University of Zurich[UZH]

Recursion

Mohammad Alminawi

Supervisors:
Prof. Prasenjit Saha

, 2021

Contents

1. Introduction

Recursive algorithms have been used in computer science since the inception of the field, hence we will start with a definition.

Theorem 1.1 (Recursion) *Recursion is a method of solving for problems, where the solution depends on solutions to smaller instances of the same problem. This is achieved using a function that calls itself within its own definition.*

The greatest benefit of using recursion is that we can define an infinite number of processes or steps using a finite number of statements. In **python** this is achieved by calling a function within itself alongside a termination condition to know when the function should output an answer, instead of calling itself.

One of the fundamental applications of recursion is in the definition of the **factorial**, to see this, let us write out a few factorial functions:

$$1! = 1$$

$$2! = 2 \cdot 1 = 2 \cdot 1!$$

$$3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2!$$

Repeating this leads to the general formula:

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! \tag{1.1}$$

Now we can convert this to a function in python as follows:

```
def factorial(n):
    if n == 0:
    # Defines a termination condition
        return 1
    else:
    # Recursion step
        return n * factorial(n-1)
```

The function is composed of two components, an *if statement* that checks if we have the base case of 0! and an *else statement* that contains the recursion step, which is defined by calling the function in the return statement.

2. Exploring Recursion

Recursion is a large topic with many aspects that are worthy of discussion, it is also closely tied to the concept of loops, therefore we will briefly discuss a variant of recursion as well as the difference between recursion and loops.
Let us consider the factorial once more. We showed that a definition using recursion is certainly possible, but it suffers from a simple flaw, to identify it, consider the following question:
Knowing that $5! = 120$ *what is* $6!$*?* The answer is simply $6 \cdot 5! = 720$. As humans, we could make use of the given information to simplify the calculation for ourselves, but the function defined previously has to go through the entire process from 6! to 0! before terminating, this is a slow process, but it can be improved if we allow our function to store results as we go, this is known as **memoization**.

2.1 Memoization

Memoization is an optimization technique for recursive algorithms that operates by storing results from recursive calls, thus we only need to perform the calculation once and we will have all the results ready to use for the next calculation. In essence, this allows the code to use information such as $5! = 120$ to calculate 6! without having to calculate 5! on its own.

Let us now look at what this would look like in python code:

```
terms = [] #Empty list to store results

def factorial_memoization(n):
    if n == 0:
    # Defines termination condition
        return 1

    if len(terms)>n-1:
    # Checks if the result has been calculated before
        return terms[n-1]
    else:
    # Adds result to the list and returns it
        terms.append( n * factorial_memoization(n-1))
        return terms[n-1]
```

The usage of memoization adds an additional step, therefore the first time calculating any factorial is slower than the "normal" method. However, calculation of subsequent factorials or the same factorial become much faster.

2.2 Iteration

While iteration is not a type of recursion, it aims to achieve similar goals and can theoretically be used in place of recursion for any problem , therefore understanding it and how it differs from recursion could help us develop a deeper understanding of the topic.

In programming, the term iteration refers to the usage of loops, which keep repeating a set of instructions until the condition returns **False**, when using python, we can apply iteration through the usage of *for loops* or *while loops*.

Iteration requires a smaller amount of computing resources when compared to recursion; this is due to recursion requiring a large number of inputs and function calls to be maintained until the final result is computed, while iteration only requires one set of inputs.

The time saved by the computer when using iteration is typically spent by the developer writing the code as it is usually more difficult to find an iterative formulation than a recursive formulation. Iterative functions typically look messier as well.

We once again return to the factorial function and provide two iterative implementations.

```
# While loop
def factorial_while(n):
    # Iteration parameter, tells us how close we are to the final
   answer
    iter_param = 0
    #Base case 0! = 1
    ans = 1
    while iter_param < n:
    # Compares iteration parameter to n, terminates at equality
        ans = ans * (iter_param + 1)
        iter_param += 1
        # Increments the parameter by one
    return ans

# For loop
def factorial_for(n):
    # Base case 0! = 1
    ans = 1
    # Define the range of numbers that we need to calculate
   factorials of (1 to n)
    li = range(1, n + 1)
    #Loop through the range of numbers
    for num in li:
        ans = ans * num
    return ans
```

2.3 Comparing the Methods

Now that we have briefly discussed each of the methods and their potential advantages, it would be interesting to see whether we observe the expected results when testing them.

Since we have an implementation of the factorial function using each method, we can easily test our hypotheses. We will calculate the factorials from 1000! to 4000! in increments of 100 and then plot the run-time to compare the speed of the methods.

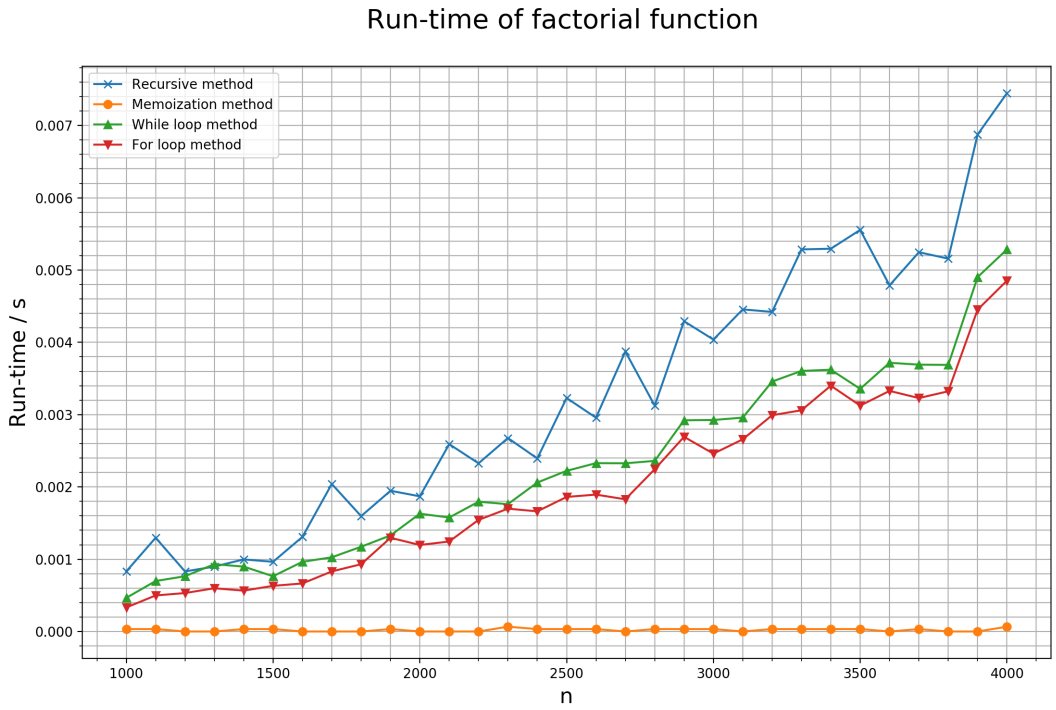


Figure 2.1: Run-time comparison factorial function

The graph verifies the statements made previously. We can see iteration running more quickly than recursion, with the difference gradually increasing as the task becomes more demanding. Additionally, we can see that memoization increases the efficiency of recursion by a large margin.

The code used to produce the graph as well as the functions used can be found at `https://github.com/M-AlMinawi/Recursion-lecture-UZH-/blob/main/Factorial.py`

3. Applications of Recursion

In this chapter, we will take a look at some applications of recursion, each section will consist of an application, an explanation and an exercise that is relevant to the application shown. Mandatory exercises will be indicated.

The code used for the applications and optional exercises can be found at `https://github.com/M-AlMinawi/Recursion-lecture-UZH-`. The solutions to the mandatory exercises will also be uploaded after the hand-in date for students to review them at their convenience.

3.1 Discrete Fourier Transform

Discrete transforms are mathematical transformations of functions between two discrete (not continuous) domains, such as a transformation from time to frequency. *Discrete Fourier transforms* are the most important among these transforms; they are used to perform Fourier analysis in many practical applications such as image and signal processing. They are also powerful mathematical tools that allow us to efficiently solve partial differential equations.

Starting with a set of N **complex** numbers $\{x_n\} = x_0, x_1, \ldots, x_{N-1}$ we transform into another set of N **complex** numbers $\{X_n\} = X_0, X_1, \ldots, X_{N-1}$ using the following equation.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N}kn} \tag{3.1}$$

We can simplify the notation by defining ω_N^{kn} as follows:

$$\omega_N^{kn} = e^{-\frac{i2\pi}{N}kn} \tag{3.2}$$

Which results in (3.2) simplifying to:

$$X_k = \sum_{n=0}^{N-1} \omega_N^{kn} x_n \tag{3.3}$$

You may now notice that (3.3) is the equation of a vector being multiplied by a matrix. Let us consider the case of $N = 2$ to see this more explicitly.

$$\begin{aligned} \omega_2^{kn} &= e^{-\frac{i2\pi}{2}kn} = e^{-i\pi kn} \\ X_0 &= \omega_2^{00} x_0 + \omega_2^{01} x_1 \\ X_1 &= \omega_2^{10} x_0 + \omega_2^{11} x_1 \end{aligned}$$

If we consider a vector $\vec{X_k}$, a vector $\vec{x_n}$ and a matrix ω_2^{kn}, then we can write the previous equations in the following form.

$$\begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} \omega_2^{00} & \omega_2^{01} \\ \omega_2^{10} & \omega_2^{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{3.4}$$

Where we evaluated ω_2^{kn} in the last step using the definition in (3.3)

