CPSC 413 June 4, 2019
University of Calgary

Homework Assignment # 2

Due: June 11, 2019, **11:59am (noon)**

LATE ASSIGNMENTS WILL NOT BE ACCEPTED

Please fill out the cover page at the end of this assignment and attach it to your solutions.

Collaboration Rules:

In order to solve the questions, you are allowed to collaborate with students from the same course, but not with anybody else. If you do collaborate with other students, you have to list them in the appropriate fields on the cover page. You have to write up the solutions *on your own* in *your own words*. You can meet with your collaborators in order to generate ideas, but you are not allowed to write up the solutions during these meetings or to take any notes, pictures, etc. away from those meetings.

You are allowed to use literature, as long as you cite that literature in a scientific way (including page numbers, URLs, etc.). In any case, your solutions must be self-contained. For example, if you use a theorem or lemma that was not covered in the lecture and that is not "well-known", then you have to provide a proof of that lemma or theorem.

If you are in doubt whether a certain form of aid is allowed, ask your instructor!

Academic misconduct (cheating, plagiarism, or any other form) is a very serious offense that will be dealt with rigorously in all cases. A single offense may lead to disciplinary probation or suspension or expulsion. The Faculty of Science follows a zero tolerance policy regarding dishonesty. Please read the sections of the University Calendar under the heading "Student Misconduct".

Note: Your submission must be well-prepared — typeset or with impeccable handwriting. Precision, conciseness and neatness count.

Question 1. Let $T : \mathbb{N} \to \mathbb{R}_{>0}$ be a function that satisfies the following properties:

$$T(1) = O(1),$$
$$T(2) = O(1), \quad \text{and}$$
$$T(n) = 3 \cdot T(\lfloor n/3 \rfloor) + O(n) \quad \text{for } n \geq 3.$$

Prove that $T = O(n \log n)$. Give a self-contained proof that does not use the "general recurrence" from class. You may use without proof known results about the sum of the first k terms of the *geometric series*.

1.

$T(1) = O(1)$

$T(2) = O(1)$

$T(n) = 3 \cdot T(\lfloor n/3 \rfloor) + O(n)$ for $n \geq 3$

Prove $T = O(n \log n)$

We assume $T(0) = 0$

we know $T1 \leq C$, $T2 \leq C$

Let : $T(1) = T(2) = C$

Show that $T(n)$ is non-decreasing $(T(n+1) \geq T(n))$

Proof:

Basecase: $T(2) = 3 \cdot T(0) + C \cdot 2 = 2C \geq C = 3T(0) + C \cdot 1 = T(1)$

I.H: Suppose $T(n) \geq T(n-1)$ for all $n \leq 2$

"Show that $T(n+1) \geq T(n)$

$$T(n+1) = 3T\left(\left\lfloor \frac{n+1}{3} \right\rfloor\right) + c\cdot(n+1) \overset{\text{by I.H}}{\geq} 3T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c\cdot n$$ since

1. $c(n+1) \geq cn$, because $n+1 > n$ &
2. $\left\lfloor \frac{n+1}{3} \right\rfloor \geq \left\lfloor \frac{n}{3} \right\rfloor$, because again $(n+1) > n$. hence the inequality holds.

Since $T(n)$ is non-decreasing, it is safe to assume that $n = 3^k$.

By replacing n.

