Don't Get Volunteered!

Name: ...
Neptun ID: ...
Email: ...

In [2]:

```
%%html

<style>
    .chessboard { border-spacing: 0; border-collapse: collapse;}
    .chessboard td { border: 0px; width: 2em; height: 2em; text-align: center; font-size: 18px;}
    .chessboard .light { background: white; color: black;}
    .chessboard .dark { background: black; color: white;}
    .chessboard .target {background: green;}
    p {text-align: center;}
</style>
```

Problem

As a henchman on Commander Lambda's space station, you're expected to be resourceful, smart, and a quick thinker. It's not easy building a doomsday device and capturing bunnies at the same time, after all! In order to make sure that everyone working for her is sufficiently quick-witted, Commander Lambda has installed new flooring outside the henchman dormitories. It looks like a chessboard, and every morning and evening you have to solve a new movement puzzle in order to cross the floor. That would be fine if you got to be the rook or the queen, but instead, you have to be the knight. Worse, if you take too much time solving the puzzle, you get "volunteered" as a test subject for the LAMBCHOP doomsday device!

To help yourself get to and from your bunk every day, write a function called answer(src, dest) which takes in two parameters: the source square, on which you start, and the destination square, which is where you need to land to solve the puzzle. The function should return an integer representing the smallest number of moves it will take for you to travel from the source square to the destination square using a chess knight's moves (that is, two squares in any direction immediately followed by one square perpendicular to that direction, or vice versa, in an "L" shape). Both the source and destination squares will be an integer between 0 and 63, inclusive, and are numbered like the example chessboard below:

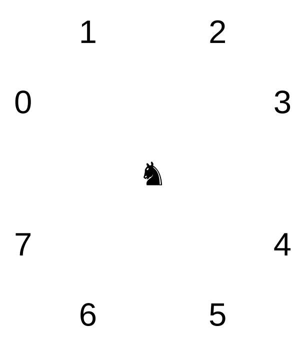
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Coordinate System

In this coordinate system moving up means -8, down +8, left -1, and right is +1. However, we need to consider the boundaries, for examples, if you are between 0 and 7, you cannot go up.

Define moves

There are 8 possible moves for a knight in chess, those moves are shown below.



In this coordinate system, it means

- Move 0: - 1 - 1 - 8 = -10
- Move 1: - 8 - 8 - 1 = -17
- Move 2: - 8 - 8 + 1 = -15
- Move 3: + 1 + 1 - 8 = -6
- Move 4: + 8 + 1 + 1 = 10
- Move 5: + 8 + 8 + 1 = 17
- Move 6: + 8 + 8 - 1 = 15
- Move 7: + 8 - 1 - 1 = 6

Method 1

An initial approach would be to add all the possible moves to the source, check if you are at the destinations, if not, add all the possible moves again to the list of sources and check again. Redo until you reach the final destination. In this case, one step has 8 possible solutions, and the second step has 8^2 because each of the 8 previous solutions has 8 new solutions. The memory allocation in this solution grows as 8^n which is really fast; the size of an integer in 64-bit version in Python is 27 bytes, so we can see in the next table the number of integers and the total memory size of it.

n	8^n	**size**
1	8	216 b
2	64	1.69 kb
3	512	13.5 kb
4	4096	108 kb
5	32768	864 kb
6	262144	6.75 Mb

Method 2

To improve this initial approach, we can add the boundary move restrictions. Therefore, the knight cannot move if it is in

- Move 0: [0, 7] and [8*n, 8*n+1]
- Move 1: [0, 15] and [8*n]
- Move 2: [0, 15] and [8*n+7]
- Move 3: [0, 7] and [8*n+6, 8*n+7]
- Move 4: [56, 63] and [8*n+6, 8*n+7]
- Move 5: [48, 63] and [8*n+7]
- Move 6: [48, 63] and [8*n]
- Move 7: [56, 63] and [8*n, 8*n+1]

where n = {0, ... , 7}

Conversely and in more coding language, we have that

- Move 0: `src > 7 and src\%8 > 1`
- Move 1: `src > 15 and src\%8 != 0`
- Move 2: `src > 15 and src\%8 != 7`
- Move 3: `src > 7 and src\%8 \< 6`
- Move 4: `src \< 56 and src\%8 \< 6`
- Move 5: `src \< 48 and src\%8 != 7`
- Move 6: `src \< 48 and src\%8 != 0`
- Move 7: `src \< 56 and src\%8 > 1`

Now the base changes based on the current position, the worst case is 8 when the knight is away from the bourders, and the best case is 2 is when the knight is at one of the corners (0, 7, 56, 63). We can see the maximum number of moves in each starting position in the next figure

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

The average number of moves is 5.25, using that as the base we can see the improvement in the next table

n	8^n	**size (8^n)**	5.25^n	**size (5.25^n)**
1	8	216 b	5.25	141.75 b
2	64	1.69 kb	27.56	744.19 b
3	512	13.5 kb	144.70	3.81 kb
4	4096	108 kb	759.69	20.03 kb
5	32768	864 kb	3988.38	105.16 kb
6	262144	6.75 Mb	20938.99	552.10 kb

It also means that it is computationally faster, since it fewer cases to check.

