

19. Computational Thinking and Problem-solving

19.1 Algorithms

Searching Algorithms

- Include **linear search** and **binary search**.

Linear Search

- **Time complexity:** $\mathcal{O}(n)$.
 - The time taken goes up **linearly** as the number of items rises **linearly**.
 - Time to search increases **more rapidly** than binary search.
- **Space complexity:** $\mathcal{O}(1)$.

```
def linear_search(array, search_value):  
    for i in array:  
        if i == search_value:  
            return i  
  
    return -1
```

Binary Search

- **Condition:** The list is **ordered**.
- **Time complexity:** $\mathcal{O}(\log n)$.
 - The time complexity is $\mathcal{O}(\log n)$ that uses logarithmic time.
 - The time taken goes up **linearly** as the number of items rises **exponentially**.
 - In the **worst case**, the time complexity is $\mathcal{O}(\log n)$.
- **Space complexity:** $\mathcal{O}(1)$.
 - A constant number of variables are required (*two variables required*).
- **Description:**
 1. Find the **middle index**.
 2. Check the value of the middle item in the list to be searched.
 3. If equal, item searched for is found.
 4. If not equal, discard the **half of the list** that doesn't contain the search item.
 5. Repeat the above steps until the item searched is found
 6. ... or lower bound > upper bound.

```
def binary_search(array, search_value):  
    l, r = 0, len(array) - 1  
    while l <= r: # the left (lower) bound is smaller than the right  
        mid = (l + r) // 2 # use exact division  
  
        if array[mid] == search_value:  
            return mid
```

```

    if array[mid] > search_value: # the value is at left (smaller)
        r = mid - 1
    else: # the value is at right (larger)
        l = mid + 1

return -1

```

Sorting Algorithms

- Include **bubble sort** and **insertion sort**.
- Factors affecting the **performance of sorting algorithms**:
 - The **initial order** of data.
 - The **number of data items** to be stored.
 - The **efficiency** of the sorting algorithm.

Bubble Sort

- **Time complexity**:
 - Worst case: $\mathcal{O}(n^2)$.
 - Best case (on sorted data): $\mathcal{O}(n)$.
- **Space complexity**: $\mathcal{O}(1)$.
- **Description**:
 - Compares **adjacent elements** in the array.
 - They are swapped if the elements are in the wrong order.
 - The algorithm iterates through the array multiple times in passes.
 - On each pass, the largest unsorted element "bubbles up" to its correct position at the end of the array.
 - The process is repeated until the entire array is sorted.

```

def bubble_sort(array):
    is_sorted = False
    passes = 0
    while is_sorted == False:
        is_sorted = True
        for i in range(len(array) - 1 - passes): # most efficient
            if array[i] > array[i+1]:
                array[i], array[i+1] = array[i+1], array[i]
                is_sorted = False
        passes += 1
    return array

```

Insertion Sort

- **Time complexity**:
 - Worst case: $\mathcal{O}(n^2)$.
 - Best case: $\mathcal{O}(n)$.
 - Efficient for **small or partially sorted** datasets.
- **Space complexity**: $\mathcal{O}(1)$.

- **Description:**

- Assumes that the first element in the array is already sorted.
- Compares the next element with the sorted portion of the array.
- If the next element is smaller, it is shifted to the left until the previous element is smaller.
- The sorted portion of the array grows with each iteration until the entire array is sorted.
- The process is repeated until all elements are in their correct positions.

```
def insertion_sort(array):  
    for index in range(1, len(array)):  
        i = index  
        while array[i-1] > array[i] and i > 0:  
            array[i], array[i-1] = array[i-1], array[i]  
            i = i - 1  
    return array
```

Abstract Data Types

- Include **queues**, **stacks**, **linked lists**, and **binary trees**.

Queue

- **Features:**
 - Each queue element contains one data item.
 - A pointer to the front of the queue; A pointer to the end of the queue.
 - Works on **First-In-First-Out**.
 - May be **circular**.
- **Array implementation** → always declare the type + length of variables/arrays.
 - Declare an **array**.
 - Declare **integer variables** for `FrontOfQueuePointer`, `EndOfQueuePointer`, `NumberInQueue`, `SizeOfQueue` (the maximum size of the queue).
 - Initialize `FrontOfQueuePointer` and `EndOfQueuePointer` to represent an empty queue.
 - Initialize `NumberInQueue` to 0 and `SizeOfQueue` to the maximum size.
- 记得先判空再取队头.

Stack

- **Features:**
 - Each stack element contains one data item.
 - A pointer to the top of the stack; A pointer to the bottom of the stack.
 - Works on **First-In-Last-Out**.
- **Array implementation:**
 - Declare an **array**, the number of elements corresponds to the size of the required stack.

- Declare **integer variables** for `TopOfStack`, `BottomOfStack`, `NumberInStack`, and `SizeOfStack`.
- Initialize pointers and variables to indicate an empty stack.
- Attempt to describe **Push** and **Pop** operations.
- Pop and Push routines need to check for **full** or **empty** conditions.

Linked List

- **Features:**
 - Each node contains data and a pointer to the next node.
 - A Pointer to the start of the list.
 - The last node in the list has a null pointer.
 - Data may be added / removed by manipulating pointers.
 - Nodes are traversed in a specific sequence.
 - Unused nodes are stored on a free list.
- **Array implementation:**
 - Define a **record** type with fields for data and the pointer.
 - Declare **one** array of the defined record type.
 - Declare **integer variables** for `StartPointer`, `NextFreePointer`.
 - Declare an **integer value** for `NullPointer`.
 - Routines are needed to add/delete/search.
- **Insert an item:**
 - Check for a free node.
 - Search for the correct insertion point.
 - Assign data value to the node pointed by the start pointer of the free list.
 - Change of pointers of related nodes.
 - The start pointer in the free list moved to the point of the next free node.
- **Advantage:** pointers determine the ordering of data → easier to add/delete data.
 - Has a **variable size**, as compared to a string.
- **Disadvantage:** need to store pointers as well as data → more complex.

Binary Tree

- **Features:**
 - **Dynamic data structure.**
 - Consists of nodes arranged hierarchically, starting with a root node.
 - A node can have **no more than two** descendants.

19.2 Recursion

- **Recursion** is a process using a function / procedure that is **defined in terms of itself** and **calls itself**.
- A recursive process must have a **base case** (a way to return without making a recursive call).
- There must also be a **general case** where the recursive call takes place.

Code Implementation of Recursion

- Below are Pseudocode and Python implementations of the factorial function.

```
FUNCTION Factorial (Number: INTEGER) RETURNS INTEGER
  DEFINE Answer: INTEGER
  IF Number = 0 THEN
    Answer <- 1
  ELSE
    Answer <- Number * Factorial(Number - 1) // func calls itself
  ENDIF
  RETURN Answer
ENDFUNCTION
```

```
def factorial(num):
    if num == 0:
        return 1 # a simplified way to avoid defining a new variable
    else:
        return num * factorial(num-1) # recursive call
```

Using Stack to Implement Recursion

- A stack is a **Last-In-First-Out** data structure.
- Each recursive call is pushed onto the stack
- ... And is then popped as the function ends.
- Enables backtracking/**unwinding**
- ... To maintain the required order.

Winding and unwinding

- Winding** is when recursive calls have been made but **not yet returning**.
 - Each call adds a **new frame** to the call stack.
- Unwinding** is when **recursive calls return**, moving back up the call stack.
 - This happens after the **base case** is hit.
- E.g., when calling `Factorial(3)` using the function implemented above.

Call number	Function call	number	answer	RETURN
1	Factorial (3)	3	3 * factorial (2)	
2	Factorial (2)	2	2 * factorial (1)	
3	Factorial (1)	1	1 * factorial (0)	
4	Factorial (0)	0	1	1
3 continued	Factorial (1)	1	1 * 1	1
2 continued	Factorial (2)	2	2 * 1	2
1 continued	Factorial (3)	3	3 * 2	6

winding

base case

unwinding

Benefits of Recursion

- Cleanness.**
 - Recursive solutions can contain fewer programming statements than an iterative solutions.
- Decomposition.**

- Recursion reduces complex problems into **smaller subproblems**.
- **Natural expression.**
 - **Divide-and-conquer** algorithms like binary search are inherently recursive; they are more naturally expressed recursively.

Drawbacks of Recursion

- **Stack overflow risk.**
 - Each recursive call uses stack memory. Deep recursion may exceed the **call stack limit** and crash the program.
- **Overhead from function calls.**
 - Recursive calls involve **pushing and popping frames** on the call stack.
 - Iterative solutions often run **faster** because of less function call overhead.
- **Harder to debug.**
 - Recursive logic can be harder to trace and debug, especially when the recursion depth grows or base cases are mishandled.