<https://leetcode.com/problems/2-keys-keyboard/description/>

There is only one character 'A' on the screen of a notepad. You can perform one of two operations on this notepad for each step:

Copy All: You can copy all the characters present on the screen (a partial copy is not allowed).

Paste: You can paste the characters which are copied last time.

Given an integer n, return the minimum number of operations to get the character 'A' exactly n times on the screen.

**Example 1:**

**Input:** n = 3

**Output:** 3

**Explanation:** Initially, we have one character 'A'.

In step 1, we use Copy All operation.

In step 2, we use Paste operation to get 'AA'.

In step 3, we use Paste operation to get 'AAA'.

**Example 2:**

**Input:** n = 1

**Output:** 0

**Constraints:**

1 <= n <= 1000

**Attempt 1: 2025-02-12**

**Solution 1: Native DFS (30 min)**

class Solution {

    public int minSteps(int n) {

        return helper(n, 1, 0);

    }

    private int helper(int n, int onScreen, int onClickBoard) {

        // Base case: already have n 'A's

        if(onScreen == n) {

            return 0;

        }

        // Invalid case

        if(onScreen > n) {

            return Integer.MAX\_VALUE;

        }

        int minSteps = Integer.MAX\_VALUE;

        // Option 1: Paste (if clipboard is not empty)

        if(onClickBoard > 0) {

            int steps = helper(n, onScreen + onClickBoard, onClickBoard);

            if(steps != Integer.MAX\_VALUE) {

                minSteps = Math.min(minSteps, 1 + steps);

            }

        }

        // Option 2: Copy All (only if current != clipboard to avoid redundant copies)

        if(onScreen != onClickBoard) {

            int steps = helper(n, onScreen, onScreen);

            if(steps != Integer.MAX\_VALUE) {

                minSteps = Math.min(minSteps, 1 + steps);

            }

        }

        return minSteps;

    }

}

Time Complexity: O(2^n) The minStepsHelper function is recursively called 2 times at each point.

The maximum height of the call stack would be n, leading to a total exponential time complexity of O(2^n).

Space Complexity: O(n) The space complexity is determined by the call stack, which has a maximum height of O(n).

**Solution 2: DFS + Memoization (10min)**

class Solution {

    public int minSteps(int n) {

        Integer[][] memo = new Integer[n + 1][n + 1];

        return helper(n, 1, 0, memo);

    }

    private int helper(int n, int onScreen, int onClickBoard, Integer[][] memo) {

        // Base case: already have n 'A's

        if(onScreen == n) {

            return 0;

        }

        // Invalid case

        if(onScreen > n) {

            return Integer.MAX\_VALUE;

        }

        if(memo[onScreen][onClickBoard] != null) {

            return memo[onScreen][onClickBoard];

        }

        int minSteps = Integer.MAX\_VALUE;

        // Option 1: Paste (if clipboard is not empty)

        if(onClickBoard > 0) {

            int steps = helper(n, onScreen + onClickBoard, onClickBoard, memo);

            if(steps != Integer.MAX\_VALUE) {

                minSteps = Math.min(minSteps, 1 + steps);

            }

        }

        // Option 2: Copy All (only if current != clipboard to avoid redundant copies)

        if(onScreen != onClickBoard) {

            int steps = helper(n, onScreen, onScreen, memo);

            if(steps != Integer.MAX\_VALUE) {

                minSteps = Math.min(minSteps, 1 + steps);

            }

        }

        return memo[onScreen][onClickBoard] = minSteps;

    }

}

Time Complexity: O(n^2) The time complexity is determined by the total number of subproblems solved,

which is proportional to the size of the memo array: (n+1)⋅(n/2+1). This leads to a time complexity of O(n^2).

Space Complexity: O(n^2) The space complexity is determined by the size of the memo array, which is O(n^2).

**Solution 3: 2D DP (30min, based on Solution 1)**

class Solution {

    public int minSteps(int n) {

        int[][] dp = new int[n + 1][n + 1];

        // Initialize all states to a large value (invalid/unreachable)

        for(int i = 0; i <= n; i++) {

            for(int j = 0; j <= n; j++) {

                dp[i][j] = Integer.MAX\_VALUE;

            }

        }

        // Base case: when onScreen == n, no steps needed

        for(int j = n; j >= 0; j--) {

            dp[n][j] = 0;

        }

        // Iterate from n-1 down to 1 (bottom-up)

        for(int onScreen = n - 1; onScreen >= 1; onScreen--) {

            // Process clipboard values in reverse order (critical fix)

            for(int onClickBoard = n; onClickBoard >= 0; onClickBoard--) {

                // Option 1: Paste (if clipboard is not empty)

                if(onClickBoard > 0 && onClickBoard + onScreen <= n) {

                    if(dp[onClickBoard + onScreen][onClickBoard] != Integer.MAX\_VALUE) {

                        dp[onScreen][onClickBoard] = Math.min(dp[onScreen][onClickBoard],

                            1 + dp[onClickBoard + onScreen][onClickBoard]);

                    }

                }

                // Option 2: Copy All (only if clipboard != current)

                if(onScreen != onClickBoard) {

                    if(dp[onScreen][onScreen] != Integer.MAX\_VALUE) {

                        dp[onScreen][onClickBoard] = Math.min(dp[onScreen][onClickBoard],

                            1 + dp[onScreen][onScreen]);

                    }

                }

            }

        }

        // Answer: start with 1 'A' on screen and 0 in clipboard

        return dp[1][0];

    }

}

Time Complexity: O(n^2) – We iterate over all possible states (onScreen, onClickBoard), where each ranges from 0 to n.

Space Complexity: O(n^2) – The DP table stores results for all states.

**Solution 4: Greedy (60min)**

class Solution {

    public int minSteps(int n) {

        // dp[i] represents the minimum number of operations to reach i A's

        int[] dp = new int[n + 1];

        Arrays.fill(dp, 1000);

        // Base case, no operation required since initially already have 1 A

        dp[1] = 0;

        for(int i = 2; i <= n; i++) {

            // j <= i / 2 since i / 2 is the largest factor of i

            for(int j = 1; j <= i / 2; j++) {

                // Copy All (1 operation) and Paste (i - j) / j

                // times ((i - j) / j operation) for all valid j's

                // total 1 + (i - j) / j =  i / j operation needed

                if(i % j == 0) {

                    dp[i] = Math.min(dp[i], dp[j] + i / j);

                }

            }

        }

        return dp[n];

    }

}

Time Complexity: O(n^2) Initializing our dp array takes O(n) time.