To transform a vector of order N we need to perform N^2 multiplications, this gives the transform a time complexity of $O(N^2)$. Generally speaking, a complexity of $O(N^2)$ is inefficient, so we look for way to improve the algorithm. There are two properties of the Fourier transform that we can exploit to increase efficiency:

- ω_N^{kn} is symmetric under an exchange of k and n, meaning that $\omega_N^{kn} = \omega_N^{nk}$
- The complex exponential is a periodic function. $e^{ix} = e^{i(x+2\pi n)}$

We can capitalize on both properties when N is a power of two ($N = 2^M$), since that guarantees that $N/2$ is an integer, which allows us to manipulate the formula according to the following steps:

- Split the sum into odd and even terms:

$$X_k = \sum_{n=0}^{N-1} \omega_N^{kn} x_n = \sum_{n=0}^{N/2-1} \omega_N^{k(2n)} x_{2n} + \sum_{n=0}^{N/2-1} \omega_N^{k(2n+1)} x_{2n+1}$$

- Notice that $\omega_N^{k(2n)}$ can be rewritten as follows:

$$\omega_N^{k(2n)} = e^{-\frac{i2\pi}{N}k2n} = e^{-\frac{i2\pi}{N/2}kn} = \omega_{N/2}^{kn}$$

- Then the split sum takes the following form:

$$X_k = \sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n} + e^{-\frac{i2\pi}{N}k} \sum_{n=0}^{N/2-1} \omega_{N/2}^{kn} x_{2n+1}$$

- We can label the first sum E_k and the second as O_k representing even and odd respectively, this yields the following equation.

$$X_k = E_k + e^{-\frac{i2\pi}{N}k} O_k$$

- Lastly we use the periodicity of the exponential. Since $\omega_{N/2}^{kn}$ is $N/2$ periodic, shifting k by $N/2$ does not change the sums. However $e^{-\frac{i2\pi}{N}k}$ is N periodic, so a factor of $e^{-i\pi} = -1$ emerges, which means that we only need to calculate X_k for k between 0 and $N/2 - 1$, then we can use the following relation:

$$X_{k+N/2} = E_k - e^{-\frac{i2\pi}{N}k} O_k$$

Applying this algorithm saves a lot of time in comparison to the direct usage of the discrete Fourier transform algorithm, to see why, let us consider a vector with N elements. Using the standard transform algorithm, we have a time complexity of $O(N^2)$. In comparison, the complexity using this algorithm can be computed as follows (where R denotes run-time and k is an arbitrary constant):

$$R(N) = 2R(N/2) + kN = 4R(N/4) + kN + kN = \dots$$

Which leads us to:

$$R(N) = NR(1) + kN\log_2(N) \tag{3.5}$$

To understand this result, let us closely examine what happens in the algorithm;

- Each evaluation is broken down into 2 evaluations of half the order and N multiplications.
- We chose $N = 2^M$ with M arbitrary.
- If we let p be a the number of iterations, the previous calculation can be written as follows:

$$R(N) = 2^pR(N/2^p) + kpN$$

- Recognizing that our algorithm terminates when $N/2^p = 1$ allows us to identify $p = \log_2(N)$
- Therefore we can say that this algorithm has a time complexity of $O(N\log_2(N))$

3.1.1 Exercise: Discrete Fourier Transform

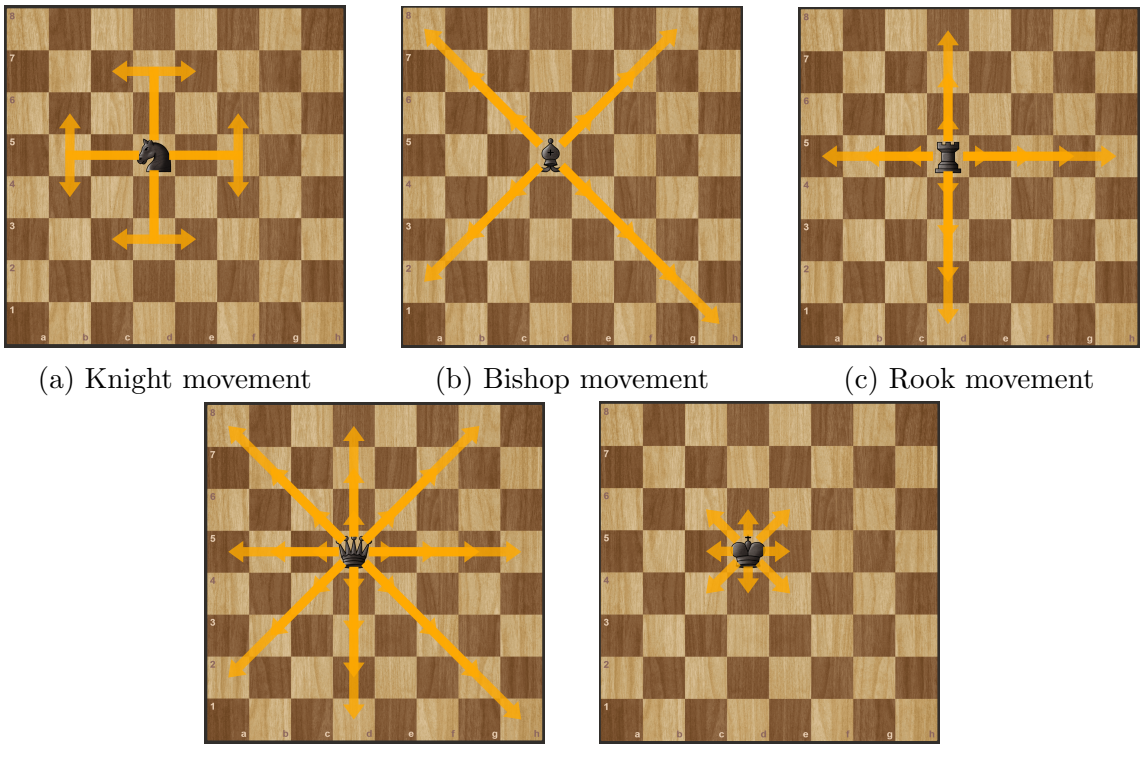
The exercise for this section consists of three components:

- Implement the basic discrete Fourier transform algorithm. (Optional)
- Implement the fast discrete Fourier transform algorithm. (Mandatory)
- Compare the run-time of the two algorithms for arrays of different lengths. (Optional)

3.2 Chess Puzzles

In this section, we will look at two types of chess puzzles, therefore it would be instructive to first review the movement of Chess pieces.

- Knights can move two squares in one direction, followed by one square in the perpendicular direction. Knights may move even if other pieces are in their path.
- Bishop can move any number of squares along a diagonal. Bishops cannot pass through other pieces.
- Rooks can move any number of squares along the row and column that they occupy. Rooks cannot pass through other pieces.
- Queens can move any number of squares along the row, column and diagonal that they occupy (Rook + Bishop). Queens cannot pass through other pieces.
- Kings can move one square in any direction. Kings cannot pass through other pieces.



(a) Knight movement (b) Bishop movement (c) Rook movement

(d) Queen movement (e) King movement

Figure 3.1: Chess pieces movement

There are more rules in Chess, but we do not need to know them for the puzzles that we will look at.

3.2.1 Non-Attacking Chess Pieces

Given an 8×8 chess board, we can place 16 kings, 8 queens, 8 rooks, 14 bishops or 32 knights such that none of them are attacking one another. For the demonstration we will take a look at the 16 kings problem, while the 8 queens problem will be left as a mandatory exercise. If you are feeling ambitious, you are free to solve the other problems.

To start solving, we recall that a chess king attacks all of its directly neighbouring squares, this means that each king will need at least a 2×2 area for himself.

The problem can be solved as follows:

- Represent the chess board by an 8×8 matrix and choose suitable values to represent a king, an attacked square and an available square. Example: (1, -1, 0)
- Place the first king and mark the attacked squares by changing the values of the matrix entries.
- Recursively add kings until there are no more available squares.
- Repeat this process while varying the position at which we place the first king.
- Store all valid solutions and then compare them with one another to remove any repeated solutions.

We will end up with 48 unique solutions. Below is a solution to the problem.

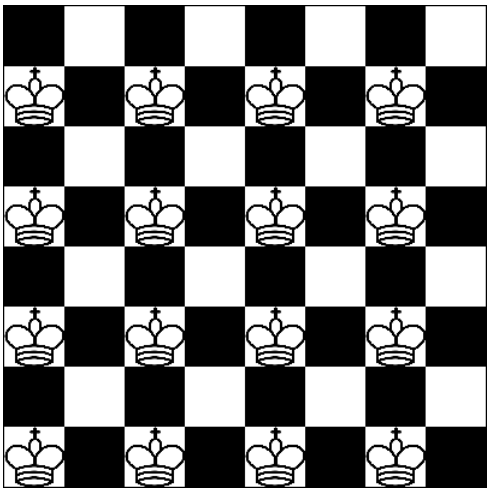


Figure 3.2: 16 Kings chess problem

Exercise: Eight Queens Problem (Mandatory)

At its core, this problem is very similar to the 16 kings problem. However, we can simplify it by noting that we can only have one queen per file or rank, meaning that we can work with a vector with eight entries instead of a matrix with 64 entries.

To solve this problem you need to provide a program capable of finding all 92 solutions to the problem. Additionally, try generalizing the code to solve the n queen problem, where we try placing n queens on an $n \times n$ chess board.

Below is an example of a valid solution to the 8 queens problem.

