1. **Solution:** Begin by mapping the 2d tape on to the 1d tape in the following manner. Note that at any point in time there is only a finite number of rows with filled cells and each row has some furthest right filled cell. Starting with the first row, copy it to the 1d tape, placing a marker between each row (& is used in the example below) and a different marker at the very end ($ in the example).

   ("_" is used to denote an empty space)

i	_	_	_	
f	g	h	_	
d	e	_	_	
a	b	c	_	

   becomes

a	b	c	&	d	e	&	f	g	h	&	i	$

   If the 2D turing machine needs to create a new row, it can shift the end marker down as needed. If the 2D turing machine needs to write a character at the end of a row, all characters after that row in the 1d machine will be sifted right to accommodate.

   Moving the tape head left or right in the 2D machine is simulated by moving the same direction on the 1D machine. However, it also has to check for moving off the left (seeing a &) or moving onto a new space on the right (seeing a & or $ and shifting all remaining characters one to the right.

   Moving the tape head up or down is simulated by moving the tape head in the 1D machine by tracking how many cells are to the right of the previous & (This can be done by marking each character it comes across until it reaches a & or using a second tape to keep track). This gives the current column in the 2D TM. Now to reach the previous or next row, the 1D TM simply moves left or right until it sees another &. Then move the recorded number of cells to the right (un-mark previously marked characters or remove characters on the second tape). Moving off the left end of the 1D tape is the same as moving down past the bottom of the 2D tape.

   The asymptotic running time of this simulation with respect to the steps t of the 2D TM would be $O(t^2)$.

   Moving left takes $O(1)$ steps as the 1D TM doesn't have to do anything special.

   Moving right takes potentially $O(t)$ steps in the 1D machine when it has to shift all characters to the right. The input comes in on the first line, so those characters will never be shifted. All other characters will have to be placed by the 1D TM before being shifted, which results in $O(t)$ potential additional steps.

   Similarly for moving up or down, the 1D machine would take $O(t)$ steps as it has to move over the length of the current or previous row. This is $O(t)$ unless the 2D TM doesn't look at every element in the input, or the number of steps t taken is much smaller than the size of the input. ■

Rubric: 10 points.

- + 7 for the 1D TM simulation
- + 3 for the running time analysis

2. **Solution:**
   - We will divide the problem of merging k sorted arrays $A_1, \dots, A_k$, each of size n, as follows.
     - Merge $\lfloor k/2 \rfloor$ sorted arrays $A_1, \dots A_{\lfloor k/2 \rfloor}$ into a single sorted array B_1.
     - Merge $\lceil k/2 \rceil$ sorted arrays $A_{\lfloor k/2 \rfloor + 1}, \dots, A_k$ into a single sorted array B_2.

     We can recursively solve the above two problems and merge B_1 and B_2 into a single sorted array using the provided routine (let's call it Merge). The algorithm is then as follows.

```
MergeMultipleArrays($A_1[1..n], \dots, A_k[1..n]$):
  if $k = 1$
      return $A_1$
  $B_1 \leftarrow$ MergeMultipleArrays($A_1, \dots, A_{\lfloor k/2 \rfloor}$)
  $B_2 \leftarrow$ MergeMultipleArrays($A_{\lfloor k/2 \rfloor+1}, \dots, A_k$)
  return Merge($B_1, B_2$)
```

     First, let us show the running time of the algorithm. At the base case, we have $T(k) = O(1)$ for $k = 1$. For the recursion, B_1 is an array of size $n\lfloor k/2 \rfloor$ and B_2 is an array of size $n\lceil k/2 \rceil$. So merging them takes $O(n\lfloor k/2 \rfloor + n\lceil k/2 \rceil) = O(nk)$ time. Thus, the recurrence is given by

$$T(k) = \begin{cases} O(1) & \text{if } k = 1, \\ 2T(k/2) + O(nk) & \text{otherwise.} \end{cases}$$

     The recurrence can be solved to get an overall running time of $O(nk \log k)$.

     To show the correctness of the algorithm, we will use induction on k. Let k be an arbitrary integer ≥ 1. Let $A_1, \dots, A_k$ be k arbitrary sorted arrays (with the assumption that all numbers in the arrays are distinct), each of size n. We wish to show that MergeMultipleArrays, on input $A_1, \dots, A_k$, merges them into a single sorted array A of kn elements.

     For the base case, we have $k = 1$. In this case A_1 is already sorted and MergeMultipleArrays simply returns the single array A_1.

     For the inductive step, assume that MergeMultipleArrays correctly merges ℓ sorted arrays, for every $\ell < k$, into a single sorted array of size ℓn. From the inductive hypothesis, it follows that B_1 is a sorted array of size $\lfloor k/2 \rfloor n$ and B_2 is a sorted array of size $\lceil k/2 \rceil n$. Since Merge correctly merges the two arrays into a single sorted array, we conclude that MergeMultipleArrays correctly merges the k sorted arrays into a single sorted array.