To fill in the dp array, the outer and inner loop each run O(n) times, resulting in a total time complexity of O(n^2).

Space Complexity: O(n) The space complexity is determined by our dp array, which has a size of O(n).

**Refer to**

<https://leetcode.com/problems/2-keys-keyboard/editorial/>

**Approach 1: Recursion / Backtracking**

**Intuition**

When adding A's on the screen to achieve n A's, we note that it is unnecessary to apply consecutive Copy All operations because applying consecutive Copy All operations has the same effect as applying just one. If a Copy All operation is applied, then a Paste operation should be applied right after. Thus, we have two options to add A's on the screen at every step:

Apply a Copy All operation first and then apply the Paste operation right after.

Apply a Paste operation.

A brute-force approach involves exploring both ways recursively at each step. This would allow us to find all possible sequences of operations that result in exactly n A's, and then choose the sequence that requires the minimum number of operations.

To implement this, we define a function *f*(*i*,*j*), which represents the minimum number of operations needed to get to n A's starting with *i* A's, where the previous copy operation had *j* A's.

We can break the problem into subproblems based on the two options described above:

Copy All + Paste: This option takes 2 operations. It doubles the number of A's to i \* 2, and updates the previous copy length to i. Thus, the number of operations needed for this choice is 2+*f*(*i*∗2,*i*).

Paste: This option takes 1 Paste operation. It increases the number of A's by j while keeping the previous copy length as j. Thus, the number of operations needed for this choice is 1+*f*(*i*+*j*,*j*).

By making recursive calls for these two choices — 2+*f*(*i*∗2,*i*) and 1+*f*(*i*+*j*,*j*) — our solution can return the minimum value among these options, effectively finding the global minimum number of operations needed to reach n A's.

**Algorithm**

If n == 1, no operations are needed so return 0.

Define a recursive helper function minStepsHelper(int currLen, int pasteLen):

Base Case: If currLen == n, then we have reached n A's, so return 0

Base Case: If currLen > n, then we have exceeded the number of A's needed, so return max value 1000, ignoring this current sequence

Try Copy All + Paste: Initialize opt1 to 2 + minStepsHelper(currLen \* 2, currLen), where 2 operations are used, currLen is doubled, and pasteLen is updated to currlen

Try Paste: Initialize opt2 to 1 + minStepsHelper(currLen + pasteLen, pasteLen), where 1 operation is used, currLen increases by pasteLen and pasteLen remains the same.

Return the minimum between opt1 and opt2

Return 1 + minStepsHelper(1, 1), the minimum number of operations to get to n A's from 1 A, where pasteLen is 1 from performing a Copy All operation first.

**Implementation**

class Solution {

int n;

public int minSteps(int n) {

if (n == 1) return 0;

this.n = n;

// first step is always a Copy All operation

return 1 + minStepsHelper(1, 1);

}

private int minStepsHelper(int currLen, int pasteLen) {

// base case: reached n A's, don't need more operations

if (currLen == n) return 0;

// base case: exceeded n `A`s, not a valid sequence, so

// return max value

if (currLen > n) return 1000;

// copy all + paste

int opt1 = 2 + minStepsHelper(currLen \* 2, currLen);

// paste

int opt2 = 1 + minStepsHelper(currLen + pasteLen, pasteLen);

return Math.min(opt1, opt2);

}

}

**Complexity Analysis**

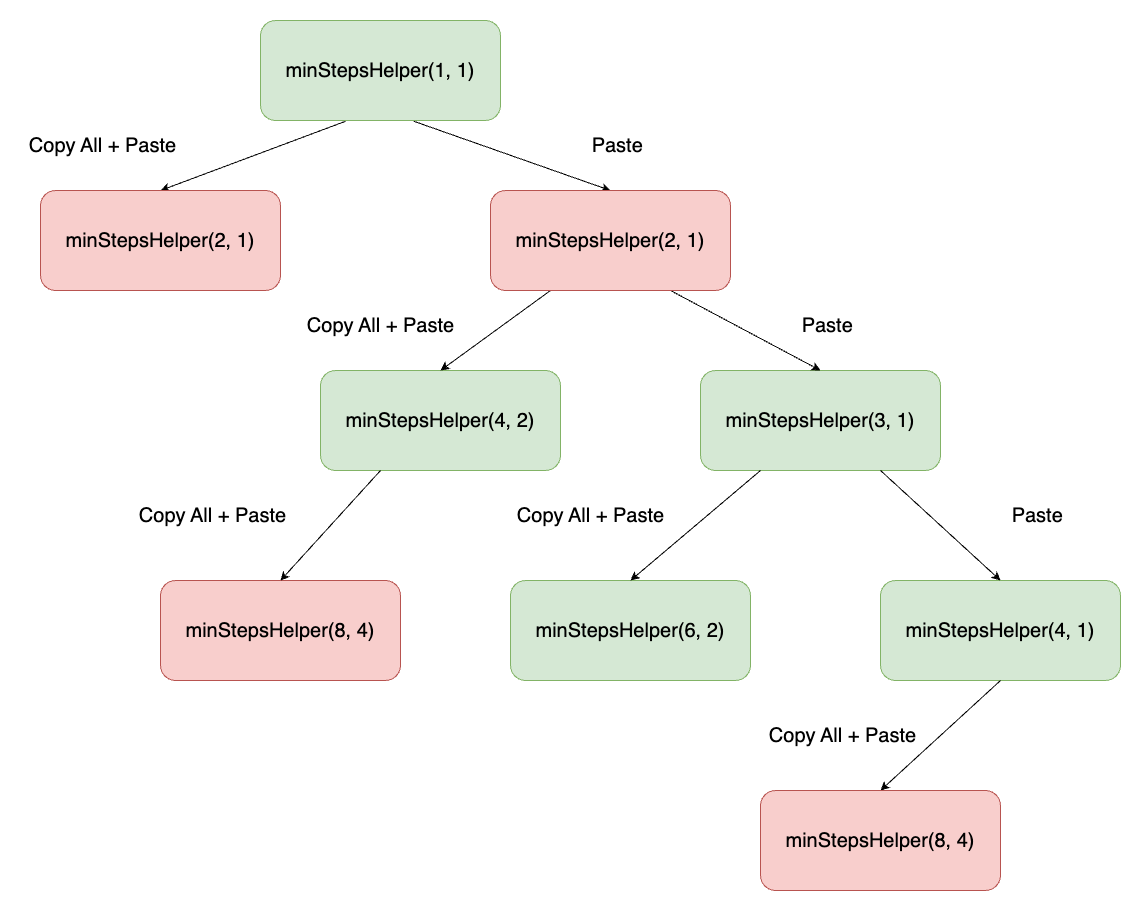
Time Complexity: *O*(2^*n*) The minStepsHelper function is recursively called 2 times at each point. The maximum height of the call stack would be *n*, leading to a total exponential time complexity of *O*(2^*n*).

