COMP 321 – Winter 2020 Problem Presentation

Team Number: 7

Team Name: Suduko

Team Members:

Linghang Liu 260768793

Nhat Hung Le 260793376

Xu Tian 260777026

Section 0: Book-keeping

- Work is distributed equally
- All ideas are produced from the discussion in our team

Juice

Section 1: Problem Analysis

- Problem: In a party of n people, you are going to make a drink with three juices: apple, banana, carrots. To enjoy the drink, each person would require the minimum amount of apple, banana and carrots juice.

  Goal: Distribute the amount of three juices to maximize the number of people who will like it.

Section 1: Problem Analysis

Input

- One line containing an integer T, the number of test cases in the input file.

For each test case, there will be:

- One line containing the integer N, the number of people going to the party.
- N lines, one for each person, each containing three space-separated numbers "A B C", indicating the minimum fraction of each juice that would like in the drink. A, B and C are integers between 0 and $10\,000$ inclusive, indicating the fraction in parts-per-ten-thousand. $A + B + C \leq 10\,000$.

You may assume that $1 \leq T \leq 2$ and $1 \leq N \leq 5\,000$.

Output

- T lines, one for each test case in the order they occur in the input file, each containing the string "Case #X: Y" where X is the number of the test case, starting from 1, and Y is the maximum number of people who will like your drink.

Section 1: Problem Analysis

- Key: - n points, say $p_0, p_1, \ldots, p_{n-1}$

  - each point has 3 integers,say,p=(a,b,c) with a+b+c<=10000

    i.e.p0=(2500,2300,5000)

  - find 3 integers, (A,B,C) with A+B+C = 10000 such that it will maximize m, the number of points,p=(a,b,c) with a<=A, b<=B and c<=C.

Section 1: Problem Analysis

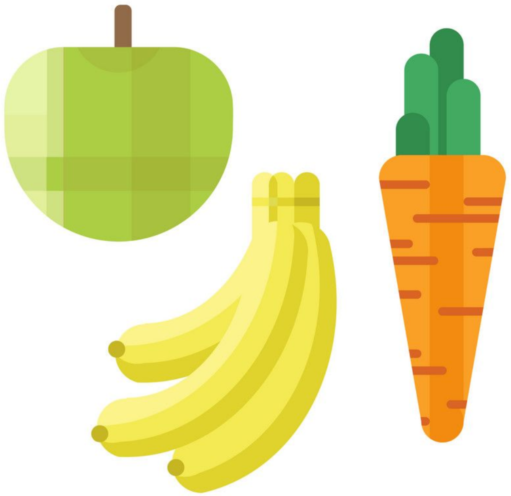
- Sample input

1.

10000	0	0
0	10000	0
0	0	10000

2.

5000	0	0
0	2000	0
0	0	4000



Section 2: Solutions

Solution 1: Brute Force

Idea: take a, b, c values from all input points as possible values A, B, C of our optimal juice.

E.g. input

a	b	c
0	1250	0
3000	0	3000
1000	1000	1000
2000	1000	2000
1000	3000	2000

-> Possible choices for

A	B	C
0	1250	0
3000	0	3000
1000	1000	1000
2000	3000	2000

Then **loop through all ABC combos**, then for each ABC combo **loop through all input points** to check for A, B, C >= a, b, c

Justification: serves as proof of correctness of approach; provides baseline for optimization e.g eliminate unnecessary, duplicated work

Solution 1: Brute Force

```python
# Driver code
lines = sys.stdin.readlines()
case_idx = i = 1

while i < len(lines):
    N = int(lines[i])
    case = lines[i+1 : i+1+N]
    case = [* map(lambda x: [* map(int, x.strip().split())], case)]
    print(f'Case #{case_idx}: {juice2(N, case)}')

    i += N + 1
    case_idx += 1
```

`N`: # points (people) in input

`case`: input as 2D array e.g. input

5000 0 0

0 2000 0

0 0 4000

```
-> case = [[5000, 0, 0],
           [0, 2000, 0],
           [0, 0, 4000]]
```

`juice2(N, case)`: function returning int solution

Solution 1: Brute Force

```python
def juice2(N, case):
    abc = [[0], [0], [0]]
    max_ppl = 0

    # Adding possible choices for a, b, c
    for i in range(3):
        for j in range(N):
            if case[j][i] not in abc[i]: # No duplicates
                abc[i].append(case[j][i])

    # Counting for all valid a, b, c combinations
    for a in abc[0]:
        for b in abc[1]:
            for c in abc[2]:
                if a + b + c > 10_000: continue

                count = 0
                for i in range(N):
                    if a >= case[i][0] and b >= case[i][1] and c >= case[i][2]:
                        count += 1

                # max_ppl = count if count > max_ppl
                max_ppl = max(count, max_ppl)
    return max_ppl
```

main loop

skip invalid ABC combo

count++ if constraint satisfied

`abc`: 2D array, possible choices for A, B, C w/o duplicates e.g.

```
case=[[0, 1250, 0],
      [3000, 0, 3000],
      [1000, 1000, 1000],
      [2000, 1000, 2000],
      [1000, 3000, 2000]]
```

-> `abc=[[0, 3000, 1000, 2000], [0, 1250, 1000, 3000], [0, 3000, 1000, 2000]]`

`count`: counts # ppl satisfied by a combination of A, B, C

`max_ppl`: maximum # ppl that can be satisfied, returned by function

Memory: only `abc` => **O(N)**

Runtime: 4 for-loops of size N => **O(N^4)**

Solution 2: (More) Optimized Brute Force

```python
def juice2(N, case):
    abc = [[0], [0]]
    max_ppl = 0

    # Adding possible choices for a, b
    # In range(2), because only considering a, b pairs
    for i in range(2):
        for j in range(N):
            if case[j][i] not in abc[i]: # No duplicates
                abc[i].append(case[j][i])

    # Counting for all valid a, b, c combinations
    for a in abc[0]:
        for b in abc[1]:
            c = 10_000 - a - b
            if c < 0: continue

            count = 0
            for i in range(N):
                if a >= case[i][0] and b >= case[i][1] and c >= case[i][2]:
                    count += 1

            # max_ppl = count if count > max_ppl
            max_ppl = max(count, max_ppl)
    return max_ppl
```

No longer stores choices for C

eliminates a for-loop

skip invalid ABC combo

`abc`: possible choices for A, B => now 2 instead of 3 rows

`c`: in main loop, `=10000-a-b`, instead of taken from `abc`
=> **reduce runtime** by removing a for loop to search for C

Memory: **O(N)**
Runtime: 1 for-loop removed => **O(N^3)**

Solution 3: Sorted Search

- Comparing all people with each (A,B,C) combination in the inner loop is too expensive.
- If we can filter people according to A,B,C respectively and stepwise, we can save some time when counting the people we can satisfy!
- Key point: check subset of people for each fixed(A,B,C)

```
case=[[0, 1250, 0],
      [3000, 0, 3000],
      [1000, 1000, 1000],
      [2000, 1000, 2000],
      [1000, 3000, 2000]]
```

```
C=[[0, 1250, 0],
   [1000, 1000, 1000],
   [2000, 1000, 2000],
   [1000, 3000, 2000],
   [3000, 0, 3000]]
```

Set c = 0,
extract `[0, 1250, 0];`
```
Set c = 1000,
extract [0, 1250, 0],
        [1000, 1000, 1000]
..................
Set c = 3000,
Extract all people
```

Solution 3: Sorted Search

```
def juice2(N, case):
        # initialize maximum number of people we can satisfy
        max_ppl = 0
        C = case
        # Sort the people by their C fraction value
        C.sort(key=(lambda x:x[2]))
        for i in range(N):
                # Let z be current C value we choose
                z = C[i][2]
                # Extract first i people satisfied with current C value
                A = C[:i+1]
                #Sort them by their A fraction value
                A.sort(key=(lambda x:x[0]))
                # A+B <= M
                M=10000-z
                # S set of people satisfied with current ABC combo
                S=[]
                for j in range(len(A)):
                        # Let x be current A value we choose
                        x = A[j][0]
                        # Let y be maximum B value under constraint M
                        y = M-x
                        # case when A+C>10000
                        if y<0: break
                        # case when A+C<=10000
                        # As we increase the A value,
                          we can add current person to the set S
                        # People in S are satisfied with fraction A and C.
                        S.append(A[j])
                        # Remove people in S who are not satisfied with B value
                        for e in S:
                                if e[1] > y:
                                        S.remove(e)
                        # Compare number of people with max number of people
                          we can satisfy so far
                        max_ppl = max(len(S),max_ppl)
        return max_ppl
```

