Junhao Zhang “Freddie”

USC ID:

CSCI 570 – HW3

~~07/20/2021~~

jzhang49@usc.edu

**1. Graded Problems**

* **[Question 1]**

For *T* (*n*), *f* (*n*) = and . We can observe that *f* (*n*) = for any . Thus, it is Case 1, and it has .

For *T’* (*n*), *f’* (*n*) = which is already . , and we want *a* be the greatest as possible, thus that will make as great as possible, which further makes > 2, so *T’*(*n*) will be .

Since we want , thus we need to let , which means , so *a* which is same to *a 49*. So the largest possible value of *a* is 49.

* **[Question 2]**

1. , thus , and observed that which has *k* = 1. It is Case 2; thus, answer is:

|  |
| --- |
|  |

1. Observed that , and , thus which means that for any . It is Case 1; thus, the answer is:

|  |
| --- |
|  |

1. Observed that thus . , and , so we observe that for any . It is Case 1, so the answer is:

|  |
| --- |
|  |

1. , thus . And for any . It is Case 3; thus, answer is:

|  |
| --- |
|  |

1. Use the change of variables , to get . Then denote , so now we can transform:

and from the new algorithm we have , which leads to . And observe that which has *k = 0*. Thus, it is Case 2, and answer is:

and since we used the change of variables, and observed that , thus the real answer is:

|  |
| --- |
|  |

1. Cannot be solved by Master Theorem, since and the cannot be negative.
2. Cannot be solved by Master Theorem, since cannot be negative, since we cannot perform negative number of operations in the recursion.

* **[Question 3]**

Let’s have an array of size *W* for memorization, and we call it *Value*, and we use capacity as index to save the optimal value that can be taken with the capacity. At index 0, which means when we have 0 capacity, the value is 0 since we have no capacity for anything. Then we have a loop, from capacity 1 to capacity *W*, at each iteration we calculate the optimal value that can be taken with the capacity.

Let’s denote *size* the given capacity (1 ≤ *size* ≤ *W*). By default, set *Value* [*size*] to *Value* [*size* -1], which would be the value if we take nothing. Then, with the given *size*, we iterate each item of all kinds, and if an item of a kind is within the given *size*, we calculate *Value*[*size*-] +and if this value is greater than what we have then we set it to be the new value.

Array *Value* [*W* + 1]

*Value* [0] = 0

For *size* in [1...*W*]:

*Value*[*size*] = *Value* [*size* - 1] // if don’t take anything.

For *i* in [1...n]:

If *size* >= : // if size is large enough

*Value*[*size*] = Max (*Value*[*size*], *Value*[*size*-] +)

After finished, *Value* [*W*] is the answer we need.

* **[Question 4]**

Boolean IsWord [length(*s*) + 1] [length(*s*) + 1] // value of cell of IsWord can be either:

// {True, False, Undefined}

for i in [0… length(*s*)]

IsWord[i][i] = True // Empty String returns True

Boolean CanBeSegmented (*s, startIndex, endIndex*):

// let’s assume *startIndex* can never be greater than *endIndex*

If IsWord[*startIndex*][*endIndex*] != Undefined:

Return IsWord[*startIndex*][*endIndex*]

If *s*[*startIndex* : *endIndex*] is a word in the dictionary:

IsWord[*startIndex*][*endIndex*] = True

Return IsWord[*startIndex*][*endIndex*]

IsWord[*startIndex*][*endIndex*] = False

for *i* in [*startIndex* + 1 … *endIndex*):

IsWord[*startIndex*][*endIndex*] |= CanBeSegmented(*s*, *startIndex, i*) || CanBeSegmented(*s, i, endIndex*)

Return IsWord[*startIndex*][*endIndex*]

Main():

Return CanBeSegmented(*s*, 0, length(*s*))

* **[Question 5]**

The most important thing of my algorithm is checking what should be the last balloon to burst so can earn the most coins.

**2. Practice Problems**

* **[Question 1]**

Sort