Junhao Zhang “Freddie”

USC ID: XXXX-XXXX-XXXX

CSCI 570

01/25/2016

* **[Question 1]**

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 , and = for any . Moreover:

, which is ,

which means

which means

which means

which means(and is roughly equal to).

Since there exists a value *k* such that , thus it is Case 3, therefore 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 *a* is not constant.
3. , thus . And observed that = for any *0.01 > > 0*. Furthermore, if plugin *a* and *b* to , then we have:

, then

, then

we have , which means there exists a value *k* such that . Thus, it is Case 3, so answer is:

|  |
| --- |
|  |

1. Cannot be solved by Master Theorem, since *a* is less than 1.
2. , thus . Moreover, observed that for any *> 0*. Furthermore, if plugin the values of *a* and *b* into , then we have:

, then

, then

, then

, then

, then

, then (for ).

Therefore, we have seen that when , then there exists a value *k* such that . Thus, it is Case 3, so answer is:

|  |
| --- |
|  |

* **[Question 2]**

Sort

* **[Question 3]**

# I. Define Subproblem:

Sdfsdf

# II. Recurrence Relation:

Sdsdf

# III. Pseudo Code:

Sdasdasdasd

# IV. Runtime Complexity:

Sdasdasdasd

* **[Question 4]**

# I. Define Subproblem:

Asdasd

# II. Recurrence Relation:

Sdasdasdasd

# III. Pseudo Code:

Sdasdasdasd

# IV. Runtime Complexity:

Sdasdasdasd

* **[Question 5]**

# I. Define Subproblem:

Let 1-dimensional array *PointsAt* stores the maximum points that can be accumulated when starting from the index *i* (for ). And we need another 1-dimensional array *PointsAfter*, so that *PointsAfter*[*i*]stores the maximum points among *PointsAt* at or after index *i*. In other words: *PointsAfter*[*i*] *=* Max (*PointsAt*[*j*]*,* for).

# II. Recurrence Relation:

The base case is when starting from the position *n*, *PointsAt*[*n*] *= PointsAfter*[*n*]= *a*[*n*]. The Recursive formula is:

*PointsAt*[*i*] *= a*[*i*] + {*PointsAfter*[*i + a*[*i*]] if *i + a*[*i*] *n*, otherwise 0}

Skip position *i* unless starting from there can earn more points:

*PointsAfter*[*i*] = max (*PointsAt*[*i*], *PointsAfter*[*i*+1]).

And the final result is *PointsAt*[*1*].

# III. Pseudo Code:

PointsAt[n] = a[n];

PointsAfter[n] = a[n];

for (int i = n-1; i >0; --i) {

PointsAt[i] = a[i];

if(a[i] + i <= n){

PointsAt[i] += PointsAfter[i];

}

PointsAfter[i] = Math.max(PointsAfter[i + 1], PointsAt[i]);

}

return PointsAt[1];

# IV. Runtime Complexity:

The runtime complexity is *O*(*n*). Because there are *n* subproblems and each costs constant time.