$$T(3^k) = 3T\left(\frac{3^k}{3}\right) + c\cdot 3^k = 3T(3^{k-1}) + c3^k$$

$$T(3^{k-1}) = 3T(3^{k-2}) + c\cdot 3^{k-1}$$

$$T(3^k) = 3(3T(3^{k-2}) + c3^{k-1}) + c3^k$$

$$= 9T(3^{k-2}) + c3^k + c3^k = 9T(3^{k-2}) + 2c\cdot 3^k$$

$$T(3^{k-2}) = 3T\cdot(3^{k-3}) + c\cdot 3^{k-2}$$

$$T(3^k) = 3^2(3T(3^{k-3}) + c\cdot 3^{k-2}) + 2c\cdot 3^k$$

$$= 3^3T(3^{k-3}) + 3^2\cdot c3^{k-2} + 2c\cdot 3^k$$

$$= 3^3\cdot T(3^{k-3}) + c3^k + 2c3^k = 3^3T(3^{k-3}) + 3c3^k$$

Assume we repeat expansion k times,

$$T(3^k) = 3^k T(3^{k-k}) + k\cdot c\cdot 3^k$$

$$= 3^k T(1) + kc3^k$$

$$= 3^k\cdot c + k\cdot c\cdot 3^k$$

$$= c\cdot 3^k(k+1)$$

We know $n = 3^k$ and $k = \log_3 n$, so

$$T(3^k) = T(n) = c\cdot n(\log_3 n + 1)$$

Verify that $T(n) = c \cdot n (\log_3 n + 1)$ works

Base case:

$T(1) = c \cdot 1 \ (\log_3 1 + 1) = c \cdot 1(0+1) = c(1) = c$ ✓

Assume it works for all $n/3$

$$T(n/3) = c \cdot \frac{n}{3} \left(\log_3 \left(\frac{n}{3}\right) + 1 \right)$$
$$= c \cdot \frac{n}{3} (\log_3 n - \log_3 3 + 1)$$
$$= c \ \frac{n}{3} (\log_3 n - 1 + 1)$$
$$= c \cdot \frac{n}{3} (\log_3 n)$$

Show it works for n

$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$
$$= 3\left(\frac{cn}{3} \log_3 n\right) + cn$$
$$= cn \ \log_3 n + cn$$

therefore $T(n) = O(n \log_3 n)$

since $\log_3 n = \Theta(\log_2 n)$

$T(n) = O(n \log n)$

Question 2. Consider a sequence $A[1..n]$ of integers. An *equivalence test* is an operation that takes as input two indices $i, j\ 1, \ldots, n$ and returns "EQUAL" if $A[i] = A[j]$, and "NOT EQUAL", otherwise.

A *majority element* in A is an index i such that the value of $A[i]$ appears more than $n/2$ times in A. Using only equivalence tests, we want to find out, whether there is a majority element $A[i]$, and if yes, determine its index i.

Design a Divide and Conquer algorithm that solves this problem. Specifically, for any array $A[]$ of n integers, if there is an integer a and a set $I \subseteq \{1, \ldots, n\}$, $|I| > n/2$, s.t. $A[i] = a$ for all $i \in I$, then your algorithm should output an arbitrary index $i \in I$. If there is no such set I, the algorithm should output ∞.

Your algorithm can access the input only through equivalence tests, and it must use $o(n^2)$ of them. For full marks, your algorithm should need at most $O(n \log n)$ equivalence tests.

Describe your Divide and Conquer algorithm, argue why it is correct and analyze its running time. For your analysis, you are allowed to use without proof the result of the "general recurrence" from class (Section 5.3). Describe in detail which data structures you are using to achieve the claimed running time. Provide pseudo-code for your algorithm in addition to a high level plain text explanation.

Hint: Consider the following, simpler *majority verification problem*. The input is an array $A[]$ and an index i, and the output is "yes", if $A[i]$ is a majority element, and "no" if it is not. Design a simple algorithm that solves the majority verification problem using $O(n)$ equivalence tests. Use this algorithm as a sub-routine in your Divide and Conquer algorithm for the majority element problem, to determine which of the solutions for the sub-problems is also a solution for the original problem.

