4 Keys Keyboard

Imagine you have a special keyboard with the following keys:

`Key 1: (A)` : Prints one 'A' on screen.

`Key 2: (Ctrl-A)` : Select the whole screen.

`Key 3: (Ctrl-C)` : Copy selection to buffer.

`Key 4: (Ctrl-V)` : Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for **N** times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.

Example 1:

```
Input: N = 3
Output: 3
Explanation:
We can at most get 3 A's on screen by pressing following key sequence:
A, A, A
```

Example 2:

```
Input: N = 7
Output: 9
Explanation:
We can at most get 9 A's on screen by pressing following key sequence:
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
```

Note:

1. 1
2. Answers will be in the range of 32-bit signed integer.

Solution 1

We use `i` steps to reach `maxA(i)` then use the remaining `n - i` steps to reach `n - i - 1` copies of `maxA(i)`

For example:
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
Here we have `n = 7` and we used `i = 3` steps to reach `AAA`
Then we use the remaining `n - i = 4` steps: Ctrl A, Ctrl C, Ctrl V, Ctrl V, to reach `n - i - 1 = 3` copies of `AAA`

We either don't make copies at all, in which case the answer is just `n`, or if we want to make copies, we need to have 3 steps reserved for Ctrl A, Ctrl C, Ctrl V so `i` can be at most `n - 3`

```
    public int maxA(int n) {
        int max = n;
        for (int i = 1; i <= n - 3; i++)
            max = Math.max(max, maxA(i) * (n - i - 1));
        return max;
    }
```

Now making it a DP where `dp[i]` is the solution to sub-problem `maxA(i)`

```
    public int maxA(int n) {
        int[] dp = new int[n + 1];
        for (int i = 0; i <= n; i++) {
            dp[i] = i;
            for (int j = 1; j <= i - 3; j++)
                dp[i] = Math.max(dp[i], dp[j] * (i - j - 1));
        }
        return dp[n];
    }
```

written by yuxiangmusic original link here

Solution 2

dp[i] = max(dp[i], dp[i-j]*(j-1)) j in [3, i)

```
public int maxA(int N) {
        int[] dp = new int[N+1];
        for(int i=1;i<=N;i++){
            dp[i] = i;
            for(int j=3;j<i;j++){
                dp[i] = Math.max(dp[i], dp[i-j] * (j-1));
            }
        }
        return dp[N];
    }
```

This one is O(n), inspired by paulalexis58. We don't have to run the second loop between [3,i). Instead, we only need to recalculate the last two steps. It's interesting to observe that dp[i - 4] * 3 and dp[i - 5] * 4 always the largest number in the series. Welcome to add your mathematics proof here.

```
public int maxA(int N) {
    if (N <= 6)  return N;
    int[] dp = new int[N + 1];
    for (int i = 1; i <= 6; i++) {
      dp[i] = i;
    }
    for (int i = 7; i <= N; i++) {
      dp[i] = Math.max(dp[i - 4] * 3, dp[i - 5] * 4);
      // dp[i] = Math.max(dp[i - 4] * 3, Math.max(dp[i - 5] * 4, dp[i - 6] * 5));
    }
    return dp[N];
  }
```

written by xiyunyue2 original link here

Solution 3

We can prove that the operations can be simplified into two types:

- [1 move] Add one A.
- [k+1 moves] Multiply the number of A's by K

Say `best[k]` is the maximum number of A's that can be printed after `k` moves. The last (simplified) operation must have been addition or multiplication. Thus, `best[k] = max(best[k-1] + 1, best[k-2] * 1, best[k-3] * 2, best[k-4] * 3, ...)`.

```
def maxA(self, N):
    best = [0, 1]
    for x in xrange(2, N+1):
        cur = best[x-1] + 1
        for y in xrange(x-1):
            cur = max(cur, best[y] * (x-y-1))
        best.append(cur)
    return best[N]
```

written by awice original link here