CPSC 413 — Design and Analysis Algorithms I
ASSIGNMENT 2

Name: Haohu Shen
UCID: 30063099

Notice: I have read and followed the guide-lines for academic conduct in CPSC 413 Spring 2020. As part of those rules, when collaborating with anyone, (1) I took no record but names away, and (2)after a suitable break, I created the assignment I am submitting without help from anyone other than the course staff.

1. **Safe by Committee [6 marks]**

   1.1. **Part 1 [2 marks of 6]**

   Let $S_1, \dots, S_k$ be the set of shifts returned by our algorithm. Assume without loss of generality that $S_1, \dots, S_k$ is sorted by finishing times. We will denote the starting and finishing times of shift S_j by $s(S_j)$ and $f(S_j)$ respectively. We first prove that $S_1, \dots, S_k$ is a valid safety committee, by proving that:

   Fact 1. *If S is a shift such that $s(S) \leq f(S_j)$, then S is overlapped by one of $S_1, \dots, S_j$.*

   Proof: The proof is by induction on j.

   Base Case: For $j = 1$, observe that the shift $S_j = S_1$ has overlapped a finishing time p such that p is the earliest among all shifts since all shifts are already sorted by finishing times. Thus, for all other shifts S_m such that $1 < m \leq k$ and $s(S_m) \leq f(S_1)$, it must be the case that $p \leq f(S_m) \leq f(S_1)$, therefore S_m overlaps S_1.

   Inductive Hypothesis: Suppose now that every shift S whose starting point is $s(S) \leq f(S_j)$ is overlapped by $S_1, \dots, S_j$.

   Consider a shift S such that $s(S) \leq f(S_{j+1})$.

   - If $s(S) \leq f(S_j)$ then by Inductive Hypothesis, we can have that S overlaps one of $S_1, \dots, S_j$.
   - Otherwise, S is a shift $f(S_j) < s(S) \leq f(S_{j+1})$ such that $S_1, \dots, S_j$ does not overlap it. We need to show that S_{j+1} overlaps S to complete our proof. Suppose S_n is the shift such that it has the earliest finishing time comparing to all shifts whose starting time is later than $f(S_j)$, then we can choose the shift S_{j+1} from all shifts that overlap $f(S_n)$ such that it has the latest finishing time, thus we can have that $f(S_n) \leq f(S)$, which indicates that S_{j+1} overlaps S.

   We have shown though induction that if S is a shift such that $s(S) \leq f(S_j)$, then S overlaps one of $S_1, \dots, S_j$.

   End Proof

   1.2. **Part 2 [2 marks of 6]**

   Let $T_1, \dots, T_m$ be the shifts in a smallest safety committee, sorted by increasing finishing time. Our algorithm returns a complete safety committee $k \geq m$, as we could have the smallest solution or it is bigger.

   Next we will establish that

Fact 2. *For $j = 1, \ldots, m$, $f(T_j) \leq f(S_j)$.*

Proof: The proof is by induction on j.

Base Case: This is clear for $j = 1$, because the shift that has the earliest finishing time must be overlapped by at least 1 other shifts in the safety committee, and the algorithm will greedily filter one of such shifts with the latest finishing time.

Inductive Hypothesis: Suppose now that the statement is true for S_j and T_j.

Consider S_{j+1} and T_{j+1}. Let S be the shift with the earliest finishing time amongst those that start after $f(S_j)$.

From the algorithm we can see that all shifts whose finishing time is earlier or equals to $f(S_j)$ are marked as covered. Meanwhile, by the Inductive Hypothesis, we know that any shift whose starting time is later than $f(S_j)$ is not able to overlapped by T_j and the algorithm will greedily filter one of shifts with the latest finishing time as S_{j+1} from all shifts that overlap S. Thus T_{j+1} also does not overlap S.

Hence $f(T_{j+1}) \leq f(S_{j+1})$.

End Proof

1.3. **Part 3 [2 marks of 6]**

In order to prove that $k \leq m$ we only need to prove that every shift overlaps an element of $S_1, \ldots, S_m$. Proof by contradiction: Indeed, if there were a shift S that does not overlap any of them, then we will have $s(S) > f(S_m)$.

This is clearly impossible, since in that case none of $T_1, \ldots, T_m$ would overlap with S. By contradiction then there is no shift $s(S) > f(S_m)$ that isn't overlapped which means $k \leq m$. Combined with earlier we can now say $k == m$, AKA our algorithm finds a minimal size committee.

2. **Recurrently [6 marks]**

2.1. **Sort Of [3 marks of 6]**

Consider the following sorting algorithm:

```
define meh_sort(A, first, last):
  if A[first] < A[last]:
      exchange A[first] and A[last]

  if (first + 1 < last):
    mid = (last - first + 1) // 3     # integer division
    meh_sort(A, first + mid, last)  # sort last two-thirds
    meh_sort(A, first, last - mid)  # sort first two-thirds
    meh_sort(A, first + mid, last)  # sort last two-thirds again
```

Write a recurrence relation that describes the worst-case running time of function `meh_sort` in terms of n, where $n = \texttt{last} - \texttt{first} + 1$. You can ignore floors and ceilings.

Solution: The recurrence relation of the worst-case running time $T(n)$ can be described as below:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 2. \\ \Theta(1) + 3T(\frac{2n}{3}) & \text{if } n > 2. \end{cases} \tag{1}$$

2.2. **Not At All [3 marks of 6]**

Consider now this even less useful algorithm:

```
define not_useful(A, first, last):
  if last < first + 4:
      x = A[last] - A[first]
  else:
    x = not_useful(A, first+1, last-1) - not_useful(A, first+2, last-2)
  return x
```

Write a recurrence relation that describes the worst-case running time of function `not_useful` in terms of n, where $n = \texttt{last} - \texttt{first} + 1$.

Solution: The recurrence relation of the worst-case running time $T(n)$ can be described as below:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < 5. \\ \Theta(1) + T(n-2) + T(n-4) & \text{if } n \geq 5. \end{cases} \tag{2}$$

3. **Test Test Testing [6 marks]**

Consider a sequence $A[1..n]$ of integers. An *equivalence test* is an operation that takes as input two indices $i, j \in \{1, \ldots, n\}$ 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. Using only equivalence tests means you cannot sort the input as you can't do a comparison or order, just equivalence. You can access the size of input of course.

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.

Solution: Since if we know a sequence $A[1 \cdots n]$ contains a majority element p, p is also the majority element of one of its halves or both, on the other hand, when we know the majority element in the left and right halves of the sequence, we can determine the global majority element of the sequence from one of them by counting the occurrences of that element in the sequence being cut using $O(n)$ equivalence tests and compare the result of counting with $m/2$ where $m \leq n$ and m is the length of the sequence being cut.

Thus, we can design a Divide and Conquer algorithm that takes an array $A[]$ of n integers as its input and if $\exists a \in \mathbb{Z}$ and a set $I \subseteq \{1, \cdots, n\}, |I| > n/2$ such that $A[i] = a$ for all $i \in I$, outputs an arbitraily i, otherwise outputs ∞. Here we define $A.size$ as the size of the array $A[]$ and it can be accessed in constant time.

The algorithm will recursively cut the sequence into left half sequence and right half sequence until the size of the sequence being cut becomes 1 since the majority element of a length-1 sequence can be ontained trivially.

Thus, if the slice of sequence being cut has length of 1, then its majority element is the only element it contains and we can directly output its index. Otherwise, we have to combine the majority elements of the slice's left and right halves through equivalence tests, in order to achieve that, we have to split the case into 4 subcases as below:

- If the index of the majority element of the left half sequence is -1 and the index of the majority element of the right half sequence is -1, that is, the majority elements in both halves of the sequence cannot be found, then we can conclude that the sequence being cut has no majority elements as well.
- If the index of the majority element of the left half sequence is not -1 and the index of the majority element of the right half sequence is -1, that is, the majority element of the left half sequence is found but the majority element of the right half sequence cannot be found, then we need to check that the majority element on the left half sequence is also the majority element of the sequence being cut by equivalence tests.
- If the index of the majority element of the left half sequence is -1 and the index of the majority element of the right half sequence is not -1, that is, the majority element of the left half sequence cannot be found but the majority element of the right half sequence is found, then we need to check that the majority element on the right half sequence is also the majority element of the sequence being cut by equivalence tests.
- If the index of the majority element of the two halves of the sequence are not -1, that is, the majority elements in both halves of the sequence are found, suppose the majority elements are p_{left} and p_{right}, then we respective check if p_{left} or p_{right} is the majority element of the sequence being cut by equivalence tests, since it is impossible that it both p_{left} and p_{right} are the global majority element since each of them will occur more than half size of the sequence being cut, the majority element must be p_{left} or p_{right}, otherwise there is no global majority elements.

