T[n - j][j + 1]

            T[n][i] = result

            if (n, i) == (m, 1): break

    # Clearly the solution is T(m, 1), however we substract 1 as the number
    # itself is not asked (in the specific question) to be considered as a partition

    return(T[m][1] - 1)

solution(200)
```

Out[41]: 487067745

4.1

You and your rescued bunny prisoners need to get out of this collapsing death trap of a space station - and fast! Unfortunately, some of the bunnies have been weakened by their long imprisonment and can't run very fast. Their friends are trying to help them, but this escape would go a lot faster if you also pitched in. The defensive bulkhead doors have begun to close, and if you don't make it through in time, you'll be trapped! You need to grab as many bunnies as you can and get through the bulkheads before they close.

The time it takes to move from your starting point to all of the bunnies and to the bulkhead will be given to you in a square matrix of integers. Each row will tell you the time it takes to get to the start, first bunny, second bunny, ..., last bunny, and the bulkhead in that order. The order of the rows follows the same pattern (start, each bunny, bulkhead). The bunnies can jump into your arms, so picking them up is instantaneous, and arriving at the bulkhead at the same time as it seals still allows for a successful, if dramatic, escape. (Don't worry, any bunnies you don't pick up will be able to escape with you since they no longer have to carry the ones you did pick up.) You can revisit different spots if you wish, and moving to the bulkhead doesn't mean you have to immediately leave - you can move to and from the bulkhead to pick up additional bunnies if time permits.

In addition to spending time traveling between bunnies, some paths interact with the space station's security checkpoints and add time back to the clock. Adding time to the clock will delay the closing of the bulkhead doors, and if the time goes back up to 0 or a positive number after the doors have already closed, it triggers the bulkhead to reopen. Therefore, it might be possible to walk in a circle and keep gaining time: that is, each time a path is traversed, the same amount of time is used or added.

Write a function of the form solution(times, time_limit) to calculate the most bunnies you can pick up and which bunnies they are, while still escaping through the bulkhead before the doors close for good. If there are multiple sets of bunnies of the same size, return the set of bunnies with the lowest prisoner IDs (as indexes) in sorted order. The bunnies are represented as a sorted list by prisoner ID, with the first bunny being 0. There are at most 5 bunnies, and time_limit is a non-negative integer that is at most 999.

For instance, in the case of [[0, 2, 2, 2, -1], # 0 = Start [9, 0, 2, 2, -1], # 1 = Bunny 0 [9, 3, 0, 2, -1], # 2 = Bunny 1 [9, 3, 2, 0, -1], # 3 = Bunny 2 [9, 3, 2, 2, 0], # 4 = Bulkhead] and a time limit of 1, the five inner array rows designate the starting point, bunny 0, bunny 1, bunny 2, and the bulkhead door exit respectively. You could take the path:

Start End Delta Time Status

```
  -   0    -    1 Bulkhead initially open
  0   4   -1    2
  4   2    2    0
  2   4   -1    1
  4   3    2   -1 Bulkhead closes
  3   4   -1    0 Bulkhead reopens; you and the bunnies exit
```

With this solution, you would pick up bunnies 1 and 2. This is the best combination for this space station hallway, so the answer is [1, 2].

In [42]:
```python
# Brute force solution

from itertools import permutations

def solution(times, time_limit):
    N = len(times)
    bunnies = [i for i in range(1, N - 1)]

    # All pair shortest paths using Floyd-Warshall

    def floyd(W):
        # deepcopy, initializing the base cases
        delta = [[W[u][v] for v in range(N)] for u in range(N)]

        for k in range(N):
            for u in range(N):
                for v in range(N):
                    if delta[u][v] > delta[u][k] + delta[k][v]:
                        delta[u][v] = delta[u][k] + delta[k][v]

        # detect negative cycles
        for u in range(N):
            if delta[u][u] < 0:
                return False

        return delta

    # Given an order (P) of traversal of a subset of bunnies
    # calculate the time time required to escape with them
    # using the shortest distance matrix delta

    def calc_path_weight(P):
        s, d = P[0], P[-1]
        weight = delta[0][s] + delta[d][N - 1]
        for i in range(len(P) - 1):
            weight += delta[P[i]][P[i + 1]]

        return weight

    delta = floyd(times)

    if not delta:
        # in case there is a negative cycle it
        # is possible to escape with all the bunnies.
        return [i - 1 for i in bunnies]

