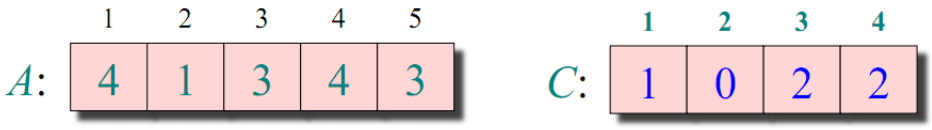
Counting Sort

Counting sort is an algorithm that takes an array A of n elements in the range $\{1, 2, ..., k\}$ and sorts the array in $O(n + k)$ time. Counting sort uses no comparisons and uses the fact that the n elements are in a limited range to beat the $O(n \log n)$ limit of comparison sorts.

Algorithm: Counting sort keeps an auxiliary array C with k elements, all initialized to 0. We make one pass through the input array A and for each element i in A that we see, we increment $C[i]$ by 1. After we iterate through the n elements of A and update C, the value at index j of C corresponds to how many times j appeared in A. This step takes $O(n)$ time to iterate through A.

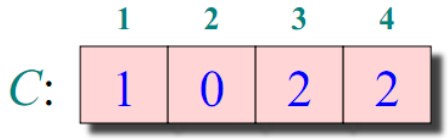


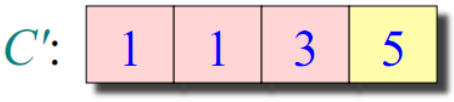
Once we have C, we can construct the sorted version of A by iterating through C and inserting each element j a total of $C[j]$ times into a new list (or A itself). Iterating through C takes $O(k)$ time.

The end result is a sorted A and in total it took $O(n + k)$ time to do so.

Note that this does not permute the elements in A into a sorted list. If A had two 3s for example, there's no distinction which 3 mapped to which 3 in the sorted result. We just counted two 3s and arbitrarily stuck two 3s in the sorted list. This is perfectly fine in many cases, but you'll see later on in radix sort why in some cases it is preferable to be able to provide a permutation that transforms A into a sorted version of itself.

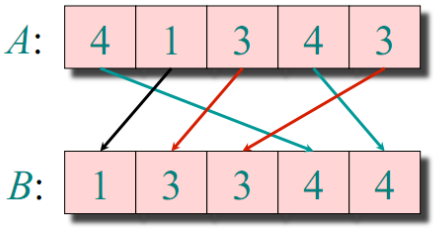
To do this, we continue from the point where C is an array where $C[j]$ refers to how many times j appears in A. We transform C to an array where $C[j]$ refers to how many elements are $\leq j$. We do this by iterating through C and adding the value at the previous index to the value at the current index, since the number of elements $\leq j$ is equal to the number of elements $\leq j - 1$ (i.e. the value at the previous index) plus the number of elements = j (i.e. the value at the current index). The final result is a matrix C where the value of $C[j]$ is the number of elements $\leq j$ in A.





Now we iterate through A backwards starting from the last element of A. For each element i we see, we check $C[i]$ to find out how many elements are there $\leq i$. From this information, we

know exactly where we can put i in the sorted array. Once we insert i into the sorted array, we decrement $C[i]$ so that if we see a duplicate element, we know that we have to insert it right before the previous i. Once we finish iterating through A, we will get a sorted list as before. This time, we provided a mapping from each element A to the sorted list. Note that since we iterated through A backwards and decrement $C[i]$ every time we see i. we preserve the order of duplicates in A. That is, if there are two 3s in A, we map the first 3 to an index before the second 3. A sort that has this property is called a **stable sort**. We will need the stableness of counting sort when we use radix sort.

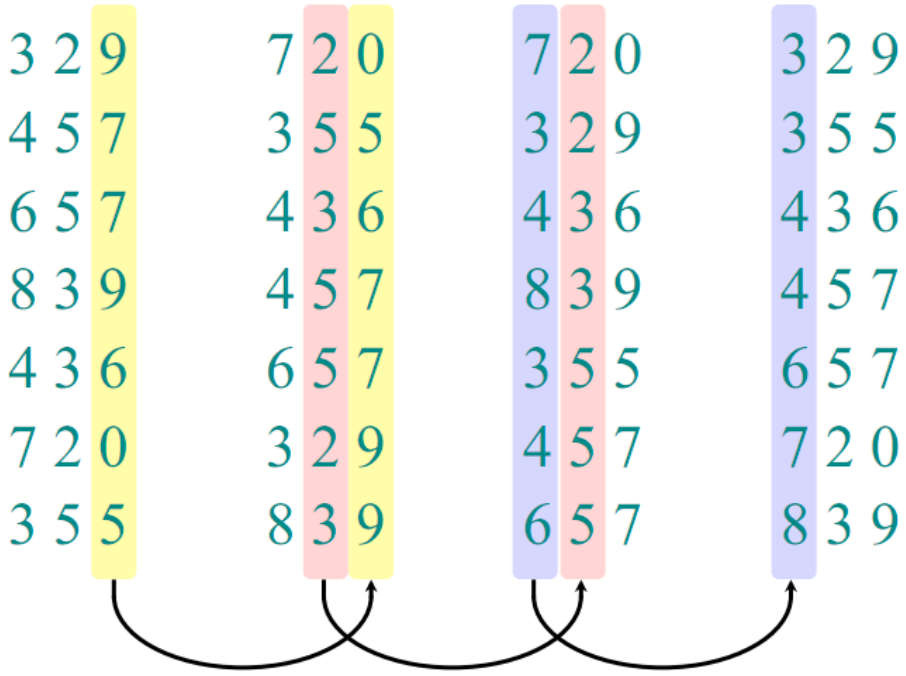


Iterating through C to change $C[j]$ from being the number of times j is found in A to being the number of times an element $\leq j$ is found in A takes $O(k)$ time. Iterating through A to map the elements of A to the sorted list takes $O(n)$ time. Since filling up C to begin with also took $O(n)$ time, the total runtime of this stable version of counting sort is $O(n+k+n) = O(2n+k) = O(n+k)$.

Radix Sort

The downfall of counting sort is that it may not be too practical if the range of elements is too large. For example, if the range of the n elements we need to sort was from 1 to n^3, then simply creating the auxiliary array C will take $O(n^3)$ time and counting sort will asymptotically do worse than insertion sort. This also takes $O(n^3)$ space which is significant larger than any of space used by any other sorting algorithm we've learned so far.

Radix sort helps solve this problem by sorting the elements digit by digit. The idea is that we can sort integers by first sorting them by their least significant digit (i.e. the ones digit), then sorting the result of that sort by their next significant digit (i.e. the tens digit), and so on until we sort the integers by their most significant digit.



How do we sort these elements by digit? Counting sort! Note that the k factor in counting sorting by digit is restricted to the range of each digit instead of the range of the elements. We have to use the stable variant of counting sort in radix sort. If we used the first version of counting sort, we wouldn't have a mapping from element to element. In the example above, when we sort the ones digit, we would be able to sort them in order but we would have no indication of whether the first 7 corresponds to 457 or 657. By using the stable variant, we get a mapping from element to element and can map the 457 to the first 7 since 457 appeared earlier in the list than 657. This stable property also ensures that as we sort more and more significant digits, duplicate digits stay in the right order according to the less significant digits, thus preserving the correctness of radix sort.

If there are n integers to sort in radix sort and the integers are represented in base k (i.e. there are k possible values for each digit) and the maximum length of the integers is d, then radix sort will execute a $O(n+k)$ counting sort for each of the d digits, resulting in a $O(d(n+k))$ runtime for radix sort.

Note that what base we choose to represent the integers in affects the value of k and d. Large bases have the advantage of shorter integers digit-wise, but the counting sort on each digit will take longer and vice versa for small bases.

Knowing n, the ideal base for radix sort is $k = n$. Say that the integers range from 0 to $u - 1$. The number of digits in the integers is at most $\log_k u$ for base k. We can substitute this into the runtime for radix sort to get $O((n+k)\log_k u)$. Using some fancy math, we can find that the best k to choose to minimize this runtime is $k = n$. In this case, our runtime for radix sort is $O(n \log_n u)$. When $u = n^{O(1)}$, the $\log_n u$ term becomes a constant and our runtime for radix sort turns out to be $O(n)$, giving us a linear time sorting algorithm if the range of the integers we're sorting is polynomial in the number of integers we're sorting.

Graph Basics

A graph contains a set of vertices V and a set of edges E. Edges are defined to be a mapping from a vertex to a vertex, often denoted as $\{a, b\}$ where a, b are vertices being connected by an edge.

In a directed graph, the order of the vertices in an edge matters. $\{a, b\}$ refers to an edge that starts from a and ends at b and is distinct from $\{b, a\}$, which is an edge that starts from b and ends at a. In an undirected graph, the order does not matter, and both $\{a, b\}$ and $\{b, a\}$ refer to the same thing: an edge that connects a and b to each other.

A graph G is defined solely by its vertices V and its edges E.