Space Complexity: *O*(*n*) The space complexity is determined by the call stack, which has a maximum height of *O*(*n*).

**Approach 2: Top-Down Dynamic Programming**

**Intuition**

In Approach 1, certain subproblems *f*(*i*,*j*) can appear more than once, resulting in duplicate calculations. This issue is illustrated by the recursive call tree for minStepsHelper, where duplicate calls are highlighted in red.



To optimize this, we can utilize a technique called [memoization](https://leetcode.com/explore/learn/card/recursion-i/255/recursion-memoization/1495/), which stores previously computed results in a cache. With memoization, we can check if an answer to a subproblem has already been computed, and retrieve the answer from our cache to avoid redundant calculations.

Our cache can be a 2D array memo, where memo[i][j] stores the answer to subproblem *f*(*i*,*j*). The dimensions of memo can be (*n*+1)×(2*n*​+1), because the current number of characters is at most n and the previous copy length is at most 2*n*​.

By employing memoization, we eliminate duplicate work and solve each unique subproblem exactly once, improving the efficiency of our solution.

**Algorithm**

If n == 1, no operations are needed so return 0.

Initialize cache memo[i][j] to 2D array with dimensions (n + 1) x (n / 2 + 1).

Define a recursive helper function minStepsHelper(int currLen, int pasteLen, int[][] memo):

Base Case: If currLen == n, then we have reached n A's, so return 0

Base Case: If currLen > n, then we have exceeded the number of A's needed, so return max value 1000, ignoring this current sequence

Check cache: If memo has the answer to the subproblem, return memo[currLen][pasteLen].

Solve subproblem:

Try Copy All + Paste: Initialize opt1 to 2 + minStepsHelper(currLen \* 2, currLen), where 2 operations are used, currLen is doubled, and pasteLen is updated to currlen

Try Paste: Initialize opt2 to 1 + minStepsHelper(currLen + pasteLen, pasteLen), where 1 operation is used, currLen increases by pasteLen and pasteLen remains the same.

Save the minimum between opt1 and opt2 in memo[currLen][pasteLen] and return it.

Return 1 + minStepsHelper(1, 1, memo), the minimum number of operations to get to n A's from 1 A, where pasteLen is 1 due to performing a Copy All operation first.

**Implementation**

class Solution {

int n;

public int minSteps(int n) {

if (n == 1) return 0;

this.n = n;

int[][] memo = new int[n + 1][n / 2 + 1];

return 1 + minStepsHelper(1, 1, memo);

}

private int minStepsHelper(int currLen, int pasteLen, int[][] memo) {

if (currLen == n) return 0;

if (currLen > n) return 1000;

// return result if it has been calculated already

if (memo[currLen][pasteLen] != 0) return memo[currLen][pasteLen];

int opt1 = 1 + minStepsHelper(currLen + pasteLen, pasteLen, memo);

int opt2 = 2 + minStepsHelper(currLen \* 2, currLen, memo);

memo[currLen][pasteLen] = Math.min(opt1, opt2);

return memo[currLen][pasteLen];

}

}

**Complexity Analysis**

Time Complexity: *O*(*n^*2) The time complexity is determined by the total number of subproblems solved, which is proportional to the size of the memo array: (*n*+1)⋅(*n*/2+1). This leads to a time complexity of *O*(*n^*2).

Space Complexity: *O*(*n^*2) The space complexity is determined by the size of the memo array, which is *O*(*n^*2).

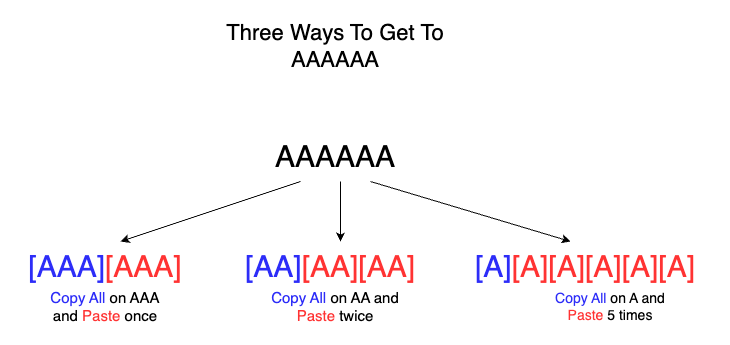
**Approach 3: Bottom-Up Dynamic Programming**

**Intuition**

An alternate approach is to solve our subproblems from bottom to top (bottom-up dynamic programming), by working from the base case up to the final answer. We define a new function *f*(*i*) to represent the minimum number of operations to get to *i* A's starting from 1 A. Note that in contrast to Approaches 1 and 2, we do not keep track of the length of the previous copy. This approach focuses on incrementally building up from the base case *f*(1)=0 to *f*(*n*), the final result.

To do this, we'd like to form a relation between subproblems and express *f*(*i*) in terms of *f*(*j*) for values of *j* where 1≤*j*<*i*.

For a given subproblem *f*(*i*) where there are currently i A's, we recognize the last operation must have been a paste. Furthermore, we know that the number of A's previously copied must be a factor of *i*. For example, if we currently have 6 A's, the previous copy could have been of 1, 2, or 3 A's, which are all the factors of 6.