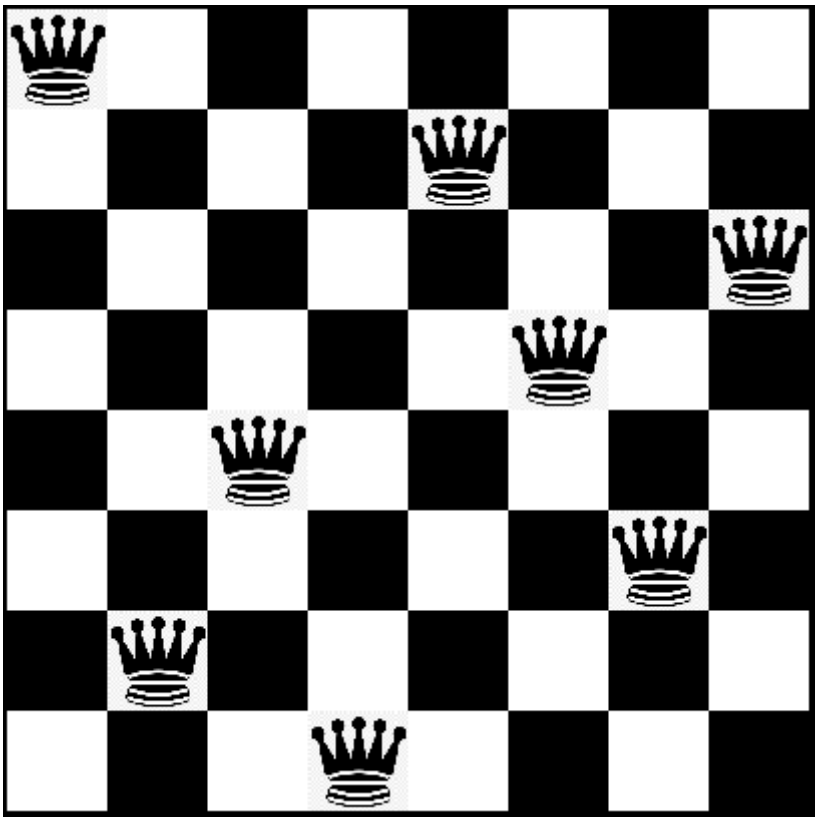


Figure 3.3: Solution to 8 queens problem

For guidance, you may want to check the code used for the 16 kings problem at `https://github.com/M-AlMinawi/Recursion-lecture-UZH-/blob/main/16%20Kings.py`

3.2.2 The Trapped Knight Puzzle

In this puzzle we consider a chess knight starting at the center of a infinite numbered board as shown in the image below.

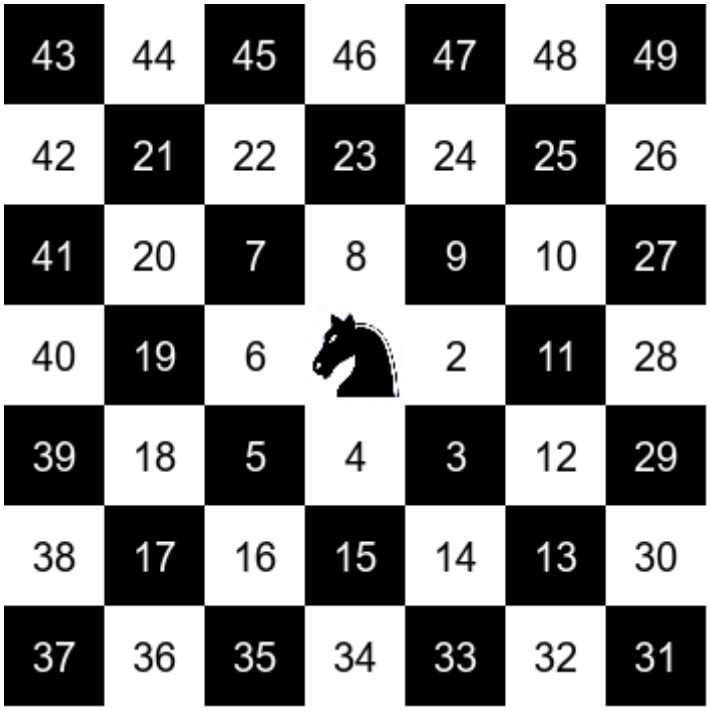


Figure 3.4: Knight at center of spirally numbered chess board

The knight must move to the smallest allowed number and may not revisit a spot that it has been to before. Initially it may seem like the knight will keep moving indefinitely, but it has been shown that the knight will eventually trap itself. We aim to use recursion to find out the number of the square at which it ends up and the number of jumps that it makes before trapping itself.

To start this problem, we need to first configure a grid that is numbered as shown in the image. This can be achieved using the following steps:

- Input N determines the number of rows and columns given by $2N - 1$. This choice is justified by step 2.
- Recognize that the largest number in each spiral layer is a square of an odd number: $1, 9, 25, 49, \ldots$
- Iterate over odd numbers and identify the starting and ending points of each layer: $(1, 1), (2, 9), (10, 25), \ldots$
- Create the arrays and store them in a list
- Define mathematical relation to layer these arrays around one another

Once we have the grid, we can move on to the actual solving of the puzzle, this is done recursively according to the following steps:

- Calculate all possible knight moves (There are always eight, since the board is infinite).
- Check if any of the positions have been visited before and remove them from the list of possibilities.
- Compare the values of the remaining positions and move the knight to the smallest number.
- Store the position in the list of visited positions and repeat the process.
- The process terminates if the knight has previously visited all of the squares that it can access.

