# BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

## *CS-F211: Data Structures and Algorithms*

## Lab 7: More Sorting

### Introduction

Welcome to week 7 of Data Structures and Algorithms! In the previous labsheet, you have learnt in great detail about Quick Sort and its applications. A few weeks back, you have also learnt the ins and outs of Insertion and Merge sort. These sorting algorithms fall under the domain of "comparison-based" sorting algorithms. In this week's labsheet, we shall be discussing some of the most crucial "space-based" sorting algorithms extensively and we shall also see their use cases, the most crucial of which is the fact that by using them we can perform sorting in linear time. Hopefully, after this discussion, your understanding of sorting algorithms will be heightened to the point where you can make informed decisions about choosing a particular sorting algorithm for a particular task of sorting.

### Topics to be covered in this labsheet

- The Trade-off between Space and Time
- Counting Sort
- Radix Sort
    - Straight Radix Sort
    - Radix Exchange Sort
- Bucket Sort
    - Simple Bucket Sort
    - Interval Sort

## The Trade-off between Space and Time

You have learnt in the lectures that the asymptotic lower bound for any *strictly comparison-based* sorting algorithm is given by $\Omega(N\log N)$, where N is the size of the array that is being sorted. This means that any comparison-based sorting algorithm, no matter the details of what it is doing, cannot have a time complexity better than $N\log N$. While this is definitely true (and mathematically proven as well), we must note that the key phrase in the above proposition is *"comparison-based"*. So, can we not attempt to come up with an algorithm that is *not* strictly comparison-based and therefore bypasses the $N\log N$ asymptotic lower bound?

This discussion brings us to an important tradeoff that we frequently encounter in data structuring and algorithms theory – **space vs time**. We can sometimes gain time (ie, take less time to run our algorithm) if we simply make our algorithm use up more space in a meaningful way. In general, there is a tradeoff between time and space complexity. An algorithm that is optimised for time complexity may use more memory, and vice versa. The tradeoff between time and space complexity can be leveraged to improve the performance of an algorithm. If an algorithm has a high time complexity but is running on a system with a lot of memory, it may be possible to improve its performance by using more memory to reduce the time complexity. On the other hand, if an algorithm has a high space complexity but is running on a system with limited memory, it may be possible to improve its performance by reducing the space complexity, even if this means increasing the time complexity.

In practice, the decision of whether to optimise for time or space complexity will depend on the specific application and the resources available. It's important to evaluate the trade-offs and choose the best approach for the given situation. To be in an educated position to make this decision, one must be well-versed with both stratagems to approach problems. The best example illustrating this trade-off is that of *caching*, which is done at the hardware level and comes under the domain of Computer Architecture and Operating Systems design. The best algorithmic examples utilising this trade-off are those of *hashing*, which falls in the purview of the next labsheet, and of *space-based sorting algorithms*, which make up the subject matter of this labsheet.

## Counting Sort

Suppose that we know some information about the elements that we are sorting. We know that each of the **n** input elements is an integer in the range {0, 1, 2, …, k-1}. In our algorithm, we first determine for each input element **x** the number of elements that are less than **x**. Given this information, we can then directly place the element **x** into its designated position in the output array. For example, if we have figured out that there are 10 elements below **x**, then we can say with certainty that **x** will be situated at position 10 in the sorted array (since positions 0 through 9 will be occupied by those 10 elements that are smaller than **x**). This is the basic premise of counting sort (or "frequency sort"). We *count* the number of elements that are there for each integer in the range (that is known to us). Then we use this count to directly place every element into its sorted positions in the output array.

Suppose our input array is **A** of length **n**; and our output array is **B** of length **n**. Knowing that our range of integers is 0 through k-1, we create another array C wherein we store the outcomes of our "counting" operations. This means that if `C[i] = a`, then there are **a** number of elements having value equal to **i** in the array A. Thus, ultimately, we can run a loop through array C and correspondingly store that many elements in their corresponding order into the output array B.

Now, say there are three 1's and two 2's in our array. In this case, the sorted position of the two 2's can only be *after* the sorted position of the three 1's preceding it. But the array C so far only stores the *count* of 2's. To obtain the required position, we must therefore compute the running sum of our elements in the C array. The *running sum* upto a position *i* in an array is the sum of all the elements that are at an index lesser than or equal to *i*.

After the running sums are computed for every element, our sorting can be completed by running a single loop from the right end to the left end of array A. In this loop, we observe the $(n-1)^{th}$ or rightmost element of A and find out from array C (since it now has the running sums) what the sorted position of our $(n-1)^{th}$ element should be, and so we place that element from array A into its correct position in array B (the output array). Once this is done, in array C we decrement the "count" (or, the sorted position) of the element we just placed. This is done so that if this key appears once again, we can place it just to the left of the earlier key (and that there is no clash).

Then, we move to the next element (second rightmost, or $(n-2)^{th}$) in array A, find its sorted position from array C, place it at that position in array B, decrement the value in array C, and move on to the next element. The same is done till we have iterated through the whole array A. And the result of this is the sorted array B. [You may wonder why this loop is run in reverse order, ie., from right to left; this is not an arbitrary choice, and we shall soon see why it is so.]

Consider the example given in figure 1 that shows you the state of each relevant array during the execution of the algorithm explained above on the sample input {2, 5, 3, 0, 2, 3, 0, 3}.

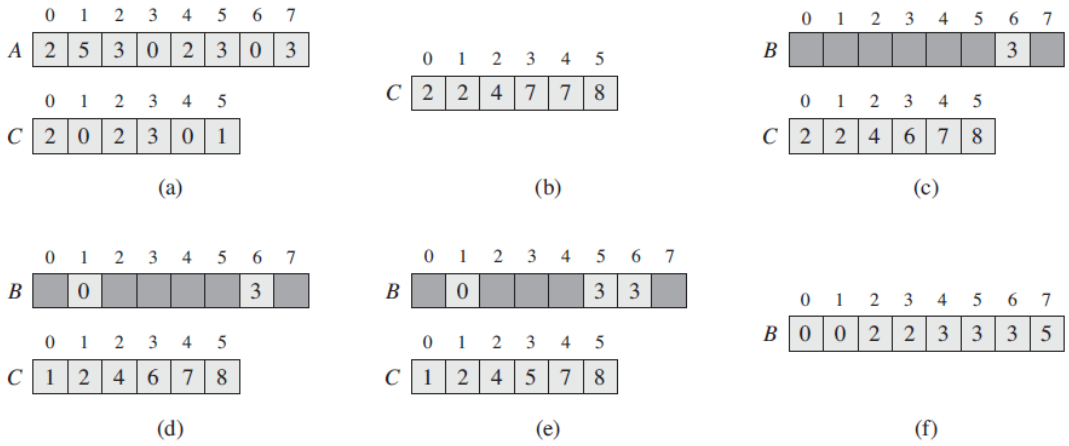



Figure 1: Illustrating counting sort with an example

(a) Arrays A and C after performing the counting operation on every element in array A and storing it in C. (b) Array C after adding up the counts of every element below an element (ie, computing the running sum), for all elements. (c)-(e) Arrays B and C after one, two, and three iterations of the final loop to populate the array B from C; only the lightly shaded elements of B have been filled in. (f) The final sorted array B which is our desired output.

Let us see how this algorithm is implemented programmatically.

```
int* counting_sort(int* A, int* B, int k, int n)
{
    // Initialize array C with all 0s
    int C[k];
    for (int i = 0; i < k; i++)
    {
        C[i] = 0;
    }

    // Count the number of times each element occurs in A and store it in C
    for (int j = 0; j < n; j++)
    {
```

```
        C[A[j]]++;
    }

    // Place the elements of A in B in the correct position
    // by computing the running sum
    for (int i = 1; i < k; i++)
    {
        C[i] = C[i] + C[i - 1];
    }
    for (int j = n - 1; j >= 0; j--)
    {
        B[C[A[j]] - 1] = A[j];
        C[A[j]]--;
    }
    return B;
}
```

Notice how we ran our final loop from right to left. This is to ensure the **stability** of our sorting algorithm. Counting sort is stable precisely due to the manner in which we are performing our last for loop; it is in reverse order, from n-1 to 0. If this loop were to be performed from 0 to n-1, counting sort would lose its property of stability. [Stability in sorting refers to the condition in which the relative positions of elements having the same value remain *unaltered* after sorting.]

Since we are only performing three separate (non-nested) loops, two of them running n times while one runs k times, the time complexity of our algorithm is O(n+k). However, since we are also storing k extra integers in memory, our auxiliary space complexity is also O(n+k). When k = O(n), then both our time and space complexities become O(n).

You have been provided with a source code file *counting_sort.c* which implements the above counting sort program and also provides a driver code to run the same by prompting the user for input. Feel free to play around with it and solidify your understanding of the algorithm.

Task 1: Given a string **str** of lowercase English alphabetical characters, arrange all the characters of the string in increasing order of their ASCII value in O(n) time, where n is the length of the string (you may ignore the string termination character). Use counting sort to do this. You can take **str** directly as input from the user.

Task 2: Write a program that, given **n** integers in the range 0 to k, preprocesses its input and then allows the user to query (or pick) one of the entered integers. Your program should now spend O(1) time to output whether the picked integer falls into the range [a … b]; where **a** and **b** are also inputs that were given by the user before preprocessing. Your program should use no more than Θ(n+k) preprocessing time. You can read the **n** integers from the file *n_integers.txt* that is given to you. This file contains arrays containing n integers (the arrays are on separate lines), each array is also preceded by a number indicating the length of that array [Hint: Can you tweak the *counting* approach from counting sort to solve this?]

Home Exercise 1: A school is trying to click the yearbook photograph of every class. The students have been asked to stand in one single-file line in *non-decreasing order* by height. Let this ordering be represented by the array **expected**, where **expected[i]** is the height of the student who is expected to be at the **i**$^{th}$ position in the line (Note: zero-indexing is used). But the students are very chaotic and are not able to form the line correctly. At this point in time, the height of a student at some position is given by the array **current**, where **current[i]** is the height of the student who is currently at the **i**$^{th}$ position in the line (Note: zero-indexing is used). Write a function that takes this array (**current**) as input and computes in O(n) time the number of students who are *out-of-place*. This means that your function should compute the number of indices for which **current[i] != expected[i]**. You have been given some sample **current** arrays in the *heights.txt* file. These arrays are each present in separate lines, preceded by a number indicating the length of the array.

Home Exercise 2: The "h-index" of a researcher is a metric that is often used to determine the quality of the research output produced by the researcher. Although there are several flaws and problems with the metric, it is still widely used. You can see it by going to the Google Scholar page of any famous researcher (for example, try googling "Thomas Cormen Google Scholar" and then visiting his scholar page).

The h-index is defined as follows: a researcher has an h-index **h** if **h** out of their **n** papers have at least **h** citations each, and the other **(n − h)** papers have less than **h** citations each. If there are several possible values for **h**, the maximum value is taken as the researcher's h-index. Refer to Figure 2 for a graphical depiction of the index.

For a particular researcher, given the array of integers **cit**, where **cit[i]** is the number of citations the researcher received for their **i**$^{th}$ paper, you must compute the researcher's h-index by using the **"counting"** concept that has been explained in the above section. Write a function **`int hIndex(int* cit, int cit_length)`** to perform the described task. Your function should run in **linear time**. Test your function by passing sample inputs to it. These can be taken from the

*h_indices.txt* file, in which you have been provided with some randomly generated **cit** arrays in different lines. The arrays themselves are preceded by a number specifying the length of the array.

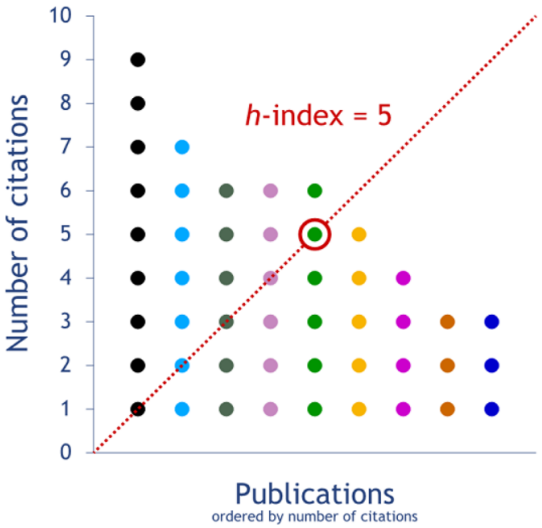



Figure 2: Graphically obtaining the h-index of an example researcher

## Radix Sort

Suppose, for example, that we want to sort items with keys that are pairs **(x, y)**, where **x** and **y** are numbers. In this case, it is natural to define an ordering on these items using the lexicographical (dictionary) convèntion, where **(x1, y1) < (x2, y2)** is true if either *one* of the following conditions holds:

- x1 < x2
- x1 == x2 and y1 < y2

This is a pair-wise version of the comparison function. It is easy to generalise the above to an n-wise comparison function for any n>2. We must first understand that this is precisely what we do for performing lexicographic comparisons and that there is no magic involved here.

Examples of this include sorting “strings” in dictionary order (we look at the first character, and only in case of a tie move to the next, and so on), sorting “timestamps” (we first look at the hour hand, and only in case of a tie move to the minute hand, and only in case of a tie there we move to the second hand), and so on.

The most important realisation with regard to the radix sort approach is that we can actually perform n-wise lexicographical comparisons based on the different digits of a number. In an n-digit number, the most-significant digit (leftmost) has the highest place value, followed by the second most-significant digit (second from left), and so on. This means we can perform some kind of lexicographical sorting on the digits of the number. This is what is known as “radix sort”. There are two distinct ways of performing this “lexicographical” (or, radix) sorting on an array of integers.

**Straight Radix Sort**

Even though radix sort is a type of lexicographical sorting only, in the straight radix sort approach we do something counter-intuitive — we start sorting from our least significant digit and move leftward. So, we start sorting from the rightmost digit and we must do so in a “stable” manner. Once we are done sorting one digit column, we move to the next digit column on the right and stably sort it, and so on. This process is illustrated in Figure 3.

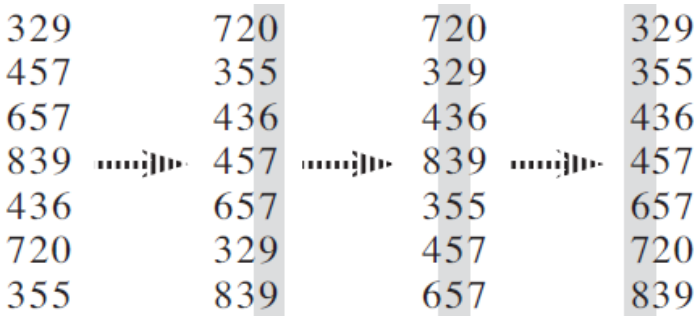



Figure 3: Illustrating straight radix sort algorithm

When we have iterated through all the digit columns of our keys, our sorting is done. Therefore, the task at hand is to perform the stable sort efficiently (it has to be stable because we should not be allowed to arbitrarily reorder the keys if they have the same digit at a particular digit column, we can only reorder them by looking at the next higher place value, which will be done only afterwards). And to do this, we shall use counting sort (recall that it has the property of stability). But there is a slight catch here. Notice that counting sort, as we have seen/implemented earlier, simply takes n integers as input and sorts them. But that is not quite our goal here. Here, we need to reorder an array of n integers based on a particular digit. Therefore, our counting sort routine needs to be modified slightly to facilitate radix sort, although the algorithm is still exactly the same.

We may summarise the straight radix sort algorithm as follows:

1. Take as input an array of 'n' integers.
2. Find the maximum of those integers and count the number of digits in that maximum integer. Let this be 'b'.
3. For i = 1 to b, perform stable sorting on the keys based on their $i^{th}$ digit from the right.

The time complexity of straight radix sort is $O(bn)$, where b is the maximum number of digits in the integers being sorted, which is typically a constant.

Task 3: Write a new counting sort function that takes as input the array, its size, and another field called "place". This "place" parameter holds the place value of the digit on which the counting sort needs to be performed (for example, place value of the hundred's digit is 100) and based on which, the entire array needs to be reordered. This function should not return anything, it must modify the array in place. You may write this function by adding it to the *counting_sort.c* that we have seen earlier.

Once the modified counting sort function is created, write a radix sort function that takes as input an array of integers, the size of the array, the number of digits, and an output array which

will hold the final sorted sequence; it performs straight radix sort on the array. The radix sort function must use the counting sort function inside it (you can write the radix sort function in a file named *radix.c* and you can "include" the *counting_sort.c* file as a header for this). You may read the input integers from *n_integers.txt* (similar to Task 2).

**Radix Exchange Sort**

Suppose that the keys we need to sort are represented in binary. Then as per the radix exchange sort algorithm, we examine the leftmost bit of the keys and sort the array with respect to the current leftmost bit. To perform this sorting, we do something very similar to the quick sort "partition" technique, and this is what is known here as "exchange".

To perform the exchange, we maintain two pointers, one scanning downwards from the top, and another scanning upwards from the bottom. We exchange the keys (note, entire keys and not just the bit) when our downward-moving pointer finds a 1 and our upward-moving pointer finds a 0. We stop scanning once our pointers coincide or cross. This process is illustrated in Figure 4.

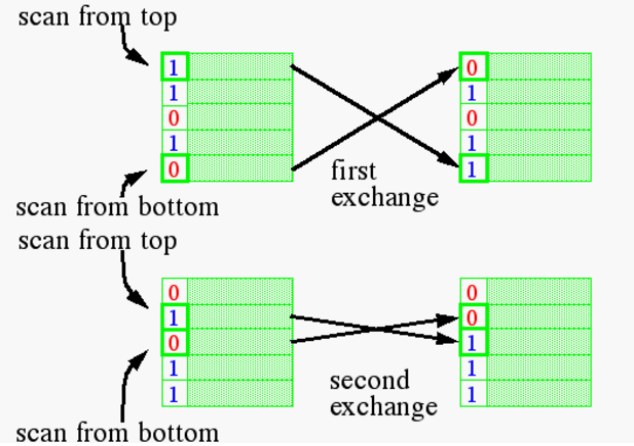



Figure 4: Illustrating the "exchange" procedure in radix exchange sort

Once we have performed all the exchanges on the leftmost bit, we now have two partitions in our array (zeroes on top and ones on bottom). So, now we recursively perform the same exchange partitioning on the two partitions ignoring the leftmost bit (the bit we just performed the exchange on). Once we are done with this up to the rightmost bit, our array is sorted.

The algorithm for radix exchange sort is summarised as follows:

*Note that we deal with the binary equivalent of the numbers in the input.*

1. Sort the array with respect to the leftmost bit by performing exchanges wherever a mismatch is found, in the manner described above.
2. Partition the array after all required exchanges are performed.
3. Recurse into the two partitions, with a mechanism to ignore the leftmost bits based on which the exchanges have been performed so far. Recursively perform the same operations till you have sorted (performed exchanges) based on the rightmost bit.

The time complexity of radix exchange sort is also $O(bn)$, where b is the number of bits in the numbers being sorted, which is typically a constant. Note that radix exchange sort is also sometimes referred to as "binary quick sort" (guess why).

Task 4: Use the algorithm described above to implement a function that performs radix exchange sort on an array of integers. Add this function to the earlier *radix.c* file in which you have made the straight radix sort function. Similar to the previous task, you may read the input integers from *n_integers.txt*.

Home Exercise 3: Write a function that takes an array of integers as input and returns the **maximum difference** between two of its successive integers in its sorted form. Your function should operate in linear time and use linear auxiliary space. You can read the input array of integers from the *n_integers.txt* file that is given to you.

Home Exercise 4: **IPv4** is the fourth version of the Internet Protocol. It is a widely used protocol that provides the foundation for communication between two devices over the internet by unique identification of each device on the internet. It is made up of four 8-bit numbers (therefore 32 bits in total), written in a dotted decimal notation in the format "*a.b.c.d*", where a–d are the four numbers in the address. For example, the Google public servers have IP addresses of 8.8.8.8 and 8.8.4.4.

Create a struct named **`IPaddr`** that holds four integers (a, b, c, and d above). Next, observe the file *IP.txt* that has been given to you. It contains a list of 1000 random IPv4 addresses. Read it and store all the addresses into an array of structs. Implement the straight radix sorting principle to sort the array of IP addresses in a lexicographical manner.

For example, the address 125.99.99.99 should appear before 130.13.13.13 in the sorted array, and the address 169.255.0.0 should appear after 169.253.255.255, and so on.

*Additional note: Can you use **char** instead of **int** for the four attributes a, b, c, and d of IPaddr? Try to think of a justification for doing so.*

## Bucket Sort

### Simple Bucket Sort

Bucket sort (or Bin sort) is not very different from counting sort, as we shall see. Suppose we know that the input consists of **n** integers in the range {1, 2, 3, …, **m**}. In that case, we shall create **m** buckets, one for each potential integer, in which we keep storing each of our **n** integers (the bucket can be implemented as a linked list). Thereafter, we can simply retrieve the elements back from the buckets in sequential order to obtain the sorted list. This takes O(m+n) time. An example of the same process is illustrated in Figure 5 on the input array {2, 1, 3, 1, 2}.

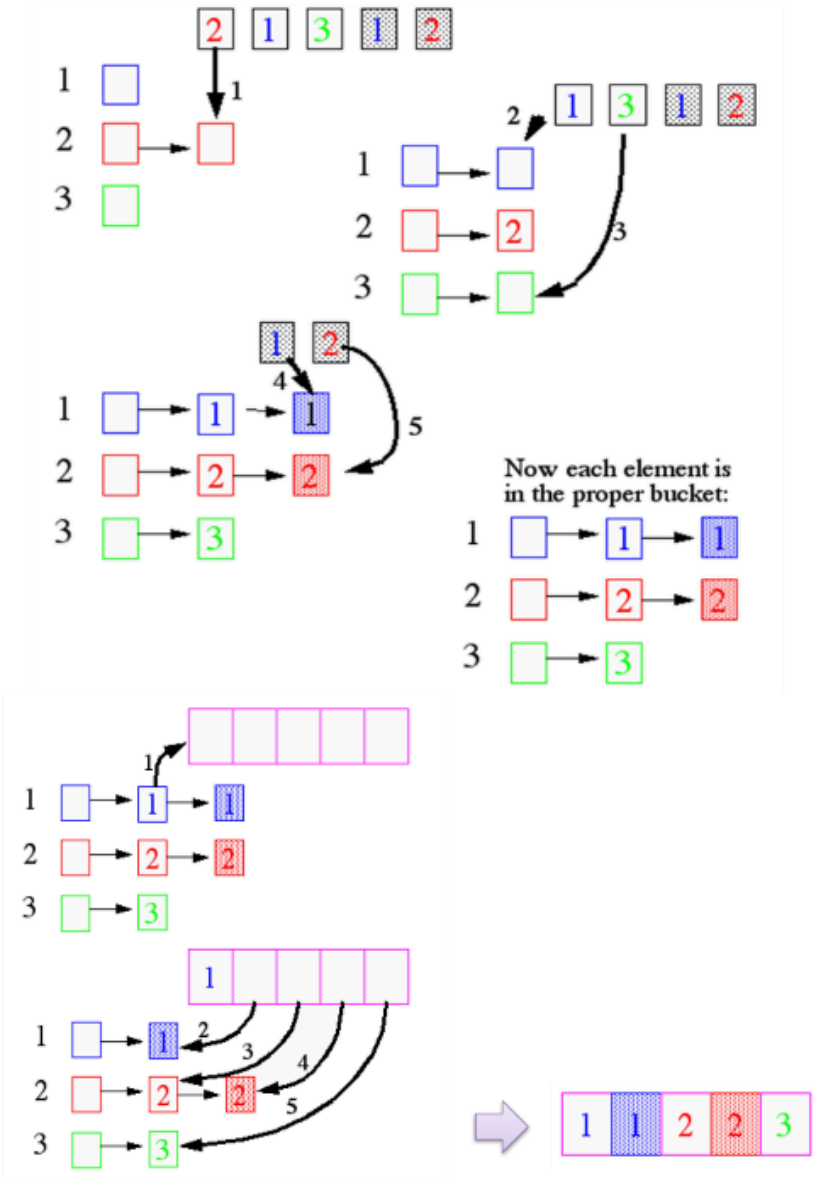



Figure 5: Illustrating simple/vanilla bucket sort

**Modified Bucket Sort (Interval Sort)**

The above algorithm is not very different from counting sort. More importantly, however, it fails to work effectively when the range of possible keys is too large, or when the keys involve real numbers. To handle these cases, there exists a more powerful version of bucket sorting known as "modified bucket sorting" or "interval sorting", which assumes that the input consists of **n** *real* numbers *uniformly distributed* (this is an assumption) over an interval **[a, b)**. This variant of bucket sort divides the values into **m** equal-sized subintervals or "buckets", which it maintains in sorted order, and then distributes the **n** input numbers into those buckets.

Consider the following example (refer Figure 6) with the numbers `29, 25, 3, 49, 9, 37, 21, 43`. They are all taken to be integers for simplicity's sake. Notice how we have distributed the eight numbers over five buckets, each of span 10. The numbers are internally sorted within these buckets. Once the distribution is complete and the internal order is maintained, all we need to do is retrieve all the elements one by one from the buckets in sequence (left to right, in this figure).

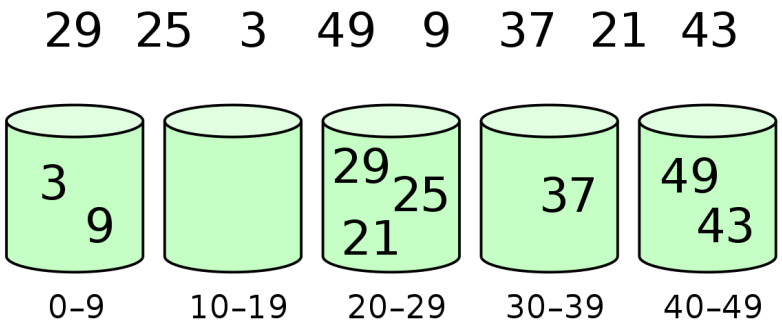



Figure 6: Example of Interval Sorting using integers

Now, we shall consider a more advanced example (refer Figure 7) to get an idea of the implementation details as well. In this example, we are actually dealing with real numbers in the range [0, 1); this is a more realistic scenario. Our numbers here are `.78, .17, .39, .26, .72, .94, .21, .12, .23, .68`. But the process we employ to sort these numbers is exactly the same. Note how we have created ten buckets (represented here by an array B of pointers pointing to a bucket), each of span 0.10. As we traverse through our original array, we keep distributing our numbers to their appropriate buckets. The buckets themselves are implemented using *linked lists*, and we use the insertion functions of the linked list to "insert in order" the numbers so that we maintain the sorted order required within these buckets. Once our distribution is complete, all we need to do is retrieve the elements (now sorted) one by one from the buckets in sequence (top to bottom, in this figure).

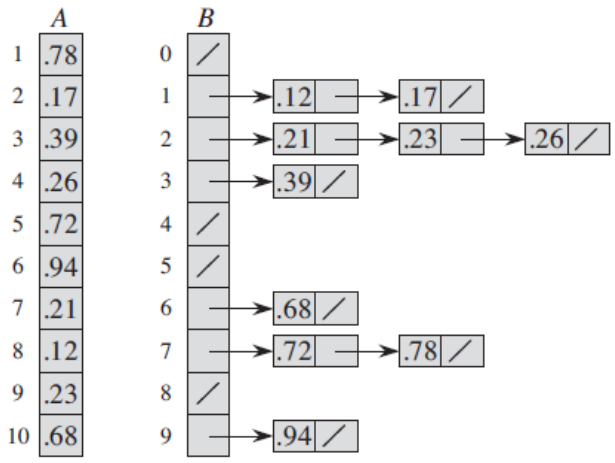



Figure 7: Example of Modified Bucket Sorting using real numbers

The time complexity of the modified bucket sort is $O(n^2)$ in the worst case, which is easy to rationalise. Imagine all the elements ended up falling into the same bucket. In that case, the time complexity of bucket sort reduces to the worst-case time complexity of insertion sort for n elements (or for the insert-in-order routine), which is $O(n^2)$. However, recall the assumption that we made about our keys here. We have assumed that they are *uniformly distributed* over the interval known to us (and this is a statistically realistic assumption). In this case, we can always compute an average case time complexity with which we can expect our algorithm to run with reasonable confidence. Computing this average case requires some mathematical rigour[1], therefore we simply take the result, which is $\Theta(n+m)$. If $m = O(n)$, then the average case reduces to $\Theta(n)$, thus achieving sorting in linear time, under reasonable conditions.

Let us take a look at how we may implement the modified bucket sorting program. Note that here our array elements are of type float. Let us assume that we create as many buckets as there are elements in the array. We first create an array of linked list structs, which is our array of buckets. Then, we create our linked lists for each such bucket. We then traverse our input array and one by one keep adding the elements to their appropriate bucket in our array of buckets. We use the insertFirst() function that we have developed in Weeks 1 and 2 to insert our elements into their buckets. We create a function sortList() that performs insertion sorting on a linked list, and then we use it to perform sorting on the individual buckets (it is an exercise for you to implement this function). Once that is done, we simply traverse the bucket array and each bucket (linked list) in sequence and add those elements to our array starting from the beginning. The array that was passed by reference is thus modified (sorted). The program for the same is as follows:

[1] Interested students may refer to pages 202-204 of Cormen T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009. Note that in their algorithm the number of buckets is taken to be exactly equal to the number of input elements, which we shall follow in our approach as well.

```
void intervalSort(float arr[], int n)
{
    int i, j;

    // Create n empty buckets
    LIST b[n];
    for(i=0; i<n; i++)
    {
        b[i] = createNewList();
    }

    // Put array elements in different buckets
    for(i=0; i<n; i++)
    {
        insertFirst(b[(int)(n*arr[i])], createNewNode(arr[i]));
    }

    // Sort individual buckets
    for(i=0; i<n; i++)
    {
        sortList(b[i]);    // sortList() function has to be implemented
    }

    // Concatenate all buckets (in sequence) into arr[]
    for(i=0, j=0; i<n; i++)
    {
        NODE temp = b[i]->head;
        while(temp != NULL)
        {
            arr[j++] = temp->ele;
            temp = temp->next;
        }
    }
}
```

Task 5: Use simple bucket sorting to solve the problem in Task 1.

Task 6: You are given a file *points.txt* that contains 1000 points in the xy-plane in the format **(x,y)**. These points are **uniformly distributed** within the unit circle centred at the origin, ie., $0 \leq x_i^2 + y_i^2 \leq 1$. Read the points from this file and store them into an array of structs, where the struct is of the form **`struct point{double x, y;}`**. Devise an algorithm with an average case running time of **Θ(n)** to sort the points in ascending order depending on their Euclidean distance from the origin, which is given by Pythagoras's theorem, whereby $d_i = \sqrt{x_i^2 + y_i^2}$.

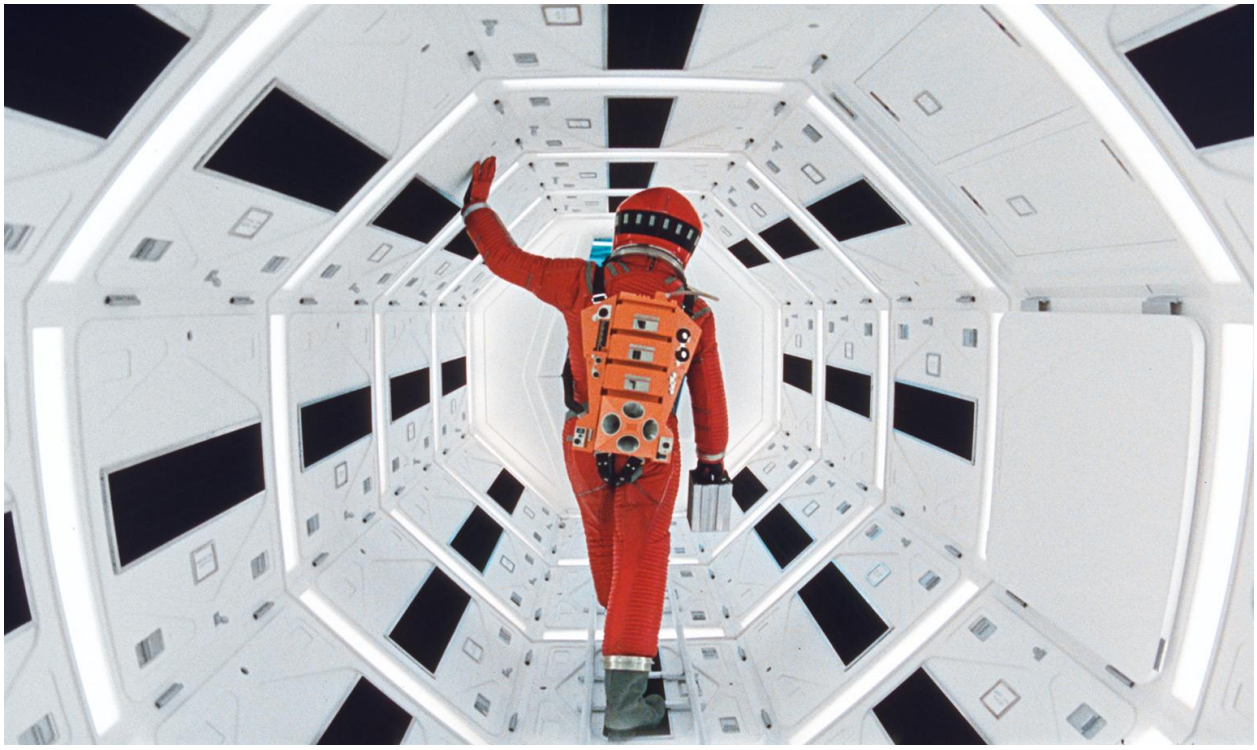
Task 7: You have been provided with the set of files having file names *datX.csv* where *X* stands for the input size (as in Lab 4). Recollect that **struct person** was defined as follows:

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
```

These files contain comma-separated entries for the details of the students. Write a program to read the data from these files and store them in a dynamically allocated array of `struct person`. In the previous labs, you had seen the performance of the other sorting algorithms like insertion sort, merge sort, and quick sort on these files. Now, modify the **counting sort**, **straight radix sort**, **exchange radix sort** and **bucket sort** algorithms discussed in this labsheet to sort arrays of `struct person` based on the **`height`** field. Plot the time taken and maximum heap space utilised and observe how they vary with the size of the input. Report the comparative performance of the space-based algorithms against a backdrop of the comparison-based ones.

Home Exercise 5: The "Discovery One" spacecraft will soon be sent to examine some anomaly in Jupiter. Two astronauts, Dr David Bowman and Dr Frank Poole, will be aboard it. The pinnacle of artificial intelligence, the **HAL 9000** supercomputer, is already going to be installed in the spacecraft. In theory, with this supercomputer installed, the astronauts will not need to worry about any computations or calculations whatsoever. Just to provide the astronauts with a different means of performing computations, they will also be provided with a handheld device called the **MINI 900**. This device is not artificially intelligent at all. It is meant to provide quick computation support to the astronauts without requiring them to take help from HAL 9000, in case the need arises. In essence, the MINI 900 is a smaller microcomputer on which the astronauts can write simple programs to solve certain problems. HAL 9000's design principle

demands that it never uses more than **constant auxiliary space** for any of its tasks. This memory constraint has enabled it to be installed into a mobile spacecraft despite being an artificially intelligent supercomputer. However, the MINI 900 has no such constraint and can in fact take up more space if required for its computations.



Suppose you are Dr Bowman aboard Discovery One, very close to Jupiter at the moment. HAL 9000 “refused” to do anything to retrieve back your colleague, Dr Poole, when he was performing a space walk. You strongly suspect that HAL 9000 may be scheming to do away with the two of you (you believe it has some ulterior motives and is not serving you as astronauts). HAL 9000 has also blocked all means to manually shut it down. Now, it is announced that some very crucial “top secret” message will in a couple of minutes be received on your radio-sensors from your research team back on earth. You and HAL 9000 are both aware that these messages come in discrete chunks and frequently in jumbled order. This message will obviously be received by HAL 9000 as well, but you do not want it to be able to decode the message before you do.

Write a program that will run on MINI 900 to sort the discrete chunks of the message into one coherent bunch and read the decoded message. This program should (obviously) run quicker than HAL 9000’s sorting program. [Make use of the fact that MINI 900 can incur extra space whereas HAL 9000 is severely limited in that regard.]

The message received on the radio-sensor will be stored in a .csv file named *"2001.csv"* (for the purposes of this question, we have already provided you with this file). This .csv file stores the words in separate lines in the format: `position,word` (the "position" is zero-indexed).