1. Recall that 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. The purpose of this problem is to show that decidable languages are closed under basic operations.

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

Solution: We will assume access to two sub-routines called IsStringInL_1() and IsStringInL_2() that are decision procedures for L_1 and L_2 respectively. The following simple algorithm takes as input a string w and correctly checks whether $w \in L_1 \cap L_2$. The correctness is easy and the fact that it always terminates follows from the fact that we are assuming that IsStringInL_1() and IsStringInL_2() are also algorithms that always terminate on their input.

```
IsStrIn L1 ∩ L2(w):
  if (IsStringInL1(w) and IsStringInL2(w)) then
      return YES
  Else
      return NO
```

For union the algorithm simply checks whether the input string w is in at least one of the two languages.

```
IsStrIn L1 ∪ L2(w):
  if (IsStringInL1(w) or IsStringInL2(w)) then
      return YES
  Else
      return NO
```

■

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

Solution: As before, we assume access to two sub-routines called IsStringInL_1() and IsStringInL_2(). A string $w \in L_1 \cdot L_2$ iff $w = xy$ where $x \in L_1$ and $y \in L_2$. Note that x, y can be ε. 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 will use the notation $w[i..j]$ to denote the substring $a_i..a_j$ of w.

```
IsStrIn L1 · L2(w):
  if (IsStringInL1(ε) and IsStringInL2(w)) then
      return YES
  Else if (IsStringInL1(w) and IsStringInL2(ε)) then
      return YES
  Else
      n ← |w|
      for (i = 1 to n) do
          if (IsStringInL1(w[1..i]) and IsStringInL2(w[i+1..n]))
              return YES
      return NO
```

■

- Show that if L_1 is decidable then L_1^* is decidable.

Solution: Recall that $\varepsilon \in L_1^*$ for any L_1. Further, a string w with $|w| \geq 1$ is in L_1^* 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. Then it is easy to see that the number of non-trivial splits is finite. In fact there are exactly $2^{|w|-1}$ valid splits; each valid split correspond to choosing for each $1 \leq i < |w|$ whether to add a split after the i'th character or not. It is easy to enumerate them. We can thus write the following high-level algorithm to check if $w \in L_1^*$.

```
IsStrInL_1^*(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 IsStringInL_1(w_i)
                  flag ← FALSE
                  BREAK
          if (flag = TRUE) return YES
      return NO
```

One can write the above more elegantly as a recursive program to avoid the explicit enumeration step.

```
IsStrInL_1^*(w):
  if (w = ε) return YES
  Else
      n ← |w|
      for (i = 1 to n) do
          if (IsStringInL_1(w[1..i]) and IsStrInL_1^*(w[i+1..n]) ) return YES
  return NO
```

■

Solution: Letting IsWord() be a decision procedure for L_1, 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_1^*, so we can use the algorithm presented in the section. ■

Rubric: 10 points. 3 points each for the first two parts and 4 points for the last part.

- -1 for minor errors

2. Suppose you are given k sorted arrays $A_1, A_2, \dots, A_k$ 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 $O(nk \log k)$ time. To prove the correctness of the algorithm you can assume a routine to merge two sorted arrays.

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], ..., 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 $T(k) = n$ for $k = 1$. There can be some confusion on this point 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.

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 $O(nk \log(k+1))$. We add a 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 $A_1, \dots, A_k$ 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. ■

- In MergeSort we split the array of size N into two arrays each of size $N/2$, recursively sort them and merge the two sorted arrays. Suppose we instead split the array of size N into k arrays of size N/k each and use the merging algorithm in the preceding step to combine them into a sorted array. Describe the algorithm formally and analyze its running time via a recurrence. You do not need to prove the correctness of the recursive algorithm.

Solution: 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 ← 1 to k − 1
      j ← (i − 1) · ⌈N/k⌉
      B_i ← NEWMERGESORT(A[j + 1 .. j + ⌈N/k⌉])
  B_k ← NEWMERGESORT(A[(k − 1) · ⌈N/k⌉ + 1 .. N])
  return MERGEMULTIPLEARRAYS(B_1, ..., 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[a]. 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

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

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. ■

[a] 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

- **Extra credit:** This is a generalization of the first part. Suppose the k arrays are of potentially different sizes $n_1, n_2, \ldots, n_k$ where $N = \sum_{i=1}^{k} n_i$. Describe and analyze an $O(N \log k)$ algorithm to obtain a sorted array.

Solution: The algorithm is the same as the one for first part. We will ignore the non-uniform sizes.

- 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)
```

The correctness of the algorithm follows the same outline as the one from the first part. The only thing to check is the running time. We will use a two parameter recurrence. Let $T(N, k)$ be the running time of merging k sorted arrays with a total of N elements. We have $T(N, k) = N$ for $k = 1$.

For the recursion, B_1 is an array of size N_1 and B_2 is an array of size N_2 where $N_1 + N_2 = N$. So merging them takes $O(N)$ time. We will assume that there is a constant c such that merging takes at most cN time. Thus, the recurrence is given by

$$T(N, k) \leq \begin{cases} cN & \text{if } k = 1, \\ T(N_1, \lfloor k/2 \rfloor) + T(N_2, \lceil k/2 \rceil) + cN & \text{otherwise.} \end{cases}$$

One can prove by induction that $T(N,k) = O(N\log(k))$ but it is a bit tedious. Intead we will consider the recursion tree approach. For simplicity assume k is a power of 2. The recursion tree is a complete binary tree with k nodes at the leaves and depth $\log k$. The work at the root node is cN. What about the work at the next level? It is $cN_1 + cN_2$ which is cN. One can prove easily by induction that the total work at each level is cN and there are $\log k$ levels and hence the total work is $O(N\log k)$.

Thus the non-uniformity in the arrays does not really matter. ■

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 justification (a full formal proof of correctness is not necessary).
- 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 justification of correctness for the algorithm (a full formal proof of correctness is not necessary).
- 5 points. 2.5 points for correct algorithm and 2.5 points for analysis of running time.

3. Sorting is a fundamental and heavily used routine and can be done in $O(n \log n)$ time for a list of n numbers. In the comparison tree model there is a lower bound of $\Omega(n \log n)$ for sorting. Selection can be done in $O(n)$ time. Although a faster Selection algorithm may not be as directly useful in practice as Sorting, the ideas behind a linear time algorithm for it are theoretically interesting and related ideas play an important role in other problems. For each of the problems below use Selection as a black box algorithm to derive an $O(n)$ time algorithm.

   - It is common these days to hear statistics about wealth inequality in the United States. A typical statement is that the the top 1% of earners together make more than ten times the total income of the bottom 70% of earners. You want to verify these statements on some data sets. Suppose you are given the income of people as an n element *unsorted* array A, where $A[i]$ gives the income of person Describe an algorithm that given A checks whether the top 1% of earners together make more than ten times the bottom 70% together. Assume for simplicity that n is a multiple of 100 and that all numbers in A are distinct.

Solution: We will use SELECT($A[1..n], k$) as a black box routine that given an array A of n numbers and an integer k such that $1 \leq k \leq n$ returns the k'th ranked element in A.

The algorithm for this problem is simple. We obtain $x = \text{SELECT}(A[1..n], 0.7n)$ and $y = \text{SELECT}(A[1..n], 0.99n - 1)$. Once we have x we can scan the array A once in $O(n)$ time to compute the sum of all numbers less than equal to x and obtain their sum s_1 which is the total income of the bottom 70% of earners. Similarly we can compute s_2 which is the sum of all numbers in A that are greater than y which gives us the total income of the top 1% of earners. We then compare if $s_1 < s_2$ to check whether the claim is true. Total time is $O(n)$ plus the time for the two calls to SELECT which by our assumption is $O(n)$.

```
IncomeIneqCheck(A[1..n]):
  x ← SELECT(A[1..n], 0.7n)
  y ← SELECT(A[1..n], 0.99n − 1)
  s_1 ← 0
  for (i = 1 to n) do
      if (A[i] ≤ x) s_1 ← s_1 + A[i]
  s_2 ← 0
  for (i = 1 to n) do
      if (A[i] > y) s_2 ← s_2 + A[i]
  if (s_1 < s_2) return YES
  Else return NO
```

■

- Describe an algorithm to determine whether an arbitrary array $A[1..n]$ contains more than $n/6$ copies of any value.

Solution: First we observe the following simple fact. Given a number x and an array $A[1..n]$ we can count the number of times that x occurs in A in $O(n)$ time by a simple scan.

Now for the main problem. We will assume that $n > 6$ for otherwise the answer is always yes. *Imagine* that we sort A and let us call the sorted array B. Then all copies of any value will be next to each other. Suppose A contains an element x which occurs more than $n/6$ times. Let i and j be the first and last occurences of x in B. Then $j - i + 1 > n/6$. This implies that at least one of the indices $\lfloor n/6 \rfloor, 2\lfloor n/6 \rfloor, \ldots, 7\lfloor n/6 \rfloor$ must lie in the interval $[i, j]$ which means that x must be the rank h element for some $h \in \{\lfloor n/6 \rfloor, 2\lfloor n/6 \rfloor, \ldots, 7\lfloor n/6 \rfloor\}$. We can use SELECT 7 times to fine the elements corresponding to these ranks. And then check for each of them whether they occur more than $n/6$ times.

```
CheckFrequentItem(A[1..n]):
  for (h = 1 to 7) do
      x_h ← SELECT(A[1..n], h⌊n/6⌋)
  for (h = 1 to 7) do
      Count the number of times x_h occurs in A. Let n_h be the count.
      if (n_h > n/6) return YES
  reutrn NO
```

There are at most 7 calls to SELECT and 7 additional scans of A. Hence the total time is $O(n)$. ■

- The *square distance* between a pair of integers x, y is defined as the quantity $(x - y)^2$. The input is an an array A of n integers and an integer k such that $1 \leq k \leq n$. Describe an algorithm to find k elements in A with the smallest square distance to the median (i.e. the element of rank $\lfloor n/2 \rfloor$ in A). For instance, if $A = [9, 5, -3, 1, -2]$ and $k = 2$, then the median element is 1, and the 2 elements in A with the smallest square distance to the median are $\{1, -2\}$. If $k = 3$, then you can output either $\{1, -2, -3\}$ or $\{1, -2, 5\}$.

Solution: The algorithm first computes the median x of A by one call to SELECT in $O(n)$ time. Then it forms a new array $B[1..n]$ where $B[i] = (A[i] - x)^2$. This takes $O(n)$ time. Then it does a second call to SELECT on B to find the rank k element y. It then go through A to find all elements whose square distance to x is at most y and stop after finding the first k. One has to be a bit careful to take care of ties with y; we will store them separately and add an appropriate amount of them at the end.

```
MinSquareDistToMedian(A[1..n], k):
  x ← SELECT(A[1..n], ⌈n/2⌉)
  Allocate an array B of size n
  for (i = 1 to n) do
      B[i] ← (A[i] − x)^2
  y ← SELECT(B[1..n], k)
  count ← k
  for (i = 1 to n) do
      if ((A[i] − x)^2 < y)
          Add A[i] to output list O
          count ← count − 1
      else if ((A[i] − x)^2 == y)
          Add A[i] to temporary list T
  Add any k − count items from temporary list T to output list O
  Output list O of size k
```

The running time is dominated by two calls to SELECT, plus a for loop, each of which takes $O(n)$ time. ■

Rubric: 10 points. 3 points for the first and third parts, 4 points for the second.

- 2 points for an $O(n)$ algorithm. −1 for a minor error; no credit for an $\omega(n)$ time algorithm.
- 1 point for brief justification (2 points for second part). A correct justification for a correct $\omega(n)$ time algorithm gets this point.