    # check for all permutations lexicographically
    # in descending order of size

    for k in range(N - 2, 0, -1):
        for P in permutations(bunnies, k):
            if calc_path_weight(P) <= time_limit:
                return [i - 1 for i in sorted(P)]

    # if no non empty subsets of bunnies returned
    # previously, then return the empty set
    return []

print(solution([[0, 1, 1, 1, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 0, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 1, 1, 1, 0]], 3))
```

```
[0, 1]
```

In case the upper limit (5) to the number of bunnies didn't exist, the above brute force solution would clearly become exponential, as it involves calculating all permutations of all lengths of the set of bunnies. The following recursive approach can possibly be used in such cases. The approach works for most cases except when there is a zero-cycle in the graph. I will later provide an algorithm that will detect a zero cycle and subsequently solve the problem at hand, but I am too lazy to code it.

In [43]:
```python
def solution(times, time_limit):
    N = len(times)

    # Bellman-Ford to calculate single source shortest paths, detect
    # negative cycles, and transform all weights into non-negative
    # weights using Johnson's method

    def bellmanford(times, s):
        delta = [inf for i in range(N)]
        delta[s] = 0

        for i in range(N - 1):
            for u in range(N):
                for v in range(N):
                    delta[v] = min(delta[v], delta[u] + times[u][v])

        for u in range(N):
            for v in range(N):
                if delta[v] > delta[u] + times[u][v]:
                    return False

        return delta

    # Recurrence, T(v, t): maximum set of vertices that can be spanned by any
    # source (0) to "v" walk that has total weight smaller than t
    # Relate, T(v, t) = {v}  union  { largest over u in Adj[v] [T(u, t - W(u, v))] }

    def T(v, t):
        # base cases, T(0, 0) = [0]
        # T(v, t) = [], if t < _delta[v], minimum distance from 0 to

        if (v, t) == (0, 0): return [0]
        elif t < _delta[v]: return []

        else:
            M = []
            for u in range(N):
                if u != v:
                    S = T(u, t - _times[u][v])
                    if len(S) > len(M):
                        M = S

            M.append(v)
            return list(set(M))

    # any random source other than 0 can be used if 0 is used then all source destination
    # shortest paths after the transformation have 0 weight, which is undesirable
    delta = bellmanford(times, 2)

    if not delta:
        # in case there is a negative cycle it
        # is possible to escape with all the bunnies.
        return [i - 1 for i in range(1, N - 1)]

    # Otherwise transform the graph using Johnson's algorithm to contain non-negative
    # weights only, where h(u) = delta(s, u). Update the new quantities accordingly.
    _times = [[(times[i][j] + delta[i] - delta[j]) for j in range(N)] for i in range(N)]
    _delta = bellmanford(_times, 0)
    _time_limit = time_limit + delta[0] - delta[N - 1]

    # Call the recursion
    T = T(N - 1, _time_limit)
    return [T[i] - 1 for i in range(1, len(T) - 1)]

print(solution([[0, 1, 1, 1, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 0, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 1, 1, 1, 0]], 3))
```

```
[0, 1]
```

Why get rid of negative weights?

The recurrence used is

$$T(v,t) = \{v\} \bigcup \max_{u \in Adj[v]} \{T(u, t - w_{uv})\}$$

, with the mentioned bases cases. In the recurrence for t to hit the base case it must eventually decrease to either 0 or $delta[v]$ for some v. However with negative weights it's possible that the t keeps on increasing with successive recursion calls (or get stuck in a loop of values), thereby causing a recursion overflow.

After the transformation with Johnson's method we have a directed graph with only *non-negative* edge weights, and we can work on this graph from here on.

What's up with zero cycles in the transformed graph?

After the previous transformation is completed, the zero edges do not bother us much. We have ensured that t doesn't increase anymore, and if not all edges have 0 weight, t would eventually decrease. What creates a difficulty here is not the zero edges but the zero cyles. For a plain example say $(u - v)$ is a zero cycle, meaning both $w_{uv} = w_{vu} = 0$. From our recurrence, we need $T(u, t)$ to calculate $T(v, t)$, and $T(v, t)$ to calculate $T(u, t)$; and there we are stuck in a loop. So we must ensure that the graph does not contain zero cycles at all.

Approach 1 (failed) : Use Johnson's to transform all the non-negative weights to strictly positive weights without disturbing the shortest paths

Say there exists a mapping $f : V \to \mathbb{R}$, such that $\bar{w}_{uv} = w_{uv} + f(u) - f(v) > 0$, $\forall$ u and v, with a strict greater than unlike the usual Johnson's method; where $\bar{w}$ represents the weights after transformation. We will show that such a mapping is not always possible and will construct a example countering the approach.

fix a u, we must have $f(u) + w_{uv} > f(v)$, $\forall\, v$

for a particular $v \neq u$, we must also have $f(v) + w_{vu} > f(u)$

Combining the above equations we have, $f(v) - w_{uv} < f(u) < f(v) + w_{vu}$

Choose $w_{uv} = w_{vu} = 0$ (the choice is valid as non-negative weights are allowed),

then, $-w_{uv} < f(u) - f(v) < w_{uv} \implies |f(u) - f(v)| < w_{uv} = 0$, which is a contradiction! Or zero cycles of length 2 would not allow the application of Johnson's algorithm to tranform non-negative edge weights of a graph to strictly positive weights.

Approach 2 (not coded) : Shrink all the zero cycles to a single vertex and work on the transformed graph

A brief description of the algorithm would be as follows.

- *identify the zero cycles*. How? Fix a source s. And remember the edge weights are non-negative now. If $(v_1, \cdots, v_k)$ is a zero cycle, i.e. $w_{v_i v_{i+1}} = 0, \forall\, i$, then the shortest distance $\delta(s, v_i) = \delta(s, v_j)$ for all $1 \leq i, j \leq k$, or all the veritces of a zero cyle would be equidistant from any source. Therefore our search space is strictly reduced now. We only need to run a single source shortest path (say dijkstra) from any random source, club the vertices which have equal distances, and search for a zero cycle inside them.
- Say $(uv \cdots z)$ is one such group of k equidistant vertices. Here, in the subgraph of these k vertices we can either run a DFS to obtain all cycles, and check for zero-sums, or we can obtain the $(k-1)!$ circular permutatioins and check for consecutive zero weights. The value of k would determine what is more efficient.
- Once such a cycle is detected, shrink the cycle into a single vertex, and for a vertex outside the cycle, the edge weight is updated to its shortest distance from the cycle. One particular way to do this would be the following. If $(uv \cdots z)$ is the zero cycle with u as the smallest vertex (assuming vertices are indexed), then in the adjacency matrix treat u as the entire cycle. For any vertex q outside the cycle, update w_{uq} and w_{qu} as the shortest distance of q from and to the cycle. In other words, for example, $w_{uq} = \min_{v \in cycle} w_{vq}$. Now pretend like the other vertices in the cycle aren't there. Or treat $w_{vq} = \infty$ for all q and for $v \in cycle$ but $v \neq u$. Now the transformed graph doesn't contain zero cycles anymore.
- From here we are good to go, there's nothing stopping t from reducinig, no increasing, no getting stuck in loops, t must reduce, and eventually hit the base cases. Now run the recursion to obtain the set $T(n-1, time\_limit)$. If vertex u is present in the set, expand it to include all the vertices of the cycles. And we're done!

4.2

You've blown up the LAMBCHOP doomsday device and broken the bunnies out of Lambda's prison - and now you need to escape from the space station as quickly and as orderly as possible! The bunnies have all gathered in various locations throughout the station, and need to make their way towards the seemingly endless amount of escape pods positioned in other parts of the station. You need to get the numerous bunnies through the various rooms to the escape pods. Unfortunately, the corridors between the rooms can only fit so many bunnies at a time. What's more, many of the corridors were resized to accommodate the LAMBCHOP, so they vary in how many bunnies can move through them at a time.

Given the starting room numbers of the groups of bunnies, the room numbers of the escape pods, and how many bunnies can fit through at a time in each direction of every corridor in between, figure out how many bunnies can safely make it to the escape pods at a time at peak.

Write a function solution(entrances, exits, path) that takes an array of integers denoting where the groups of gathered bunnies are, an array of integers denoting where the escape pods are located, and an array of an array of integers of the corridors, returning the total number of bunnies that can get through at each time step as an int. The entrances and exits are disjoint and thus will never overlap. The path element path[A][B] = C describes that the corridor going from A to B can fit C bunnies at each time step. There are at most 50 rooms connected by the corridors and at most 2000000 bunnies that will fit at a time.

For example, if you have: entrances = [0, 1] exits = [4, 5] path = [[0, 0, 4, 6, 0, 0], # Room 0: Bunnies [0, 0, 5, 2, 0, 0], # Room 1: Bunnies [0, 0, 0, 0, 4, 4], # Room 2: Intermediate room [0, 0, 0, 0, 6, 6], # Room 3: Intermediate room [0, 0, 0, 0, 0, 0], # Room 4: Escape pods [0, 0, 0, 0, 0, 0], # Room 5: Escape pods]

Then in each time step, the following might happen: 0 sends 4/4 bunnies to 2 and 6/6 bunnies to 3 1 sends 4/5 bunnies to 2 and 2/2 bunnies to 3 2 sends 4/4 bunnies to 4 and 4/4 bunnies to 5 3 sends 4/6 bunnies to 4 and 4/6 bunnies to 5

So, in total, 16 bunnies could make it to the escape pods at 4 and 5 at each time step. (Note that in this example, room 3 could have sent any variation of 8 bunnies to 4 and 5, such as 2/6 and 6/6, but the final solution remains the same.)

In [44]:
```python
# find maximum flow subject to limited capacity from multiple sources to multiple sinks

