LAB3

Wu Jiayao, 517370910257

0. Problem Description

The Knapsack problem is defined as follows. Given a set **S** and a number **n** find a subset of **S** whose elements add up exactly to **n**.

Note: I think that the definition of this problem is a little confusing that it takes me a lot of time to convince myself, since the running time is a little too large.

In my opinion, it's just the concept of subset in mathematics. I think the problem means

$$\text{find } S^{'} \subseteq S \text{ that } sum(S^{'}) = n$$

1. Two incorrect implementation

The counter example, the test case is written as followed

```
51
1 2 16 17 10 5 30 21
```

The standard output should be

```
[1, 2, 5, 10, 16, 17]
[1, 2, 10, 17, 21]
[5, 16, 30]
[21, 30]
```

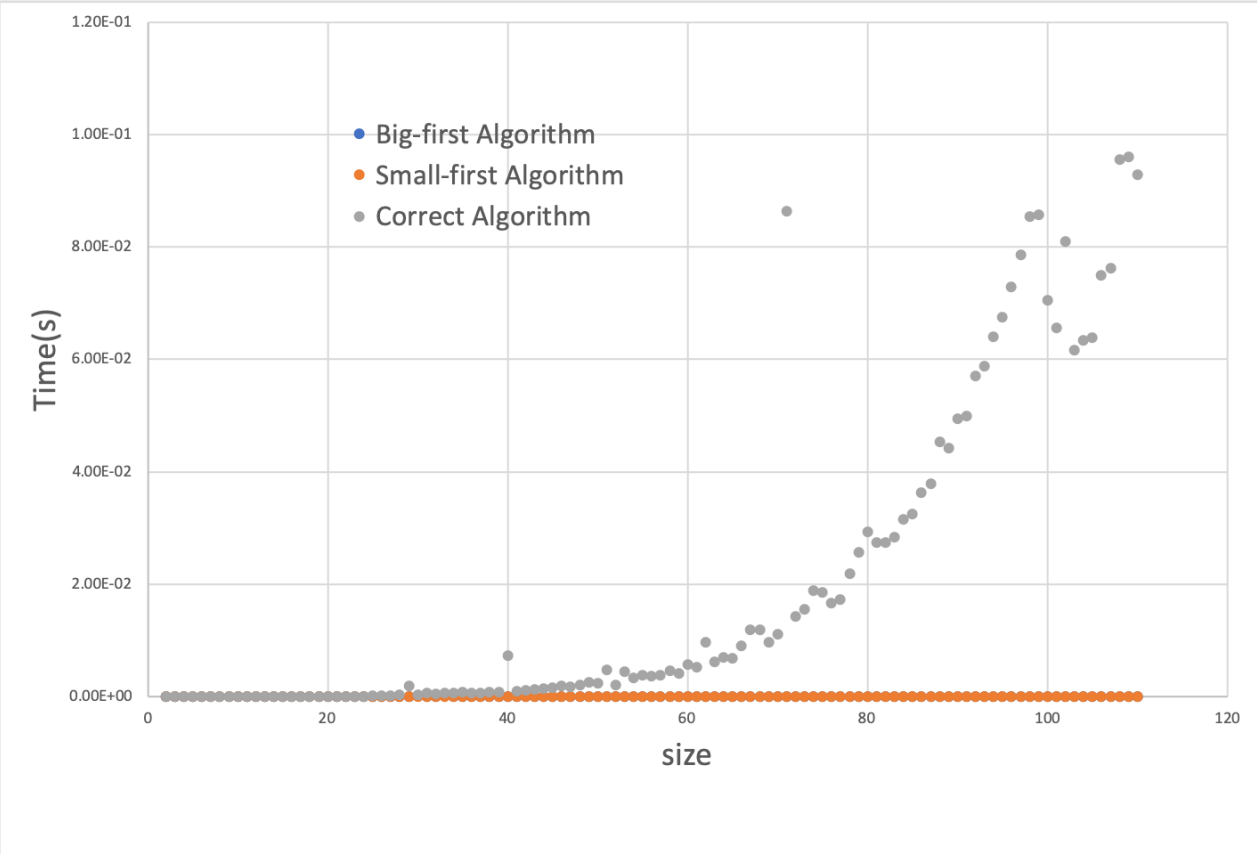
The running process is recorded with picture

```
➜  LAB3 git:(master) ✗ python3 knapsack.py < in.txt
[1, 2, 5, 10, 16, 17]
[1, 2, 10, 17, 21]
[5, 16, 30]
[21, 30]
➜  LAB3 git:(master) ✗ python3 small_start.py < in.txt
[1, 2, 5, 10, 16, 17]
➜  LAB3 git:(master) ✗ python3 big_start.py  < in.txt
[21, 30]
➜  LAB3 git:(master) ✗
```