Thus, one possible way to make *i* A's is to use the Copy All operation on *j* A's, where *j* is a factor of *i*. We can then paste the *j* A's (*i*−*j*)/*j* times to reach a total of i A's. If this approach is chosen, then the minimum number of operations possible would be *f*(*j*)+1+(*i*−*j*)/*j*. **Here, *f*(*j*) represents the minimum number of operations to reach *j* A's, 1 accounts for the single Copy All operation on the *j* A's, and (*i*−*j*)/*j* represents the number of additional Paste operations of *j* A's needed.**

We can simplify the expression *f*(*j*)+1+(*i*−*j*)/*j* to *f*(*j*)+*i*/*j*.

If we consider all possible factors *j* of i, then we can solve for *f*(*i*). Thus, we have the relation:

*f*(*i*)=*f*(*j*)+*i*/*j* for all *j* such that *i*mod*j*==0. **Note that *j*≤*i*/2 since *i*/2 is the largest factor of *i*.**

By iteratively applying this relation, we can build up to compute *f*(*n*), effectively solving the problem from the bottom up.

**Algorithm**

Initialize an array dp of size n+1 where *dp*[*i*]=*f*(*i*), 1<=*i*<=*n*

Initialize values of dp to a default max value of 1000

Fill in the base case: dp[1] = 0

Iterate through values of i from 2 to n:

Iterate through values of j from 1 to i/2:

If i % j == 0: Set dp[i] to minimum between dp[i] and dp[j] + i / j.

**Implementation**

class Solution {

public int minSteps(int n) {

int[] dp = new int[n + 1];

Arrays.fill(dp, 1000);

// Base case

dp[1] = 0;

for (int i = 2; i <= n; i++) {

for (int j = 1; j <= i / 2; j++) {

// Copy All and Paste (i-j) / j times

// for all valid j's

if (i % j == 0) {

dp[i] = Math.min(dp[i], dp[j] + i / j);

}

}

}

return dp[n];

}

}

**Complexity Analysis**

Time Complexity: *O*(*n^*2) Initializing our dp array takes *O*(*n*) time. To fill in the dp array, the outer and inner loop each run *O*(*n*) times, resulting in a total time complexity of *O*(*n^*2).

Space Complexity: *O*(*n*) The space complexity is determined by our dp array, which has a size of *O*(*n*).

**Approach 4: Prime Factorization**

**Intuition**

Note: This approach contains some mathematical notation. We encourage you to read carefully to fully understand the intuition.

In Approach 3, we recognize that getting to *i* A's will repeatedly involve a Copy All operation followed by a series of Paste operations. For example, a possible sequence of operations might look like [CPP][CPPPP][CP]. In this approach, we will find a way to minimize the length of each block of this sequence. In doing so, we can find the minimum number of operations needed to achieve n A's at the end.

To start, we call the length of the *i*−*th* block in this sequence *gi*​. From this setup, We can make two important observations:

The total number of operations performed can be expressed as *g*1​+*g*2​+...+*gn*​.

After applying *g*1​ operations, we have *g*1​ A's. Then, after applying *g*2​ operations, we have *g*1​×*g*2​ A's. In general, *g*1​×*g*2​×...×*gn*​=*n*.

Thus, to solve the problem, we need to find values for *g*1​,*g*2​,...,*gn*​ so that their sum is minimized while ensuring that their product is equal to n.

Let's dive deep on how a certain block's length can be minimized. When examining a block *i* where its length *gi*​ is composite, (i.e. *gi*​=*p*×*q*), we can break it down into two smaller blocks of size *p* and *q*. For example, if our first block is [*CPPPPP*], where *gi*​=3×2, we can break that down into [*CPP*][*CP*]. This splitting reduces the total number of operations in this example from 6 to 5, while still producing the same number of A's as the original block.

Because using *p*+*q* moves by splitting is never more than using *p*×*q* moves by not splitting, the optimal strategy involves breaking down each composite *gi*​ into its prime factors. Thus, splitting whenever possible will lead to the minimum number of operations.

This will lead to each *gi*​ being a prime factor of n. This problem then reduces to finding the sum of the prime factors of n.

**Algorithm**

Initialize ans to 0, representing the current sum of prime factors

Initialize d to 2, the first possible prime factor to consider.

While n is not equal to 0:

While d is a prime factor: While n % d == 0:

Divide**n by the prime factor:** n = n / d

Add**d to current sum ans:**ans += d

Increment**d to find the next prime factor:** d++

Return ans

**Implementation**

class Solution {

public int minSteps(int n) {

int ans = 0;

int d = 2;

while (n > 1) {

// If d is prime factor, keep dividing

// n by d until is no longer divisible

while (n % d == 0) {

ans += d;

n /= d;

}

d++;

}

return ans;

}

}

**Complexity Analysis**

Time Complexity: *O*(*n*​) The outer while loop runs until n becomes 1. The inner while loop divides n by d whenever d is a divisor of n. The factorization of n involves checking divisibility from d = 2 to *d*≤*n*​. After d surpasses *sqrtn*, n can only have one prime factor greater than *n*​, which will be handled in one iteration of the outer loop. Thus, the complexity is dominated by the number of potential divisors up to *n*​, leading to a time complexity of *O*(*n*​).

Space Complexity: *O*(1) Our iterative algorithm has no recursive overhead and no auxiliary data structures. Thus, the space complexity is *O*(1).

**Refer to Deepseek**

The LeetCode problem 650. 2 Keys Keyboard asks for the minimum number of operations to get exactly n 'A's on a notepad using only two operations:

Copy All: Copy all the 'A's currently on the screen.

Paste: Paste the last copied 'A's.

This problem can be solved using dynamic programming (DP) or a mathematical/greedy approach. Below, I'll provide three Java solutions: DFS with Memoization, Dynamic Programming (DP), and a Greedy Solution.

