Yu xinmiao 于欣淼 518021910792

VE477 Lab3

Q1

1. Sort and count

```
def merge_count(l1, l2):
    count, i, j = 0, 0, 0
    list_all = []
    while i < len(l1) and j < len(l2):
        if l1[i] <= l2[j]:
            list_all.append(l1[i])
            i += 1
        else:
            list_all.append(l2[j])
            count += len(l1) - i
            j += 1

    if i >= len(l1):
        list_all[len(list_all):] = l2[j:]
    else:
        list_all[len(list_all):] = l1[i:]
    return count, list_all


def sort_count(input_list):
    if len(input_list) == 1:
        return 0, input_list
    else:
        left = input_list[0:int(len(input_list) / 2)]
        right = input_list[int(len(input_list) / 2):]
        count1, list1 = sort_count(left)
        count2, list2 = sort_count(right)
        count3, list3 = merge_count(list1, list2)

    count = count1 + count2 + count3
    return count, list3
```

```
# sample input: a list of integer in one line. eg. 3 4 2 1

list0 = [int(item) for item in input().split()]
count_all, result_list = sort_count(list0)
print(count_all)
print(result_list)
```

2. Gale-Shapley

```
def gale_shaply(man_pre, woman_pre):
    w_engage = [-1 for i in range(len(woman_pre))]  # engaged
woman's pair
    m_free = [1 for i in range(len(man_pre))]  # all man free
    m_propose = [[0] * len(woman_pre) for i in range(len(man_pre))]
# whether a man propose to a woman
    while 1 in m_free:
        m_index = m_free.index(1)
        m_pre = man_pre[m_index]
        favorite = 0
        while m_propose[m_index][favorite] != 0:  # find the
favorite woman not proposed
            favorite += 1

        m_propose[m_index][favorite] = 1          # man propose to
woman
        wo_index = m_pre[favorite]

        if w_engage[wo_index] == -1:
            m_free[m_index] = 0
            w_engage[wo_index] = m_index
        else:
            m_dot = w_engage[wo_index]
            wo_pre = woman_pre[wo_index]
            if wo_pre.index(m_dot) < wo_pre.index(m_index):
                m_free[m_index] = 1
            else:
                w_engage[wo_index] = m_index
                m_free[m_index] = 0
                m_free[m_dot] = 1
```

```python
    return w_engage


man_num = int(input())
man_count, woman_count = 0, 0
man_list = []
while man_count < man_num:
    man_list.append([int(item) for item in input().split()])
    man_count += 1

woman_num = man_num
woman_list = []
empty_line = input()  # discard the empty line
while woman_count < woman_num:
    woman_list.append([int(item) for item in input().split()])
    woman_count += 1

w_en = gale_shaply(man_list, woman_list)
result = []
for i in range(len(w_en)):
    result.extend([[i, w_en.index(i)]])

print(result)
```

Q2

1.1 Fit the knapsack with the smallest items first

```python
def small_first(arr, target):  # fit the knapsack with the smallest
items first
    arr.sort()
    result = []
    for i in range(len(arr)):
        if target - sum(result) < arr[i]:
            return []
        else:
            result.append(arr[i])
        if sum(result) == target_sum:
            return result
```

When input

```
6
1 3 5 7
```

The output will be `[]`

```
➜  lab3 git:(master) ✗ python3 greedy.py
6
1 3 5 7
small first greedy algorithm: []
large first greedy algorithm: [1, 5]
➜  lab3 git:(master) ✗
```

1.2 Fit the knapsack with the largest items first

```
def large_first(arr, target):
    arr.sort(reverse=True)
    result = []
    for i in range(len(arr)):
        if target - sum(result) < arr[i]:
            continue
        else:
            result.append(arr[i])
        if sum(result) == target_sum:
            return result
```

When input

```
5
2 3 4
```

The output will be `[]`

```
➜  lab3 git:(master) ✗ python3 greedy.py
5
2 3 4
small first greedy algorithm: [2, 3]
large first greedy algorithm: []
➜  lab3 git:(master) ✗
```

2. Dynamic Programming

To properly solve Knapsack problem, we should use dynamic programming. The key idea is to break down the problem and store the result of the subproblems.

The implementation is attached in the next question.