   - The algorithm is as given below. We split the array of size N into k arrays of size $\lceil N/k \rceil$. Note that the kth array is dealt outside the for loop since $k \cdot \lceil \frac{N}{k} \rceil$ can be larger than N. Note also that each array B_i is of size $\lceil \frac{N}{K} \rceil$, except B_k. This can be easily fixed by appending large numbers at the end of B_k. We have skipped over this detail to keep the algorithm brief.

```
NewMergeSort($A[1..N]$):
  if $N = 1$
      return $A$
  for $i \leftarrow 1$ to $k - 1$
      $j \leftarrow (i-1) \cdot \lceil \frac{N}{k} \rceil$
      $B_i \leftarrow$ NewMergeSort($A[j+1..j+\lceil \frac{N}{k} \rceil]$)
  $B_k \leftarrow$ NewMergeSort($A[(k-1) \cdot \lceil \frac{N}{k} \rceil + 1..N]$)
  return MergeMultipleArrays($B_1, \dots, B_k$)
```

At the base case, we have $T(N) = O(1)$ for $N = 1$. At each step, it takes $O(1)$ to set up each recurrence There are a total of k recurrences, so it takes a total of $O(k)$ time to set them all up[1]. Finally, it takes $O(N \log k)$ time to run the MergeMultipleArrays routine (since $n = N/k$). This eclipses the $O(k)$ time taken to set up the recurrences (since $N > k$). Thus, the recurrence is given by

$$T(N) = \begin{cases} O(1) & \text{if } N = 1, \\ kT(\frac{N}{k}) + O(N \log k) & \text{otherwise.} \end{cases}$$

To solve the recurrence relation, note that at level i in the recurrence tree there are a total of k^i nodes. Each node represents a problem of size N/k^i. So the total work done at level i of the recurrence tree is $O(k^i \frac{N}{k^i} \cdot \log k) = O(N \log k)$. Since there are $\log_k N$ levels, the total work done (at the non-leaf nodes) is given by

$$\begin{aligned}\sum_{i=0}^{\log_k N - 1} O(N \cdot \log k) &= O(N \cdot \log_k N \cdot \log k) \\ &= O(N \cdot \frac{\log N}{\log k} \cdot \log k) \\ &= O(N \log N).\end{aligned}$$

Since there are a total of $O(k^{\log_k N}) = O(N)$ leaves, the total work done at leaves is $O(N)$. Thus, we conclude that the NewMergeSort algorithm runs in $O(N \log N)$ time, which is no better (asymptotically) than the regular merge sort.

To show the correctness of the algorithm, we will use induction on N. Let N be an arbitrary integer ≥ 1. We wish to show that NewMergeSort, on input an unsorted array A, sorts A.

For the base case, we have $N = 1$. In this case A is already sorted and NewMergeSort simply returns A.

For the inductive step, assume that NewMergeSort correctly sorts any arbitrary input array of size $\ell < N$. From the inductive hypothesis, it follows that each B_i, for $i \in [1, k]$, is a sorted array of size $\lceil N/k \rceil$. Since MergeMultipleArrays correctly merges the k sorted arrays (from the previous part), we conclude that NewMergeSort correctly sorts A.

■

[1]This also captures the time taken to append large numbers to B_k. This is because we will need to append at most k numbers and that will take $O(k)$ time

Rubric:
- 5 points.
  - 1 for minor error in algorithm (incorrect initialization, smaller problem size wrong by one value etc.)
  - 2 for error in algorithm.
  - 1 point for missing/ error in analyzing the running time.
  - 2 points for missing/ error in the correctness proof.
- 5 points.
  - 1 for minor error in algorithm (incorrect initialization, smaller problem size wrong by one value etc.)
  - 2 points for error in algorithm.
  - 2 points for missing/error in analyzing the running time.
  - 1 point for missing/error in the correctness proof of the algorithm.

3. **Solution:** (a) *General Idea*: Use QuickSelect to quickly find the top 1% and the bottom 70%, and iterate over A to find the sums of earnings.

Algorithm:

- Perform QuickSelect to find the index i containing the $(.70)n$-ranked element.
- Perform QuickSelect to find the index j containing the $((.99)n+1)$-ranked element.
- Iterate through the array and compute sum_{70} and sum_1 where sum_{70} is the sum of all incomes less than or equal to $A[i]$, and sum_1 is the sum of all incomes greater than or equal to $A[j]$.
- Return $10 \cdot sum_{70} < sum_1$.

Running Time: QuickSelect takes linear time, 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.