Method 3

It is still possible to improve because you can reach the same position by two different paths. For example, starting at 42, we have that

$$42 \to [25, 27, 32, 36, 48, 52, 57, 59].$$

Adding one more step

$$\begin{aligned}25 &\to [8, 10, 19, 35, 40, 42]\\27 &\to [10, 12, 17, 21, 33, 37, 42, 44]\\32 &\to [17, 26, 42, 49]\\36 &\to [19, 21, 26, 30, 42, 46, 51, 53]\\48 &\to [33, 42, 58]\\52 &\to [35, 37, 42, 46, 58, 62]\\57 &\to [40, 42, 51]\\59 &\to [42, 44, 49, 53].\end{aligned}$$

The previous method would save all possibilities for the next step as

$$[8, 10, 19, 35, 40, 42, 10, 12, 17, 21, 33, 37, 42, 44, 17, 26, 42, 49, 19, 21, 26, 30, 42, 46, 51, 53, 33, 42, 58, 35, 37, 44, 49, 53],$$

sorting this list we have

$$[8, 10, 10, 12, 17, 17, 19, 19, 21, 21, 26, 26, 30, 33, 33, 35, 35, 37, 37, 40, 40, 42, 42, 42, 42, 42, 42, 42, 42, 44, 44, 58, 58, 62],$$

now it is easy to see how it computes the same thing a couple of times. To fix that we can get just the unique numbers like the following list

$$[8, 10, 12, 17, 19, 21, 26, 30, 33, 35, 37, 40, 42, 44, 46, 49, 51, 53, 58, 62],$$

then we drop from 38 possible positions to 20. In general, this new method sets the maximum number of nodes to 32. This is because we can reach half of the board if you have the other half as possible initial position since the knight always moves from dark to light or light to dark square.

To have some analytical way to look at it, let's look at the following table

n	**min(2^n, 32)**	**min(3^n, 32)**	**min(4^n, 32)**	**min(6^n, 32)**	**min(8^n, 32)**
1	2	3	4	6	8
2	4	9	16	32	32
3	8	27	32	**32**	**32**
4	16	32	**32**	32	32
5	32	**32**	32	32	32
6	**32**	32	32	32	32

This means that the maximum number of steps need to reach the destination is 6 for just 2 moves, 5 for 3 moves, 4 for 4 moves, and 3 for 6 and 8 moves. There are 4 locations for 2 moves, 8 for 3 moves, 20 for 4 moves, 16 for 6 and 8 moves, you can check it in the previous figure. Making a weighted average we have that the average number of steps is 3.75. Therefore, the average maximum number of nodes is $5.25^{3.75} = 501.88$

It is hard to precisely express this last method mathematically, the average base still a good approximation, but the number of nodes is decreased after the average limit calculated in the previous paragraph. Then one possible description would be $\min(5.25^n, 501.88)$.

n	8^n	**size (8^n)**	5.25^n	**size (5.25^n)**	**min(5.25^n, 501.88)**	**size (min(5.25^n, 501.88))**
1	8	216 b	5.25	141.75 b	5.25	141.75 b
2	64	1.69 kb	27.56	744.19 b	27.56	744.19 b
3	512	13.5 kb	144.70	3.81 kb	144.70	3.81 kb
4	4096	108 kb	759.69	20.03 kb	501.88	13.23 kb
5	32768	864 kb	3988.38	105.16 kb	501.88	13.23 kb

n	8^n	size (8^n)	5.25^n	size (5.25^n)	min(5.25^n, 501.88)	size (min(5.25^n, 501.88))
6	262144	6.75 Mb	20938.99	552.10 kb	501.88	13.23 kb

