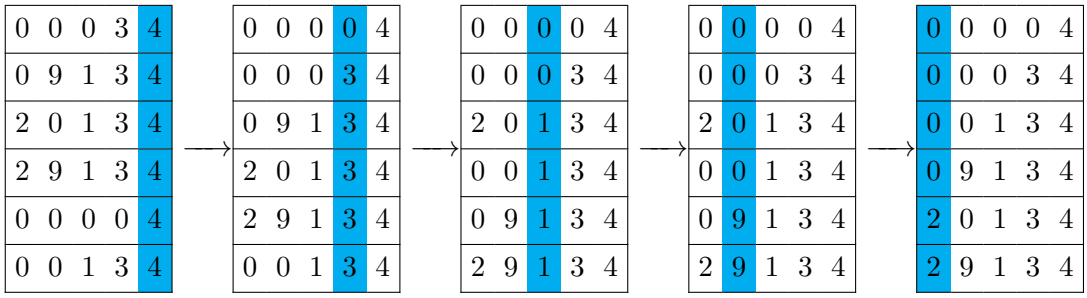
Luis Gascon, Ethan Webb, Femi Dosumu
COSC 336
April 30, 2023

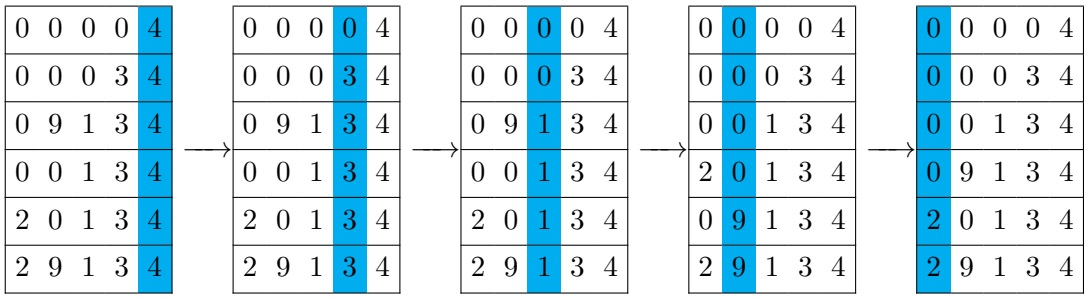
Assignment 7

Exercise 1. Show similarly to Fig 8.3 on page 198 in the textbook, how RadixSort sorts the following arrays:

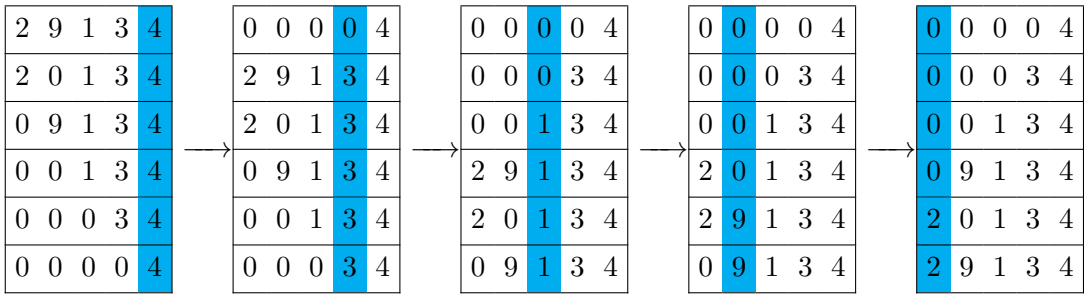
1. 34, 9134, 20134, 29134, 4, 134



2. 4, 34, 134, 9134, 20134, 29134



3. 29134, 20134, 9134, 134, 34, 4



Exercise 2. Present an $O(n)$ algorithm that sorts n positive integer numbers $a_1, a_2, \ldots, a_n$ which are known to be bounded by $n^2 - 1$ (so $0 \leq a_i \leq n^2 - 1$, for every $i = 1, \ldots, n$. Use the idea of textbook).

Note that in order to obtain $O(n)$ you have to do Radix Sort by writing the numbers in a suitable base. Recall that the runtime of Radix Sort is $O(d(n + k))$, where d is the number of digits, and k is the base, so that the number of digits in the base is also k. The idea is to represent each number in a base k chosen so that each number in $\{0, 1, \ldots, n^2 - 1\}$ requires only 2 "digits," so $d = 2$. Explain what is the base that you choose and how the digits of each number are calculated, in other words how you convert from base 10 to the base. Note that you cannot use the base 10 representation, because $n^2 - 1$ (which is the largest possible value) requires $\log_{10}(n^2 - 1)$ digits in base 10, which is obviously not constant and therefore you would not obtain an $O(n)$-time algorithm. By the same argument we see that no base k that is constant works, therefore k has to depend on n. In your explanations you need to indicate the formula that gives k as a function of n, and show that $d = 2$ "digits" are enough to represent all the numbers in the range $\{0, 1, \ldots, n^2 - 1\}$.

Illustrate your algorithm by showing on paper similar to Fig. 8.3, page 198 in the textbook (make sure you indicate clearly the columns) how the algorithm sorts the following sequence of 12 positive integers:

45, 98, 3, 82, 132, 71, 72, 143, 91, 28, 7, 45.

In this example $n = 12$, because there are 12 positive numbers in the sequence bounded by $143 = 12^2 - 1$.