  Base on the approach described above, we can split the algorithm into an inner algorithm and an outer algorithm, the inner algorithm takes a non-empty sequence $A[]$, a number p, two indices l, r of $A[]$ such that $1 \leq l \leq r \leq A.size$ as inputs and outputs the number of occurrences of p in $A[l, \cdots, r]$, we name it CountNumberInRange and give its pseudocode with annotation as below:

```
CountNumberInRange($A, p, l, r$)
1  $counter = 0$    // Set a counter to count the occurrences of $p$ and initialize with 0
2  for $i = l$ to $r$        // Run $r - l + 1$ iterations
3      if $A[i] == p$    // The branch will be executed $r - l + 1$ times at the worst case
4          $counter = counter + 1$   // Cost 2 operations on addition and assignment
5  return $counter$      // Cost 1 operation to return
```

  Suppose that $A.size = n$ such that n is a positive integer, then the worst case of this algorithm is that the range covers from 1 to n and p is the only element that appears in the range, that is, $l = 1, r = n$ and $A[i] = p$ for all i such that $1 \leq i \leq n$. Thus, if its running time of the worst case is $T(n)$, then

$$\begin{aligned} T(n) &= 1 + (n - 1 + 1) + 3(n - 1 + 1) + 1 \\ &= 1 + n + 3n + 1 \\ &= 4n + 2 \end{aligned}$$

  Let $g(n) = n$, we prove $T(n) = \Theta(n)$ by Limit Test for Θ on $g(n)$, since:

$$\lim_{n\to\infty} T(n) = \lim_{n\to\infty} (4n+2) = \infty$$

And

$$\lim_{n\to\infty} g(n) = \lim_{n\to\infty} (n) = \infty$$

we can apply *L'Hôpital's Rule* as below:

$$\begin{aligned}\lim_{n\to\infty} \frac{T(n)}{g(n)} &= \lim_{n\to\infty} \frac{4n+2}{n} \\ &= \lim_{n\to\infty} (4 + \frac{2}{n}) \\ &= \lim_{n\to\infty} 4 + \lim_{n\to\infty} (\frac{2}{n}) \\ &= 4\end{aligned}$$

Since 4 is a constant such that $4 > 0$, we can have that $T(n) = \Theta(g(n)) = \Theta(n)$.

Now we describe the outer algorithm, it contains the main logic of Divide and Conquer, it takes a non-empty sequence $A[]$, two indices l, r of $A[]$ such that $1 \leq l \leq r \leq A.size$ as inputs. If the majority element in $A[l \cdots r]$ exists, output an arbitrary i such that $A[i]$ is the majority element in $A[l \cdots r]$, otherwise output ∞. We name this algorithm IndexOfMajorityElement and give its pseudocode with annotation as below:

```
IndexOfMajorityElement($A, l, r$)
// Case 1: If the size of the test range $A[l \cdots r]$ is only 1, the majority element is
// the only element it contains since its number of the occurrences is $1 > 1/2 = A.size/2$
// and we can return its index directly and stop the recursion
if $l == r$
    return $l$
// Otherwise, we recursively cut the slice of $A[l \cdots r]$ in halves and obtain the indices
// of majority elements in both halves

$m = \lfloor (l + r)/2 \rfloor$     // Initialize a variable that stores the middle index of $l$ and $r$

// Initialize a variable that stores the index of the majority element in the left
// half slice of $A[l \cdots r]$
$l\_i$ = IndexOfMajorityElement($A, l, m$)

// Initialize a variable that stores the index of the majority element in the right
// half slice of $A[l \cdots r]$
$r\_i$ = IndexOfMajorityElement($A, m + 1, r$)

// Case 2: If the majority elements are not found in both halves of the slice $A[l \cdots r]$,
// then it indicates that the slice contains no majority elements, thus we will return $\infty$
if $l\_i == \infty$ and $r\_i == \infty$
    return $\infty$

// Case 3: If the majority element is found on the left half slice of $A[l \cdots r]$ but not found
// on the right half slice of $A[l \cdots r]$, then we need to check that the majority element on
// the left half slice is also the majority element of the whole slice
elseif $l\_i \neq \infty$ and $r\_i == \infty$
    // Obtain the majority element in the left half
    $p = A[l\_i]$
    // Obtain the size of the slice
    $length = r - l + 1$
    // Obtain the number of occurrences of the majority element of the left slice on the
    // whole slice
    $counter$ = CountNumberInRange($A, p, l, r$)

    // If the number of occurrences is larger than half of the length of the
    // slice, stop the recursion by returning the index. Otherwise, stop the recursion by
    // returning $\infty$
    if $counter > length/2$
        return $l\_i$
    else
        return $\infty$
```

```
// Case 4: If the majority element cannot be found
// on the left half slice but is found on the right half slice of $A[l \cdots r]$, then we need
// to check that the majority element on the right half slice is also the majority
// element of the whole slice
elseif $l\_i == \infty$ and $r\_i \neq \infty$
    // Obtain the majority element in the right half
    $p = A[r\_i]$
    // Obtain the size of the slice
    $length = r - l + 1$
    // Obtain the number of occurrences of the majority element of the right slice on the
    // whole slice
    $counter = \textsc{CountNumberInRange}(A, p, l, r)$

    // If the number of occurrences is larger than half of the length of the
    // slice, stop the recursion by returning the index. Otherwise, stop the recursion by
    // returning $\infty$
    if $counter > length/2$
        return $r\_i$
    else
        return $\infty$

// Case 5: If the majority elements are respectively found on the left half and on the
// right half of the slice of $A[l \cdots r]$, we need to compute and compare that the number of
// occurrences of the majority element on the left half slice and the number of occurrence of
// the majority element on the right half slice in the original slice.
else
    // Obtain the majority element in the left half of the slice
    $p = A[l\_i]$
    // Obtain the majority element in the right half of the slice
    $q = A[r\_i]$
    // Obtain the size of the slice
    $length = r - l + 1$
    // Obtain the number of occurrences of the majority element of the left slice on the
    // whole slice
    $counter\_l = \textsc{CountNumberInRange}(A, p, l, r)$
    // Obtain the number of occurrences of the majority element of the right slice on the
    // whole slice
    $counter\_r = \textsc{CountNumberInRange}(A, q, l, r)$

    // Sub-case 1: $p$ occurs more than $length/2$ and $q$ occurs less than $length/2$
    if $p > length/2$
        return $l\_i$
    // Sub-case 2: $p$ occurs less than $length/2$ and $q$ occurs more than $length/2$
    if $q < length/2$
        return $r\_i$
    // Sub-case 3: $p$ occurs the same as $q$ in $A[l \cdots r]$
    return $\infty$
```

Thus, in order to analyze the running time, from the algorithm we can see that

- If $l = r$, then the 4th line will be executed and we have $\textsc{IndexOfMajorityElement}(A, l, r) = l$, which takes $\Theta(1)$ steps.
- If $l < r$, let $m = \lfloor (l + r)/2 \rfloor$, then in the 13th $\textsc{IndexOfMajorityElement}(A, l, m)$ and $\textsc{IndexOfMajorityElement}(A, m, r)$ will be recursively called in the 13th and the 17th lines. The return values from these two calls will be compared to ∞ and processed into 4 cases, which corresponds to Case 2 to Case 4 in our pseudocode.
  * In Case 2: Since ∞ will be returned directly, the case takes $\Theta(1)$ steps.
  * In Case 3: Except the function call in the 34th line takes $\Theta(r - l + 1)$ steps, other lines take $\Theta(1)$ steps totally, thus the case takes $\Theta(r - l + 1)$ steps.
  * In Case 4: Except the function call in the 54th line takes $\Theta(r - l + 1)$ steps, other lines take $\Theta(1)$ steps totally, thus the case takes $\Theta(r - l + 1)$ steps.
  * In Case 5: Except the function call in the 77th line takes $\Theta(r - l + 1)$ steps and the function call in the 80th line takes $\Theta(r - l + 1)$ steps, other lines take $\Theta(1)$ steps totally, thus the case takes $\Theta(r - l + 1)$ steps.

Since from the problem all information of the input we are given is only an array $A[]$ whose size is n, since we need to find the index of the majority element in the whole array, we will call $\textsc{IndexOfMajorityElement}(A, 1, n)$.

The algorithm makes two recursive calls on the slice of $A[1, \cdots \lfloor n/2 \rfloor]$ and the slice of $A[\lceil n/2 \rceil, \cdots, n]$ respectively, and in addition performs $\Theta(n)$ steps after that. Hence the running time $T(n)$ of the whole algorithm satisfies

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases} \tag{3}$$

Thus, let $a = b = 2 > 1$, when $n > 1$,

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) \\ &\leq T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + \Theta(n) \\ &= 2T(\lceil n/2 \rceil) + \Theta(n) \\ &= aT(\lceil n/b \rceil) + \Theta(n) \end{aligned}$$

Thus, by *Master Theorem*, we can conclude that the running time of the algorithm is $T(n) = \Theta(n \log n)$.

