Assignment for Class Computer Science for Big Data D3

Combinatorial Algorithms, Points: 7

Scrum is a method to organize work for smaller teams as weekly sprints during which Scrum-tasks are cooperatively planned, assigned, and performed. Scrum helps teams to better understand workflows and encourages transparency in the team, see: https://www.scrum.org/resources/what-is-scrum .

An essential part of Scrum is estimating effort and impact of individual tasks planned for the next Sprint. Tasks can be creating new features or resolving bugs. The Scrum master maintains tasks in the Sprint backlog. During Sprint planning, the team estimates tasks in terms of their **benefits** (based on urgency, usefulness or contribution to the product) and in terms of **cost** (effort to complete a task, see also https://www.atlassian.com/agile/project-management/estimation, using a scheme with story points).

The following Scrum tasks have been extracted from the Scrum management system with estimates of benefits (blue numbers) and costs (red numbers) already assigned by the team:

```
scrum_tasks = [
    # task_id, benefit, cost estimates, description
    ('id_428', 50, 10, 'bug #3150: left column alignment broken'),
    ('id_528', 25, 20, 'bug #4254: logout leaves hourglass on screen'),
    ('id_644', 30, 10, 'idea: move shopping cart icon to the right'),
    ('id_222', 15, 80, 'investigate #4259: logout slow'),
    ('id_680', 30, 60, 'feat #4386: color scheme for fall skin'),
    ('id_220', 30, 20, 'feat #4255: internationalize for Spanish'),
--> cut off line for team capacity of 200
    ('id_421', 30, 20, 'feat #4256: internationalize for Russian'),
    ('id_682',120, 30, 'bug #4002: empty password crashes backend'),
    ('id_368', 30, 10, 'fix comments in optimization module'),
    ('id_932', 30, 25, 'bug #4846: exception thrown for empty field'),
    ('id_387', 30, 25, 'feat #8427: new option to change color scheme'),
]
```

Since the Team is working on multiple projects simultaneously, it has a weekly capacity of 200 story points for this project creating a cut off line after 6 tasks in the list above when this capacity is exceeded (sum(10, 20, 10, 80, 60, 20) = 200).

Simple sequential execution of tasks will yield an expected benefit of 180 points during the next Sprint (sum(50, 25, 30, 15, 30, 30) = 180).

Since there are no dependencies marked between tasks, the Scrum Master wonders whether he can do better by rearranging tasks such that they yield a higher benefit than 180 and still meet the capacity constraint of 200.

1.) In this example, what is the highest possible benefit that can be achieved by rearranging tasks under consideration of the capacity constraint (any arrangement of tasks with cost <= 200)? [1 Pts]

2.) How would you (in the example) manually determine the task arrangement that yields the highest benefit under the capacity constraint? Briefly describe your approach. [1 Pts]

3.) What do you think is the most relevant limiting factor in your approach? [1 Pts]

In computer science, one approach to the problem is called *Brute-force search*, which is based on the idea to generate all possible variations of solutions and evaluate them to return the best solution, see https://en.wikipedia.org/wiki/Brute-force_search .

For smaller problems sizes (such as in our example), *Brute-force search* is a feasible solution because it always finds the best (optimal) solution by performing an exhaustive search of the problem space. Furthermore, it is usually easy to implement and consists of the following steps:

1. Generate all possible solutions.
2. Evaluate each solution candidate:
    a. does the solution candidate meet constraints? If not, disregard.
    b. otherwise, calculate a benefit score for the solution candidate:
        i. if this score is greater than the score of previous solutions, put the solution candidate at the top spot.
3. Return the highest scoring solution candidate.

This simple approach belongs to the category of generative algorithms since it relies on generating and evaluating potential solutions. Continuously looking for a higher-scoring solution and selecting it when found is also known as *Greedy Algorithm*, see https://en.wikipedia.org/wiki/Greedy_algorithm, which has many applications in Computer Science.

4.) When we apply this idea to the example of Scrum tasks, what is the underlying mathematical or combinatorial concept "to generate all possible solutions" of Scrum tasks? – Is it :

a. a subset,
b. a powerset or
c. a permutation of tasks? [1 Pts]

5.) The following function generates all possible variations of numbers from a list, for example [0, 1, 2]. It can be used to index scrum_tasks[ ] and hence can be used in our example:

```
def generate(_items: [int]) -> [[int]]:
    res = [[]]
    for item in _items:
        new_set = [r+[item] for r in res]
        res.extend(new_set)
    return res                --> vs. yield res ?

result = generate([0,1,2])
print(str(result))            --> vs. str(list(result))

output:
[[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2]]
```

6.) Briefly describe how the implementation of the generate() - function works. [1 Pts]

7.) How many elements does the generated list contain for the specific example of scrum_tasks[ ] and how many for the general case with an input list of *n* elements? [1 Pts]

8.) Describe the difference between **`return res`** and **`yield res`** in the generate() - function.

a. What is returned from function generate() in either case?
b. When is the list actually generated in either case? [1 Pts]

Combinatorial Algorithms, Points: 7 + 7 EP

Using this method, the optimal combination of scrum tasks is an arrangement that yields the highest benefit score while remaining within the cost limit of 200.

