iJRASET
International Journal For Research in
Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 7 **Issue:** VII **Month of publication:** July 2019

DOI: http://doi.org/10.22214/ijraset.2019.7217

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.177
Volume 7 Issue VII, July 2019- Available at www.ijraset.com

Special Keyboard Problem – O (1) Approach

Varun Chopra[1], Ankit Khatri[2]
[1]Accolite India
[2]National Institute of Technology, Kurukshetra

Abstract: In this research paper, we propose a faster and efficient solution to the special keyboard problem. This problem states that we have a special keyboard with four keys - Key 1 prints 'A' on the screen, Key 2 performs selection (Ctrl-A), Key 3 copies selection to the buffer (Ctrl-C) and Key 4 prints the buffer on-screen (Ctrl-V). The purpose is to print a maximum number of A's on the screen with a limited number of keystrokes N. The existing solutions use recursive solution or a dynamic programming approach to solve this problem in worst-case time complexity of $O(n^2)$. Our suggested approach recognises a pattern and this optimization to push the complexity further down to O(1), thus achieving a great improvisation over existing solutions. We have discussed both O(n) and O(1) approach.

Keywords: **Algorithms, Optimizations, Special Keyboard problem, Dynamic Programming, Recursion**

I. INTRODUCTION

Special Keyboard problem states that given a Keyboard with four keys that can perform a different operation, our goal is to print the maximum number of characters (for simplicity 'A'). More formally, the problem can be stated as below.

A. *Problem Statement*

1) *Key 1:* Prints 'A' on screen
2) *Key 2:* (Ctrl-A): Select screen
3) *Key 3:* (Ctrl-C): Copy selection to buffer
4) *Key 4:* (Ctrl-V): Print buffer on the screen

If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of A's that you can produce).

TABLE I
EXAMPLE INPUT/OUTPUT

Input (N)	Output(M)
1	1
2	2
3	3
4	4
5	5
6	6
7	9
8	12
9	16
10	20
11	27
12	36
13	48

II. EXISTING APPROACHES

Here we will discuss the existing solutions and eventually introduce efficient and optimized approach.

Let's consider the below observations before proceeding to the solution:

For N < 7, the output is N itself

Ctrl-V can be used multiple times to print current buffer

The idea is to compute the optimal string length for N keystrokes by using a simple insight. The sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's . (For N > 6)

A. Recursive Approach

In the recursive approach, the idea is that a sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's (For N > 6). So we traverse back from N-3 to 1, to find the breakpoint which results in a maximum value for a given number of keystrokes.

TABLE II RECURSIVE SOLUTION

```
def findoptimal(N):
        if N<= 6:
                return N
        maxi = 0
        for b in range(N-3, 0, -1):
                curr =(N-b-1)*findoptimal(b)
                if curr>maxi:
                        maxi = curr
        return maxi
if __name__=='__main__':
  N = int(input("Input N"))
  print("Result : " + findoptimal(N))
```

The above recursive algorithm computes same sub-problems again and again, so recomputations could be avoided by storing the solutions to sub-problems and thus solving in bottom up manner. Now an existing Dynamic programming solution is discussed.

B. Dynamic Programming Approach

This Dynamic programming approach uses Bottom up strategy to calculate result values of N, by storing subsequent values and thus avoiding computation of same sub-problems again and again.

TABLE III DYNAMIC PROGRAMMING-BOTTOM UP APPROACH

```
def findoptimal(N):
  if (N <= 6):
    return N
  screen = [0]*N
  for n in range(1, 7):
    screen[n-1] = n
  for n in range(7, N + 1):
    screen[n-1] = 0
  for b in range(n-3, 0, -1):
    curr = (n-b-1)*screen[b-1]
    if (curr > screen[n-1]):
      screen[n-1] = curr
  return screen[N-1]
if __name__ == "__main__":
  N = int(input("Input N"))
  print("Result : " + findoptimal(N))
```

Performance: The above discussed DP solution achieves an improvisation over the recursive solution and brings down the complexity to $O(n^2)$.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.177
Volume 7 Issue VII, July 2019- Available at www.ijraset.com

III. LINEAR APPROACH

We analyzed the existing DP and recursive algorithms and managed to devise a linear solution, having a complexity of O(n) and subsequently to a more optimized solution, having a complexity of O(1).

A. Linear Solution

Approach: We observed that existing solutions can be optimized further by taking into consideration that the inner loop need not traverse back till 1, to find the breakpoint. We can only consider last three values, multiply them by 2, 3, 4 respectively and take the maximum of all three to get the result.

Let's try and understand the intuition with the below scenario.

Example Scenario: Find the maximum number of characters to be printed for seven keystrokes (N = 7)

We listed the cases for keystrokes (N = 7) in *Table IV*. We see the maximum number of characters in the third row, third column. 'ACV' and 'V' operations essentially doubles the previously printed characters as explained in the fourth column.

TABLE IV
MAXIMUM CHARACTERS FOR KEYSTROKES(N=7)

Key Strokes (N)	Repeat	Ctrl-A / Ctrl-C / Ctrl-V	Math Pattern
1	1 + 6 = 7	1 + ACV + V + V + V = 5	1 * 5 = 5
2	2 + 5 = 7	2 + ACV + V + V = 8	2 * 4 = 8
3	3 + 4 = 7	3 + ACV + V = 9	3 * 3 = 9
4	4 + 3 = 7	4 + ACV = 8	4 * 2 = 8
5	5 + 2 = 7	Minimum of three steps required	
6	6 + 1 = 7	Minimum of three steps required	

We can define results formally,

result = max(result[n-3] * 2, result[n-4] * 3, result[n-5] * 4)

Note: Through more careful observation, we found that we can skip the (result[n-3] * 2) from the above calculation, as this never results in a higher value then result[n-3] * 3 and result[n-5] * 4.

TABLE V
O(N) APPROACH

```
def specialkey(N):
    if (N <= 6):
        return N
    screen = [0]*N
    for n in range(1, 7):
        screen[n] = n
    for n in range(7, N + 1):
        x = screen[n-5] * 3
        y = screen[n-6] * 4
        if(x>y):
            maxi = x
        else:
            maxi = y
        screen[n-1] = maxi
    return screen[N-1]
if __name__ == "__main__":
    N = int(input("Input N : "))
    print("Result : " + str(specialkey(N)))
```

IV. CONSTANT TIME APPROACH O(1)

We found a pattern in the result set which can be generalized to get the output in a constant time worst case complexity. Let's observe the results that produce the maximum number of A's using the four keys available, to find a pattern.

TABLE VI
RESULT VALUES AND % INCREASE

Key Strokes(N)	Result	% Increase
2	2	1.0
3	3	0.5
4	4	0.33
5	5	0.25
6	6	0.2
7	9	0.5
8	12	0.33
9	16	0.33
10	20	0.25
11	27	0.35
12	36	0.33

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.177
Volume 7 Issue VII, July 2019- Available at www.ijraset.com

13	48	0.33
14	64	0.33
15	81	0.26
16	108	0.33
17	144	0.33
18	192	0.33
19	256	0.33
20	324	0.26
21	432	0.33
22	576	0.33
23	768	0.33
24	1024	0.33
25	1296	0.26

By carefully observing the results data, we found a pattern after a breakpoint. If we multiply each result value after that breakpoint by 4, we would get the value, skipping 4. This is supported by the observations made from *Table VI* above, which shows a clear pattern in the subsequent percentage increase of result values.

We are capable of finding the result values for all N, using only a limited set of result values after a breakpoint (N=11, 12, 13, 14, 15).

TABLE VII

LIMITED SET OF VALUES TO BE STORED AS BREAKPOINT

N	Output
11	27
12	36
13	48
14	64
15	81

A. *Example*

Each result value after the breakpoint is calculated by skipping 4 values behind and multiplying the 5th value by 4. Through this, we were able to devise a formula which would eventually produce result values for any given input N no of keystrokes.

For N = 16, result = result[N = 11] * 4 = result[10^{th} index] * 2^2 = 108

For N = 21, result = result[N = 11] * 4 * 4 = result[10^{th} index] * 2^4 = 432

For N = 26, result = result[N = 11] * 4 * 4 * 4 = result[10^{th} index] * 2^6 = 1728

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.177
Volume 7 Issue VII, July 2019- Available at www.ijraset.com

Let's define the above logic programmatically,

offset: denotes index of the value from the limited set of breakpoints

exp: represents the multiplication of the offset values in powers of two

For N = 16,

offset = 10

exp = 2

result = result[10^{th} index] * 2^2 = 108

For N = 17,

offset = 11

exp = 2

result = result[11^{th} index] * 2^2 = 144

General Formula:

Given,

offset = (11 + (N - 11) % 5) - 1

exp = int((N - offset) / 5) << 1

Define Result,

Result = refer[N-1] for N <= 15

Result = refer[offset] << exp for N > 15

We conclude that using the above formula, every result value can be produced, using only a limited set of result values.

The Dynamic programming approach solves the problem in O(n^2), the current approach further pushes the complexity down to O(1). We can observe that, by storing only a limited set of values as a reference, we can calculate result values for all values of N.

TABLE VIII

THE O(1) SOLUTION

```
def findoptimal(N):
   if (N <= 6):
      return N
   refer = [1,2,3,4,5,6,9,12,16,20,27,36,48,64,81]
   if(N<=15):
      return refer[N-1]
   offset = (11 + (N - 11) % 5) - 1
   exp = int((N - offset)/5) << 1
   result = refer[offset] << exp
   return result

if __name__ == "__main__":
   N = int(input("Input N : "))
   print(findoptimal(N))
```

Explanation: In the above approach, we define a refer array which stores be pre-calculated results up to 15 keystrokes (in line with the observation that the result values of 11,12,13,14,15 keystrokes can produce all other results). If the input value N is less than or equal to 15, we would return the result from our refer array, otherwise, we would use the formula below.

Given,

offset = (11 + (N - 11) % 5) - 1

exp = int((N - offset) / 5) << 1

Define Result,

Result = refer[N-1] for N <= 15

Result = refer[offset] << exp for N > 15

Where,

N: input keystrokes

offset: denotes index of the value from the limited set of breakpoints

exp: represents the multiplication of the offset values in powers of two

Result: result of input N keystrokes

refer: the array storing pre-calculated results up to 15 keystrokes

V. FAILED ATTEMPTS

Initially, we tried observing a pattern for a GP with the result values being the elements of a GP, such that we can directly input the value of n to calculate a particular element of GP, and thus finding the result value.

VI. RESULTS

The performance comparison of the existing solution and our solution reflects a drastic improvisation. Using the techniques discussed above, we came up with a better linear solution O(n). After that, we managed to push the complexity further down to O(1) by using the understanding of the pattern in the result values.

REFERENCES

[1] Practice problem (GeeksForGeeks platform) https://practice.geeksforgeeks.org/problems/special-keyboard/0
[2] Problem Statement (GeeksForGeeks platform) https://www.geeksforgeeks.org/how-to-print-maximum-number-of-a-using-given-four-keys/