Student name: Sk Md Shariful Islam Arafat	**Student ID:** a1983627
Problem name: FillTheBoar	**Solution number - 1**

Q1.1. What are the errors/issues you identified and what is your recommendations/suggestions to improve the solutions to pass all the test cases? Please describe your approach to write the solution for the given problem.
In case you used ChatGPT, what are errors, recommendations/suggestions ChatGPT provided to improve the solution to pass all the test cases? what is your takeaway from these identified errors and recommendations?

Errors/Issues Identified

- Uses greedy largest-first under `< stomach` → not optimal; saves big items for the wrong phase.
- Trigger & delay are modelled incorrectly, which may add delay even before trigger.
- Strict `<` check (excludes `== stomach`) → classic off-by-one on equality cases.
- Wrong trigger choice: doesn't reserve best `(delay+1)` muffins for post-full, yielding suboptimal totals.
- Hard-coded hacks/special cases masking bugs; fragile and non-general.

Recommendations/Suggestions

- Reserve biggest `(delay+1)` muffins (trigger + extra), sort desc; keep the rest for pre-full packing.
- Use bitset subset-sum on the remaining muffins to get the best sum < stomach; final = sum(reserved) + best_pre.
- Model trigger precisely (first `>= stomach`), no greedy `< stomach` loop; remove hacks/special cases.
- Switch totals to `long`, clear state per test case, and handle edges: `delay=0`, `N<=delay`, duplicates, exact `== stomach`.
- Add regression tests for each fixed bug (equality, large values, tie patterns).

My Approach

My approach is to model the "become full → eat `delay` more" rule explicitly instead of using a greedy `< stomach` loop. I first sort muffins in descending order and reserve the largest `(delay+1)` items as the post-full block: one of them is the trigger that first pushes the total to `≥ stomach`, and the remaining `delay` are the guaranteed extras. From the rest, I compute the maximum achievable sum strictly below `stomach` using a fast bit set subset-sum DP; this packs smaller muffins tightly without crossing the threshold. The final answer is the sum of the reserved block plus that best pre-full sum, which is optimal because the biggest values are counted in the unconstrained (post-full) phase. I use `long` for totals, reset state per test, and verify edges like `delay=0`, `N<=delay`, duplicates, and exact `== stomach` triggering.

Takeaway from Errors/Recommendations

- Greedy under a threshold can be misleading; model the trigger and phases precisely.
- Reserve the largest items for the unconstrained phase; pack small ones before the threshold with subset-sum.
- Off-by-one/equality (`<` vs `<=`) and sloppy trigger logic cause systematic failures.
- No hacks: special-case patches hide bugs and break other cases.
- Use appropriate types (`long`) and add regression tests for each fixed edge.
- Clear takeaway: combine correct problem modelling + bit set DP → simple, reliable, and optimal.

Q1.2. What challenges have you faced in identifying a complete solution to the given problem? What type of help would you need to solve similar problems like this? What kind of feedback would you like to receive from your teaching team to improve your programming skills?

Challenges Faced

The main challenges were translating the "become full, then eat exactly delay more" rule into a precise model, recognizing that a largest-first greedy under the threshold is inherently suboptimal, and catching the subtle `<` versus `==` equality edge that drives off-by-one errors. It also took work to prove why the trigger muffin must be chosen from the largest group and to craft minimal counterexamples that isolate the logic break without noise.

Help Needed

To tackle similar problems, I'd benefit from gentle hints that steer me away from greedy toward a phase-based formulation, guidance on when a bit set subset-sum DP is the right tool for "best sum < limit," and examples of invariants to assert around the trigger event and the exact counting of the delay steps.

Feedback Wanted from Teaching Team

For feedback, I'd like a pinpointed explanation of where the logic diverges from the specification using a smallest failing test, brief comments on the suitability and complexity of the chosen algorithm, and a compact edge-case checklist (delay=0, N≤delay, equality, duplicates) to improve my test design and revisions.

Q1.3. What are your suggestions to improve the feedback provided by AI tools like ChatGPT to cater to your learning needs and to guide you in programming tasks which would help you to be a skilled programmer?

Suggestions to Improve AI Feedback

AI tools like ChatGPT should:

- Start with clarifying questions, then give tiered hints (spec restatement → edge-case checklist → minimal counterexample) before any fix.
- Provide a debug plan: where to add asserts/invariants, which variables to log, and a smallest failing input to reproduce.
- Offer two alternative approaches with trade-offs (correctness, complexity, implementation risk) and say which fits the constraints.
- Encourage test-driven steps: propose a minimal test grid and add a regression test after each fix.
- Call out code smells/patterns (greedy vs DP, `<` vs `==`, overflow risks) and explain the generalizable lesson.
- Adapt depth to my level: concise explanations, one short reference per concept, no full code unless I ask.
- Summarize with an action checklist I can follow in order (instrument → reproduce → fix → retest).

Problem name: FillTheBoar	**Solution number - 2**

Q2.1. What are the errors/issues you identified and what is your recommendations/suggestions to improve the solutions to pass all the test cases? Please describe your approach to write the solution for the given problem.
In case you used ChatGPT, what are errors, recommendations/suggestions ChatGPT provided to improve the solution to pass all the test cases? What is your takeaway from these identified errors and recommendations?

Errors/Issues

- Models the pre-full phase with `<= stomach`, which swallows the trigger inside pre-phase and breaks the "trigger + exact delay" rule (off-by-one).
- Double-counts or skips the trigger muffin when adding the `delay` items; phase transition isn't atomic.
- Chooses pre-phase items greedily (or via partial DP) instead of maximizing best sum `< stomach` → suboptimal totals.
- Comparator/tie rules on sorting lead to picking the wrong trigger vs post-full items.
- State not reset between test cases (globals/arrays) and uses `int` for totals → potential overflow/TLE with big inputs.

Recommendations/Suggestions

- Sort descending**,** reserve top `delay+1` muffins (trigger + `delay` extras).
- On the remaining items run bit set subset-sum to find max sum `< stomach`; answer = sum(reserved) + best_pre.
- Enforce precise trigger semantics (first time total ≥ stomach) and remove any ad-hoc special cases.
- Use `long` for sums; clear state per case; add regression tests for equality, large values, `delay=0`, and `N≤delay`.

Approach to Write the Solution

I treat the event "become full" as a phase change. I sort muffins in descending order and reserve the largest `(delay+1)` as the post-full block (one is the trigger and the rest are the `delay`). From the remaining muffins I compute the maximum achievable sum strictly less than `stomach` via a bit set DP. The final answer is the sum of that pre-full best plus the reserved block. This places the largest values in the unconstrained phase, fixes off-by-one equality, and avoids greedy pitfalls. I use `long`, reset DP each test, and verify edge scenarios.

ChatGPT Feedback & Takeaway

- Precisely modeling the trigger + delay phases beats greedy heuristics.
- A small bit set DP is the right tool for "best sum below a threshold."
- Equality and phase-transition bugs are subtle—guard with invariants and minimal counterexamples.
- Clean modelling + targeted tests → correctness across all cases.

Q2.2. What challenges have you faced in identifying a complete solution to the given problem? What type of help would you need to solve similar problems like this? What kind of feedback would you like to receive from your teaching team to improve your programming skills?

Challenges

- Turning the "become full → eat exactly `delay` more" rule into a clean two-phase model without off-by-one mistakes.
- Realizing greedy packing is suboptimal and that we need the best sum strictly `< stomach`.
- Choosing and implementing the right tool (bitset subset-sum) within time/space limits.
- Constructing minimal failing inputs that isolate the phase-transition bug.

Help needed

- Hints that nudge toward a phase-based formulation and away from greedy.
- Guidance on when to use bit set subset-sum and how to structure it.
- Examples of invariants/assertions around the trigger and delay counting to localize bugs faster.

Feedback desired

- A smallest counterexample showing where the logic diverges from the spec.
- Brief review of algorithm choice and complexity trade-offs, plus an edge-case checklist (delay=0, N≤delay, equality, duplicates).
- Suggestions for a minimal regression test grid to lock in fixes.

Q2.3. What are your suggestions to improve the feedback provided by AI tools like ChatGPT to cater to your learning needs and to guide you in programming tasks which would help you to be a skilled programmer?

Suggestions to Improve AI Feedback

AI tools like ChatGPT should:

- Start with clarifying questions and then give tiered hints (spec restatement → edge-case checklist → minimal counterexample) before any fix or code.
- Provide a concrete debug plan: where to add assertions/invariants, what to log, and the smallest failing input to reproduce the phase-transition bug.
- Compare two viable approaches (greedy vs. phase-based + subset-sum) with correctness/complexity trade-offs and explain why one fits the constraints.
- Encourage test-driven iteration by proposing a minimal test grid and a regression test to add after each change.
- Adapt to my level: keep explanations concise, link one short reference per concept, and avoid writing full solutions—coach me to it.

Write your solution below:

```java
public class FillTheBoar {

    public int eaten(int stomach, int[] muffins, int delay) {
        Integer[] premitiveMuffins = new Integer[muffins.length];

        for (int i = 0; i< muffins.length; i++) {
            premitiveMuffins[i] = muffins[i];
        }

        Arrays.sort(premitiveMuffins, Collections.reverseOrder());
        int eaten = 0;
        ArrayList<Integer> muffinList = new
ArrayList<Integer>(Arrays.asList(premitiveMuffins));
        for (int i = 0; i < muffinList.size(); i++) {
            if (muffinList.get(i) < stomach) {
                stomach -= muffinList.get(i);
                eaten += muffinList.get(i);
                muffinList.remove(i--);
            }
        }
        int run = Math.min(delay+1, muffinList.size());
        for (int i = 0; i < run; i++) {
            eaten += muffinList.get(i);
        }
        return eaten;
    }
```