Junhao Zhang “Freddie”

USC ID:

CSCI 570 – HW3

07/21/2021

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 let , 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. First, we divide T(n) on both sides, and get:

Then, we divide on both sides, get:

Next, we let *S* (*n*) = , then we can transform the above into:

*S* (*n*) = *S* (*2n*) – 1, which is equivalent to *S* (2*n*) = *S* (*n*) + 1, and we simplify it to *S* (*n*) = + 1. If we expand it, will get:

*S* (*n*) = + 1= + 1= + 1=… = +

= +

And because *S* (*n*) = , so = + , so:

, and if we expand:

**…**

|  |
| --- |
| = |

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

* **[Question 3]**

Let *OPT* (*c*) denote the optimal solution (most value) with given capacity *c*. We are going to loop through each capacity from *1* to *W*, and find the local optimal result with the given capacity for 1 *W*. At each recursive relation, we can either take an item, or don’t take anything at all and remain same as what has, whichever is greater:

*OPT* () = max { max{ }

And the initial state = 0, since with zero capacity we cannot take anything thus the value is 0. At last, the *OPT* (*W*) will be our answer. The time complexity is *O* (*Wn*).

* **[Question 4]**

Let *OPT* (*l, r*) denote the result of whether the substring from index *l* to *r* can be segmented. And, let *k* be *l k r*, then we can see that *OPT* (*l, k*) and *OPT* (*k+1, r*) they are two subproblems. Thus, in each recurrence relation, we do:

*OPT* (*l, r*) is “True” if either substring from index *l* to *r* is a word in the dictionary, or

*OPT* (*l, r*) || (*OPT* (*l, k*) & *OPT* (*k+1, r*)) for *l k r*

And *OPT* (*l, r*) initially is “False”, and if in cases that *r < l* then such *OPT* (*l, r*) has “False” as well. At the end, *OPT* (*0, length of the string - 1*) will give us the answer. The time complexity of this algorithm is *O* ().

* **[Question 5]**

Let *OPT* (*l, r*) denote the maximum number of coins that can obtain from bursting all balloons from *l* to *r* for *l* *r*. The most important thing of my algorithm is checking what should be the last balloon to burst so can earn the most coins. Let *k* (*l k r*) be the last balloon to burst, then the maximum coins can be obtained by bursting balloons from *l* to *k – 1*, and from *k + 1* to *r* will be two separate subproblems. Therefore, we have the following recurrence relation:

*OPT* (*l, r*) =

And, if *l > k*, then *OPT* (*l, r*) = 0. The time complexity will be *O* ().

**2. Practice Problems**

* **[Question 1]**

The problem is finding whether there is a majority card in the given cards (in the form of array), and by “majority card”, we mean the kind of card that more than half of given cards that are equivalent to it, and we will keep using this definition in the rest of this answer. We are going to use divide-and-conquer to solve this question. For each recurrence relation, we divide the given array of cards into two halves, and then find the majority card of each, and combine them in the manner below:

* If both their majorities are the same, then we combine them without the need to do anything extra.
* If both don’t have majorities, we combine them as well, but the combined array still doesn’t have a majority.
* If one has majority and the other one doesn’t, then that majority will also be the majority of the combined array.
* If both have majorities but different, then the one with more occurrence in the combined array will be the majority.

We keep doing that until all subarrays are combined, then the majority of the final combined array will be our answer, or there isn’t such majority at all. The time complexity of this algorithm will be *O* (*n log n*), because we can use the Master Theorem to solve the following:

*T* (*n*) = 2 *T* () + *n*

let *T* (*n*) denote the time that the algorithm takes to execute, and *n* be the time to combine each pair of subarrays. If we use Master Theorem to solve this, we will get the result *O* (*n log n*) and that is the time complexity.

* **~~[Question 2]~~**

~~The problem is finding whether~~

* **[Question 3]**

Let *OPT* (*l, r*) denote the maximum sum of qualities of the substring from the index *l* to the index *r*, where *l* *r*. Then, let’s have a number *k* and *l k r*, then we can observe that calculating maximum sum of qualities of substring from *l* to *k* and substring from *k + 1* to *r* become two subproblems. When we need to combine them, we simply just need to add them together, but *OPT* (*l, r*) is either the maximum sum among all, or the quality of the substring from *l* to *r*, whichever is greater. Therefore, the recurrence relation is:

*OPT* (*l, r*) = max {quality (substring from *l* to *r*), }

At the end, *OPT* (0, length of the whole string – 1) will be the answer.

The time complexity is *O* ().

* **~~[Question 4]~~**

~~The problem is finding whether~~

* **[Question 5]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Minute 1** | **Minute 2** | **Minute 3** | **Minute 4** |
| **A** | 2 | 1 | 2 | 100 |
| **B** | 1 | 1 | 4 | 1 |

The right answer is 105, but the false algorithm will find 7.

We need two arrays of size *n*, each for storing the maximum values so far of machine *A* or *B* respectively at given minute *i*. Then, we loop through from minute 2 to minute *n* to calculate the optimal values on machine A and B at each of these minutes. The recurrence relations are:

For each given minute *i*, Array\_A [i] = max { Array\_A [i - 1], Array\_B [i - 2] } + and Array\_B [i] = max { Array\_B [i - 1], Array\_A [i - 2] } + . So, either came from the position *i – 1* of the same machines, or was moved from the other machine and waited for a minute, whichever is greater shall be the optimal value of given minute *i*. Please note, Array\_A [0] = 0 and Array\_B [0] = 0, Array\_A [1] = and Array\_B [1] = . And at the end, the max { Array\_A [*n*] , Array\_B [*n*] } is the answer. The time complexity is *O* (*n*).

* **~~[Question 6]~~**

~~The problem is finding whether~~