שאלה 1 מטלה 4

```python
def egalitarian_allocation(valuations: list[list[float]]):

    num_agents = len(valuations)
    num_resources = len(valuations[0])
    variables = []
    utilities = []

    for i in range(num_agents):
        utility = 0
        for j in range(num_resources):
            variables.append(cvxpy.Variable(num_agents, integer=True)) # We must make sure that we
            utility += variables[j][i] * valuations[i][j]
        utilities.append(utility)

    min_utility = cvxpy.Variable() # So we can make sure they all at least the minimum amount
```

נשים לב שהוספנו integer=True על מנת שנקבל מספרים שלמים (אין אפשרות לקבל חצי חפץ).

```python
fixed_constraints = \
    [variables[i][j] >= 0 for i in range(num_resources) for j in range(num_agents)] + \
    [variables[i][j] <= 1 for i in range(num_resources) for j in range(num_agents)] + \
    [utilities[i] >= min_utility for i in range(num_agents)] + \
    [sum(variables[i]) == 1 for i in range(num_resources)]

prob = cvxpy.Problem(cvxpy.Maximize(min_utility), constraints=fixed_constraints)
start_time = time.time()
prob.solve(solver=cvxpy.GLPK_MI)  # changed to GLPK_MI as we are dealing with linear programming and whole numbers
end_time = time.time()
execution_time = end_time - start_time

for i in range(num_agents):
    print(f"player {i} gets: ", end=" ")
    for j in range(num_resources):
      if(variables[j][i].value > 0):
            print(f"item {j} ({round(variables[j][i].value)*valuations[i][j]}), ", end="")
    print()

return execution_time, num_resources
```

השימוש ב GLPK_MI(mixed-integer solvers) במקום ECOS הוא בשביל שנוכל לחשב מספרים שלמים.
בנוסף הוספנו חישוב של כמות הזמן שלוקח לפתור את הבעיה והחזרנו את כמות הזמן ומספר המשאבים לצורך הגרף של סעיף ב.

```
[10] execution_times_discrete = []
     num_resources_list_discrete = []


     def run_discrete(valuations: list[list[float]]):
       execution_time, num_resources = egalitarian_allocation(valuations)
       execution_times_discrete.append(execution_time)
       num_resources_list_discrete.append(num_resources)
```

```
[11] execution_times_continuous = []
     num_resources_list_continuous = []


     def run_continuous(valuations: list[list[float]]):
       execution_time, num_resources = egalitarian_allocation_continuous(valuations)
       execution_times_continuous.append(execution_time)
       num_resources_list_continuous.append(num_resources)
```

בחלק ב' יצרנו ארבעה מערכים, 2 בשביל החפצים הבדידים ו2 בשביל הרציפים אשר שומרים את זמן ההרצה ואת מספר החפצים בכל סבב.

```
# Set a seed for reproducibility
np.random.seed(42)

# Create three arrays of changing sizes with random numbers
array1 = np.random.rand(100)
array2 = np.random.rand(100)
array3 = np.random.rand(100)

valuations=[array1, array2, array3]
```

```
run_discrete(valuations)

print("time: ", execution_times_discrete)
print("number: ", num_resources_list_discrete)
```

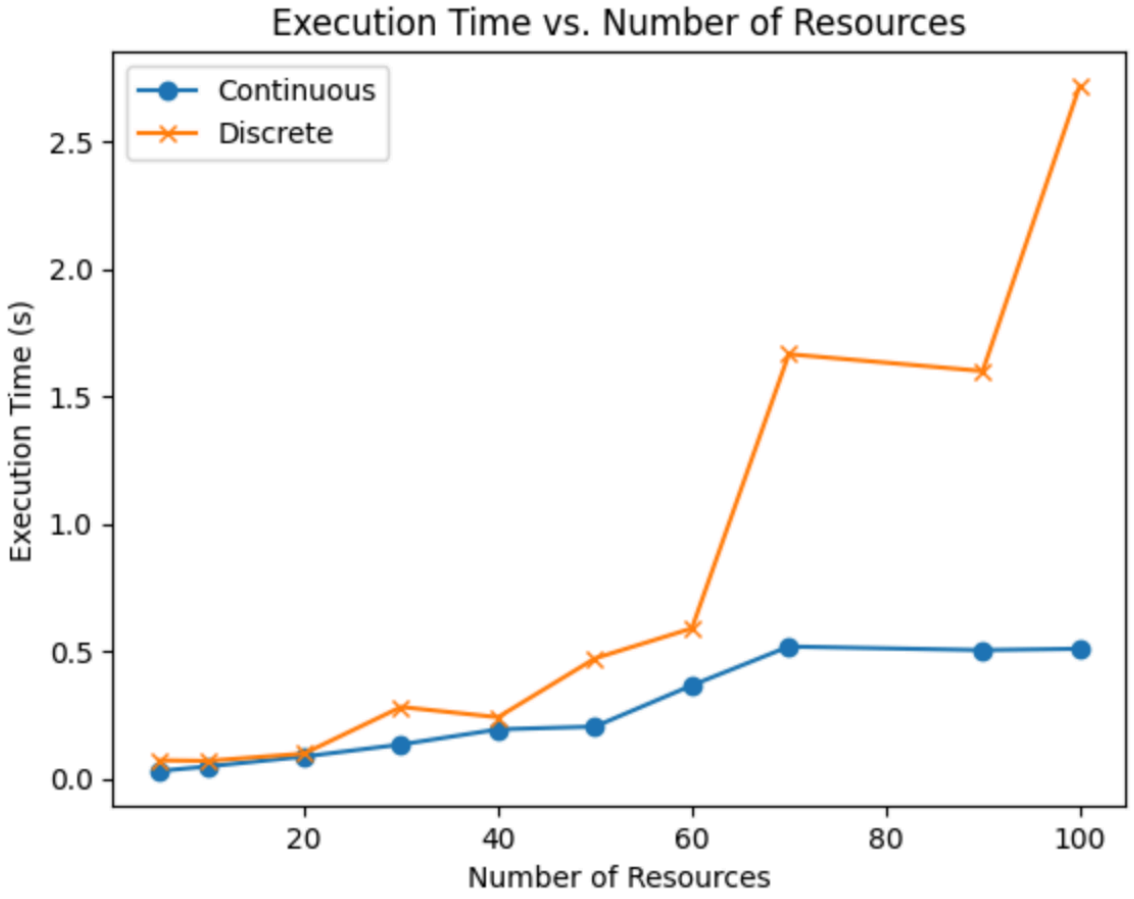
```
time:  [0.07171821594238281, 0.06940007209777832, 0.09813857078552246, 0.28099966049194336, 0.24066734313964844, 0.4
number:  [5, 10, 20, 30, 40, 50, 60, 70, 90, 100]
```

```
run_continuous(valuations)

print("time: ", execution_times_continuous)
print("number: ", num_resources_list_continuous)
```

```
time:  [0.029811620712280273, 0.047356605529785156, 0.08641505241394043, 0.13392186164855957, 0.19307875633239746, 0
number:  [5, 10, 20, 30, 40, 50, 60, 70, 90, 100]
```

בכל סבב שינינו את הvaluations ומספר החפצים אבל שמרנו על אותו input בשני המקרים על מנת שנוכל לקבל תוצאות מדויקות.



כפי שניתן לראות בגרף הנ"ל, חלוקה של חפצים בדידים לוקחת יותר זמן מאשר חלוקה של חפצים רציפים והפער יגדל ככל שמספר החפצים גדל.