Spring 2021 - BBM204: Software Practicum Assignment 2

Hacettepe University Computer Engineering Department

Due Date: April 14, 2021 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.

The Story

It was a sunny day at the park, Muscle Man and Hi Five Ghost were cutting the grass while Skips was trying to fix the golf cart which seemed ruined. Rigby looked at Mordecai and said:

“Hey dude, I heard that they brought a new arcade machine. Wanna bail and try it out?”

“Sure, let’s bail out and pla...”

He was not able to finish the sentence as Benson angrily stepped in and started to yell at them as they were already playing video games inside the house, instead of doing their chores.

“What’s wrong with you two!? You are supposed to be working. Either finish all the chores from the previous days or YOU ARE FIRED!” yelled Benson.

“Okay Benson” said Mordecai but also he saw the impossibility of the task at hand.

They were slacking off for days and there was no way they would be able to finish all those remaining chores in a day.

“Uhh, Benson...” said Mordecai, “I think we have a problem”.

Figure 1: Mordecai and Rigby in their natural habitat

“What is it now!?” said Benson, he was looking very red and angry.

“Listen, we have A LOT of work to do and there is no way we can finish it in one day.”, said Mordecai with a trembling voice.

Benson seemed thoughtful, he did not think about this aspect before. They really were slacking off for who knows how many days and the chores had piled up.

He realized that for the park to be bearable enough for visitors to come, he had to choose some of their chores and explicitly tell them which jobs to do. He took a careful look at their schedule and seemed confused.

“How would I be able to pick the most important jobs in this list?” he thought to himself. “Only if there was a way for me to input them to the computer and get a list for each day...” he continued. Then it occurred to him, Techmo was able to solve very difficult computer problems before. “I should call him and ask for help”, he thought. This way, the park would be a bearable, nice place full of people as soon as possible. He then turned to those two slackers and said:

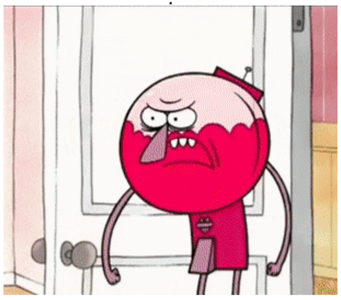


Figure 2: Benson doing what he loves

“Just wait you too, be grateful that I don’t fire you right away. I will be able to come up with a good solution I think.”

He called Techmo after sending them off to the arcade machine. He started to explain the problem to him:

“So listen, tech guy. I have a pile of work laying around, which needs to be taken care of. So here is the deal, you help me organize those, and I pay you the percentage I am going to cut off of those slackers” said Benson, he was still red with anger.

“Deal”, said Techmo thoughtfully. “But are there any constraints to those assignments?”

“Yeah” says Benson, “as a matter of fact there is. There are some important jobs that Mr. Maellard wants to be done until the end of the day so those must be included in the list no matter what. Other than that, jobs have some associated importance values to them thanks to Skips. As the last constraint, you will have to consider that I am paying every worker here 10$ per hour, so the longer it takes them to do, the longer we will lose money, considering this, if a job takes a long time to finish, we can postpone them as they will have less importance. I need you to code me a computer program that I can use so that I can plan ahead of the day and those slackers cannot outsmart me like they did today. Please make sure that your code works fast as we don’t have an amazing computer at the park.”

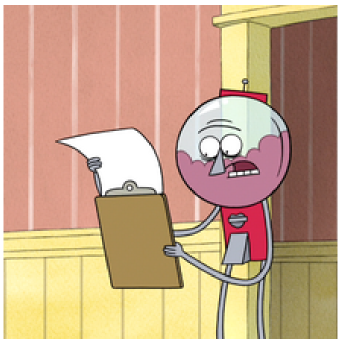
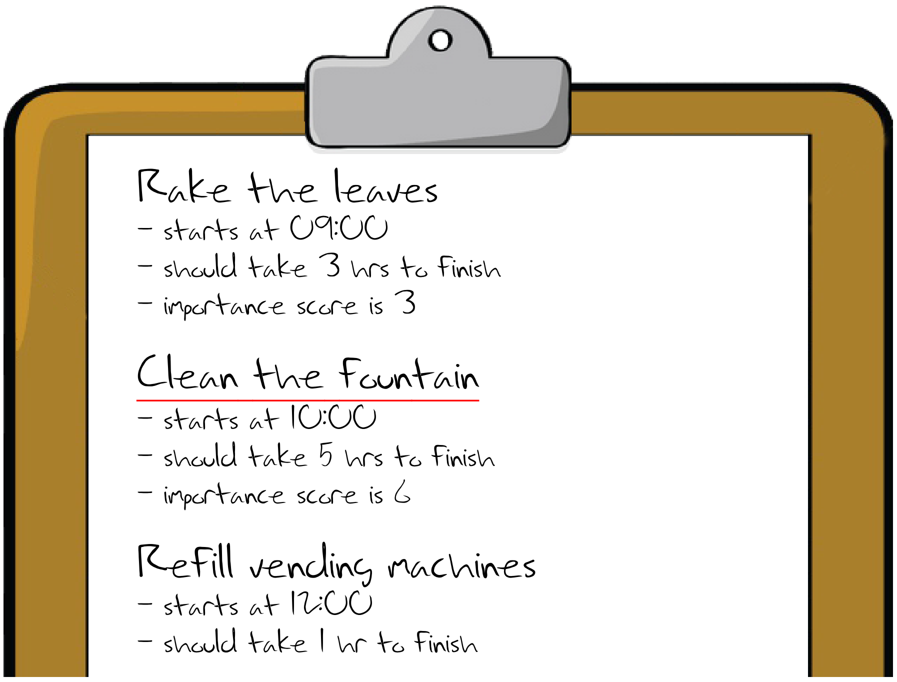


