Code Jam 2015 - Qualification Round

Analysis: Infinite House of Pancakes

As the head server, you have two choices for each minute:

1. **eat**: you do nothing and let every diner with a non-empty plate eat one pancake from his or her plate.
2. **move**: you ask for the diners' attention for a *special* minute, choose a diner with a non-empty plate, and move some number of pancakes from that diner's plate to another diner's (empty or non-empty) plate.

Imagine that you, as the head server, have planned a strategy to move the pancakes. Let's call it **strategy A**. In this strategy, you pick a successive pair of minutes **t** and **t+1** such that at minute **t** you choose **eat** and at minute **t+1** you choose **move**. As a natural philosopher, you start to question your decision, "What if I swap my plan at minute **t** with my plan at minute **t+1**, i.e. **move** pancakes at minute **t** (instead of **t+1**) and **eat** at minute **t+1** (instead of **t**)?". Out of curiosity, you swap your plan at minute **t** and **t+1** and calculate the breakfast end time. Surprisingly, breakfast does not take longer. After that, you start to find several other successive pairs of minutes with the same property and do other swaps. You find that breakfast never takes longer than if you strictly follow **strategy A**. Surprisingly, on certain swaps, breakfast even ends earlier than **strategy A**!

Of course you become more curious! You start to observe what happened when you did those swaps. If the plan at minute **t** is **eat** and the plan at minute **t+1** is **move**, you notice that all pancakes that were eaten at time **t** before you did the swap were also eaten at time **t+1** after swapping. Hence, the number of pancakes that were eaten after time **t+1** does not decrease, which implies that breakfast will not take longer. After you swap the plan at **t** with **t+1**, you notice that several plates that were empty at time **t** might become non-empty at time **t+1** after swapping (and hence, the number of pancakes that were eaten after time **t+1** might increase by one).

> "Whoa! That means if I swap **<eat, move>** pairs to **<move, eat>**, the amount of pancakes that were eaten at time **t** before the swap is either less than or equal to the amount of pancakes that were eaten at time **t+1** after the swap. Hence, such swaps are always good (i.e. it will not make the breakfast time longer but might make the breakfast time shorter)!"

Excited with your finding, you grab a whiteboard and start to swap every successive pairs of **<eat, move>** to be **<move, eat>**, until none of the **<eat, move>** pairs are left. You look at your revised strategy.

> "**Move, move, move, eat, eat, eat, eat, eat**" you read out loudly. "Of course! *The only strategy without a* ***<eat, move>*** *pair is the strategy in which we do all the moves at the beginning, and always eat after that*. If we look at this carefully, swapping pairs of elements that are in the wrong order (in this case **eat** then **move**) until none are left will actually *sort* the strategy. This is basically doing bubble sort on **strategy A**!"

Knowing that you are going to get a bonus from your boss for ending breakfast early, you quickly present this solution to your boss. Your boss is not convinced. He replies without showing much interest, "Well, I understand that **moving** pancakes at the beginning will always lead to an optimal solution. But what is more important is that for every **move**, do you know how you are going to move the pancakes?" You quickly answer, "At least I know that we can always move pancakes to empty plates! There are infinitely many of them, and we do not have any

reason to let diners with non-empty plates to eat more than is needed. We cannot even wait for them to finish their existing pancakes!"

Your boss stops reading emails and starts to pay attention to your explanation. "How will you pick the plates to **move** pancakes from?" he asks. "Perhaps from the customer with the most pancakes on their plate? They need the most help to finish their pancakes," you reply.

> "You mean we can end breakfast earlier if we keep moving several pancakes from the plates with most pancakes to empty plates, and stop moving after several minutes?" he enthusiastically says. "Congratulations! You get a big bonus and extra time off!"

You are extremely happy. You gather all the kitchen staff and explain the new strategy... of course after bragging a bit on social media! A new intern staff raises her hand and asks, "When should we stop **moving** pancakes? Also, how many pancakes should we move every time?"

The whole kitchen goes silent. Everyone is looking at you expecting answers. Breakfast will start in half an hour. "Quick, think of something useful," you tell yourself. Your mind feels like it is exploding!

"Erm, honestly I don't know. Let's try to simulate breakfast to get some insights. Please bring me two plates of pancakes with **15** and **17** pancakes respectively. Ah, don't forget to bring me several empty plates too," you order one of your staff.

You start to simulate your new strategy multiple times with different amount of pancakes to be moved and stop moving at different times. Suddenly, your new intern says to you, "Look! You take **10** pancakes from the **plate 1** and put it on an **empty plate 3**. Later, you picked **3** pancakes from **plate 3** and moved it to another **empty plate 4**. Instead of doing that, why didn't you move **7** pancakes from first plate to an **empty plate 3**, and then move **3** pancakes from first plate to another **empty plate 4**? That way, you can always move pancakes from plates that are initially non-empty."

> "I see," you say, "That means we can take half the pancakes from the plate with largest amount of pancakes, then move it to an empty plate, then pick another plate with the largest amount of pancakes and do the same thing multiple times..."
>
> She replies quickly, "Your strategy is bad. Imagine that you had a plate of **9** pancakes. If we use your strategy, the best we can do is to split the plate into plates with **4** and **5** pancakes, and let the diners eat them. It costs us **6** minutes to end breakfast. But if you split the plate into three plates with **3** pancakes each (using **2 moves**), we can finish breakfast in only **5** minutes!"

"I have a suggestion," she continues, "If you expect the customer to finish their pancakes **x** minutes after you stop moving pancakes, any strategy that satisfies this will be equivalent to repeatedly moving at most **x** pancakes from every initially non-empty plate to an empty plate until the number of pancakes on the initially non-empty plate becomes no more than **x**."

"And if you always move exactly **x** pancakes, for a plate with $\mathbf{P_i}$ pancakes you need only $\mathbf{M(P_i)=ceil(P_i/x)-1}$ moves until the number of pancakes in the plate is no more than **x**. You cannot do less than that! Overall, we are going to move **sum of** $\mathbf{M(P_i)}$ times, where $\mathbf{P_i}$ is the number of pancakes on **plate i**. That is the minimum amount of moves that we can do for the given **x**! We can try all possible values of **x** to find the optimal **x** that has earliest breakfast end time!" Everyone gives her a standing ovation.

> "Let's summarize our strategy. First of all, we fix a number **x** to be the number of minutes that we expect breakfast to end in after we stop moving pancakes. After that, we pick a plate with more than **x** pancakes, take **x** pancakes from that plate and

> move the pancakes to an empty plate. We keep doing that until all plates has at most **x** pancakes, then we let the customers eat their pancakes and breakfast ends the earliest for that **x** value! If we move **sum of M(P_i)** times, in total, the breakfast ends exactly after **x + sum of M(P_i)** minutes," you say. "And we can try all possible values of **x**, since the amount of pancakes cannot be more than **1000**. The complexity of the algorithm is **O(D*M)**, where **D** is the number of diners and **M** is the maximum number of pancakes. This is fast enough to solve our problem. We can use a spreadsheet to..."

"Certainly, but I have prepared for this," says your intern. She opens her laptop and types some really short functions, of course in **C++**. "Why? Because **C++** is cool!"

```
// Get the minimum possible breakfast end time, given
// P[i] is the number of pancakes of diner i initially.
int f(const vector<int>& P) {
  const int max_pancakes = *max_element(P.begin(), P.end());
  int ret = max_pancakes;
  for (int x = 1; x < max_pancakes; ++x) {
    int total_moves = 0;
    for (const int Pi : P) {
      // (Pi - 1) / x is equivalent to M(Pi),
      // which is ceil(Pi / x) - 1
      total_moves += (Pi - 1) / x;
    }
    ret = min(ret, total_moves + x);
  }
  return ret;
}
```

"Whooa! Run it, run it!" you exclaim.

"Hold on, hold on. Although the algorithm above is fast enough to solve our problem, I have an even faster algorithm. Notice that the list of **ceil(a/1), ceil(a/2), ...** only changes values at most **2*sqrt(a)** times. For example, if **a=10**, the list is: **10, 5, 3, 3, 2, 2, 2, 2, 2, 1, 1, ...**. That list only changes value **4** times which is less than **2 * sqrt(10)**! Therefore, we can precompute when the list changes value for every diner in only **O(D*sqrt(M))**. We can keep track these value changes in a table. For example, if **P_i=10**, we can have a table **T_i**: **10, -5, -2, 0, -1, 0, 0, 0, 0, -1, 0, ...**. Notice that the prefix sum of this table is actually: **10, 5, 3, 3, 2, 2, 2, ...**. More importantly, this table is sparse, i.e. it has only **O(sqrt(M))** non-zeroes. If we do vector addition on all **T_i**, we can get a table where every entry at index **x** of the prefix sum contains sum of **ceil(P_i/x)**. Then, we can calculate sum of **ceil(P_i/x)-1** in the code above by subtracting the **x^{th}** index of the prefix sum with the number of diners. Hence, only another **O(M)** pass is needed to calculate candidate answers, which gives us **O(D*sqrt(M) + M)** running time. A much faster solution!"

> "One more thing, we cannot do binary search or ternary search, at least in trivial ways, on a function that maps **x** to its minimum breakfast end time as the function can have multiple minimas. e.g. for **2** diners with **9** pancakes each, the function forms the following mappings: **(1,17), (2,10), (3,7), (4,8), (5,7), (6,8), (7,9), (8,10), (9,9)**."

"I don't need that! The previous algorithm is fast enough. Please run it," you say impatiently.

"Sigh. I won't tell you how to code the faster solution then. The answer is..." Everyone gasps while the program blinks. "... **42**."