651. 4 Keys Keyboard

Medium Math Dynamic Programming

Leetcode Link

Problem Description

Imagine you are given a special text editor which only has four keys:

- `A`: This key simply prints one 'A' character on the screen.
- `Ctrl-A`: This key selects everything that is currently on the screen.
- `Ctrl-C`: This key is used to copy the selected text into a buffer. This buffer holds the copied text until it is overwritten by a subsequent use of `Ctrl-C`.
- `Ctrl-V`: This key pastes the content of the buffer onto the screen immediately after what's already displayed.

Your task is to figure out the maximum number of 'A's that you can print on the screen if you are allowed to press the keys a total of **n** times. The problem is essentially to find the optimal way to use these key presses to maximize the number of 'A's.

Intuition

To arrive at the solution, we should think about the optimal sequence of key presses as n grows larger. Initially, pressing `A` is our only option. However, once we have the ability to use `Ctrl-A`, `Ctrl-C`, and `Ctrl-V`, we need to strategically decide when to use these to multiply the amount of 'A's we display.

Pasting (`Ctrl-V`) is only useful if we've copied (`Ctrl-C`) a significant amount of 'A's to begin with. Moreover, for each sequence involving `Ctrl-A`, `Ctrl-C`, followed by multiple `Ctrl-V` presses, there is a point where the best move is to start a new sequence of `Ctrl-A`, `Ctrl-C`, then `Ctrl-V`s rather than continuing to `Ctrl-V`.

The solution involves using dynamic programming to keep track of the best possible outcome for each number of presses up to n. The `dp` array represents the maximum number of 'A's we can get for every number of key presses from 0 to n. The formula `dp[i] = max(dp[i], dp[j - 1] * (i - j))` updates the `dp[i]` with the maximum value between the current `dp[i]` and the product of `dp[j - 1]` and `(i - j)`. Here, `dp[j - 1]` represents the number of 'A's before we press `Ctrl-A`, `Ctrl-C`, and `(i - j)` represents the number of times we can press `Ctrl-V` after `i - j - 2` key presses spent on selecting and copying.

By iterating through all possible breakpoints for copying and pasting, the algorithm ensures that it finds the maximum number of 'A's possible for each total number of key presses.

Solution Approach

The solution employs dynamic programming, which is an optimization technique used to solve problems by breaking them down into simpler subproblems. Here, the goal is to maximize the number of 'A's that can be printed with a given number of key presses.

The implementation defines a `dp` array of size `n + 1`, where each element at index `i` holds the maximum number of 'A's that can be printed using `i` key presses. The initial values in the `dp` array are set equal to their indices, which corresponds to pressing the 'A' key as many times as possible without using any other keys.

The key idea is to consider that, for each position `i` in the `dp` array, the maximum number of 'A's `dp[i]` can be obtained by:

- Using the first `j-1` presses to get a certain number of 'A's (denoted as `dp[j-1]`)
- Then, using one press to `Ctrl-A`, one press to `Ctrl-C`, and the remaining `i - j` presses to `Ctrl-V` multiple times to double, triple, or further multiply the number of 'A's we had at `dp[j-1]`

This is expressed in the code as:

```
dp[i] = max(dp[i], dp[j - 1] * (i - j))
```

This line iterates through all potential breakpoints `j`, where we switch from pressing 'A' to using the `Ctrl-A`, `Ctrl-C`, and then `Ctrl-V` sequence. For each `j`, it considers the number of 'A's we had at `j-1` presses, and multiplies this by the number of times we can paste, which is `i - j`.

The loops in the code are constructed as follows:

1. The outer loop goes from `3` to `n+1` because we need at least 3 key presses to perform the full sequence of `Ctrl-A`, `Ctrl-C`, and `Ctrl-V` at least once.
2. The inner loop tries to find the optimal point `j` where we should switch from pressing 'A' to using the multi-press sequence. It runs from `2` to `i-1`.

The process continues until we check all possible combinations of key presses up to `n`. The final answer is the last element of the `dp` array, which contains the maximum number of 'A's for `n` key presses.

```
class Solution:
    def maxA(self, n: int) -> int:
        dp = list(range(n + 1))
        for i in range(3, n + 1):
            for j in range(2, i - 1):
                dp[i] = max(dp[i], dp[j - 1] * (i - j))
        return dp[-1]
```

With this approach, the solution maximizes the output by finding the best time to perform each action, resulting in the highest number of 'A's possible.

Example Walkthrough

Let's consider an example where the total number of key presses allowed is `n = 7`. We will walk through the dynamic programming approach explained in the solution to understand how we can maximize the number of 'A's printed.