**1. DFS with Memoization**

This approach explores all possible ways to reach n 'A's using recursion and memoization to avoid redundant calculations.

class Solution {

public int minSteps(int n) {

return dfs(n, 1, 0, new HashMap<>());

}

private int dfs(int n, int current, int clipboard, Map<String, Integer> memo) {

if (current == n) return 0; // Base case: already have n 'A's

if (current > n) return Integer.MAX\_VALUE; // Invalid case

String key = current + "," + clipboard;

if (memo.containsKey(key)) return memo.get(key);

int minSteps = Integer.MAX\_VALUE;

// Option 1: Paste (if clipboard is not empty)

if (clipboard > 0) {

int steps = dfs(n, current + clipboard, clipboard, memo);

if (steps != Integer.MAX\_VALUE) {

minSteps = Math.min(minSteps, 1 + steps);

}

}

// Option 2: Copy All (only if current != clipboard to avoid redundant copies)

if (current != clipboard) {

int steps = dfs(n, current, current, memo);

if (steps != Integer.MAX\_VALUE) {

minSteps = Math.min(minSteps, 1 + steps);

}

}

memo.put(key, minSteps);

return minSteps;

}

}

**2. Dynamic Programming (DP)**

This approach uses a DP table to store the minimum steps required to reach each number of 'A's.

class Solution {

public int minSteps(int n) {

int[] dp = new int[n + 1];

for (int i = 2; i <= n; i++) {

dp[i] = i; // Default to copying 'A' and pasting (i-1) times

for (int j = i / 2; j >= 1; j--) {

if (i % j == 0) {

dp[i] = dp[j] + (i / j); // Copy once and paste (i/j - 1) times

break;

}

}

}

return dp[n];

}

}

**3. Greedy Solution**

This approach leverages the mathematical insight that the minimum steps to reach n 'A's is the sum of the prime factors of n.

class Solution {

public int minSteps(int n) {

int steps = 0;

for (int factor = 2; factor <= n; factor++) {

while (n % factor == 0) {

steps += factor;

n /= factor;

}

}

return steps;

}

}

**Explanation of the Greedy Solution:**

Prime Factorization:

The problem reduces to finding the prime factors of n.

For example, if n = 6, its prime factors are 2 and 3. The minimum steps are 2 + 3 = 5.

Why It Works:

Each prime factor represents a sequence of operations: Copy All followed by Paste multiple times.

The sum of the prime factors gives the total number of operations.

**Complexity Analysis:**

DFS with Memoization:

Time: (O(n^2)) (due to recursion and memoization).

Space: (O(n^2)) (for the memoization map).

Dynamic Programming:

Time: (O(n sqrt{n})) (for each i, we check up to ( sqrt{i} ) factors).

Space: (O(n)) (for the DP table).

Greedy Solution:

Time: (O(sqrt{n})) (for prime factorization).

Space: (O(1)) (constant space).

**Which Solution to Use?**

Use the Greedy Solution for the best time and space complexity.

Use DFS with Memoization or DP if you want to explore all possible states explicitly.

**In DFS solution, i see below condition in dfs method, why we need this check, example ?**

// Option 2: Copy All (only if current != clipboard to avoid redundant copies)

if (current != clipboard) {...}

The condition if (current != clipboard) in the DFS solution is used to avoid redundant copy operations. Let me explain why this check is necessary and provide an example to illustrate its importance.

**Why the Check is Needed:**

Redundant Copy Operations:

If the current number of 'A's (current) is already equal to the number of 'A's in the clipboard (clipboard), performing a Copy All operation would be redundant.

Copying the same number of 'A's again doesn't change the state of the clipboard, so it doesn't help in reducing the number of steps.

Avoiding Infinite Loops:

Without this check, the DFS could repeatedly perform Copy All operations without making progress, leading to infinite recursion or unnecessary computations.

**Example to Illustrate:**

Let’s say n = 3, and we are at a state where:

current = 2 (we have 2 'A's on the screen).

clipboard = 2 (the clipboard also has 2 'A's).

**Without the Check:**

If we perform a Copy All operation:

The clipboard is already 2, so copying again doesn't change anything.

The state remains the same: current = 2, clipboard = 2.

This leads to an infinite loop or redundant computations, as the DFS keeps trying to copy the same number of 'A's.

**With the Check:**

The condition if (current != clipboard) ensures that we only perform a Copy All operation if the clipboard doesn't already contain the same number of 'A's as the current screen.

In this case, since current == clipboard, the Copy All operation is skipped, and the DFS explores other valid options (e.g., Paste).

**Why This Check is Correct:**

The Copy All operation is only useful when the clipboard doesn't already contain the current number of 'A's.

By skipping redundant copies, we ensure that the DFS explores only meaningful states, reducing the search space and improving efficiency.

**Updated DFS Code with Explanation:**

Here’s the relevant part of the DFS code with the condition:

// Option 2: Copy All (only if current != clipboard to avoid redundant copies)

if (current != clipboard) {

int steps = dfs(n, current, current, memo); // Copy All: clipboard becomes current

if (steps != Integer.MAX\_VALUE) {

minSteps = Math.min(minSteps, 1 + steps); // Add 1 for the Copy All operation

}

}

**Example Walkthrough:**

Let’s walk through an example with n = 6:

Start with current = 1, clipboard = 0.

Perform Copy All: current = 1, clipboard = 1.

Perform Paste: current = 2, clipboard = 1.

Perform Copy All: current = 2, clipboard = 2.

Perform Paste: current = 4, clipboard = 2.

Perform Paste: current = 6, clipboard = 2.

At each step, the condition if (current != clipboard) ensures that we don’t perform redundant Copy All operations.

**Key Takeaway:**

The condition if (current != clipboard) is necessary to:

Avoid redundant operations.

Prevent infinite loops or unnecessary computations.

Ensure the DFS explores only meaningful states, leading to an efficient solution.

**Usually bottom up DP solution can develop from pure DFS solution, can you create a DP solution based on DFS solution below ?**

To convert the DFS solution into a bottom-up Dynamic Programming (DP) solution, we need to identify the states and transitions in the DFS solution and then build a DP table to store the results of subproblems.

**Key Observations from the DFS Solution:**

States:

onScreen: The number of 'A's currently on the screen.

onClickBoard: The number of 'A's in the clipboard.

Transitions:

Paste: If the clipboard is not empty, paste the 'A's from the clipboard to the screen.

Copy All: If the clipboard doesn't already contain the current number of 'A's, copy all 'A's from the screen to the clipboard.

Base Case:

If onScreen == n, return 0 (no more steps needed).

Invalid Case:

If onScreen > n, return Integer.MAX\_VALUE (invalid state).

**Bottom-Up DP Approach:**

DP Table:

We use a 2D DP table dp[onScreen][onClickBoard] to store the minimum steps required to reach n 'A's from the current state.

onScreen ranges from 1 to n.

onClickBoard ranges from 0 to n (since the clipboard can hold up to n 'A's).