In [7]:
```
def move_knight(pos):
    """
    Return a list that contains all the possible final positions for the kni
ght.

    First, it checks all the allowed moves for the knight in the pos list,
    then applies the move to each position and save all the final locations
    into a list. The list might not have all unique elements, because the
    function returns all the possible moves, and there is frequently more
    than one way to reach the same location.

    Parameters
    ----------
    pos : list
        The initial positions of the knight

    Returns
    -------
    list
        All the possible final locations
    """
    # To store the return
    all_out = []

    # Check each positions
    for p in pos:
        # To store the final location of a single position
        out = []
        # To store the allowed moves
        moves = []

        # Check the allowed moves
        if (p > 7 and p%8 > 1):
            moves.append(-10)
        if (p > 15 and p%8 != 0):
            moves.append(-17)
        if (p > 15 and p%8 != 7):
            moves.append(-15)
        if (p > 7 and p%8 < 6):
            moves.append(-6)
        if (p < 56 and p%8 < 6):
            moves.append(10)
        if (p < 48 and p%8 != 7):
            moves.append(17)
        if (p < 48 and p%8 != 0):
            moves.append(15)
        if (p < 56 and p%8 > 1):
            moves.append(6)

        # Apply the allowed moves into the positon
        for m in moves:
            out.append(p+m)

        # Add each result to te final list
        all_out += out

    return all_out

def unique(l):
    """
    Return all the unique elements in a list sorted.

    First, it finds all the unique elements in a list, then it sorts
    the list in ascending order.

    Parameters
    ----------
```

```python
    l : list
        List to find the uniques elements and to be sorted

    Returns
    -------
    list
        Unique and sorted list
    """
    # To store the return
    unique_l = []

    # Check every element
    for e in l:
        # If it is unique, add to the final list
        if e not in unique_l:
            unique_l.append(e)

    # Sort the unique list
    unique_l.sort()

    return unique_l

def answer(src, dest):
    """
    Return the lowest number of knight moves to get from src to dest.

    It calculates every possible allowed path to reach from source to
    the destination until it reaches, then it returns the lowest
    number of steps needed.
    It does not exclude the back and forth path moves, but it is
    faster than calculating every single move because it just takes
    into account the unique starting locations.

    Parameters
    ----------
    src : int
        Source, he initial location
    dest : int
        Destination, the final location

    Returns
    -------
    int
        Number of fewest moves necessary
    """
    # Number of steps
    n = 0
    # Set the initial position into a list
    positions = [src]
    # Repeat until the final destination
    while not(dest in positions):
        # Make the moves and save it back to positions
        positions = unique(move_knight(positions))
        # Increment the number of steps.
        n +=1

    return n
```

Test Cases

- src = 19, dest = 36, should return 1
- src = 0, dest = 1, should return 3

In [8]:
```
answer(19, 36)
```
Out[8]: 1

In [9]:
```
answer(0, 1)
```
Out[9]: 3

Method 4

There is still one more last improvement that could be done, which is to avoid loops, we don't need to go back to where we were. From the last example, position 42 is going to be check again for the third step. If it was possible to reach the destination from there, then we would find the solution with just one step.

Hence, the idea is to remove the positions from 2 steps backward, or remove the most repeated path in each step, excluding the first step, where everything is unique. This would reduce the number of possible moves in one after the first step. Let's consider that it removes one move from all steps, so the average number of steps would be 4.25, then using the same number of average steps, we get a maximum of $4.25^{3.75} = 227.23$. Therefore, we have

n	8^n	5.25^n	**min(5.25^n, 501.88)**	**min(4.25^n, 227.23)**
1	8	5.25	5.25	4.25
2	64	27.56	27.56	18.06
3	512	144.70	144.70	76.77
4	4096	759.69	501.88	227.23
5	32768	3988.38	501.88	227.23
6	262144	20938.99	501.88	227.23

n	**size (8^n)**	**size (5.25^n)**	**size (min(5.25^n, 501.88))**	**size (min(4.25^n, 227.23))**
1	216 b	141.75 b	141.75 b	114.75 b
2	1.69 kb	744.19 b	744.19 b	487.69 b
3	13.5 kb	3.81 kb	3.81 kb	2.02 kb
4	108 kb	20.03 kb	13.23 kb	5.99 kb
5	864 kb	105.16 kb	13.23 kb	5.99 kb
6	6.75 Mb	552.10 kb	13.23 kb	5.99 kb

However, this version was not implemented, since it was not a huge leap, as you can see in the next plot.

In [10]:

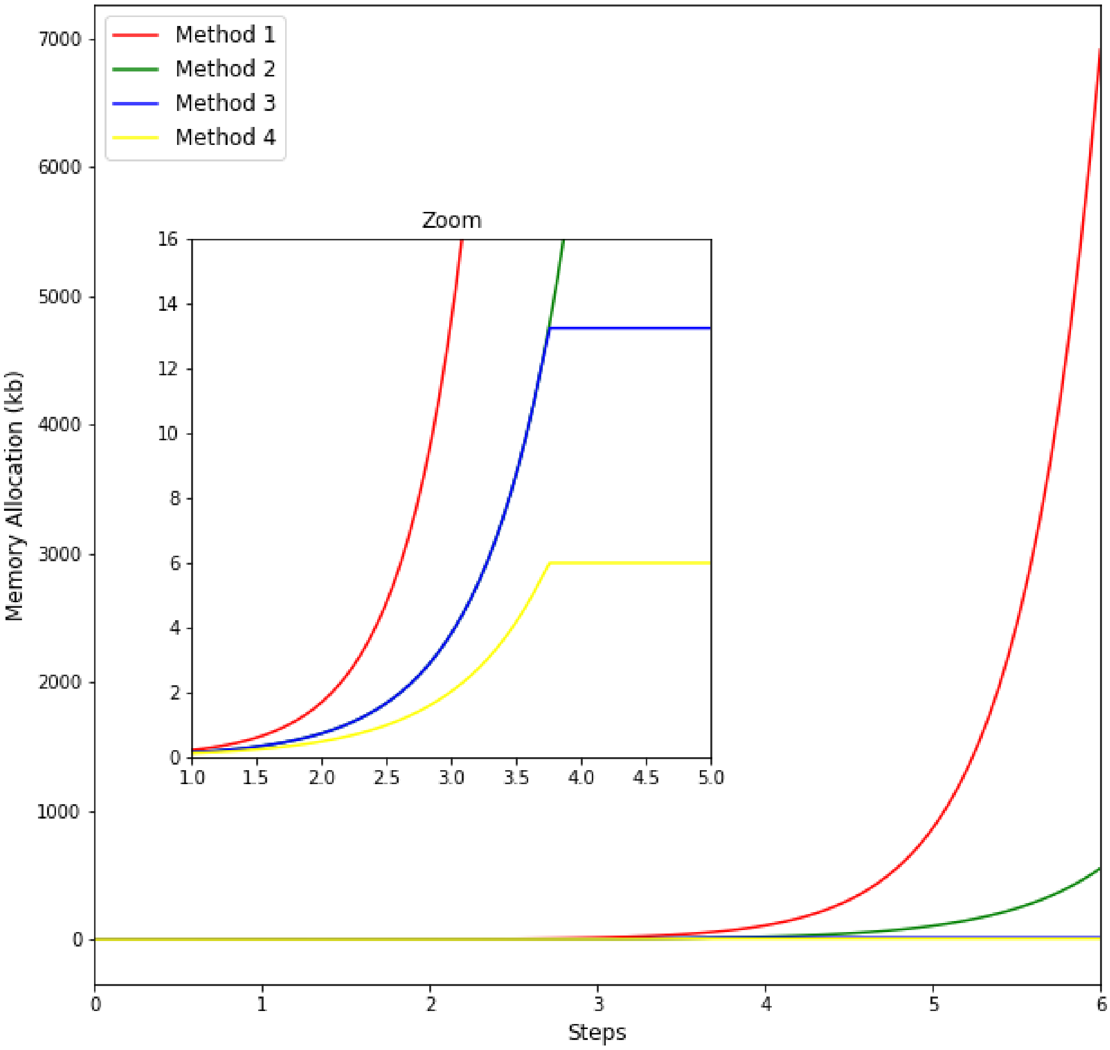
```
import matplotlib.pyplot as plt
import numpy as np

# Plot
fig, ax = plt.subplots(figsize=(10, 10))
# Method 1
x = np.linspace(0, 6, 100)
y1 = 8**x*27/1024
plt.plot(x, y1, label='Method 1', c='red')
# Method 1
y2 = 5.25**x*27/1024
plt.plot(x, y2, label='Method 2', c='green')
# Method 3
y3 = 5.25**x*27/1024
y3 = np.where(y3 < 13.23, y3, 13.23)
plt.plot(x, y3 , label='Method 3', c='blue')
# Method 4
y4 = 4.25**x*27/1024
y4 = np.where(y4 < 5.99, y4, 5.99)
plt.plot(x, y4 , label='Method 4', c='yellow')

# Plot Settings
plt.legend(prop={'size': 'large'}, loc='upper left')
ax.set_xlabel("Steps", size='large')
ax.set_ylabel("Memory Allocation (kb)", size='large')
ax.set_xlim(0, 6)

# Zoom plot
a = plt.axes([0.2, 0.3, 0.4, 0.4])
plt.plot(x, y1, c='red')
plt.plot(x, y2, c='green')
plt.plot(x, y3, c='blue')
plt.plot(x, y4, c='yellow')
plt.title('Zoom')
plt.xlim(1, 5)
plt.ylim(0, 16)


plt.show()
```


Method 1
Method 2
Method 3
Method 4
Zoom
Memory Allocation (kb)
Steps
0
1000
2000
3000
4000
5000
6000
7000
0
1
2
3
4
5
6
0
2
4
6
8
10
12
14
16
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

In [ ]: