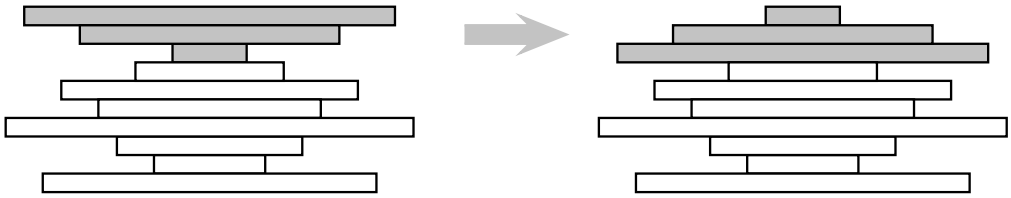
1. Suppose we have a stack of n pancakes of different sizes. We want to sort the pancakes so that the smaller pancakes are on top of the larger pancakes. The only operation we can perform is a *flip* - insert a spatula under the top k pancakes, for some k between 1 and n, and flip them all over.



(a) Describe an algorithm to sort an arbitrary stack of n pancakes and give a bound on the number of flips that the algorithm makes. Assume that the pancake information is given to you in the form of an n element array A. $A[i]$ is a number between 1 and n and $A[i] = j$ means that the j'th smallest pancake is in position i from the bottom; in other words $A[1]$ is the size of the bottom most pancake (relative to the others) and $A[n]$ is the size of the top pancake. Assume you have the operation Flip(k) which will flip the top k pancakes. Note that you are only interested in minimizing the number of flips.

Solution: Our algorithm will utilize an idea similar to that of selection sort. We start by finding the largest pancake. If it's not in the correct position, then we flip it to the top of the stack and flip it one more time to place it at the very bottom. The algorithm will then proceed by finding the next largest pancake, flipping it to the top (if it's not already in its correct position), and then flipping it to its correct position above the largest pancake. We do this process for each pancake.

```
PancakeSort(A):
  n = length of A
  for i = 1 to n:
      if A[i] ≠ n − i + 1:
          m = FindIndexOfLargest(A, i)
          Flip(n − m + 1)
          Flip(n − i + 1)
```

```
FindIndexOfLargest(A, j):
  loc = 0
  value = −∞
  for i = j to n:
      if A[i] > value:
          loc = i
          value = A[i]
  return loc
```

The correctness here follows by an easy induction (although a proof of correctness is not required for full credit on this problem).

The maximum number of flips required is $2n = O(n)$, as each pancake requires at most two flips to move to its correct position. Note that this bound can be reduced slightly to $2n - 3$ flips by recognizing that the second-smallest pancake only requires at most a single flip to place it in the correct position, and the smallest pancake requires no flips.

In addition, note that the actual running time of the algorithm is asymptotically greater than $O(n)$ since each search for the largest pancake takes linear time. The implementation shown above is not very efficient - it runs in $O(n^2)$ time. One can do better by using data-structures, but it is not trivial.

■

(b) Suppose one side of each pancake is burned. Describe an algorithm that sorts the pancakes with the additional condition that the burned side of each pancake is on the bottom. Again, give a bound on the number of flips. In addition to A, assume that you have an array B that gives information on which side of the pancakes are burned; $B[i] = 0$ means that the bottom side of the pancake at the i'th position is burned and $B[i] = 1$ means the top side is burned. For simplicity, assume that whenever Flip(k) is done on A, the array B is automatically updated to reflect the information on the current pancakes in A.

No proof of correctness necessary.

Solution: We will run the same algorithm as in part (a), except with a slight modification. After we flip the largest unsorted pancake to the top, we check to see which side is burnt. If the burnt side is on top, then we do nothing, and the next step in our algorithm will flip it into its correct position with the burnt side on the bottom. However, if the largest unsorted pancake has a burnt side facing down when it is at the top of the stack, then we do a flip so that the burnt side is facing up.

```
BurntPancakeSort(A):
  n = length of A
  for i = 1 to n:
      if A[i] ≠ n − i + 1 or B[i] ≠ 0:
          m = FindIndexOfLargest(A, i)
          Flip(n − m + 1)
          if B[n] = 0:
              Flip(1)
          Flip(n − i + 1)
```

The number of flips required is $3n = O(n)$. Each pancake requires at most three flips to move to its correct position: one to be flipped to the top, one flip to make sure the burnt side is in the right side, and one to put the pancake in the correct position. Note that once again, the actual running time of the algorithm is asymptotically greater than $O(n)$ since each search for the largest pancake takes linear time. ■

Rubric: 5 points each for parts (a) and (b), split as follows:

- 3 points for a correct algorithm.
  - Score scaled to 3 points per part if number of flips is worse than $O(n)$.
- 2 points for correctly analyzing the number of flips.
  - -1 point if asymptotic bound on the number of flips is correct, but exact bound is not mentioned while explaining the bound.

2. 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 $O(nk \log k)$ 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 $A_1, \ldots, A_k$, 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 $T(k) = O(1)$ 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 $A_1, \ldots, A_k$ 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: 5 points split as follows.
   – 2 points for the algorithm.
   – 1 point for analyzing the running time.
   – 2 points for proving the correctness of the algorithm.

- 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.

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[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.

[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

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. ■

Rubric: 5 points split as follows.

- 2 points for the algorithm.
- 2 points for analyzing the running time.
- 1 point for proving the correctness of the algorithm.

3. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

Solution: The algorithm is formally described below. We use the fact that the selection problem can be solved in linear time. That is, given an unsorted array A of n values and an index j between 1 and n, we can find the j-th ranked element in A in $O(n)$ time. We denote this black box algorithm as SELECT(A[1..N], j) which returns the value of the j-th ranked element in A. To determine whether an element appears more than $n/4$ times, we select values with rank $n/4$, $2n/4$, and $3n/4$. If an element x appears more than $n/4$ times, it follows that at least one of these selected values is equal to x. Thus, we can scan and count the number of occurrences of each of these selected values.

```
ContainsN/4Duplicates(A[1..N]):
  x_1 ← Select(A, ⌈N/4⌉)
  x_2 ← Select(A, ⌈2N/4⌉)
  x_3 ← Select(A, ⌈3N/4⌉)
  for i ← 1 to 3
      count ← 0
      for j ← 1 to N
          if A[j] = x_i
              count ← count + 1
      if count > N/4
          return True
  return False
```

Since SELECT runs in $O(n)$ time, finding x_1, x_2, and x_3 also takes $O(n)$ time. Looping over the array of length n a total of 3 times takes $O(n)$ time. Thus, this algorithm runs in the required $O(n)$ time.

To prove correctness of the algorithm, we must show that if an element appears more than $n/4$ times, it must be at least one of the selected values with rank $\lceil n/4 \rceil$, $\lceil 2n/4 \rceil$, or $\lceil 3n/4 \rceil$. Assume an element x appears $i > n/4$ times. Then, there must be consecutive ranks $j, \ldots, j+i-1$ with value x. Without loss of generality, consider the number of values of rank between $\lceil n/4 \rceil$ and $\lceil 2n/4 \rceil$ (excluding the outside values). Since $\lceil n/4 \rceil \geq n/4$ and $\lceil 2n/4 \rceil \leq 2n/4+1$, the maximum number of values is given by $(2n/4+1)-(n/4)-1 = n/4$. Thus, there are at most only $n/4$ spots for more than $n/4$ values. By pigeonhole principle, one of the selected values must be equal to x. ■

Rubric: 5 points split as follows.

- 2 points for the algorithm.
- 1 point for analyzing the running time.
- 2 point for proof of correctness.

(b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

Solution: To solve this problem, we first solve the following, more general, sub-problem. Given an array A with n elements and another array B of ranks $1 \leq i_1 < i_2 < ... < i_m \leq n$, design an algorithm that returns the k elements in A that have ranks $i_1, ..., i_m$. In the following solution we use SEGREGATE(A, p) to denote the function that partitions array A with pivot p into two sub-arrays. FINDRANK will take in an array of elements, A, an array of ranks B, and populate the values with those ranks in C.

```
FindRank(A[1..N], B[1..M], C[1..M]):
  if M = 0
      return
  mid ← ⌈M/2⌉
  rank_m ← Select(A, B[m]) ⟨⟨Find the mid ranked element to use as pivot⟩⟩
  (A_s, A_l) ← Segregate(A, rank_m) ⟨⟨Partition A by the pivot⟩⟩
  B_s ← B[1, .., mid − 1]
  B_l ← B[mid + 1, .., M]
  for b in B_l ⟨⟨Readjust indicies for right sub-array to start from mid⟩⟩
      b ← b − mid
  FindRank(A_s, B_s, C[1, .., mid − 1])
  C[mid] ← rank_m
  FindRank(A_s, B_s, C[mid + 1, .., M])
```

Since SELECT and SEGREGATE function both take $O(n)$ time to run, the recurrence is defined as

$$T(n,m) = \begin{cases} O(1) & \text{if } m = 0, \\ T(r, m/2) + T(n-r, m/2) + O(n) & \text{otherwise.} \end{cases}$$

where n is the length of A, m is the length of B, and r is the size of the left partition. The work done at each level of the recursion tree can be seen to be $O(n)$. The height of the recursion tree is $O(\log m)$, so the total running time is $O(n \log m)$.

Going back to the original problem, we use FINDRANK on ranks that are multiples of k, that is, $(k, 2k, ..., \lfloor n/k \rfloor * k)$. Similar to the previous question, we assert that if an element x appears more than k times, one of the pivots must be equal to x. Previously, we only had to scan through the array a constant number of times. However, using the same approach would now take $O(n) * \lfloor n/k \rfloor = O(n^2/k)$ time.

One may notice that assuming FINDRANK modifies the input array, A will be sorted such that elements smaller than any pivot are before that pivot and elements larger are after that pivot. This allows us to check only the k values before and after each pivot for duplicates. This algorithm is defined below.

```
ContainsKDuplicates(A[1..N], k):
  B ← [k, 2k, ..., ⌊n/k⌋ * k]
  C ← [0] * Len(B) ⟨⟨Array of zeros of same length as B⟩⟩
  FindRank(A, B, C) ⟨⟨This partially sorts A as specified above⟩⟩
  for i ← 1 to Len(B)
      count ← 0
      pivotIdx ← B[i]
      for j ← (pivotIdx − k) to (pivotIdx + k)
          if A[j] = A[pivotIdx]
              count ← count + 1
      if count > k
          return True
  return False
```

This approach iterates $2k$ * $\lfloor n/k \rfloor$ times, resulting in a running time of $O(n)$. However, since FindRank still has running time $O(n \log n/k)$, this algorithm also runs in $O(n \log n/k)$ time.

To prove correctness, we use a very similar approach to part (a). We want to show that if an element x appears more than k times, one of the pivots will have value x. Assume an element x appears $i > k$ times. Then, there must be consecutive ranks $j, ..., j+i-1$ with value x. However, between any two pivots, there are at most $k-1$ values. Thus, there are at most $k-1$ slots for more than k values. By pigeonhole principle, one of the pivots must be equal to x.

■

Rubric: 5 points split as follows.

- 2 points for correct algorithm.
- 2 points for having the optimal running time.
- 1 point for proof of correctness.