1. Initialize `dp` array with the number of key presses because the minimum we can do is press the 'A' key `n` times, hence `dp = [0, 1, 2, 3, 4, 5, 6, 7]`.
2. The first 3 key presses would go in the normal way – printing 'A' three times since we need at least 3 key presses to start using the `Ctrl-A`, `Ctrl-C`, `Ctrl-V` sequence. So, `dp[1] = 1`, `dp[2] = 2`, and `dp[3] = 3`.
3. Now, for `i = 4`, we have two options: either press 'A' one more time to have `dp[4] = 4` or do the sequence `Ctrl-A`, `Ctrl-C`, and `Ctrl-V` to double the 2 'A's we had at `dp[2]`. This would give us `dp[4] = dp[2] * (4 - 2) = 2 * 2 = 4`. We pick the maximum, which is still `4`.
4. For `i = 5`, again we could press 'A' (`dp[5] = 5`) or use the sequence `Ctrl-A`, `Ctrl-C`, `Ctrl-V`, `Ctrl-V`. There are now two places this sequence might start:
   - After 1 'A' (`dp[1]`) and then paste it 3 times (`i - 2`), giving us `dp[5] = dp[1] * (5 - 2) = 1 * 3 = 3`, or
   - After 2 'A's (`dp[2]`) and then paste 2 times, giving us `dp[5] = dp[2] * (5 - 3) = 2 * 2 = 4`. The maximum is `dp[5] = 5`, so we keep it as is.
5. For `i = 6`, we consider starting the sequence after having 1, 2, or 3 'A's and then using `Ctrl-V`:
   - `dp[6] = dp[1] * (6 - 2) = 1 * 4 = 4` (starting after 1 'A')
   - `dp[6] = dp[2] * (6 - 3) = 2 * 3 = 6` (starting after 2 'A's)
   - `dp[6] = dp[3] * (6 - 4) = 3 * 2 = 6` (starting after 3 'A's) The maximum we can get is `dp[6] = 6`, and as it's equal to simply pressing 'A' six times, no change is made.
6. Finally, for `i = 7`, we check the same as above, considering where to start the sequence:
   - `dp[7] = dp[1] * (7 - 2) = 1 * 5 = 5` (starting after 1 'A')
   - `dp[7] = dp[2] * (7 - 3) = 2 * 4 = 8` (starting after 2 'A's)
   - `dp[7] = dp[3] * (7 - 4) = 3 * 3 = 9` (starting after 3 'A's)
   - `dp[7] = dp[4] * (7 - 5) = 4 * 2 = 8` (starting after 4 'A's) The best we can do is start the sequence after having 3 A's, leading to `dp[7] = 9`.

So, the maximum number of 'A's we can print with 7 key presses is `dp[7] = 9`. The solution identifies that it's more efficient to start the `Ctrl-A`, `Ctrl-C`, `Ctrl-V` sequence after having pressed 'A' three times, and then use the remaining key presses to multiply these 'A's.

Python Solution

```
class Solution:
    def maxA(self, n: int) -> int:
        # Initialize a list `dp` with values equal to the indexes
        # as the max achievable by direct A keypresses
        dp = list(range(n + 1))

        # Start from the 3rd index because the first two do not need any calculations
        for i in range(3, n + 1):
            # Loop over the range to find out the breaking point for optimal
            # CTRL-V presses after a CTRL-A, CTRL-C sequence.
            for j in range(2, i - 1):
                # Calculate the max between current dp value and the sequence
                # of selecting all (dp[j - 1]), copying and pasting (i - j) times.
                # i-j represents the remaining key presses after (j-1) presses are used
                # to reach optimal select and copy.
                dp[i] = max(dp[i], dp[j - 1] * (i - j))
        # Return the last element which contains the maximum number of 'A's that can be produced
        return dp[-1]
```