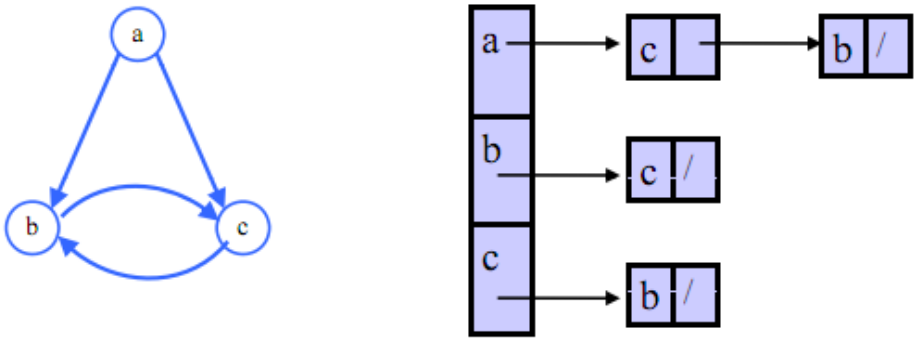
Graph Representation

As humans, we like to visualize graphs in diagrams where we can see the vertices being connected to each other by the edges in the graph. However, when we program with computers, it makes little sense to try to make the computer interpret and manipulate a diagram of the graph. Since the graph is solely defined by its set of vertices and its set of edges, we only need those sets to represent a graph.

We have four main options in representing graphs.

Adjacency List

We keep an array of $|V|$ linked lists where each list corresponds to the set of neighboring vertices for a particular vertex. The set of edges are extracted from the neighbor relationships between vertices.



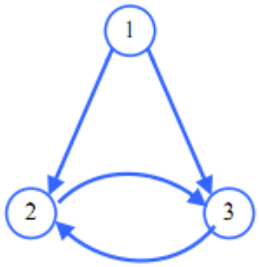
The total number of elements in the $|V|$ linked lists is $\Theta|E|$ since each edge is represented as an element once (in the case of directed graphs) or twice (in the case of undirected graphs). We also need to allocate space for $\Theta|V|$ linked lists. Representing each node in the linked lists takes some space, say $\Theta(w)$ bits. The size of the adjacency list representation would thus be $\Theta((|E| + |V|)w)$.

Incidence List

We keep an array of $|V|$ linked lists (or hash tables) where each list (or hash table) corresponds to the set of edges that originate from a particular vertex. Like an adjacency list, but the actual edges are stored instead of the neighboring vertices. This also has $\Theta((|E| + |V|)w)$ space.

Adjacency Matrix

We keep a $|V| \times |V|$ matrix where an element in row i and column j is 1 if there is an edge from the ith vertex to the jth vertex and 0 otherwise.



1	2	3	
0	1	1	1
0	0	1	2
0	1	0	3

Adjacency matrices has $|V|^2$ entries of just 1 bit each (since all entries are 1 or 0) so the space an adjacency matrix takes is $\Theta(|V|^2)$. This is advantageous for very dense graphs since the size of the graph representation does not depend on the number of edges, only on the number of vertices. Adjacency and incidence lists on the other hand are better for sparse graphs since their space complexity does not have a quadratic factor in the number of vertices.

Implicit Representation

Instead of storing a graph's vertices and edges, implement a function Adj(u) that calculates the neighboring vertices of some vertex u on the spot. For example, in the graph representation of a Rubik's Cube, each vertex is a cube state and the neighbors can be calculated by applying all the legal moves on that cube state. Storing the entire graph may be very costly in this case, but if we have a general method of calculating the neighbors of any given vertex, we can use this function Adj(u) instead to represent the entire graph. Requires no space, but needs a generalizable way of finding neighbors for any given vertex.

Time Comparisons

We can compare the runtime of common graph operations under each of these representations. Implicit representation was left out because most of these operations either don't apply to implicit representation (add edge) or depend on what the runtime of the Adj(u) function is.

Add edge: Adding an edge to the adjacency list, adjacency matrix, and incidence list are all $O(1)$ time. The matrix just has to update the corresponding bit in the matrix to add an edge and the lists just need to insert the neighbor/edge into the right linked list.

Check/Remove edge: To check to see if there exists an edge from a to b, an adjacency matrix just has to check the corresponding bit in the matrix and see if it's a 1 or 0, which takes O(1). The lists however potentially need to iterate through all the edges or neighboring vertices of a to find if b is indeed a neighbor.

Visit all neighbors: To visit all of the neighbors of a, the lists again iterates through all the edges or neighboring vertices of the linked list associated with a. However, in the adjacency matrix, we need to iterate through all of the $|V|$ entries in a's row or column, keeping track of which vertices has a 1 bit. This takes $O(|V|)$ time.