Question 2:

- Divide & conquer algorithm:

The algorithm begins by splitting the array in half repeatedly and calling itself on each half. This is similar to what is done in merge sort. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return value from its two recursive calls

There are 4 scenarios:

1. Both return "no majority". Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore the call returns "no majority"

2. The right side is a majority and the left isn't. The right side is a majority, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is majority element the return that element, else return "no majority"

3. The left side has a majority, but the right hasn't. Same as (2)

4. Both sub-calls return a majority element. Count the number of equal elements that are candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return "no majority"

In the end we compare the two final arrays of size n/2 and return either a majority element or no majority

ALGORITHM:

```
Function Maj_Element(A[]){
    if |A| ==1
        return 0
    else
        Split A[] into two lists:
        a1[] = A[0...n/2]
        a2[] = A[(n/2)+1...n]
        left_mayority=Maj_Element(a1)
        right_mayority=Maj_Element(a2)

        if (left_mayority != ∞ and right_mayority != ∞){
            if checkMajority(A,left_majority)
                return left_majority
            else if checkMajority(A, right_majority)
                return right_majority
            else
                return ∞
        }
        else if left_majority != ∞
            if checkMajority(A, left_majority){
                return left_majority
            }
        else if right_majority != ∞
            if checkMajority(A, right_majority){
                return right_majority
            }
        else{
            return ∞
        }
}
```

****HINTED ALGORITHM**

```
Function checkMajority(A[], index){
        counter=0
        n=A.size
        k=0
       ( for every k <= n)
                    if(A[k] == A[index])){
                            count ++
                    }
        if counter > (n/2){
                     return true
        }
        else{
              return false
        }
}
```

-We can see that at each level two recursive calls are made, with each call having an array of size $\frac{n}{2}$ (half of the original array). Additionally, regarding the non-recursive cost, we can see that at each level we have to compare each number at most twice. Therefore the non-recursive cost is at most $2n$ comparisons when the array has a size of n.

Therefore

$T(n) = 2T(\frac{n}{2}) + 2n$

By the Master Theorem case 2 we can determine that

$T(n) \in \Theta(n \log n)$

thus $T(n) \in O(n \log n)$ as well.

Question 3.

The input for the Stone Grouping Problem is a positive integer n and an array $val[]$ of length n that assigns each number i of stones a value $val[i]$ such that $val[i+1] \geq val[i]$, for $i \in \{1, \ldots, n-1\}$. A solution for this problem is a sequence $s_1, \ldots, s_k$ of positive integers, such that $s_1 + \cdot \cdot \cdot + s_k = n$. The value of this solution is $val[s_1] + \cdot \cdot \cdot + val[s_k]$. An optimal solution is one of maximal value.

(a) For the input below, give an optimal solution and its value.

$n = 7$;

number of stones	1	2	3	4	5	6	7
value	3	6	8	9	10	17	18

For the previous input, there are two optimal solutions,

One optimal solution for it is:
Sequence: {1,1,1,1,1,1,1} with a maximum value of 21

(b) Give a Bellman Equation that describes the value of the optimal solution for any input to the Stone Grouping Problem. Briefly explain why the Bellman Equation is correct.

Bellman equation for the Stone grouping problem:

$$OPT(i) \begin{cases} OPT[0] = 0 & \text{for } i=0 \\ OPT[i] = \max(OPT[i-1], \{OPT[i-(k+1)]+v(i)\ ) & \text{otherwise} \end{cases}$$

Base Case:
i=0 if the array is empty then the maximum grouping is going to be 0 by convention

Other Cases:
i>=1

In this case we are saying that for any array i, the optimal solution is either the last calculated (OPT[i-1] solution or the maximum of any other optimal solution calculated (OPT[i-(k+1)) and to that since we had a lower limit for those cases we can always pick at least 1 more stone (...+val(i)). Since we are comparing all possible solutions, therefore we should get the optimal solution at the end.

