# Lab: Algorithms-Introduction

This document defines the exercises for the ["Python Advanced" course at @Software University.](https://softuni.bg/trainings/4370/python-advanced-january-2024)

Please submit your solutions (source code) to all the below-described problems in [Judge](https://judge.softuni.org/Contests/4590/Algorithms-Introduction-Lab).

## Recursive Array Sum

Write a program that finds the sum of all elements in an integer array. Use **recursion**.

**Note**: In practice, this recursion should not be used here (instead use an **iterative solution**), this is just an exercise.

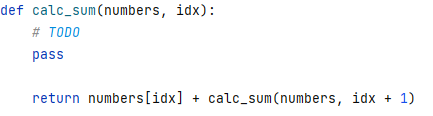
### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 | 10 |
| -1 0 1 | 0 |

### Hints

Write a **recursive** method. It will take as arguments the **input array** and an **index**.

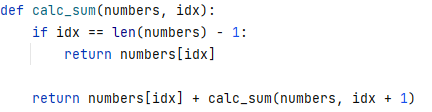
* The method should return the **current element** + the **sum of all next elements** (obtained by recursively calling it):



* The recursion should stop when there are no more elements in the array:



This is how the complete solution should look:



## Recursive Factorial

Write a program that calculates the recursively factorial of a given number.

**NOTE**: In practice, this recursion should not be used here (instead use an **iterative solution**).

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 | 120 |
| 10 | 3628800 |

### Hints

Write a **recursive** method. It will take as arguments an integer number.

* The method should return the **current element** \* the **result of calculating the factorial of the current   
  element - 1** (obtained by recursively calling it).
* The recursion should stop when the last element is reached.

## Recursive Drawing

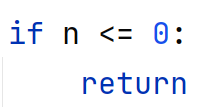
Write a program that draws the figure below depending on **n**.

### Examples

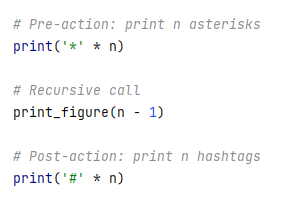
|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 | \*\*  \*  #  ## |
| 5 | \*\*\*\*\*  \*\*\*\*  \*\*\*  \*\*  \*  #  ##  ###  ####  ##### |

### Hints

* Set the bottom of the recursion:



* Define **pre** and **post** recursive behavior:



## Sum of Coins\*

Write a program that gathers a sum of money using the least possible number of coins. This is the **range of possible coin values**:

* **{ 1, 2, 5, 10, 20, 50 }**

You will receive the **desired sum**. The goal is to **reach the sum using as few coins as possible using a greedy approach**. We'll assume that each coin value and the desired sum are **integers**.

**Input**

* On the first line, you will receive the **coins**.
* On the next line, you will receive the target **sum**.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1, 2, 5, 10, 20, 50  923 | Number of coins to take: 21  18 coin(s) with value 50  1 coin(s) with value 20  1 coin(s) with value 2  1 coin(s) with value 1 | 18\*50 + 1\*20 + 1\*2 + 1\*1 = 900 + 20 + 2 + 1 = 923 |
| 1  42 | Number of coins to take: 42  42 coin(s) with value 1 |  |
| 3, 7  11 | Error | Cannot reach the desired sum with these coin values |
| 1, 2, 5  2031154123 | Number of coins to take: 406230826  406230824 coin(s) with value 5  1 coin(s) with value 2  1 coin(s) with value 1 | The solution should be fast enough to handle a combination of small coin values and a large desired sum |
| 1, 10, 9  27 | Number of coins to take: 9  2 coin(s) with value 10  7 coin(s) with value 1 | The greedy approach produces a non-optimal solution (9 coins to take instead of 3 with the value of 9) |

### Hints

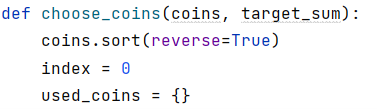
**Greedy Approach**

In the context of this problem, a greedy algorithm employs a strategy to iteratively select the optimal coin values. It strives to choose the best possible coin value, starting with the largest and progressing to the next largest, until the desired sum is attained or there are no more coin values left. The number of coins to take for each value may vary. In scenarios where the desired sum is significantly larger than individual coin values, returning the result as a **List[]** could be inefficient and may even lead to exceptions. A more practical approach is to utilize a dictionary **{key: int, value: int}**, where the keys represent the coin values, and the values indicate the number of coins to take for each specified coin value. For example, if the coin values are **{1}** and the target sum is **42**, instead of returning a list with **42** elements, a more efficient choice is to return a dictionary with a single key-value pair:   
**{1, 42}**.

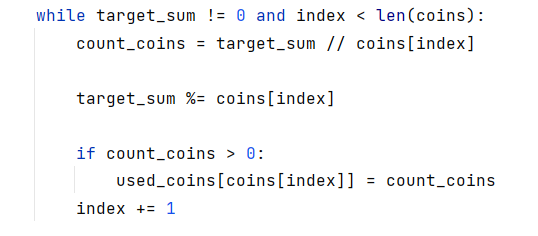
**Greedy Algorithm Implementation**

Your task is to implement the **choose\_coins()** method:

* The input list of coins is **sorted in descending order**. This is crucial for the greedy approach, as it allows us to consider the **largest coin values first**.
* Variables like **index** and **used\_coins** are initialized to keep track of the current coin value being considered and the selected coins.
* Given these variables, when do we stop taking coins? There are two possibilities:
* We have reached the target sum.
* We have run out of coin values



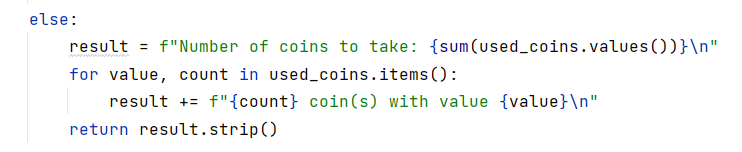
* The **while** loop iterates as long as the **target\_sum** is **not zero** and there are **more coin values to consider**.
* Inside the loop, it calculates how many coins of the current value can be taken and updates the **target\_sum** accordingly.
* If **at least one coin of the current value can be taken**, it updates the **used\_coins** dictionary with the count of coins.



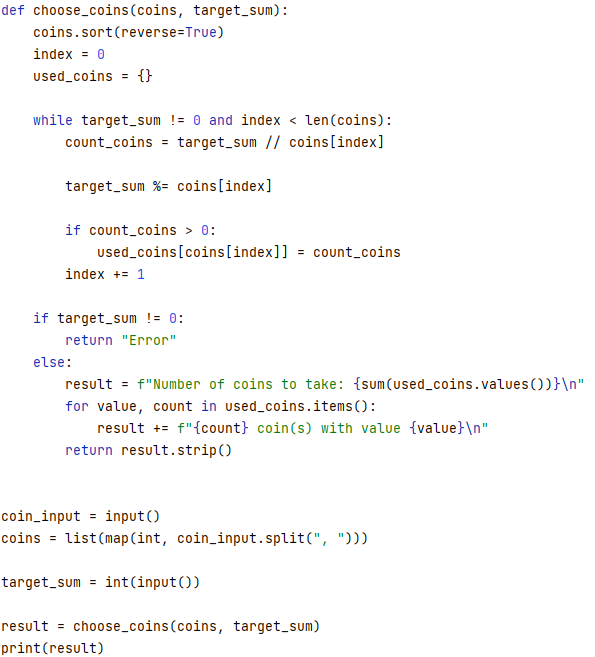
* After the loop, it checks if the **target\_sum** is **still not zero**. If so, it means the desired sum cannot be reached with the available coins, and an **"Error"** message is returned.



* If the **target\_sum** is **zero**, it calculates the **total number of coins taken** and builds a string representation of the selected coins.



This is how the complete solution should look:



## Set Cover\*

Write a program that finds **the smallest subset of** sets,which **contains all elements** from a given **sequence**.

In the Set Cover problem, we are given two sets - a set of sets (we'll call it sets) and a universe(**a sequence**).The sets contain all elements from the universe and no others, however, some elements are repeated. The task is to **find the smallest subset of** sets **that contains all elements in the** universe**.**

**Input**

* On the first line, you will receive the **universe**.
* On the second line, you will receive the target **number of sets**.
* On the next lines, you will receive **different** **sets of sets**.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1, 2, 3, 4, 5  4  1  2, 4  5  3 | Sets to take (4):  { 2, 4 }  { 1 }  { 5 }  { 3 } |
| 1, 2, 3, 4, 5  4  1, 2, 3, 4, 5  2, 3, 4, 5  5  3 | Sets to take (1):  { 1, 2, 3, 4, 5 } |
| 1, 3, 5, 7, 9, 11, 20, 30, 40  6  20  1, 5, 20, 30  3, 7, 20, 30, 40  9, 30  11, 20, 30, 40  3, 7, 40 | Sets to take (4):  { 3, 7, 20, 30, 40 }  { 1, 5, 20, 30 }  { 9, 30 }  { 11, 20, 30, 40 } |

### Hints

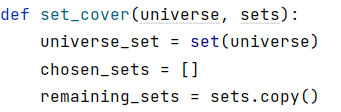
**Greedy Approach**

Using the greedy approach, at each step, we'll take the set which contains the most elements present in the universe which we haven't yet taken. In the first step, we'll always take the set with the largest number of elements, but it gets a bit more complicated afterward. To simplify our job (and not check against two sets at the same time), when taking a set, we can remove all elements in it from the universe. We can also remove the set from the sets we're considering.

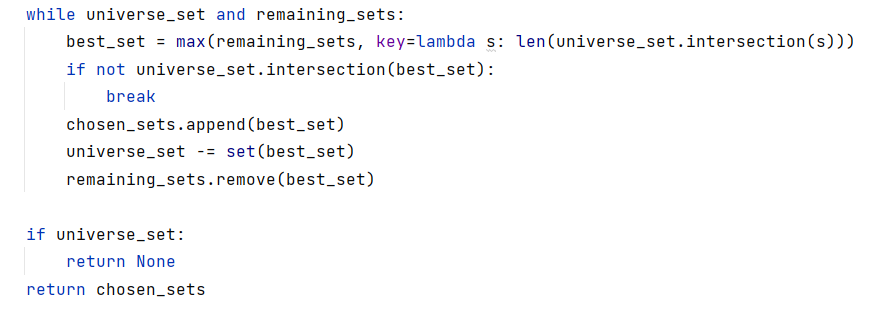
**Greedy Algorithm Implementation**

Your task is to implement the **set\_cover()** method:

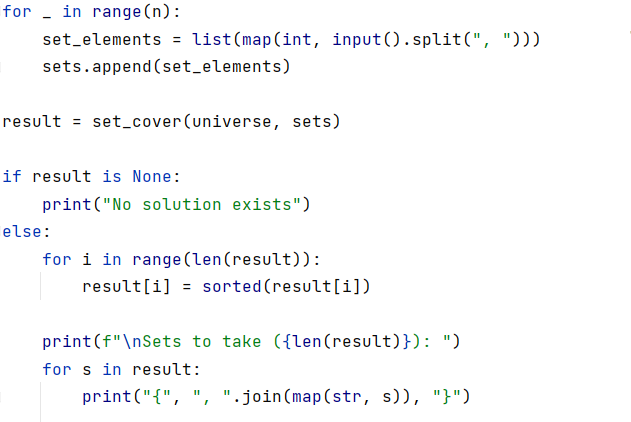
* This function **set\_cover** takes two parameters: **universe** (a list representing the universe of elements) and **sets** (a list of sets where each set contains some elements from the universe).



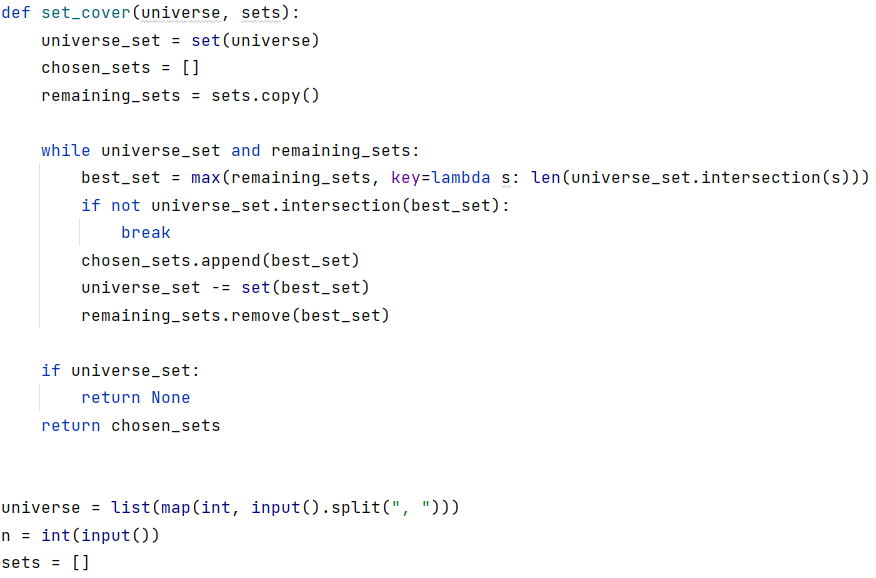
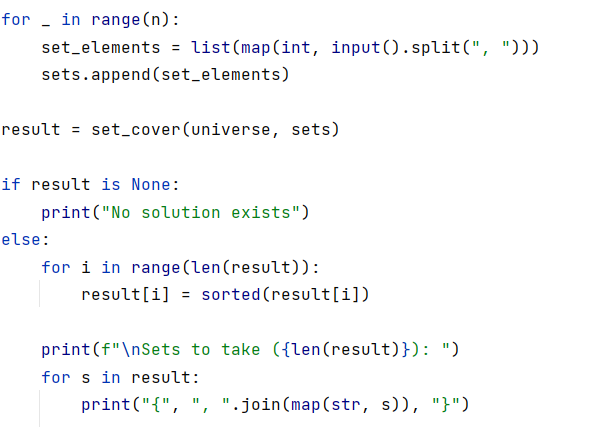
* The **while** loop runs until the **universe\_set** is not empty. In each iteration:
  + **best\_set** - finds the set from the list of **sets** that covers the maximum number of remaining elements in the universe. The key parameter uses the intersection of sets with the universe to determine the size of the overlap.
  + **chosen\_sets** - appends the selected set (**best\_set**) to the **chosen\_sets** list.
  + **universe\_set** - removes the elements covered by the selected set from the remaining universe.
* The function returns the list of selected sets that cover the entire universe.



* Calls the **set\_cover** function with the provided **universe** and **sets**.
* Sorts the elements within each set in **ascending order**.
* Prints the final result, displaying the sets to take to cover the entire universe.



This is how the complete solution should look:

## Binary Search

Implement an algorithm that finds the index of an element in a sorted array of integers in logarithmic time.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 3 4 5  1 | 0 | An index of 1 is 0 |
| -1 0 1 2 4  1 | 2 | An index of 1 is 2 |

### Hints

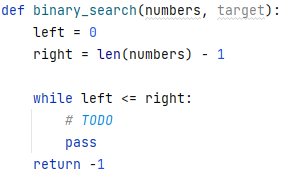
First, if you're not familiar with the concept, read about binary search in [Wikipedia](https://en.wikipedia.org/wiki/Binary_search_algorithm).

In short, if we have a **sorted collection** of comparable elements, instead of doing a linear search (which takes linear time), we can eliminate half the elements at each step and finish in logarithmic time. Binary search is a **divide-and-conquer** algorithm; we start at the middle of the collection, if we haven’t found the element there, there are three possibilities:

* The element we’re looking for is smaller – then look to the left of the current element, we know all elements to the right are larger
* The element we’re looking for is larger – look to the right of the current element
* The element is not present, traditionally, return -1 in that case

Start by defining a **binary\_search()** method:

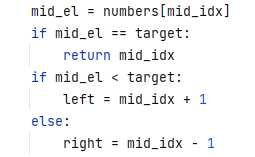
* Inside the method, define two variables defining the bounds to be searched and a **while** loop:



Inside the **while** loop, we need to find the **midpoint**:



If the key is to the left of the midpoint, move the right bound. If the key is to the right of the midpoint, move the left bound:



## Selection Sort

Write an implementation of **Selection Sort**. You should read an array of integers and sort them.

**Output**

* You should print out the sorted list in the format described below.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 1 2 3 4 5 |
| 13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65 7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43 | 0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25 27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77 78 80 83 85 86 86 86 88 89 93 93 96 97 98 |

## Bubble Sort

Write an implementation of Bubble Sort. You should read an array of integers and sort them.

**Output**

* You should print out the sorted list in the format described below.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 1 2 3 4 5 |
| 13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65 7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43 | 0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25 27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77 78 80 83 85 86 86 86 88 89 93 93 96 97 98 |

## Insertion Sort

Write an implementation of Insertion Sort. You should read an array of integers and sort them.

**Output**

* You should print out the sorted list in the format described below.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 1 2 3 4 5 |
| 13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65 7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43 | 0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25 27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77 78 80 83 85 86 86 86 88 89 93 93 96 97 98 |