Java Solution

```
class Solution {
    // Function to find out maximum number of 'A's that can be produced by
    // pressing keys in a certain order, given a limit of key presses
    public int maxA(int n) {
        // Initialize an array to store the subproblem results
        int[] dp = new int[n + 1];

        // Base case: For 1 presses, you can at most get 1 'A's by pressing 'A' 1 times
        for (int i = 0; i <= n; ++i) {
            dp[i] = i;
        }

        // Start filling dp table for each number of presses (index)
        for (int i = 3; i <= n; ++i) {
            // Explore the effect of using Ctrl-V after pressing Ctrl-A and Ctrl-C
            // starting from j=2 as you need at least one 'A' for Ctrl-A and Ctrl-C to make sense
            for (int j = 2; j < i - 1; ++j) {
                // Calculate the maximum of either just pressing 'A'
                // or using a combination of Ctrl-A, Ctrl-C, and Ctrl-V
                // dp[j - 1] represents the number of 'A's on the screen before copying,
                // (i - j) represents the remaining number of presses after copying which is used solely for pasting
                // the product of those two numbers represent the total 'A's after utilizing the copy-paste operation
                dp[i] = Math.max(dp[i], dp[j - 1] * (i - j));
            }
        }

        // The last element of dp array contains the answer for n key presses
        return dp[n];
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>

class Solution {
public:
    // The maxA function calculates the maximum number of 'A's that can
    // be produced by pressing keys on the keyboard for 'n' times.
    int maxA(int n) {
        // Initialize a dynamic programming array to store the intermediate results.
        // dp[i] represents the maximum number of 'A's that can be produced by pressing
        // keys i times.
        std::vector<int> dp(n + 1);

        // The base cases: If you press i times, the maximum you can get is i 'A's,
        // by pressing the key 'A' i times (for first two i's, for i > 2,
        // you might get more by using Ctrl-A, Ctrl-C, and Ctrl-V.
        std::iota(dp.begin(), dp.end(), 0);

        // Loop through each possible number of presses from 3 to n
        for (int i = 3; i <= n; ++i) {
            // Check all the possibilities of the last sequence of 'Ctrl-A, Ctrl-C followed by
            // Ctrl-V's'. The Ctrl-A, Ctrl-C must be at some point j before i, and the remaining
            // presses (i - j) are for Ctrl-V.
            for (int j = 2; j < i - 1; ++j) {
                // The number of 'A's after pressing Ctrl-V (i - j times) following a copy operation
                // at j-th press will be the number of 'A's at (j - 1) multiplied by (i - j).
                // We update dp[i] if this yields more 'A's than we've seen for i presses.
                dp[i] = std::max(dp[i], dp[j - 1] * (i - j));
            }
        }

        // After filling dp array, the answer for n presses is at dp[n].
        return dp[n];
    }
};
```

Typescript Solution

```
// Initialize a global dynamic programming array to store the intermediate results.
// dp[i] represents the maximum number of 'A's that can be produced by pressing
// keys i times.
const dp: number[] = [];

// The maxA function calculates the maximum number of 'A's that can
// be produced by pressing keys on the keyboard for 'n' times.
function maxA(n: number): number {
    // Resize the dp array to store results for up to n key presses.
    dp.length = n + 1;

    // The base case configurations: If you press i times, the maximum you can get is i 'A's,
    // by pressing the 'A' key i times (for the first few i's).
    for (let i = 0; i <= n; i++) {
        // This simulates pressing the 'A' key i times.
        dp[i] = i;
    }

    // Loop through each possible number of presses from 3 to n
    for (let i = 3; i <= n; i++) {
        // Check all possibilities where the last sequence of keys pressed is
        // 'Ctrl-A, Ctrl-C followed by Ctrl-V's'. The Ctrl-A, Ctrl-C must
        // happen at some point j before i, so the remaining
        // presses (i - j) are for Ctrl-V.
        for (let j = 2; j < i - 1; j++) {
            // Any sequence of operations ending with Ctrl-A, Ctrl-C and (i - j) Ctrl-Vs
            // results in the screen containing the clipboard contents repeated (i - j) times.
            // So the optimal sequence length at this state is the number of 'A's that
            // can be produced by j - 1 presses times (i - j) for the subsequent Ctrl-V presses.
            const current = dp[j - 1] * (i - j);

            // Update dp[i] to the maximum number of 'A's seen after i presses.
            dp[i] = Math.max(dp[i], current);
        }
    }

    // After filling out the dp array, the answer for n presses is at dp[n].
    return dp[n];
}

// Example usage:
// let result = maxA(7); // Should calculate the maximum number of 'A's for 7 key presses
```

Time and Space Complexity

The time complexity of the provided code is $O(n^3)$. This is because there are two nested loops where the outer loop runs from 3 to `n` (in the range of `n-2` iterations), and the inner loop runs from 2 up to `i-1`, which in the worst case is `n-3` iterations for the inner loop when `i` is at its maximum. Since the inner loop is nested within the outer loop, we multiply the number of iterations of both these loops leading to `(n-2)*(n-3)` for the worst case, and since there is another constant operation inside the inner loop, it results in an overall cubic complexity.

The space complexity of the code is $O(n)$. This is because the code uses a one-dimensional list, `dp`, that stores a value for each integer from 0 to `n`. The size of this list scales linearly with the value of `n`, hence the space complexity is directly proportional to `n`.