(c) Give a Dynamic Programming algorithm in pseudo-code that computes the *value* of an optimal solution to the Stone Grouping Problem. Your algorithm must be based on the Bellman Equation you designed in Part (b), and have polynomial running time. For full marks,its running time should be $O(n^2)$. State the running time of your algorithm and briefly explain why your statement is correct.

```
Max_StoneGroup_val( values[], n) {

 OPT[n+1];

 OPT[0] = 0;

 for(j=1;j<=n;j++) {

    max = OPT[j-1];

    for(i=0;i<n;i++) {

        x = j-(i+1);

       if(x >= 0 && (OPT[x] + values[i]) > max) {

          max = OPT[x] + values[i];

       }

       OPT[j] = max;

    }

 }

 return OPT[n];

}
```

Analysis of running time:

The algorithm will calculate the max value of a stone group of 0 which takes O(1) time afterward for each following array size until n [O(n)] will be calculated by comparing all possible combinations found before the current step [O(n)]. Since these two operations are nested together in for loops, this entire section has a running time of worst running case n^2 . **[O(n^2)]**

(d) Give an algorithm in pseudo-code that computes the optimal *solution* to the Stone Grouping Problem. Your algorithm can use as a sub-routine your algorithm from Part (c). It must have polynomial running time, and for full marks its running time should be $O(n^2)$. State the running time of your algorithm and briefly explain why your statement is correct.

For part d) we will call algorithm describe in part c) but we also will make some changes.

```
Max_StoneGroup_val( values[], n, items[]) {

 OPT[n+1];

 OPT[0] = 0;

 items[0] = -1; //we store the value of the items used to get the max_value



 for(j=1;j<=n;j++) {

    items[j] = items[j-1];

    max = OPT[j-1];

    for(i=0;i<n;i++) {

        x = j-(i+1);

       if(x >= 0 && (OPT[x] + values[i]) > max) {

          max = OPT[x] + values[i];

          items[j] = i;

       }

       OPT[j] = max;

    }

 }

 return items[];

}
```

***Asume Max_StoneGroup_val() was called on main and returns the value items[] we send to this function

```
function Optimal_items_list(n, items[]){
 k = n;
OPT_list[];
  instances[n];

  for(i=0;i<n;i++)
     instances[i] = 0;
  while(k >= 0) {
     x = items[k];
     if(x == -1) STOP;
     instances[x] += 1;
     k -= items[k]+1; //calculates new limit of stones after having picked a value
  }
  for(i=0;i<n;i++)
     for(int j=0;j<instances[i];j++){
       ADD i+1 to OPT_list[]
     }
return OPT_list[];
}
```

Analysis of running time:

First we first run Max_StoneGroup_val just like part c), since the only thing we add is saving a value inside an array (O(1)) the execution of this routine give us again a worst case running time of n^2 as described before.

Then we execute the routine optimal_items_list which will run for every element in the item array (up to n), having choosing an item it defines what the next element to be added to the list is by choosing the optimal for an array with a new size k. hence we will iterate again with a new array with size < n. Until we get an array of size 0. Therefore this concatenated for loop runs at a worst running case O(n^2).

And finally we will print as many elements of each group of rocks as we have. O(n)

Finally we will have a total running time of 2n^2 + n which is O(n^2)

Cover Page for CPSC 413
Homework Assignment # 2

Name: **Juan Luis de Reiset Jimenez-Carbo**

Course Section: CPSC413-Tut 02

Collaborators:

Question 1: Daniel Sohn, Steve Khanna, Coskun Sahin

Question 2: Daniel Sohn, Steve Khanna, Coskun Sahin

Question 3: Daniel Sohn, Steve Khanna, Coskun Sahin

Question 4: Daniel Sohn, Steve Khanna, Coskun Sahin

Other Sources:

Question 1:

Question 2: https://www.geeksforgeeks.org/majority-element/ http://www.utdallas.edu/~hal/CS6363/Hw1.pdf

Question 3:
https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/06DynamicProgrammingI.pdf?fbclid=IwAR2lBC4NHHo5WpWt4Frx1LTDS-qv-jpYfRwP5i5TBCiF_lsQ1oUCWbsnhU0

Question 4:

Declaration:

I have written this assignment myself. I have not copied or used the notes of any other student.

Date/Signature:

<u>6/11/2019</u>

<u>Juan Luis de Reiset Jimenez-carbo</u>