TJ IOI 2017
Written Round Solutions

Thomas Jefferson High School for Science and Technology

Saturday, January 13, 2018

Contents

1 Introduction

This document will outline possible solutions to the written round of TJ IOI 2017. At the time of the contest, grading was performed without formally written solutions, so these have been written after the fact. These solutions may be useful for teams preparing for future TJ IOI contests, and should be considered more relevant than written rounds from the TJ IOI contests in 2012 or 2013.

It is worth noting that errors in the problem statements exist; we regret the presence any such errors, and have tried to clarify the ones we have found in the solutions below. If more errors are found, please contact the TJ IOI officers so that this document can be updated appropriately.

2 Bead sort

This section was intended as a fun complement to the T-shirt designed, which prominently featured bead sort on the reverse. Though it wasn't very serious, we hope it was enjoyable to think about!

Problem 2.1 (4 points)**.** According to physics, the motion of a falling object that starts at rest is described by

$$\Delta y = -\frac{1}{2}gt^2$$

where g is the acceleration of gravity relative to Earth's surface. Explain why bead sort can be said to have a complexity of $O(\sqrt{N})$, where N is the number of integers to sort.

Solution: Solving for t in the above equation, we see that

$$t = \sqrt{\frac{2\Delta y}{g}}$$

By construction, the maximum distance a bead can fall Δy is proportional to the total number of beads N. As a result, t grows like $\sqrt{N}$ asymptotically, so it is appropriate to say that the time complexity is $O(\sqrt{N})$.

Problem 2.2 (4 points)**.** Despite its supposed $O(\sqrt{N})$ runtime, there are many issues that would prevent an actual implementation of bead sort from achieving such a runtime. Explain why it is impossible to implement a sort that performs faster than $O(N)$ (without techniques such as parallel or quantum computing).

Solution: Possible answers could include, but are not limited to:

- In order to set up the problem, we'll need to position the beads in place. Since the maximum height of a bead is proportional to N, this will take at least $O(N)$ time.
- We can't represent the N elements in a list without examining the position of each element at least once, which takes $O(N)$ time.

Problem 2.3 (4 points)**.** Besides questions about its complexity, bead sort is more limited than the other algorithms in this section, such as selection sort. Give an example of one such limitation.

Solution: Possible answers could include, but are not limited to:

- Only positive integers can be sorted, and relatively small ones at that.
- Other types of items, such as strings, can't be sorted.

3 Elementary sorts

This section first covered selection and insertion sort, which were covered in the study guide and which many students may have studied before. The questions here were to help students develop their understanding of sorting algorithms, their analysis, and associated terminology. Though the provided algorithms contained minor errors, we hope they were not significant enough so as to affect comprehension of the algorithms.

3.1 Selection sort

Note: The provided selection sort algorithm was incorrect; we regret the error. Below is a corrected version.

Algorithm 1 Selection sort

```
function SelectionSort(A)
    N ← length of A
    for i ∈ 0 ... N do
        min ← i                                   ▷ Corrected from min ← 0
        for j ∈ i + 1 ... N do
            if A[j] < A[min] then
                min ← j
        swap A[i] and A[min]
```

Problem 3.1 (4 points)**.** Compute the number of comparisons and exchanges that the above algorithm performs on an array of length N.

Solution:
Comparisons. When the loop iteration is i, then the value of j ranges from $i+1$ to N, giving $N-1-i$ comparisons for each iteration. So the total number of comparisons is

$$(N-1)+(N-2)+\cdots+2+1+0=\frac{N(N-1)}{2}$$

Full points were given for the answers $\frac{N(N-1)}{2}$, $\frac{N^2}{2}$, and $O(N^2)$, while half credit was awarded for N^2.

Exchanges. For each loop iteration, the algorithm performs one exchange, for a total of N exchanges. In fact, only $N-1$ exchanges are necessary, as the last loop iteration is not neccessary. Full credit was awarded for both of these and $O(N)$.

Problem 3.2 (4 points)**.** Before every iteration of the loop, what do the values of $A[0 \ldots i]$ represent, with respect to the original array? We can call this an *invariant*, or a condition that must remain true.

Solution: The values of $A[0 \ldots i]$ always represent the smallest i elements in the array A on any loop iteration, in sorted order.

Problem 3.3 (4 points)**.** Does the above algorithm result in a stable sort? If it is unstable, what would you change to make it stable?

Solution: No, the algorithm does not result in a stable sort. Consider, for example, the array $[7, 7, 0]$. On the first loop iteration, the 7 at position 0 will be moved to position 2, putting it after the 7 at position 1. One way to fix this is by shifting elements to insert $A[min]$ in the correct position, though this is less efficient.

3.2 Insertion sort

Note: The provided insertion sort algorithm was incorrect; we regret the error. Below is a corrected version.

Algorithm 2 Insertion sort

```
function InsertionSort(A)
    N ← length of A
    for i ∈ 1...N do
        temp ← A[i]
        k ← i
        while k > 0 and A[k − 1] > temp do          ▷ Corrected from A[k] < A[k − 1]
            A[k] ← A[k − 1]
            k ← k − 1
        A[k] ← temp
```

Problem 3.4 (4 points)**.** What is the invariant that the algorithm maintains on $A[0 \dots i]$ before each time the main loop executes? Explain how this invariant differs from that in selection sort. How does this explain why i starts at 1 rather than 0 as in selection sort?

Solution: As with selection sort, one invariant is that the values $A[0 \dots i]$ are sorted on any loop iteration. However, with insertion sort, they represent the first i elements in A in sorted order. We can start at $i = 1$ because this invariant automatically holds for $A[0 \dots 1]$, which consists of only the first element of the array.

Problem 3.5 (6 points)**.** We stated above that insertion sort was a comparison-based sort. However, the implementation shown here is written in terms of array accesses. Re-write the loop of the algorithm to use only compare and exchange operations.

Solution: One example of a possible solution written in pseudocode is as follows.

Algorithm 3 Insertion sort

```
function InsertionSort(A)
    N ← length of A
    for i ∈ 1...N do
        k ← i
        while k > 0 and A[k − 1] > A[k] do
            swap A[k − 1] and A[k]
            k ← k − 1
```

Problem 3.6 (8 points)**.** Unlike selection sort, the number of operations performed by insertion sort depends on the data. Calculate the minimum and maximum number of comparisons and exchanges when sorting an array of length N, and describe under what conditions they are achieved.

Solution: The best case is when the array is sorted, in which 0 exchanges are performed, and one comparison is performed for each loop iteration, or $N - 1 \in O(N)$. This is why insertion sort is often a good choice for data that is nearly sorted.

When the array is sorted in reverse order, however, we see that for a loop iteration with some value of i, that i comparisons and exchanges are performed on that iteration. The total number of comparisons and exchanges is then $1 + 2 + \cdots + (N - 2) + (N - 1) = \frac{N(N-1)}{2}$, which is the worst case.

Problem 3.7 (4 points)**.** Compare the number of comparisons and exchanges performed using selection sort and insertion sort. Under what conditions does insertion sort perform more or less comparisons than selection sort?

Solution: The number of comparisons performed by insertion sort, which ranges from $N-1$ to $\frac{N(N-1)}{2}$, is always less than or equal to the number performed by selection sort, which is $\frac{N(N-1)}{2}$. On average, insertion sort performs half as many comparisons as selection sort, making it generally the better choice.

4 Linearithmic sorts

In this section, we turned to linearithmic sorting algorithms, which perform faster than the ones we have studied previously. In particular, we considered merge sort which may have been unfamiliar to many students, with the goal of introducing the divide-and-conquer paradigm used in later on in MSD radix sort.

4.1 Merge sort

In merge sort, we divide the problem into two sub-problems of equal size. The obvious way to accomplish this is to simply divide the input array into the first and second half at the midpoint of the array, which is constant time. We then recursively sort each half, and then merge the two sorted half-arrays into a single sorted array. The pseudocode below outlines this process.

Algorithm 4 Merge sort

```
function MergeSort(A)
    mid ← N/2
    MergeSort(A[0 ... mid])
    MergeSort(A[mid ... N])
    return merge(A[0 ... mid], A[mid ... N])
function Merge(A, B)
    // Your code here                                  ▷ refer to problem 4.1
```

Problem 4.1 (8 points)**.** Write the pseudocode for the merging procedure. Your pseudocode should be a function called "merge" that takes two sorted arrays as input and outputs a single, sorted array whose length is the sum of the length of the input arrays. Your implementation of the merging procedure must run in $O(N)$ or better.

Solution: One possible solution written in pseudocode is provided below.

Algorithm 5 Merging procedure

```
function Merge(A, B)
    i ← 0
    j ← 0
    N ← the length of A and B
    C ← empty array of length 2N
    while i < N or j < N do
        if j = N or (i < N and A[i] ≤ B[j]) then
            C[i + j] ← A[i]
            i ← i + 1
        else                                            ▷ i = N or A[i] > B[j]
            C[i + j] ← B[j]
            j ← j + 1
    return C
```

Problem 4.2 (4 points)**.** What is the space complexity of the merging procedure? Is it in-place?

Solution: Since we need to allocate an auxiliary array of size N, the space complexity of the merging procedure is $O(N)$, and therefore it is not in-place.

Problem 4.3 (6 points)**.** Let $T(N)$ be the number of comparisons needed to sort an array of size N. Show that

$$T(N) = 2T\left(\frac{N}{2}\right) + O(N) \tag{1}$$

Solution: Using the "divide and conquer" metaphor, the first step in mergesort is to *divide* the problem into sorting two subarrays of size $\frac{N}{2}$, which is represented by $2T\left(\frac{N}{2}\right)$. After this, we *conquer* by merging both subarrays into a larger array of size N, which is represented by $O(N)$.

Problem 4.4 (8 points)**.** Assume that $T(\frac{N}{2}) = \frac{N}{2}\log\frac{N}{2}$. This is known as our inductive hypothesis. Show that $T(N) = O(N \log N)$ by substituting the expression into equation (1).

Solution: First, so that we only need to deal with integers, assume that N is a power of two. Given the response to the previous question, we see that $T(N) = 2T\left(\frac{N}{2}\right) + N$ exactly, replacing the $O(N)$ with an exact quantity N. Furthermore, then we can assume that the base of the logarithm expression is 2, because we always split the array into two halves. We proceed as follows:

$$\begin{aligned}
T(N) &= 2T\left(\frac{N}{2}\right) + N \\
&= 2\left(\frac{N}{2}\log_2\frac{N}{2}\right) + N \\
&= N\left(\log_2\frac{N}{2}\right) + N \\
&= N(\log_2 N - 1 + 1) \\
&= N\log_2 N
\end{aligned}$$

Thus, we can conclude that $T(N) \in O(N \log N)$, or in other words, the function $T(N)$ has order of growth $O(N \log N)$. Although we did not need to determine the exact value of $T(N)$, which required making additional assumptions above, it helps clarify the process.

Problem 4.5 (4 pionts)**.** Our proof appears to only account for N that are powers of two. Explain why we can generalize this result to hold for all positive integers N.

Solution: To account for this, we can round up N to the next highest power of 2 while retaining the same order of growth, $O(N \log N)$. This discrepancy is one reason we might prefer to state the order of growth of $T(N)$ rather than attempting to determine an exact value.

5 Radix Sort

The intention of this section was to expose teams to a completely new idea, building upon ideas developed in the previous sections. Many teams found the problems difficult, and we attempted to award as much credit as possible if solutions were generally on the right track: it was not necessary to get all of the details right.

5.1 Key-Indexed Sorting

Problem 5.1 (4 points)**.** Write pseudocode to count the number of students in each grade. Your pseudocode should be a function that takes an integer array as input and outputs an integer array of length four. The contents of the input array consists entirely of 9, 10, 11, and 12. The first entry of the output array should be the number of 9th graders, the second the number of 10th graders, and third the number of 11th graders, and the fourth the number of 12th graders.

Solution: One possible solution written in pseudocode is provided below.

Algorithm 6 Counting students in each grade

```
function CountStudents(A)
    C ← empty array of length 4
    for i = 0 ... N do
        index ← A[i] − 9
        C[index] ← C[index] + 1
    return C
```

Problem 5.2 (4 points)**.** Supposed that the number of 9th, 10th, 11th, and 12th graders are c_1, c_2, c_3, and c_4 respectively. What are the bounds on possible indices for 9th, 10th, 11th, and 12th graders in the auxiliary array? (In other words, between which two indices must all 9th graders be in the auxiliary array, and same for 10th, 11th, and 12th graders?)

Solution: The indices corresponding to each grade level are as follows.

- All 9th graders must be in indices $0 \dots c_1$.
- All 10th graders must be in indices $c_1 \dots (c_1 + c_2)$.
- All 11th graders must be in indices $(c_1 + c_2) \dots (c_1 + c_2 + c_3)$.
- All 12th graders must be in indices $(c_1 + c_2 + c_3) \dots (c_1 + c_2 + c_3 + c_4)$.

Problem 5.3 (6 points)**.** Based on your response to problem 4.2, write pseudocode to sort the 9th, 10th, 11th, and 12th graders using key-indexed sorting. Your pseudocode should be a function that takes as input an integer array and outputs an integer array. Each entry in the input array is either a 9, 10, 11, or 12. The output array should be the input array in sorted order. Your code may call the function KeyIndexSort and assume that it works correctly, regardless of whether your solution to problem 5.1 is correct.

Solution: One possible solution written in pseudocode is provided below. Note that D can also be calculated using a loop.

Algorithm 7 Counting students in each grade

```
function KeyIndexSort(A)
    N ← length of A
    out ← empty array of length N
    C ← CountStudents(A)
    D ← empty array of length 4                    ▷ determine starting indices in out
    D[0] ← 0                                       ▷ from problem 5.2
    D[1] ← C[0]
    D[2] ← C[0] + C[1]
    D[3] ← C[0] + C[1] + C[2]
    for i = 0 ... N do
        key ← A[i] − 9                             ▷ determine the index in D
        out[D[key]] ← A[i]                         ▷ D[key] is where to put the next value with key
        D[key] ← D[key] + 1
    return out
```

Problem 5.4 (8 points)**.** For most problems, however, we are required to sort items into more than just four categories. Generalize your algorithm in problem 4.3 so that it sorts an array of R different values using key-indexed sorting. R, the number of unique keys (or "digits"), is known as the *Radix*. Your pseudocode should be a function that takes as input an integer array and integer R, and outputs an integer array. The input array should consist of integers between 0 and $R-1$, and the output array should be the input array in sorted order.

Solution: One possible solution written in pseudocode is provided below. The difference between this and the previous problem is that we no longer have a convenient *CountStudents* function at our disposal, though assuming the presence of an analogous function would have been acceptable. Furthermore, we can no longer hardcode the calculation of D, and must use a loop.

Note that for complete generality, we include a function *GetKey*, which we can use to determine a key between 0 and $R-1$ corresponding to an array element. Since the problem specifies that the values themselves are keys, this is not necessary, but it is worth noting that this is not necessarily the case. For example, in the previous problem, the key was determined by subtracting 9 from the value.

Algorithm 8 Generalized key-indexed sort

```
function KeyIndexSort(A, R)
    N ← length of A
    out ← empty array of length N
    C ← empty array of length R                    ▷ count frequency of different values
    for i = 0 ... N do
        key ← GetKey(A[i])
        C[key] ← C[key] + 1
    D ← empty array of length R                    ▷ determine starting indices in out
    D[0] ← 0
    for i = 1 ... R do                             ▷ prefix sum
        D[i] ← D[i − 1] + C[i − 1]
    for i = 0 ... N do
        index ← GetKey(A[i])                       ▷ determine the index in D
        out[D[key]] ← A[i]
        D[key] ← D[key] + 1
    return out
```

Problem 5.5 (4 points)**.** What is the time complexity of key-indexed sorting?

Solution: The time complexity of key-indexed sorting is $O(N + R)$. We can see this from the solution to problem 5.4, because the loops in the algorithm execute N, R, and N times respectively, and the loop bodies all run in constant time.

Problem 5.6 (4 points)**.** What is the space complexity of key-indexed sorting. Is it in-place?

Solution: Because we need to allocate extra arrays of size N and R, the space complexity is $O(N+R)$, and therefore key-indexed sorting is not in place.

Problem 5.7 (8 points)**.** Prove that key-indexed sorting is stable. A semi-rigorous argument can be enough to get full credit.

Solution: Consider two elements in A that have the same key. Since they have the same key, they will fall into the same "bucket" (represented by $D[key]$ above). Since elements are processed in order of their appearance in the initial array, they will be inserted into the bucket in the same order in which they appear initially.

5.2 Sorting Strings

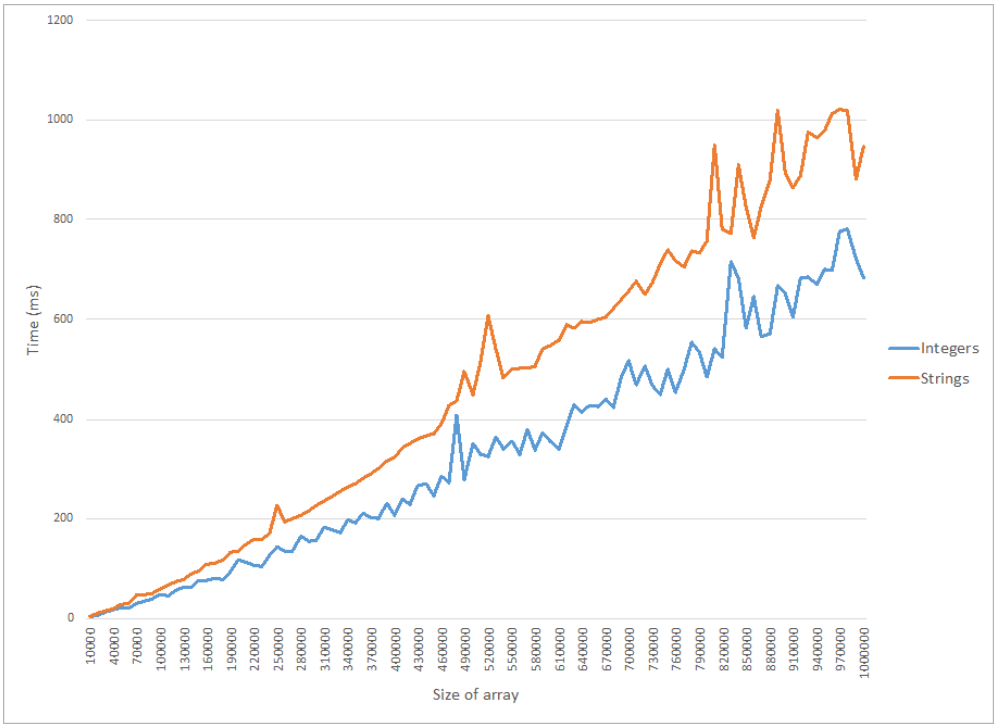

Figure 1: Merge sort on integers and strings.

Problem 5.8 (6 points)**.** Previously, we considered comparison-based sorts, which we used to sort integers in $O(N \log N)$ time. The above graph shows the results of using merge sort to sort integers and strings of length 10. Give an explanation for why sorting strings takes significantly longer than sorting integers. What is the runtime complexity of sorting N strings, each with length up to K?

Solution: When using a comparison-based sort, we must account for the time required to compare two strings. If the length of the strings is bounded by K, then comparing two strings can be done in $O(K)$ time. Since mergesort is $O(N \log N)$ in the number of comparisons, and we need to account for an $O(K)$ runtime for each comparison, then running mergesort would take $O(NK \log N)$ time overall.

5.3 Least Significant Digit Radix Sort

Note: The provided implementation of LSD assumed an implementation of $KeyIndexSort$ with a third argument $buffer$ that specified the output array. This is inconsistent with the implementation of $KeyIndexSort$ provided previously; we regret any confusion this may have caused. Furthermore, R was used instead of conflated M, the length of the strings.

Below, we have provided an updated version that assumes a similar implementation of $KeyIndexSort$ to the one provided previously, but with *index* as an additional argument. The value of *index* would be passed to a *GetKey* function such as the one indicated below, in order to sort strings by the character located at that position.

Algorithm 9 LSD Radix Sort

```
function GetKey(S, R, index)
    return S[index] as an integer in [0, R)
function LSDRadixSort(A, R)
    M ← length of strings in A
    index ← M − 1                                   ▷ changed from R − 1
    while index ≥ 0 do
        A ← KeyIndexSort(A, R, index)               ▷ changed from KeyIndexSort(A, index, buffer)
        index ← index − 1
    return A
```

Problem 5.9 (8 points)**.** Given a list of 15 strings of length 4, sort the strings by tracing through Algorithm 3. Write down the contents of the array after each call of KeyIndexSort.

```
BEAD
AAEB
BADA
EBAC
BECE
ECAD
BECC
ACAC
CBDD
DECD
CADD
EEAA
CABB
AADE
ACAC
```

Solution: The partial results are shown in each column. The rightmost column contains the final, sorted list of strings.

BADA	EEAA	CABB	AADE
EEAA	EBAC	BADA	AAEB
AAEB	ACAC	CADD	ACAC
CABB	ACAC	AADE	ACAC
EBAC	BEAD	AAEB	BADA
BECC	ECAD	EBAC	BEAD
ACAC	CABB	CBDD	BECC
ACAC	BECC	ACAC	BECE
BEAD	DECD	ACAC	CABB
ECAD	BECE	ECAD	CADD
CBDD	BADA	EEAA	CBDD
DECD	CBDD	BEAD	DECD
CADD	CADD	BECC	EBAC
BECE	AADE	DECD	ECAD
AADE	AAEB	BECE	EEAA

Problem 5.10 (6 points)**.** We will now show that LSD Radix Sort is correct for strings of any length using induction. Assume that we are able to sort strings of length k using LSD Radix Sort. Show that this implies that we can also sort strings of length $k+1$ using LSD Radix Sort.

Solution: For our induction hypothesis, we assume that LSD Radix Sort is able to sort a list of strings of length k. Suppose we want to sort a list of strings of length $k+1$; we begin by first sorting the last k characters of each string using LSD Radix Sort. We know that this works by our inductive hypothesis.

Now, we use key-indexed sort on the first character of each string. If we consider the set of all strings with a given first character, we see that they must be sorted by our induction hypothesis, and because key-indexed sort preserves stability. Furthermore, all such sets of strings must appear in order, because we just sorted by the first character. Thus, the entire list of strings of length $k+1$ is sorted.

Problem 5.11 (4 points)**.** Prove that LSD Radix Sort is $O(MN)$, where M is the length of the longest string.

Solution: We must pass over each item in the list in order to determine which bucket it is in, for a total of N operations per pass. We must pass over the list once for each character, and since the longest string is of length M, we must pass over the list M times. Since we pass over the list M times, with each pass consisting of an operation on N words, the total complexity is $O(MN)$.

Note that since we also process an array of length R on every pass, the actual complexity would be $O(M(N+R))$, though this does not matter when R is small.

Problem 5.12 (4 points)**.** Is LSD Radix Sort stable? If yes, explain why. If not, give a counterexample.

Solution: We have previously shown that key-indexed sort is stable. Since LSD Radix Sort simply consists of repeated key-indexed sorts, we can conclude that LSD Radix Sort is stable.

Problem 5.13 (8 points)**.** (a) Explain (and give an example) of why key-indexed sorting from right to left will not work if we need to sort strings of different length (4 points). (b) Explain how LSD Radix Sort can be used to sort a list of strings if not all strings are of the same length (4 points).

Solution:

(a) The least significant digit will vary in location depending on the length of the list. For example, consider a list of 2 strings, "E" and "AZ". Clearly, when sorting alphabetically, "AZ" would come before "E". However, when sorting by least significant digit, if we compare "E" and "Z", then we would end up sorting "E" before "AZ", which is likely not the behavior we want.

(b) If we would like to sort a list of strings of different lengths, we must pad the end of all shorter strings with spaces so that all strings are of the same length. We can use other characters as well, as long as they sort before the other characters that we use lexiographically.

Problem 5.14 (6 points)**.** Above, we used LSD Radix Sort to sort strings with an alphabet of 128 ASCII characters. Explain how LSD Radix Sort can be used to sort a list of 64-bit unsigned integers.

Solution: A 64-bit unsigned integer can be broken down into a string of eight bytes, from most significant to least significant (where 1 byte = 8 bits). By custom, such integers are left-padded with 0s if they require less than 64 bits to represents. This would allow us to sort a list of such integers using LSD radix sort with a radix of $R = 2^8 = 256$.

We can also break it down in other ways: for a chunk size of c bits, we have $R = 2^c, M = \frac{64}{c}$. Recall that the space complexity is $O(N+R)$ and the time complexity is $O(M(N+R))$. This means that large values of c are impractical, because R will be too large; on the other hand, we can get with as low as $c = 1$, which would be splitting on every bit.

Problem 5.15 (6 points)**.** Although LSD Radix Sort has a better big-O complexity than Quicksort and Merge sort ($O(N)$ vs. $O(N \log N)$), it sometimes performs slower. Explain why this might be the case. (Hint: What aspect of the actual time complexity does big-O notation not tell us?)

Solution: Though big-O notation tells us the order of growth of the running time of an algorithm, it does not tell us the associated constant factor. In LSD Radix Sort can be quite significant: examining key-indexed sort, we see that we need to access a string in the array $2M$ times, where M is the length of the maximum string.

In contrast, merge sort performs approximately $\log N$ times. Though comparisons may need to access up to M characters, they will not require that many accesses in the majority of cases. Furthermore, the process of comparing two strings linearly benefits from memory locality, so is generally faster than accessing single characters from different strings.

5.4 Most Significant Digit Radix Sort

Problem 5.16 (10 points)**.** Write the pseudocode for MSD Radix Sort. Your pseudocode should be a function that takes a string array as input and outputs a string array. Your function may call a recursive helper-function if necessary.

Solution: One possible solution written in pseudocode is provided below. We have changed the *GetIndex* function to account for when *index* is not a valid index. Of course, this was not necessary to consider in the actual contest, since *GetIndex* did not exist then.

As another note, the calculation of C and D is duplicated from *KeyIndexSort* here, though providing a simple textual note would suffice for credit. One difference is that D must be length $R+1$ in order to be able to use $D[i]$ and $D[i+1]$ to represent the starting and ending indices of the subarray corresponding to the ith radix. Incidentally, if you are the first student to report the existence of this sentence to the TJ IOI Officers, you will be eligible to receive a free extra T-shirt at TJ IOI 2018.

Finally, note that since *MSDHelper* does not modify the input A, we need to copy the recursive output into a variable *temp*. Correctly handling such minor details is not necessary to obtain full credit.

Algorithm 10 MSD Radix Sort

```
function GetIndex(S, R, index)
    if index < Length(S) then
        return S[index] as an integer in [0, R)
    else
        return −1                                   ▷ empty char sorts before all others
function MSDHelper(A, R, index)
    if A is empty then return A
    A ← KeyIndexSort(A, R, index)
    C ← frequency count from KeyIndexSort
    D ← empty array of length R + 1                 ▷ D[i] represents an index in A
    D[0] ← 0
    for i = 1 ... R + 1 do                          ▷ prefix sum
        D[i] ← D[i − 1] + C[i − 1]
    for i = 0 ... R do                              ▷ each radix value has a subarray
        temp ← MSDHelper(A[D[i] ... D[i + 1]], index + 1)   ▷ sort each subarray
        copy temp into A[D[i] ... D[i + 1]]
    return A
function MSDRadixSort(A, R)
    return MSDHelper(A, R, 0)
```

Problem 5.17 (8 points)**.** Given 15 strings of arbitrary length at most 5, sort the strings by tracing through MSD Radix Sort. Write down the contents of the array after each key-index sort.

```
EDACD
BD
EDCEB
EDCA
EBE
E
DCCA
AEEAB
DDCD
ECCBA
AEEA
DAB
EABA
ECCA
EDC
```

Solution: The partial results are shown in each column. The rightmost column contains the final, sorted list of strings.

AEEAB	AEEAB	AEEAB	AEEAB	AEEA
AEEA	AEEA	AEEA	AEEA	AEEAB
BD	BD	BD	BD	BD
DCCA	DAB	DAB	DAB	DAB
DDCD	DCCA	DCCA	DCCA	DCCA
DAB	DDCD	DDCD	DDCD	DDCD
EDACD	E	E	E	E
EDCEB	EABA	EABA	EABA	EABA
EDCA	EBE	EBE	EBE	EBE
EBE	ECCBA	ECCBA	ECCA	ECCA
E	ECCA	ECCA	ECCBA	ECCBA
ECCBA	EDACD	EDACD	EDACD	EDACD
EABA	EDCEB	EDCEB	EDC	EDC
ECCA	EDCA	EDCA	EDCA	EDCA
EDC	EDC	EDC	EDCEB	EDCEB

Problem 5.18 (8 points)**.** What is the worst-case number of array accesses that MSD performs? What about the best case? In your answer you may refer to W as the length of the longest string.

Solution: The best-case number of accesses would be if every string differed in the first character. In this case, only $2N$ array accesses would be performed, as only one call to $KeyIndexSort$ would be needed. The worst case, on the other hand, would be if all strings were equal. This would require $2NW$ array accesses, as W calls to $KeyIndexSort$ would be needed.

Note that we only consider accesses to the underlying strings, and not in intermediate calculations; this was not made clear in the problem statement.

Problem 5.19 (4 points)**.** Explain why on average, MSD takes less time than LSD.

Solution: MSD needs to perform less comparisons than LSD, as evidenced by the response to the previous question; it only needs to examine enough characters in a string to determine its position relative to other strings, and no more. In contrast, LSD always examines all characters of every string.

As for other aspects, one issue with MSD is that recursion incurs some memory and temporal overhead in terms of function calls. On the other hand, MSD might benefit from memory locality, since it sorts progressively smaller arrays.

Problem 5.20 (8 points)**.** Explain bead sort in terms of Radix Sort. One reason for its supposed speed is that the beads in each column are permitted to fall simultaneously and independently. Why can we do this with bead sort, but not with LSD or MSD?

Solution: Bead sort can be thought of as a special case of radix sort, in which we represent positive integers in base 1; that is, as a string of 1s corresponding in number to the integer, followed by 0s to pad each integer to the length of the longest integer.

When we "release" the beads, so to speak, we are simultaneously performing a sort on every digit. However, in LSD or MSD, we cannot sort each digit concurrently, and so we must process one digit at a time, absent parallel computing.