**CS469 Data Structures and Algorithms**

**HOS08 Greedy Algorithm**

05/17/2023 Updated by Christopher Sharp

02/20/2024 Reviewed by Anh Nguyen

09/29/2024 Reviewed by Shahid Khan

School of Technology and Computing (STC) @ City University of Seattle (CityU)

**Before You Start**

* The document’s examples are written in Python. Please finish the Python tutorial in the Module00 folder before your start the assignment.
* Some steps are not explained in the tutorial**.** If you are not sure what to do:
  1. Consult the resources listed below.
  2. If you cannot solve the problem after a few tries, ask a TA for help.

**Learning Outcomes**

Students will be able to:

* Understand greedy algorithm
* Implement greedy algorithm

**Resources**

* Bhargava, Y, A. (2016). Grokking algorithms GitHub repository. Retrieved from: <https://github.com/egonSchiele/grokking_algorithms>
* Python Tutor. <https://pythontutor.com/visualize.html>
* *Optimal substructure*: <https://en.wikipedia.org/wiki/Optimal_substructure>
* Chapter 16 Greedy Algorithms: <https://www.cs.rochester.edu/u/gildea/csc282/slides/C16-greedy.pdf>
* Activity selection problem: <https://en.wikipedia.org/wiki/Activity_selection_problem>
* Interval scheduling: <https://en.wikipedia.org/wiki/Interval_scheduling>
* Continuous knapsack problem: <https://en.wikipedia.org/wiki/Continuous_knapsack_problem>

# 1. What is Greedy Algorithm

Greedy algorithm is an algorithm that takes the best or optimal choice in the current state in each step of the selection, with hope that the result is the best or optimal. Greedy algorithms need to fully explore the conditions in the problem. There is no fixed pattern. Solving greedy algorithms requires certain intuition and experience.

Greedy algorithms do not get the overall optimal solution to all problems. The problems that can be solved by the greedy algorithm need to have the three properties of *greedy-choice*, *non-aftereffect*, and *optimal substructure*:

* *greedy-choice property*: It can be said that the problem has *greedy-choice property* if the overall optimal solution of a problem can be achieved through a series of local optimal solution choices, and each choice can depend on the choices made before, but not on the choices to be made later. The choice that appears to be the best at that moment for all the sub-problems, leads us to an overall optimal solution by never reconsidering our earlier decisions. For a specific problem, to determine whether it has the nature of greedy choice, it must be proved that the greedy choice made at each step ultimately leads to the overall optimal solution of the problem.
* *Non-aftereffect property*: Once the state of a certain stage is determined, it is not affected by the state of the future decision-making, that is, the process after a state will not affect the previous state, but only related to the current state.
* A given problem has *Optimal Substructure Property* if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Maybe you have learned some classic algorithms that use greedy algorithms, such as Kruskal's Minimum Spanning Tree (MST), Prim's MST, Dijkstra's Shortest Path, and Huffman Coding. These algorithms all use greedy algorithm thinking to solve specific problems.

# 2. Program for Maximizing Number of Tasks

Imagine you are a very busy person, and you have lots of interesting things to be done within a short span of time T. You would want to do maximum of those interesting to-do things in the short time you have.

You have an integer array A, where each element ai represents the time taken to complete a task. Your main task is now to compute the maximum number of tasks that can be done in the limited time T.

## Analysis

By carefully observing the problem, we can say that this problem requires nothing but a simple application of the Greedy algorithm. Our natural greedy instinct says that to accomplish maximum tasks, we must do the tasks that require minimum amount of time.

In every iteration, we greedily select the tasks which take minimum completion time. This is achieved by maintaining two variables currentTime and numberOfTasks. To solve this problem, we follow the below steps:

1. Sort the given array A in ascending order.
2. Select a to-do task one-by-one. Loop:
   1. Add the time taken to complete that task to the currentTime variable.
   2. Break loop if currentTime > T.
   3. Increment numberOfTasks by 1.

## Implement

Let us implement it in Python.

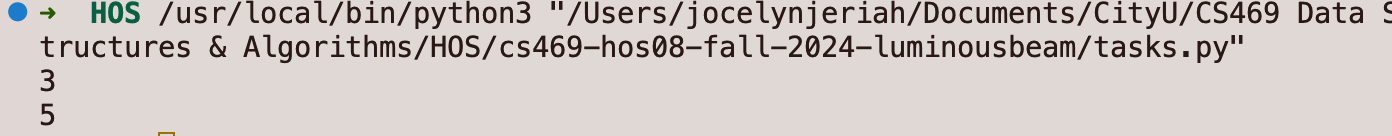
In the repository, create a file called tasks.py.

Type the following code:

文本

描述已自动生成

**Add the screenshot of the console output of the above code.**

****

# Program for Activity Selection Problem

The activity selection problem is also known as the Interval scheduling maximization problem (ISMP), which is a special type of the more general Interval Scheduling problem.

Let be the set of **activities** that compete for a resource. Each activity has its **starting time** and **finish time** with , namely, if selected, takes place during time . No two activities can share the resource at any time point. We say that activities and are **compatible** if their time periods are disjoint.

The activity-selection problem is the problem of selecting the **largest set** of mutually compatible activities.

Now you are given n activities with their start and finish times. Finding the largest solution set of non-conflicting activities.

**Example input:**

|  |
| --- |
| start time = [0, 3, 4, 2, 5, 3, 9, 11, 10]  finish time = [3, 4, 6, 8, 9, 12, 13, 14, 15] |

Example visualization:

Chart

Description automatically generated

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

In this algorithm the activities are first sorted according to their finishing time, from the earliest to the latest, where a tie can be broken arbitrarily. Then the activities are greedily selected by going down the list and by picking whatever activity that is compatible with the current selection.

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array.
3. Do the following for the remaining activities in the sorted array.
   1. If the start time of this activity is greater than or equal to the finish time of the previously selected activity, then select this activity.

## Implement

Let us implement it in Python.

In the **Module08** folder, create a file called activity.py.

Type the following code. The test case data is at the end to copy and paste.

Text

Description automatically generated

Text

Description automatically generated

Test Case Data:

1. 0, 3, 4, 2, 5, 3, 9, 11, 10

3, 4, 5, 8, 9, 12, 13, 14, 15

1. 1, 3, 0, 5, 8, 5

2, 4, 6, 7, 9, 9

1. 4, 0, 1, 1, 3

6, 2, 3, 6, 4

The space complexity of this algorithm O(1) in terms of time complexity, the sorting part can be as small as O(nlogn) and the other part is O(n) as thhe total time complexity is .

If you want to know more details about proof of optimality, you can read <https://www.cs.rochester.edu/u/gildea/csc282/slides/C16-greedy.pdf> or <https://en.wikipedia.org/wiki/Activity_selection_problem#Proof_of_optimality>

# Push Your Work to GitHub

**Make sure your files are saved in the module folder.**

Open a terminal on visual studio code and make sure you’re in the repository folder. (i.e: hos01\_courseName\_GitHubUserName)

**Type the following command to upload your work**:

>>> git add .

>>> git commit -m "Submission for HOS8 - YourName"

>>> git push origin master