def solution(sources, sinks, capacity):
    N = len(capacity)
    source, sink = sources[0], sinks[0]
    # shrink the sources to a single vertex, source[0] would represent all the sources
    # similarly for sinks

    # transform the graph capacities according to the updated source and sinks
    def transform():

        # The row (columns) corresponding to the new source (sink) is the sum of all rows
        # (columns) corresponding to all the sources (sinks)
        for i in range(N):
            capacity[i][source] = sum([capacity[i][s] for s in sources])
            capacity[source][i] = sum([capacity[s][i] for s in sources])
            capacity[i][sink] = sum([capacity[i][s] for s in sinks])
            capacity[sink][i] = sum([capacity[s][i] for s in sinks])

        # after the updates are done the remaining source vertices are not required anymore
        # all corresponding capacities are therefore updated to 0
        for i in range(1, len(sources)):
            s = sources[i]
            for j in range(N):
                capacity[s][j], capacity[j][s] = 0, 0

        # Similarly for sinks
        for i in range(1, len(sinks)):
            s = sinks[i]
            for j in range(N):
                capacity[s][j], capacity[j][s] = 0, 0

    # Breadth fast search for source-sink paths with positive flow
    def BFS(residue):
        # initialize single source with Queue
        visited, parent = [False for i in range(N)], [None for i in range(N)]
        visited[source] = True
        Q = [source]
        flag = False # to denote whether the sink is reached

        # continue as long as Q is non empty and sink is not reached
        while Q != [] and not flag:
            u = Q.pop()
            for v in range(N):
                # visit those un-visited vertices from which flow is possible
                if not visited[v] and residue[u][v] > 0:
                    Q.insert(0, v)
                    visited[v], parent[v] = True, u
                    if v == sink:
                        flag = True # sink is reached
                        break

        return parent if flag else False

    # Ford Fulkerson Edmonds Karp Algorithm
    def maxflow():
        residue = [c for c in capacity] # initial residue
        paths = [] # store all paths here
        maxflow = 0 # initial flow

        # keep finding source sink paths and add their flow to the maxflow
        while True:
            parent = BFS(residue)
            # if no source sinks paths found
            if not parent: break

            # construct path using parent list
            path, v = [source], sink
            while v != source:
                path.insert(1, v)
                u = parent[v]
                v = u

