Hacettepe University Computer Engineering Department
BBM204 Software Practicum II Spring 2023
Programming Assignment 2

Topics: Dynamic Programming, Greedy Programming

TAs: [redacted]
Programming Language: Java (OpenJDK 11)

Due Date: April 20, 2023 23:59

Introduction

A Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to a global solution are best fit for Greedy.

Dynamic programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub-problems so that we do not have to re-compute them when needed later.

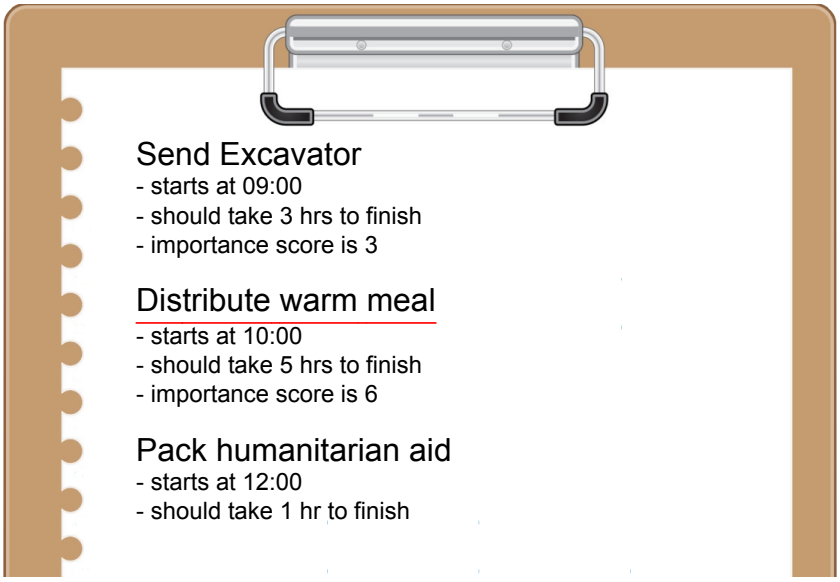
In this experiment, you will explore and utilise a dynamic programming approach along with a greedy approach on similar problems with the intention of reducing the computational complexity compared to brute-force approach.

Background

In the wake of a natural disaster, scheduling relief tasks is essential. Volunteers and resources often rush to the affected area, but without proper coordination, the relief effort can quickly

become disorganized and ineffective. By scheduling tasks, organizers can ensure that volunteers and resources are used efficiently, assigning individuals with specific skills to the most suitable tasks and directing supplies to where they are most needed.
Scheduling relief tasks also ensures that support is sustained over time, as a surge of volunteers in the immediate aftermath may dwindle later on. This sustained effort is necessary for communities to receive consistent and necessary aid. Additionally, scheduling tasks provides structure and purpose to volunteers who may feel overwhelmed upon arriving in the disaster zone, allowing them to be more confident and productive in their relief efforts. Ultimately, scheduling relief tasks is critical in maximizing the impact of the relief effort and providing effective and sustainable support to the affected communities.

In this assignment, as Hacettepe BBM and AIN students, you are supposed to help schedule relief efforts by using dynamic programming and greedy programming approaches. Assume that a list of tasks will be given to a team of volunteers. Some tasks will overlap and it will not be possible to finish all the tasks from the list. Therefore, the team needs a planner that will optimize the task schedule for maximum benefit.



Problem Definition

A Must-Use Starter Code Template

You **must use this starter code**. **Do not change any Class or Method names or signatures, otherwise you will not pass the automatic Tur²Bo Grader tests.** **Usage of any external libraries other than the given one is forbidden and will result in failing the Tur²Bo Grader tests.**