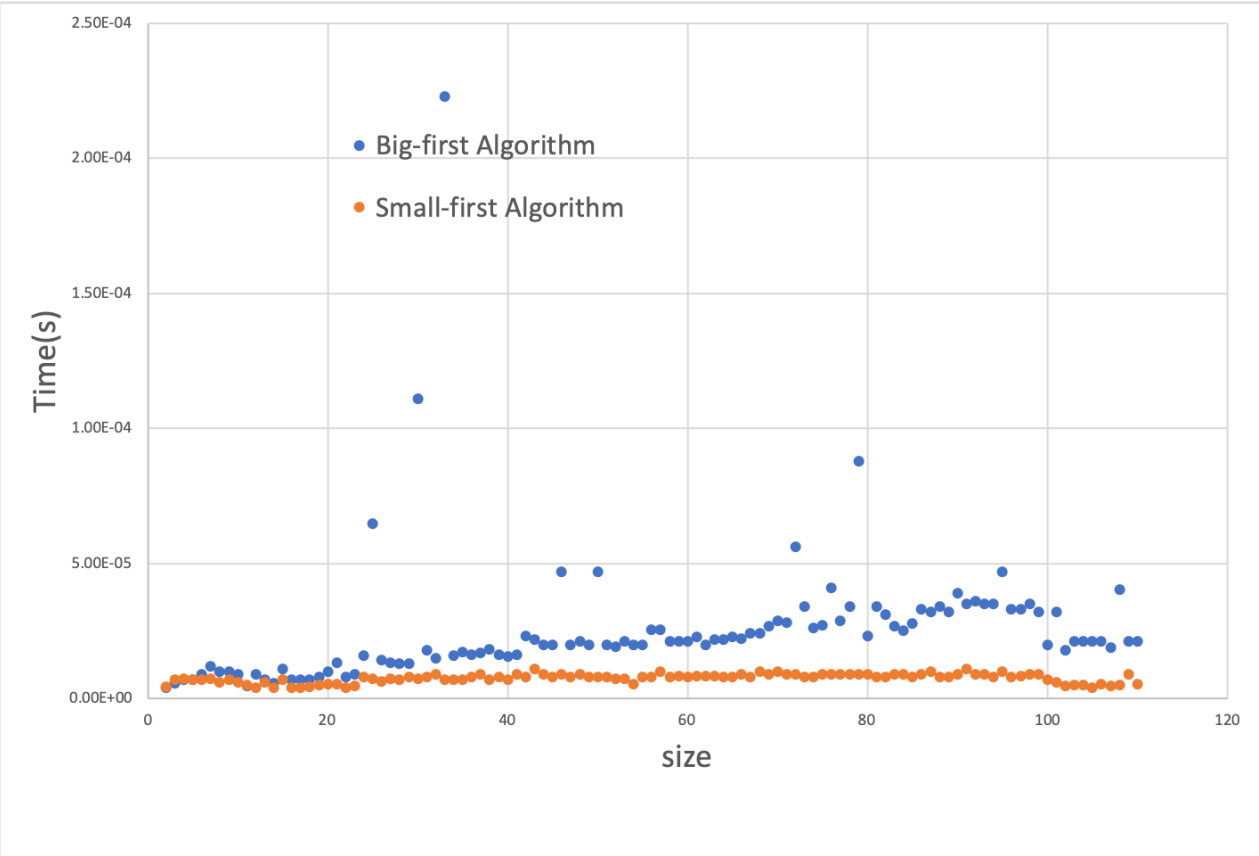
2. Running time

There are 109 test cases in all. The size of the list $nums$ ranges from 2 to 110. For each test case, the list element are a random permutation of $1, 2, \cdots, len(nums)$. To avoid the influence brought by the given sum sum, sum is set as $len(nums)/2$

The testing result is shown in the plot below:



The correct algorithm takes much more time than the two in-correct algorithms. According to table 2.1, the time complexity of the correct alogrithm is between $\mathcal{O}((len(nums))^3)$ and $\mathcal{O}(2^{len(nums)})$. For a set of n elements, the total number of its subset is 2^n. The result is kind of as expected.



The difference between small-first and big-first algorithm is not significant. In most time, the result of such two is not correct. I think that the comparison is of no use.