Figure 3: Benson looking confused

“Alright”, said Techmo, “but I need some time to think”.

“Here is a list of jobs from the previous day” said Benson, while showing his clipboard to Techmo.

Techmo took a photo of the clipboard using his cybernetic eye and admired the amount of detail. Can you help Techmo make some money?


Rake the leaves
- starts at 09:00
- should take 3 hrs to finish
- importance score is 3

Clean the fountain
- starts at 10:00
- should take 5 hrs to finish
- importance score is 6

Refill vending machines
- starts at 12:00
- should take 1 hr to finish

Figure 4: Benson’s Clipboard

1 Problem Definition

1.1 Step 1: Reading the Input File

Given a list of assignments as a *json* file (you are encouraged to parse it using the *gson* library, trust me it is waay easier ☺) you will have to fill an array with *Assignment* objects. You are required to get the file name as the first program argument. A sample input *json* is illustrated in the figure below:

```json
[
  {
    "name": "Rake the leaves",
    "start": "09:00",
    "duration": 3,
    "importance": 30,
    "maellard": false
  },
  {
    "name": "Organize a movie night",
    "start": "17:00",
    "duration": 1,
    "importance": 6,
    "maellard": false
  },
  {
    "name": "Clean the fountain",
    "start": "10:00",
    "duration": 5,
    "importance": 61,
    "maellard": true
  },
  {
    "name": "Refill vending machines",
    "start": "12:00",
    "duration": 1,
    "importance": 45,
    "maellard": false
  },
  {
    "name": "Wash Benson's car",
    "start": "16:00",
    "duration": 1,
    "importance": 12,
    "maellard": false
  },
  {
    "name": "Help Pops find where the sky is ",
    "start": "16:00",
    "duration": 1,
    "importance": 6,
    "maellard": false
  }
]
```

1.2 Step 2: Sorting the Array

You are expected to the sort the array you have filled in a non-decreasing order of the finish times using *Arrays.sort()* method. You should implement a method in the *Assignment* class named *getFinishTime()* which returns the finish time of the assignment as a string. After implementing *getFinishTime()*, you should implement a *compareTo()* method in the *Assignment* class which uses the former.

1.3 Step 3: Calculating Weights

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

$$weight = \frac{importance \times (maellard\ ?\ 1001 : 1)}{duration}$$

The weight of an assignment should be acquirable within your program using *getWeight()* method of the *Assignment* class.

1.4 Step 4: Finding Compatible Assignments

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

Considering the sample input given in the Fig. 5, the value of $C[4]$ should be 2, as it is the first assignment that is compatible with the assignment 4. Since the array is already sorted by the finish time, you should use *binary search* algorithm for finding compatible assignments for each of the assignments.

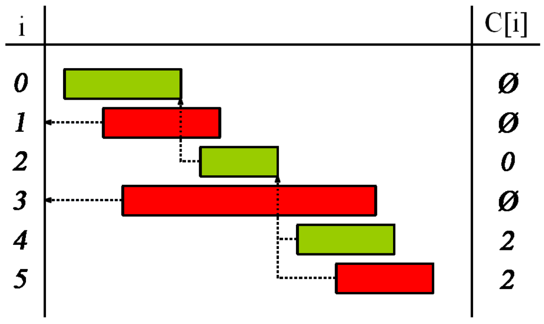


Figure 5: $C[i]$ values w.r.t. i values

1.5 Step 5: Calculating the Total Maximum Weight Value

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

Case 1: assignment i is in the solution
Case 2: assignment i is not in the solution

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

Considering the second case, if assignment i is not in the solution, then the procedure should check the assignment 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 assignments in the array. To calculate the maximum value for assignment i, it is required that the maximum values for $i-1$ and $C[i]$ are known, which creates overlapping subproblems.

To exploit this property of the problem, you are required to create and fill an integer array *max* which will store the maximum values for each assignment i such that $i = 0,\ i = 1,\ ...,\ i = N-1$.

1.6 Sample Printed Output

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

```
calculateMax(5): Prepare
calculateMax(4): Prepare
calculateMax(2): Prepare
calculateMax(1): Prepare
calculateMax(0): Zero
calculateMax(0): Zero
calculateMax(3): Prepare
calculateMax(2): Present
calculateMax(2): Present
calculateMax(4): Present
```

Prepare should be printed when the value is being calculated for the first time. *Present* should be printed when the value is being used from the stored array. *Zero* should be printed when the *calculateMax()* method is being called with the parameter 0.

1.7 Step 6: Finding a Solution

After filling the *max* array properly, another pass is required to find a solution. You should define and fill an *Assignment* array called *solutionDynamic*. There are again a few cases to consider while implementing this part.

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

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

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

1.8 Sample Printed Output

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

```
findSolutionDynamic(5)
Adding Assignment{name='Organize a movie night',...} to the dynamic schedule
findSolutionDynamic(4)
findSolutionDynamic(3)
Adding Assignment{name='Wash Benson's car',...} to the dynamic schedule
findSolutionDynamic(2)
Adding Assignment{name='Clean the fountain',...} to the dynamic schedule
```

Note that you should print assignments using the *toString()* method given in the coding template.

1.9 Step 7: JSON Output

You are required to output the *solutionDynamic* array to a *json* file named “solution_dynamic.json”. Usage of the *gson* library is encouraged.

For instance, for the sample input given in the figure in section 1.1, the following sample output should be acquired:

```
[
  {
    "name": "Clean the fountain",
    "start": "10:00",
    "duration": 5,
    "importance": 61,
    "maellard": true
  },
  {
    "name": "Wash Benson's car",
    "start": "16:00",
    "duration": 1,
    "importance": 12,
    "maellard": false
  },
  {
    "name": "Organize a movie night",
    "start": "17:00",
    "duration": 1,
    "importance": 6,
    "maellard": false
  }
]
```

1.10 Step 8: Greedy Implementation

Assume that no weights are present for any of the assignments. Using your previously developed structure and correctly sorted array of assignments, you are expected to implement a greedy algorithm to tackle a modified version of the problem.

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

1.11 Sample Printed Output

Your program should output every addition:

```
Adding Assignment{name='Rake the leaves', ...} to the greedy schedule
Adding Assignment{name='Refill vending machines', ...} to the greedy schedule
Adding Assignment{name='Wash Benson's car', ...} to the greedy schedule
Adding Assignment{name='Organize a movie night', ...} to the greedy schedule
```

Note that you should print assignments using the *toString()* method given in the coding template.

1.12 Step 9: JSON Output

You are required to output the *solutionGreedy* array to a *json* file named "solution_greedy.json". Again usage of the *gson* library is encouraged.

For instance, for the sample input given in the figure in section 1.1, the following sample output should be acquired:

```json
[
  {
    "name": "Rake the leaves",
    "start": "09:00",
    "duration": 3,
    "importance": 30,
    "maellard": false
  },
  {
    "name": "Refill vending machines",
    "start": "12:00",
    "duration": 1,
    "importance": 45,
    "maellard": false
  },
  {
    "name": "Wash Benson's car",
    "start": "16:00",
    "duration": 1,
    "importance": 12,
    "maellard": false
  },
  {
    "name": "Organize a movie night",
    "start": "17:00",
    "duration": 1,
    "importance": 6,
    "maellard": false
  }
]
```

Notes

1. You should submit your code using codepost.io.
2. You should submit your work until 14.04.2021, 23:59.
3. You should run your code using the below command:

   `java Main.java <inputfile>.json`
4. The assignment should conform to the given **coding template**.
5. To acquire full points, your code should pass one or more unit tests for each step.
6. No more notes, good luck to you all ☺

Grading

Grading will be performed in two parts, comments (10%) and automated tests (90%). Some of the automated tests that will be executed on your assignment is given below.

Unit Tests

1. **Main.readFile():** Check if the created *Assignment*[] is correct
2. **Main.writeOutput():** Check if it can create the correct file with the correct contents
3. **Scheduler.binarySearch():** Check if the method works as expected
4. **Scheduler.calculateC():** Check if the array C is filled with appropriate values
5. **Scheduler.calculateMax():** Check if the array *max* is filled with appropriate values
6. **Scheduler.findSolutionDynamic():** Check if the array *solutionDynamic* is filled with appropriate values
7. **Scheduler.scheduleDynamic():** Check if the returned schedule is correct
8. **Scheduler.scheduleGreedy():** Check if the returned schedule is correct and the array *solutionGreedy* is filled with appropriate values
9. **Scheduler():** Check if the constructor correctly throws the specified exception. Check if the assignmentArray property is initialized after constructor call.
10. **Assignment.get*():** Check if they return the expected values
11. **Assignment.compareTo():** Check if it returns the correct comparison value
12. **Assignment.toString():** Check if it returns the correct value

Output Tests

13. **JSON Files:** Check if the files are valid JSON files which include the same value as Assignment.schedule*() methods
14. **Printed Output:** Check if the recursive methods are working correctly

Inputs

1. Array with no elements is given
2. Array with only one element is given
3. Multiple tests with different inputs, checking for the correct output

Policy

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your classmates about the given assignments, 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. I wonder if anybody reads these thingies. In short, turning in work of someone else (from 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.