Assignment 11.1: Traveling Salesman Problem and Branch-and-Bound (4 points)

In traveling salesman problem, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. The problem is known to be NP-hard. See Sec. 7.4.4 of the background material for the introduction to the problem.

No efficient (polynomial time) algorithm is known for the problem. With a small amount of cities we can test all the possible permutations using backtracking and select the shortest route. However, the computation time quickly becomes unbearable when the amount of cities increases. Using branch-and-bound we can reduce the computation time remarkably by reducing the number of solutions to be investigated. This approach still always produce the optimal solution, but can be used with a larger number of cities than the simple backtracking.

Create a Python function `salesman(city_map: list)` that solves the traveling salesman problem using branch-and-bound algorithm (see Week 8 material and slides). The function takes a distance matrix (`city_map[i][j]` tells the distance between ith and jth city) of the cities as an input value and returns a order of the cities traversed in a list. Cities are labeled as integers starting from zero and the traverse always starts from the first city (0th city).

Target:

- The function finds the optimal route with 10 cities in less than 3 seconds on CodeGrade (2 points)
- The function finds the optimal route with 10 cities in less than 1 second on CodeGrade (4 points)

Note, that testing all permutations with simple backtracking is too slow. In order the achieve either of the targets, the function needs to use branch-and-bound. To obtain 4 points, a more accurate estimate for the lower bound is needed.

A code template with an example program:

```
def salesman(city_map):
    # TODO


if __name__ == "__main__":

    cost = 0

    city_map = [
    #     0   1   2   3   4
        [ 0, 12, 19, 16, 29],   # 0
        [12,  0, 27, 25,  5],   # 1
        [19, 27,  0,  8,  4],   # 2
        [16, 25,  8,  0, 14],   # 3
        [29,  5,  4, 14,  0]    # 4
        ]

    path = salesman(city_map)
    for i in range(len(city_map)):
        cost += city_map[path[i]][path[i+1]]

    print(path)
    print(cost)
```

Output:

```
$ python salesman.py
[0, 1, 4, 2, 3, 0]
45
```

Submit your solution in CodeGrade as `salesman.py`.

Assignment 11.2: The Bin Packing Problem and Approximation Algorithms (4 points)

When the size of the problem increases even the more sophisticated exact algorithms (e.g. branch-and-bound and dynamic programming) become insufficient. In such cases, we typically need to settle for approximation algorithms. Such algorithms do not (always) provide the optimal solutions, but usually provide a good solution fast.

Bin packing problem: given a set of n items with sizes $[s_1, s_2, s_3, \ldots, s_n]$, find a way to pack the items to bins with size S so that the total amount of bins is minimized.

Design and implement an approximation algorithm for the bin packing problem. The algorithm must be able to find a solution for very large set of items. Therefore, brute-force approach or other algorithms that always find the optimal solution are not suitable. You will get points based on how close the solutions your algorithm produces are to the optimal solutions.

Create a function `binpack(items: list, S: int)` in Python. The function takes list of items and a maximum bin size S as an input value and returns a list of all bins. Each bin is a list of items.

Target:

- the algorithm is able to find an (approximate) solution in less than 1 seconds when n, and
- the solution (number of bins) is at most 50 larger than the optimal amount of bins (1 point)
- the solution (number of bins) is at most 5 larger than the optimal amount of bins (2 points)
- the solution (number of bins) is at most 1 larger than the optimal amount of bins (4 points)

simple solutions produce almost always much better solution than the theoretical worst case.

Note that since there are no one unique correct solution (packing), it is not possible to use input-output tests in CodeGrade. This might make it challenging to see purely based on the AutoTest results, why the code is not working correctly. We recommend to test you code locally with different items and bin sizes if it does not pass the CodeGrade tests. Note that you can freely modify code under "`if __name__ == "__main__":`" as this part is ignored by the CodeGrade AutoTest.

A code template with an example program:

```
# binpack.py


def binpack(items, S):
    # TODO


if __name__ == "__main__":

    items = [9, 3, 3, 6, 10, 4, 6, 8, 6, 3]
    B = 10

    bins = binpack(items, B)

    for i in range(len(bins)):
        print(f"bin {i+1}: {bins[i]}")
```

One possible output (you code might have a different output and still work equally well or better):

```
$ python binpack.py
bin 1: [9]
bin 2: [3, 3, 4]
bin 3: [6, 3]
bin 4: [10]
bin 5: [6]
bin 6: [8]
bin 7: [6]
```

Submit your solution in CodeGrade as `binpack.py`.

Last modified: Friday, 22 November 2024, 1:03 PM