16 (100 PTS.) Piazza.

Recently the ECE department discussed Piazza engagement. Haitham believes the student engagement in CS/ECE 374B is highly skewed, despite the many questions being asked. He claims that the top 5% of student askers (who ask many questions) contribute more than ten times the total number of questions asked by the bottom 90% of student askers (who ask little to no questions). To verify this claim, Andrew downloaded the Piazza statistics which give him an n element *unsorted* array A, where $A[i]$ is the number of questions asked by student i.

16.A. (20 PTS.) Describe an $O(n)$-time algorithm that given A checks whether the top 5% contribute more than ten times the questions asked by the bottom 90% together. Assume for simplicity that n is a multiple of 100 and that all numbers in A are distinct. Note that sorting A will easily solve the problem but will take $\Omega(n \log n)$ time.

Solution:

The general idea is to use **Select**(k, A) algorithm to quickly find the top 5% and the bottom 90%. Then, iterate over A to find the sum of the number of questions asked by the top 5% and bottom 90%.

Algorithm:

```
VerifyClaim(A)
  b ← Select((0.90)n, A)
  t ← Select((0.95)n + 1, A)
  S_90% ← 0
  S_5% ← 0
  for i = 0, ..., n − 1 do
      if A[i] ≤ A[b]
          S_90% ← S_90% + A[i]
      if A[i] ≥ A[t]
          S_5% ← S_5% + A[i]
  if S_5% > 10 × S_90%
      return true
  return false
```

Running Time: **Select** takes linear time using median of medians, so the first two steps are $O(n)$. The third step is a single iteration through the array, so that is also $O(n)$, giving us ***O(n) time*** as desired.

16.B. (70 PTS.) More generally we may want to compute the total number of questions of the top p% of student askers for various values of p. Suppose we are given A and k distinct numbers $0 < p_1 < p_2 < \ldots < p_k < 100\%$ and we wish to compute the total number of questions of the top p_i% of student askers for each $1 \leq i \leq k$. Assume for simplicity that np_i is an integer for each i.

Describe an algorithm for this problem that runs in $O(n \log k)$ time. You should prove the correctness of the algorithm and its runtime complexity.

Note that sorting will allow you to solve the problem in $O(n \log n)$ time but when $k \ll n$, $O(n \log k)$ is faster. Also, an $O(nk)$ time algorithm is relatively easy by repeating the previous part k times.

Solution:

The general idea is to convert the p's into ranks (which are integers). Let R be the sorted array of ranks. Then, use divide-and-conquer on R to find out top $p_i\%$.

Algorithm:

- *Step I*: Convert p's into ranks (which are integers between 1 and n). That is, compute $(100 - p_i)n/100$. Note that by assumption, this is an integer. These ranks are sorted in decreasing order, because the p's are sorted. Denote by R this sorted array of ranks.
- *Step II*: Define the following functions:
  - **Sum**(X) returns $\sum_{i=1}^{n} X[i]$ for an array $X[1, ..., n]$.
  - **Split**(X, j) returns two arrays X_s and X_l, which contain all the elements in X that are strictly smaller and strictly larger than the $X[j]$ respectively. Note that **Split**(X, j) may return an empty array if no such element exists, and it takes $O(n)$ time.

The pseudocode for our recursive divide-and-conquer algorithm on R is shown below. It prints out the total number of questions asked for each rank in R. We execute **FindNumQ**$(A, R, 0)$ presented below. The third parameter is the sum of the number of questions on the right side of the current array interval, i.e., if the current subarray is $A[i...j]$, then S_{right} is **Sum**$(A[j + 1..n])$.

```
FindNumQ(A, R, S_right):
if len(R) = 0
   return
m ← ⌈len(R)/2⌉
a_m ← Select(A, R[m])
(A_s, A_l) ← Segregate(A, a_m)
q ← Sum(A_l) + S_right
print q
R_l ← R[1, ..., m − 1]
R_s ← R[m + 1, ..., len(R)]
for r in R_l:
   r ← r − R[m]
FindNumQ(A_s, R_s, q)
FindNumQ(A_l, R_l, S_right)
return
```

Running Time:

- Step I takes $O(k)$.
- Step II takes $O(n \log k)$. **Select** and **Segregate** functions both takes $O(n)$ time to run, the recursive form is defined as

$$T(n, k) = T(r, k/2) + T(n - r, k/2) + O(n),$$

where n is the length of A and k is the length of R. The height of the recursion tree is $\log k$, and each level takes $O(n)$ time. Therefore, the running time for **FindNumQ** is $O(n \log k)$.

Therefore, the total running time is $O(n \log k)$.

Correctness:

Lemma 6.1. *For R of any size k, **FindNumQ**(A, R, S_{right}) prints the total earnings of $R[i]$ percentile rank earners in A plus S_{right} for all i.*

Proof: The proof is by induction on k.

- **Base case**: $k = 0$.
  In this case, $len(R) = 0$. Since we have nothing to output, **FindNumQ** is correct.
- **Inductive Step**: $k > 0$. Assume that **FindNumQ** is correct for all $c < k$. Now, we prove the statement for $c = k$. In the first few lines, m is computed to be half the size of R. We need to prove that, for all values in R, the outputted values are correct. We consider three different cases:
  - Case 1: $i = m$. In the algorithm, we first find the $R[m]$ rank element in A, a_m by QuickSelect. We then segregate the values using a_m and output q the sum of all the values greater than a_m plus S_{right}. This is exactly the number of questions of the $R[i]$ rank askers plus S_{right}.
  - Case 2: $i > m$. Here, we call the function **FindNumQ**(A_l, R_l, S_{right}). By the inductive hypothesis, this prints the total number of questions the of $R_l[i]$ rank askers in A_l plus S_{right} for all i. By construction, all the values in A_l are greater than the values in A_s, and hence would appear after them in sorted A. Thus, $R_l[i]$ rank askers in A_l plus S_{right} would be equal to $R[i]$ rank earners in A plus S_{right}.
  - Case 3: $i < m$. Here, we call the function **FindNumQ**(A_s, R_s, q). By the inductive hypothesis, this prints the total earnings of $R_s[i]$ rank askers in A_s plus $q = S_{right} +$ **Sum**(A_l) for all i. By construction, all the values in A_s are less than the values in A_l, and hence would appear before them in sorted A. Thus, number of questions in A_l should be added to outputs in the branch. Thus, $R_s[i]$ rank askers in A_s plus $S_{right} +$ **Sum**(A_l) would be equal to $R[i]$ rank askers in A plus Sum_{right}.

Thus, by induction, we have proven the correctness.

Rubric:

16.A
- 10 points for finding the boundary elements in linear time.
- 5 point for finding necessary sums in linear time.
- 5 point for computing the final answer and stating the correct running time.

16.B Out of 70 points, we would allocate:
- 30 points for an $O(n \log k)$ time algorithm.
- 20 points for stating the correct running time.
- 20 points for correctness proof.

Max 20 points for algorithms slower than $O(n \log k)$ but faster than $O(nk)$ and $O(n \log n)$. Max 0 points for $O(nk)$ or $O(n \log n)$ algorithms.

16.C. (10 PTS.) In an effort to encourage more discussion on Piazza, you will receive 10 points of credit if by March 23, 2020, you have asked a question or answered a question or contributed to any Piazza discussion at least once.

17 (100 PTS.) Strings

Let Σ be a finite alphabet and let L_1 and L_2 be two languages over Σ. Assume you have access to a subroutine **IsStringInL**(u, L) which returns true if $u \in L$ and false otherwise. Assume that **IsStringInL**(u, L) runs in constant time $O(1)$.

Using the subroutine as black boxes describe an efficient algorithm that given an arbitrary string $w \in \Sigma^*$ decides whether $w \in (L_1 \bullet L_2)^*$. Evaluate the running time of your algorithm in terms of $n = |w|$.

Solution:

Let $w \in \Sigma^*$ be the input, and let n be its length. We define the bloolean function: **IsStrCon**(u, L_1, L_2) that returns `true` if $u \in L_1 \bullet L_2$. We also define a boolean array $InStar(i)$, which is `true` if and only if the substring $w_i w_{i+1} \dots w_{n-1} w_n$ is in $(L_1 \bullet L_2)^*$. (Note that $InStar(n+1) =$ `true`, because the corresponding substring is the empty string and $\varepsilon \in (L_1 \bullet L_2)^*$)

By the definition of Kleene star, w is in $(L_1 \bullet L_2)^*$ if and only if w is either ε, or can be expressed as $w = w_1 w_2$ where $w_1 \neq \varepsilon$ is in $L_1 \bullet L_2$ and $w_2 \in (L_1 \bullet L_2)^*$. This gives us the following recursive definition:

$$InStar(i) = \begin{cases} \texttt{true} & \text{if } i = n+1 \\ \texttt{true} & \text{if } \exists\, k \text{ st. } i \leq k \leq n, \textbf{IsStrCon}(w_i w_{i+1} \dots w_k, L_1, L_2) \wedge InStar(k+1) \\ \texttt{false} & \text{Otherwise} \end{cases}$$

We need to compute $InStar(1)$.

We can memoize all function values into a one-dimensional array $InStar[1..n+1]$. Each array entry $InStar[i]$ depends only on the entries to the right. Thus, we can fill the array from right to left. The recurrence gives us the following pseudocode:

```
InStarCon?(w):
InStar[n + 1] ← true
for i ← n down to 1
    InStar[i] ← false
    for k ← i to n
        if InStar[k + 1] ∧ IsStrCon(w_i w_{i+1} ... w_k, L_1, L_2)
            InStar[i] ← true
            break
return InStar[1]
```

```
InStrCon(w, L_1, L_2):
n ← len(w)
for i ← 0 to n
    if IsStringInL(w_1 ... w_i, L_1) ∧ IsStringInL(w_{i+1} ... w_n, L_2)
        return true
return false
```

Running Time: The array assignments are constant time, as are the calls to **IsStringInL**. Each call to **IsStrCon** takes $O(n)$.

In **InStarCon?**, the outer loop runs n times, and the inner loop runs at most n times per iteration of the outer loop. The work inside of the loops is $O(n)$ due to **IsStrCon**. Hence, the total runtime of the algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

Rubric: Standard Dynamic Programming Rubric (Max 80 points for a slower solution, scale partial credit accordingly)

18 (100 PTS.) **Invest.**

You have a group of investor friends who are looking at n consecutive days of a given stock at some point in the past. The days are numbered. $i = 1, 2, \ldots, n$. For each day i, they have a price $p(i)$ per share for the stock on that day.

For certain (possibly large) values of k, they want to study what they call *k-shot strategies*. A k-shot strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$, where $0 \leq m \leq k$ and

$$1 \leq b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leq n.$$

We view these as a set of up to k nonoverlapping intervals, during each of which the investors buy 1,000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given k-shot strategy is simply the profit obtained from the m buy-sell transactions, namely,

$$1000 \cdot \sum_{i=1}^{m} (p(s_i) - p(b_i)).$$

Design an efficient algorithm that determines, given the sequence of prices, the k-shot strategy with the maximum possible return. Since k may be relatively large, your running time should be polynomial in both n and k.

Solution:

Let $Opt(m, i)$ denote the optimal m-shot strategy for days 1 through i. This definition assumes we do not own any stock at the end of day i. Clearly, $Opt(0, i) = 0$ as we cannot do any trading. Likewise, if $i < 2$, then $Opt(m, i) = 0$ as we do not have time to buy stock and sale at a later date. Now, consider the optimal m-shot strategy for days 1 through i. If we do not sell stock on day i, then the value of this strategy is equal to $Opt(m, i-1)$. Otherwise, we must buy at some earlier date j for a profit of $1000(p(i) - p(j))$, and then follow up with an optimal $(m-1)$-shot strategy for days 1 through $j-1$. We would want the best choice for j, leading us to the following recurrence.

$$Opt(m, i) = \begin{cases} 0 & \text{if } m = 0 \\ 0 & \text{if } i < 2 \\ \max\left\{ Opt(m, i-1), \max\limits_{1 \leq j \leq i-1} \Big(1000(p(i) - p(j)) + Opt(m-1, j-1) \Big) \right\} & \text{otherwise} \end{cases}$$

```
MaxStrategy(p[1...n]):
Opt ← 2-dimensional array indexed by [0...k][0...n]
for m ← 0 to k
  for i ← 0 to n
    if m = 0 or i < 2
      Opt[m][i] ← 0
    else
      Opt[m][i] ← Opt[m][i − 1]
      for j ← i − 1 down to 1
        if Opt[m][i] < Opt[m − 1][j − 1] + 1000(p[i] − p[j])
          Opt[m][i] ← Opt[m − 1][j − 1] + 1000(p[i] − p[j])
return Opt[k][n]
```

The above iterative algorithm computes the return for the optimal k-shot strategy. The algorithm fills a 2-dimensional array Opt where $Opt[m][i]$ is the return for the optimal m-shot strategy for days 1 through i. To make sure that the solutions for the subproblems that pair (m, i) depend on are available when we compute the solution for pair (m, i), we fill the array from low m to high m and from low i to high i within that.

Correctness. Note that we have already justified the recurrence. The iterative algorithm fills in entries based on the recurrence, and each entry is available because the recurrence always demands lower items in the arrays and we fill the arrays from low to high.

Running time. There are $O(nk)$ distinct subproblems and we can compute the solution for each subproblem in $O(n)$ time. Therefore the total running time is $O(n^2k)$.

Rubric: Standard dynamic programming rubric. -20 (out of 100) for an algorithm slower than $O(n^2k)$.