Lecture 6: Designing and Analyzing Algorithms // Developing and Testing with Unit Tests // Version Control with GitHub // Tips for Final Project

Chiheb Ben Hammouda

Mathematical Institute, Utrecht University

Python and R Course (WISB153)

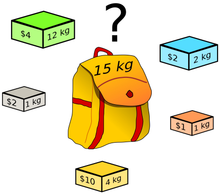
May 27, 2025

Course Progress Overview

1. Week 1:
   - Thinking like a Computer, Computational thinking process
   - Introduction to Python (variables, expressions, functions, basic syntax, input and output).
2. Week 2:
   - Complexity notion: time/space complexity;
   - Big-O asymptotic: notation and definition; worst/best/average complexity;
   - Examples of complexity analysis;
   - conditionals (if); functions, python modules; exceptions, debugging.
3. Week 3:
   - Data structures (Array, List, queue, stack, dictionary)
   - Iterative algorithms examples (Prime sieve, root) and complexity
   - Loops/repetition (for, while)
4. Week 4:
   - Recursive algorithms: examples, general form of recursive algorithm, and the master theorem
   - Applications of the Master Thm
5. Week 5:
   - Iterative vs Recursive- Memoization
   - Object-Oriented Programming

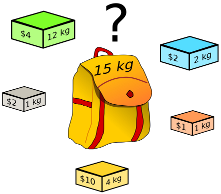
Plan of Lecture 6

Knapsack Problem



- The values $\{v_i\}_{i=0}^{n-1}$ and weights $\{w_i\}_{i=0}^{n-1}$ of a collection of n items are given.
- **Aim:** Select a subset of items with the maximum/minimum combined value and a combined weight under or equal to a given limit, W.

Knapsack Problem



- The values $\{v_i\}_{i=0}^{n-1}$ and weights $\{w_i\}_{i=0}^{n-1}$ of a collection of n items are given.
- **Aim:** Select a subset of items with the maximum/minimum combined value and a combined weight under or equal to a given limit, W.
- **Applications:**
  - Resource Allocation: each project represents an item, the cost of the project represents the weight, and the expected return is the value.
  - Scheduling of jobs on a single machine: each job represents an item, the processing time of the job represents the weight, and the number of jobs completed before their deadline is the value.
  - Selection of investments and portfolios.

Mathematical Formulation

- The state space (feasible set) $\mathcal{S}$ consists of all possible subsets $\mathcal{K}$ of $\{0, 1, \ldots, n-1\}$ such that (they satisfy the constraint)

$$\sum_{i \in \mathcal{K}} w_i \leq W.$$

- We can represent the states as binary numbers $t = (t_0, t_1, \ldots, t_{n-1})$ where

$$t_i = \begin{cases} 1 & \text{if } i \in \mathcal{K}, \\ 0 & \text{if } i \notin \mathcal{K}. \end{cases}$$

- The number of possible states is

Mathematical Formulation

- The state space (feasible set) $\mathcal{S}$ consists of all possible subsets $\mathcal{K}$ of $\{0, 1, \ldots, n-1\}$ such that (they satisfy the constraint)
$$\sum_{i \in \mathcal{K}} w_i \leq W.$$
- We can represent the states as binary numbers $t = (t_0, t_1, \ldots, t_{n-1})$ where
$$t_i = \begin{cases} 1 & \text{if } i \in \mathcal{K}, \\ 0 & \text{if } i \notin \mathcal{K}. \end{cases}$$
- The number of possible states is 2^n.
- The objective function c assigns a value to a given state t
$$c(t) = \sum_{i=0}^{n-1} t_i v_i = \sum_{i \in \mathcal{K}} v_i.$$

Mathematical Formulation

- The state space (feasible set) $\mathcal{S}$ consists of all possible subsets $\mathcal{K}$ of $\{0, 1, \ldots, n-1\}$ such that (they satisfy the constraint)

$$\sum_{i \in \mathcal{K}} w_i \leq W.$$

- We can represent the states as binary numbers $t = (t_0, t_1, \ldots, t_{n-1})$ where

$$t_i = \begin{cases} 1 & \text{if } i \in \mathcal{K}, \\ 0 & \text{if } i \notin \mathcal{K}. \end{cases}$$

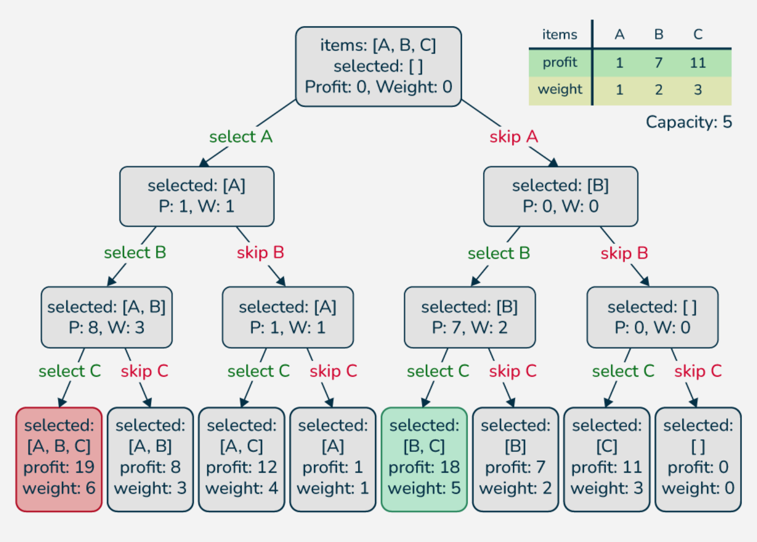
- The number of possible states is 2^n.
- The objective function c assigns a value to a given state t

$$c(t) = \sum_{i=0}^{n-1} t_i v_i = \sum_{i \in \mathcal{K}} v_i.$$

- **Problem**: find a state $t \in \mathcal{S}$ for which $c(t)$ is as small or as large as possible.

Method 1: Enumeration/Brute-force

The problem can be solved through enumeration: generate all possible states and choose the best one.



Generating all possible combinations from a list in Python

```
from itertools import combinations

# Example list of 3 items
items = ['A', 'B', 'C']

# Generate all possible combinations of the list
for r in range(1, len(items) + 1):
    comb = combinations(items, r)
    for c in comb:
        print(c)
```

Output:

```
()
('A',)
('B',)
('C',)
('A', 'B')
('A', 'C')
('B', 'C')
('A', 'B', 'C')
```

Method 1: Enumeration/Brute-force for Knapsack in Python (Part 1)

```
from itertools import combinations

# Define the items and their properties
items = {
    'A': {'value': 1, 'weight': 1},
    'B': {'value': 7, 'weight': 2},
    'C': {'value': 11, 'weight': 3},
}
# Define the capacity of the knapsack
capacity = 5
```

Method 1: Enumeration/Brute-force for Knapsack in Python (Part 2)

```python
def knapsack_brute_force(items, capacity):
    best_value = 0
    best_weight = 0
    best_combination = []

    # Get all items as a list of keys
    item_keys = list(items.keys())

    # Check all possible combinations of items from 1 item to all items
    for i in range(1, len(items) + 1):
        for combo in combinations(item_keys, i):
            total_weight = sum(items[item]['weight'] for item in combo)
            total_value = sum(items[item]['value'] for item in combo)

            # Check if this combination is better than what we have found before
            # and if it fits in the knapsack
            if total_weight <= capacity and total_value > best_value:
                best_value = total_value
                best_weight = total_weight
                best_combination = combo

    return best_combination, best_value, best_weight

# Get the best combination and its value
best_combination, best_value, best_weight = knapsack_brute_force(items, capacity)
print("Best combination:", best_combination)
print("Best value:", best_value)
print("Weight for best combination:", best_weight)
```

Time Complexity of Brute Force Knapsack Solution

Time Complexity of Brute Force Knapsack Solution

- Each item can either be included or excluded, leading to a total of 2^n possible combinations, where n is the number of items.

Time Complexity of Brute Force Knapsack Solution

- Each item can either be included or excluded, leading to a total of 2^n possible combinations, where n is the number of items.
- For each of the 2^n combinations, operations include:
  - Calculating the total weight and total value of the selected items.
  - Comparing the current combination's total value with the best value found so far.
  - Checking if the total weight exceeds the capacity.

Time Complexity of Brute Force Knapsack Solution

- Each item can either be included or excluded, leading to a total of 2^n possible combinations, where n is the number of items.
- For each of the 2^n combinations, operations include:
  - Calculating the total weight and total value of the selected items.
  - Comparing the current combination's total value with the best value found so far.
  - Checking if the total weight exceeds the capacity.

  Each of the above operations takes constant time for each item in the combination, summing to $O(n)$ operations per combination in the worst case.

Time Complexity of Brute Force Knapsack Solution

- Each item can either be included or excluded, leading to a total of 2^n possible combinations, where n is the number of items.
- For each of the 2^n combinations, operations include:
  - Calculating the total weight and total value of the selected items.
  - Comparing the current combination's total value with the best value found so far.
  - Checking if the total weight exceeds the capacity.

  Each of the above operations takes constant time for each item in the combination, summing to $O(n)$ operations per combination in the worst case.

  $\Rightarrow$ The total time complexity is

Time Complexity of Brute Force Knapsack Solution

- Each item can either be included or excluded, leading to a total of 2^n possible combinations, where n is the number of items.
- For each of the 2^n combinations, operations include:
  - Calculating the total weight and total value of the selected items.
  - Comparing the current combination's total value with the best value found so far.
  - Checking if the total weight exceeds the capacity.

Each of the above operations takes constant time for each item in the combination, summing to $O(n)$ operations per combination in the worst case.

$\Rightarrow$ The total time complexity is $O(n \cdot 2^n)$.

Method 2: Recursion Approach

- Consider all subsets of items, two cases are considered for every item:
  1. Case 1: The item is included in the optimal subset.
  2. Case 2: The item is not included in the optimal set.

Method 2: Recursion Approach

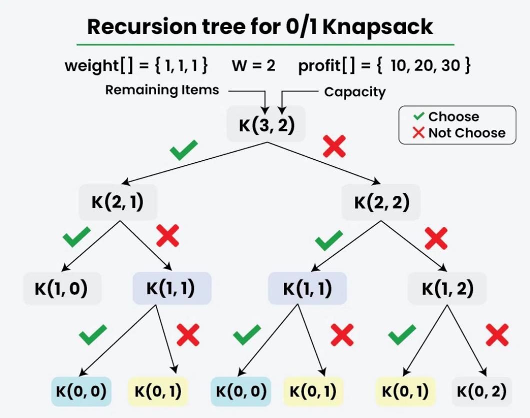
- Consider all subsets of items, two cases are considered for every item:
  1. Case 1: The item is included in the optimal subset.
  2. Case 2: The item is not included in the optimal set.
- The maximum value obtained from 'n' items is the maximum of the following two values:
  1. Case 1 (exclude the nth item):
     - Maximum value obtained by $n-1$ items
     - W weight (unchanged)
  2. Case 2 (include the nth item):
     - Value of the nth item plus the maximum value obtained by the remaining $n-1$ items
     - remaining weight: W minus the weight of the nth item.

Method 2: Recursion Approach

- Consider all subsets of items, two cases are considered for every item:
  1. Case 1: The item is included in the optimal subset.
  2. Case 2: The item is not included in the optimal set.
- The maximum value obtained from 'n' items is the maximum of the following two values:
  1. Case 1 (exclude the nth item):
     - Maximum value obtained by $n - 1$ items
     - W weight (unchanged)
  2. Case 2 (include the nth item):
     - Value of the nth item plus the maximum value obtained by the remaining $n - 1$ items
     - remaining weight: W minus the weight of the nth item.
- If the weight of the nth item is greater than W, then the nth item cannot be included, and Case 2 is the only possibility.

Recursion Tree for Knapsack Problem

K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.



Recursion Approach: Python implementation

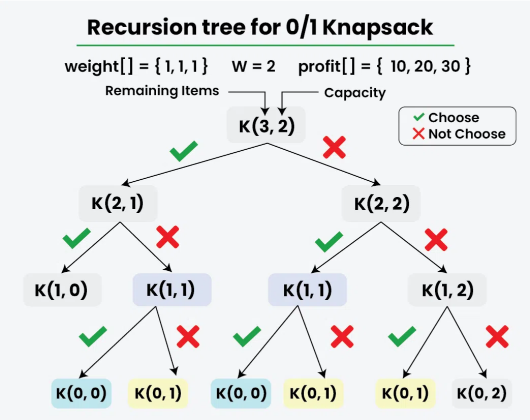
```python
# A naive recursive implementation of Knapsack Problem
# Returns max value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    # Base Case
    if n == 0 or W == 0:
        return 0

    # If weight of the nth item is more than Knapsack of
    # capacity W, then this item cannot be included
    # in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)

    # return the maximum of two cases:
    # (1) nth item included # (2) not included
    else:
        return max(
            val[n-1] + knapSack(W - wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
```

Recursion Tree for Knapsack Problem

K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.



Do you see an issue?

Method 3: Dynamic Programming (Recursion + Memoization)

- Assume $w_1, w_2, \ldots, w_n, W$ are strictly positive integers.
- Define $m[i, w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).

Method 3: Dynamic Programming (Recursion + Memoization)

- Assume $w_1, w_2, \ldots, w_n, W$ are strictly positive integers.
- Define $m[i, w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).
- We can characterize $m[i, w]$ recursively as follows:

$$
\begin{aligned}
m[0, w] &= 0 \\
m[i, w] &= m[i-1, w] \quad \text{if } w_i > w \\
&\quad\quad \text{(the new item is more than the current weight limit)} \\
m[i, w] &= \max(m[i-1, w],\ m[i-1, w-w_i] + v_i) \quad \text{if } w_i \leq w
\end{aligned}
$$

- The solution can then be found by calculating $m[n, W]$. To do this efficiently, we can use a table to store previous computations.

Method 3: Dynamic Programming (Recursion + Memoization)

- Assume $w_1, w_2, \ldots, w_n, W$ are strictly positive integers.
- Define $m[i, w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).
- We can characterize $m[i, w]$ recursively as follows:

$$
\begin{aligned}
m[0, w] &= 0 \\
m[i, w] &= m[i-1, w] \quad \text{if } w_i > w \\
&\quad \text{(the new item is more than the current weight limit)} \\
m[i, w] &= \max(m[i-1, w],\ m[i-1, w - w_i] + v_i) \quad \text{if } w_i \leq w
\end{aligned}
$$

- The solution can then be found by calculating $m[n, W]$. To do this efficiently, we can use a table to store previous computations.
- Visualization of Dynamic Programming for Knapsack

Dynamic Programming Solution in Python

```python
def knapsack(weights, values, W):
    n = len(weights)
    # Create a table to store the maximum value that can be
    # attained with weight less than or equal to w
    m = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Fill the table in bottom-up manner
    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if weights[i-1] > w:
                m[i][w] = m[i-1][w]
            else:
                m[i][w] = max(m[i-1][w],
                              m[i-1][w-weights[i-1]] + values[i-1])

    return m[n][W]

# Example usage:
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
W = 5
max_value = knapsack(weights, values, W)
print(f"The maximum value that can be attained is: {max_value}")
```

Pros and Cons of Dynamic Programming for Knapsack Problem

Dynamic programming

- (Similar to brute force) Guarantees to find the optimal solution for the knapsack problem, as long as the values and weights are integers.
- Is better then recursive algorithm

Pros and Cons of Dynamic Programming for Knapsack Problem

Dynamic programming

- (Similar to brute force) Guarantees to find the optimal solution for the knapsack problem, as long as the values and weights are integers.
- Is better then recursive algorithm (Time Complexity: $\mathcal{O}(n \times W)$ compared to $\mathcal{O}(2^n)$).
- Requires a lot of memory to store the table, which can be impractical for large or continuous inputs.
- Space Complexity:

Pros and Cons of Dynamic Programming for Knapsack Problem

Dynamic programming

- (Similar to brute force) Guarantees to find the optimal solution for the knapsack problem, as long as the values and weights are integers.
- Is better then recursive algorithm (Time Complexity: $\mathcal{O}(n \times W)$ compared to $\mathcal{O}(2^n)$).
- Requires a lot of memory to store the table, which can be impractical for large or continuous inputs.
- Space Complexity: $O(n \times W)$ (compared to $O(n)$ for recursive algorithm.)

Method 4: Heuristic/Greedy Algorithms

- Heuristic: guides to a good solution (not necessarily the optimal solution) by reducing the search space.
- Greedy methods are simpler and faster than dynamic programming.
- They work by making a local and immediate choice at each step, hoping that it will lead to a global and optimal outcome.
- For the knapsack problem:
  1. Greedy methods sort the items by some criterion, such as value, weight, or value-to-weight ratio.
  2. Then they start packing the items in that order until the knapsack is full or no more items can be added.

Greedy method fo the knapsack problem

- Suppose the value of item i is v_i and the weight is w_i.

1. We sort the items by their value/weight ratio
$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$
Since we like adding a lot of value per unit weight, a natural heuristic is to add items which largest value/weight ratio.
2.
    - The item with the maximum value/weight ratio is selected first,
    - The process continues in Greedy fashion: add the 'best' remaining item that satisfies the weight constraint, one at a time
3. Repeat 2 until the knapsack is exactly full.

Greedy method fo the knapsack problem

- Suppose the value of item i is v_i and the weight is w_i.
  1. We sort the items by their value/weight ratio
  $$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$
     Since we like adding a lot of value per unit weight, a natural heuristic is to add items which largest value/weight ratio.
  2. - The item with the maximum value/weight ratio is selected first,
     - The process continues in Greedy fashion: add the 'best' remaining item that satisfies the weight constraint, one at a time
  3. Repeat 2 until the knapsack is exactly full.
- Time Complexity:

Greedy method fo the knapsack problem

- Suppose the value of item i is v_i and the weight is w_i.
  1. We sort the items by their value/weight ratio

     $$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

     Since we like adding a lot of value per unit weight, a natural heuristic is to add items which largest value/weight ratio.
  2. - The item with the maximum value/weight ratio is selected first,
     - The process continues in Greedy fashion: add the ‘best’ remaining item that satisfies the weight constraint, one at a time
  3. Repeat 2 until the knapsack is exactly full.
- Time Complexity: $\mathcal{O}(n\ logn)$ (e.g., complexity of Quicksort/mergesort algorithm).

Greedy method fo the knapsack problem

- Suppose the value of item i is v_i and the weight is w_i.
  1. We sort the items by their value/weight ratio
  $$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$
  Since we like adding a lot of value per unit weight, a natural heuristic is to add items which largest value/weight ratio.
  2.
     - The item with the maximum value/weight ratio is selected first,
     - The process continues in Greedy fashion: add the 'best' remaining item that satisfies the weight constraint, one at a time
  3. Repeat 2 until the knapsack is exactly full.
- Time Complexity: $\mathcal{O}(n\ logn)$ (e.g., complexity of Quicksort/mergesort algorithm).

⚠ The V/W ratio greedy strategy: gives optimal solution for fraction Knapsack problems. For 0/1 (binary) Knapsack it may not give optimal solution.

Recall: Sorting algorithms complexities

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure: Sorting algorithms complexities, `www.bigocheatsheet.com`

Visualization of the algorithms:

- `https://visualgo.net/en/sorting`

Pros and cons of greedy methods

- Greedy methods are appealing for their simplicity and speed.
- They do not need to store any intermediate results.
- They can also handle large or continuous inputs without any problem.
- Greedy methods do not guarantee to find the optimal solutions.
- They can also be misled by local choices that seem good but prevent better choices later on. They do not provide any way to recover from a bad decision or to explore alternative solutions.

Method 5: Branch and Bound Algorithm

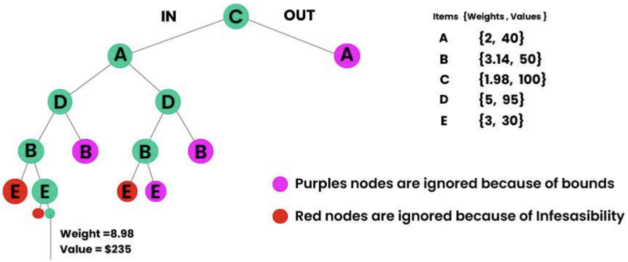
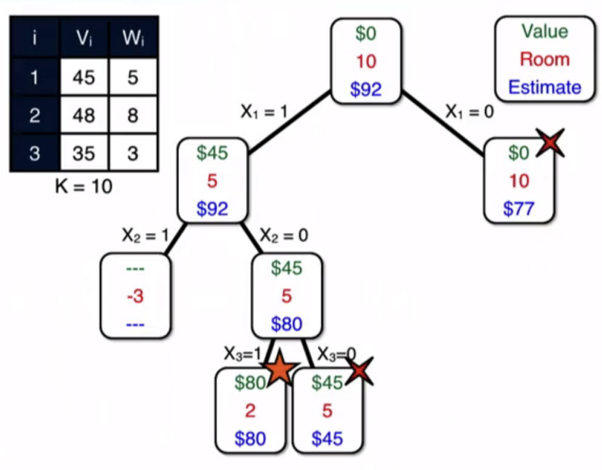


Figure: Knapsack capacity $W = 10$

- Search tree exploration: branch on possible options.
- Prune (do not explore branch) if cannot lead to optimal or if it is unfeasible (too much weight in the knapsack).
- This method is more efficient than brute force and can handle larger datasets.
- Upper bounds (best solutions) are computed using a Greedy approach.

Branch and Bound Algorithm: Example



Prevent Bugs by Testing

- In a project, you'd rather not spend 20% of the time on development and 80% on debugging (removing errors).

Prevent Bugs by Testing

- In a project, you'd rather not spend 20% of the time on development and 80% on debugging (removing errors).
- Better to develop steadily with confidence in correctness.

Prevent Bugs by Testing

- In a project, you'd rather not spend 20% of the time on development and 80% on debugging (removing errors).
- Better to develop steadily with confidence in correctness.
- One way to do this is by using unit tests, i.e., tests of an individual function or method.

Prevent Bugs by Testing

- In a project, you'd rather not spend 20% of the time on development and 80% on debugging (removing errors).
- Better to develop steadily with confidence in correctness.
- One way to do this is by using unit tests, i.e., tests of an individual function or method.
- Write them yourself and be creative in doing so: try to make your function fail.

Using assert Statements

- A useful sanity check in your code can be done using `assert`.
- An `assert` checks a boolean value, and gives an error message if the value is `False` (assert raises an AssertionError).
- Assertion Example:
  - Code continues:

```
assert 3 > 2
numbers = [1, 2, 3, 4, 5]
assert 4 in numbers
```

  - AssertionError is thrown and code terminates in the following case

```
numbers = [1, 2, 3, 4, 5]
assert 10 in numbers
```

Using assert Statements

- A useful sanity check in your code can be done using `assert`.
- An `assert` checks a boolean value, and gives an error message if the value is `False` (assert raises an AssertionError).
- Assertion Example:
  - Code continues:

    ```
    assert 3 > 2
    numbers = [1, 2, 3, 4, 5]
    assert 4 in numbers
    ```

  - AssertionError is thrown and code terminates in the following case

    ```
    numbers = [1, 2, 3, 4, 5]
    assert 10 in numbers
    ```

- A well-placed `assert` can save you a lot of debugging time later.
- Use:
  - Precondition: check the input to a function is as expected, e.g. is it indeed an integer: `assert isinstance(x,int)`.
  - Postcondition: check the output to a function is as expected.

Assertion: print message

We can add a message to be displayed.

```
assert x > y, "x at most y!"
```

- If $x > y$ is `True`, the code will continue as usual.
- If it is `False`, an AssertionError is thrown, code terminates and the text gets printed.

Example of Using assert Statement in Unit Test

```python
def root(a, eps):
    """
    Approximate the square root of a number with
    given accuracy.
    Recursive formula:
      x_{k+1} = (x_k + a / x_k) / 2
    """
    # Initialization
    x = a  # Initial guess
    k = 0  # Iteration counter

    # Iteratively improve the guess
    while abs(x**2 - a) > eps and k < 100:
        #print(f"Iteration {k}: x = {x}, x^2 = {x**2}")
        x = (x + a / x) / 2  # Update the guess
        k += 1  # Increment the counter

    return x
```

Example of Using assert Statement in Unit Test

```
# Example Test Cases
print("Testing the root function...")

# Perfect Square Test
expected = 2.0
actual = root(4, 1e-6)
assert abs(actual - expected) < 1e-6, ...
f"Failed: root(4, 1e-6) = {actual}, expected {expected}"
print("Test 1 passed: root(4, 1e-6) is correct!")

# Non-Perfect Square Test
expected = 2**0.5
actual = root(2, 1e-6)
assert abs(actual - expected) < 1e-6, ...
f"Failed: root(2, 1e-6) = {actual}, expected {expected}"
print("Test 2 passed: root(2, 1e-6) is correct!")

# Large Number Test
expected = 1000.0
actual = root(1e6, 1e-6)
assert abs(actual - expected) < 1e-6, ...
f"Failed: root(1e6, 1e-6) = {actual}, expected {expected}"
print("Test 3 passed: root(1e6, 1e-6) is correct!")

# Small Number Test
expected = 1e-3
actual = root(1e-6, 1e-12)
assert abs(actual - expected) < 1e-12, f"Failed: root(1e-6, 1e-12) = {actual}, expected {expected}"
print("Test 4 passed: root(1e-6, 1e-12) is correct!")
```

A Good Collection of Unit Tests

- The tests should cover all possible inputs, so especially edge cases are important.
- The tests should cover all possible paths in the code.

Use of Unit Tests//When to apply unit tests?

- Perform all tests after every change in your code.
- Unit tests assist you in designing your code.
- Use unit tests during the development of your code to detect errors (find bugs).
- Add tests when you have found a bug.
- First make a simple version (possibly inefficient), and then improve your code.

Integration Tests

- To see if several functions work well together, perform an integration test.
- This can also apply to the whole set of functions. That's the final test.

Integration Tests

- To see if several functions work well together, perform an integration test.
- This can also apply to the whole set of functions. That's the final test.
- Automate the testing: write a program that calls all the tests and that you can run with one push of a button.
- There are various ways to (partially) automate this process, using testing frameworks, including `pytest`, `unittest`, ...
  https://trinket.io/embed/python3/ceaa8dd2b4

Pytest Example

```
# test_file.py
import math

# test_1
def test_sq():
    assert 5 * 5 == 20

# test_2
def test_search():
    assert "Py" in "GeekPython"

# test_3
def test_type():
    assert type([1, 2, 3]) == list

class TestCondition:
    # test_4
    def test_reverse(self):
        sequence = "GeekPython"
        assert sequence[:: -1] == "nohtyPkeeG"

    # test_5
    def test_value(self):
        assert round(math.pi) == 3.14
```

Pytest Output

- Install pytest and run: pytest test_file.py in terminal

```
(base) Hammo009@Chihebs-MacBook-Pro-2 week 6 % pytest test_file.py
========================================= test session starts ==========================================
platform darwin -- Python 3.11.7, pytest-8.3.4, pluggy-1.5.0
rootdir: /Users/Hammo009/Documents/My_courses_research_repos/UU courses:: programs:: seminars:theses/programming-in-mathematics_R_Python_personal/Spring 2025
 R_python course/Course material/slides/week 6
collected 5 items

test_file.py F...F                                                                               [100%]

=============================================== FAILURES ===============================================
______________________________________________ test_sq _________________________________________________

    def test_sq():
>       assert 5 * 5 == 20
E       assert (5 * 5) == 20

test_file.py:6: AssertionError
_______________________________________ TestCondition.test_value _______________________________________

self = <test_file.TestCondition object at 0x1062f6510>

    def test_value(self):
>       assert round(math.pi) == 3.14
E       assert 3 == 3.14
E        +  where 3 = round(3.141592653589793)
E        +    where 3.141592653589793 = math.pi

test_file.py:24: AssertionError
======================================= short test summary info ========================================
FAILED test_file.py::test_sq - assert (5 * 5) == 20
FAILED test_file.py::TestCondition::test_value - assert 3 == 3.14
===================================== 2 failed, 3 passed in 0.03s ======================================
(base) Hammo009@Chihebs-MacBook-Pro-2 week 6 %
```

Unittest in Python

- We can run our unit test via `unittest.main()`.
- In order to define the tests, you define a subclass of `unittest.TestCase` and override the `runTest` method or start the method name with `test`.
- Instead of one `assert`, there are statements for specific cases such as `assertRaises(exception)` or `assertEqual(input1,input2)`.

Unittest Example

```
# testing.py
import unittest

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def perimeter(self):
        return 2*(self.width + self.height)

class TestRectangle(unittest.TestCase):
    def test_area_rectangle(self):
        r = Rectangle(2, 3)
        self.assertEqual(r.area(), 6, "incorrect area for 2x3 rectangle")

    def test_perimeter_rectangle(self):
        r = Rectangle(5, 2)
        self.assertEqual(r.perimeter(), 30,
                         "incorrect perimeter for 5x10 rectangle")

    def test_negative_dimensions(self):
        with self.assertRaises(ValueError):
            Rectangle(-1, 2)
        with self.assertRaises(ValueError):
            Rectangle(2, -3)

unittest.main()
```

Unittest Output

- Install unittest and run testing.py (otherwise you can also run the file here https://trinket.io/embed/python3/ceaa8dd2b4)

```
(base) Hammo009@Chihebs-MacBook-Pro-2 ~ % /Users/Hammo009/anaconda3/bin/python "/Users/Hammo009/Documents/My_courses_research_repos/UU courses::seminars:theses/programming
-in-mathematics_R_Python_personal/Programming for Mathematics winter 2024/course material/slides/lecture 7 /testing.py"
.FF
======================================================================
FAIL: test_negative_dimensions (__main__.TestRectangle.test_negative_dimensions)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Hammo009/Documents/My_courses_research_repos/UU courses::seminars:theses/programming-in-mathematics_R_Python_personal/Programming for Mathematics winter 202
4/course material/slides/lecture 7 /testing.py", line 24, in test_negative_dimensions
    with self.assertRaises(ValueError):
AssertionError: ValueError not raised

======================================================================
FAIL: test_perimeter_rectangle (__main__.TestRectangle.test_perimeter_rectangle)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/Hammo009/Documents/My_courses_research_repos/UU courses::seminars:theses/programming-in-mathematics_R_Python_personal/Programming for Mathematics winter 202
4/course material/slides/lecture 7 /testing.py", line 20, in test_perimeter_rectangle
    self.assertEqual(r.perimeter(), 30,
AssertionError: 14 != 30 : incorrect perimeter for 5x10 rectangle

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=2)
(base) Hammo009@Chihebs-MacBook-Pro-2 ~ %
```

More about Unittest and Pytest

- Pytest
  - https://docs.pytest.org/en/stable/.
  - https://realpython.com/pytest-python-testing/.
  - https://betterstack.com/community/guides/testing/pytest-guide/
- Unittest
  - https://docs.python.org/3/library/unittest.html.
  - https://realpython.com/python-unittest/.
  - https://www.dataquest.io/blog/unit-tests-python/

1. Algorithm Design Techniques: Knapsack Problem as Case Study
2. Developing
   - Testing
   - Handeling Exceptions
3. About Git/Github and Version Control
4. Tips for Final Project

Difference between Syntax Error and Exceptions

Errors are problems in a program due to which the program will stop the execution.

- **Syntax Error:**
  - Caused by incorrect syntax in the code.
  - Leads to program termination.
  - *Example*: Missing colon and/or incorrect indentation.

```
amount = 10000
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

Difference between Syntax Error and Exceptions

Errors are problems in a program due to which the program will stop the execution.

- **Syntax Error:**
  - Caused by incorrect syntax in the code.
  - Leads to program termination.
  - *Example*: Missing colon and/or incorrect indentation.

```
amount = 10000
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

- **Exceptions:**
  - Occur despite correct syntax.
  - Do not stop execution but change the flow.
  - *Example*: ‘ZeroDivisionError‘ when dividing a number by zero.

```
marks = 10000
a = marks / 0
print(a)
```

Common Python Exceptions

- **TypeError**: Raised when an operation is applied to an object of an inappropriate type.
- **NameError**: Raised when a name is not found.
- **IndexError**: Raised when a list index is out of range.
- **KeyError**: Raised when a dictionary key is not found.
- **ValueError**: Raised for invalid argument or input.
- **AttributeError**: Raised when an attribute or method is not found.
- **ZeroDivisionError**: Raised when dividing by zero.
- **ImportError**: Raised when an import fails.

Exceptions

- Example of object-oriented programming: `BaseException` has `Exception` as child which has `ArithmeticError` as child, which has children:
  - `OverflowError`
  - `FloatingPointError`
  - `ZeroDivisionError`

  If you define a new one, take `Exception` (or lower) as parent.
- You can check the exception hierarchy here (https://docs.python.org/2/library/exceptions.html#exception-hierarchy).

Exceptions

- Example of object-oriented programming: `BaseException` has `Exception` as child which has `ArithmeticError` as child, which has children:
  - `OverflowError`
  - `FloatingPointError`
  - `ZeroDivisionError`

  If you define a new one, take `Exception` (or lower) as parent.
- You can check the exception hierarchy here (https://docs.python.org/2/library/exceptions.html#exception-hierarchy).
- It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to handle errors and prevent the program from crashing.

Try-Except Mechanism:

- **Try Clause:** Executes code that might raise an exception.
- **Except Clause:** Handles specific exceptions like ValueError.
- **Flow:** Skip except if no exception, otherwise handle and continue.

Example Code: Repeatedly asking for a valid integer as user input.

```
while True:

    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("That is no valid number. Try again...")
```

Example: Handling TypeError in Python

- **Scenario:** Attempting to add an integer and a string.
- **Code:**

```
x = 5
y = "hello"
z = x + y
```

- **Output:** *TypeError: unsupported operand type(s) for +: 'int' and 'str'*

Resolving TypeError with Try-Catch:

- **Code:**

```
x = 5
y = "hello"
try:
        z = x + y
except TypeError:
        print("Error: cannot add an int and a str")
```

- **Output:** *Error: cannot add an int and a str*

Catching Exceptions: Handling out-of-bound array index Example

Code:

```
a = [1, 2, 3]
try:
   print("Second element = %d" %(a[1]))
   print("Fourth element = %d" %(a[3]))
except:
   print("An error occurred")
```

Output:
Second element = 2
An error occurred

The Finally Keyword in Python

Functionality of Finally:

- Executed after try and except blocks.
- Runs regardless of exception occurrence.

Syntax:

```
try:
   # Code that might raise an exception
except:
   # Handling the exception
finally:
   # Code that is always executed
```

Example Code:

```
try:
   k = 5//0
except ZeroDivisionError:
   print("Can't divide by zero")
finally:
   print('This is always executed')
```

Advantages and Disadvantages of Exception Handling

Advantages of Exception Handling:

- **Improved Program Reliability:** Prevents crashes and incorrect results.
- **Simplified Error Handling:** Separates error handling from main logic.
- **Cleaner Code:** Avoids complex conditional error checks.
- **Easier Debugging:** Tracebacks show where exceptions occur.

Advantages and Disadvantages of Exception Handling

Advantages of Exception Handling:

- **Improved Program Reliability:** Prevents crashes and incorrect results.
- **Simplified Error Handling:** Separates error handling from main logic.
- **Cleaner Code:** Avoids complex conditional error checks.
- **Easier Debugging:** Tracebacks show where exceptions occur.

Disadvantages of Exception Handling:

- **Performance Overhead:** Slower than conditional error checks.
- **Increased Code Complexity:** More complex with multiple exceptions.

More about Exceptions in Python

- https://docs.python.org/3/tutorial/errors.html.
- https://docs.python.org/3/tutorial/errors.html.
- https://maths-with-python.readthedocs.io/en/latest/09-exceptions-testing.html

Why Version Control and Git?

- Version control:
  - Keep track of changes in batches and attach comments.
  - Ability to restore to previous versions.
  - At any moment, you can return to a previous (working) version and view differences.
  - Can create ‘branches’ to work on multiple options at the same time
- Collaboration: With a specific group (multiple people) on the same project, and from different workstations.
- Access your data from everywhere

Version Control for Resolving Conflicts

- You and a partner are working on a program. Both of you have modified a file and want to combine these changes. Emailing, cutting, and pasting?
- Better: prevent conflicts by always working on the most recent version, making task agreements, and thus being able to work independently of each other.
- If you still work on the same file simultaneously, you can combine the changes via the system.

Git and Github

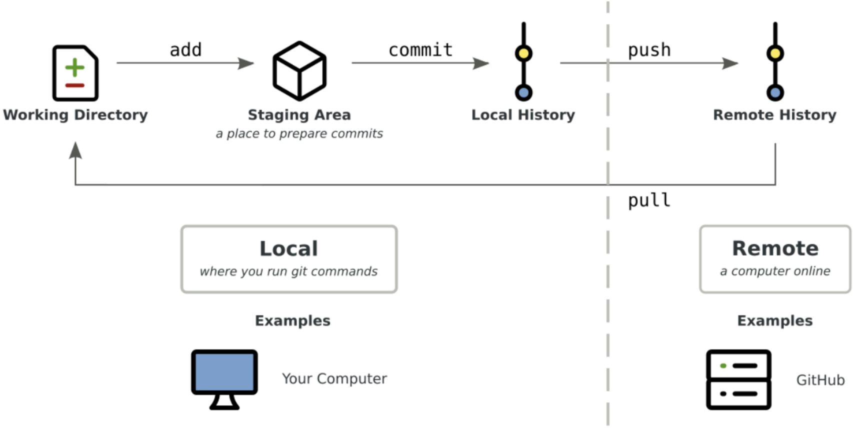
- Git is 'distributed version control software'.
- Github is a platform for using git.
- Alternative: Bitbucket is a platform for mercurial.
- Python packages such as Pandas and Numpy are hosted on Github!
- Illustration:[1]
  - https://github.com/Asabeneh/30-Days-Of-Python
  - https://github.com/DJwu05/SET-MAIN
  - https://github.com/Jerros/Programing-for-Math-Assignment/commits/main/

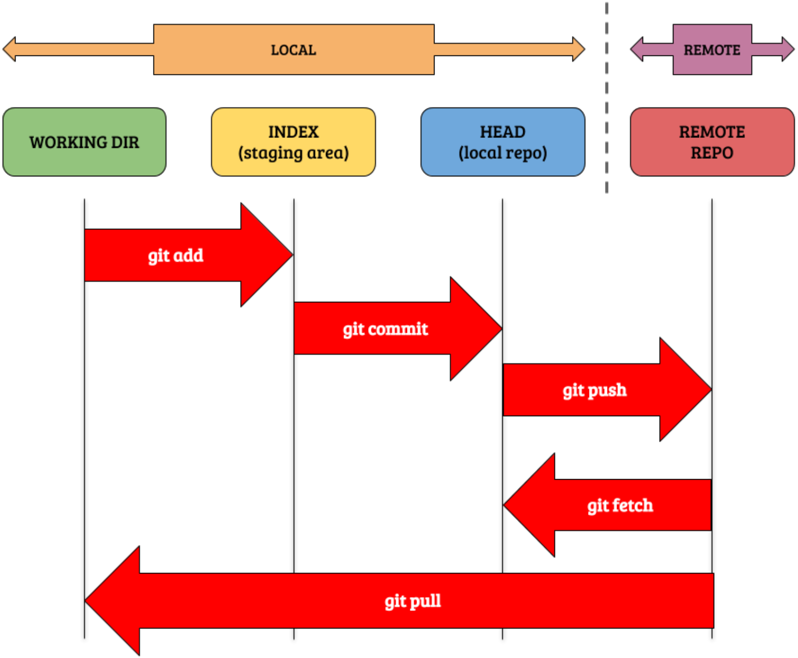
[1] Some Github repositories will be shown only in class as they are private

Tutorials on how to set up and work with Git and Github

1. YouTube Links:
   - Using GitHub Desktop app
   - Using Terminal commands
   - Using Terminal commands
2. Other Non-Video Resources:
   - Using GitHub Desktop
   - Using Terminal commands

Different levels




LOCAL
REMOTE
WORKING DIR
INDEX
(staging area)
HEAD
(local repo)
REMOTE
REPO
git add
git commit
git push
git fetch
git pull

From your directory to the repository

- `git add <file>` is used to ensure the changes to the file are tracked.
- `git commit <file>` is used to move the changes made to the file from the staging area to the local repository.
    - Can add a message using `-m` option.
- `git push` syncs your local repository into the remote repository.

From the repository to your directory

- `git fetch` is used to copy the changes made to the remote repository, into your local repository.

  Fetch only 'loads' the changes made to the remote and you still need to 'merge' these with your local content yourself.

- `git pull` performs both fetch and merge: simply said it merges the remote repository into your working directory.

Key Commands summary

- `clone` - make a local copy from a central repository (server)
- `commit` - save a change (locally)
- `push` - copy a change to the server
- `pull` - copy a change from the server
- `branch` - create a new branch (locally), a split-off
- `merge` - merge a branch

Final Project: Report

- Problem statement: explain the context / background and challenges of the question.

Final Project: Report

- Problem statement: explain the context / background and challenges of the question.
- Algorithm: explanation of your algorithm and analysis.
  - Which data structures do you use and why?
  - Which classes do you use?
  - Key functions or clever tricks?

Final Project: Report

- Problem statement: explain the context / background and challenges of the question.
- Algorithm: explanation of your algorithm and analysis.
  - Which data structures do you use and why?
  - Which classes do you use?
  - Key functions or clever tricks?
- Code manual: how can the code be used? Give examples and clearly describe the inputs and outputs. More instructions in next weeks.

Final Project: Report

- Problem statement: explain the context / background and challenges of the question.
- Algorithm: explanation of your algorithm and analysis.
  - Which data structures do you use and why?
  - Which classes do you use?
  - Key functions or clever tricks?
- Code manual: how can the code be used? Give examples and clearly describe the inputs and outputs. More instructions in next weeks.
- Conclusion/discussion: summary and possible improvements / extensions.

Final Project: Report

- Problem statement: explain the context / background and challenges of the question.
- Algorithm: explanation of your algorithm and analysis.
  - Which data structures do you use and why?
  - Which classes do you use?
  - Key functions or clever tricks?
- Code manual: how can the code be used? Give examples and clearly describe the inputs and outputs. More instructions in next weeks.
- Conclusion/discussion: summary and possible improvements / extensions.
- Structure and care: carefully proofread for typos, neat diagrams, clear structure.

Tips for the Report

- Be consistent in your choices and think at each paragraph what you want to convey (and whether it's clearly stated).
- Use descriptive headings such as 'introduction', 'problem statement', 'algorithm', 'manual', 'conclusion', with possible subheadings.
- LaTeX: software for writing mathematical articles. Collaboration possible via Overleaf.
- Start early so you can describe all parts and have time to proofread. You can write some sections before the code is completely finished.

Final Project: Code

- Design: how well thought out are the functions, classes, and data structures used?
- Usability: does the code work? Have you done anything extra?
- Readability: clear naming in code, use of comments to explain what happens.

Tips for Coding

- Divide the problem into subproblems.
    - Then you can work in parallel.
    - Start with simple tasks and build from there; e.g., is there an easier case you can try first?
- Write your own tests to see if the code works.
- Git: version control system.
    - Work on the same code simultaneously and merge changes.
    - Easily revert to previous versions.
    - Access the code from anywhere.