650. 2 Keys Keyboard

**Dynamic Programming** 

## **Problem Description**

Medium

perform two operations: • Copy All: You can copy everything present on the screen at that moment.

The given problem describes a situation where we have a notepad that initially contains only one character, 'A'. We are allowed to

• Paste: You can paste the last copied set of characters onto the screen.

- The objective is to figure out the minimum number of operations needed to get exactly 'n' occurrences of 'A' on the screen. We

are to return this minimum number of operations.

To approach this problem, we need to think in terms of how we can build up 'n' characters from the initial single 'A' in the least

number of steps. Intuitively, pasting the same characters repeatedly seems like an efficient way to reach our goal. However, we

## Intuition

can only paste what we've last copied, and we can't do a partial copy. Given this constraint, a key insight is understanding that getting to 'n' 'A's efficiently often means generating 'A's in multiples that are factors of 'n'. If we can create a sequence where we copy 'x' 'A's and then paste them (n/x - 1) times, we will end up with 'n'

characters in fewer steps, especially if 'x' is large. The solution is recursive in nature and works by dividing the problem into smaller subproblems. Each time we come across a factor 'i' of 'n', we can consider the minimum number of steps required to get 'i' number of 'A's and then add the steps required to

copy it (which is 'i') and paste it (n/i - 1) times to complete the 'n' characters. The function uses memoization (via the @cache decorator), to store and reuse the results of subproblems which drastically reduces the number of redundant calculations, effectively improving performance.

Solution Approach The solution applies a depth-first search (DFS) strategy to solve the problem by breaking it into smaller subproblems. The dfs

# For other values of n, the function iterates over possible factors of n starting from 2 (since 1 would not reduce the problem),

operations).

checking if n % i == 0 to determine if i is a factor. The significance of checking factors is that if n can be divided by i, then we can achieve n 'A's by first achieving i 'A's and then copying and pasting the i 'A's group (n/i) times (which needs i

If a factor is found, the function calls itself recursively with the new target n // i, which represents the subproblem of reaching

The base case of the recursion is when n equals 1, at which point no further operations are required (return 0).

function is a recursive function that finds the minimum number of operations for a given target n.

operations, so we calculate and store the minimum (ans = min(ans, dfs(n // i) + i)). The while loop ensures that we're only iterating up to the square root of n. This is an optimization because if n is not prime, it must have a factor less than or equal to its square root, thus we don't need to check beyond that. Python's @cache decorator is used on the dfs function to memoize its results. This means that the results of the dfs calculations

for each unique input are stored, so if the same value of n is passed to the function again, it doesn't compute it but retrieves the

result from the cache. This significantly reduces the number of calculations and the time complexity of the algorithm, as it

After defining the dfs function with memoization, the main function minSteps directly calls and returns the result of dfs(n),

i 'A's. This recursive call adds the cost of i operations to the result (dfs(n // i) + i). We want the minimum number of

In summary, the solution is recursive, uses memoization for optimization, and leverages the mathematical insight that the number of steps for n 'A's is linked to the factors of n. **Example Walkthrough** 

itself because we need to perform at least one copy and paste). Here is a step-by-step walk through using the factor of 4:

We start with one 'A' on the notepad. We need to reach exactly 8 'A's. We look for factors of 8, which are 2 and 4 (excluding 8

## 5. Step 4: Paste (operation count: 4) - Notepad content is 'AAAA'

With the 'AA' ready:

for any n.

Solution Implementation

def dfs(current):

def minSteps(self, n: int) -> int:

if current == 1:

return 0

divisor += 1

from functools import lru\_cache

if current % divisor == 0:

1. Step 5: Copy All (operation count: 5)

1. Initial State: Notepad content is 'A' (operation count: 0)

3. Step 2: Paste (operation count: 2) - Notepad content is 'AA'

4. Step 3: Paste (operation count: 3) - Notepad content is 'AAA'

2. Step 1: Copy All (operation count: 1)

one extra copy and one paste.

Let's illustrate the solution approach with a small example using n = 8 'A's.

prevents us from recomputing the minimum steps for values we've already solved.

which computes the minimum number of steps needed to reach n 'A's on the notepad.