Q3

The input size is from 2 to 150. To reduce the time cost, number of `timeit` is set as 100. `sum` is settled as *a ramdom number from 10-20 * length of input array.* The numbers in input array is selected randomly from 0 to 299.

The driver program to record the time for running each algorithm and plot the result as scatter plot is

```
import os
from tqdm import tqdm
import random
import matplotlib.pyplot as plt
import pandas as pd

numbers = [i for i in range(300)]
# input size: from 2 to xmax
smalltime, largetime, kptime = [], [], []
xmax = 150
for a in tqdm(range(2, xmax)):
    _sum = random.randint(10, 20) * a  # target number
    array_input = random.sample(numbers, a)
    with open("subset.in", "w+") as f:
        print(_sum, file=f)
        print(*array_input, sep=' ', file=f)
    # write the file used for input

    timetwo = os.popen("python3 greedy.py < subset.in").read()
    time_one = os.popen("python3 knapsack.py < subset.in").read()
    timelist = list(map(float, timetwo.split()))
    smalltime.append(timelist[0])
    largetime.append(timelist[1])
    kptime.append(float(time_one))

xlist = [i for i in range(2, xmax)]
```

```
df = pd.DataFrame(data=smalltime, index=xlist, columns=
["small_first"])
df["large_first"] = largetime
df["correct_algo"] = kptime
# print(df)
plt.scatter(xlist, smalltime, marker='o', color='blue',
label="small first", alpha=0.6)
plt.scatter(xlist, largetime, marker='x', color='red', label="large
first", alpha=0.6)
plt.scatter(xlist, kptime, marker='*', color='green',
label="correct algo", alpha=0.6)
plt.legend(loc='best')
plt.show()
```