4. **Got Stones [6 marks]**

Recently, we've researched a new currency, based on grouping various numbers of stones together. Each number k of stones is assigned a value $val(k)$, for some non-decreasing function val. The researchers realized, that by splitting a fixed number of stones into smaller groups, one can obtain different sums of values for all groups combined. For example, consider the value-function in the table below. If we split 8 stones into three groups of 1, 3, and 4 stones, respectively, then the total value is $val(1) + val(3) + val(4) = 1 + 8 + 9 = 18$. On the other hand, splitting the 8 stones into two groups of 2 and 6 stones, respectively, yields a total value of $val(2) + val(6) = 5 + 17 = 22$.

number of stones	1	2	3	4	5	6	7	8
value	1	5	8	9	10	17	17	20

The researchers are interested in finding an efficient method of grouping a given number of stones, so that the total value of all groups combined is as large as possible. Can you help them?

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 $i \in \{1, \ldots, n\}$.

A solution for this problem is a sequence $s_1, \ldots, s_k$ of positive integers, such that $s_1+\cdots+s_k = n$. The value of this solution is $val[s_1] + \cdots + val[s_k]$. An optimal solution is one of maximal value.

4.1. **Riddles me Stones [1 marks of 6]**

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

$n = 7$;

i	1	2	3	4	5	6	7
$val[i]$	3	6	8	9	10	17	18

Solution: One optimal solution for $n = 7$ is $(1, 1, 1, 1, 1, 1, 1)$ since $1+1+1+1+1+1+1 = 7$, its value is $val[1]+val[1]+val[1]+val[1]+val[1]+val[1]+val[1] = 3+3+3+3+3+3+3 = 21$.

4.2. **The Bell Stone [1 marks of 6]**

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.

Solution: According to the description of the question, we define $\mathcal{O}_n$ to be an optimal solution for input n and an array $val[]$ of length n, we also define OPT_n to be the value of $\mathcal{O}_n$. Then we have 3 cases as below:

- if $n = 0$, it is impossible since n must be a positive integer given by the problem's description, thus $\mathcal{O}_n = \mathcal{O}_0 = \emptyset$ and $OPT_n = OPT_0 = 0$.
- if $n = 1$, that is, there is only 1 stone, then it cannot be split into smaller groups anymore, thus total value is also its value, which is $val[1]$.
- if $n > 1$, then we have 2 subcases:
  Subcase 1: $n \in \mathcal{O}_n$. Then n is the only element in the sequence of the solution since the summation of the sequence must equal to n. Hence, $\mathcal{O}_n = \{n\}$, $OPT_n = val[n]$.

Subcase 2: $n \notin \mathcal{O}_n$. Then any element from $1, 2, \cdots, n-1$ could be part of an optimal solution, so if $\mathcal{O}_i$ is a part of its optimal solution such that $1 \le i \le n-1$, then $n-i$ is the only element we can add to O_i to obtain O_n, and the value of O_n would be $OPT_i + val[n-i]$, moreover, in order to obtain the maximal value, we must maximize $OPT_i + val[n-i]$, thus $OPT_n = \max\{val[i] + OPT_{n-i} \mid 1 \le i \le n-1\}$.

Thus, we can combine two subcases and have:

$$OPT_n = \max\{val[i] + OPT_{n-i} \mid 1 \le i \le n\}$$

Therefore, we can combine all three cases above and have the following Bellman Equation:

$$OPT_n = \begin{cases} 0 & \text{if } n = 0 \\ \max\{val[i] + OPT_{n-i} \mid 1 \le i \le n\} & \text{if } n > 0 \end{cases} \tag{4}$$

4.3. **42 Stones [2 marks of 6]**

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.

Solution: According to the Bellman Equation above, we can give a Dynamic Programming algorithm in pseudo-code with annotation below, the algorithm takes a positive integer n and an array $val[]$ of length n such that $val[i+1] \ge val[i]$ for all $1 \le i \le n-1$, we name it ValueOfStoneGrouping, notice that the function $\max(a, b)$ cost $O(1)$ steps since it only takes 2 numbers for comparison and returns the maximal one:

```
ValueOfStoneGrouping($val, n$)
 1  Let $M[0, \cdots, n]$ be an array of length $n+1$, assume it takes $O(n)$ steps
 2  // Initialize $M[0]$ with 0, which takes $O(n)$ steps
 3  $M[0] = 0$
 4
 5  // Outer loop runs $n$ iterations
 6  for $k = 1$ to $n$
 7      // Initialize a variable with $-\infty$ to keep track of the current maximal value
 8      $p = -\infty$
 9      // Inner loop runs $k$ iterations
10      for $i = 1$ to $k$
11          // Cost $O(1)$ operation to compare and update $p$
12          $p = \max(p, val[i] + M[k-i])$
13      // Update $M[k]$ with $p$
14      $M[k] = p$
15  return $M[n]$  // Cost $O(1)$ operation to return
```

Thus, the algorithm takes $O(n)$ steps to run from the 1st and the 3rd line and takes $O(n^2)$ steps to run the doubly-nested loop structure from the 9th to the 13th lines since the inner loop forms an arithmetic series from line 10th to line 12th, moreover, the algorithm takes $O(1)$ steps on other trivial operations. Therefore, we can conclude that the running time of the algorithm $T(n) = O(n^2)$.

4.4. **Stone Tablets [2 marks of 6]**

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.

Solution: In order to re-construct the optimal solution using the maximal value obtained from the previous problem, we add an array $S[1, \cdots, n]$ whose length is n in our algorithm in the previous algorithm and $S[i]$ represents the first number in the optimal solution of the sub-problem where the number of stones is i such that $1 \leq i \leq n$. After that, we can obtain each number in the sequence of the optimal solution by reducing n with $S[n]$ until $n \leq 0$ since $S[n]$ is included in n. We assume that it takes $O(1)$ steps to append an element to the back of the array, and we name it the algorithm SolutionOfStoneGrouping and give the pseudo-code below, it takes a positive integer n and an array $val[]$ of length n as input such that $val[i+1] \geq val[i]$ for all $1 \leq i \leq n-1$ and outputs a sequence $s_1, \cdots, s_k$ of integers such that $s_1 + \cdots + s_k = n$ and the value $val[s_1] + \cdots + val[s_k]$ is maximal.

```
SolutionOfStoneGrouping($val, n$)
 1  Let $R$ be an empty array that stores the sequence of the optimal solution,
 2  assume it takes $O(1)$ steps to initialize
 3
 4  Let $S[1, \cdots, n]$ be an array of length $n$, assume it takes $O(n)$ steps
 5  // Initialize $S[1, \cdots, n]$ with 0, which takes $O(n)$ steps
 6  for $k = 1$ to $n$
 7      $S[k] = 0$
 8
 9  Let $M[0, \cdots, n]$ be an array of length $n + 1$, assume it takes $O(n)$ steps
10  // Initialize $M[0]$ with 0, which takes $O(1)$ steps
11  $M[0] = 0$
12
13  // Outer loop runs $n$ iterations
14  for $k = 1$ to $n$
15      // Initialize a variable with $-\infty$ to keep track of the current maximal value
16      $p = -\infty$
17      // Inner loop runs $k$ iterations
18      for $i = 1$ to $k$
19          // Cost $O(1)$ operation to compare and update $p$ and $S[k]$
20          if $val[i] + M[k - i] > p$
21              $p = val[i] + M[k - i]$
22              $S[k] = i$
23      // Update $M[k]$ with $p$
24      $M[k] = p$
25
26  // Start the re-construction on the sequence by reducing $S[n]$
27  // Since $n$ will at least be reduced by 1 after each loop, the WHILE loop
28  // takes $O(n)$ steps to run
29  while $n > 0$
30      append $s[n]$ to $R$    // Cost $O(1)$ to append
31      $n = n - s[n]$    // Update $n$
32
33  return $R$        // Cost $O(1)$ operation to return
```

Thus, the algorithm takes $O(n)$ steps to run from the 4th and the 9th line and takes $O(n^2)$ steps to run the doubly-nested loop structure from the 14th to the 24th lines since the inner loop forms an arithmetic series from line 18th to line 22th. It also takes $O(n)$ to execute the loop from the 29th line to the 31st line. Moreover, the algorithm also takes $O(1)$ steps on other trivial operations. Therefore, we can conclude that the running time of the algorithm $T(n) = O(n^2)$.