Using the stored positions, we can plot the path taken by the knight across the board until it gets trapped, the graph below shows that path.

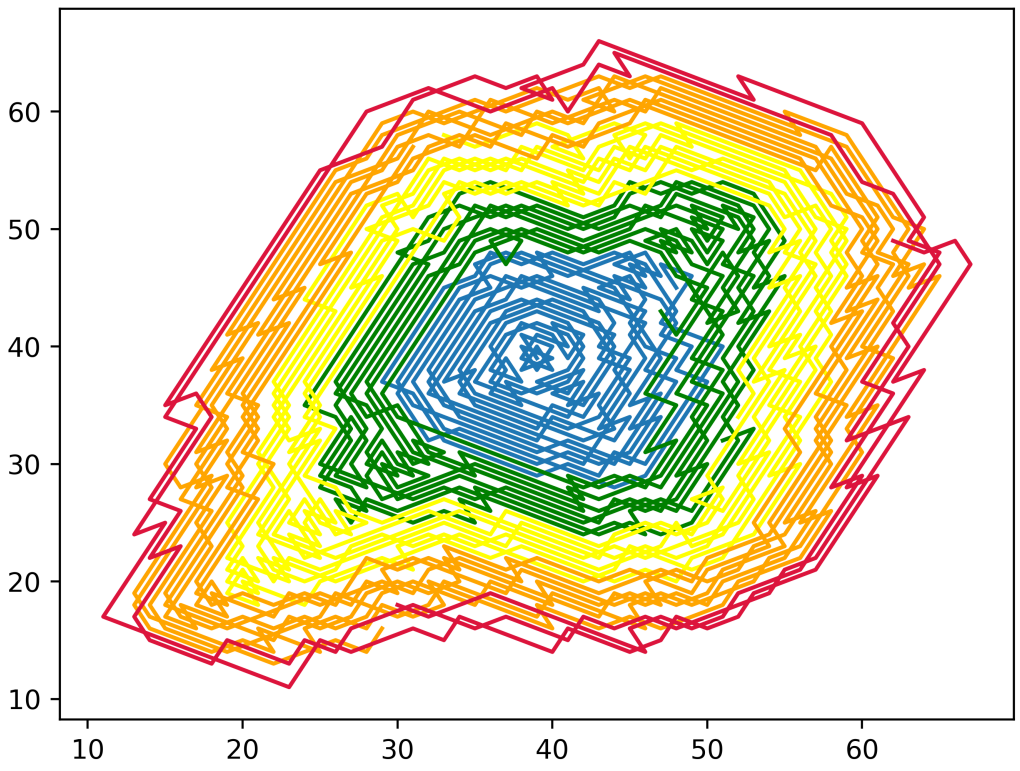


Figure 3.5: Path of trapped knight

The code used to solve this problem can be found at `https://github.com/M-AlMinawi/Recursion-lecture-UZH-/blob/main/Trapped%20Knight.py`

Corner Knight Exercise (Optional)

Let us now consider a knight starting at the left corner of an infinite board that is numbered as follows.

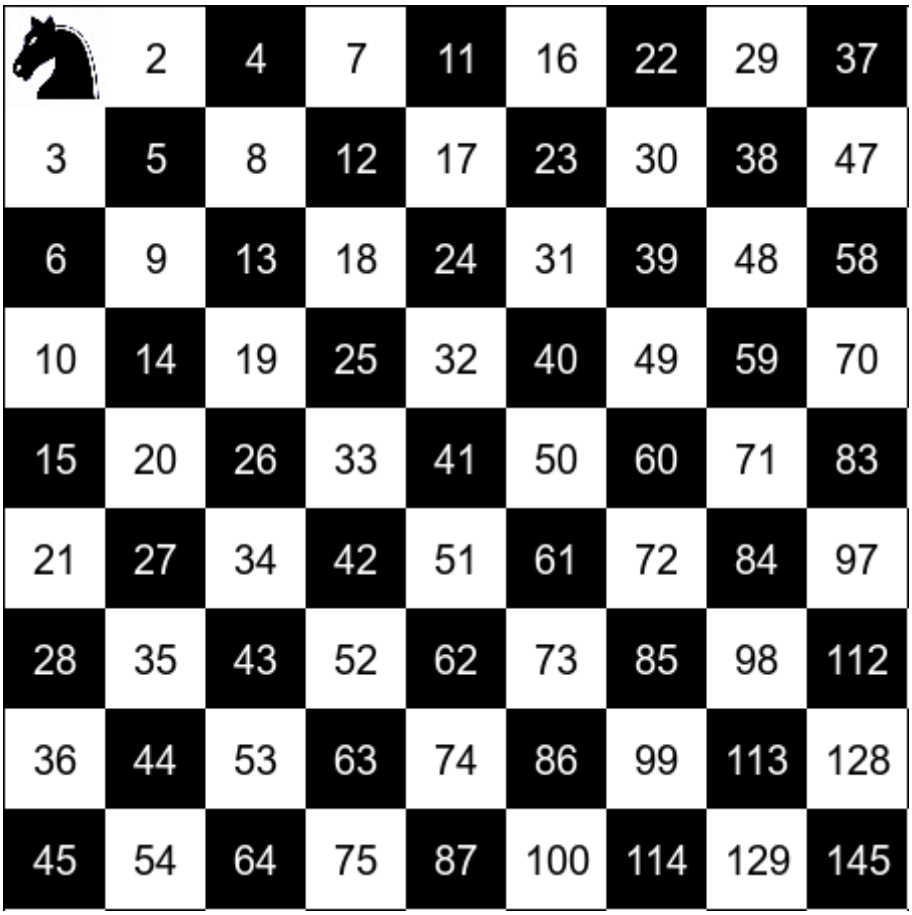


Figure 3.6: Diagonally numbered board

Only the first couple rows and columns are numbered correctly, since we are showing a finite image of an infinite board, but it should be sufficient to see that the numbering scheme is obtained by moving to the right and then diagonally for the same number of steps.

Once again we will only allow the knight to jump to the smallest number without revisiting any square that it has previously been to. The knight will eventually trap itself as well.

To successfully complete the puzzle, you will need to write code to:

- Find the number of steps after which the knight becomes trapped.
- Find the value (number) of the square that the knight ends up on.
- Produce a graph representing the path of the knight from the starting square until it gets trapped.

3.3 Sorting algorithms

A sorting algorithm is a set of steps that allow us to place a list of objects in a desired order. An example would be arranging a set of numbers in ascending order. Sorting algorithms are very compatible with recursive approaches, since any set can be treated as the union of two smaller sets.

There are many ways of going about sorting a list of objects and discussing them all is outside the scope of this lecture, therefore we will only concern ourselves with three simple algorithms.

3.3.1 Bubble sort

Bubble sort refers to an algorithm in which we compare each element with the one after it, if our element is larger than the element following it, then we swap them, otherwise we leave them as they are and move on to the next two elements.

Once we have passed through the entire list once, we can guarantee that the largest element has made its way to the end of the list. The name *bubble sort* comes from the fact that the largest element bubbles to the front of the list.

On average, we will need to pass through the entire list once for each element to be placed in the correct position and each pass requires us to make as many comparisons as there are elements in the list, thus we can say that bubble sort has a time complexity of $O(n^2)$; meaning that the time taken to sort a list is proportional to the square of the number of elements in the list.

3.3.2 Quicksort

Quicksort is an example of a *divide and conquer* algorithm, it operates by dividing the problem into smaller problems, solving those and then combining their solutions to obtain the final answer; the steps for the algorithm are:

- Choose a pivot element.
- Place all elements smaller than the pivot to the left of it and the remaining ones to the right of it.
- Apply the procedure again to the two sub-lists that we created in the second step

Quicksort is an improvement in efficiency over bubble sort in the average case, while matching it in the worst case scenario, making it a better sorting algorithm. The time complexity of quicksort is $O(n \log(n))$ rather than $O(n^2)$

3.3.3 Exercise: Selection sort (optional)

Selection sort is an iterative algorithm in which we loop over a list of items, select the smallest element, place it at the beginning of the list and repeat the process with the remaining elements until we have an ordered list.

To implement selection sort in Python, start by breaking down the algorithm into steps:

- Start with an empty list and random list of elements.
- Find the smallest element in the random list and add it to the empty list.
- Check if the two lists are now of the same length. If they are, the process terminates. Otherwise, we repeat the previous step.

You may want to take a look at the code used to implement bubble sort and quicksort as a reference `https://github.com/M-AlMinawi/Recursion-lecture-UZH-`.

3.4 New Formulae for π

There are numerous ways of calculating pi, hence it is no surprise that we can find recursive (or iterative) methods to perform the calculation. We will look at *Machin-like formulae*.

