ELEN90097 Workshop 2 – Modelling and Analysis for AI

This workshop is allocated 3 weeks.

Topics Covered

This workshop will cover the fundamentals of data types/structures and algorithms with an emphasis on recursion and dynamic programming.

- Data types and structures
  - Sets
  - Stacks, queues
  - Graphs, trees
- Algorithms
  - Recursion, dynamic programming
  - Sorting, search

Workflow and Assessment

This subject follows a problem- and project-oriented approach. This learning workflow focuses on solving practical (engineering) problems, which motivates acquiring theoretical (background) knowledge.

Objectives

- Use these problems as a motivation to learn the fundamentals covered in lectures.
- Learn how to formulate and solve problems in practice.
- Familiarise yourself with practical software tools.
- Connect theoretical and practical knowledge by doing it yourself.

The goal is to gain hands-on experience and learn by doing! Understand how theoretical knowledge discussed in lectures relates to practice. Develop motivation for gaining further theoretical and practical knowledge beyond the subject material.

> **Self-learning** is one of the most important skills you should acquire as a student. Today, self-learning is much easier than it used to be thanks to many online resources.

Assessment Process

1. Follow the procedures described below, perform the given tasks, and answer the workshop questions **in this Python/Jupyter notebook! The resulting notebook will be your Workshop Report!**
2. Submit the workshop report at the announced deadline
3. Demonstrators will conduct a brief oral quiz on your submitted report in the subsequent weeks.
4. Your workshop marks will be based on the report you will submit and oral quiz results (with a heavy weight on the oral component).

> **The goal is to learn**, NOT blindly follow the procedures in the fastest possible way! **Do not simply copy-paste answers (from the Internet, friends, etc.). You can and should use all available resources but only to improve your understanding. If you copy-paste, you may perform poorly on the oral quiz!**

Notes

- Use the table of contents feature of Jupyter Notebook for your convenience, see View menu above, Table of Contents. You can also turn on section numbers there if you wish.
- Similarly, you can turn on "show line numbers" in the same menu.

Data Types and Structures

A data structure is a data organization and storage format on a computer. As you can imagine there are many ways of organising and storing data. Therefore, different programming languages approach this in different ways. Usually, high-level programming languages like Python provide many convenient *built-in* data structures, whereas low-level system languages like C only provide basic data types. In both cases, external libraries such as Numpy provide additional support for advanced data structures (e.g. vectors and matrices in the case of Numpy).

In this subject, we will focus on **Python** which famously adopts a ``batteries included" philosophy. This does *not* mean Python has the best data structures ever. It just means we want consistency over our AI subjects and Python is the current popular language in AI/ML, and a comparative study of data structures of languages is beyond our scope.

Python Data Types, Structures, and Libraries

Python has a lot of built-in datatypes. You don't need to know them all and you can learn about them as you need. There are many tutorials online on this topic such as this one.

In addition to built-in data types, there are many respected libraries in Python providing advanced data structures such as Numpy (vectors, matrices, Matlab-like functionality), NetworkX (data structures for graphs), etc.

Next Step: a selective tour

You should already know about the basics of Python built-in basic data types such as lists, strings, and dictionaries. This workshop is not about learning Python basics and we don't have time to cover all data structures, either. Therefore, we will go over a few interesting, engineering-relevant data types and structures with selected examples and exercises.

Sets in Python

A set in Python is an unordered collection data type that is iterable and has no duplicate elements. This matches quite perfectly with the mathematical definition we know since primary school. There are many nice tutorials on the topic, see for example, this one or this or these exercises.

Let's play with Python sets a bit.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
myset = {"Geeks", "for", "Geeks"}
print(type(myset))  # type of variable
print(myset)        # note no duplicates allowed!
print(len(myset))   # cardinality of the set
```

```
<class 'set'>
{'for', 'Geeks'}
2
```

Question: Basic Set Operations in Python

We are given the sets $A = \{2,3,4,5,6\}$, $B = \{4,5,6,7,8\}$, and $C = \{3,6,8\}$.

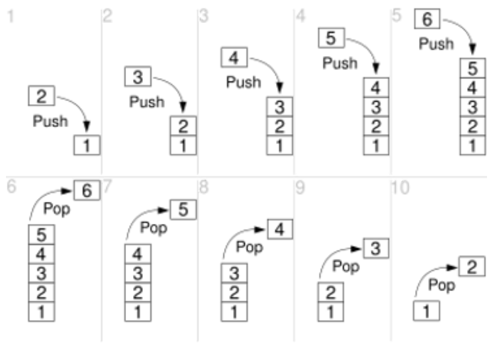
1. Define these sets in Python.
2. Find $A \cup B$, $A \cap B$, $A \setminus B$, and cardinalities of these.
3. Try a few of the basic set methods but don't lose to much time on this.
4. [*Optional*] If you wish you can visualize these using the nice Matplotlib-venn library. But first, you need to install it. `!pip install matplotlib-venn`

Answer as text here

In [3]:
```
''' Answer as code here '''
```

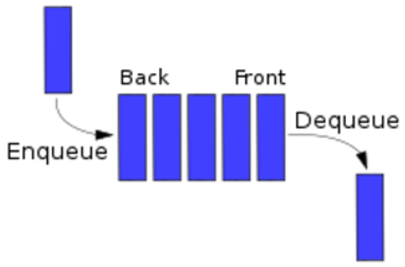
Out[3]:
```
' Answer as code here '
```

Stacks and Queues in Python



Stack is an old datatype, that goes back to the 1940s. In addition to software, stacks are implemented at the hardware level (CPU, memory, controllers) as a means of allocating and accessing memory.

This is a nice tutorial on stacks in Python. Note that you can implement them in 3 different ways! For the following exercise, your choice does not matter. You can use a list or collections.deque



A **queue** is an ordered collection of items. Queues are widely used in engineering and IT and there is the entire mathematical field of *queueing theory* which studies queues and their properties.

As in the case of stacks, there are multiple options for implementing queues in Python. However, there is a nice queue class already built in.

Question: Balanced Parentheses using Stack Data Type

Solve the balanced parenthesis checker problem discussed in the lectures. The goal is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. Specifically and appropriately, we will use the stack data type for this purpose.

1. Implement a basic balanced parenthesis "(" checker using **stack** in Python implemented either using built-in list or collections.deque data types.
2. Extend your checker to all types of parentheses "(", "[", "{".

Hint: see this reference for inspiration but be careful; that book uses very old/obsolete Python sometimes.

Answer as text here

In [4]:
```
''' Answer as code here '''
```

Out[4]:
```
' Answer as code here '
```

Question: Implementing a Simple Printing Queue

To gain a bit of experience with queues, implement and simulate a printing queue as described here. However, you don't need to follow what is written there exactly and we encourage you to **keep things simple** (remember queueing theory is a big topic)!

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.
2. Write a function for generating print tasks.
3. Write a function for printer behaviour (how long it takes to print pages and add a bit of randomness to it)
4. Simulate and print the behaviour of the system (as in the example).
5. What happens when the printer is slow/fast? What happens if there are too many tasks generated in a short interval of time? Very briefly discuss.

Answer as text here

In [5]:
```
''' Answer as code here '''
```

Out[5]:
```
' Answer as code here '
```

Vectors and Matrices

Vectors and matrices are mathematical data types that play a crucial role in modern engineering and machine learning. Numpy supplements Python nicely in linear algebra. It has many capabilities comparable to Matlab that you know well.

Question: Linear Algebra with NumPy

This is a quick exercise to refresh linear algebra skills using NumPy.

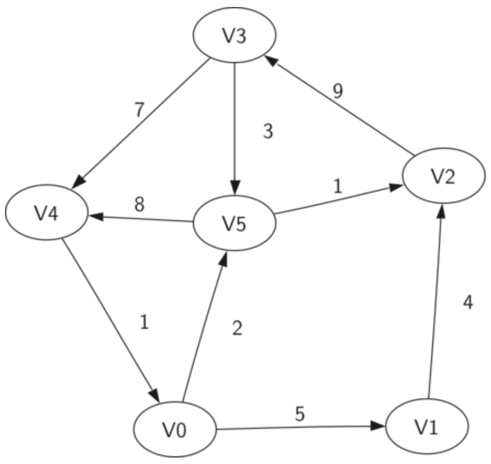
1. Define a 3×3 matrix A.
2. Extract a row and column vector from A using slicing.
3. Multiply A with a 3-dimensional vector from left and right.
4. Find the inverse, transpose, trace, rank, condition number, and determinant of A.
5. Find various norms of A.
6. Find the eigenvalues and vectors of A.
7. Define a 3-dimensional vector b and solve $Ax = b$ for x.

Answer as text here

In [6]:
```
''' Answer as code here '''
```

Out[6]:
```
' Answer as code here '
```

Graphs and Graph Theory

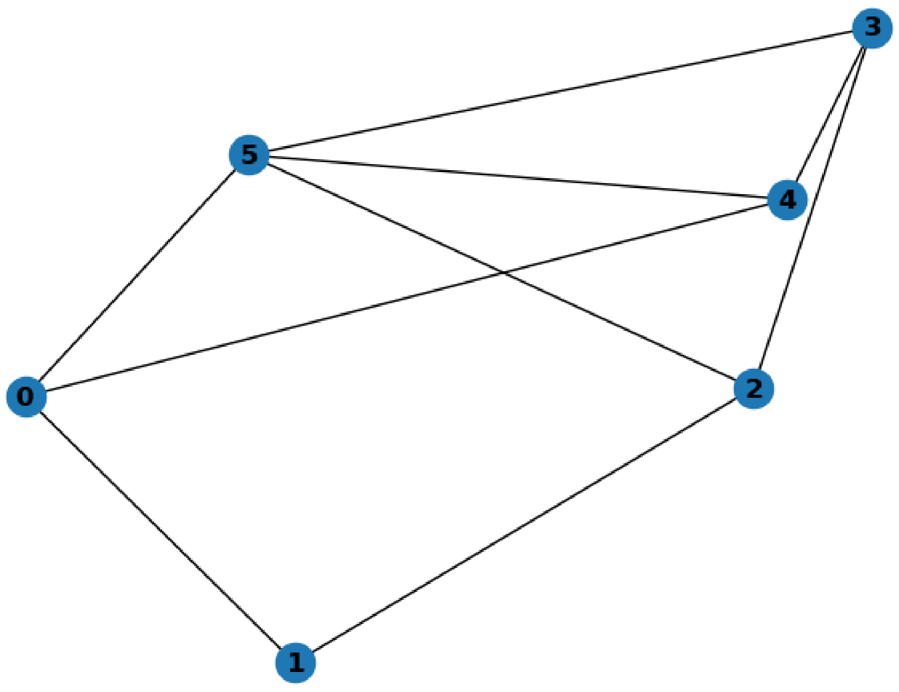


Graph data type implements the mathematical definition of a graph from the field of graph theory in mathematics! A basic definition of a graph is the tuple $G = (V, E)$, where V is the set of vertices and E is a set of edges. Each edge is a tuple (v, w) where $v, w \in V$ are vertices.

NetworkX is a powerful and widely used library in Python for the study and manipulation of graphs and networks. Have a quick look at its tutorial to get an idea of its capabilities. It is possible to write your own graph functions as shown here or here, but this is often unnecessary.

Let's create the graph shown above as an **undirected** graph (no arrows on edges). There are multiple ways of doing this.

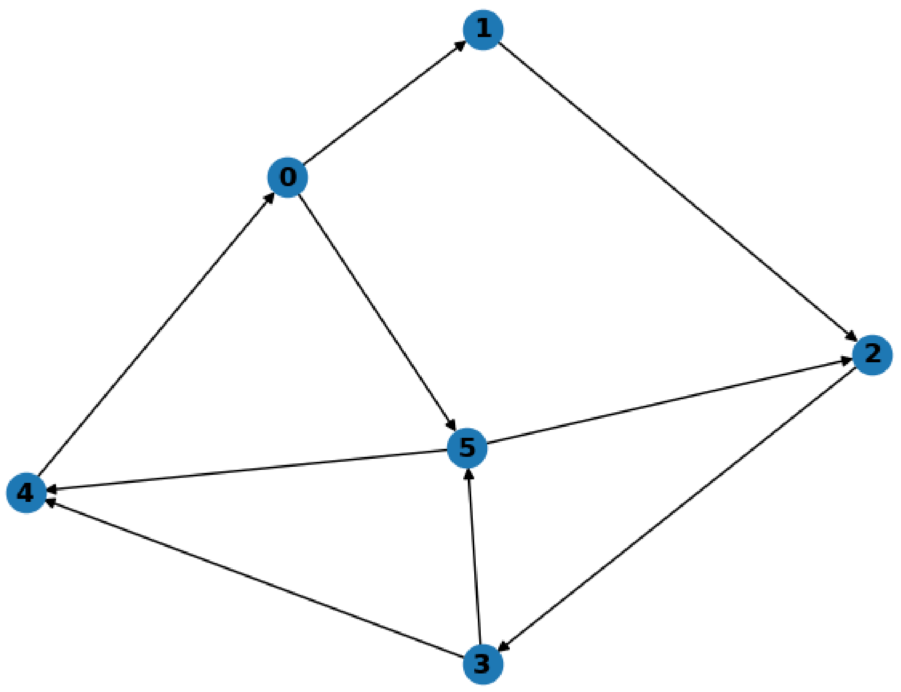
```
In [4]: import networkx as nx
        # G = nx.Graph([(0,1),(0,5),(1,2),(2,3),(3,4),(3,5),(4,0),(4,5),(5,2),(5,4)])    # from edgelist
        G = nx.Graph({0: [1,5], 1: [0, 2], 2: [1, 3, 5], 3: [2,4,5], 4: [0,3,5], 5: [0,2,3,4]})  # from dictiona
        G.nodes()

Out[4]: NodeView((0, 1, 2, 3, 4, 5))

In [5]: G.edges()

Out[5]: EdgeView([(0, 1), (0, 5), (0, 4), (1, 2), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5)])

In [6]: adj_mat = nx.adjacency_matrix(G).toarray()  # the adjacency matrix showing connections (ordered by nodel
        adj_mat

Out[6]: array([[0, 1, 0, 0, 1, 1],
               [1, 0, 1, 0, 0, 0],
               [0, 1, 0, 1, 0, 1],
               [0, 0, 1, 0, 1, 1],
               [1, 0, 0, 1, 0, 1],
               [1, 0, 1, 1, 1, 0]])

In [7]: # We could have created the graph directly from a adjacency matrix
        # G = nx.Graph(adj_mat)

In [8]: nx.draw(G, with_labels=True, font_weight='bold')
```



```
In [9]: # let's create a directed version of the graph exactly as shown in image
        H = nx.DiGraph({0: [1,5], 1: [2], 2: [3], 3: [4,5], 4: [0], 5: [2,4]})  # from dictionary
        nx.adjacency_matrix(H).toarray()
```

```
Out[9]:  array([[0, 1, 0, 0, 0, 1],
                [0, 0, 1, 0, 0, 0],
                [0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 1, 1],
                [1, 0, 0, 0, 0, 0],
                [0, 0, 1, 0, 1, 0]])
```

```
In [10]: nx.draw(H, with_labels=True, font_weight='bold')
```



Question: Simple Graph Analysis using NetworkX

Define a new small-sized (directed or undirected) graph.

1. Find the degrees of the nodes.
2. Find the Laplacian matrix of the graph.
3. Check if it is directed or not.
4. Find the minimum spanning tree of the graph.
5. Find the shortest path between two nodes.

Answer as text here

```
In [14]: ''' Answer as code here '''
```

```
Out[14]: ' Answer as code here '
```

Tree Data Type

A **Tree**) is a special case of a graph but is important in its own right and widely used. Each node in the tree can be connected to many children (depending on the type of tree) but must be connected to exactly one parent. The root (top-most) node of the tree has no parent by definition. Thus, there are no cycles or "loops" in trees and each child can be treated like the root node of its own subtree. Due to these properties, trees are relevant to recursion algorithms and vice versa.

There is no built-in Tree library in Python but it is not difficult to create one. However, to save time, we introduce the anytree library which provides a simple, lightweight, and extensible Tree data structure.

```
In [11]: !pip install anytree
```

In [12]:
```python
from anytree import Node, RenderTree
udo = Node("Udo")
marc = Node("Marc", parent=udo)
lian = Node("Lian", parent=marc)
dan = Node("Dan", parent=udo)
jet = Node("Jet", parent=dan)

print(RenderTree(udo))
```

```
Node('/Udo')
├── Node('/Udo/Marc')
│   └── Node('/Udo/Marc/Lian')
└── Node('/Udo/Dan')
    └── Node('/Udo/Dan/Jet')
```

In [13]:
```python
marc.new_property = "awesome"
print(RenderTree(marc))
```

```
Node('/Udo/Marc', new_property='awesome')
└── Node('/Udo/Marc/Lian')
```

Algorithms

Algorithms are the heart of computing and behind every reasonable computer program. As a simple definition: "An algorithm is a procedure to accomplish a specific task by taking inputs and transforming them into the desired output." Likewise simplified, *engineering* is about the application of scientific principles to design or build desired things. There is a close connection between algorithms, computing, and engineering, especially in the 21st century. *(note that, it is not that simple. There are multiple definitions of engineering and computer scientists still debate what an algorithm is!)*

Recursion

We will start with one of the most common techniques used in designing algorithms: **recursion**. Recursive algorithms adopt a **reduction** or **divide-and-conquer** approach. Whatever the name used, the underlying principle is the same: break down a given (difficult) problem into smaller, more easily solved pieces. The idea is basically "simplify and delegate". Reducing one problem X to another problem Y means writing an algorithm for X that uses an algorithm for Y as a (black box) subroutine. A nice informal way of describing recursion (to solve a problem) is:

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more simpler instances of the same problem. Note that, this approach is also closely related to induction in mathematics.

Example: Binary Search

A **Binary Search Algorithm** is a searching algorithm (to find a number x in a **sorted** array $Y = [y_0, y_1, \ldots, y_N]$), which repeatedly divides the search interval in half until it finds the target value or the interval is empty. This is a nice simple problem and algorithm for demonstrating the basic principles of recursion and divide-and-conquer approach.

In [14]:
```python
# It returns location of x in given array arr
def binarySearch(arr, low, high, x):

    while low <= high:

        mid = low + (high - low) // 2

        # Check if x is present at mid
        if arr[mid] == x:
            return mid

        # If x is greater, ignore left half
        elif arr[mid] < x:
            low = mid + 1

        # If x is smaller, ignore right half
        else:
            high = mid - 1

    # If we reach here, then the element was not present
    return False
```

In [15]:
```
np.random.seed(126548451)
Y = np.random.randint(0,100, size=10) # random array of integers in [0,30]
Y = np.sort(np.unique(Y))             # remove duplicates and sort
#print(Y)                             # search domain

x = 4  # searching for x in Y
result = binarySearch(Y, 0, len(Y)-1, x)
if result:
    print("Element ", x," is present at index", result," of the given array", Y)
else:
    print("Element ", x," is not present in the given array ", Y)
```

```
Element  4  is present at index 1  of the given array [ 2  4  6 15 24 30 45 79 82]
```

As an **exercise**, you can try different random arrays to explore how this algorithm works.

Example: Fibonacci Numbers

The **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones. It is historically one of the most famous sequences due to its long history and recursive nature.

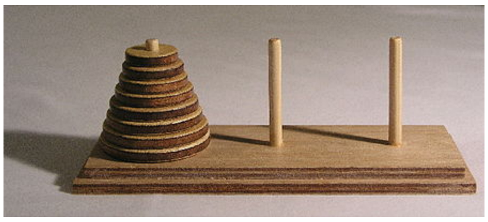
The definition is simple: $F_0 = 0$, $F_1 = 1$, and for $n > 1$, $F_n = F_{n-1} + F_{n-2}$, which yields $[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots]$. The code below is from this nice tutorial.

In [16]:
```
# basic version without memoisation
def fibonacci_of(n):
    if n in {0, 1}:  # Base case
        return n
    return fibonacci_of(n - 1) + fibonacci_of(n - 2)  # Recursive case
```

In [17]:
```
[fibonacci_of(n) for n in range(15)]
```

Out[17]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Example: Tower of Hanoi



Tower of Hanoi is a famous puzzle that has been used to teach recursion to students for a long long time. To be honest, the recursive solution looks very much like magic, probably because it may be counter-intuitive to some.

There are many solutions online such as this, another one or yet another one. Here is a solution that keeps track of the state of the system.

In [18]:
```
# Recursive Python function to solve the Tower of Hanoi

# System state definition
# Rods are: A left, B middle, C right. A is source, C is destination. Smaller disks have a smaller numbe
state = {'A': [] , 'B': [], 'C' : []}

# Recursive DP algorithm
def TowerOfHanoi(n , source, destination, auxiliary):
    ## Trivial case
    if n==1:
        # update state
        disk = state[source].pop()
        state[destination].append(disk)
        print ("Move disk 1 from source",source,"to destination",destination, state)
        return
    ## STEP 1 of Algorithm
    TowerOfHanoi(n-1, source, auxiliary, destination)
    ## STEP 2 of Algorithm
    # update state
    disk = state[source].pop()
    state[destination].append(disk)
    print ("Move disk",n,"from source",source,"to destination",destination, state)
    ## STEP 3 of Algorithm
    TowerOfHanoi(n-1, auxiliary, destination, source)
```

In [19]:
```
# Simulation
n = 3  # number of disks

# initial state setup
state = {'A': list(range(n,0,-1)), 'B': [], 'C' : []} # all disks in A, small disks have smaller numbers
print("Initial state of the tower ",state)

# solution
TowerOfHanoi(n,'A','C','B')
```

```
Initial state of the tower  {'A': [3, 2, 1], 'B': [], 'C': []}
Move disk 1 from source A to destination C {'A': [3, 2], 'B': [], 'C': [1]}
Move disk 2 from source A to destination B {'A': [3], 'B': [2], 'C': [1]}
Move disk 1 from source C to destination B {'A': [3], 'B': [2, 1], 'C': []}
Move disk 3 from source A to destination C {'A': [], 'B': [2, 1], 'C': [3]}
Move disk 1 from source B to destination A {'A': [1], 'B': [2], 'C': [3]}
Move disk 2 from source B to destination C {'A': [1], 'B': [], 'C': [3, 2]}
Move disk 1 from source A to destination C {'A': [], 'B': [], 'C': [3, 2, 1]}
```

Solution Analysis

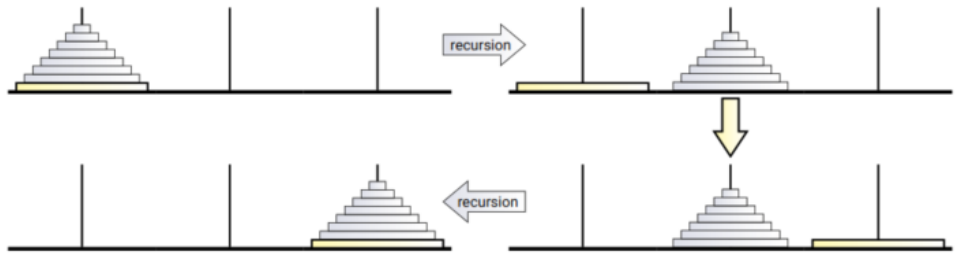
It is like magic, isn't it? Try starting from $n = 1$, and try increasing numbers of n. However, this may not be the best way to understand this. In fact, according to Prof Jeff Erickson (author of Algorithms book), this is not advised:

> It may be tempting to think about how all those smaller disks move around or more generally, what happens when the recursion is unrolled — but really, don't do it. For most recursive algorithms, unrolling the recursion is neither necessary nor helpful. Our only task is to reduce the problem instance we're given to one or more simpler instances or to solve the problem directly if such a reduction is impossible.

> The secret to solving this puzzle is to *think recursively*. Instead of trying to solve the entire puzzle at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are in the way. So first we have to move those n-1 smaller disks to the spare peg **(Step 1)**. Once that's done, we can move the largest disk directly to its destination **(Step 2)**. Finally, to finish the puzzle, we have to move the n-1 smaller disks from the spare peg to their destination **(Step 3)**. *That's it! We're done!* *We've successfully reduced the n-disk Tower of Hanoi

Maybe a better way to understand the solution algorithm is: problem to two instances of the (n-1)-disk Tower of Hanoi problem!*

The figure below summarises this line of thinking, followed by the simulation steps in the following. Note how the main steps of the algorithm unroll!



```
Initial state of the tower  {'A': [3, 2, 1], 'B': [], 'C': []}
Move disk 1 from source A to destination C {'A': [3, 2], 'B': [], 'C': [1]}
Move disk 2 from source A to destination B {'A': [3], 'B': [2], 'C': [1]}     1
Move disk 1 from source C to destination B {'A': [3], 'B': [2, 1], 'C': []}
Move disk 3 from source A to destination C {'A': [], 'B': [2, 1], 'C': [3]}    2
Move disk 1 from source B to destination A {'A': [1], 'B': [2], 'C': [3]}
Move disk 2 from source B to destination C {'A': [1], 'B': [], 'C': [3, 2]}    3
Move disk 1 from source A to destination C {'A': [], 'B': [], 'C': [3, 2, 1]}
```

Optional Reading: why do some people find recursion counter-intuitive?

Anecdotal evidence from (Computer Science) education indicates that students often find the concept of recursive programming difficult. I had serious difficulty understanding dynamic programming myself. Maybe it is very easy for some of you (especially if are used to thinking recursively).

Here are a few speculative ideas on why some people may find recursive thinking difficult and unintuitive:

- It almost seems like circular reasoning.
- When we give instructions to people or computers, we rarely direct them recursively. For example, imperative programming languages (including Python as well as Assembly) are mainly a series of instructions. Moreover, there is no recursion at the CPU level.
- When we solve problems we usually solve them step by step, iteratively, rather than recursively. Most recipes/tutorials are iterative (repeat ... until, while ... do).
- As the puzzle above shows, it is hard to keep track of how a recursive algorithm changes the system state (or works). Unrolling recursion requires the mental construction of a large tree rather than a simple loop (and is not advised). Maybe because of this recursive algorithms are to interpret intuitively.
- It is hard to describe the solution using the system state, which would require unrolling.

What do you think?

Question: Find the Powerset a given Set

The powerset of a set S is defined as *the set of all subsets of S, including the empty set and S itself.*

For a given set (of numbers for convenience), find the powerset using a **recursive** algorithm. There is an iterative solution but that is not what we want here.

Hints:

1. Use list data type as input and in your code.
2. The Wikipedia article provides the algorithm you need!

In [23]:
```python
np.random.seed(1252334531)
Sarr = np.random.randint(0, 100, size=4) # random array of integers in [0,100]
Sarr = np.sort(np.unique(Sarr))
S = [str(x) for x in Sarr]
S
```

Out[23]:
```
['16', '48', '74', '86']
```

Answer as text here

In [1]:
```python
''' Answer as code here '''
```

Out[1]:
```
' Answer as code here '
```

In [ ]:
```python
```

Sorting

Sorting algorithms provide a nice introduction to divide-and-conquer and recursion approaches, which are fundamental to many other algorithms. Unsurprisingly, almost every introductory algorithm book focuses on them.

Sorting problem: Take a (finite) sequence of elements as *input* and order them at the *output* according to a rule, e.g. from small to larger.

The general divide and conquer algorithm pattern:

1. Divide the given instance of the problem into several independent smaller instances of the same problem.
2. Conquer by applying the recursive solution to each smaller/simpler instance.
3. Combine the solutions for the smaller instances into the final solution.

Question: Mergesort Algorithm

Implement the well-known mergesort algorithm. The steps of the algorithm are:

> - Divide the unsorted array into two sub-arrays, half the size of the original.
> - Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
> - Merge two sub-arrays by always putting the lowest value first.
> - Keep merging until there are no sub-arrays left.

Optional Question: Quicksort Algorithm

Implement the well-known quicksort algorithm. The steps of the algorithm are:

> - Choose a value in the array to be the pivot element.
> - Order the rest of the array so that lower values than the pivot element are on the left, and higher values are on the right.
> - Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and higher values.
> - Do the same operations (recursively) for the sub-arrays on the left and right sides of the pivot element.

Refer to Chapter 1 of the Algorithms book for both in addition to numerous online resources.

Answer as text here

```
In [2]: ''' Answer as code here '''
```

```
Out[2]: ' Answer as code here '
```

```
In [ ]:
```

Dynamic Programming

Dynamic Programming (DP) is a method for designing algorithms. DP gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency). By storing the consequences of all possible decisions and using this information in a systematic way, the total amount of work is minimized. In a nutshell, dynamic programming is recursion without repetition.

Dynamic programming efficiently implements a recursive algorithm by storing partial results. Hence, DP (via Memoization) essentially trades off space for time.

Once you understand it, DP is probably the easiest algorithm design technique to apply in practice. Until you understand it, DP seems like magic.

Example: Fibonacci numbers revisited

Let's try compute Fibonacci numbers with memoisation to improve efficiency. Then, we look at the iterative version with tabulation.

In [3]:
```
# Fibonacci with memoisation
cache = {0: 0, 1: 1}   # remember dictionaries are (now) ordered

def fibonacci_of_wm(n):
    if n in cache:  # Base case
        return cache[n]
    # Compute and cache the Fibonacci number
    cache[n] = fibonacci_of_wm(n - 1) + fibonacci_of_wm(n - 2)  # Recursive case
    return cache[n]
```

In [4]:
```
[fibonacci_of_wm(n) for n in range(15)]
```

Out[4]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

In [5]:
```
print(cache)
```

```
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55, 11: 89, 12: 144, 13: 233, 14: 37
7}
```

In [6]:
```
# Fibonacci iterative with tabulation version without recursion
def fibonacci_of_tab(n):

    # Handle the base cases
    if n in {0, 1}:
        return n

    previous, fib_number = 0, 1
    for _ in range(2, n + 1):
        # Compute the next Fibonacci number, remember the previous one
        previous, fib_number = fib_number, previous + fib_number
        # note that there are no recursive calls to the function in this version!

    return fib_number
```

In [7]:
```
[fibonacci_of_tab(n) for n in range(15)]
```

Out[7]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

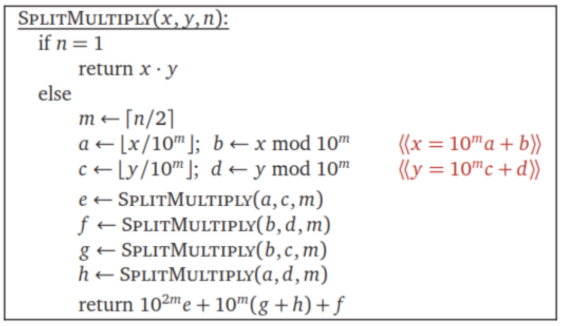
Question: Fast Multiplication

It is possible to improve efficiency of multiplication compared to the classical (goes back to antiquity) version, also known as long multiplication, you learned in primary school. You can read here why speeding up multiplication matters and recent efforts on this.

The idea for speeding it up is based on a divide-and-conquer approach as shown in this formula:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd.$$

The resulting algorithm (from Chapter 1 in Algoiorthms book) is shown here in pseudo-code:



```
SPLITMULTIPLY(x, y, n):
  if $n = 1$
      return $x \cdot y$
  else
      $m \leftarrow \lceil n/2 \rceil$
      $a \leftarrow \lfloor x/10^m \rfloor$;  $b \leftarrow x \bmod 10^m$        $\langle\langle x = 10^m a + b \rangle\rangle$
      $c \leftarrow \lfloor y/10^m \rfloor$;  $d \leftarrow y \bmod 10^m$        $\langle\langle y = 10^m c + d \rangle\rangle$
      $e \leftarrow$ SPLITMULTIPLY($a, c, m$)
      $f \leftarrow$ SPLITMULTIPLY($b, d, m$)
      $g \leftarrow$ SPLITMULTIPLY($b, c, m$)
      $h \leftarrow$ SPLITMULTIPLY($a, d, m$)
      return $10^{2m} e + 10^m (g + h) + f$
```

1. Implement split multiply in Python using a recursive function.

2. An improvement on this is known as Karatsuba algorithm, which requires only three multiplications. Read about it from the given link and implement this more efficient version.

Answer as text here

In [8]:
```
''' Answer as code here '''
```

Out[8]:
```
' Answer as code here '
```

In [ ]:
```
```

Example: Knapsack Problem

$4
12kg
$2
2kg
15 kg
$2
1kg
$10
4kg
$1
1kg

The knapsack problem is a well-known canonical problem in combinatorial optimisation formulated as

> Given a set of N items, each with a weight w and a value v, determine which items to include in the collection (bag) so that the total weight, $\sum_i w_i x_i$, is less than or equal to a given limit (bag capacity W) and the total value, $\sum_i v_i x_i$, is as large as possible (maximised).

The 0-1 knapsack problem can be formally stated as

$$\max_x \sum_{i=1}^{N} v_i x_i \quad \text{s.t.} \quad \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{w_i x_i} \leq \mathrm{W}, \ \mathrm{x_i} \in \{0,1\} \ \forall \mathrm{i}.$$

A recursive, brute-force solution to the problem is given below. Another implementation is here.

The brute force algorithm steps are:

> 1. Consider each item one at a time.
>   A. If there is capacity left for the current item, add it by adding its value and reducing the remaining capacity with its weight. Then call the function on itself for the next item.
>   B. Also, try not adding the current item before calling the function on itself for the next item.
> 2. Return the maximum value from the two scenarios above (adding the current item, or not adding it).

In [9]:
```
def knapsack_brute_force(capacity, n):
    print(f"knapsack_brute_force({capacity},{n})")
    if n == 0 or capacity == 0:
        return 0

    elif weights[n-1] > capacity:
        return knapsack_brute_force(capacity, n-1)

    else:
        include_item = values[n-1] + knapsack_brute_force(capacity-weights[n-1], n-1)
        exclude_item = knapsack_brute_force(capacity, n-1)
        return max(include_item, exclude_item)

values = [300, 200, 400, 500]
weights = [2, 1, 5, 3]
capacity = 10
n = len(values)
```

```
print("\nMaximum value in Knapsack =", knapsack_brute_force(capacity, n))
```

```
knapsack_brute_force(10,4)
knapsack_brute_force(7,3)
knapsack_brute_force(2,2)
knapsack_brute_force(1,1)
knapsack_brute_force(1,0)
knapsack_brute_force(2,1)
knapsack_brute_force(0,0)
knapsack_brute_force(2,0)
knapsack_brute_force(7,2)
knapsack_brute_force(6,1)
knapsack_brute_force(4,0)
knapsack_brute_force(6,0)
knapsack_brute_force(7,1)
knapsack_brute_force(5,0)
knapsack_brute_force(7,0)
knapsack_brute_force(10,3)
knapsack_brute_force(5,2)
knapsack_brute_force(4,1)
knapsack_brute_force(2,0)
knapsack_brute_force(4,0)
knapsack_brute_force(5,1)
knapsack_brute_force(3,0)
knapsack_brute_force(5,0)
knapsack_brute_force(10,2)
knapsack_brute_force(9,1)
knapsack_brute_force(7,0)
knapsack_brute_force(9,0)
knapsack_brute_force(10,1)
knapsack_brute_force(8,0)
knapsack_brute_force(10,0)

Maximum value in Knapsack = 1200
```

Analysis with a tree structure

As in the case of Tower of Hanoi, this output tells us almost nothing. It is much easier to interpret the output if we consider the solution in terms of a tree rather than an iterative loop. After all, this is exactly what the recursive algorithm does! Therefore, we create a tree to store the output of the recursive algorithm.

In [10]:
```
from anytree import Node, RenderTree, Walker

# Use a simple tree to store solutions
root_node = Node("Start")     # the solution tree
root_node.val = 0             # temp variable to store the calculated value
root_node.total_val = 0       # the total value of a solution
root_node.soln_node = []      # the solution node/branch

def knapsack_brute_force(capacity, n, node, verbose):
    if verbose:               # write output or not
        print(f"knapsack_brute_force({capacity},{n})")

    # out of items or capacity
    if n == 0 or capacity == 0:
        return 0

    # if weight of item is more than capacity, skip it
    elif weights[n-1] > capacity:  # n starts from end of list, then n-1, n-2, until 0
        return knapsack_brute_force(capacity, n-1, node, verbose)

    # crate two branches one that includes the next item and one that does not
    else:
        # ------------------------------------------
        # create and store new item node in the tree
        new_node = Node(str((values[n-1], weights[n-1])), parent = node)
        new_node.val = values[n-1] + node.val # cumulative value
        # is this the current solution?
        if new_node.val > root_node.total_val:  # assuming one solution for simplicity here
            root_node.total_val = new_node.val  # if current soln, then store cumulative value
            root_node.soln_node = new_node     # store solution
        # ------------------------------------------
        # Main Algorithm
        # branch that includes the item
        include_item = values[n-1] + knapsack_brute_force(capacity-weights[n-1], n-1, new_node, verbose)
        # branch that does not include the item
        exclude_item = knapsack_brute_force(capacity, n-1, node, verbose)
        # return the final value
        return max(include_item, exclude_item)
```

In [11]:
```
# Problem definition
values = [300, 200, 400, 500]  # v, item values
weights = [2, 1, 5, 3]         # w, item weights
capacity = 10                  # W, bag capacity
n = len(values)                # total number of items
```

In [12]:
```
print("\nMaximum value in Knapsack =", knapsack_brute_force(capacity, n, root_node, False))
```

```
Maximum value in Knapsack = 1200
```

In [13]:
```
print(RenderTree(root_node))
```

```
Node('/Start', soln_node=Node('/Start/(500, 3)/(400, 5)/(300, 2)', val=1200), total_val=1200, val=0)
├── Node('/Start/(500, 3)', val=500)
│   ├── Node('/Start/(500, 3)/(400, 5)', val=900)
│   │   ├── Node('/Start/(500, 3)/(400, 5)/(200, 1)', val=1100)
│   │   └── Node('/Start/(500, 3)/(400, 5)/(300, 2)', val=1200)
│   ├── Node('/Start/(500, 3)/(200, 1)', val=700)
│   │   └── Node('/Start/(500, 3)/(200, 1)/(300, 2)', val=1000)
│   └── Node('/Start/(500, 3)/(300, 2)', val=800)
├── Node('/Start/(400, 5)', val=400)
│   ├── Node('/Start/(400, 5)/(200, 1)', val=600)
│   │   └── Node('/Start/(400, 5)/(200, 1)/(300, 2)', val=900)
│   └── Node('/Start/(400, 5)/(300, 2)', val=700)
├── Node('/Start/(200, 1)', val=200)
│   └── Node('/Start/(200, 1)/(300, 2)', val=500)
└── Node('/Start/(300, 2)', val=300)
```

In [14]:
```
print(root_node.soln_node)
#print(root_node.soln_node.parent)
walk = Walker()
walk.walk(root_node,root_node.soln_node)
```

```
Node('/Start/(500, 3)/(400, 5)/(300, 2)', val=1200)
```

Out[14]:
```
((),
 Node('/Start', soln_node=Node('/Start/(500, 3)/(400, 5)/(300, 2)', val=1200), total_val=1200, val=0),
 (Node('/Start/(500, 3)', val=500),
  Node('/Start/(500, 3)/(400, 5)', val=900),
  Node('/Start/(500, 3)/(400, 5)/(300, 2)', val=1200)))
```

Question: Memoization Approach for 0/1 Knapsack Problem

Extend the brute force algorithm above by creating either (a) an array *memo* or (b) the given tree, to store previous results. To avoid doing the same calculation more than once, every time you call your function, check first if the result is already stored.

Answer as text here

In [41]:
```
''' Answer as code here '''
```

Out[41]:
```
' Answer as code here '
```

In [ ]:

In [ ]:

Workshop Assessment Instructions

You should complete the workshop tasks and answer the questions within the allocated session! **Submission deadline is usually the end of the last week of the workshop. Please check Canvas for the exact deadline!**

It is **mandatory to follow all of the submission guidelines** given below. *Don't forget the Report submission information on top of this notebook!*

1. The completed Jupyter Notebook and its PDF version (you can simply print-preview and then print as PDF from within your browser) should be uploaded to the right place in Canvas. *It is your responsibility to follow the announcements!* **Late submissions will be penalised (up to 100% of the total mark depending on the delay amount)!**
2. Filename should be "ELEN90088 Workshop **W: StudentID1-StudentID2** of session **Day-Time**", where **W** refers to the workshop number, **StudentID1-StudentID2** are your student numbers, **Day-Time** is your session day and time, e.g. *Tue-14*.
3. Answers to questions, simulation results and diagrams should be included in the Jupyter Notebook as text, code, and plots. *If you don't know latex, you can write formulas/text to a paper by hand, scan it and then include it as an image within Markdown cells.*
4. Please submit your report as a group.

Workshop Marking

- Each workshop has a fixed number of points corresponding to a certain percentage of the total subject mark inclusive individual oral examination. You can find the detailed rubrics on Canvas.
- Individual oral quizzes will be scheduled within the next two weeks following the report submission. They will be during workshop hours. Therefore, you must attend the workshops!
- The individual oral examination will assess your answers to workshop questions, what you have done in that workshop, and your knowledge of the subject material in association with the workshop.

Additional guidelines for your programs:

- Write modular code using functions.
- Properly indent your code. But Python forces you to do that anyway ;)
- Heavily comment on the code to describe your implementation and to show your understanding. No comments, no credit!
- Make the code your own! It is encouraged to find and get inspired by online examples but you should exactly understand, modify as needed, and explain your code via comments. If you resort to blind copy/paste, you will certainly not do well in the individual oral quizzes.

In [ ]: