1. A Turing Machine (TM) M decides a language L if on any input string w the machine M halts in an accept state if $w \in L$ and in a reject state if $w \notin L$. In other words M is an algorithm for deciding membership in L. Note that we do not have any upper bound on the running time of M. We say that L is decidable if there is a TM M that decides L. We also called such a language L recursive. The purpose of this problem is to show that decidable languages are closed under basic operations.

   - (2 pts) Show that if L_1, L_2 are decidable then $L_1 \cap L_2$ and $L_1 \cup L_2$ are decidable.

   Solution: The following simple algorithm takes as input a string w and correctly checks whether $w \in L_1 \cap L_2$. Correctness follows directly from the definition of intersection and the fact that M_1 and M_2 are TMs for L_1 and L_2 respectively. Furthermore, we know that $M_\cap(w)$ will always terminate because M_1 and M_2 are guaranteed to terminate on every input.

   ```
   M_∩(w):
     if (M_1(w) and M_2(w)) then
         return YES
     Else
         return NO
   ```

   We can give an algorithm for $L_1 \cup L_2$ in essentially the same way; we simply have to check if w is in at least one of the two languages.

   ```
   M_∪(w):
     if (M_1(w) or M_2(w)) then
         return YES
     Else
         return NO
   ```

   ■

   Rubric: One point for each algorithm, -0.5 on each for minor errors.

   - (2 pts) Show that if L_1 and L_2 are decidable then $L_1 L_2$ is decidable (concatenation).

   Solution: A string $w \in L_1 L_2$ iff $w = xy$ where $x \in L_1$ and $y \in L_2$ (note that x, y can be ε). Since we don't care about efficiency, we can simply brute-force check every possible split of w into xy and see if any of them puts $w \in L_1 L_2$. This gives the following algorithm for deciding $L_1 L_2$:

   ```
   M.(w):
     if (M_1(ε) and M_2(w)) then
         return YES
     Else if (M_1(w) and M_2(ε)) then
         return YES
     Else
         n ← |w|
         for (i = 1 to n) do
             if (M_1(w[1..i]) and M_2(w[i+1..n]))
                 return YES
         return NO
   ```

Where for a string $w = a_1 a_2 \ldots a_n$ of length $n \geq 1$ and indices $1 \leq i \leq j \leq n$, we let the notation $w[i..j]$ to denote the substring $a_i..a_j$ of w. As in the previous part, we know that $M.$ terminates on all inputs because M_1 and M_2 do. ■

Rubric: 2 points for a correct algorithm. -1 for minor mistakes (for example not considering the case where x and/or y is ε).

- (2 pts) Show that if L is decidable then L^* is decidable.

Solution: Recall that $\varepsilon \in L^*$ for any L. Any non-empty string w is in L^* iff $w = w_1 w_2 \ldots w_k$ for some k such that for each $1 \leq i \leq k$, $w_i \in L_1$ and $|w_i| \geq 1$. Call a split of w into $w_1 w_2 \ldots w_k$ a non-trivial split if $|w_i| \geq 1$ for each i. The the above tells us that the following algorithm (with M as a TM for L) will correctly decide L^*:

```
M*(w):
  if (w = ε) return YES
  Else
      for each non-trivial split w_1 w_2 ... w_k of w do
          flag ← TRUE
          for (i = 1 to k)
              if not M(w_i)
                  flag ← FALSE
                  BREAK
          if (flag = TRUE) return YES
      return NO
```

To see that this algorithm always halts, we need to show that the number of non-trivial splits of w is finite (and can be enumerated over in finite time). Note that each non-trivial split can be uniquely identified by a length $|w| - 1$ bit string s, where for each $1 \leq i < |w|$ we let $s_i = 1$ if and only if we split w after the ith character. This tells us that there are finitely many non-trivial splits (in particular, $2^{|w|-1}$ of them), as well as giving us a natural way to enumerate them in finite time.

■

Solution: One can write the previous solution much more elegantly as a recursive program, which avoids the explicit enumeration step. In particular, note that a string w is in L^* if and only if either $w = \varepsilon$ or $w = xy$ where $x \in L$ and $y \in L^*$. Using the same brute-force technique from when we considered concatenation, we get the following algorithm for L^*, where M is a TM for L:

```
M*(w):
  if (w = ε) return YES
  Else
      n ← |w|
      for (i = 1 to n) do
          if (M(w[1..i]) and M*(w[i+1..n]) ) return YES
  return NO
```

A straightforward inductive proof shows that for all n, M_* halts on all inputs

of length n.

■

Solution: Letting IsWord() be a decision procedure for L, the text segmentation problem discussed in Section 2.5 of Jeff's textbook is exactly the problem of testing if a string is in L^*, so we can use the algorithm presented in the section. ■

Rubric: 2 points for a correct algorithm. -1 for minor mistakes (for example not considering the case where w is ε).

- (4 pts) Now suppose L is recursively enumerable. That is, there is a TM M with the following properties (i) if $w \in L$ then M on input w halts and accepts w (ii) if $w \notin L$, M either halts and rejects w or does not terminate. Show that L^* is recursively enumerable by reading about the technique called *dovetailing*. A useful resource is the following set of notes https://courses.grainger.illinois.edu/cs373/sp2009/lectures/lect_24.pdf.

Solution: Let Simulate be a subroutine that runs a given TM on a given input for a given number of steps, and "gives up" if the TM has not terminated by that time. Formally, Simulate(M, w, t) outputs YES if $M(w)$ outputs YES within t steps of computation, and outputs NO otherwise (that is, if $M(w)$ outputs NO within t steps *or* does not halt by the time we get to t steps). Importantly, note that Simulate(M, w, t) will *always* terminate, even if $M(w)$ does not.

With this in mind, we consider the first algorithm we wrote in the previous part for L^*. As written, it doesn't work for this part, as if we check the "wrong" non-trivial split first, some $M(w_i)$ for $w_i \notin L$ may never terminate, causing our algorithm to get stuck. To get around this, we use Simulate to limit how long we're willing to wait for M to terminate. Formally, consider the following modified algorithm:

```
M*(w):
  if (w = ε) return YES
  Else
      t ← 1
      repeat forever:
          for each non-trivial split w_1 w_2 ... w_k of w do
              flag ← True
              for (i = 1 to k)
                  if not Simulate(M, w_i, t)
                      flag ← False
                      BREAK
              if (flag = True) return YES
          t ← t + 1
```

Note first that, similar to the previous part, M_* will never output YES if $w \notin L^*$ (in particular, it is guaranteed to never terminate). Thus, all we have to show is that when $w \in L^*$, M_* will eventually terminate and output YES. Since $w \in L^*$, there is some non-trivial split of $w = w_1 w_2 \dots w_k$ such that for

all i, $w_i \in L$. By our guarantees on M, we know that $M(w_i)$ will output YES in finite time for each i; let $t^* \in \mathbb{N}$ be large enough so that $M(w_i)$ halts in at most t^* steps for each i. By our definition of SIMULATE, we know that for all i, SIMULATE(M, w_i, t^*) will output YES. Since M_* only spends a finite amount of time on each value of t (it makes a finite number of calls to SIMULATE, which itself always terminates), we know that it will eventually reach $t = t^*$, unless it terminates and outputs YES before then. Either way, we know that $M_*(w)$ will correctly terminate and output YES on any $w \in L^*$. ■

Rubric: 4 points for a correct algorithm. This is not the only correct way to do dovetailing.

- **Not to submit:** Show that if L_1 and L_2 are decidable then $(L_1 \cup L_2)^*$ is decidable.

For each of the problems you should describe a simple TM that for the given language in a high-level fashion assuming that L_1 has a TM M_1 and L_2 has a TM M_2. One way to think of the problem is to give a program for the given language using sub-routines for L_1 and L_2. No proof is necessary but clarity of your algorithm is important; give a brief description if necessary. Note that in decidability we do not pay attention to the quality of the running time so brute-force algorithms are fine.