Step 1: Reading the Input File

Given a list of tasks as a *json* file (you are encouraged to parse it using the *gson* library) you will have to fill an array with *Task* objects. You are required to get the file name as the first

program argument. A sample input *json* is illustrated in the figure below:


```
[
  {
    "name": "Send the excavator",
    "start": "09:00",
    "duration": 3,
    "importance": 30,
    "urgent": false
  },
  {
    "name": "Organize school for children",
    "start": "17:00",
    "duration": 1,
    "importance": 6,
    "urgent": false
  },
  {
    "name": "Distribute warm meal",
    "start": "10:00",
    "duration": 5,
    "importance": 61,
    "urgent": true
  },
  {
    "name": "Pack humanitarian aid",
    "start": "12:00",
    "duration": 1,
    "importance": 45,
    "urgent": false
  },
  {
    "name": "Check if all tents have a heater",
    "start": "16:00",
    "duration": 1,
    "importance": 12,
    "urgent": false
  },
  {
    "name": "Register missing persons",
    "start": "16:00",
    "duration": 1,
    "importance": 6,
    "urgent": false
  }
]
```


Step 2: Sorting the Array

You are expected to sort the array you have filled in increasing order of the finish times using *Arrays.sort()* method. You should implement a method in the *Task* class named *getFinishTime()*, which returns the finish time of the task as a string (in the same format as the start time). After implementing *getFinishTime()*, you should implement the *compareTo()* method of *Comparable* interface in the *Task* class, which uses *getFinishTime()*.

Step 3: Calculating Weights

Weight of each task should be calculated using the formula below:

$$weight = \frac{importance \times (urgent\ ?\ 2000 : 1)}{duration}$$

The weight of a task should be acquirable within your program using *getWeight()* method of the *Task* class.

Step 4: Finding Compatible Tasks

To tackle this problem, you must fill a compatibility array such that $compatibility[i]$ holds the index of the first compatible task before the task i. Compatibility between two tasks a and b can be defined as follows; task a is said to be compatible with task b if the finish time of task a, namely f_a, is less than or equal to the start time of the task b, namely s_b.

Considering the sample input given in Fig. 1, the value of $compatibility[4]$ should be 2, as it is the first task compatible with task 4. Since the array is already sorted by the finish time, you should use the *binary search* algorithm to find compatible tasks for each task.

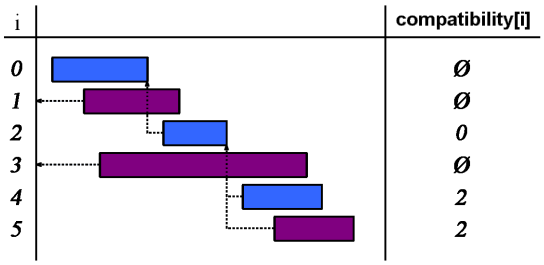


Figure 1: $compatibility[i]$ values w.r.t. i values

Step 5: Calculating the Total Maximum Weight Value

Independently of the optimization approach taken (dynamic or greedy), there are only a few cases to be considered while implementing the solution.

Case 1: task i is in the solution *Case 2: task i is not in the solution*

Considering the first case, if task i is in the solution, then the weight of task i must be combined with the result of the recursive call for the **first compatible task** before i.

Considering the second case, if task i is not in the solution, then the procedure should check the task before i.

On each recursive call, the procedure should find the maximum of those two cases. To calculate the maximum weight that can be acquired for all jobs, the procedure should calculate all maximum values for all N tasks in the array. To calculate the maximum value for task i, it is required

that the maximum values for $i-1$ and *compatibility*[i] are known, which creates overlapping subproblems.

To exploit this property of the problem, you are required to create and fill a double array *maxWeight*, which will store the maximum values for each task i such that $i = 0$, $i = 1$, ..., $i = N - 1$.

Sample Printed Output

Your program should output every recursive call like the sample given below:

```
Calculating max array
---------------------
Called max(5)
Called max(4)
Called max(2)
Called max(-1)
Called max(1)
Called max(0)
Called max(-1)
Called max(-1)
Called max(0)
Called max(-1)
Called max(-1)
Called max(3)
Called max(2)
Called max(2)
Called max(4)
```

Step 6: Finding a Dynamic Programming Solution

The objective of this part of the assignment is to schedule tasks such that the total weight of the scheduled tasks is maximized. After properly filling the *maxWeight* array, another pass is required to find a solution. You should fill the given instance variable named *planDynamic* that is defined as an array of *Task*s. There are, again, a few cases to consider while implementing this part.

Case 1: It is better to include task i in the solution *Case 2: It is worse to include task i in the solution*

Considering the first case, i should be added to the *planDynamic*, and another recursive call is required for inspecting the task *compatibility*[i]. You should use the *max* array to determine if it is better to include task i or not.