1.) Write a python function that solves the problem and returns a tuple describing the optimal solution for arranging scrum tasks. The returned solution should be comprised of its total benefit, total cost and the task list arrangement provided as a list [ ] of indexes in `scrum_tasks[]`. [2 Pts]

```
def optimize_scrum_tasks(_cost_constraint: int) -> (int, int, [int]):
  _sol_benefit, sol_cost, _solution = 0, 0, []
  # ...code
  # example: return 230, 60, [0, 2, 7, 8]
  return _sol_benefit, _sol_cost, _solution
```

2.) Test your implementation with varying cost constraints producing different solutions:

```
for cost_limit in [9, 10, 20, 30, 40, 50, 60, 100, 150, 200, 1000]:
    sol = optimize_scrum_tasks(cost_limit)
    print('cost_constraint:' + str(cost_limit).rjust(5) + \
          ',  b/c: ' + str(str(sol[0]) + '/' + str(sol[1])).rjust(7) + \
          ',  solution: ' + str(sol[2]))

==> example outputs
cost_constraint: 20, b/c:  80/20,  solution: [0, 2]
...
cost_constraint: 50, b/c: 200/50,  solution: [0, 2, 7]
cost_constraint: 60, b/c: 230/60,  solution: [0, 2, 7, 8]
```

The solution for cost constraint 20, for example, found a task allocation [0, 2] yielding a benefit of 80 under a cost of 20 with [0, 2] as indexes in `scrum_tasks[]`:

```
0: ('id_428', 50, 10, 'bug #3150: left column alignment broken'),
1: ...
2: ('id_644', 30, 10, 'idea: move shopping cart icon to the right'),
```

Also compare the other solutions. [1 Pts]

3.) Now consider the optimal task allocation under the cost constraint of 200 and compare it with the initial sequential allocation yielding a total benefit of 180 (at a cost of 200).

- What is the optimized total benefit?
- What is the optimized total cost? [1 Pts]

4.) Append more tasks to `scrum_tasks[]` as shown below. Leave tasks *id_564*, *id_412* in comments. Re-run the program. What is now the optimized solution under the cost constraint of 200 :

b) total benefit? b) total cost? c) Task arrangement (as index [ ])? [1 Pts]

```
  # append more tasks to scrum_tasks[]
  ('id_578', 15, 60, 'feat #5206: ...'),
  ('id_390', 20, 20, 'feat #9524: ...'),
  ('id_340', 40, 10, 'feat #4954: ...'),
  ('id_234',100, 30, 'bug 2893: ...'),
  ('id_193', 50, 10, 'bug #3466: ...'),
  ('id_450', 40, 20, 'bug #3205: ...'),
  ('id_344', 60, 30, 'feat #4503: ...'),
  #('id_564', 20, 20, 'feat #7422: ...'),
  #('id_412', 30, 10, 'bug #4566: ...'),
]
```

5.) Uncomment tasks *id_564*, *id_412* and re-run the program. You will notice a significant slow-down in producing solutions. How would you explain this observation? [1 Pts]

6.) With the additional 9 tasks, the size of `scrum_tasks[]` increases from 11 to 20. How big is the expansion of the solution space that is being (exhaustively) searched by our *Brute-force search* algorithm? [1 Pts]

Extra points:

Obviously, *Brute-force search* is limited to small problems sizes (e.g. task lists of up to 20). Other approaches exist that solve the problem in (almost) polynomial time.

The following EP-tasks are <u>not obligatory</u> and only for those interested in earning extra points and / or in learning more about these faster methods.

EP 1.) Learn about the *Knapsack problem* and *Linear Programming (LP)*, if you are not aware:

- MIT OpenCourseWare, Victor Costan, R21. Dynamic Programming: Knapsack Problem, 1:10h, https://www.youtube.com/watch?v=wFP5VHGHFdk .

EP 2.) Develop a Dynamic Programming (DP)-implementation for function `optimize_scrum_tasks()`. Demonstrate it with the test data set. [6 Pts]

EP 3.) Demonstrate the solution for the test-data set in `scrum_tasks[]` with adding extra tasks 4x (copy & paste) bringing the task list to size: 11 + 4 x 9 = 47 . [1 Pts]