2. This problem is on sorting and selection.

- (4 pts) Suppose you are given k sorted arrays $A_1, A_2, \ldots, A_k$ each of which has n numbers. Assume that all numbers in the arrays are distinct. You would like to merge them into single sorted array A of kn elements. Recall that you can merge two sorted arrays of sizes n_1 and n_2 into a sorted array in $O(n_1 + n_2)$ time. Use a divide and conquer strategy to merge the sorted arrays in $O(N \log k)$ time where $N = nk$ is the total number of elements in the k arrays.

Solution: We will divide the problem of merging k sorted arrays $A_1, \ldots, A_k$, each of size n, as follows.

- Merge $\lfloor k/2 \rfloor$ sorted arrays $A_1, \ldots A_{\lfloor k/2 \rfloor}$ into a single sorted array B_1.
- Merge $\lceil k/2 \rceil$ sorted arrays $A_{\lfloor k/2 \rfloor + 1}, \ldots, A_k$ 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], ..., A_k[1..n]):
  if k = 1
      return A_1
  B_1 ← MergeMultipleArrays(A_1, ..., A_{⌊k/2⌋})
  B_2 ← MergeMultipleArrays(A_{⌊k/2⌋+1}, ..., A_k)
  return Merge(B_1, B_2)
```

First, let us show the running time of the algorithm. At the base case, we have $T(1) = n$. (There can be some confusion on whether $T(1) = 1$ or $T(1) = n$; returning the array requires potentially copying it and it is safer to assume it takes time proportional to n. This turns out to not affect our asymptotics.)

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. We will assume that there is a constant c such that merging takes at most cnk time. Thus, the recurrence is given by

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

The recurrence can be solved to get an overall running time of $O(nk \log(k+1))$, which simplifies to $O(N \log k)$. (Note that we add the plus 1 to handle the case of $k = 1$.)

To show the correctness of the algorithm, we will use induction on k. Let k be an arbitrary integer ≥ 1. Let $A_1, \ldots, A_k$ 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 $A_1, \ldots, A_k$, 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. ■

Rubric: 3 points for a correct algorithm, 1 point for time analysis. Maximum of 1 point for an algorithm that runs in time $\Omega(Nk)$.

- You would like to do some data analysis. Suppose you are given the income of people as an n element unsorted array A, where $A[i]$ gives the income of i.
  - (2 pts) Describe an $O(n)$-time algorithm that given A checks whether the top 2% of earners together make more than ten times the total of the bottom 80%. 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: Recall from lecture that we can find the jth smallest element in an unsorted list of size n in time $O(n)$. We give an algorithm for this problem by first using the algorithm from class to find the income of the people at the 98th and 80th percentile of our list. We can then in a single pass over our list add up the incomes of everyone above the 98th percentile, and separately the incomes of everyone below the 80th percentile. Formally, letting Select be the algorithm from class, we have the following algorithm for this question:

```
CheckEconomicInequality(A):
  Let n = |A|
  Compute upper = Select(A, n · .98)
  Compute lower = Select(A, n · .8)
  Initialize totalHigh = totalLow = 0
  For (i = 1 to n) do
      If A[i] ≥ upper
          Add A[i] to totalHigh
      If A[i] ≤ lower
          Add A[i] to totalLow
  If totalHigh > 10 · totalLow return YES
  Else return NO
```

Correctness This follows immediately from the fact that the bottom 80% of earners are exactly those whose income is at most that of the 80th percentile, and so are exactly those whose incomes get added to totalLow. Similarly, the top 2% of earners are exactly those whose incomes get added to totalHigh.

Run Time Note that we make two calls to Select, each of which takes $O(n)$ time. After that we just have to make a single pass over the entire input list, which will also be $O(n)$ time. Thus, the total time our algorithm takes is $O(n)$ as desired. ■

Rubric: 1.5 points for a correct algorithm, 0.5 points for time analysis. Maximum of 0.5 points for an algorithm that runs in time $\Omega(n \log n)$.

– (4 pts) More generally we may want to compute the some additional statistics. Suppose we are given A and k numbers $\alpha_1 < \alpha_2 < \ldots < \alpha_k$ each of which is a number between 0 and 100 and we wish to compute the total earnings of the top $\alpha_i\%$ of earners for $1 \leq i \leq k$. Assume for simplicity that $\alpha_i n$ is an integer for each i. Describe an algorithm for this problem that runs in $O(n \log k)$ time. 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. An $O(nk)$ time algorithm is relative easy.

Solution: Our main idea is to do divide and conquer on our list of percentages $(\alpha_1, \ldots, \alpha_k)$. In order to make this efficient, we note that for all α_i below our pivot, we don't need to care about anyone with too low incomes (in particular, any income below the income corresponding to our pivot), since they don't affect our answers. Similarly, for all the α_i above our pivot, we know that every income above the one corresponding to our pivot will be added to their final answers, so we can ignore those incomes in our recursive call and simply add them in at the very end.

Formally, we give the following algorithm. For simplicity, we assume our algorithm takes the *rank* of the cutoff earner rather than the percentage. In order to solve the question as posed, we would simply run ComputeStatistics with each r_i set to $(1 - \frac{\alpha_i}{100}) \cdot |A| + 1$.

```
ComputeStatistics(A, (r_1, ..., r_k)):
  ⟨⟨For each i, we want the sum of all incomes of rank r_i or higher in A⟩⟩
  ⟨⟨We are guaranteed that |A| ≥ r_1 > r_2 > ... > r_k ≥ 1.⟩⟩
  If k = 1
      Compute cutoff = Select(A, r_1)
      Return sum of all incomes in A that are at least cutoff
  Else
      Let n = |A| and m = ⌊k/2⌋ + 1
      Compute pivot = Select(A, r_m)
      Compute A^+ = {a ∈ A | a > pivot}
      Compute A^- = {a ∈ A | a ≤ pivot}
      ⟨⟨High cutoffs: drop incomes below pivot, update ranks to match⟩⟩
      Compute left = ComputeStatistics(A^+, (r_1 − r_m, ..., r_{m−1} − r_m))
      ⟨⟨Low cutoffs: add high incomes to all⟩⟩
      Compute right = ComputeStatistics(A^-, (r_m, ..., r_k))
      Compute sum of all incomes in A^+ and add to each element of right
      Return left concatenated with right
```

Correctness We show that the sum of incomes at or above rank r_i is computed correctly by considering two cases. First, if $r_i > r_m$, all incomes at or above rank r_i are contained in A^+; their ranks in A^+ are r_m lower than they were in A. Thus, the sums for these ranks are computed correctly in left. Second, if $r_i \leq r_m$, all incomes in A^+ need to be included in the sum. In addition to these, we also need to add all incomes of rank at least r_i in A^-. Thus, the sums for these ranks are correctly computed in right.

Run Time Outside of recursive calls, everything in ComputeStatistics can be done using linear-time passes over A and associated arrays, for a total of $O(n)$ time. In addition to this, we make two recursive calls, where the value of k is halved in each and the sum of the sizes of the recursive arrays is equal to n. Thus, we get the recursion formula

$$T(n,k) = T\left(n_1, \frac{k}{2}\right) + T\left(n_2, \frac{k}{2}\right) + O(n)$$

for some n_1, n_2 such that $n_1 + n_2 = n$. If we draw out the recurrence tree for this, we note that the work at each layer is always cn for some constant c. Since the recursion terminates when $k = 1$, the tree will have $O(\log k)$ depth, which gives us our final $O(n \log k)$ run time.

■

Rubric: 1 point for each: base case ($k = 1$), divide step (splitting up A and percentage / rank cutoffs), conquer step (merging the answers to the split problems), and time analysis. Maximum of 1 point for an algorithm that runs in time $\Omega(n \log n)$ or $\Omega(nk)$.