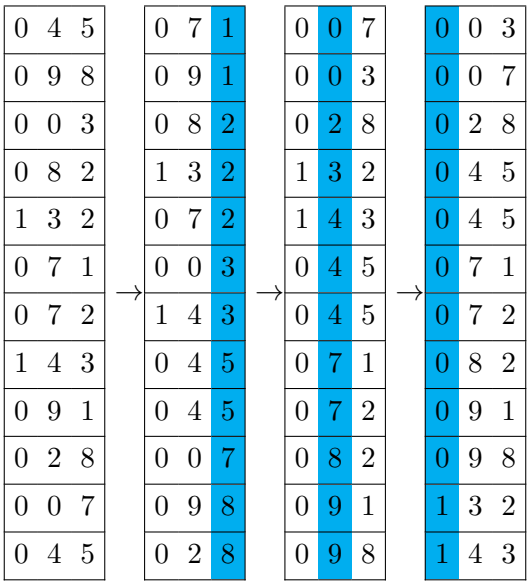
The formula that gives the appropriate base k can be obtained by taking the value of n and setting k to n. The reason why this works is because of the constraint that the values are bounded by $n^2 - 1$.

Converting the base 10 numbers to base k can be done recursively done with following algorithm in pseudo-code:

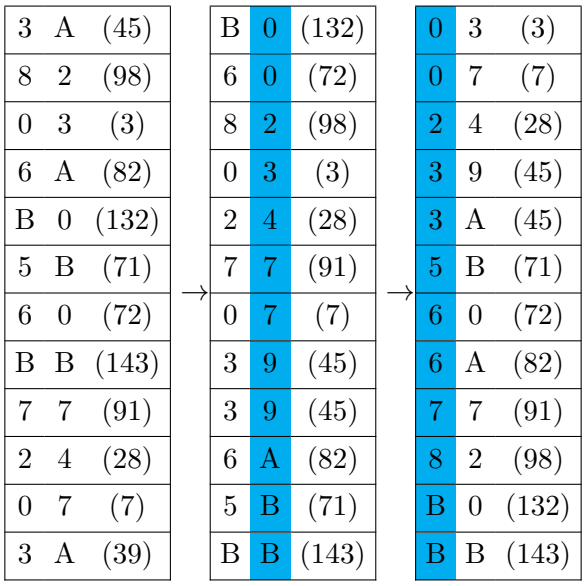
```
def decTo_nBase(el, n) -> String:
  # Assume that we have a list of characters with the appropriate
    encoding of base k.
  encodingCharArray[0 ... k]
  if x == 0:
      return ""
  remainder = x % n
  return decTo_nBase(x / n) + encodingCharArray[remainder]
```

We can then run this function for each element in the array so we'll have an array of converted values of base k.

To illustrate the correctness of our formula, we'll test it on our example set of 12 integers



If we convert our set to base 12, we should get the same result. To make it easy to follow the Radix sorting procedure, the base 10 representations are in parentheses and should be ignored by Radix sort.



Programming Task.
Our inputs consists of an integer on the first line that gives us the number of nodes that are in our graph and the nodes are labeled from $0 \ldots n$ and the next line is an array that is an adjacency matrix to represent the edges in our graph.

To start, we first have to create an object of `Adj_List_Graph`. We've created a function that reads in the integer array of the adjacency matrix. The function then returns an object of `Adj_List_Graph` that we can use as our starting graph G.

```
static Adj_List_Graph generate_graph(List<Integer> seq, int
   vertices) {
     Adj_List_Graph a = new Adj_List_Graph(vertices);
     for (int x = 0; x < seq.size(); x++)
         if (seq.get(x) == 1)
             a.addEdge(x / vertices, x % vertices);
     return a;
    }
```

The algorithm starts off by creating a graph with the same nodes as the graph G, which we called a in our function definition, so we call the constructor with the value of `n` of a.

```
Adj_List_Graph res = new Adj_List_Graph(a.n);
```

Iteration starts by iterating through each nodes of the graph.

```
for (int i = 0; i < a.n; i++) {...}
```

To replicate the argument graph G, we add the same edges that G has for the respective node by iterating the list of a.

```
  for (int i = 0; i < a.n; i++) {
    // Iterate through each list of adj
    for (int neighbor : a.adj.get(i)) {
        // Add the same edges that the parameter have to our new
   Adj_List_Graph
        res.addEdge(i, neighbor);
        ...
```

After iterating through the neighbor of the node$_i$, we iterate through the neighbors of the neighboring nodes, and add the neighbor's neighbor nodes, which would have a length of 2.

```
for (int neighborsNeighbor : a.adj.get(neighbor))
    res.addEdge(i, neighborsNeighbor);
```

The algorithm in its entirety

```
static Adj_List_Graph squaredGraph(Adj_List_Graph a) {
  Adj_List_Graph res = new Adj_List_Graph(a.n);

  // Iterate through each nodes
  for (int i = 0; i < a.n; i++) {
      // Iterate through each list of adj
    for (int neighbor : a.adj.get(i)) {
          // Add the same edges that the parameter have to our
   new Adj_List_Graph
          res.addEdge(i, neighbor);
          // Add edges to the nodes that are connected to the
   node that's connected to the
          // current node
          for (int neighborsNeighbor : a.adj.get(neighbor))
              res.addEdge(i, neighborsNeighbor);
      }
  }
    return res;
}
```

The algorithm outputs the following by calling the function `printGraph()` for each input:

```
Squared graph from input-7-1.txt

Adjacency list of vertex 0
head -> 1 -> 2

Adjacency list of vertex 1
head -> 2

Adjacency list of vertex 2
head
```

```
Squared graph from input-7-2.txt

Adjacency list of vertex 0
head -> 1 -> 2 -> 3

Adjacency list of vertex 1
head -> 2 -> 3 -> 4

Adjacency list of vertex 2
head

Adjacency list of vertex 3
head -> 4

Adjacency list of vertex 4
head
```