3. Discussion

I am not sure **by subset it means subset in mathmatics or subarray in programming.** I wonder that the big-first and small-first at least work not bad in the concept of subarray. Following is the version of subarray.

4. Two implementation [Subarray]

The test case is written as followed

```
51
1 2 16 17 10 5 30 21
```

The output should be

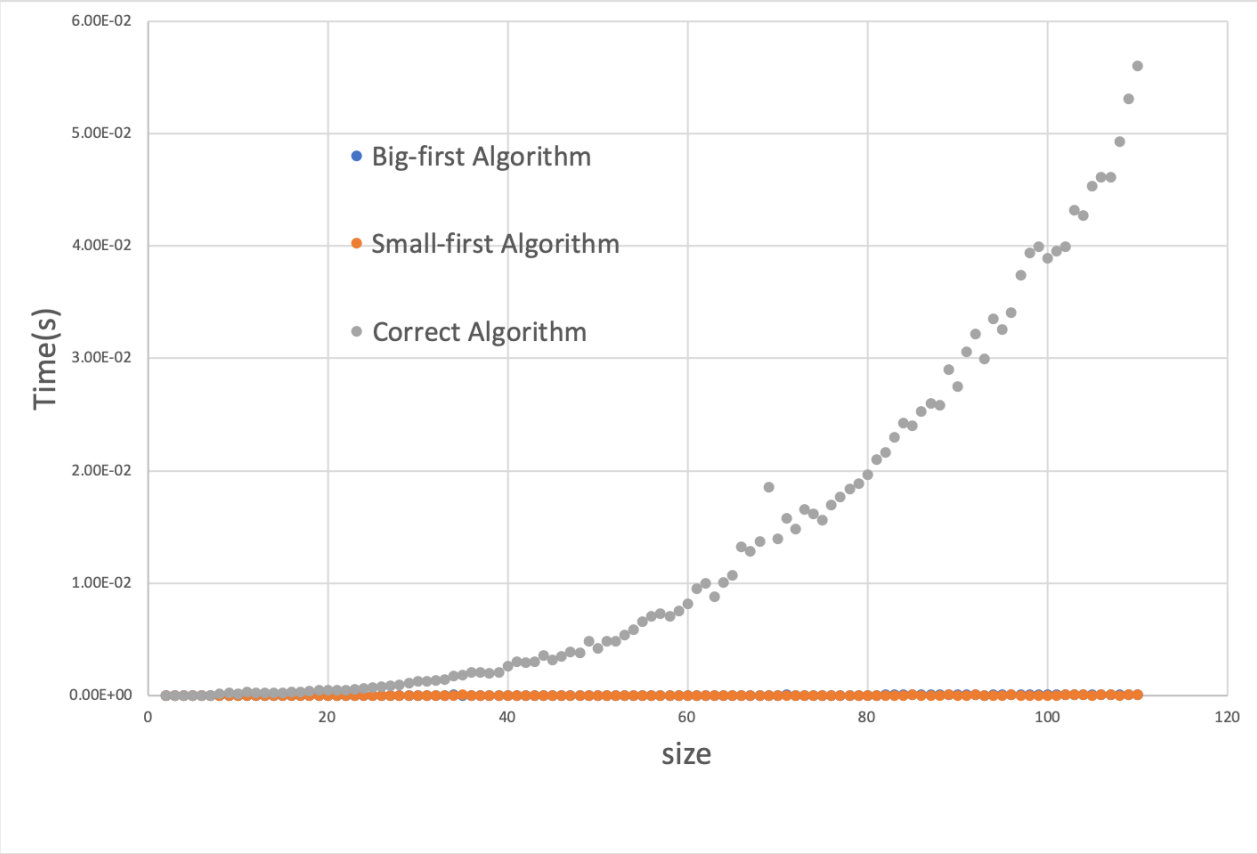
```
[1, 2, 16, 17, 10, 5]
[30, 21]
```

```
➜ LAB3 git:(master) ✗ python3 knapsack.py < in.txt
[1, 2, 16, 17, 10, 5]
[30, 21]
➜ LAB3 git:(master) ✗ python3 small_start.py < in.txt
[1, 2, 16, 17, 10, 5]
➜ LAB3 git:(master) ✗ python3 big_start.py  < in.txt
[30, 21]
➜ LAB3 git:(master) ✗
```