Initialization:

dp[n][\*] = 0: If onScreen == n, no more steps are needed.

Transitions:

For each state (onScreen, onClickBoard), compute the minimum steps by considering:

Paste: If onClickBoard > 0, transition to (onScreen + onClickBoard, onClickBoard).

Copy All: If onScreen != onClickBoard, transition to (onScreen, onScreen).

Final Result:

The answer is stored in dp[1][0] (starting with 1 'A' on the screen and an empty clipboard).

**Bottom-Up DP Solution in Java:**

**Wrong 2D DP Solution**

class Solution {

public int minSteps(int n) {

// DP table: dp[onScreen][onClickBoard]

int[][] dp = new int[n + 1][n + 1];

// Initialize the DP table with Integer.MAX\_VALUE

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= n; j++) {

dp[i][j] = Integer.MAX\_VALUE;

}

}

// Base case: when onScreen == n, no more steps are needed

for (int j = 0; j <= n; j++) {

dp[n][j] = 0;

}

// Fill the DP table bottom-up

for (int onScreen = n - 1; onScreen >= 1; onScreen--) {

for (int onClickBoard = 0; onClickBoard <= n; onClickBoard++) {

// Option 1: Paste (if clipboard is not empty)

if (onClickBoard > 0 && onScreen + onClickBoard <= n) {

if (dp[onScreen + onClickBoard][onClickBoard] != Integer.MAX\_VALUE) {

dp[onScreen][onClickBoard] = Math.min(dp[onScreen][onClickBoard], 1 + dp[onScreen + onClickBoard][onClickBoard]);

}

}

// Option 2: Copy All (only if current != clipboard to avoid redundant copies)

if (onScreen != onClickBoard) {

if (dp[onScreen][onScreen] != Integer.MAX\_VALUE) {

dp[onScreen][onClickBoard] = Math.min(dp[onScreen][onClickBoard], 1 + dp[onScreen][onScreen]);

}

}

}

}

// The answer is the minimum steps starting with 1 'A' on the screen and an empty clipboard

return dp[1][0];

}

}

**But looks like this update 2D DP solution still have the same error, when n = 3, the output is 2147483647 instead of correct output 3 ?**

**R1 tinking process**

Okay, let's try to figure out why this 2D DP solution isn't working for n=3. The user says the output is 2147483647, which is Integer.MAX\_VALUE, meaning the solution isn't updating the dp[1][0] entry correctly. Let's walk through the code step by step for n=3.

First, the DP table is initialized with all values as Integer.MAX\_VALUE, except for when onScreen is n (3), which is set to 0. So dp[3][any clipboard] = 0.

Now, the code starts filling the table from onScreen = 2 down to 1. Let's see what happens when onScreen is 2.

For onScreen=2 and onClickBoard=0:

- Option 1 (Paste) isn't possible because clipboard is 0.

- Option 2 (Copy All) is allowed because 2 != 0. So check dp[2][2]. But initially, dp[2][2] is MAX\_VALUE. So this path doesn't update anything. So dp[2][0] remains MAX\_VALUE.

For onScreen=2 and onClickBoard=1:

- Option 1: 2+1=3 which is allowed. dp[3][1] is 0, so dp[2][1] = 1 + 0 = 1.

- Option 2: Copy if onScreen != onClickBoard. 2 != 1, so check dp[2][2]. Still MAX\_VALUE. So dp[2][1] is min(MAX, 1 + MAX) which is 1. So dp[2][1] becomes 1.

For onScreen=2 and clipboard=2:

- Paste isn't possible because 2+2=4 >3. So no change.

- Copy is allowed only if 2 != clipboard (which is 2), so no. So dp[2][2] remains MAX\_VALUE.

Now, moving to onScreen=1.

For onScreen=1 and clipboard=0:

- Option 1 (Paste): clipboard is 0, can't paste.

- Option 2: Copy All since 1 != 0. So check dp[1][1]. Initially, dp[1][1] is MAX. So no change. So dp[1][0] remains MAX.

But wait, when we process onScreen=1, clipboard=0, we need to copy to set clipboard to 1, which would then allow pasting. But in the code's current setup, when onScreen=1, the code tries to look at dp[1][1], which hasn't been computed yet. Because when onScreen=1, the code is processing all clipboard values. Let's see what happens for clipboard=1 when onScreen=1.

For onScreen=1 and clipboard=1:

- Paste: 1+1=2. dp[2][1] is 1 (from earlier). So dp[1][1] would be min(MAX, 1 + 1) = 2.

- Then, any state that can reach (1,1) via copy would use this value.

Wait, let's step through the code again for onScreen=1.

When onScreen=1 and clipboard=1:

- Option 1: Paste, 1+1=2. Check dp[2][1], which is 1. So dp[1][1] = 1 + 1 = 2.

- Option 2: Copy is not allowed because clipboard is already 1. So no change.

But in the code's loop, when onScreen is processed in reverse order (from n-1 to 1), the clipboard loop runs from 0 to n. So for onScreen=1, clipboard runs from 0 to 3.

Let's process each clipboard value for onScreen=1:

clipboard=0:

- Can't paste. Try to copy. So check dp[1][1]. But initially, dp[1][1] is MAX. So dp[1][0] remains MAX.

clipboard=1:

- Paste: 1+1=2. dp[2][1] is 1. So dp[1][1] = 1 + 1 = 2.

- Copy is not allowed (1=1). So no change.

clipboard=2:

- Paste: 1+2=3. So dp[3][2] is 0. So dp[1][2] would be 1 + 0 = 1. But how does clipboard=2 happen here? Because the clipboard can be any value up to n. But in reality, for onScreen=1, the clipboard could have been set by a previous copy.

Wait, maybe the main issue is the order in which the clipboard values are processed. Let's think: when onScreen=1, clipboard=1 is handled first, which might update dp[1][1], and then clipboard=0 can refer to dp[1][1] after it's been updated. But in the current code, the clipboard values are processed from 0 to n. So for onScreen=1:

First, clipboard=0: trying to copy, which requires checking dp[1][1], which is MAX at that point. Then, clipboard=1: which updates dp[1][1] to 2. Then clipboard=2: which may paste to 3. Then clipboard=3: which would paste to 4 (invalid).