2. Step 6: Paste (operation count: 6) - Notepad content is 'AAAAAAA'

In this example, we needed a total of 6 operations to get to 8 'A's. We created 4 'A's in 4 operations and then doubled that with

use the same logic for the factors themselves. @cache would ensure that the calculation for each amount of 'A's is done only

Realizing that the factors of n are critical to finding the solution, we can see that as n gets larger or is a prime number, the

number of operations needed will increase accordingly, and the algorithm efficiently finds the optimal set of operations needed

1. Initial State: Notepad content is 'A' (operation count: 0) 2. Step 1: Copy All (operation count: 1)

Now, we have 'AAAA' on the notepad. The factor 4 is reached with 4 operations. Next:

1. Step 3: Copy All (operation count: 3) 2. Step 4: Paste (operation count: 4) - Notepad content is 'AAAA' 3. Step 5: Copy All (operation count: 5)

4. Step 6: Paste (operation count: 6) - Notepad content is 'AAAAAAAA'

3. Step 2: Paste (operation count: 2) - Notepad content is 'AA'

We could use the other factor, which is 2, to achieve our goal:

- The factor 2 also took 6 operations, so in this case, both factors provide the same result. The recursive function dfs would calculate the number of steps for getting 'A' in quantities of factors of 8 and would recursively
- once, even if needed multiple times throughout the recursion. For this example, the minimum number of operations that dfs would report is 6, which matches our manual calculation.
- **Python** class Solution:

@lru cache(maxsize=None) # Use LRU cache to memoize the results of subproblems

# If current is divisible by divisor, then it can be obtained by

# add the number of steps needed for the current divisor

ans = min(ans, dfs(current // divisor) + divisor)

// This method calculates the minimum number of steps to get 'n' 'A's on the notepad

// Start dividing 'n' from factor 2 onwards, as 1 wouldn't change the number

// This is because, in terms of operations, if 'n' is divisible by 'i',

// While 'n' is divisible by 'i', keep dividing it and add 'i' to the result.

// it means we can get to 'n' by doing 'i' operations (copy, paste 'i'-1 times) that many times.

// Initialize the result to store the minimum number of steps

steps += i; // Add the factor to the total steps

// A memoization table to store results of subproblems

// A memoization table to store results of subproblems

// Helper function to perform the depth-first search

// If only one 'A' is needed, 0 steps are required

// Try to find the minimal steps by finding divisors of 'n'

ans = Math.min(ans, dfs(n / i) + i);

// Save the computed answer to the memoization table

// Return the minimum number of steps calculated

// Function to compute the minimum steps required to get 'n' 'A's on the notepad

// Call the recursive depth-first search function to compute the answer

// If the result has already been computed, return it instead of recomputing

const fill = (arr: number[], value: number, start: number, end: number): void => {

# Base case: If the current number is 1, no steps are needed

# Iterate over possible divisors to find the minimum steps

# Call the recursive function with the initial value

# Initialize ans with the maximum possible steps needed (which is n)

memo.fill = (value: number, start?: number, end?: number): number[] => {

// Example override of the initial fill function to provide the specific functionality needed.

// Initialize the answer with the maximum value, which is copying 'A' one by one

// Initialize the memoization table with '-1' to indicate that no subproblem is solved yet

n /= i; // Divide 'n' by the current factor 'i'

// Once 'n' is reduced to 1, we have found the minimum steps and return it

// starting with only one 'A'. It involves a series of copy—all and paste operations.

# Compute the minimum steps recursively for the quotient and

# Base case: If the current number is 1, no steps are needed

# adding the divisor to itself (n / divisor) times

# In case n is a prime number, the answer would be n itself

# Initialize ans with the maximum possible steps needed (which is n) ans = current # Iterate over possible divisors to find the minimum steps divisor = 2while divisor \* divisor <= current:</pre>

#### return ans # Call the recursive function with the initial value return dfs(n)

public int minSteps(int n) {

for (int i = 2; n > 1; ++i) {

while (n % i == 0) {

int steps = 0;

return steps;

```
# print(s.minSteps(10)) # Output would be 7
Java
```

C++

public:

class Solution {

vector<int> memo;