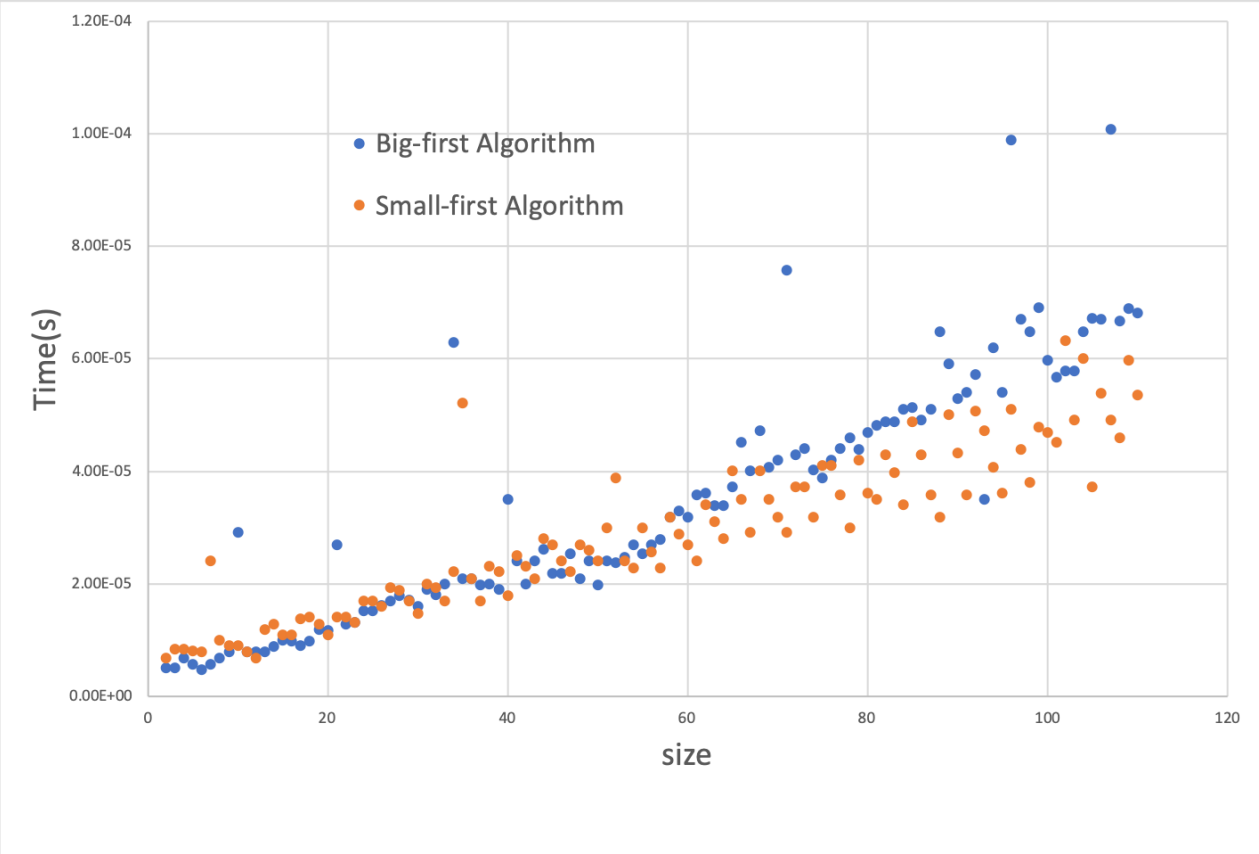
5. Running time [Subarray]

There are 109 test cases in all. The size of the list $nums$ ranges from 2 to 110. For each test case, the list element are a random permutation of $1, 2, \cdots, len(nums)$. To avoid the influence brought by the given sum sum, sum is set as $sum(nums)/2$

The testing result is shown in the plot below:



The running time is close to $\mathcal{O}(n^3)$ in the table. The time complexity of the algorithm should be $\mathcal{O}(len(nums) \times sum)$ Since $sum(num)/2 = \mathcal{O}(n^2)$, the time complexity of the correct algorithm matchers expectation.



The running time of Big-first and Small first algorithm is close to $\mathcal{O}(n \log n)$, where n is the length of sum. In serval cases, such two algorithm manage to give the correct answer. In pratical usage, such two algorithm can be used to some extent to save time cost.