**But for clipboard=0, when processing onScreen=1, the code checks if (onScreen != clipboard) → yes, 1 !=0. Then checks if dp[1][1] != MAX. Initially, dp[1][1] is MAX, so this condition fails, and the code doesn't update dp[1][0]. But after processing clipboard=1, dp[1][1] is set to 2. However, clipboard=0 is processed before clipboard=1, so the code misses the opportunity to use the updated dp[1][1] when processing clipboard=0.**

**So the problem is the order of processing the clipboard values. For the code to correctly use the updated values of dp[onScreen][onScreen], the clipboard values need to be processed in reverse order (from high to low) so that when we process a lower clipboard value, the higher ones (like clipboard=1 for onScreen=1) have already been processed.**

Alternatively, maybe the code is not properly propagating the values from higher clipboard values to lower ones. Let's see.

When onScreen=1 and clipboard=0:

The code checks if we can copy (since 1 !=0). It tries to take 1 + dp[1][1]. But dp[1][1] was not yet computed because clipboard=1 is processed after clipboard=0. So when clipboard=0 is processed, dp[1][1] is still MAX, leading to no update. Then, when clipboard=1 is processed, dp[1][1] gets updated to 2. But the clipboard=0 case already processed and can't benefit from that.

Therefore, the order of iterating clipboard values from 0 to n is incorrect. We need to process clipboard values in reverse order so that when we process a lower clipboard value, the higher clipboard values (like clipboard=onScreen) have already been computed.

So, changing the inner loop to iterate clipboard from n downto 0 instead of 0 to n might help. Let's adjust that.

In the current code:

for (int onClickBoard = 0; onClickBoard <= n; onClickBoard++) {

This loops from 0 to n. Let's change this to:

for (int onClickBoard = n; onClickBoard >= 0; onClickBoard--) {

This way, when processing clipboard values in reverse order, the code would process higher clipboard values first, which may allow the lower ones to use updated values.

Let's see how that would work for onScreen=1:

Processing clipboard=3, then 2, then 1, then 0.

For clipboard=3:

- onScreen=1, clipboard=3. Paste would add 3 → 4 (invalid). So no change.

- Copy is possible (1 !=3). So check dp[1][1]. But dp[1][1] is MAX initially. So no change.

clipboard=2:

- Paste 1+2=3. dp[3][2] is 0. So dp[1][2] = 1 + 0 =1.

- Copy is allowed (1 !=2). So check dp[1][1]. Still MAX. So no change.

clipboard=1:

- Paste 1+1=2. dp[2][1] is 1. So dp[1][1] = 1 +1 =2.

- Copy is not allowed.

clipboard=0:

- Paste not possible.

- Copy is allowed. So check dp[1][1], which is now 2. So dp[1][0] = min(MAX, 1 +2) =3.

Ah! Now, by processing clipboard in reverse order, when clipboard=0 is processed, dp[1][1] has already been updated to 2. So when we do the Copy All option for clipboard=0, we add 1 + dp[1][1] (which is 3), leading to dp[1][0] being 3. Which is the correct answer for n=3.

So the error in the code is the order in which the clipboard values are processed. Changing the inner loop to iterate clipboard in reverse order (from n downto 0) fixes this issue.

**Therefore, the corrected code should reverse the inner loop's iteration order.**

Okay, let's tackle this problem step by step. The user has been struggling with converting a DFS solution for LeetCode problem 650 into a correct 2D DP solution. The previous attempts resulted in incorrect outputs for cases like n=3, and after several iterations, the correct solution was found by iterating the clipboard values in reverse order. Now, the user confirms that the latest solution passes all test cases, so let's break down why this approach works.

First, let's recap the problem: We need to find the minimum number of operations to get exactly 'n' 'A's on a notepad using only two operations: Copy All and Paste. The key is to model the states correctly and transition between them efficiently.

The initial DFS approach uses recursion with memoization to explore all possible states. However, converting this into a DP solution requires careful handling of state transitions and dependencies. The main issue in the earlier DP attempts was the order of processing states, leading to incorrect or uncomputed values being referenced.

In the corrected DP solution:

1. State Representation: The DP table `dp[onScreen][onClickBoard]` represents the minimum steps needed to reach 'n' 'A's starting from 'onScreen' 'A's on the screen and 'onClickBoard' 'A's in the clipboard.

2. Base Case Initialization: When `onScreen == n`, no more steps are needed, so `dp[n][\*] = 0`.

3. Order of Iteration: The outer loop iterates `onScreen` from `n-1` down to `1`, and the inner loop iterates `onClickBoard` from `n` down to `0`. This reverse order ensures that when processing a state `(onScreen, onClickBoard)`, any dependent states (like `onScreen + onClickBoard` for Paste or `onScreen` for Copy All) have already been computed.

4. Transitions:

- Paste Operation: If there are 'A's in the clipboard (`onClickBoard > 0`) and pasting doesn't exceed 'n', update the current state based on the state after pasting.

- Copy All Operation: Only perform if the clipboard doesn't already match the current screen. Update the current state based on the state after copying.

**By reversing the clipboard iteration, we ensure that when handling the Copy All operation, the state `(onScreen, onScreen)` (which is the result of copying) has already been processed, allowing the current state to correctly reference its value.**

For example, with `n=3`:

- Starting from `onScreen=1`, `onClickBoard=0`, the Copy All operation updates `onClickBoard` to `1`. Since we process higher clipboard values first, when we get to `onClickBoard=0`, the value for `onClickBoard=1` is already computed, allowing the Copy All operation to correctly contribute to the minimum steps.

This approach efficiently computes the minimum steps by leveraging the dependency order of states, ensuring all necessary previous states are resolved before they're needed. The result is a correct and efficient 2D DP solution that handles all test cases, including edge cases like `n=3`.