            # the flow of a path is the the minimum flow of all edges
            flow =  min( [residue[path[i]][path[i + 1]] for i in range(len(path) - 1)] )

            # update the residues after the flow of the current path is determined
            for i in range(len(path) - 1):
                u, v = path[i], path[i + 1]
                residue[u][v] -= flow
                residue[v][u] += flow

            paths.append(path)
            maxflow += flow

        return maxflow


    transform()
    return maxflow()


solution([0, 1], [4, 5], [[0, 0, 4, 6, 0, 0],
                          [0, 0, 5, 2, 0, 0],
                          [0, 0, 0, 0, 4, 4],
                          [0, 0, 0, 0, 6, 6],
                          [0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0]])
```

Out[44]: 16

5

Oh no! You've managed to escape Commander Lambdas collapsing space station in an escape pod with the rescued bunny prisoners - but Commander Lambda isnt about to let you get away that easily. She's sent her elite fighter pilot squadron after you - and they've opened fire!

Fortunately, you know something important about the ships trying to shoot you down. Back when you were still Commander Lambdas assistant, she asked you to help program the aiming mechanisms for the starfighters. They undergo rigorous testing procedures, but you were still able to slip in a subtle bug. The software works as a time step simulation: if it is tracking a target that is accelerating away at 45 degrees, the software will consider the targets acceleration to be equal to the square root of 2, adding the calculated result to the targets end velocity at each timestep. However, thanks to your bug, instead of storing the result with proper precision, it will be truncated to an integer before adding the new velocity to your current position. This means that instead of having your correct position, the targeting software will erringly report your position as sum(i=1..n, floor(i*sqrt(2))) - not far enough off to fail Commander Lambdas testing, but enough that it might just save your life.

If you can quickly calculate the target of the starfighters' laser beams to know how far off they'll be, you can trick them into shooting an asteroid, releasing dust, and concealing the rest of your escape. Write a function solution(str_n) which, given the string representation of an integer n, returns the sum of (floor(1*sqrt(2)) + floor(2*sqrt(2)) + ... + floor(n*sqrt(2))) as a string. That is, for every number i in the range 1 to n, it adds up all of the integer portions of i*sqrt(2).

For example, if str_n was "5", the solution would be calculated as floor(1*sqrt(2)) + floor(2*sqrt(2)) + floor(3*sqrt(2)) + floor(4*sqrt(2)) + floor(5*sqrt(2)) = 1+2+4+5+7 = 19 so the function would return "19".

str_n will be a positive integer between 1 and 10^100, inclusive. Since n can be very large (up to 101 digits!), using just sqrt(2) and a loop won't work. Sometimes, it's easier to take a step back and concentrate not on what you have in front of you, but on what you don't.

TLDR

In other words, for an interger input $n, 1 \leq n \leq 10^{100}$ find, $\sum_{k=1}^{n} \lfloor k \cdot \sqrt{2} \rfloor$. Clearly, n can be tremendously large, hence using just a loop won't work. Refer: Beatty Sequences, Rayleigh Theoerem (the subtle hint at the last line of the problem statement)

In [48]:
```python
from math import sqrt, floor, log
def solution(n):
    n = int(n)
    # define a = sqrt(2) - 1 = r/t, use 100 digits for the required precision
    r = 414213562373095048801688724209698078569671875376948073176679737990732478462107038850
387534327641572735013846230912297024924836055850737212644121497099935831413222665927505592755
79995050115278206
    t = 10000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000

    # Use the following recurrence. Define m = floor(a * n)
    # Let S(n) be the required beatty sum up to n terms, then S(0) = 0
    # S(n) = mn + n(n+1)/2 - m(m+1)/2 + S(m)
    # The is obtained using Rayleigh Theorem

    def S(n):
        if n == 0: return 0
        else:
            m = int((r * n)/t)
            M = int(m * (m + 1) / 2)
            N = int(n * (n + 1) / 2)
            return m * n + N - M - S(m)

    return(str(S(n)))

solution(10**100)
```

Out[48]: '70710678118654751219058373283591209266417825186537355376805400360939302026678459186858883087
80782938311983929965271337362334692939138900065161829412770294060659694449882559980436725671150
39630963996899'

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: