## What is a Numpy array?

[**NumPy**](https://www.geeksforgeeks.org/python-numpy/) is the fundamental package for scientific computing in [Python](https://www.geeksforgeeks.org/python-programming-language/). Numpy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences. Numpy is not another programming language but a Python extension module. It provides fast and efficient operations on arrays of homogeneous data.

**Some important points about Numpy arrays:**

* We can create an N-dimensional array in Python using [Numpy.array().](https://www.geeksforgeeks.org/basics-of-numpy-arrays/)
* The array is by default Homogeneous, which means data inside an array must be of the same Datatype. (Note You can also create a structured array in Python).
* Element-wise operation is possible.
* Numpy array has various functions, methods, and variables, to ease our task of matrix computation.
* Elements of an array are stored contiguously in memory. For example, all rows of a two-dimensioned array must have the same number of columns. A three-dimensional array must have the same number of rows and columns on each card.

## What is Python List?

A **Python** [**list**](https://www.geeksforgeeks.org/python-list/) is a collection that is ordered and changeable. In Python, lists are written with square brackets.

**Some important points about Python Lists:**

* The list can be homogeneous or heterogeneous.
* Element-wise operation is not possible on the list.
* Python list is by default 1-dimensional. But we can create an N-Dimensional list. But then too it will be 1 D list storing another 1D list
* Elements of a list need not be contiguous in memory.

## Comparison between Numpy array and Python List

**Python Lists**

1. **Element Overhead:** Lists in Python store additional information about each element, such as its type and reference count. This overhead can be significant when dealing with a large number of elements.
2. **Datatype:** Lists can hold different data types, but this can decrease memory efficiency and slow numerical operations.
3. **Memory Fragmentation:** Lists may not store elements in contiguous memory locations, causing memory fragmentation and inefficiency.
4. **Performance:** Lists are not optimized for numerical computations and may have slower mathematical operations due to Python’s interpretation overhead. They are generally used as general-purpose data structures.
5. **Functionality:** Lists can store any data type, but lack specialized NumPy functions for numerical operations.

**Numpy Arrays**

1. **Homogeneous Data:** NumPy arrays store elements of the same data type, making them more compact and memory-efficient than lists.
2. **Fixed Data Type:** NumPy arrays have a fixed data type, reducing memory overhead by eliminating the need to store type information for each element.
3. **Contiguous Memory:** NumPy arrays store elements in adjacent memory locations, reducing fragmentation and allowing for efficient access.
4. **Array Metadata:** NumPy arrays have extra metadata like shape, strides, and data type. However, this overhead is usually smaller than the per-element overhead in lists.
5. **Performance:** NumPy arrays are optimized for numerical computations, with efficient element-wise operations and mathematical functions. These operations are implemented in C, resulting in faster performance than equivalent operations on lists.

**Memory consumption between Numpy array and lists**

In Python, a list is a built-in data structure that can hold elements of varying data types. However, the flexibility of lists comes at the cost of memory efficiency.

Python’s NumPy library supports optimized numerical array and matrix operations.

## Counting sort

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **counting sort** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm) a collection of objects according to keys that are small positive [integers](https://en.wikipedia.org/wiki/Integer); that is, it is an [integer sorting](https://en.wikipedia.org/wiki/Integer_sorting) algorithm. It operates by counting the number of objects that possess distinct key values, and applying prefix sum on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

**function** CountingSort(input, *k*)

count ← array of *k* + 1 zeros

output ← array of same length as input

**for** *i* = 0 **to** length(input) - 1 **do**

*j* = key(input[*i*])

count[*j*] = count[*j*] + 1

**for** *i* = 1 **to** *k* **do**

count[*i*] = count[*i*] + count[*i* - 1]

**for** *i* = length(input) - 1 **down to** 0 **do**

*j* = key(input[*i*])

count[*j*] = count[*j*] - 1

output[count[*j*]] = input[*i*]

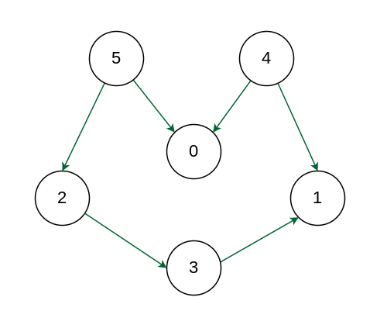
**return** output

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore, the time for the whole algorithm is the sum of the times for these steps, **O(n + k).**

## Topological Sorting

Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.

**Note:** Topological Sorting for a graph is not possible if the graph is not a **DAG**.



Example

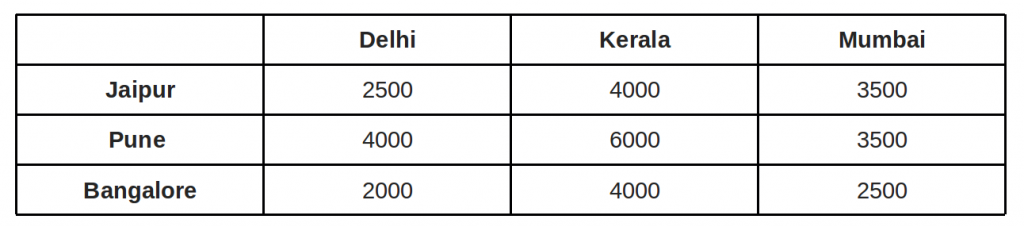
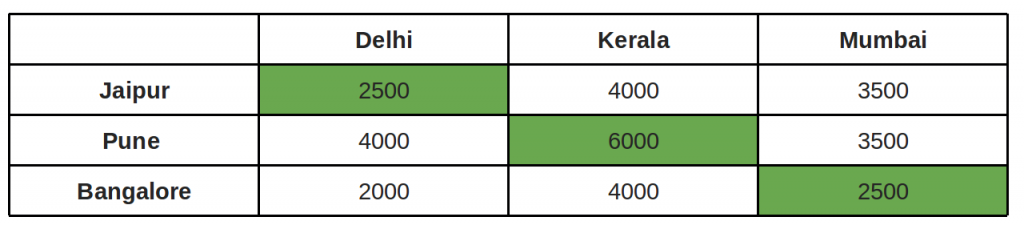
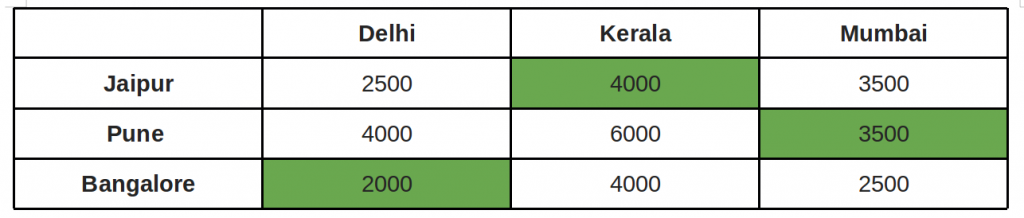
**Output:** 5 4 2 3 1 0  
**Explanation:** The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).  A topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. Another topological sorting of the following graph is “4 5 2 3 1 0”.

**Algorithm for Topological Sorting using DFS:**

Here’s a step-by-step algorithm for topological sorting using Depth First Search (DFS):

* Create a graph with **n** vertices and **m**-directed edges.
* Initialize a stack and a visited array of size **n**.
* For each unvisited vertex in the graph, do the following:
  + Call the DFS function with the vertex as the parameter.
  + In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.
  + Once all the neighbors have been visited, push the vertex onto the stack.
* After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.
* The resulting list is the topologically sorted order of the graph.

## Hungarian Algorithm for Assignment Problem

Let there be n agents and n tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized. **Example:** You work as a manager for a chip manufacturer, and you currently have 3 people on the road meeting clients. Your salespeople are in Jaipur, Pune and Bangalore, and you want them to fly to three other cities: Delhi, Mumbai and Kerala. The table below shows the cost of airline tickets in INR between the cities: The question: where would you send each of your salespeople in order to minimize fair? Possible assignment: Cost = 11000 INR Other Possible assignment: Cost = **9500** INR and this is the best of the **3!** possible assignments. **Brute force solution** is to consider every possible assignment implies a complexity of **Ω(n!)**. The **Hungarian algorithm, aka Munkres assignment algorithm**, utilizes the following theorem for polynomial runtime complexity (**worst case O(n3)**) and guaranteed optimality: *If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix.* We reduce our original weight matrix to contain zeros, by using the above theorem. We try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is **zero**. **Core of the algorithm (assuming square matrix):**

1. For each row of the matrix, find the smallest element and subtract it from every element in its row.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. Do the same (as step 1) for all columns.
2. Cover all zeros in the matrix using minimum number of horizontal and vertical lines.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

1. *Test for Optimality:* If the minimum number of covering lines is n, an optimal assignment is possible and we are finished. Else if lines are lesser than n, we haven’t found the optimal assignment, and must proceed to step 5.
2. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

Изображение выглядит как текст, Шрифт, снимок экрана, число

Автоматически созданное описание