(b) *General idea:* First convert α's into ranks (which are integers). Denote by R the sorted array of ranks. Then, use divide-and-conquer on R to find out top α_i%.

Algorithm:

- *Step I*:
  In the preprocessing step, we convert α's into ranks (which are integers between 1 and n). That is, compute $\lfloor(100-\alpha_i)n/100\rfloor$.[2]. These ranks are sorted in decreasing order, because the α's are sorted. Denote by R this sorted array of ranks.
- *Step II*:
  For an array $X[1,...,n]$, define $sum(X) = \sum_{i=1}^{n} X[i]$. Then, denote by $total = sum(A)$. For simplicity, let $total$ be a global variable. That is, $total$ can be accessed at any level of the recursive call. Given an array X and an integer j, where $1 \le j \le len(X)$, Segregate(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 jth element in X, respectively. Note that Segregate(X, j) may return an

[2]For simplicity, assume that $(100-\alpha_i)n/100$ is an integer for all i. If not, then one needs to pay some attention to the step of updating r in FindSum to make sure that the rounding has been done properly.

empty array if no such element exists, and it takes $O(n)$ time. Then, we execute FindSum$(A, R, 0)$ presented below. The third parameter is the sum on the right side of the current array interval, i.e., if the current subarray is $A[i...j]$, then $rightSum$ is $sum(A[j+1..n])$.

```
FindSum($A, R, rightSum$):
    if $len(R) = 0$      ⟨⟨Base Case⟩⟩
        return
    $m \leftarrow \lceil len(R)/2 \rceil$      ⟨⟨Find m s.t. R[m] is R's median⟩⟩
    $a_m \leftarrow$ QuickSelect$(A, R[m])$      ⟨⟨Find A's (R[m])-th ranked element and use it as a pivot⟩⟩
    $(A_s, A_l) \leftarrow$ Segregate$(A, a_m)$      ⟨⟨Partition A by the pivot⟩⟩
    Output $sum(A_l) + rightSum$
    $R_l \leftarrow R[1, ..., m-1]$
    $R_s \leftarrow R[m+1, ..., len(R)]$
    for $r$ in $R_l$:
        $r \leftarrow r - R[m]$
    FindSum$(A_s, R_s, rightSum + sum(A_l))$
    FindSum$(A_l, R_l, rightSum)$
    return
```

Running Time:

- Step I takes $O(k)$.
- Step II takes $O(n \log k)$ as argued below: Since QuickSelect and Segregate function 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 FindSum is $O(n \log k)$.

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

Correctness:

Now, we prove correctness of the algorithm. The proof is by induction on k, the number of α's.

- **Induction Statement**: For R of any size k, $FindSum(A, R, rightSum)$ prints the total earnings of $R[i]$ percentile rank earners in A plus $rightSum$ for all i.
- **Base case**: $k = 0$.
  In this case, $len(R) = 0$. Since we have nothing to output, FindSum(A, R) is correct.
- **Inductive Step**: $k > 0$. Assume that FindSum(A, R) is correct for all $c < k$. Now, we prove the statement for $c = k$. In the first few lines, m is computed to be the 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, say a, by QuickSelect, segregate the values using a and output the sum of all the values greater than a plus $rightSum$. This is exactly the earnings of the $R[i]$ rank earners plus $rightSum$.
  - Case 2: $i > m$. Here, we call the function $FindSum(A_l, R_l, rightSum)$. By induction hypothesis, this prints the total earnings of $R_l[i]$ rank earners in

A_l plus $rightSum$ for all i. We notice, by construction, that 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 earners in A_l plus $rightSum$ would be equal to $R[i]$ rank earners in A plus $rightSum$.

- Case 3: $i < m$. Here, we call the function $FindSum(A_s, R_s, rightSum + sum(A_l))$. By induction hypothesis, this prints the total earnings of $R_s[i]$ percentile rank earners in A_s plus $rightSum + sum(A_l)$ for all i. We notice, by construction, that all the values in A_s are less than the values in A_l, and hence would appear before them in sorted A. Thus, salaries in A_l should be added to outputs in the branch. Thus, $R_s[i]$ rank earners in A_s plus $rightSum + sum(A_l)$ would be equal to $R[i]$ rank earners in A plus $rightSum$.

Thus, by induction, we have proven the correctness.

■

Rubric: (a)
- 1 point for finding the boundary elements in linear time.
- 1 point for finding necessary sums in linear time.
- 1 point for computing the final answer and stating the correct running time.

(b) Out of 7 points, we would allocate:
- 3 points for an $O(n \log k)$ time algorithm.
- 2 points for stating the correct running time.
- 2 points for correctness proof.

Max 2 points for algorithms slower than $O(n \log k)$ but faster than $O(nk)$.
Max 1 point for $O(nk)$ algorithms.