These formulae can be obtained through a combination of two results.

$$\arctan(1) = \frac{\pi}{4} \tag{3.6}$$

and

$$\tan(\alpha + \beta) = \frac{\tan(\alpha) + \tan(\beta)}{1 - \tan(\alpha)\tan(\beta)} \tag{3.7}$$

If we choose $\alpha = \arctan(\frac{a_1}{b_1})$ and $\beta = \arctan(\frac{a_2}{b_2})$ and apply arctan to both sides of (3.7) we obtain the following equation:

$$\arctan(\frac{a_1}{b_1}) + \arctan(\frac{a_2}{b_2}) = \arctan\left(\frac{a_1 \cdot b_2 + a_2 \cdot b_1}{b_1 \cdot b_2 - a_1 \cdot a_2}\right) \tag{3.8}$$

Derivation

Left hand side:

$$\arctan\left(\tan\left(\arctan(\frac{a_1}{b_1}) + \arctan(\frac{a_2}{b_2})\right)\right) = \arctan(\frac{a_1}{b_1}) + \arctan(\frac{a_2}{b_2})$$

Right hand side:

$$\arctan\left(\frac{\tan(\arctan(\frac{a_1}{b_1})) + \tan(\arctan(\frac{a_2}{b_2}))}{1 - \tan(\arctan(\frac{a_1}{b_1})) \cdot \tan(\arctan(\frac{a_2}{b_2}))}\right) =$$

$$\arctan\left(\frac{\frac{a_1}{b_1} + \frac{a_2}{b_2}}{1 - \frac{a_1}{b_1} \cdot \frac{a_2}{b_2}}\right) =$$

$$\arctan\left(\frac{a_1 \cdot b_2 + a_2 \cdot b_1}{b_1 \cdot b_2 - a_1 \cdot a_2}\right)$$

Using this, we can start with an arbitrary choice of a_1 and b_1, then writing $\pi/4 = \arctan(1/1)$ we can find a_2 and b_2. Furthermore, we can find $2\arctan(a_1/b_1)$ and then use that to find higher order formulae, which would converge to π more quickly.

3.4.1 Exercise (Optional)

In this optional exercise, you should use iteration or recursion to find formulae of π by using the procedure outlined above.

Be careful while implementing the equations, since tan is a periodic function, meaning that the equations yield answers of $\frac{\pi}{4} \mod \pi$.

An implementation of these formulae can be found at `https://github.com/M-AlMinawi/Recursion-lecture-UZH-/blob/main/New%20Pi%20formulae.py`

Iteration is significantly more efficient for these formulae, thus it is preferred over recursion.

4. Helpful Tools and Practices

4.1 Integrated Development Environment (IDE)

When coding, it is beneficial to use an editor that simplifies the process, this can come in the form of syntax detection, auto-completion, a console, an interpreter and so on. There are many well known IDEs for Python, but the two that stand out the most and are most commonly used in the industry are *Visual Studio Code* and *PyCharm*.

As students at the University of Zurich, you can access either editor for free. PyCharm comes with more innate features and requires very little set up, but it can be demanding in terms of memory and CPU power. Visual Studio Code on the other hand is highly customizable and lightweight, but requires a bit more time to set up.

If you are interested in using either IDE, then you can download them using the following links:

- Visual Studio Code: `https://code.visualstudio.com/`
- PyCharm: `https://www.jetbrains.com/pycharm/`

4.2 Python Libraries

By this point in the course, you have definitely encountered many Python libraries, therefore I will only mention a few that you may not be very familiar with, but could benefit from whenever you are coding.

- **time**:
  The time library allows Python to understand the concept of time, this allows us to measure run-time, pause the program for a certain period of time or access date, time zones, etc ...
- **sys**:
  Short for "system", this library allows us to interact with the Python interpreter and change its settings.
- **pygame**:
  pygame is intended for game development in Python, but it can also be used to produce graphics, such 3.2

4.3 Python Expressions

Python's syntax can often allow us to shorten our code while achieving the same results, below are a few tricks that you can use to write clearer code:

- f strings:
  An f string allows us to insert Python code within a string, which can shorten expressions significantly when using print statements.

```
    print(f"The numbers between 1 and 40 are {range(1,40)}")

```

  This code will then print the string "The numbers between 1 and 40 are" followed by a list containing the numbers.

- List indexing:
  You are likely familiar with the usage of square brackets [ ] to select items from a list based on indices, but perhaps you are not aware of the fact that we can give three inputs, namely [start:stop:step]. Furthermore, we can use negative indices and steps to read the list in the opposite order.

```
    sample = list(range(10)) # Creates a list of numbers from 0 to 9
    sample[4] # returns the element with index 4 (5th element)
    sample[:6] # returns elements with indices 0 up to (but not including) 6
    sample[1:] # returns elements with indices 1 and up
    sample[1:7:2] # returns elements with indices between 1 and 7 (not included) in steps of two sample[1], sample[3], sample[5]
    sample[::2] # returns elements in list in steps of 2 sample[0], sample[2], sample[4], sample[6], sample[8]
    sample[-1] # returns the last element in the list
    sample[::-1] # reverses the list

```

- Returning multiple values from functions:
  We can return and store any number of values from a function. In some cases, we can make our code more readable by choosing an appropriate way of storing the values.

```
    def x():
    return 1, 2, 3, 4
    res = x() # Stores the values in a list in the order than they are returned
    a,b,c,d = x() # Stores the values as four separate variables

```