# Example usage:

# s = Solution()

class Solution {

```
// Main function to compute the minimum steps required to get 'n' 'A's on the notepad
int minSteps(int n) {
    // Initialize the memoization table with '-1', to indicate that no subproblem is solved yet
   memo.assign(n + 1, -1);
    // Call the recursive depth-first-search function to compute the answer
    return dfs(n);
// Helper function to perform the depth-first search
int dfs(int n) {
   // If only one 'A' is needed, 0 steps are required
    if (n == 1) return 0;
   // If the result has already been computed, return it instead of recomputing
    if (memo[n] != -1) return memo[n];
   // Initialize the answer with the maximum value, which is 'n' (copying 'A' one by one)
    int ans = n;
   // Try to find the minimal steps by finding divisors of 'n'
    for (int i = 2; i * i <= n; ++i) {
       // If 'i' is a divisor of 'n'
       if (n % i == 0) {
            // Recursively solve for the smaller problem 'n / i' and add 'i' steps
            // (the steps to paste 'A's 'i-1' times after copying once)
            ans = min(ans, dfs(n / i) + i);
   // Save the computed answer to the memoization table
    memo[n] = ans;
    // Return the minimum number of steps calculated
    return ans;
```

// Recursively solve for the smaller problem 'n / i' and add 'i' steps (the steps to paste 'A's 'i - 1' times after copyi

#### for (let i = start; i < end; ++i) {</pre> arr[i] = value; **}**;

**}**;

**TypeScript** 

**}**;

const memo: number[] = [];

return dfs(n);

let ans = n;

memo[n] = ans;

return ans;

return memo;

def dfs(current):

if current == 1:

return 0

ans = current

divisor = 2

return dfs(n)

# print(s.minSteps(10)) # Output would be 7

Time and Space Complexity

# Example usage:

# s = Solution()

**Time Complexity** 

memo.fill(-1, 0, n + 1);

const dfs = (n: number): number => {

if (memo[n] !== -1) return memo[n];

for (let i = 2; i \* i <= n; ++i) {

if (n % i === 0) {

// If 'i' is a divisor of 'n'

if (n === 1) return 0;

const minSteps = (n: number): number => {

class Solution: def minSteps(self, n: int) -> int: from functools import lru\_cache @lru cache(maxsize=None) # Use LRU cache to memoize the results of subproblems

fill(memo, value, start || 0, end || memo.length);

```
while divisor * divisor <= current:</pre>
    # If current is divisible by divisor, then it can be obtained by
    # adding the divisor to itself (n / divisor) times
    if current % divisor == 0:
        # Compute the minimum steps recursively for the quotient and
        # add the number of steps needed for the current divisor
        ans = min(ans, dfs(current // divisor) + divisor)
    divisor += 1
# In case n is a prime number, the answer would be n itself
return ans
```

// Utility function to fill an array from the `start` index up to but not including the `end` index with the `value`.

## search approach that looks for the minimum number of operations to get n characters by only using copy and paste operations. For each n, the function iterates from 2 to the square root of n to find factors, and at each factor i, it recursively calculates

## dfs(n // i) + i, which adds the operations needed to obtain n from n // i plus the operations to get i characters in the first place.

**Space Complexity** 

The worst-case time complexity is hard to evaluate directly because of the recursive nature and the caching of intermediate results. However, a factor that influences the time complexity is the number of divisors of n. In the worst case, these divisors

The time complexity of the minSteps function primarily is from the dfs function that it calls. The dfs function is a depth-first

form a tree, where we explore n // i for each factor i of n. Since the largest prime less than m can be O(m/log(m)) and the prime factorization of n can have at most log(n) terms, the upper bound on the time complexity can be seen as O((n / log(n)) \* log(n)), which simplifies to O(n). The use of

caching/memoization via the @cache decorator avoids re-calculating the number of steps for the same n in the recursive calls.

The space complexity is mainly due to two contributors - the system call stack due to recursion and the space used for caching the results. In the worst case, the recursion depth is equal to the number of distinct prime factors which are O(log(n)). Hence, O(log(n)) space is used in the call stack. Additionally, the @cache decorator potentially stores the result for every unique argument to dfs between 1 and n, therefore,

taking up O(n) space in the worst case, if very little overlapping occurs in the computations. As a result, the overall space

complexity of the algorithm is O(n + log(n)), which simplifies to O(n) when we drop the lower-order term.