Appendix

Fit the knapsack with the smallest items first

small_start.py

```
import time


def knapsack(nums, _sum):
    # nums is sorted array
    if len(nums) == 1:
        if nums[0] == _sum:
            return [nums]
        else:
            return []
    if _sum == 0:
        if 0 in nums:
            return [[0]]
        return []
    res = []
    i = 0
```

```
    if nums[i] == _sum:
        res.append([nums[i]])
    elif nums[i] > _sum:
        return []
    else:
        target = _sum - nums[i]
        tmpList = [nums[i]]
        afterRes = knapsack(nums[i+1:], target)
        for i in range(len(afterRes)):
            afterRes[i] = tmpList + afterRes[i]

        res += afterRes
    return res


def sortRes(list):
    return list[0]


if __name__ == '__main__':
    _sum = int(input().strip())
    nums = list(map(int, input().strip().split()))
    nums.sort()
    ticks = time.time()
    res = knapsack(nums[1:], _sum - nums[0])
    for i in range(len(res)):
        res[i] = [nums[0]]+res[i]
    print(time.time()-ticks)
```

Fit the knapsack with the biggest items first

big_start.py

```
import time


def knapsack(nums, _sum):
    # nums is sorted array
    if len(nums) == 1:
        if nums[0] == _sum:
            return [nums]
        else:
            return []
    if _sum == 0:
        if 0 in nums:
            return [[0]]
        return []
    res = []
```

```
    i = 0
    if nums[i] == _sum:
        res.append([nums[i]])
    elif nums[i] > _sum:
        return []
    else:
        target = _sum - nums[i]
        tmpList = [nums[i]]
        afterRes = knapsack(nums[i+1:], target)
        for i in range(len(afterRes)):
            afterRes[i] += tmpList

        res += afterRes
    return res


def sortRes(list):
    return list[0]


if __name__ == '__main__':
    _sum = int(input().strip())
    nums = list(map(int, input().strip().split()))
    nums.sort(reverse=True)

    ticks = time.time()
    res = knapsack(nums[1:], _sum - nums[0])
    for i in range(len(res)):
        res[i] += [nums[0]]
    print(time.time()-ticks)
```

Correct implementation

knapsack.py

```
def knapsack(nums, _sum):
    # nums is sorted array
    if len(nums) == 1:
        if nums[0] == _sum:
            return [nums]
        else:
            return []
    if _sum == 0:
        if 0 in nums:
            return [[0]]
        return []
```

```
    res = []
    for i in range(len(nums)):
        if nums[i] == _sum:
            res.append([nums[i]])
        elif nums[i] > _sum:
            break
        else:
            target = _sum - nums[i]
            tmpList = [nums[i]]
            afterRes = knapsack(nums[i + 1:], target)
            for i in range(len(afterRes)):
                afterRes[i] = tmpList + afterRes[i]

            res += afterRes
    return res


def sortRes(list):
    return list[0]


if __name__ == '__main__':

    _sum = int(input().strip())
    nums = list(map(int, input().strip().split()))
    nums.sort()
    res = knapsack(nums, _sum)
    if _sum == 0:
        print(res)

    if len(res) > 0:
        res.sort(key=sortRes)
        for file in res:
            print(file)
```

Generating scripts

generate.cpp

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void randperm(int Num)
{
```

```
    vector<int> temp;
    for (int i = 0; i < Num; ++i)
    {
        temp.push_back(i + 1);
    }

    random_shuffle(temp.begin(), temp.end());

    for (int i = 0; i < temp.size(); i++)
    {
        cout << temp[i] << " ";
    }
}

int main(int argc, char **argv)
{
    int num = stoi(argv[1]);
    cout << "50" << endl;
    randperm(num);
}
```

app.sh

```
#modify here
#No space between number and '='
start=2
end=110
add=1
# modify above

# How to use
# 1. Modify above as you like
# 2. Keep this this file fold named testcase, and filefold testcase should be in the same
dir with Makefile
# 3. Comment or uncomment sort/selection part (already divided by =======)
# 4. open terminal in testcase type ./app.sh in terminal and ENTER!
#Exec file should be in the same dir

rm generate
g++ -o generate generate.cpp

rm -r stdInput/

mkdir stdInput
cd stdInput

for ((i = $start; i <= $end; i = $i + $add)); do
    echo $i
    ../generate $i >"${i}"
```

```
done
cd ../

rm -r stdOutput
mkdir stdOutput
cd stdOutput

for file in $(ls ../stdInput); do
    echo "!$file"
    echo $file >>1.csv
    python3 ../kp_time.py <../stdInput/$file >>res.csv
    python3 ../small_start.py <../stdInput/$file >>resSmall.csv
    python3 ../big_start.py <../stdInput/$file >>resBig.csv
done

cd ../
```