Solution 3: Sorted Search

- Run-time: worst case: O(n*n*n)

- Memory : O(n)

- Although the runtime is the same as solution 2's, this solution uses fewer operations

Solution 4: Insertion Sort Pruned Search

Main improvements are:

- C++ -> faster than Python

- In-place sorting instead of sorting copies of arrays (which is slower)

- Insertion sort

Solution 4: Insertion Sort Pruned Search

```
int juice2(int N, int P[][3]) {
    int a, b, c, count, size, max_ppl = 0;
    vector<int> S; // List of people satisfied with current ABC combo   3

    // Sort people by C value
    insertion_sort(P, c_idx, 0, N-1);

    for (int i = 0; i < N; ++i) {
        // In sublist of people with the same C value,
        // skip to the last person
        if (i < N-1 && P[i][c_idx] == P[i+1][c_idx]) continue;   1
        // Let c = current C value
        c = P[i][c_idx];

        // Select people satisfied with current C and
        // sort them by A value
        insertion_sort(P, a_idx, 0, i);   2

        // Clear list of satisfied people
        S.clear();

        // Loop through sorted A values
        for (int j = 0; j <= i; ++j) {
            // Let a, b current A, B values
            a = P[j][a_idx];
            b = 10000-a-c;

            // Stop searching once the maximum possible B is < 0
            if (b < 0) break;
            size = S.size();
            count = size; // Assume everyone in S is currently satisfied
3           if (P[j][b_idx] <= b) {
                // Add newly satisfied people by B value
                S.push_back(P[j][b_idx]);
                count++;
            }

            // Among previously assumed satisfied people,
            // check for anyone unsatisfied
            for (int k = 0; k < size; ++k) {
                if (S[k] > b) S[k] = -1; // Set unsatisfied people to -1
                if (S[k] == -1) count--; // Decrement count
            }
            max_ppl = count > max_ppl ? count : max_ppl;
        }
    }
    return max_ppl;
}
```

1. Only search once for each unique C value => eliminate some duplicated work

2. Insertion sort is always best case when sorting by A values in the inner loop, as the the subarray is always “almost sorted”

3. `S` is int vector instead of 2D: less memory, faster operations. Holds B value of satisfied people, and sets unsatisfied people to -1 instead of removing element from array

Solution 4: Insertion Sort Pruned Search

- Run-time: O(n*n*n)

- Although the runtime is still O(n^3), we save many operations from in-place insertion sort.

- Memory : O(n)

- This is our best solution so far because this passed kattis and reduced much CPU time than before.

- This is not the best solution (the top Kattis solution is 0.07s). It passes through many elements in S more than once in the inner loop. We believe dynamic programming must be used for an optimal solution.

Section 3: Kattis Performance

Solution 1:

ID	DATE	PROBLEM	STATUS	CPU	LANG
5545145	20:35:24	Juice	✗ Time Limit Exceeded	> 12.00 s	Python 3

TEST CASES: ✓✓✓✓✓✓✓✓✗☐☐☐☐☐☐

Solution 2 & 3:

ID	DATE	PROBLEM	STATUS	CPU	LANG
5544904	18:40:17	Juice	✗ Time Limit Exceeded	> 12.00 s	Python 3

TEST CASES: ✓✓✓✓✓✓✓✓✓✓✓✗☐☐☐

Solution 4:

ID	DATE	PROBLEM	STATUS	CPU	LANG
5548426	02:09:47	Juice	✓ Accepted	7.57 s	C++

TEST CASES: ✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