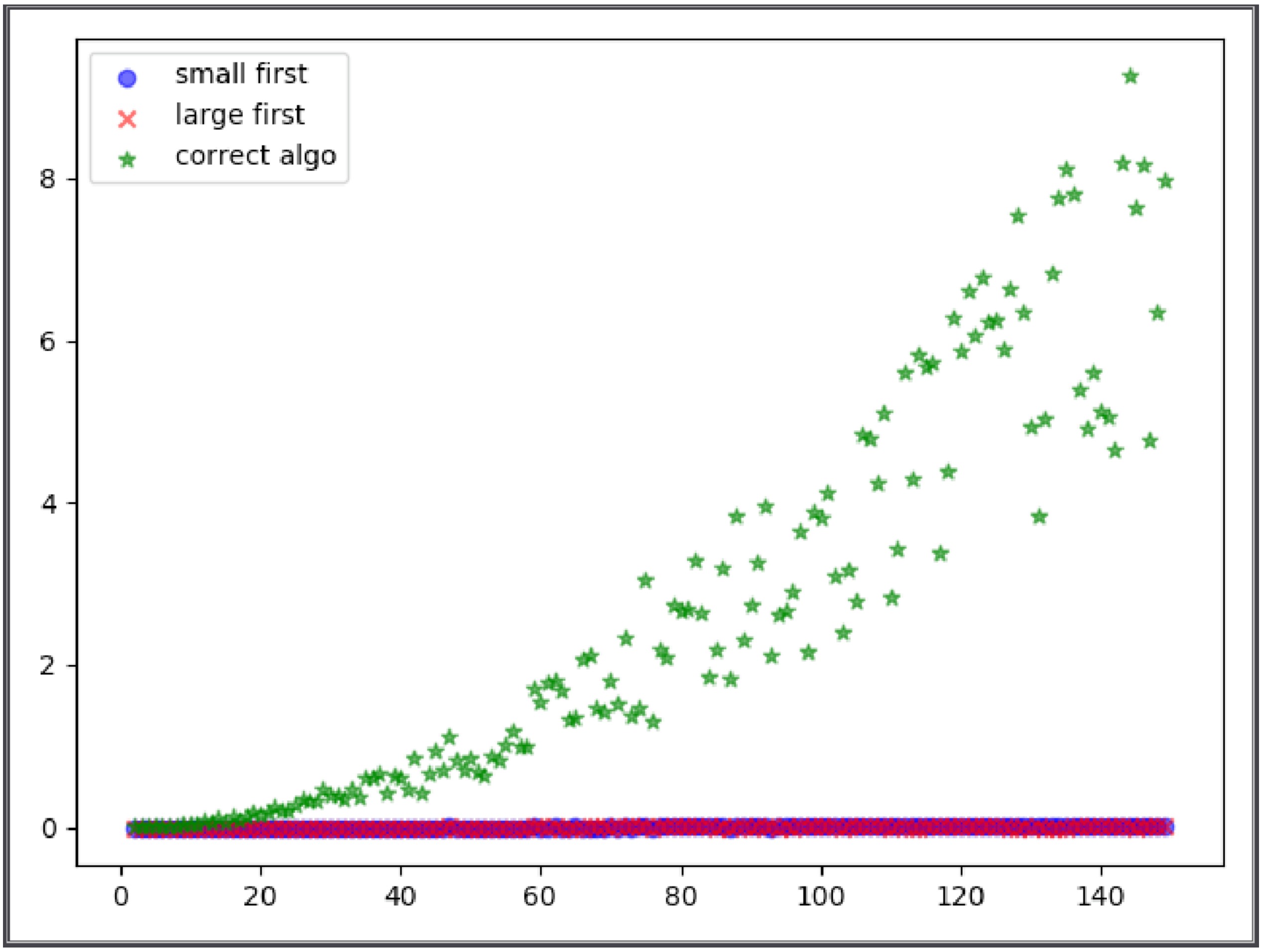
As in each program, `timeit` is added to test the time needed for each function.

```
#  greedy.py
def small_first(arr, target):  # fit the knapsack with the smallest
items first
    arr.sort()
    result = []
    for i in range(len(arr)):
        if target - sum(result) < arr[i]:
            return []
        else:
            result.append(arr[i])
        if sum(result) == target:
            return result


def large_first(arr, target):
    arr.sort(reverse=True)
    result = []
    for i in range(len(arr)):
        if target - sum(result) < arr[i]:
            continue
        else:
            result.append(arr[i])
        if sum(result) == target_sum:
            return result
    return []
```

```
target_sum = int(input())
input_arr = [int(item) for item in input().split()]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("small_first(input_arr, target_sum)",
                        setup="from __main__ import small_first,
input_arr, target_sum",
                        number=100))
    print(timeit.timeit("large_first(input_arr, target_sum)",
                        setup="from __main__ import large_first,
input_arr, target_sum",
                        number=100))

#  knapsack.py
def subset_sum(arr, target):
    subset = [[False for i in range(target + 1)] for i in
range(len(arr) + 1)]
    for i in range(len(arr) + 1):
        subset[i][0] = True

    for i in range(1, len(arr) + 1):
        for j in range(1, target + 1):
            if j < arr[i - 1]:
                subset[i][j] = subset[i - 1][j]
            else:
                subset[i][j] = subset[i - 1][j] or subset[i - 1][j
- arr[i - 1]]

    return subset


def print_set(arr, target, num, result, dp):
    if num == 0 and target != 0 and dp[1][target]:
        result.append(arr[num])
        print(sorted(result))
        return

    if target == 0:
        print(sorted(result))
```

```
        return

    if dp[num][target]:
        new_result = result.copy()
        print_set(arr, target, num - 1, new_result, dp)

    if target >= arr[num]:
        if dp[num][target - arr[num]]:
            result.append(arr[num])
            print_set(arr, target - arr[num], num - 1, result, dp)


target_sum = int(input())
input_arr = [int(item) for item in input().split()]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("subset_sum(input_arr, target_sum)",
                        setup="from __main__ import subset_sum,
input_arr, target_sum",
                        number=100))
```

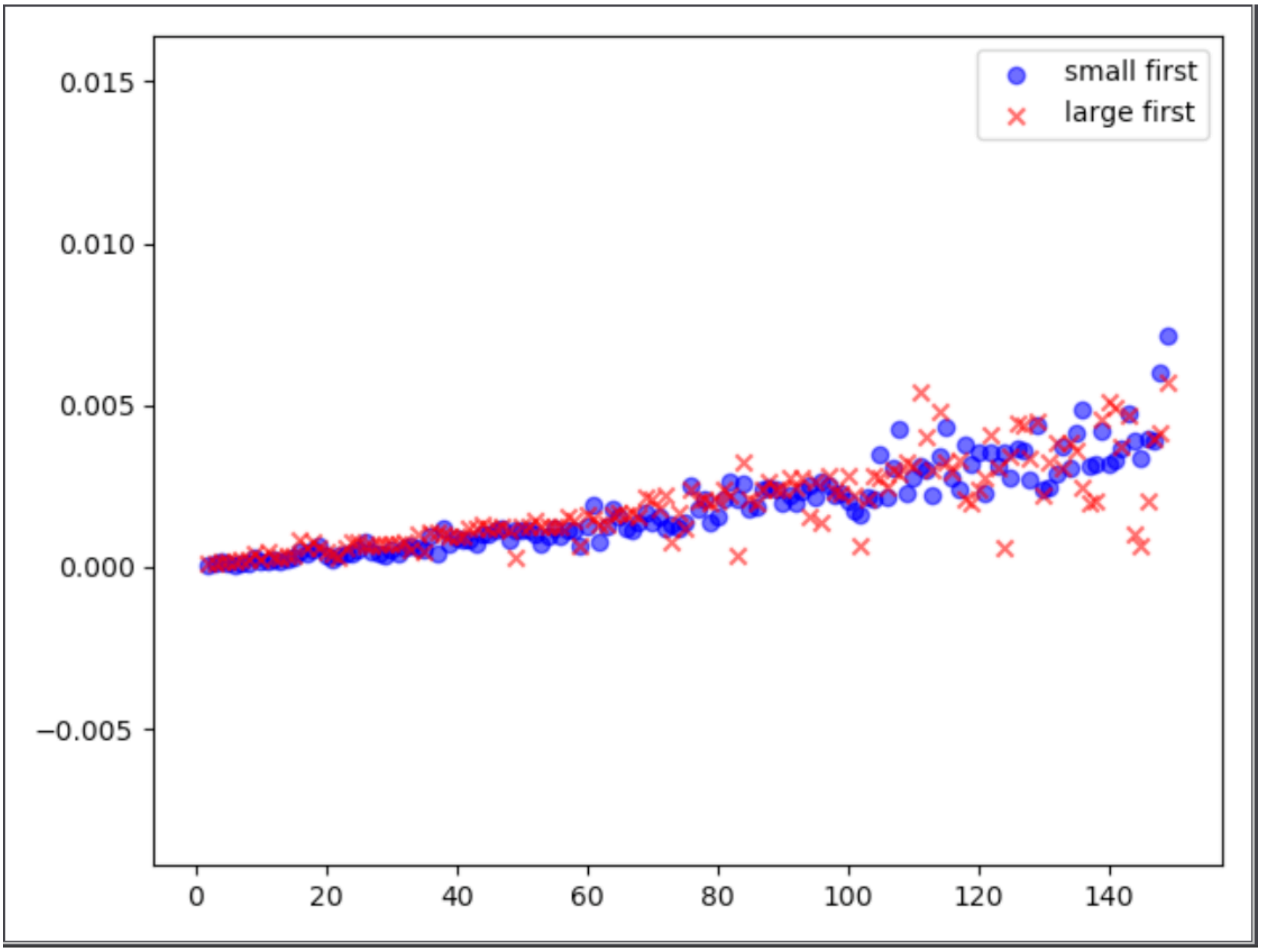
And the time is in unit *seconds*, which is the time needed to run 100 times of the input function



Time needed for the correct algorithm is much larger than the two greedy algorithms. And no much difference shown between the small first and large first algorithm. But the result generated through the greedy algorithm is not correct so the comparision result is not meaningful.

In theory, as the input array length is $2 \leq n \leq 150$ and the target number is $random(10, 20) * n$, the time complexity of the dynamic programming should be $\mathcal{O}(len(arr) * target) = \mathcal{O}(n^2)$ and meets the actual result. In greedy algorithm, because the array is sorted at first then traverse for at most one time, the time complexity is $\mathcal{O}(n \cdot logn)$.

If we only compare the two greedy algorithms, we could see that they do not differ too much. However with larger input size, put the large element first will spend less time. That might caused by the larger element will be easier to exceed the target number and the process terminated.


small first
large first
0.015
0.010
0.005
0.000
−0.005
0
20
40
60
80
100
120
140