Considering the second case, another recursive call is required for inspecting the task $i - 1$.

Sample Printed Output

Your program should output every recursive call and then the created plan to the console like the sample given below:

```
Calculating the dynamic solution
--------------------------------
Called findSolutionDynamic(5)
Called findSolutionDynamic(4)
Called findSolutionDynamic(3)
Called findSolutionDynamic(2)

Dynamic Schedule
----------------
At 10:00, Distribute warm meal.
At 16:00, Check if all tents have a heater.
At 17:00, Organize school for children.
```

Step 7: Greedy Implementation

In this part, the objective is slightly changed. You are expected to schedule tasks such that the maximum number of tasks gets done. For this part, assume that no weights are present for any of the tasks. Using your previously developed structure and correctly sorted array of tasks, you are expected to implement a greedy algorithm to tackle this modified version of the problem.

The algorithm should work in the following way: After sorting as explained in Step 2, for each task i starting from 1 **as the first task should always get selected**, you should check if the task is compatible with the most recently selected task, and if it is compatible, then you are required to simply add the task to another solution array called *planGreedy*.

Sample Printed Output

Your program should output every addition:

```
Greedy Schedule
---------------
At 09:00, Send the excavator.
At 12:00, Pack humanitarian aid.
At 16:00, Check if all tents have a heater.
At 17:00, Organize school for children.
```

Important Notes

- **Brute force solutions for dynamic and greedy programming parts will not be accepted and will be graded with 0.**
- Do not miss the submission deadline: **20.04.2023, 23:59**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (https://piazza.com/hacettepe.edu.tr/spring2023/bbm204), and you are supposed to be aware of everything discussed on Piazza.

- You should run your code using the commands below:

  Linux

  ```
  javac -cp *.jar *.java -d .
  java -cp .:* Main io/input/input6.json
  ```

  Windows Powershell

  ```
  javac -cp *.jar *.java -d .
  java -cp '.;*' Main io/input/input6.json
  ```

  Windows CMD

  ```
  javac -cp *.jar *.java -d .
  java -cp .;* Main io/input/input6.json
  ```

- The assignment must conform to the given **coding template**.
- You must test your code via **Tur^{2}Bo Grader** comingsoon (**does not count as submission!**).
- To acquire full points, your code should pass one or more unit tests for each step.
- You will submit your work via https://submit.cs.hacettepe.edu.tr/ with the file hierarchy given below:

  b<studentID>.zip
  - Main.java <FILE>
  - Task.java <FILE>
  - Planner.java <FILE>
  - *.java <FILE> (*optional*)

- The name of the main class that contains the main method should be **Main.java**. You **must use this starter code**. **Do not change any Class or Method names or signatures, otherwise you will not pass the autograding tests**.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).
- **Usage of any external libraries other than gson library is forbidden.**
- **Do not use any packages. Only use the default package. Usage of a package will likely result in a grade of ZERO.**
- Do not submit any .jar or .class files.

Grading Policy

Grading will be strictly performed using autograder. Requests for manual grading will not be accepted!

- Submission: 1%
- Output tests: 9%
- Unit tests: 90% (18% each)

Unit Tests

The methods given in the starter code below should be implemented to pass our unit tests:

1. **CalculateCompatibilityTest:** This test checks if the array *compatibility* is filled with appropriate values. *Planner.calculateCompatibility()* should be implemented correctly.
2. **CalculateMaxTest:** This test checks if the array *maxWeight* is filled with appropriate values. *Planner.calculateMaxWeight()* should be implemented correctly.
3. **PlanDynamicTest:** This test checks if the returned schedule from *Planner.planDynamic()* is correct and the array *planDynamic* is filled with appropriate values.
4. **PlanGreedyTest:** This test checks if the returned schedule from *Planner.planGreedy()* is correct and the array *planGreedy* is filled with appropriate values.
5. **CompareToTest:** This test checks if *Task.compareTo()* returns the correct comparison value so that *Task* objects are sorted by their finish times in increasing order.

Output Tests

6. **ConsoleOutputTest:** This test checks if the printed console output matches the expected console output.

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in suspension of the involved students.