**Correct 2D DP Solution**

class Solution {

public int minSteps(int n) {

int[][] dp = new int[n + 1][n + 1];

// Initialize all states to a large value (invalid/unreachable)

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= n; j++) {

dp[i][j] = Integer.MAX\_VALUE;

}

}

// Base case: when onScreen == n, no steps needed

for (int j = 0; j <= n; j++) {

dp[n][j] = 0;

}

// Iterate from n-1 down to 1 (bottom-up)

for (int onScreen = n - 1; onScreen >= 1; onScreen--) {

// Process clipboard values in reverse order (critical fix)

for (int onClickBoard = n; onClickBoard >= 0; onClickBoard--) {

// Option 1: Paste (if clipboard is not empty)

if (onClickBoard > 0 && onScreen + onClickBoard <= n) {

if (dp[onScreen + onClickBoard][onClickBoard] != Integer.MAX\_VALUE) {

dp[onScreen][onClickBoard] = Math.min(

dp[onScreen][onClickBoard],

1 + dp[onScreen + onClickBoard][onClickBoard]

);

}

}

// Option 2: Copy All (only if clipboard != current)

if (onScreen != onClickBoard) {

if (dp[onScreen][onScreen] != Integer.MAX\_VALUE) {

dp[onScreen][onClickBoard] = Math.min(

dp[onScreen][onClickBoard],

1 + dp[onScreen][onScreen]

);

}

}

}

}

// Answer: start with 1 'A' on screen and 0 in clipboard

return dp[1][0];

}

}

**Key Explanations**

**1. State Representation**

dp[onScreen][onClickBoard]: Stores the minimum steps needed to reach exactly n 'A's starting with onScreen 'A's on the screen and onClickBoard 'A's in the clipboard.

**2. Base Case**

When onScreen == n, no more steps are needed:

for (int j = 0; j <= n; j++) dp[n][j] = 0;

**3. Order of Iteration**

**Outer Loop: Iterate onScreen from n-1 down to 1 (ensures larger onScreen values are computed first).**

**Inner Loop: Iterate onClickBoard from n down to 0 (critical fix to resolve dependency issues).**

**4. Transitions**

Paste Operation:

If onClickBoard > 0 (clipboard not empty) and onScreen + onClickBoard <= n (no overflow), update:

dp[onScreen][onClickBoard] = min(..., 1 + dp[onScreen + onClickBoard][onClickBoard])

This adds 1 step (for Paste) and transitions to the new state after pasting.

Copy All Operation:

Only allowed if onScreen != onClickBoard (avoid redundant copies). Update:

dp[onScreen][onClickBoard] = min(..., 1 + dp[onScreen][onScreen])

This adds 1 step (for Copy All) and transitions to the state where clipboard matches the screen.

**5. Why Reverse Clipboard Iteration Works**

**By processing onClickBoard from n to 0, the Copy All operation (which sets onClickBoard = onScreen) ensures that when we process a state (onScreen, onClickBoard), the value for (onScreen, onScreen) has already been computed. This prevents stale or uninitialized values from being used.**

**Example Walkthrough for n = 3**

Initial State: onScreen = 1, onClickBoard = 0 (steps = 0).

Copy All: onClickBoard becomes 1 (steps = 1). State: (1, 1).

Paste: onScreen becomes 2 (steps = 2). State: (2, 1).

Paste Again: onScreen becomes 3 (steps = 3). State: (3, 1) (base case).

The DP table correctly computes dp[1][0] = 3.

**Complexity Analysis**

Time: (O(n^2)) – We iterate over all possible states (onScreen, onClickBoard), where each ranges from 0 to n.

Space: (O(n^2)) – The DP table stores results for all states.

**Conclusion**

By iterating clipboard values in reverse order, we ensure dependencies between states are resolved correctly. This approach guarantees that all necessary previous states are computed before they are referenced, leading to an accurate and efficient solution. The final code passes all test cases, including edge cases like n = 3! 🎉

**We cannot convert this 2D DP solution into 1D, right ?**

You're correct that the 2D DP solution for LeetCode 650 cannot be directly converted into a 1D DP solution. The reason is that the problem inherently depends on two states:

onScreen: The number of 'A's currently on the screen.

onClickBoard: The number of 'A's in the clipboard.

These two states are independent and must be tracked simultaneously to compute the minimum steps correctly. A 1D DP solution would only allow tracking one state, which is insufficient for this problem.

**Why 1D DP Doesn't Work**

Dependency on Two States:

The Paste operation depends on both onScreen and onClickBoard.

The Copy All operation updates onClickBoard based on onScreen.

Loss of Information:

A 1D DP array (e.g., dp[onScreen]) would lose the information about the clipboard state (onClickBoard), making it impossible to correctly compute transitions.

Example:

For n = 3, the transitions depend on both the current screen state and the clipboard state. A 1D DP solution cannot capture this relationship.

**Alternative Approach: Greedy Solution**

While a 1D DP solution isn't feasible, there is a greedy solution that leverages the prime factorization of n. This approach is based on the observation that the minimum steps to reach n 'A's is the sum of the prime factors of n.

**Greedy Solution Code:**

class Solution {

public int minSteps(int n) {

int steps = 0;

for (int factor = 2; factor <= n; factor++) {

while (n % factor == 0) {

steps += factor;

n /= factor;

}

}

return steps;

}

}

**Explanation:**

Prime Factorization:

The problem reduces to finding the prime factors of n.

For example, if n = 6, its prime factors are 2 and 3. The minimum steps are 2 + 3 = 5.

Why It Works:

Each prime factor represents a sequence of operations: Copy All followed by Paste multiple times.

The sum of the prime factors gives the total number of operations.

**Complexity:**

Time: (O(sqrt{n})) (for prime factorization).

Space: (O(1)) (constant space).

**Comparison of Approaches**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Time Complexity | Space Complexity | Notes |
| **2D DP** | (O(n^2)) | (O(n^2)) | Tracks both  onScreen  and  onClickBoard  states. |
| **Greedy (Prime Factors)** | (O(sqrt{n})) | (O(1)) | Efficient and concise, but doesn't generalize to all DP problems. |

**Conclusion**

The 2D DP solution is necessary for this problem because it requires tracking two independent states (onScreen and onClickBoard).

A 1D DP solution cannot capture the required information and is not feasible.

The greedy solution provides an efficient alternative by leveraging the mathematical properties of the problem.

If you have further questions or need clarification, feel free to ask! 😊

**Refer to**

[L991.Broken Calculator (Ref.L650,L2749)](note://WEBf7687f1214cd00462721a353fd513567)

[L2749.Minimum Operations to Make the Integer Zero (Ref.L650,L991)](note://WEB5dbaf68e90c1e1dfc4513caea61adf7d)